Introduction to Concurrent Programming in C

Filip Strömbäck
Linköping University

2025-11-12
(commit: `1f6ca8b`)

Preface

The goal of this book is to provide a detailed introduction to concurrent programming. As such, the goal is to cover the fundamentals required to write and maintain programs that are correct according to the language specification. That is, the main focus is on *correctness* rather than on *maximizing performance*. That being said, the book will of course cover obvious performance issues with certain approaches.[1] However, the book will *not* cover advanced techniques for maximizing performance by, for example, using lock-free data structures. Rather, the book focuses on the basics required to understand the problems that concurrent programming introduces, and the tools required to address those issues. This is sufficient for at least 90% of all cases, but it also provides a solid starting point for studying the advanced techniques.

This book will try to take a hands-on approach to a subject that is otherwise quite theoretical in nature. It does so by illustrating the ideas with examples and exercises. In particular, the book contains many boxes labeled *Practice* that contains small exercises related to the text so that you can experiment with the material while you read the chapter.

The book uses the C programming language, but the concepts covered are general enough to apply to most imperative and object-oriented languages that rely on the *shared-memory model* of concurrency, such as C++, Java, Rust, Python, etc. Compared to C, these languages provide more powerful facilities for building abstractions. This means that they make it possible to use the concepts from this book in a more ergonomic way. However, these abstractions often hide details that are important to consider when working with concurrent programs. As such, the simpler abstraction facilities available in C is actually beneficial for novices since these important details become more explicit.

The book assumes familiarity with programming in some C-like language. It is, however, not necessary to be deeply familiar with C in order to be able to follow along.

[1] After all, a typical goal with concurrent programming is to improve performance.

Contents

Chapter 1

Tools and Software

This book uses two main tools for examples and exercises: a C compiler and the visualization tool Progvis. The C compiler along with the threading library will be used to compile and run small programs to observe their behavior in a "real" setting. Progvis is a complement to the C compiler. It is also able to compile and run simple C programs. The difference is that Progvis visualizes how the program is being executed, and even lets you influence the execution to find concurrency issues.

1.1 Examples and Exercises from the Book

All the examples in the book are also available as source code in an archive available at `https://storm-lang.org/progvis-book/`. You simply need to download and extract the contents of the archive somewhere on your system to be able to follow along with the examples in the book.

The archive contains two directories for each chapter in the book. The directories are named `xx-practice` and `xx-exercises`, where `xx` is the chapter number. The first one contains the code discussed in the chapter. As such, it is where you will find the source code for the problems in the *Practice* boxes in the text. The second one (`xx-exercises`) contains the code for the exercises at the end of the chapter. There are also example solutions for some of the problems. They are located separately in the directory `solutions/xx`.

1.2 C Compiler

The code in this book has been developed on a GNU/Linux system using GCC. While the code should work on other systems as well, you might encounter cases where "incorrect" examples behave differently on your system.[1] While this usually only serves to illustrate the point of the book better, it might lead to unnecessary confusion. As such, we advise you to use a GNU/Linux system if possible. If you are running Windows, you can use Windows Subsystem for Linux (WSL) to conveniently run Linux tooling from your Windows installation. On MacOS, the compiler from XCode should be similar enough, but a

[1] This is since they contain *undefined behavior*, so anything may happen.

Linux Virtual Machine is required to run Progvis anyway, so you might as well use that for the C compiler as well.

1.2.1 Installation

The code in this book only depends on the system's C libraries,[2] and Make. In many cases they are installed by default. If not, you can install them using commands similar to the ones below:

Debian, Ubuntu:	`sudo apt install gcc make`
Arch Linux:	`sudo pacman -S gcc make`

Apart from a C compiler, you also need the threading library provided with this book. This library does not need to be installed since it is small enough to be compiled alongside each example. This is done automatically as long as you are working with any of the examples provided with the book. See Chapter A for documentation for the library.

If you wish to use the library for other programs, you need to create a make-file that includes the library. How this is done is shown below.

1.2.2 Usage

The tooling provided with this book is designed for small programs that consist of a single source file. To compile a program, open a terminal and change to the directory where the file you wish to compile is located. Then simply type:

```
make <name>
```

Where `<name>` is the name of the file you wish to compile, without the `.c` extension. So for example, to compile the file `test.c`, you type `make test` and hit enter. This process creates the file `<name>` (e.g. `make test` produces the file `test`). You can then run the program by typing `./<name>` (e.g. `./test`).

Often, you end up wanting to compile a program and then directly execute it. To make this more convenient, you can chain the two commands with the `&&` operator, like:

```
make <name> && ./<name>
```

The makefile also provides the option to remove all compiled files by running the command:

```
make clean
```

1.2.3 Optimizations

By default, the makefile does not enable compiler optimizations. This is often exactly what we want when working with the examples in this book, since it turns out that the optimizations in the compiler often breaks incorrectly

[2] More specifically, pthreads for Linux

synchronized concurrent programs in interesting ways. It is possible to enable optimizations by adding `OPT=1` to the `make` command:

```
make OPT=1 <name>
```

This also forces the program to be re-compiled, so it is not necessary to run `make clean` before enabling optimizations. This is not true in the other direction, so if you have compiled a file with optimizations (e.g. `make OPT=1 test`) and then wish to compile it without optimizations (e.g. `make test`), make will simply say "Nothing to be done..." and not re-compile the program. In this case, you need to either delete the compiled file first (`rm test`) or run `make clean` to remove all compiled programs first.

The text in the book will explicitly mention when it is useful to enable optimizations.

1.2.4 Creating Custom Examples

If you wish to create your own examples to experiment with the concepts in this book, you have two options:

- You can add your own `.c` file to any of the directories that already contain examples from the book. This makes it possible to compile your code the same way as you compile and run the examples.
- If you want a separate directory for your own experiments, you also need to create a file named `Makefile` in that directory (note the capital M) with the following contents:

  ```
  BOOK_LIB_PATH := <path_to_lib>
  include $(BOOK_LIB_PATH)/Make.template
  ```

  Replace the text `<path_to_lib>` with the path to the `lib` directory you extracted from the archive that contained the examples to this book. After this, you should be able to compile and run your code as outlined above.

1.3 Progvis

It is also possible to run the examples in this book in Progvis. This lets you see their behavior in more detail, and even affect the execution of the program. The programs in this book require Progvis version 0.6.7 or later.

1.3.1 Installation

How to install and run Progvis depends on your operating system as described below:

Debian, Ubuntu Progvis is available as a package through the system's package manager. You can install it by running `sudo apt install progvis` in a terminal. You can then start Progvis from the system menu (look under the *Education* category), or by typing `progvis` in a terminal.

Make sure that the version of Progvis installed is 0.6.7 or later (the release cycles of Debian and Ubuntu are quite long). You can check the version by opening Progvis and selecting *Help* ⇒ *About...* in the menu. If Progvis is too old, you can follow the instructions for Linux below instead.[3]

Linux For other distributions of Linux, first try to use the Storm files available on `https://storm-lang.org/`. They are known to work on Debian, Ubuntu, and Arch Linux at least. Download the archive for your system from the *Downloads* page, extract the files to some directory using `tar xf storm*.tar.gz` or any graphical tool available in your system.[4] The archive contains a file named `progvis.sh`. Run that file in a terminal or by clicking it in your graphical environment.

If Progvis does not start, you likely need to compile it from source. This is described on `https://storm-lang.org/` under *Getting Started* ⇒ *Developing in Storm* ⇒ *Compile from Source*. When you are done, you can launch Progvis using the command `mymake Main -- -f progvis.main` in the directory where you compiled Storm.

Windows Even if you are using WSL to compile C code, it is recommended to use the native Windows version of Progvis. Progvis includes its own C-compiler, so you don't need to install any other C compilers to use it.

Download Storm from the Downloads page of `https://storm-lang.org/`. Extract the zip-file to some location. This can be done by opening the zip file in Explorer and dragging the files to some other directory. After this is done, you can launch Progvis by clicking the file `Progvis.bat`.

Depending on how you downloaded and extracted the files, you may see a message with a title like *Windows has protected your computer* the first time you launch Progvis. If this happens, click the *More information* link. This shows a button labeled *Run anyway*. Click that button to launch Progvis. This only happens the first time you launch Progvis.

[3] Note: version 0.6.3 is enough for almost all examples. You can update when you encounter compilation errors.

[4] Chrome sometimes uncompresses the downloaded file when it is downloaded. If `tar` complains, try to rename the file from `.tar.gz` to just `.tar`.

1.3.2 Usage

Once you have installed and started Progvis as described above, you will see the main window of Progvis. The first step to use Progvis is to open a program. To do this, select *File* ⇒ *Open file...*, and select a program. Note that Progvis wants you to select the source code of the program you wish to run. You *don't* need to compile the program beforehand. Progvis takes care of the entire compilation process.

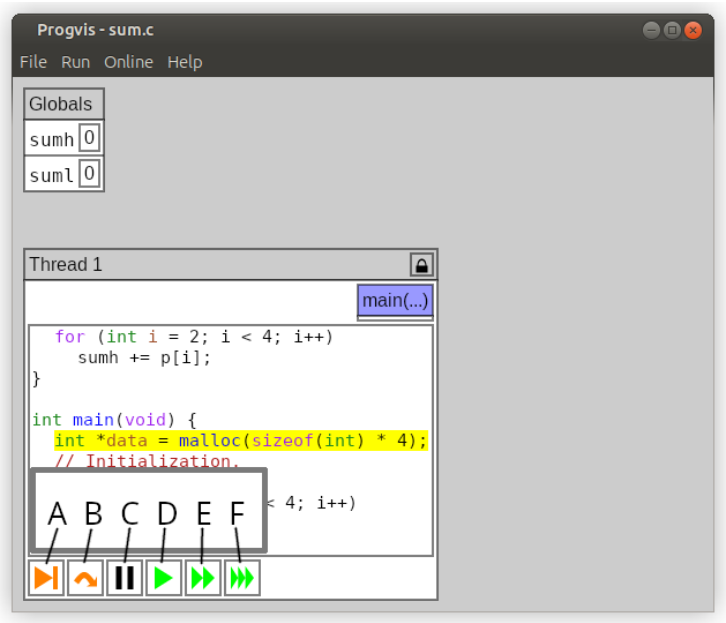


Figure 1.1: Progvis displaying the starting point of a program. In this case, the program `sum.c` from the built-in examples.

When you have selected a program, Progvis will automatically compile the program for you. If the compilation succeeded, Progvis will display the initial state of the program in its main window, similarly to what is shown in Fig. 1.1. The box labeled *Thread 1* shows the next line that the main thread of the program will execute next highlighted in yellow. Any local variables will also be shown in the blue box labeled `main(...)`. Global variables are shown in the box labeled *Globals*. The remainder of the gray area will be used for heap-allocated data as the program runs. Program execution can be controlled by pressing the buttons labeled A–F in *Thread 1*. They do the following:

A: Let the current thread run to the next statement.

B: Let the current thread run to the next *barrier*. These steps are typically much longer than stepping a single statement. The exact meaning will be covered later in the book.

C: Pause automatic stepping.

D: Start stepping automatically. Equivalent to clicking button A once every second.

E: Start stepping automatically. Equivalent to clicking button A twice every second.

F: Start stepping automatically. Equivalent to clicking button A four times every second.

All boxes in the main window of Progvis can be moved by dragging the title of the box. It is also possible to move everything by dragging the background. A few boxes have a small padlock in the upper-right corner. They are locked to the edges of the main window by default. If you drag them around, they will become detached and behave like the other boxes. You can lock them to the edge again by clicking the padlock.

If any errors were encountered during compilation, Progvis will display an dialog that highlights the error. Progvis still remembers which file you were attempting to open, so after you have fixed the problem (using your preferred text editor) you can simply select *File* ⇒ *Reload program* to open the same program again.

This ability is also useful when experimenting. If you made changes to your program and wish to see the new behavior of the program, you can simply select *File* ⇒ *Reload program* instead of locating the same file again.

As an additional convenience, Progvis is also able to open the current program in a text editor using *File* ⇒ *Open in editor*. This command attempts to open the default editor in your system. If this is not the editor you prefer, you can override the default in *File* ⇒ *Settings*. Note that on Linux, you can simply type the name of a command in the box *Editor to use to open code for Progvis*.

Finally, it is worth noting that Progvis supports a subset of the full C language. This is mainly to make C a bit more type-safe so that Progvis is able to visualize and analyze the program properly. The limitations are described in further detail in Chapter B.

Chapter 2

Programming in C

This chapter provides a brief introduction to the C language as it is used in the remainder of this book. The reader is assumed to already be familiar with some other imperative programming language (e.g. C++). Readers who are already familiar with C can largely skip this chapter and use as a reference while reading the remainder of the book. Section 2.8 might, however, still be useful to read as it outlines the conventions used to create abstract data types in the book.

The examples covered in this chapter can be visualized in Progvis to illustrate their behavior in more detail if desired. To do this, simply select *File* ⇒ *Open...* in Progvis and select the example in the `02-c-programming` directory. Then press the leftmost button in the box labeled *Thread 1* to single-step the program and see the behavior of the constructs.

2.1 Functions

C requires that all program code appears inside a function. Functions are declared as below:

```
<result> <name>(<parameters>) {
    <code>
}
```

Where `<name>` is the name of the function, `<result>` is the type of the value returned by the function, and `<parameters>` is a comma-separated list of parameters to the function. For functions that do not return anything, `<result>` should be `void`. Similarly, but perhaps surprisingly, `<parameters>` should be `void` if the function does not accept any parameters.[1] As such, a function `f` that accepts zero parameters and does not return any result looks like below:

```
void f(void) {
    /* ... */
}
```

[1] Empty parentheses (i.e. `f()`) actually means that the parameter list is *unknown* and will be specified later. This will change in an upcoming C standard, however.

The parameter list (`<parameters>`) is a comma-separated list of parameters. Each element has the form `<type> <name>` and defines a single parameter. For example, a function `f` that returns an integer and accepts two integer parameters looks as follows:

```
int f(int a, int b) {
    /* ... */
}
```

The function `main` is special. It dictates the start of program execution and will be automatically called by the system when the program is started. The `main` function is expected to return an integer, which will be returned to the system so that whoever started your program can assess whether your program succeeded in doing its job or not. The `main` function may also accept command-line parameters from the system.

In the context of this book (for compatibility with Progvis), the main function will always look like below:

```
int main(void) {
    return 0; // For success.
}
```

Note that the `return` statement is required by Progvis, even though it is optional according to the C standard.

Parameters to functions are always passed *by value*. This means that the values passed to a function through parameters are always *copied* into the function. Any changes the called function (the *callee*) makes to its parameters are thereby not visible in the caller. This is illustrated by the program `by-value.c`:

```
int f(int x) {
    x = 20;
    return 12;
}

int main(void) {
    int a = 10;
    int b = f(a);
    printf("a=%d\n", a);
    printf("b=%d\n", b);
    return 0;
}
```

The program will print `a=10` and `b=12`. Even though `a` is passed as the first parameter to `f`, and `f` sets `x` to 20, the modification is not reflected in `a` since `f` creates its own copy of the variable. What happens can be illustrated by the program below, where `f` has been inserted inside `main`, and the copy made explicit (also available in `by-value-expanded.c`):

```
int main(void) {
    int a = 10;
    int b;
    {
        int x = a;
        x = 20;
        b = 12;
    }
    printf("a=%d\n", a);
    printf("b=%d\n", b);
    return 0;
}
```

Note that C does not support references (i.e. `&` in C++). It is, however, possible to use pointers to explicitly let a called function modify variables in the calling function.

2.2 Variables

As we saw above, variables are declared as `<type> <name>`, just like parameters. For example, an integer variable named `x` is declared as follows:

```
int x;
```

Declarations like the above can appear either on their own, at global scope, inside functions, or inside data structures. A variable that at global scope defines a global variable. A variable inside a function defines a variable that is local to each function call. As we saw above, it is possible to add additional blocks inside functions to limit the scope of local variables as desired. Variables defined inside a struct become members of the struct.

By default, global variables are initialized to 0, while the contents of local variables are undefined.[2] To avoid difficulties in debugging programs, it is therefore a good habit to *always initialize variables explicitly*. In C, initialization looks like an assignment to the newly declared variable:

```
int x = 0;
```

Of course we can initialize local variables using an arbitrary expression. We are, however, more limited when we try to initialize global variables.

Note that C allows multiple variables to be declared at the same time (e.g. `int x, y;`). This notation is not supported by Progvis, and will not be used in this book.

2.3 Output

Formatted output in C is most commonly done through the `printf` function (from the header `<stdio.h>`). The first parameter to `printf` is a *format string* that describes the text that should be printed. We can use `printf` to print a simple message as follows:

[2] Progvis shows them as 0 in an effort to be deterministic.

```
printf("message\n");
```

Note that the string ends with a newline character (i.e. `\n`). The `printf` function does not automatically add a newline character. As such, if we omit it from the format string, any subsequent calls to `printf` would appear on the same line. Additionally, the C standard library typically buffers entire lines of output. This means that without the explicit newline character, the message may not be visible until much later, when another call to `printf` outputs a newline character.

As the name *format string* implies, we can not only use it to output plain text. We can also include instructions on how to format the contents of variables. For `printf` this is done by adding a *format specifier* to the format string. A format specifier consists of a percent sign (`%`) followed by a description of what to output. This causes `printf` to replace the format specifier with a formatted version of the next parameter to `printf`.

In this book, we will use the following format specifiers:

- `%s` outputs a string (i.e. type `const char *`).
- `%c` outputs a character (i.e. type `char`).
- `%d` outputs an integer (i.e. type `int`).
- `%%` outputs a single percent sign.

Strings and integers can optionally be padded to a specified width by adding a number between the `%` and the `s` or `d`. For example, `%10d` will output an integer, but padded with spaces until the resulting number is at least 10 characters wide. If the number is positive (e.g. `%10d`) the spaces are added to the left, and if the number is negative (e.g. `%-10d`) they are added to the right. Integers can optionally be padded with zeros if the number starts with a zero (e.g. `%010d`).

Usage of `printf` is illustrated by the program `printf.c`, which is also depicted below:

```
int main(void) {
    int a = 1;
    int b = 20;
    const char *s = "test";

    printf("%s\n", s);          // Outputs: "test\n"
    printf("%s%d\n", s, a);     // Outputs: "test1\n"
    printf("%s %d\n", s, b);    // Outputs: "test 20\n"
    printf("%s: %3d\n", s, a);  // Outputs: "test:   1\n"
    printf("%s: %3d\n", s, b);  // Outputs: "test:  20\n"
    return 0;
}
```

Note that format specifiers do not have to be delimited by spaces. If there are spaces or other characters between format specifiers they will appear in the output, just as when passing normal strings to `printf`. Also note that it is possible for a single format string to contain multiple format specifiers. In this case, the first format specifier uses the first parameter after the format string, the second uses the second, and so on.

2.4 Types

C provides a number of built-in integer types. This book uses the following types:

- `int`: a signed integer. In general it is only safe to assume that an integer is at least 16 bits wide, but on most modern desktop systems it is safe to assume that an `int` is at least 32 bits wide. This is the case in Progvis, and what is assumed in this book.
- `char`: a single character. It is usually safe to assume that a `char` is 8 bits wide, since the POSIX standard (which e.g. Linux follows) requires 8 bits in a `char`.
- `bool`: a boolean truth value (i.e. `true` or `false`). Note that `bool` is not a built-in type in C. It is implemented in the header `stdbool.h`, so it is necessary to `#include <stdbool.h>` to use booleans. The headers in the threading library for this book does this for you, so you usually do not have to worry about including it separately.

All types can be prefixed with `const` to refer to a constant value of the type. Most notably is the absence of a dedicated string type. We will see later in this chapter how strings are represented in C.

It is also possible to define custom data structures using the `struct` keyword. We will therefore refer to them as *structs*. A struct is a collection of variables that are stored one after another in memory. The variables in a struct are furthermore handled as a unit, so if we create an instance of a struct, we will create an instance of each of the variables that are a member of the struct. Just like with other variables, it is possible to copy structs with the assignment operator (`=`).

The program `struct.c` below illustrates how structs can be defined and used in C:

```
struct data {
    int x;
    int y;
};

int main(void) {
    struct data d;
    d.x = 10;
    d.y = 20;
    struct data e = d;
    printf("e.x=%d, e.y=%d\n", e.x, e.y);
    return 0;
}
```

Notice that we need to write `struct data` to name the struct. Simply writing `data` (like in C++) does not work, since the name of structs are in a different namespace by default.[3]

[3]It is possible to get around this using `typedef`, but this is not supported by Progvis and therefore not used in this book.

It is also worth pointing out that a struct may *only* contain variables. In contrast to a class in an object oriented languages, structs may *not* contain functions. We will see in Section 2.8 how we can use structs and functions to create abstract data types that are similar to "classes".

2.5 Pointers

Pointers are a central part of C. A pointer is essentially an integer variable that we know contains the *address* of some other data in memory. Because they refer to some other data, it is usually convenient to represent them as an arrow that points to some other data. The idea that a pointer is just a numeric value is, however, important to keep in mind. It helps reasoning about the behavior of pointers, both regarding assignments of pointers and regarding the semantics of pointer variables in a concurrent program.

C provides special *pointer types* to denote which variables are pointers, and what type of data they refer to. This makes it more difficult to accidentally treat a pointer as an integer, or to accidentally interpret the data pointed to as an integer when it is actually something else. A pointer is expressed using an asterisk (`*`) after the name of a type (e.g. `int *` is a pointer to an `int`).

In addition to pointer types, C provides two operators that create pointer values and traverse pointers, respectively. These are illustrated in the program `pointers-1.c` below:

```
int main(void) {
    int a = 1;
    int *p = &a;
    *p = *p + 1;
    printf("a=%d\n", a);
    return 0;
}
```

The program first creates the integer variable `a` and initializes it to 1. The next line then uses the *address-of operator*, `&`, to compute the address of the variable `a` (i.e. `&a`). One way to think of this operation is that the `&` operator creates a pointer value that refers to `a`. The type of this pointer value is the type of `a` (i.e. `int` in this case) with an additional asterisk added at the end. In this case, this produces the type `int *` which matches with the declared type of the variable `p`.

At this point it is useful to see how Progvis represents pointers. Load the program `pointers-1.c` in Progvis by selecting *File* ⇒ *Open...* Then click the leftmost button in the *Thread 1* box a few times, and you will see the representation in Fig. 2.1. In particular, notice the smaller box labeled *main(...)* in the top-left corner. It contains a list of the local variables inside the `main` function. The first line contains the variable `a` and shows that it contains the value 1. The second line shows the variable `p`. Its content is the address where the variable `a` is located. To make it easy to see what happens, Progvis represents this as an arrow that points to the data in the variable `a`.

The next line in the program (the one highlighted in yellow in Fig. 2.1) contains the next pointer-specific operator, the asterisk (`*`). This operator is used to *dereference* a pointer, meaning that we examine the address in the pointer, go to that location in memory and read whatever value is there. In

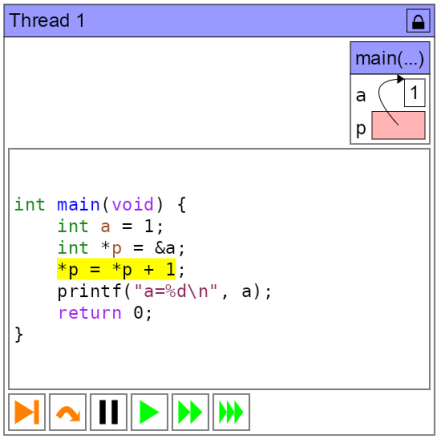


Figure 2.1: Progvis' representation of pointers in the program `pointers-1.c`.

the case of `*p` in Fig. 2.1 it means that we look at the arrow stored in `p`, and instead of operating on `p` itself, we operate on the data that the arrow points to instead. The type of the expression `*p` is the type of `p` (i.e. `int *`) but with the rightmost asterisk removed, which in this case is just an `int`.

> **Practice:** To see the difference between `p` and `*p`, change the line `*p = *p + 1` to `p = p + 1` and see what happens in Progvis. Remember that `p` refers to the pointer itself (i.e. the integer address that a pointer is), while `*p` dereferences the pointer and operates on that data instead.

At this point it is useful to spend a moment to compare the syntax used to denote pointer types with the operators that operate on pointers. We saw that the address-of operator (`&`) essentially creates pointers. Therefore it might be surprising that the same operator is not used to denote a pointer type. The reason for this is that types in C are designed to reflect how the variable is *used*. That is, one way to think of the declaration `int *p` is: "We have a variable `p`, which has a type so that the expression `*p` is an integer." This is the reason why it might at first feel like the roles of `*` and `&` are swapped in type declarations compared to their usage in code. This is also one reason why this book uses the notation `int *x` rather than `int* x`.[4]

Since parameters in C are always passed by-value, pointers are often used to simulate by-reference semantics. This both allows passing large structs without copying them, but also allows functions to modify data provided by the caller easily. This is illustrated by the program `pointers-2.c` which is shown below. The program contains two functions, `main` and `f`. The `main` function creates a variable `a` that it wishes to let `f` modify. Therefore, it passes the address of `a` to `f` by using the address-of operator (`&`). The function `f` then receives the address as an `int *` parameter, and modifies the variable `a` from `main` by using the dereference operator (`*`). As such, the program prints `a=10`.

[4]The idea that variable declarations reflect the usage of the variable is also apparent if multiple variables are defined at once. For example, `int *x, y;` defines two variables. `x` is of type `int *` while `y` is of type `int`! Please don't do this.

```c
void f(int *x) {
    *x = 10;
}

int main(void) {
    int a = 0;
    f(&a);
    printf("a=%d\n", a);
    return 0;
}
```

Again, it is useful to see how Progvis represents this situation. As such, open the program in Progvis and single-step the program until you reach the state in Fig. 2.2. The representation is very similar to Fig. 2.1. The difference is that *two* boxes are visible in the top-right corner. The topmost contains the local variables in `f` and the one that is partially covered contains the local variables in `main`. With this in mind, we can easily see that the pointer variable `x` refers to a variable in `main`. We do not see the name of the variable from this particular figure,[5] but we can easily deduce that it has to be the variable `a`.

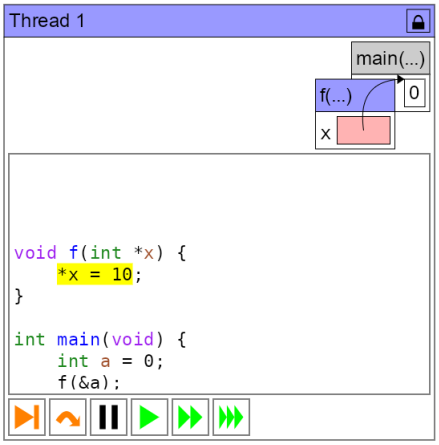


Figure 2.2: Progvis' representation of pointers in the program `pointers-2.c`.

Practice: Use Progvis to see what happens if you change `f` to accept a `int x` instead of `int *x`. What other parts of the program do you need to change, and what will the program print when you run it?

[5] This is actually intentional. The name `a` is not accessible from `f`, so the only way to access the variable is if we have a pointer to it!

One problem with the dereference operator (`*`) is that it is applied after the member access operator (`.`) operator. This is sub-optimal when working with pointers to structs, as illustrated by the program `pointers-3.c`:

```
struct data {
    int a;
    int b;
};

int main(void) {
    struct data d;
    d.a = 1;
    d.b = 2;

    struct data *p = &d;
    printf("a: %d\n", (*p).a);
    printf("b: %d\n", p->b);

    return 0;
}
```

Note that the program defines the pointer `p` that refers to `struct data`. If we wish to access the variable `a` inside `struct data`, we need to write `(*p).a`. We can not just write `p.a` since the member access operator (`.`) expects its left hand side to be a struct, not a pointer to a struct. This is why we need the dereference operator (`*`). Adding a dereference operator like `*p.a` will not solve the problem since it means `*(p.a)`. This is why we need an explicit parenthesis as in the first `printf` statement: `(*p).a`. The dereference operator first "converts" the pointer into a value, so that the member access operator receives a struct instead of a pointer as its left hand side.

This is, however, not very convenient. For this reason C provides the `->` operator as a convenience. The expression `x->y` is equivalent to `(*x).y`. As such, instead of writing `(*p).a` we can simply write `p->a`. This is what `pointers-3.c` does to access the variable `b` in the second `printf` statement.

2.6 Arrays

An array in C is a collection of elements of the same type that are stored after each other. Note that unlike some other languages (e.g. Java or `vectors` in C++) C stores *no* additional information about the array. In particular, this means that it is up to the programmer to keep track about the size of the array and to make sure to not access elements out of bounds. While this is sometimes problematic, it allows a convenient analogy between arrays and pointers. We use the program `arrays.c` to illustrate this.

```
int main(void) {
    int arr[3] = { 1, 2, 3 };
```

The listing above shows the first two lines of `arrays.c`. The program starts by defining a variable `arr` that contains array that is large enough to contain 3 elements. We can once more see that declarations in C are designed to resemble how the variable is used. As we will see, we can access elements in the array

using square brackets (e.g. `arr[0]`), which is why the size of the array appears *after* the variable name rather than as a part of the type (i.e. not `int[3] arr` as is the case in Java). The program also initializes the array to contain the numbers 1, 2, 3. As for other variables in C, initialization is optional. Note that the size of the array needs to be a constant value (either a number or the name of a constant).[6] We will see how we can allocate dynamic arrays in Section 2.7.

```
    {
        int first = arr[0];
        printf("First: %d\n", first);
    }
```

The next part of the program reads the first element of the array using the square bracket operator (`arr[0]`) and prints it. Of course the 0 could be replaced with an arbitrary expression.

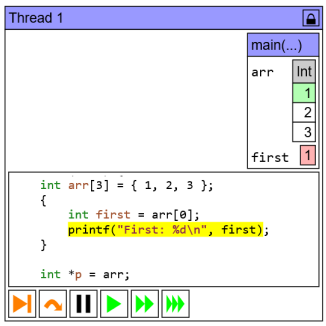


Figure 2.3: Progvis' illustration of the program `arrays.c` at the first call to `printf`.

Figure 2.3 shows how Progvis visualizes the program at this point. In the box labeled *main(...)* in the top-right part of the picture, we see that two variables (`arr` and `first`) are currently in scope in `main`. We can also see that the array (`arr`) is represented as a sequence of integer elements. Progvis labels the array with the type of the array elements, similarly to how it represents structs. The similarity between arrays and structs is no accident. Both are laid out sequentially in memory. A struct that contains 3 integer variables is actually identical to an array that contains 3 elements in memory.

```
    int *p = arr;
    {
        int first = *p;
        printf("First: %d\n", first);
    }
```

The first line in the next part of the program shows an interesting property of arrays in C, namely that they are almost always automatically converted to

[6] Some compilers do, however, support dynamically-sized arrays as local variables as an extension to the standard. There are a number of problems with this extension, in particular that it is easy to accidentally overflow the stack without the ability to easily avoid it.

pointers when they are used.[7] In this case, the program utilizes the automatic conversion to get a pointer that is stored in `p`. As we can see in Fig. 2.4, the automatic conversion produces a pointer to the first element of the array, which is how arrays are often represented in C. The next statement in the program then uses the dereference operator to access and print the first element of the array, as shown in Fig. 2.4. Again, note that `p` contains *no* information about the size of the array. It is therefore up to us as programmers to keep track of the size separately.

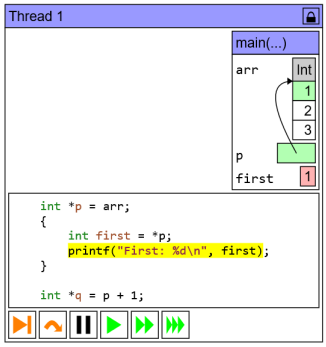


Figure 2.4: Progvis' illustration of the program `arrays.c` at the second call to `printf`. Note that `p` is a pointer to the first element of the array.

```
    int *q = p + 1;
    {
        int second = *q;
        printf("Second: %d\n", second);
    }
```

The next part of the program starts by doing something that is perhaps surprising. It creates a variable `q` and initializes it to `p + 1`. To understand why this makes sense, remember that pointers are really just integers that we interpret as the address of some other data in memory. In this interpretation, we can interpret `p + 1` to mean that we add 1 to the address stored in the pointer value. This is, however, not exactly what happens. To make it easier to work with arrays through pointers, C helps us a bit. Since we have a pointer to an integer, C will helpfully interpret `p + 1` as "compute the address of the next integer in an array". As such, instead of just adding 1 to the address, it adds `1 * sizeof(int)` instead. As shown in Fig. 2.5, this gives us a pointer to the second element in the array which we can access as `*q`. Subtraction (both between two pointers and between a pointer and an integer) works similarly.

```
    {
        int second = p[1];
        printf("Second: %d\n", second);
    }
    return 0;
}
```

[7] This is called that arrays *decay* into pointers. More or less the only situation where this does *not* happen is when using the `sizeof` operator to get the total size of the array.

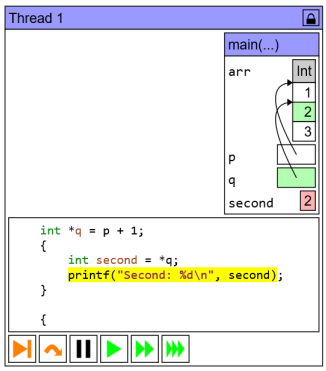


Figure 2.5: Progvis' illustration of the program `arrays.c` at the third call to `printf`. Note that `q` contains the expression `p + 1`, which is a pointer to the second element of the array.

The final part of `arrays.c`, perhaps surprisingly, uses the square bracket operator to access the second element of the pointer `p`, even though `p` is *not* an array. This works because the expression `x[y]` is equivalent to `*(x + y)`.[8] As such, `p[1]` simply means `*(p + 1)`, which as we saw earlier is equivalent to accessing the second element in the array.

This shows the deep connection between pointers and arrays. We previously saw that array variables are automatically converted into pointers when they are used. We have also seen that this conversion does not really matter, since we can treat a pointer as an array anyway, since the square brackets can be used regardless. In fact, the expression `arr[0]` we used to access an element in the array is no different from what we are doing here. The array `arr` is first implicitly converted to a pointer, and then the square bracket operator adds zero and dereferences the pointer in order to access the first element.

2.6.1 Strings

As we noted before C does not have a special type for representing strings. Instead, C represents strings as an array of characters (i.e. the `char` type). Since it is usually not possible to get the length of an array, C strings end with the NUL character (i.e. `'\0'`) by convention. This idea is illustrated by the program `strings.c`:

```
int main(void) {
    const char *str = "test";
    printf("%s\n", str);
    printf("%c\n", str[0]);
    return 0;
}
```

The program creates a variable `str` of the type `const char *` — a pointer to an array of constant characters. The `const` is needed since we are not allowed

[8] A fun consequence of this is that since `*(x + y)` is the same as `*(y + x)`, it means that `x[y]` is the same as `y[x]`. This means that `p[1]` is the same as `1[p]`, which is a very confusing way to access elements in an array. Please don't do that.

to modify the contents of the string literal (it is stored in a portion of memory that is usually write-protected).

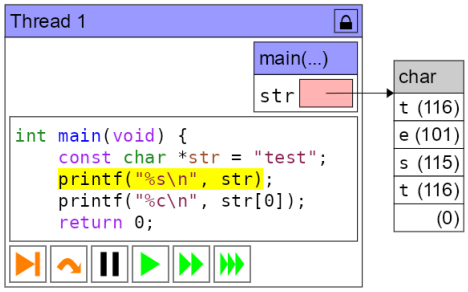


Figure 2.6: The program `strings.c` in Progvis.

Figure 2.6 shows how strings are visualized by Progvis. From the figure, we can see that `str` is a pointer to an array of characters. Each element is an 8-bit number as shown in parenthesis. To aid readability, Progvis also shows which character the number corresponds. Of particular note is that even though the string `test` only consists of 4 characters, the array has 5 elements so that there is room for a final `'\0'` at the end to signal the end of the string.

The C standard library provides a few utility functions to manipulate strings. The ones relevant to this book are:

```
#include <string.h>
```

The utilities are available in the header `string.h`. Perhaps notably, the constant `NULL` is also defined in this header.

```
int strlen(const char *);
```

Computes the number of characters in a string. Note that the final NUL character is *not* counted. As such `strlen("test")` returns 4, not 5.

```
char *strdup(const char *);
```

Creates a copy of the string by allocating memory for the string using `malloc`. The copied string therefore needs to be `free`'d later.

```
int strcmp(const char *, const char *);
```

Compares the contents of the two supplied strings. If they are equal, 0 is returned. Otherwise a positive or a negative number is returned to indicate whether the first string is lexicographically before or after the second string.

2.7 Memory Management

So far we have only used global and stack-allocated data. C also supports dynamic memory management to make it possible to work with dynamic data structures, such as dynamic arrays. This is achieved through the functions `malloc` and `free` that are defined in the `<stdlib.h>` header.

```
void *malloc(size_t size);
```

Allocates `size` bytes of memory and returns a pointer to it. If the allocation fails (e.g. because there is not enough available memory), `NULL` is returned. Note that `size_t` is an integer type similar to `int`, but it is unsigned and sometimes larger than an `int`. The contents of the allocated memory is undefined.

Typically the size supplied to `malloc` is computed using the `sizeof` operator.[9] Either as `malloc(sizeof(T))` or `malloc(sizeof(T) * x)`.

```
void free(void *memory);
```

Deallocates memory that was previously allocated by `malloc`. This makes the memory available for future allocations by `malloc`. After memory is `free`'d, it is an error to use the memory after freeing it.

The program `malloc-1.c` illustrates how `malloc` and `free` can be used to allocate structs that represent nodes in a singly linked list.

```
struct node {
    int value;
    struct node *next;
};

int main(void) {
    struct node *n = malloc(sizeof(struct node));
    n->value = 0;

    n->next = malloc(sizeof(struct node));
    n->next->value = 1;

    free(n->next);
    free(n);
    return 0;
}
```

One thing particularly worth noting is that it is *not* necessary to convert the `void *` returned by `malloc` into `struct node *`. C treats `void *` as a pointer to some data, but trusts the programmer to keep track of what kind of data. As such, in assignments and when calling functions, C allows implicitly converting a `void *` into any other pointer type (it does not even produce warnings). This makes usage of `malloc` quite convenient.[10]

Progvis illustrates heap-allocated memory as shown in Fig. 2.7. Note that memory allocated by `malloc` (also known as *heap-allocated data*) is drawn outside of the box labeled *Thread 1*. That is, all stack-allocated data is inside the box of a thread (in the top-right corner, that represents the stack) while all heap-allocated data is drawn outside the boxes of threads.

Before terminating, the program frees the memory previously allocated using `malloc` by calling `free`. Figure 2.8 shows how Progvis illustrates the situation. First and foremost, the rightmost node has a red cross drawn on top of

[9] In fact, Progvis requires that `sizeof` is used, otherwise it does not know the type of the data that is allocated.

[10] Another reason for not including the explicit type cast is that Progvis does not support `void *` or type-casts in an effort to be type-safe.

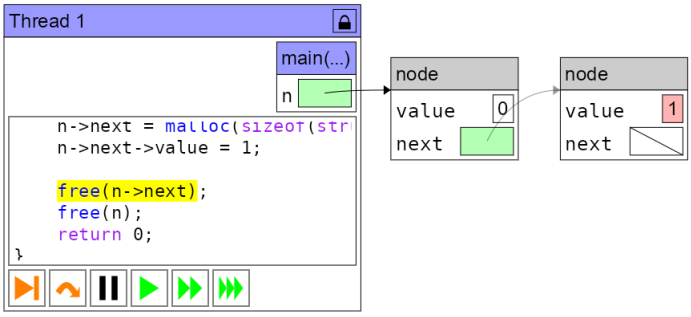


Figure 2.7: The program `malloc-1.c` visualized by Progvis.

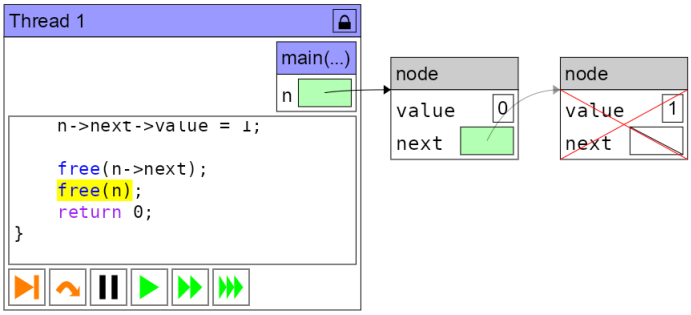


Figure 2.8: The program `malloc-1.c` after freeing the second node.

it to indicate that the memory is freed. We can also notice that `n->next` still points to the memory that was freed. Since `free` is a function just like any other, it can not modify the parameter we pass to it, and therefore it is unable to set the pointer to `NULL` for us. Even though we *could* still try to access data in `next`, such as `n->next->value`, doing so is undefined. Progvis will notify us if this happens.

It is also possible to use `malloc` to allocate dynamic arrays by simply allocating space for more than one element at once. This is illustrated by the program `malloc-2.c`:

```
struct elem {
    int a;
    int b;
};

int main(void) {
    struct elem *array = malloc(sizeof(struct elem) * 3);
    array[0].a = 0;
    array[0].b = 1;
    array[1].a = 2;
    array[1].b = 3;
    free(array);
    return 0;
}
```

Notice that the program calls `malloc` with the size `sizeof(struct elem) * 3`, which means that we requires memory that is large enough to contain 3 instances of `struct elem` back-to-back. Since pointers can be viewed as arrays, we can then use the pointer returned from `malloc` as any other array.

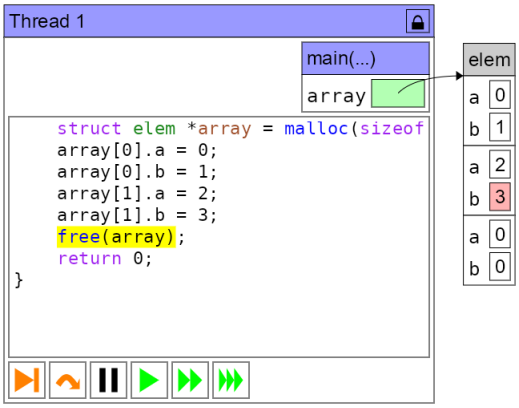


Figure 2.9: Progvis illustrating dynamic arrays allocated by `malloc-2.c`.

In Fig. 2.9, we can see how Progvis illustrates the array. As before, we can see that the array is outside the *Thread 1* box since it is heap-allocated. The representation of the array itself is otherwise identical to stack-allocated arrays. In this case the array is different from what we saw before since we allocated an array of structs, rather than an array of integers or characters.

2.8 Abstract Data Types

Structs in C can only contain variables. In particular, it is not possible to create member functions like in other object-oriented languages. This does not mean that it is impossible to create abstract data types in C, they just have to look a bit different compared to other languages. There are different ways to do this. This chapter introduces the conventions used throughout this book.

To illustrate these conventions, we will implement a simple abstract data type `vec` that represents a simple dynamic array (i.e. a simpler version of `vector` from C++). In terms of object oriented programming, the goal is to create a class (`vec`) with private data members and public member functions.

To represent this idea in C, we start by creating a struct that represent the abstract data type itself. As shown below, the struct contains an array of elements (`data`) and the number of elements in the array (`count`).

```
struct vec {
    int *data;
    int count;
};
```

The next point is to introduce a convention to determine which functions are a part of the abstract data type (i.e. which functions do we consider to be member functions in the "class"). All other functions should treat the data

type as a black box, and should not access the variables inside `struct vec`.[11] The convention used in this book is that all functions we consider to be a part of an abstract data type starts with the name of the data type followed by an underscore (i.e. `vec_` in this case). Furthermore, they should all accept a pointer to an instance of the data structure as their first parameter (this corresponds to the implicit `this` parameter in many OOP languages).

Using this convention, we can implement all functions that manipulate the contents of our simple vector. We leave construction and destruction of the abstract data type for later, as they have special semantics. In particular, we implement functions to get the number of elements (`vec_count`), to get an element by its index (`vec_get`), to set an element by its index (`vec_set`) and to print the contents of the vector (`vec_print`) as follows:

```
int vec_count(struct vec *v) {
    return v->count;
}

int vec_get(struct vec *v, int pos) {
    return v->data[pos];
}

void vec_set(struct vec *v, int pos, int value) {
    v->data[pos] = value;
}

void vec_print(struct vec *v) {
    for (int i = 0; i < vec_count(v); i++)
        printf("%2d: %2d\n", i, vec_get(v, i));
}
```

What remains is to create functions to create and destroy the data structure. There are two ways to structure this, each with its own benefits and drawbacks. The difference is whether we let the function that creates the data structure allocate memory for the data structure, or leave it to the user of the data structure. We therefore refer to them as *external allocation* and *internal allocation*.

2.8.1 External Allocation

The first option is to let the user of the abstract data type allocate memory for the struct. The memory allocation is therefore *external* to the abstract data type itself. Because of this, the function that creates an instance of the abstract data type only needs to initialize the struct to a known state. As such, we call the function `<type>_init` (i.e. `vec_init` in this case), and let it accept a pointer to the memory that should be initialized as its first parameter. In the case of `struct vec`, the `vec_init` function allocates an array of suitable size so that it can initialize `data` and `count` as below:

[11] In this book, we do not leverage C to enforce this. There are ways to hide the definition of `struct vec` so that it is not *possible* for code outside of the abstract data type to access variables in `struct vec`. This is, however, out of scope for this book.

```
void vec_init(struct vec *v, int count) {
    v->data = malloc(sizeof(int) * count);
    v->count = count;
}
```

Since the abstract data type contains dynamically allocated memory, we also need to provide a function for destroying the struct. By convention, we call this function `<type>_destroy` (i.e. `vec_destroy` in this case). It accepts a pointer to the instance that should be destroyed, and is responsible for freeing any resources allocated by the abstract data type. Note, however, since the `vec_init` function did *not* allocate memory for the struct itself, `vec_destroy` will not free the struct. Both functions leave memory management of `vec` up to the user of the abstract data type. In this case, `vec_destroy` can be implemented as below:

```
void vec_destroy(struct vec *v) {
    free(v->data);
}
```

Since `vec_init` and `vec_destroy` do not manage the memory of `struct vec`, it the responsibility of the user to manage the memory of `struct vec`. This gives the user flexibility to allocate the memory wherever is convenient in each situation. For example, the `struct vec` could be allocated on the stack, as a global variable, or in another struct. The simplest case is perhaps to allocate it on the stack as in the code below:

```
    {
        struct vec v;
        vec_init(&v, 2);
        vec_set(&v, 0, 1);
        vec_set(&v, 1, 2);
        vec_print(&v);
        vec_destroy(&v);
    }
```

The code above first allocates memory for a `struct vec` in the form of a local variable. It then initializes the memory by calling `vec_init` (and passing a pointer to the data structure). It then sets element 0 to 1 and element 2 to 1 before printing the contents of the array. Finally, it destroys the abstract data type by calling `vec_destroy`.

2.8.2 Internal Allocation

The other second option is to let the abstract data type itself handle allocation of the struct. This removes the ability of the user to choose *how* the struct is allocated. It does, however, make it possible to hide the contents of the struct from the user, thereby making it impossible to accidentally access the contents of the struct from functions that do not belong to the abstract data type.

Since the abstract data type is in charge of managing memory for its struct, the function that creates the abstract data type can simply return the created instance (in contrast to `vec_init` above). By convention, we create this function `<type>_create` (i.e. `vec_create` in this case). It does not need to accept

additional parameters, but is expected to return a pointer to the newly created instance. It can be implemented as follows:

```
struct vec *vec_create(int count) {
    struct vec *v = malloc(sizeof(int) * count);
    v->data = malloc(sizeof(int) * count);
    v->count = count;
    return v;
}
```

Note that the implementation above contains an additional call to `malloc` compared to `vec_init`. Apart from that, `vec_init` and `vec_create` are very similar.

Similarly to before, we also need to provide a function for deallocating the abstract data type. In this case, this function also frees the memory for the struct, since the memory was allocated by the `<type>_create` function. Therefore, we call this function `<type>_free`. For the `vec` type, it can be implemented as follows:

```
void vec_free(struct vec *v) {
    free(v->data);
    free(v);
}
```

Again, note that `vec_free` has an additional call to `free` compared to `vec_destroy`.

The differences between `vec_init` and `vec_create` as well as `vec_destroy` and `vec_free` also affects how the abstract data type is used. In particular, since the abstract data type is in charge of managing memory for the struct, the user of the data type just needs to store the address of the struct in a pointer. This can look like below:

```
    {
        struct vec *v = vec_create(2);
        vec_set(v, 0, 1);
        vec_set(v, 1, 2);
        vec_print(v);
        vec_free(v);
    }
```

Chapter 3

Concurrent Programming

Many operating systems allow programs to start multiple *threads* that run concurrently. These programs are sometimes called *multithreaded programs* or *concurrent programs*. We will use the term *concurrent programs* in this book.

As we shall see, writing correct concurrent programs is not as simple as starting additional threads. There are a number of additional concerns that need to be taken into account when multiple threads need to co-exist and share resources within a single process. To properly understand these concerns, we need a fairly detailed model of program execution. This chapter thus focuses on this model. We start by examining the model in terms of normal, sequential, programs and then extend it to also cover concurrent programs. We will then use the model to illustrate some of the problems that arise when multiple threads need to co-exist in a single program.

3.1 Execution Model

We will start by looking closer at how sequential programs are executed. This helps understanding why the execution model for concurrent programs is more complex compared to the model for sequential programs.

As the name implies, sequential programs can be considered to be executed in sequence, one statement after the other. To illustrate this, consider the program in Listing 3.1. The program contains two global variables, `a` and `b`. The C language guarantees that they are initialized to zero at the start of the program since they are global variables.[1] The `main` function manipulates the variables and then prints the value of them. In particular, it first stores the value 10 in the variable `a` and the value 20 in the variable `b`. After that, it reads the value in `a`, multiplies it with `5` and stores the result in the variable `a`. After that, it reads `a` and `b` in some order, adds the values together and stores the result in `b`. At the end of all of this, `a` contains 50 and `b` contains 70 as we can see by running the program.

The largest benefit of this sequential model of program execution is that it is easy to understand. This is one of the reasons why Progvis uses the model

[1] As you probably know, this is *not* the case for stack-allocated variables in functions.

```c
int a;
int b;

int main(void) {
    a = 10;
    b = 20;
    a = a * 5;
    b = a + b;
    printf("a=%d\n", a);
    printf("b=%d\n", b);
    return 0;
}
```

Listing 3.1: A simple program that manipulates global variables.

to visualize programs, and why the model that describes concurrent execution will be based upon it later on.

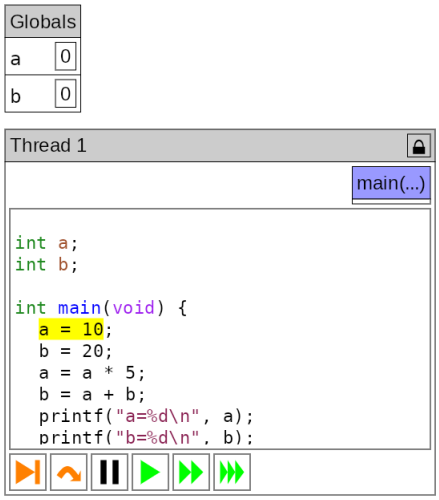
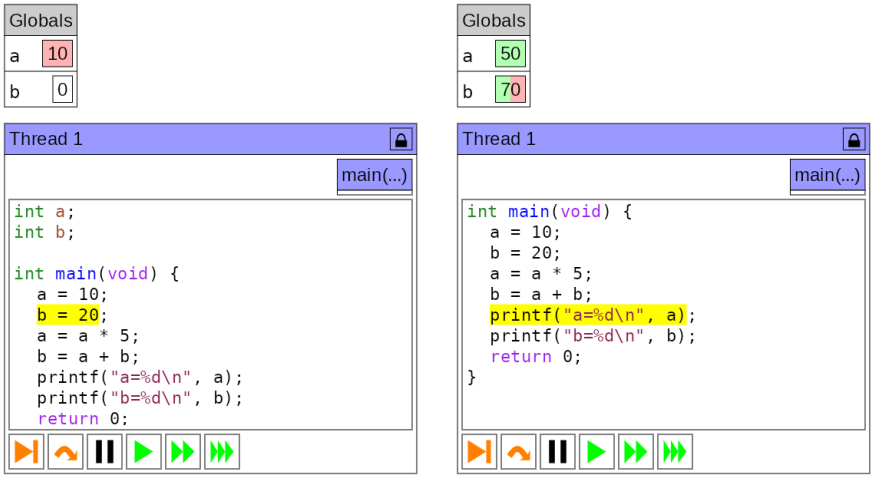


Figure 3.1: Progvis after loading the program `seq-a.c`.

To see how Progvis visualizes the program, we can open it by starting Progvis, selecting *File* ⇒ *Open program...*, and opening the file. After doing this, Progvis should display the contents of Fig. 3.1. In particular, we can see that the global variables `a` and `b` are both zero, and that Progvis paused the program just before executing the line `a = 10`, since that line is highlighted in yellow.

At this point, we want to turn off some more advanced parts of the visualization. Select *Run* ⇒ *Report data races* ⇒ *Disabled*. After that, we can inspect each step of the program execution by clicking the leftmost button in the box *Thread 1*. Each click executes the statement that is highlighted in yellow and pauses it again before the next statement. At each step, Progvis also highlights the memory locations the program accessed. Reads are colored in green and writes are colored red. For example, after executing the line `a`

= 10, Progvis shows that a was written by coloring the background in red as can be seen in Fig. 3.2a. In cases where a single variable was both read from and written to in the same statement, such as after executing b = a + b, it is highlighted in both colors as is shown in Fig. 3.2b.



(a) After running c = a * 5 (b) After running b = a + b

Figure 3.2: Progvis executing the program seq-a.c.

We can use this step-by-step model to reason about the program. This program has two distinct stages. First it manipulates data in memory (i.e. the variables a and b) and then it prints them using printf. In normal program execution, we are only able to observe the second stage, the output from printf. Because of this, the important part of the program is, in some sense, the call to printf. Most of the time we don't care how the computations leading up to printf were performed, as long as they produce the same result as the program we have written.

This idea is general enough that it is applicable to *any* program. For the purposes of this discussion, we will only consider programs running in user-mode.[2] Since programs in user-mode are isolated from each other, their behavior is only visible to the rest of the system through the system calls they perform. As such, we can think of program execution as the pseudocode below:

```
int main() {
    while (true) {
        void *syscall_args = computations();
        system_call(syscall_args);
    }
}
```

That is, a program can be thought of as performing some amount of computations that affect the contents of memory, followed by a system call. This process is repeated until the result from the computations decide to call the

[2]The reasoning works for kernel-mode and embedded systems as well. But since we can not think about system calls in those contexts, we need another definition of observability.

system call `exit` to terminate the program. To illustrate this idea, we can think of the program `seq-a.c` as three iterations of the loop above. The first call to `computations()` performs the assignments to the variables `a`, `b`, and `c` and returns the arguments for the call to `printf`.[3] The second call to `computations()` only needs to return the arguments for the second call to `printf`. The second system call would thereby `printf` again. The final call to `computations()` determines that the program is done and returns the parameters to call `exit`, which becomes the third and final system call performed by the program.

This way of thinking about programs may initially seem strange and convoluted. It does, however, give us one important insight: namely that it does *not* matter *how* the computations are performed. The only thing that matters is that `computations()` modifies global state and returns `syscall_args` so that the system call behaves properly.[4] Since it is only possible to observe the state at each system call, it does not matter in what order `computations()` is performed the computations, as long as the end result looks *as if* the program was executed line by line. Similarly, it does not matter that all computations were actually performed, again as long as the end result is *as if* they were performed.

```c
int a;
int b;

int main(void) {
    a = 10;
    a = a * 5;
    b = 20;
    b = a + b;
    printf("a=%d\n", a);
    printf("b=%d\n", b);
    return 0;
}
```

(a) `seq-b.c`

```c
int a;
int b;

int main(void) {
    a = 50;
    b = 70;


    printf("a=%d\n", a);
    printf("b=%d\n", b);
    return 0;
}
```

(b) `seq-c.c`

Figure 3.3: Two equivalent implementations of `seq-a.c`.

To illustrate the implications of this observation, consider the two versions of `seq-a.c` in Fig. 3.3. The difference between the original version and the leftmost version Fig. 3.3a is that the two lines `b = 20` and `a = a * 5` are swapped. Since the line `a = a * 5` does not use the variable `b` at all, we can quite easily see that this change does not affect the behavior of the program in terms of the final outputs from the program.

Similarly, but perhaps surprisingly, the program in Fig. 3.3b is also equivalent to both `seq-a.c` and `seq-b.c` in terms of output: if we were to observe

[3] For the purposes of this discussion, we can consider `printf` to be a system call, even though it is mostly implemented in usermode. It does, however, eventually invoke the system call `write` to output characters to the screen.

[4] The parameters to the system call may contain pointers to any memory, this is why we also need to consider at least a subset of the global state to be observable when a system call is performed.

the value of the global variables `a` and `b` at each system call, we would not be able to tell them apart. This is true even though `seq-c.c` has removed all computations that `seq-a.c` and `seq-b.c` were doing and replaced them with the results of those computations.

This notion of equivalence is important, as it is the underlying reason for why compilers can perform any form of meaningful optimizations at all. We consider two programs as equivalent if they have the same *observable* behavior. In this case, we can think of *observable* in terms of the state that is observably through the system calls they perform (i.e. which system call, and any data passed to the system calls either directly or indirectly via pointers). This also gives rise to the *as-if rule* in C and C++: the compiler is allowed to re-structure your code as much as it likes, as long as the modified version behaves *as-if* the original version was executed. As such, all versions of `seq-a.c` above are legal optimizations that a compiler may choose to do. The third one (`seq-c.c`) in particular is a common optimization called *constant-folding*. If you compile `seq-a.c` with anything but the most basic compiler with optimizations turned on, the compiler will certainly produce a program that looks very much like `seq-c.c`, even if you compiled `seq-a.c`.

While we consider the three programs `seq-a.c`, `seq-b.c`, and `seq-c.c` to be equivalent in terms of their observable behavior, they behave quite differently internally. For example, it is quite easy to tell them apart if we are able to observe all memory accesses to `a` and `b`. If we run `seq-a.c`, we will see that the value 50 is stored in `a` *after* 20 has been stored in `b`. If we run `seq-b.c` instead, the order will be reversed. Similarly, if we run `seq-c.c` we will observe that both `a` and `b` are written to once, rather than twice in the other two programs. As we saw above, these differences are usually not a problem, since the program is still executed *as if* the original program was executed and produces the same result in all cases. In fact, we *expect* our compiler to make these types of changes, since we want the compiler to optimize our code. It turns out that this kind of changes are safe to do for sequential programs in general, since it is not possible to observe the differences between the versions just by observing the program's behavior.[5] This is *not* true for concurrent programs. As we shall see, concurrent programs *are able to* observe these transformations (both by the compiler and by the hardware), and it turns out that some of these transformations will break concurrent programs. The difficult part about concurrent programming is therefore not to start threads, but rather to avoid the *problematic* transformations from making our programs behave incorrectly.

3.2 Threads, Processes and Programs

As mentioned in the beginning of this chapter, the difference between a sequential program and a concurrent program is that a concurrent program "contains" more than one thread. However, to properly understand what this means we need to take a brief detour and define some more specific terms:

Program The term *program* refers to the overall behavior of some piece of software. It is often useful to think of this in terms of the *source code* of

[5] We exclude execution time from what we consider to be observable. We *want* the compiler to reduce execution time as much as possible!

the program. The source code specifies how the program should behave. We can then compile the source code into *machine code* that is an alternative representation of the same program. Note, however, that *program* refers to a passive entity. A program by itself in this context does not *do* anything, it is just a specification of some computation that we can execute.

Process If we execute a program, we get a *process*. In contrast to a *program*, a *process* is therefore an *active* entity that is actively performing some form of computation. More concretely, a *process* is a piece of memory in the operating system that the process may use as it sees fit. One part of this memory contains the *machine code* of the *program* that the process is executing. The memory also contains other state, such as the value of global variables (e.g. `a` and `b` from the programs above), open files, etc.

Generally different processes do not share memory,[6] and they are therefore isolated enough to not interfere with each other. Furthermore, if we start the same *program* more than once, each invocation of the program becomes its own *process* with its own memory space. The different processes therefore have *their own copy* of all state, even global variables. This means that even if we run the program `seq-a.c` in multiple processes at the same time, the different processes will not interfere with each other since each process has its own copy of the global variables.

Thread Each process contains one or more *threads*. Each thread can be thought of as a virtual CPU core that executes code in the process. As such, threads are responsible for actually performing the computations described in the *program*, using the memory provided to them by the *process* they are associated with. As such, a process that contains zero threads is not able to do anything, not even launch new threads. Because of this the operating system automatically creates a thread that executes the `main` function when it creates a process for a program. This thread is sometimes referred to the *main thread* of the program. The *main thread* is usually a bit special, since the operating system terminates the entire process when the main thread terminates.

One important aspect of threads is that the belong to exactly one *process*. Each thread is able to access the memory of that particular process only. This means that two threads that belong to the same process share memory with each other, while two threads that belong to different processes do not.[7]

The relation between these terms are also depicted in Fig. 3.4. To the left is the program `multi-a` (a version of `seq-a` which we will use shortly). As before, we can think of this entity as the abstract behavior of the program, represented by the code stored on disk. The right part of the figure shows two processes that are both running the program `multi-a`. As mentioned previously, the processes are independent even though they are running the same program. This is illustrated in `fig:program-process-threads` by both

[6] Processes may actively *ask* the operating system to share a piece of their memory, but that is outside the scope of this book.

[7] See the note about explicitly shared memory above, however.

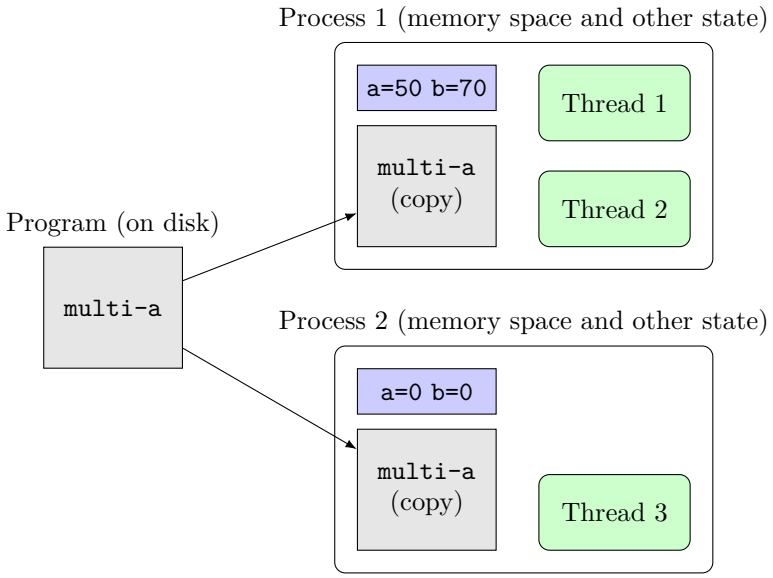


Figure 3.4: The relation between the terms *program*, *process*, and *thread*.

processes having their own copy of the code in the program, as well as their own copy of global variables. The figure also shows that a total of three threads are running. Threads 1 and 2 belong to process 1, while thread 3 belongs to process 2 (perhaps process 2 has not yet had time to start its second thread). Since threads 1 and 2 execute code in the context of the same process, they share memory with each other. In particular, both have access to all memory allocated to process 1. Thread 3 does not share anything with threads 1 and 2 since it belongs to a different process.

With a clearer view of these terms, we can more clearly define what we mean with concurrent programming. This book focuses on concurrent programming in the *shared-memory* model. That is, concurrent programming with multiple threads that communicate through memory accessible to all threads. This is the case for threads 1 and 2 in the picture above. We would therefore consider process 1 to be a process that uses concurrency. This is not the case for process 2 since it has not (yet) started additional threads. The program `multi-a` as a whole is, however, a concurrent program since it launches additional threads at least in some situations (since process 1 contains multiple threads). It is worth noting that even if we launch multiple process of a sequential program (such as `seq-a`), we would not consider the program to be concurrent since the two threads belonging to the two processes would not share memory.[8]

[8]There are, however, other forms of concurrent programming which covers this type of applications, such as message-passing.

3.3 Starting Threads

The threading library introduced in Chapter 1 with this book provides the function `thread_new` that starts a new thread in the process of the calling thread.[9] The function expects a pointer to the function the new thread should execute, followed by zero or more pointers that are passed as parameters to the function in the new thread.

```
int a;
int b;

void thread_fn(void) {
    a = 10;
    b = 20;
    a = a * 5;
    b = a + b;
    printf("a=%d\n", a);
    printf("b=%d\n", b);
}

int main(void) {
    thread_new(&thread_fn);
    thread_fn();
    return 0;
}
```

Listing 3.2: Modified version of `seq-a`.

The program `multi-a.c` (in Listing 3.2) is a modified version of `seq-a.c` that uses `thread_new` to start an additional thread. As can be seen from the figure, the code that was previously in the `main` function has been moved to a function named `thread_fn`. The `main` function first starts a new thread using `thread_new` and then calls `thread_fn`. Since `thread_fn` is passed as the first parameter to `thread_main`, the new thread will also execute `thread_fn`. The end result is therefore that we have two threads that both run `thread_fn` concurrently.

To explore exactly what this means, and what implications this has on program behavior, we use Progvis. Open Progvis and open the program `multi-a.c`. Once again, select Select *Run* ⇒ *Report data races* ⇒ *Disabled* to tell Progvis that we do not want to be informed about the concurrency issues in this program. As we shall see, there are many of them, so we start by exploring the program ourselves without having Progvis helping us identify the issues.

After loading the program, Progvis will show the starting point of the program like in Fig. 3.5. As before, Progvis has paused the program right before the first statement in the `main` function. That is, program execution was paused *before* the program had the chance to call `thread_new`, and therefore only one thread, the *main thread* is displayed currently.

If we ask Progvis to let the thread continue to the next statement, it will execute `thread_new`, which creates a new thread. Progvis illustrates the new

[9] On Linux, the underlying library call is `pthread_create`, and on Windows it is `CreateThread`.

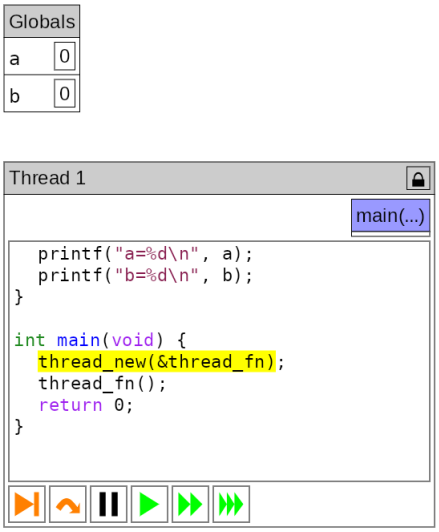


Figure 3.5: Progvis after loading the program `multi-a.c`.

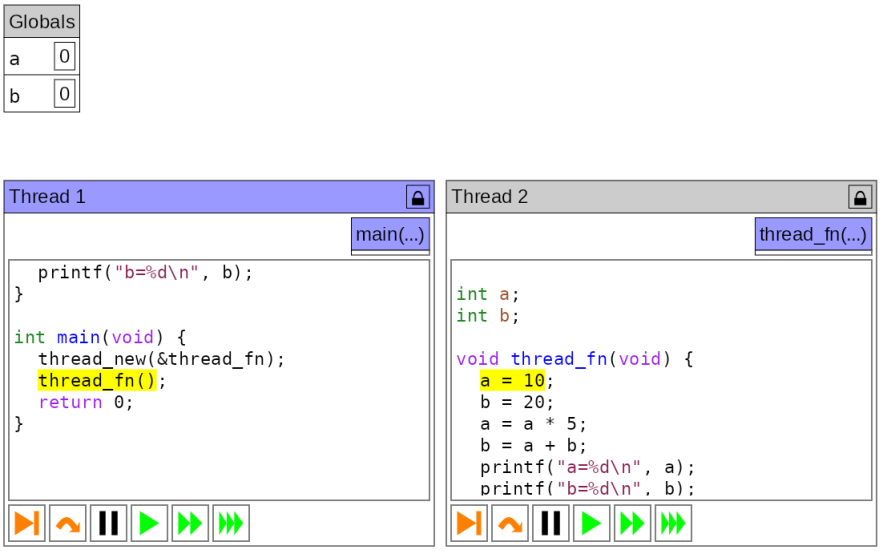


Figure 3.6: Progvis after letting thread 1 create an additional thread.

thread as a new box labeled *Thread 2*. The contents of this box is similar to the box for thread 1, but with some notable differences. In particular, they are about to execute different pieces of the program. Thread 1 is about to call `thread_fn`, while thread 2 is already inside `thread_fn` and is about to execute the line `a = 10`. Another difference is that the call stack is different. Progvis displays the call stack in the top-right corner of each box, right below the padlock. The representation in Fig. 3.6 is not very interesting at since neither `main` nor `thread_fn` have any local variables. We can, however, see the name of the currently executing function. If we ask Progvis to step thread 1

once by clicking the leftmost button in the box labeled Thread 1, it will jump into the call to `thread_fn`, and we will see the representation in Fig. 3.7.

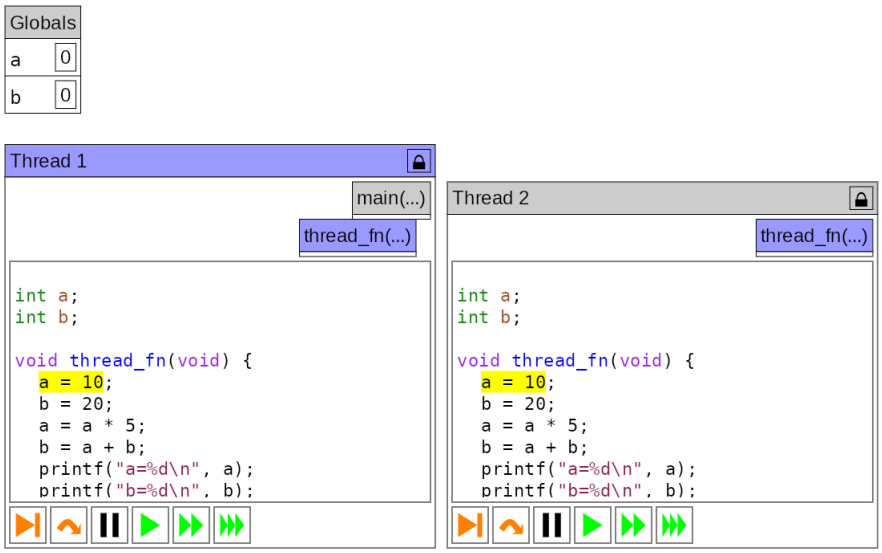


Figure 3.7: Progvis after letting thread 1 jump into `thread_fn`.

Figure 3.7 shows the representation of the stack trace in a bit more detail. In the top right corner of Thread 1's box, there are now two squares. The bottom one that is highlighted in blue is labeled `thread_fn`, and the top one is labeled `main`. This means that once thread 1 has finished executing `thread_fn`, program execution will resume inside `main`. The box labeled `main` is actually drawn behind the `thread_fn` box. This will be more apparent in future examples, where the boxes will also contain any local variables in the functions. In contrast to thread 1, thread 2 only contains one box labeled `thread_fn` in its upper right corner. This means that once thread 2 finishes executing `thread_fn`, the thread will terminate. Thread 2 will therefore never return back to `main`.

One key observation we can make at this point is that each thread has their own set of controls in Progvis. So, instead of choosing to step thread 1 before, we could also have chosen to step thread 2, which would of course have caused the program to end up in a different state. This is how Progvis represents *concurrent execution*. When two or more threads execute concurrently, the user is able to determine in which order the threads should be allowed to execute, and how long each thread should be allowed to execute before other threads get a chance to execute. For example, we can choose to let thread 1 run to completion before thread 2 gets a chance to run. We could equally well choose to let thread 2 run to completion before letting thread 1 continue, or anything in between.

This might seem odd to let the user choose between such a wide range of possible executions. However, it turns out that this matches what is allowed to happen when multiple threads coexist. All choices you can make when using Progvis are choices the scheduler in your operating system can make when your program is running. As such, the goal to keep in mind when writing con-

current programs is that the program should behave correctly for *all* possible executions. That is, regardless of which order you press the *step* button for different threads in a program, the program should behave correctly. We call one such order for an *interleaving*, since each such order corresponds to one way to interleave the statements of the different threads. As you might imagine, ensuring that the program behaves correctly in spite of this non-deterministic execution is what is difficult about concurrent programming.

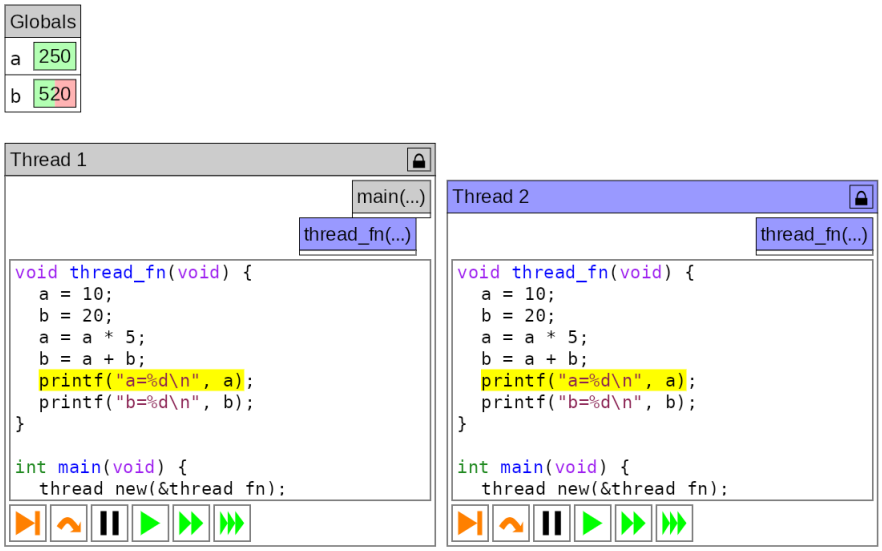


Figure 3.8: Progvis after loading the program `multi-a.c`.

Practice: There are many different interleavings of the few lines in `multi-a.c`. Some of them cause both threads to output the expected result (`a=50` and `b=70`), while others produce more unexpected results. Find an interleaving of the two threads that leads to the state shown in Fig. 3.8, where `a` is 250 and `b` is 520. Use the menu item *Run* ⇒ *Restart program* (Ctrl+R) to quickly restart the program from the beginning and try a new interleaving.

One important observation from the task above is that the possible interleavings are highly dependent on the code that is actually executed. For example, we previously concluded that we would not be able to distinguish between the programs `seq-a.c`, `seq-b.c`, and `seq-c.c` by running them since they have the same observable behavior. This is *not* true if we introduce multiple threads in our programs. If we were to explore the programs `multi-a.c`, `multi-b.c`, and `multi-c.c` (which are modified versions of `seq-a.c`, `seq-b.c`, and `seq-c.c` respectively), we would see that the possible outcomes differ, even though we previously concluded that the programs were equivalent.

Practice: Examine the programs `multi-b.c` and `multi-c.c` like you examined `multi-a.c` above. Not all programs are able to print the same values. For example, it is not possible to get `multi-c.c` to print `a=250` and `b=520`. However, both `multi-a.c` and `multi-c.c` are able to do so. Which of the programs (`multi-a.c`, `multi-b.c`, and `multi-c.c`) are able to produce the following outputs?

- `a=50` and `b=70`
- `a=50` and `b=80`
- `a=250` and `b=320`

The reason why we are able to differentiate between the programs `multi-a.c`, `multi-b.c`, and `multi-c.c`, even though we previously concluded that they were equivalent, is that our assumptions regarding *observability* no longer hold. A core assumption we made was that it is only possible to inspect the program's internal state (e.g. global variables) at each system call. This assumption is *no longer true* since the two threads in the program share memory and are therefore able to observe the program state at any time. This is the reason why we are able to produce different results from the different versions of `multi-*.c`.

3.4 Scheduling

Before we move on to see the full implications of the strange observations above, we will take a brief detour to examine how different threads are scheduled in practice. This helps understanding *why* the model for concurrent program execution looks the way it does (in particular, why we can make few assumptions in general).

Modern systems tend to have multiple physical CPU cores. Each CPU core can be thought of as an independent CPU that is able to execute its own stream of instructions in sequence. This maps very well to how threads work. We can therefore think of each CPU core as a hardware representation of a thread in our program. For example, a CPU with 4 cores can therefore execute 4 threads by allocating each thread to one of the CPU cores. In this case, we say that the threads execute *in parallel*, meaning that the threads execute instructions more or less at the same time. If the program `multi-a.c` would be executed in this way, it is possible that the two threads execute the line `a = a * 5` at the *same* time, not one after the other. As you might imagine, this can lead to interesting results, even more so than what we saw in Progvis before.[10]

Most systems have more threads active than the number of available CPU cores, so it is not possible to dedicate an entire CPU core to each thread. To solve this issue, the operating system employs a technique called *time-sharing* to allow multiple threads to share a single CPU core. To illustrate the idea, imagine that we run the program `multi-a.c` on a system that only has a

[10] Progvis does not model this form of parallel execution. As we shall see, this does not matter for its ability to detect problems. It only means that Progvis is not able to reproduce some states reachable by *incorrect* programs that might happen on a real CPU.

single CPU core. Initially, the operating system can let the main thread of `multi-a.c` uninterrupted until it starts the second thread. At this point the operating system has two threads but only a single CPU core. At this point, the operating system has a few options. It could let the main thread continue executing on the CPU core until it is done, and then switch to the second thread and let it resume. Another option would be to run the newly created thread first and let the main thread wait. A third option is to let one of the threads execute for a while (a few milliseconds, perhaps), then switch to another thread and let that execute for a while, and so on until the threads terminate. If we switch between threads fast enough, this third option gives the illusion that the system is able to execute more threads that what the hardware actually supports. However, threads do not actually execute *in parallel* in this model. We do, however, consider them to execute *concurrently* with each other. Note that threads that run *in parallel* are also considered to run *concurrently*. The term *concurrent execution* is thereby broader than *parallel execution*.

One interesting observation is that time-sharing concurrent execution is what Progvis simulates. One way of thinking of Progvis' visualization is therefore that *you* are the scheduler in an operating system with one CPU core. You are therefore in charge of deciding which thread should be executed at each time. This gives you the power to explore how different scheduling decisions affect the behavior of concurrent programs.

As you might imagine, many details of this task switching are up to the operating system. For example, how long should we let one thread execute before switching to the next one? If we switch too often, the overhead of switching between threads will reduce the overall system performance. On the other hand, if we switch too infrequently the illusion of concurrent execution will be broken, and it will instead look like we just run one thread after another. Furthermore, if the system contains more than two threads, the system also needs to decide *which* thread to run based on some criteria. All of this gets more complicated in systems that have more than one CPU core. Then the operating system also needs to determine which CPU cores should execute which threads so that threads get their fair share of CPU time.

All of these design decisions and the decisions made by the scheduler in response to current system load will affect the interleavings experienced by a concurrent program. For example, executing `multi-a.c` on a system with 4 CPU cores that are mostly idle will look very different from executing it on a system that has a single CPU core and many other threads that wish to execute. Since we want our concurrent programs to work correctly regardless of the number of CPU cores and system load, it is important that our programs work correctly for *all possible* interleavings. In Progvis, this corresponds to enumerating all combinations in which you can step the threads, trying all those combinations and verifying that the program behaves correctly in all of them. As you probably imagine, this is not practical to do for all but the simplest programs. Because of this, we will need to reason about the program's behavior at a higher level. Progvis also has a *model checker* that essentially automates checking all combinations that we will use later to verify that our programs are actually correct.

3.5 Problems in Concurrent Programs

Before generalizing our observations from the previous sections into a model for how concurrent programs execute, we will take a moment to closer investigate how the issues we found previously show themselves in programs running on real hardware. As such, in this section we will compile and run programs outside of Progvis, using the system's C compiler.

Before we proceed, it is worth mentioning that all of the programs used in this section exhibit *undefined behavior*. That means that essentially anything may happen when they are executed. While the programs are designed to most likely show the behavior described in this book, there is a chance that you will see slight differences depending on the hardware and software in your system.

```
int result;

void thread_fn(void) {
    for (int i = 0; i < 100000; i++) {
        result += 2;
    }
}

int main(void) {
    thread_new(&thread_fn);
    printf("result=%d\n", result);
    return 0;
}
```

Listing 3.3: The program `add-1.c`.

The first of the programs we will explore in this section is `add-1.c` as shown in Listing 3.3. The main thread first starts a new thread that executes `thread_fn` and then prints the contents of the variable `result`. The thread that executes `thread_fn` adds 2 to `result` 100 000 times, so the final contents of `result` will be 200 000 when the loop finishes.

Practice: Based on what we have observed from the previous section, predict the output of the program `add-1.c`. When you have done so, compile and run the program using:

```
make add-1
./add-1
```

Does the actual output match your observations? Does the program produce the same output every time you run it?

When running the program, you will most likely have seen that it does not print `result=200000` as we hoped that it would. Instead it prints some smaller number (usually between 10 000 and 20 000 on my system, but this varies greatly between different hardware). The reason for this is that the main thread does not wait for the started thread to finish executing. As soon as the main thread starts the second thread, both threads are able to execute

concurrently. This means that the scheduler may allocate them to a core at any time from that point onwards. The reason that the program produces different outputs is that the scheduler makes slightly different decisions each time the program is run.

Practice: Load the program `add-1-progvis.c` in Progvis. This program is the same as `add-1.c` with the exception that the loop only adds 2 to `result` 4 times instead of 100 000. As before, select *Run* ⇒ *Report data races* ⇒ *Disable* to tell Progvis to not complain about the issues in the program for now. Find interleavings that cause the program to print `result=`x for $x = 0, 2, 4, 6, 8$. What is the difference between these interleavings?

As you have likely seen from above, the scheduling decisions greatly affect the behavior of this program. In one extreme, the scheduler chooses to let the main thread continue to execute immediately after `thread_new` while making the other thread wait. This makes the program print `result=0`. In the other extreme, the other thread gets to execute to completion before the main thread gets a chance to continue after `thread_new`. In this case the program prints 200 000 in the case of `add-1.c` or 8 in the case of `add-1-progvis.c`. The scheduler may also choose to let the other thread run for a while before letting the main thread resume. This produces some value in between the two extremes. All of these options are legal for the scheduler to make. As such, we can not make *any* assumptions about *when* different threads execute, or even how *fast* they appear to execute in relation to each other (the other thread usually manages to execute a few thousand iterations of the loop before the main thread manages to print the result, for example).

```
int result;
bool done;

void thread_fn(void) {
    for (int i = 0; i < 100000; i++) {
        result += 2;
    }
    done = true;
}

int main(void) {
    thread_new(&thread_fn);
    while (!done)
        ;
    printf("result=%d\n", result);
    return 0;
}
```

Listing 3.4: The program `add-2.c` with an attempted solution to the problems in `add-1.c`.

We could try to solve this issue as shown in Listing 3.4 (`add-2.c`), by adding a boolean variable (`done`) that the started thread uses to tell the main thread when it is done. The variable is `false` initially. After completing the loop, `thread_fn` sets it to `true`. Finally, the main thread waits for it to become `true` using a `while`-loop.

Practice: Compile and run the program using:

```
make add-2
./add-2
```

Do you see any problems with this approach? Consider both problems related to the discussion about program equivalence from Section 3.1, and any other problems you can think of.

If you wish to explore the program using Progvis, you can use the program `add-2-progvis.c`. This program is the same as `add-2.c`, but again with a reduced number of loop iterations.

As you have likely found above, the program *appears* to work correctly. This appears to be the case even if we compile the program with optimizations (`make OPT=1 add-2`). The program does, however, exhibit undefined behavior and may do anything. This is a case where the program *happened* to work even though it is faulty.

To illustrate why, consider the program transformations that the compiler and the hardware are able to do based on the notion of equivalence covered in Section 3.1. One valid transformation of `thread_fn` is as follows:

```
void thread_fn(void) {
    done = true;
    for (int i = 0; i < 100000; i++) {
        result += 2;
    }
}
```

This is valid since it does not affect the behavior of `thread_fn` according to what is observable through system calls. Both the version above and the original version sets `done` to `true` and `result` to 200 000. However, we can quite easily see that setting `done` to `true` before finishing the computation of `result` would make the program behave just as poorly as `add-1.c`.

The reason we don't see this behavior is that the compiler performs an additional optimization when we have turned on optimizations. Since `thread_fn` does is simple enough, the compiler can see that `result` will always be 200 000 at the end. Therefore, it transforms the function to the code below:

```
void thread_fn(void) {
    result = 200000;
    done = true;
}
```

The order of the lines above is arbitrary, but do not matter in this case since `thread_fn` usually finishes before the main thread gets a chance to continue execution. This is why it appears as though our changes work in spite of

enabling compiler optimizations. However, this optimization actually hides another more serious problem in the `main` function.

If we add a call to the library function `prevent_optimization` inside the loop like below, we get the code in the program `add-3.c`. The call to `prevent_optimization` only prevents the compiler from performing the two optimizations above. It has no other side effects.

```
void thread_fn(void) {
    for (int i = 0; i < 100000; i++) {
        result += 2;
        prevent_optimization();
    }
    done = true;
}
```

Now that the compiler is unable to optimize `thread_fn` as much, the issue in `main` becomes visible. We can see the issue by compiling and running `add-3.c` as follows:

```
make add-3
./add-3
```

The program appears to never finish. If you look at the CPU usage in your system (e.g. using `htop`, press `q` to quit) you will find that the program is not idle, but is using all CPU time it can get. You can press Ctrl+C in the terminal to quit it.

What happens is that the compiler has transformed the `while (!done)` loop that the main thread uses to wait for the other thread into one of the two versions below:[11]

```
bool tmp = !done;

while (tmp)
    ;
```

```
if (!done) {
    while (true)
        ;
}
```

As we can see, the compiler has helpfully avoided reading the global variable `done` in our loop to avoid the cost of accessing memory. The left version reads `done` once and places the negation of it in the local variable `tmp`.[12] The right version also reads `done` once, but this time by extracting the loop condition into a separate if-statement before the loop.

Both of these transformations are valid for sequential programs, as they preserve the behavior of the original program. If `done` is `true` then the program continues. If `done` is `false` the program enters an infinite loop. Again, this equivalence only holds under the assumption that the program is sequential. Since we use the original loop to wait for another thread to change the variable, these transformations break our program since they only check `done` *once*. What happens when we run `add-3.c` is therefore that the main thread checks `done` *once*, and when it realizes that the other thread is not yet done, it enters an infinite loop and never checks `done` again. This is why the program never

[11] If you are curious, you can use a debugger to inspect which version your compiled produced. With `gdb`, you can type `disas main` to see the assembler version of the code.

[12] Which is then placed in a register to make it cheaper to access.

terminates. This also illustrates a problem with using a `while` loop to wait for another thread to complete, doing so wastes CPU-time doing nothing.

```
int result;
bool done;

void thread_fn(void) {
    for (int i = 0; i < 100000; i++) {
        result += 2;
    }
    done = true;
}

int main(void) {
    thread_new(&thread_fn);
    for (int i = 0; i < 100000; i++) {
        result += 5;
    }
    while (!done)
        ;
    printf("result=%d\n", result);
    return 0;
}
```

Listing 3.5: The program `add-4.c`.

All of the problems above are caused by *shared data* between the two threads. So far, we have only seen what happens when one thread (the main thread in this case) reads from shared data while another thread writes modifies it. As illustrated by the program `add-4.c` in Listing 3.5, problems also arise when two threads attempt to modify the same data. The program `add-4.c` further expands `add-2.c`, which worked correctly without optimizations. This program adds a loop in the `main` function that adds 5 to `result` 100 000 times. They are otherwise identical.

Practice: Predict the output of the program `add-4.c`. Then compile and run the program using the commands below (i.e. *without* optimizations):

```
make add-4
./add-4
```

Does the actual output match your prediction? Does the program output the same result every time?

Even though we would have expected the program to output 200 000 + 500 000 = 700 000, we typically get a number that is smaller than that, even though we know that both loops run to completion before the result is printed.

The reason for this behavior is that the addition of 2 and 5 to `result` are not performed as a single operation. Even though we use the `+=` operator, the CPU eventually needs to perform three distinct operations: 1) read the value of `result`, 2) increment the value by 2 or 5, and 3) write the value back to `result`. In C, this would look like the code below:

```
void thread_fn(void) {
    for (int i = 0; i < 100000; i++) {
        int tmp = result;
        tmp += 2;
        result = tmp;
    }
    done = true;
}
```

As such, the reason why `add-4.c` does not always output 700 000 is that some increments of `result` get lost. The `+=` operators have been expanded as above in the program `add-5-progvis.c`, so that you can explore how it affects the program.

Practice: Load `add-5-progvis.c` in Progvis. As before, select *Run* ⇒ *Report data races* ⇒ *Disable* to tell Progvis to not complain about the issues in the program for now. Find an interleaving where the program prints a result that is lower than $(2 + 5) \cdot 4 = 28$.

From the task above, you will notice that the introduction of the temporary variable means that changes to `result` are not always immediately visible to the computations done by other threads. This means that threads may accidentally overwrite the results of each others' computations. As we saw previously, this happens even if we use constructs like `result += 2` that don't *look* like they introduce temporary variables. Since the hardware eventually needs to perform this kind of operation as three separate steps, there will always be some form of temporary involved, even if it is not visible in the program's source code, or even in the machine code (e.g. if an *increment* instruction is available, which is the case on x86). Note, however, that this behavior is not visible in Progvis, since it treats entire statements as a unit. As we will see soon Progvis will still notice the problem. It will just not give an example of *what* may happen.

To illustrate exactly what happens, we can load `add-5-progvis.c` in Progvis and step both threads until they have both read the value of `result` into `tmp`. In this case `result` is 0 since it is the first iteration of both loops. This is shown in Fig. 3.9. In this figure, we can also see how Progvis visualizes local variables. Notice that the loop variable `i` is displayed inside each thread's box, under the name of the function. This representation helps remind that local variables and parameters in functions are *not accessible* to other threads by default.

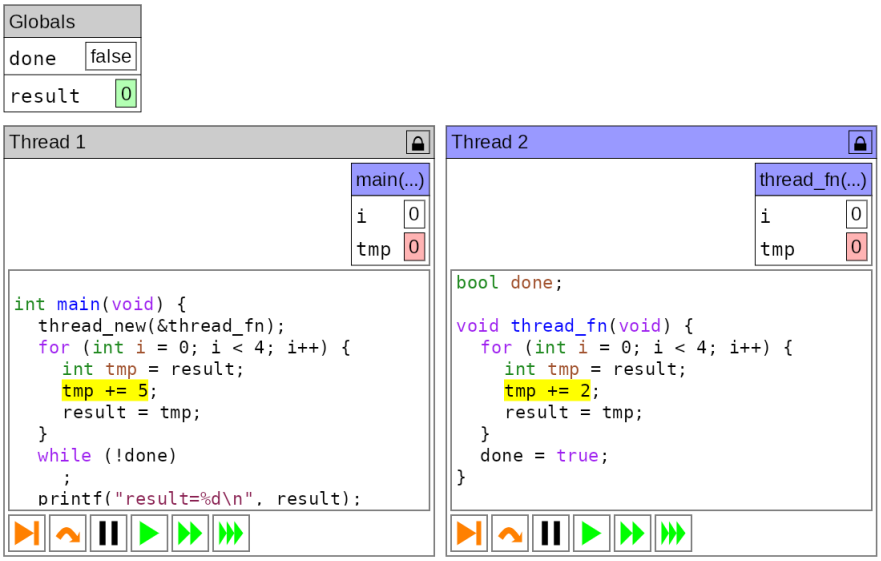


Figure 3.9: Progvis visualizing `add-5-progvis.c`.

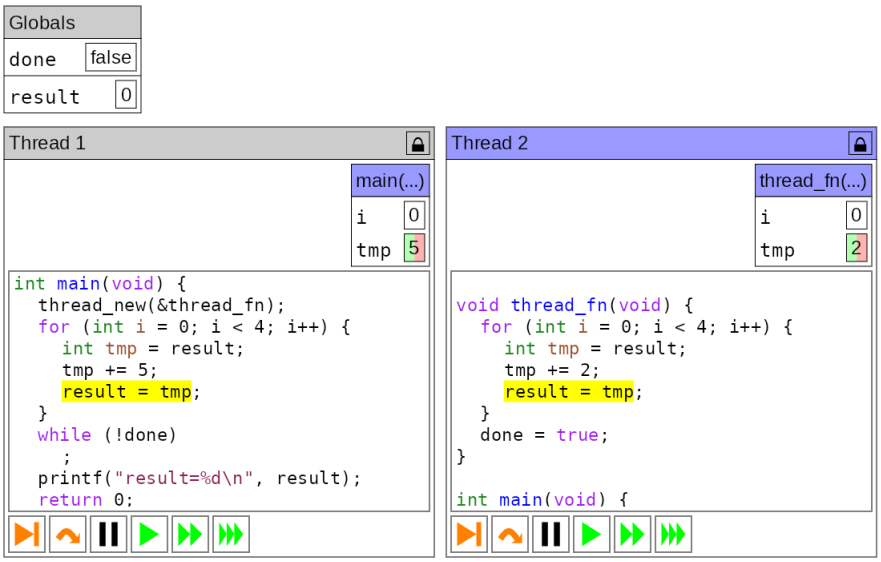


Figure 3.10: After advancing both threads one step.

Figure 3.10 shows the state of the program after we have advanced both threads to the next step. Here, we can see that both threads have updated their `tmp` variable to 5 and 2 respectively. Already here, we can imagine that the result of both threads writing their `tmp` back to `result` will not produce the desired results.

We can see that this is indeed true by advancing thread 1 to get the state in Fig. 3.11. Here, thread 1 has written its `tmp` variable to `result`, and `result` thereby contains 5. If we now let thread 2 advance one step, we will get the

Figure 3.11: After advancing thread 1.

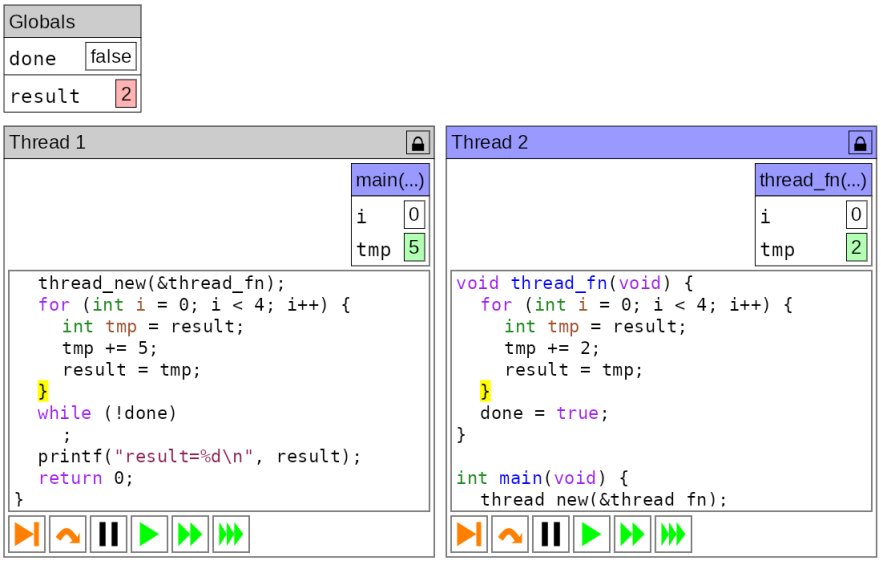


Figure 3.12: After advancing thread 2.

output in Fig. 3.12. We can see that `result` now contains 2 since thread 2 stored the value from its `tmp` variable into `result`, thereby overwriting the 5 that thread 1 stored there previously. This will eventually lead to `result` being too low, since we have "lost" the addition of 5 by thread 1.

At this point, we can notice that the above is *one* example of what can go wrong. We will "lose" more than one addition of 5 if we let thread 1 run for more iterations before stepping thread 2. Similarly, if we would have let thread 2 run first, we would have lost thread 2's work instead. What happens

in practice when we run the program outside of Progvis is some combination of all of these incorrect situations.

3.5.1 Summary

In summary, the examples from above show that *shared data* between multiple threads need special attention in concurrent programs. This is for two main reasons:

- Sometimes we need to *wait* for another thread to complete their work and write their result to some shared variable. This is the problem we saw in `add-1.c` and attempted to solved in `add-2.c`.
- We also need to be careful when multiple threads *access* the same data at the same time. Since the compiler and the hardware assumes that programs are sequential by default, access to shared data does not behave as we would expect which leads to the problems we saw in `add-2.c`, `add-3.c`, and `add-4.c`.

Even though Progvis does not model exactly how programs may misbehave if shared data is accessed concurrently, it *is* able to detect *when* problems would happen and inform you about them. Up until now, we have explicitly disabled these messages since the programs we have used so far all have major problems. If you select *Run* ⇒ *Report data races* ⇒ *Full*, you re-enable these messages. For example, if you run `add-5-progvis.c` with this setting, Progvis will notice that the two threads access the same variable concurrently and report it. Furthermore, it will draw a red cross over the problematic variable as shown in Fig. 3.13.

Figure 3.13: How Progvis illustrates which variables are accessed concurrently. Notice the red cross over the value of the global variable `result`.

3.6 The Concurrent Execution Model

Now that we have seen a number of examples of problems that arise when we introduce multiple threads to sequential programs, we are ready to summarize our findings into an *execution model*. Just as our sequential execution model, the goal is to provide a *model* that we can use to reason about the behavior of concurrent programs. We want this model to be *simple* but *accurate* so that we can understand it. It is worth noting that the model does *not* specify what happens at the compiler- or hardware level. Just as the model for sequential execution, the compiler and hardware may do whatever is efficient, as long as programs appear *as if* they are executed according to the model.

The model described here is based on the memory model used by C and C++. The memory models used by other popular languages (e.g. Java, Rust, etc.) are largely similar, but there are small differences. Luckily, the memory model used by C and C++ is usually stricter, so programs that are correct according to the model presented here are almost always correct according to the memory models in other languages as well.

The concurrent execution model can be summarized as follows. Each of the points will be discussed in more detail afterwards.

1. Programs that are free from *data races* behave *as if* they are executed in program order.
2. Multiple threads execute *concurrently* with each other. The relative execution speed and timing of multiple threads are *arbitrary* and *often vary* between different program executions. The only thing we can assume is that each thread will *eventually* execute *some* code.
3. A *data race* is a situation where one thread *writes* to data that another thread *accesses* (i.e. reads and/or writes) concurrently.
4. The behavior of a program where a *data race* is possible is *undefined*. If a data race is possible, we say that a program *contains a data race*.

The points above are intentionally short and to the point to make it easier to refer back to them later. This does, however, make them a bit hard to understand at first. Below is a more elaborate description that hopefully clarifies more of the nuances of each point:

1. This point is similar to the sequential execution model: programs are executed as if each line in the source code is executed line by line. The only difference is the point about *data races*. As we shall see, data races can not happen without multiple threads. This means that the concurrent execution model is the same as the sequential execution model for sequential programs, and as long as we make sure we don't have data races we can reason in the same way for concurrent programs.
2. This point covers the fact that concurrent execution is implemented differently on different systems (e.g. time-sharing vs. hardware parallelism), and that concurrent execution is often non-deterministic (i.e. two consecutive runs of the same program may produce different results). This means that the *only thing* we can assume is that threads will eventually

be able to execute some portion of their code. We can not make assumptions of how often this will happen, or how quickly one code executes code in relation to another thread. Therefore, if it is important that some piece of code executes before some other piece, we need to make sure that they are executed in the right order.

3. This point defines a *data race*. Intuitively, a data race occurs when two threads access the same data and at least one of them writes to the data. In the examples above, the shared data is in the form of global variables, but data may be shared via other means as well. From the examples above, we saw that all problems with concurrent executions were in some way related to shared data, and one thread modifying the data at an inconvenient time.

4. The point goes on to state that all programs where data races are *possible* are undefined. As in point 1, this essentially says that "if a data race is possible, anything may happen when the program is executed". This means that if we want to be able to reason about the behavior of our program, it may not contain any data races. In combination with point 2, that we don't know when threads execute in relation to each other, this means that data races should not be possible for *any* legal interleaving of the program. This means that we as programmers need to make sure that data races are not possible by *protecting* shared data in a suitable manner.

3.6.1 Implications of the Model

Now that we hopefully understand what the model *is*, it is also useful to understand *why* the model looks the way it does.

First and foremost, the model is similar to that of sequential program execution. This is nice, since the line-by-line execution model is simple to reason about, and most programmers are already familiar with it. Since sequential programs cannot contain data races, the concurrent execution model is the same as the sequential execution model for sequential programs. This is nice, since it lets us use *one* execution model regardless, and we can instead speak about *the* execution model.

As we saw earlier in this chapter, (almost) all problems in concurrent programs are related to shared data. Either because one thread needed to wait for another thread to produce data, or because both threads attempted to update data at the same time. Both of these situations exposed the non-deterministic nature of thread execution, or exposed compiler optimizations that re-ordered memory accesses. The execution model gets around these problems by asking the programmer to avoid data races. What this essentially means is that the programmer need to *synchronize* access to shared data, which has the effect of marking these accesses in the program. If the compiler can assume that all accesses to shared data are marked correctly, then it is able to optimize concurrent programs just as well as sequential programs, as long as it is "careful" around the marked accesses.

The big downside of just leaving the behavior of programs with data races undefined is, of course, that it is difficult to detect if a program contains data races or not. Some data races are rare in practice, so they are unlikely to

be noticed by just running the program and observing its output. It is also possible that the choices your compiler and hardware makes happen to match with what you intended the program to do. That way the program happens to work fine in spite of it containing undefined behavior. However, this may no longer be true if you upgrade your hardware and/or your compiler, at which point the program starts to fail and you don't know where to start looking. The takeaway from these observations is that it is nearly impossible to check whether or not a program is correct by just running it. It is necessary to carefully reason about the places in the program that deal with shared data!

3.6.2 The Execution Model in Progvis

Now that we know what the model says, we can also examine what Progvis visualizes in more detail. Progvis visualizes the program *as if* is is executed sequentially. According to point 1 in our model, this is in line with point 1 in the model. To help with identifying problems, Progvis also keeps track of all memory accesses made by the program. If it ever detects that a thread wrote to a variable that was accessed by another thread it knows that a data race has occurred, and reports it as such. Since data races are undefined by the model, anything may happen. As such, there is no single "accurate" way to show what might have happen in those cases. This is why we could not see some problems in Progvis — it selects *one* possible behavior for undefined programs, but some other behavior may happen in practice. However, for programs that are free from data races, Progvis matches the execution model.

It is worth noting that the observations Progvis use to detect data races are based on the interleavings selected by the user. As such, to be entirely certain that a program is free from data races, you would have to try *all possible* interleavings. As you can imagine, this is quite tedious. Therefore Progvis also has a *model checker* that tests them for you. You can start it by selecting *Run* ⇒ *Look for errors...*. If Progvis finds any interleavings that lead to a data race, it will show them as an animation. Otherwise it will report that it was unable to find any as the program is used currently.

If you have used the mode *Run* ⇒ *Report data races* ⇒ *Full* you might have noticed that Progvis sometimes colors variables in a faded-out nuance of red and green. Just like the regular, stronger, colors this represents variables that have been written to and read from. The faded-out color represents that the variables were not accessed by the last statement, but further back in the program. In particular, they were accessed since the last *barrier* (e.g. starting a new thread, protecting a variable, we will define it in more detail later). It turns out that the memory model allows us to treat blocks between such barriers as a single unit. The second button in each thread (labeled B in Fig. 1.1) jumps to the next such synchronization event.

3.7 Exercises

The code used in the exercises can be compiled either with your system's C compiler, or visualized in Progvis. If you are using Progvis, you can turn off messages about data races using *Run* ⇒ *Report data races* ⇒ *None* and turn them back on using *Run* ⇒ *Report data races* ⇒ *Full*.

1. The file `01-equivalence.c` contains four versions of the function `update`. The function `update1` is the original version of the function, and `update2`, `update3`, and `update4` are modifications to the original. Which of the modifications are equivalent to the original according to the execution model discussed in this chapter?

2. The file `02-concurrent-equivalence.c` contains the same four versions of the `update` function as in the previous exercise. This time, the main function starts another thread before calling one version.

   a) If you run the program in Progvis (turn off messages about data races) and use the original version (`update1`) you can get two different final values of `x`, which ones?

   b) If you instead let the program run other versions in each thread, what final values can `x` take then? Only use the versions of `update` that you found to be equivalent to the original one in the previous exercise.

3. The files `03-global-a.c` and `03-global-b.c` contain two similar programs. What variables are shared between threads in each of the programs? If there is any shared data, does any of the programs contain a data race?

   You can verify your solution using Progvis with messages about data races turned on.

4. The file `04-ptr.c` contains a program that launches a thread and passes a pointer to it. Does the program contain any shared data? If so, what data is shared? Does the program contain a data race?

   You can verify your solution using Progvis with messages about data races turned on.

5. The file `05-local.c` contains a program that computes 2^x in two threads. Does the program contain any shared data? If so, what data is shared? Does the program contain a data race?

   You can verify your solution using Progvis with messages about data races turned on.

Chapter 4

Semaphores

In the previous chapter we saw the problems that arise when we allow more than one thread to execute concurrently in the same process. Even though we found different types of problems, all of them originate from careless use of *shared data*. We also realized that we did not yet have the necessary tools to solve the problems we found. We tried to solve one of the problems that we found using a boolean variable and a while loop, and while the approach initially *appeared* to work, we saw that it stopped working as soon as we asked the compiler to optimize our code.[1] The problems we found can be grouped into two broad groups based on the type of solution that would be needed to solve the problem:

- One thread needs to *wait for* something to happen.
- We need to *avoid data races*, by making sure that two threads do not access the same data concurrently (i.e. ensure *mutual exclusion*).

In this chapter we will focus on solving the first problem (*waiting for* something to happen). We will, however, see that the second problem is just a variant of the first problem. It is, however, often useful to think of them as two separate problems.

4.1 Semantics

Since we are not able to solve this type of synchronization issues using clever programming techniques, we need some other tool. In particular, we need to use a *synchronization primitive*. These are implemented using specific low-level primitives that inform both the compiler and the hardware that they protect some shared data. Therefore, both the compiler and the hardware know that they need to be careful whenever some synchronization primitive is used.

A *semaphore* is one of a few synchronization primitives that is available in many programming languages. A semaphore can be considered to contain a counter variable. The counter variable is an integer that is not allowed to

[1] On some CPU architectures, the solution may not even work without turning on optimizations.

become negative. The semaphore is an opaque data structure, so the counter can not be manipulated directly. It is necessary to use the three functions below to manipulate the counter.[2] The presentation below focuses on how the functions are *used*. The definitions are available in Chapter A for the curious reader.

```
struct semaphore sema;
```

The line above defines a variable named `sema` that contains a semaphore. As with other variables in C, semaphores can be declared either at global scope, inside data structures, or as local variables inside functions.

```
sema_init(&sema, 0);
```

Each semaphore variable needs to be initialized exactly once before it can be used. This is done using the `sema_init` function as shown above. The `sema_init` function takes two parameters. The first parameter is a pointer to the semaphore that should be initialized. As such, we pass `&sema` to get a pointer to the semaphore variable. The second parameter is an integer that is used to initialize the counter inside the semaphore. As such, the second parameter cannot be negative. In this case, we initialize the counter inside the semaphore `sema` to 0.

When the semaphore is initialized, it is safe to call the functions `sema_down` and `sema_up`. Furthermore, it is safe to call these functions from different threads concurrently, which means that we can use them to solve the concurrency problems we found previously!

```
sema_up(&sema);
```

The function `sema_up` accepts a single parameter, a pointer to the semaphore that should be updated. As the name implies, the function increases the value of the counter inside the semaphore by 1.

```
sema_down(&sema);
```

The function `sema_down` also accepts a pointer to the semaphore to update as its single parameter. Again, as the name implies `sema_down` attempts to decrease the counter inside the semaphore by 1. However, remember that the counter is not allowed to become negative. As such, if a thread, T, calls `sema_down` to decrease a semaphore with a counter that is already at 0, `sema_down` will let T sleep until another thread increases the counter above zero (by calling `sema_up`). When T wakes up, it decreases the counter and continues execution.

We can make a few important observations from the descriptions above. First and foremost, we can see that `sema_up` will always increase the counter inside the semaphore by 1, and that `sema_down` will always decrease it by 1 (potentially after sleeping for some time). This makes it possible to think of the counter as if it counting the availability of some resource in the program. We will see how this can help us to synchronize programs in a bit.

[2] You can think of these functions as member functions in a class named `semaphore`. This is how they are often implemented in object oriented languages.

Another important observation is that only `sema_down` may cause a thread to sleep. For this reason, the function `sema_down` is sometimes called *wait*, and `sema_up` is sometimes called *signal*.

4.2 Semantics in Sequential Programs

Now that we know what operations are available and how they work, it is time to examine closer how they behave in actual programs. To keep things simple initially, we start with the sequential program `simple-semantics.c`.

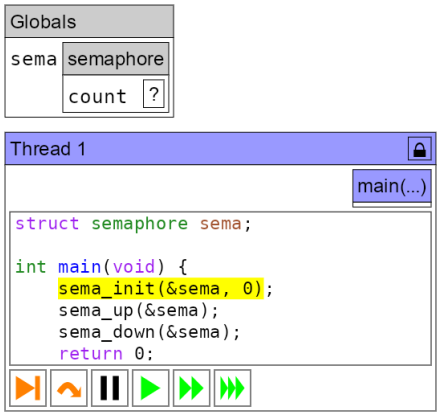
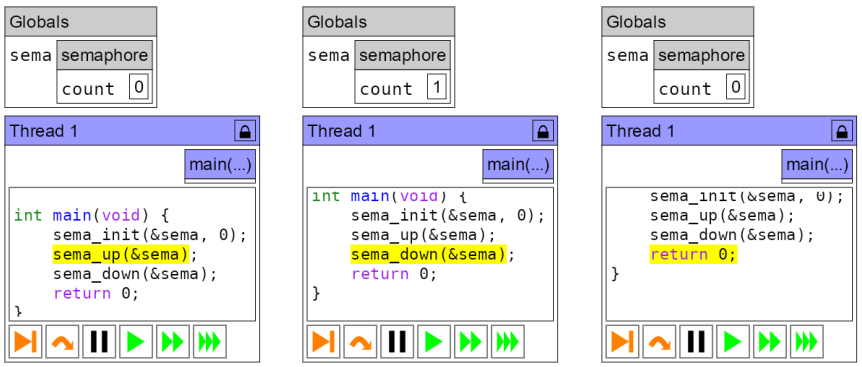


Figure 4.1: Progvis' visualization of a semaphore at the start of the program `simple-semantics.c`.

If you open `simple-semantics.c` in Progvis, you will see the representation in Fig. 4.1. Notice how Progvis visualizes the `sema` variable inside the *Globals* box. Since a semaphore is a complex data structure, it is shown as its own box. The title of the box is the name of the type (i.e. `semaphore` in this case). The type is shown to contain a single variable named `count`. This is how Progvis displays the current value of the counter inside the semaphore. Since the semaphore has not yet been initialized, the value of the counter is undefined and therefore displayed as a question mark. Notice that even though Progvis shows `count` as a member of the `semaphore` type, it is not possible to access it from the program. As mentioned earlier, it is only possible to manipulate the semaphore by using the three functions `sema_init`, `sema_up`, and `sema_down`, and none of them allows reading the value of the counter.[3]

If we continue to step thread 1 we will see the steps in Fig. 4.2. The first step (Fig. 4.2a) shows the state after running `sema_init`. Here, the counter in the semaphore is initialized to 0, since we passed 0 as the second parameter to `sema_init`. The second step (Fig. 4.2b) shows the semaphore after running `sema_up`. Since `sema_up` increases the counter by 1, the counter now contains 1. Finally (Fig. 4.2c), after running `sema_down` the counter is decreased to 0 again.

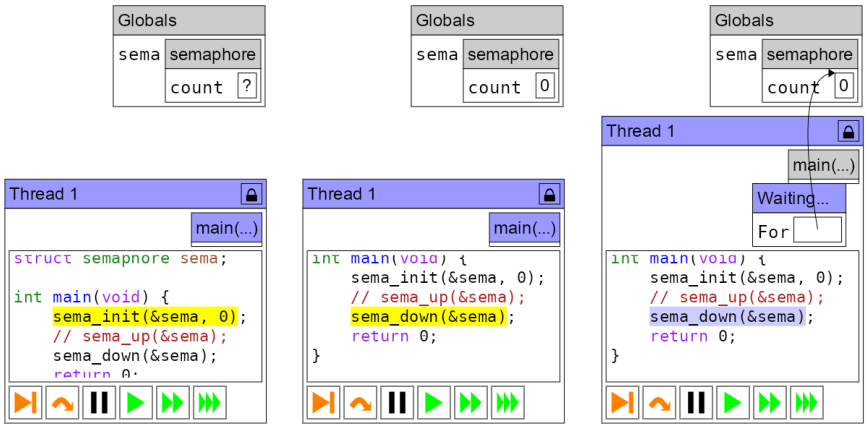
[3]The reason for this is that manually inspecting the value of the counter rarely corresponds to correct ways of using the semaphore.



(a) After `sema_init` (b) After `sema_up` (c) After `sema_down`

Figure 4.2: Progvis executing the remaining steps of `simple-semantics.c`.

The program `simple-semantics.c` has not shown what makes semaphores special yet. The behavior we have seen so far could easily have been replaced by incrementing and decrementing an integer variable. To see what makes a semaphore special, remove the call to `sema_down` in `simple-semantics.c` (e.g. by commenting it out).



(a) Before `sema_init` (b) After `sema_init` (c) After `sema_down`

Figure 4.3: Progvis executing the remaining steps of `simple-semantics.c`.

If we run the modified program and step thread 1, we will see the steps in Fig. 4.3. As we can see in Fig. 4.3b, the counter in the semaphore will be 0 when `sema_down` is called since we removed the call to `sema_up`. As mentioned before, the counter inside the semaphore is not allowed to become negative. To avoid decreasing the counter to a negative number, the semaphore puts the thread to sleep to wait for some other thread to increase the counter. In Fig. 4.3c we can see how Progvis shows a sleeping thread. First, another box is shown on top of the call to the `main` function with the label *Waiting...*, and that the current line is highlighted in purple instead of yellow. This means

that the thread is waiting for something, rather than being ready to run. This means that the scheduler may *not* schedule the thread to run at this time. If you try to step the thread in this state, you will see that nothing happens. The *Waiting...* box also contains a line *For* that shows what the thread is waiting for. In this case, the thread is waiting for the counter in the semaphore to become positive, which is illustrated by an arrow that points to the counter in the semaphore.

In this case there are no other threads in the program. Therefore, no other thread is able to increase the counter in the semaphore. This means that thread 1 will wait forever. Progvis detects this situation and reports it as a *deadlock*. We will examine deadlocks closer in a future chapter. For now, it is enough to know that if all threads in a program are waiting at the same time, Progvis will report it as a deadlock.

4.3 Semantics in Concurrent Programs

As we saw in the last example, using a semaphore in a sequential program is not very useful. Since semaphores are used to coordinate multiple threads, we also need to consider how semaphores affect the behavior of multiple threads. To illustrate this, we will examine the program `semantics.c` in more detail.

```
struct semaphore sema;

void thread_fn(void) {
    sema_up(&sema);
}

int main(void) {
    sema_init(&sema, 0);
    thread_new(&thread_fn);
    sema_down(&sema);
    return 0;
}
```

Listing 4.1: Source code of the program `semantics.c`.

As we can see in Listing 4.1, the program `semantics.c` is similar to the sequential program in the previous section. It defines a semaphore (`sema`) as a global variable and initializes it to zero in `main`. After initializing the semaphore, it starts a new thread that runs `thread_fn` and then calls `sema_down`. As such, if we just consider the contents of `main`, the program behaves exactly like `simple-semantics.c` when we removed the call to `sema_up`. The difference is that `semantics.c` started another thread that *will* call `sema_up` and thereby wake up the main thread.

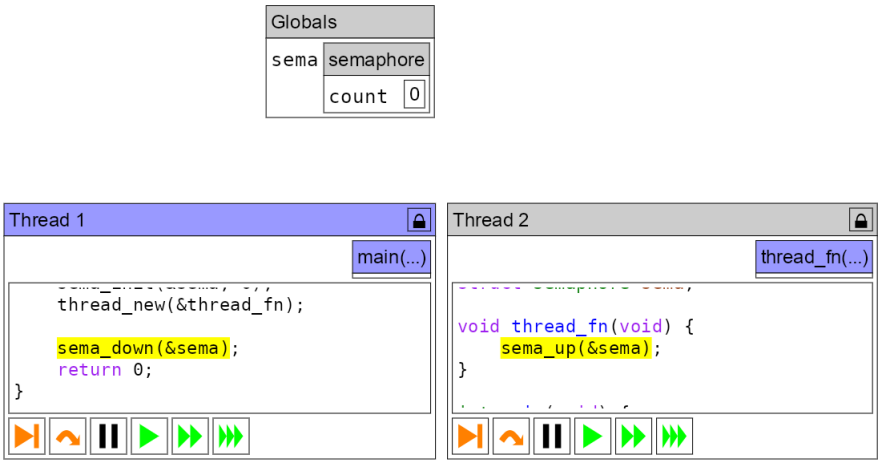


Figure 4.4: The program `semantics.c` just before thread 1 calls `sema_down`.

To see how `semantics.c` behaves, we examine it in Progvis. If we step thread 1 until just before the call to `sema_down` the program will be in the state depicted in Fig. 4.4. Just like in the previous example the semaphore's counter is at zero. This means that just as the previous example, the call to `sema_down` will cause thread 1 to wait.

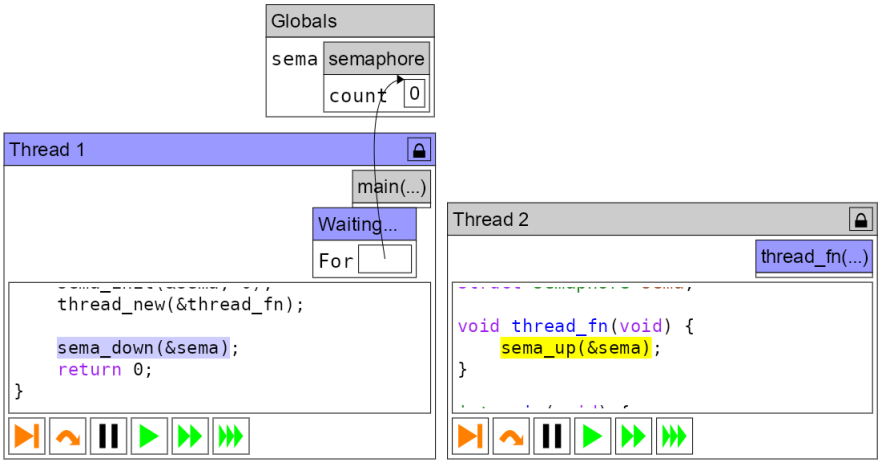


Figure 4.5: The program `semantics.c` when thread 1 is waiting for the semaphore.

Indeed, if we let thread 1 continue its execution we will reach the state in Fig. 4.5. Just as in the previous example the counter inside `sema` was zero, so the semaphore causes the thread to wait until the counter is increased. Since thread 1 is waiting, clicking its *step* button in Progvis does nothing. A similar thing happens if we run `semantics.c` outside of Progvis. When a thread is waiting it is not eligible to be scheduled by the system's scheduler, and it may therefore not continue executing. This is why the *step* button has no effect.

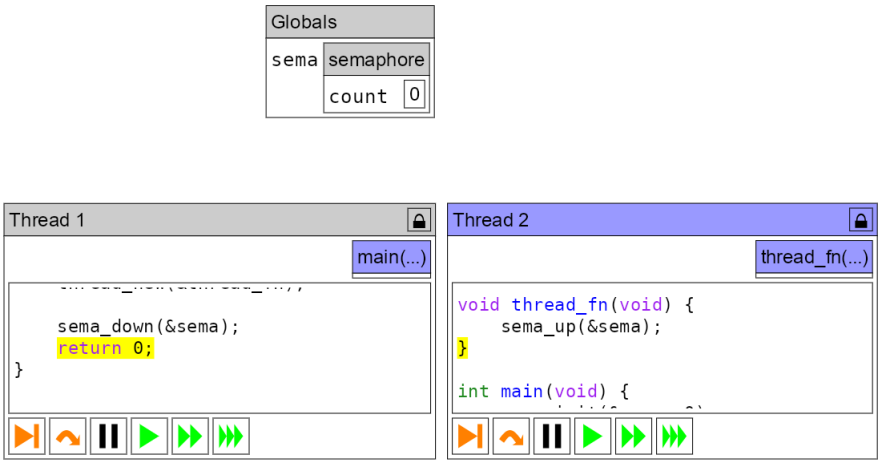


Figure 4.6: The program `semantics.c` after thread 2 has called `sema_up`.

Since thread 1 is waiting, our only option is to step thread 2. Luckily, thread 2 is about to call `sema_up`. If we step thread 2, then two things will happen after each other. Progvis do, however, consider them to be a single step, so they appear to happen at the same time:[4]

1. Thread 2 calls `sema_up` and increases the counter inside `sema` to 1.
2. Since the counter is now positive, thread 1 wakes up and continues executing `sema_down`.[5] Since the counter is 1, `sema_down` decreases the counter as usual, bringing it back to 0.

After this has happened we get the state in Fig. 4.6. In particular, we note that thread 1 is no longer waiting and that it has returned from `sema_down`. We can also see that the counter in the semaphore is still at zero, even though thread 2 called `sema_up`. As mentioned above, this is because thread 1 executed `sema_down` immediately after thread 2 called `sema_up`.

Practice: The example above showed what happens when thread 1 calls `sema_down` before thread 2 calls `sema_up`. Use Progvis to investigate what happens when thread 1 calls `sema_down` *after* thread 2 calls `sema_up`. What is the final value of the semaphore in this situation?

By examining what happens when thread 1 calls `sema_down` *after* thread 2 calls `sema_up` you should have observed that the counter in `sema` ended up at 0 regardless of the order. This is an important observation, since it means that we don't have to worry about the order in which `sema_down` and `sema_up`

[4]Depending on how semaphores are implemented, they may actually be one step. The mental model presented here is, however, still accurate and makes it easier to think about the behavior in the future.

[5]Technically, thread 1 cannot "notice" anything since it is sleeping. Thread 2 notices that threads are waiting for the semaphore when it executes `sema_up` and wakes one of them up.

was called. If we think about it, this is not too surprising. We initialized the counter to 0, and we called `sema_up` once and `sema_down` once. Since the sum of the changes to the counter is zero regardless of the order, it is natural that the final value of the counter is the same as the value we initialized the counter to. Furthermore, this is true regardless of what we initialized the counter to.

Another important observation here is that since we initialized the semaphore to 0, we know that when `sema_down` returns, some other thread must have called `sema_up` first. The reason we know this is that the counter inside the semaphore is not allowed to become negative. As such, the only way for `sema_down` to successfully decrement the counter is if thread 2 has incremented the counter first. As such, regardless of which interleaving was chosen by the scheduler, we know that the code before `sema_up` in thread 2 (marked as `A` in Listing 4.2) has to finish executing *before* the code after `sema_down` in thread 1 (marked as `B` in Listing 4.2). This is how we can use semaphores to wait for other threads to complete their work.

```
void thread_fn(void) {
    // A
    sema_up(&sema);
}

int main(void) {
    sema_init(&sema, 0);
    thread_new(&thread_fn);
    sema_down(&sema);
    // B
    return 0;
}
```

Listing 4.2: Illustration of how semaphores can be used to ensure that A completes before execution of B begins.

Practice: Using the semaphore to wait for thread 2 to call `sema_up` only works if the semaphore is initialized to 0. What happens if you change the line `sema_init(&sema, 0)` to `sema_init(&sema, 1)`? Why does the program behave differently?

4.4 Waiting for Completion

Now that we know how semaphores work, it is time to use them to solve one of the problems from Section 3.5. To make it easier to visualize the program flow in Progvis, we will focus use the version of the problem adapted to Progvis in this chapter (i.e. the ones called `*-progvis.c` in Chapter 3). As such, in this chapter the files called `add-N.c` contain the version adapted for Progvis. Of course, the solution presented here works equally well for both versions of the problem since the only difference is that the number of iterations in the loop have been reduced from 100 000 to 4.

The file `add-1.c` (Listing 4.3) contains our attempted solution to the problem from Section 3.5. Remember that we realized that the code in `thread_fn` needs to run to completion before the main thread calls `printf`. We attempted

```
int result;
bool done;

void thread_fn(void) {
    for (int i = 0; i < 4; i++) {
        result += 2;
    }
    done = true;
}

int main(void) {
    done = false;
    thread_new(&thread_fn);
    while (!done)
        ;
    printf("result=%d\n", result);
    return 0;
}
```

Listing 4.3: The attempted solution from Section 3.5.

to solve this by adding the variable `done`, and wait for it to become `true` by using a `while` loop. However, since `done`[6] is accessed from both threads concurrently and thereby constitutes a data race. Since data races are undefined, anything may happen when we run the program. As we saw earlier, the program appeared to work fine initially, but started to misbehave once we turned on optimizations.

Practice: Open `add-1.c` in Progvis. Make sure that *Run ⇒ Report data races* is set to *Full*. Find an interleaving that causes Progvis to report a data race that involves the `done` variable.

To solve this issue we need to use some synchronization primitive to avoid data races. Since we have only introduced semaphores so far, we will use a semaphore to solve the problem. Luckily, the problem here has a similar to the structure we saw in Listing 4.2. Since our goal is to the code in `thread_fn` complete before we run the last part of `main`, we can do the following:

1. Remove `done` and replace it with a semaphore that we call `has_result`.
2. Instead of initializing `done` to `false` at the start of `main`, we initialize the semaphore to zero.
3. Instead of the `while`-loop in `main`, we call `sema_down`.
4. Instead of setting `done` to `true` at the end of `thread_fn`, we call `sema_up`.

These changes result in the code in `add-2.c` (Listing 4.4). The variable `done` no longer causes a data race since it was replaced with the semaphore `has_`

[6] Technically also `result`, but as we will see we don't have to worry about it if we solve the problem with `done`.

```
int result;
struct semaphore has_result;

void thread_fn(void) {
    for (int i = 0; i < 4; i++) {
        result += 2;
    }
    sema_up(&has_result);
}

int main(void) {
    sema_init(&has_result, 0);
    thread_new(&thread_fn);
    sema_down(&has_result);
    printf("result=%d\n", result);
    return 0;
}
```

Listing 4.4: A correct solution using a semaphore.

`result`. Sharing a semaphore (or other synchronization primitives) in this way is safe, as long as the semaphore is initialized *before* it is shared. We initialize the semaphore in the main function, before the second thread is started. This makes it impossible for the second thread to access the semaphore before it is initialized. After that, we only access the semaphore using `sema_down` and `sema_up`, which are designed to be safe to use from multiple threads concurrently.

What is perhaps less obvious is why the global variable `result` does not lead to a data race even though it is used by both threads. Remember that for a data race to occur, threads have to access shared data concurrently, and at least one access has to be a write. Here, `result` is shared, both threads access the variable, and one of them write to the variable (`thread_fn`). However, `thread_fn` and `main` do *not* access `result` concurrently. The `main` function only accesses `result` after calling `sema_down`. Since `sema_down` waits until the second thread calls `sema_up`, we know that the second thread has finished its work that involves `result` when `main` is allowed to continue.

4.4.1 Verification in Progvis

We can verify that the program is correct using Progvis. To be sure that the program is correct, we would need to try all possible interleavings of the two threads. In this case there are not very many interleavings, but it is easy to miss one or more. To help out in situations like this, Progvis contains a *model checker* that automates this. Intuitively, the model checker simply tries all possible interleavings and verifies that none of them cause an error.[7] To run the model checker, simply open the program in Progvis and select *Run* ⇒ *Look for errors*. For `add-2.c`, Progvis will quickly conclude that the program is free from concurrency errors.[8]

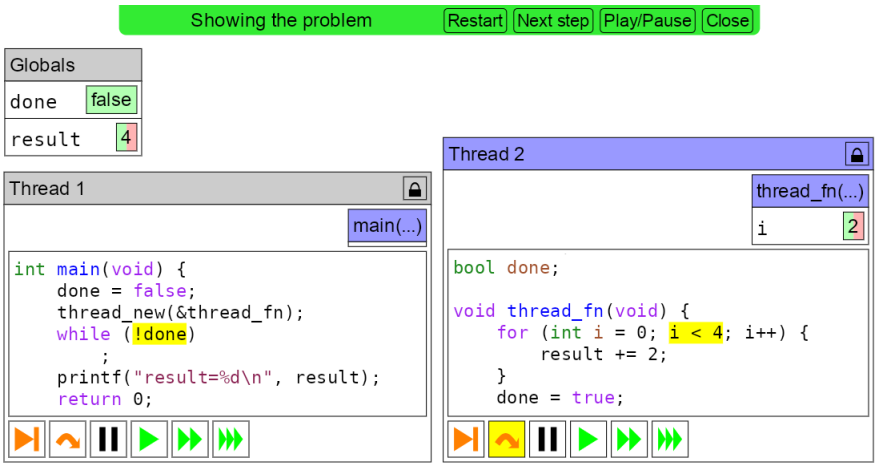


Figure 4.7: Progvis showing an interleaving that leads to a data race.

If the loaded program contains concurrency errors, Progvis will provide an example of an interleaving that causes the issue it found. To see how this works, load the program `add-1.c`, which we know contains an error, and select *Run* ⇒ *Look for errors* once again. Progvis will quickly find a *data race* and notify you about this. When you click the *Show* button you will now see a green bar with the title *Showing the problem* at the top of the Progvis window, as shown in Fig. 4.7. This green bar indicates that Progvis is now in control of stepping the program. In this state you are therefore not able to step threads manually (clicking the buttons will have no effect). Instead, Progvis highlights the button that corresponds to the next step in yellow. In Fig. 4.7, the second button in the right thread is highlighted. This means that when you click *Next step* in the green bar, Progvis will click that button for you, thereby letting thread 2 execute for a while, and then highlight the next step in the problematic interleaving. If you don't want to click *Next step* manually, you can also click *Play/Pause* to have Progvis automatically go through the full sequence for you. At any point, you may click *Close* to go back to the normal view where you

[7] It does this in a somewhat intelligent manner to reduce the search space, but the end result matches this intuition.

[8] The message includes "...at least not with this main program." to remind you that it will only find errors in code that is actually executed by your `main` function. We will see the importance of this later in the book.

are in control. You can even click *Close* mid-way through the sequence to see what happens if you make different decisions towards the end of the sequence.

Practice: The file `add-3.c` (see below) contains 3 commented lines marked A, B, and C. Which of these three lines cause a data race when uncommented? First try to reason about the behavior of the program, and then use the model checker in Progvis to verify your answer.

```
int result;
struct semaphore has_result;

void thread_fn(void) {
    for (int i = 0; i < 4; i++) {
        result += 2;
    }
    sema_up(&has_result);
    // result += 1; // (A)
}

int main(void) {
    sema_init(&has_result, 0);
    // printf("result=%d\n", result); // (B)
    thread_new(&thread_fn);
    // printf("result=%d\n", result); // (C)
    sema_down(&has_result);
    printf("result=%d\n", result);
    return 0;
}
```

4.4.2 How to Use Semaphores?

Now that we have seen how to solve a simple problem in practice, it is worth taking a moment to reflect on the solution process. In this case, we were lucky enough that the program we wanted to synchronize was similar enough to our previous experiment. As you might imagine, many situations are not that clear-cut in practice. We want to find a more general approach.

In Section 4.1 (when we introduced the operations), we noted that it is possible to consider the semaphore to keep track of some resource in the program. This view is indeed often useful to help figure out what value to initialize the semaphore to, when to call `sema_up` and when to call `sema_down`.

To illustrate this idea, consider the program `add-2.c` that we just added a semaphore to. The semaphore here is named `has_result` rather than `done`. This change in name was done to highlight the idea that the semaphore counts the number of finished results that have been produced by the other threads. In this case we only have one other thread, so the semaphore will either contain 0, meaning "`result` does not contain anything useful yet" or 1, meaning "`result` contains a meaningful value".

Once we have found a proper resource to count, the usage of the semaphore follows naturally. Since we don't have a result at the start of the program, it follows that we need to initialize the semaphore to 0. When `thread_fn` is done, it has *produced* a value inside `result`. As such we want to increase the

semaphore from 0 to 1, hence we call `sema_up`. Similarly, in `main` when we call `printf`, we want to *consume* a result, so we call `sema_down` to decrease the semaphore. Note that we don't technically "destroy" the contents of the variable in this program. The idea of *producing* and *consuming* values is, however, useful regardless as it allows us to think about the program in terms of which thread "owns" some shared data currently. In this case we consider `thread_fn` to "own" the variable `result` initially. This means that `thread_fn` is free to do what it pleases with the variable until it declares that it is done by calling `sema_up`, thereby officially declaring that it has *produced* the final value of `result`. This allows `main` to *consume* the value (i.e. `sema_down` is allowed to return). At this point `main` has successfully *consumed* the value, and thereby "owns" the variable `result`, again meaning that `main` may do what it pleases with the variable. If other threads would be interested in the value, we could let `main` *produce* the value in `result` after printing it (an potentially modifying it). This would let other threads *consume* it and use it for themselves.

As you might imagine, the difficult part of the idea of counting some resource is to *find* a suitable resource to count. This is crucial since only `sema_down` waits when the semaphore's counter would become negative. The key idea is to find some resource that is zero exactly when some thread wishes to wait for it. One way to perform a sanity-check of some choice is to first add the relevant calls to `sema_init`, `sema_up`, and `sema_down` as dictated by the choice of resource. Then, imagine that any threads started by calls to `thread_new` take a very long time to start, and think about what happens when the remaining thread continues to execute. Will it eventually reach a call to `sema_down` where you would expect the thread to wait? Furthermore, does the semaphore passed to `sema_down` contain zero at that time? If so, you have likely picked a good resource to count. We can see this in `add-2.c`. If we imagine that the thread running `thread_fn` is delayed for a long time, we can easily see that the main thread quickly reaches `sema_down` and that the semaphore `has_result` is zero, which means that the thread will wait for `thread_fn` to start. This can either be done by tracing the program in your mind, or by using Progvis and simply choosing to only step one of the threads.

Practice: The program `add-two.c` (see below) contains a program that is similar to the one we have used so far. The difference is that it starts *two* threads and produces *two* result variables (`result1` and `result2`). How can we use *a single* semaphore to synchronize the program? You can solve the problem by adding one or more calls to `sema_init`, `sema_up`, and `sema_down` to the lines that contain a question mark (?). If done correctly, the program should always print `result_a=8` and `result_b=12`. You can use the model checker to verify your solution.

Hint: Consider `result_a` and `result_b` to be a single resource, and count how many of them are filled.

```
int result_a;
int result_b;
struct semaphore has_result;

void thread_fn_a(void) {
    for (int i = 0; i < 4; i++) {
        result_a += 2;
    }
    // ?
}

void thread_fn_b(void) {
    for (int i = 0; i < 4; i++) {
        result_b += 3;
    }
    // ?
}

int main(void) {
    // ?
    thread_new(&thread_fn_a);
    thread_new(&thread_fn_b);
    // ?
    printf("result_a=%d\n", result_a);
    printf("result_b=%d\n", result_b);
    assert(result_a == 8);
    assert(result_b == 12);
    return 0;
}
```

4.5 Multiple Semaphores

In many cases it is not enough to use a single semaphore to synchronize a program. In particular, when we want a thread to wait, we usually wants it to *wait for* something particular to happen. If there are multiple things that happens in the program that some thread might want to wait for, then we need one semaphore for each thing we want to wait for to achieve this.

To illustrate this idea, consider the program in `add-multi-1.c` (also in Listing 4.5). It is a version of the program `add-2.c` but with two threads, `thread_fn_a` and `thread_fn_b`, that write to `result_a` and `result_b` respectively. As

```c
int result_a;
int result_b;
struct semaphore has_result;

void thread_fn_a(void) {
    for (int i = 0; i < 4; i++) {
        result_a += 2;
    }
    sema_up(&has_result);
}

void thread_fn_b(void) {
    for (int i = 0; i < 4; i++) {
        result_b += 3;
    }
    sema_up(&has_result);
}

int main(void) {
    sema_init(&has_result, 0);
    thread_new(&thread_fn_a);
    thread_new(&thread_fn_b);

    sema_down(&has_result);
    printf("result_a=%d\n", result_a);

    sema_down(&has_result);
    printf("result_b=%d\n", result_b);

    return 0;
}
```

Listing 4.5: The program `add-multi-1.c`.

before, the code in `main` starts the threads, and prints the results after waiting for the threads to finish. In this case, the main function wishes to print `result_a` first. To do this, the main thread only needs to wait for `thread_fn_a` to complete, and therefore it calls `sema_down` once before printing the result. Afterwards, it calls `sema_down` once more to wait for `thread_fn_b` to complete and then prints `result_b`.

Practice: The program `add-multi-1.c` is not correct. It contains a data race and may print `result_a=0`. Can you see why? Use Progvis to find the error. Start by trying to find the issue yourself by stepping the program manually. You can also use the model checker to find the problem.

As you likely realized above, the fundamental problem is that we are not able to specify *which* thread to wait for since we only have one semaphore. Using the analogy of resources from the previous section, we have decided to use the semaphore to count *how many results are produced*, thereby treating both

results as equivalent.[9] This means that after we call `sema_down` once in the main function, we know that *one* result is ready, but we don't know which one. As such, if `thread_fn_b` finishes first and signals its semaphore, the main thread will wake up and print whatever is in `result_a`, even though `result_a` was not yet ready.

```
int result_a;
int result_b;
struct semaphore has_result_a;
struct semaphore has_result_b;

void thread_fn_a(void) {
    for (int i = 0; i < 4; i++) {
        result_a += 2;
    }
    sema_up(&has_result_a);
}

void thread_fn_b(void) {
    for (int i = 0; i < 4; i++) {
        result_b += 3;
    }
    sema_up(&has_result_b);
}

int main(void) {
    sema_init(&has_result_a, 0);
    sema_init(&has_result_b, 0);
    thread_new(&thread_fn_a);
    thread_new(&thread_fn_b);

    sema_down(&has_result_a);
    printf("result_a=%d\n", result_a);

    sema_down(&has_result_b);
    printf("result_b=%d\n", result_b);

    return 0;
}
```

Listing 4.6: The program `add-multi-2.c` which fixes the issue in `add-multi-1.c`.

To solve the problem, we need to use two separate semaphores. One that counts whether `result_a` contains a result, and another that counts whether `result_b` contains a result. This can be implemented as in `add-multi-2.c`. The program replaces the single semaphore `has_result` with two separate semaphores, `has_result_a` and `has_result_b`. The function `thread_fn_a` then calls `sema_up` on `has_result_a` when it is done producing `result_a`.

[9] As you probably saw in the last section, this is completely fine to do as long as we wait for both results to become ready, or have some other way to determine which result to actually use.

Similarly, `thread_fn_b` calls `sema_up` on `has_result_b` when it is done producing `result_b`. This means that when the main thread wishes to consume `result_a`, it can call `sema_down` on `has_result_a`. This means that the main thread no longer waits for any of the two threads to become ready, but instead explicitly waits for `thread_fn_a` which produces `result_a` to become ready. This change solves the problem, and the program now works as expected.

Practice: Use Progvis to compare the differences between `add-multi-1.c` and `add-multi-2.c`. Pay special attention to what happens if you let the main thread execute until it starts waiting during its first call to `sema_down`, and you then let the thread running `thread_fn_b` (thread 3) run to completion without touching the thread running `thread_fn_a` (thread 2).

The situation above illustrates why we need to use multiple semaphores to specify what to wait for. A similar (but related, depending on your view) situation is when it is important *which* out of multiple threads that should wake up when `sema_up` is called. This issue is illustrated by the program `multi-sema-1.c` (see Listing 4.7).

```
int result;
struct semaphore has_result;

void thread_a(void) {
    result = 10;
    sema_up(&has_result);
}

void thread_b(void) {
    sema_down(&has_result);
    result += 5;
    sema_up(&has_result);
}

int main(void) {
    sema_init(&has_result, 0);
    thread_new(&thread_a);
    thread_new(&thread_b);

    sema_down(&has_result);
    printf("result=%d\n", result);
    // assert(result == 15);

    // Avoid confusing the model checker:
    sema_up(&has_result);
    return 0;
}
```

Listing 4.7: The program `multi-sema-1.c`.

This program resembles the programs we have seen so far, but with some differences. First and foremost, the loops are replaced with assignments. This

simplification is just done to help focus on the synchronization issues, and has no impact on the synchronization. Perhaps more importantly, the program now attempts to run the code in the three functions in sequence. First, `result = 10`, then `result += 5` and finally `printf(..., result)`.

To achieve this, the programmer considered the semaphore `has_result` to count whether `result` contains a useful value or not. Similarly to before, the programmer then thought that `thread_a` first "owns" the variable `result`. As such, the thread is free to store the value 10 in `result`. It then declares that a value has been *produces* by calling `sema_up`. As such, some other thread may now take over ownership of the variable by *consuming* it using `sema_down`. The intention was that `thread_b` would then *consume* the the variable by calling `sema_down`. Once `sema_down` returns, the thread owns the variable and may update it by increasing it by 5. When it is done, it declares that it has *produced* a new value in the variable and calls `sema_up`. This means that the main thread is finally able to *consume* the value and print the final result.

> **Practice:** As you might have realize from the wording above, the program `multi-sema-1.c` does not work as intended. The intended behavior is for the program to always print `result=15`. However, the program sometimes prints `result=10`. In contrast to most other examples we have seen so far, this program is actually free from data races and thereby well-defined according to the C standard. It misbehaves in spite of this.
>
> Use Progvis to examine the program and find the problem. Since the model checker does not know what the expected output of the program is, it will not automatically find the error. You can use an `assert` statement (available as a comment in the program) to let the model checker know that you expect `result` to be 15. That way, the model checker will be able to find the error.

From above, you have likely realized that the problem in `multi-sema-1.c` is that when `thread_a` calls `sema_up`, *one of* the two threads (`thread_b` and `main`) are allowed to continue. The programmer intended that `thread_b` should always wake, but as we saw above this is not always the case. It is equally valid for the implementation to let the main thread wake up immediately, which means that `thread_b` never gets a chance to execute.[10] This is a key insight, which is worth highlighting:

- When one thread calls `sema_up` on a semaphore that two or more threads are waiting for, the implementation chooses any one of the threads to wake up. This choice is *arbitrary*.

Since the implementation may pick an *arbitrary* thread out of the ones that are waiting, it means that it is equally valid for either `thread_b` or `main` to wake up when `thread_a` calls `sema_up`. This arbitrary choice is what causes the non-deterministic behavior in `multi-sema-1.c`. In this case it leads to the program behaving incorrectly, since the program did not take this behavior into account.

[10] This is the reason for the extra `sema_up` at the end of `main`. Without it, `thread_b` never wakes up, which the model checker in Progvis reports as a deadlock.

Before we continue, it is worth noting that Progvis simulates a slightly stricter model in order to be easier to understand. In Progvis, semaphores (and other synchronization primitives) wake threads in the order they started waiting, using a FIFO queue. This is *one* of many valid choices for this arbitrary behavior. Semaphores in your system are likely not this strict, even though they will still be fair in the long run (i.e. if we do the same thing many times after each other, it will pick either thread roughly 50% of the time). However, in spite of the semaphore being completely fair, we see the same problem in Progvis. The reason for this is that we generally cannot control in what order threads start waiting for the semaphore. This uncertainty therefore gives rise to the same type of non-determinism that is already a part of the semaphore, and the reason why it is usually not worth the performance impact of implementing a FIFO behavior in practice.

```
int result;
struct semaphore thread_a_done;
struct semaphore thread_b_done;

void thread_a(void) {
    result = 10;
    sema_up(&thread_a_done);
}

void thread_b(void) {
    sema_down(&thread_a_done);
    result += 5;
    sema_up(&thread_b_done);
}

int main(void) {
    sema_init(&thread_a_done, 0);
    sema_init(&thread_b_done, 0);
    thread_new(&thread_a);
    thread_new(&thread_b);

    sema_down(&thread_b_done);
    printf("result=%d\n", result);
    assert(result == 15);
    return 0;
}
```

Listing 4.8: The program `multi-sema-2.c` which solves the problems from `multi-sema-1.c`.

To be able to control which thread wakes up when we call `sema_up` we must once again use more than one semaphore. That is, instead of treating whether `result` contains a value or not as a *single* resource that we count with a semaphore, we think of it as two different resources. The first one is: "has `thread_a` produced a value in `result`?" and the second one is "has `thread_b` updated the value in `result`?". Again, if we replace the single semaphore `has_result` with two different semaphores to reflect this new view of the problem

(we call them `thread_a_done` and `thread_b_done`), we get the program in `multi-sema-2.c` (Listing 4.8).

It is particularly interesting to note that it is now clear that `thread_a` produces a result that only `thread_b` consumes, since those are the only two functions that use `thread_a_done`. Similarly, the link between `thread_b` and `main` is clear, since those two functions are the only places where `thread_b_done` is used. In essence, we have applied the pattern of using semaphores to wait for something twice.

It is worth noting that some care needs to be taken in cases like this. Here we have multiple semaphores that together make sure that `result` is never accessed by more than one thread at the same time. In this case, it is quite easy to see that this is the case since the semaphores cause the threads to execute in a fixed order. This may, however, not always be the case.

To make the separation of a single resource into two separate resources clearer, we could have also split the `result` variable into two separate variables: `result_a` that contains the output from `thread_a`, and `thread_b` that contains the output from `thread_b` (see `multi-sema-3.c` and Listing 4.9). This division makes it clearer that we are working with two different resources, and thereby what the purpose of the two semaphores are. It is, however, not always trivial to do this kind of restructuring.

```
int result_a;
int result_b;
struct semaphore thread_a_done;
struct semaphore thread_b_done;

void thread_a(void) {
    result_a = 10;
    sema_up(&thread_a_done);
}

void thread_b(void) {
    sema_down(&thread_a_done);
    result_b = result_a + 5;
    sema_up(&thread_b_done);
}
```

Listing 4.9: Another possible solution of the problem in `multi-sema-1.c`. The full code is available in `multi-sema-3.c`.

4.6 Mutual Exclusion

As mentioned in the beginning of this chapter, concurrency problems can be grouped into two large groups. We have now seen how semaphores can be used to solve problems in the first group, waiting for something. What remains is the second group, to ensure *mutual exclusion*. This means that we need to protect shared data that may be modified concurrently in order to avoid *data races*.

It turns out that semaphores are powerful enough for this task too. In fact, it is possible to implement all other synchronization primitives using only semaphores. Thereby it is possible to solve all synchronization problems using only semaphores if we wish to. However, as we shall see in the next section, semaphores are not always the *most convenient* way to solve all problems. Ensuring *mutual exclusion* is one such area. While it is possible to use semaphores for this task, it is often better to use *locks*. Since locks are intended to ensure mutual exclusion they communicate the intent of the programmer clearer, and they are able to detect some common usage bugs related to how locks are used. Since locks are more suitable for this task, this chapter only introduces the problem and the semaphore-based solution briefly. The topic will be discussed in greater depth in the next chapter.

To illustrate the problem, remember the last program in Section 3.5 where we attempted to add 2 and 5 to a shared variable concurrently. If we solve the waiting problem using semaphores, we get the program in `mutex-1.c` (see Listing 4.10).

```
int result;
struct semaphore has_result;

void thread_fn(void) {
    for (int i = 0; i < 4; i++) {
        result += 2;
    }
    sema_up(&has_result);
}

int main(void) {
    sema_init(&has_result, 0);
    thread_new(&thread_fn);
    for (int i = 0; i < 4; i++) {
        result += 5;
    }
    sema_down(&has_result);
    printf("result=%d\n", result);
    return 0;
}
```

Listing 4.10: The program `mutex-1.c`.

This program is similar to `add-2.c` (Listing 4.4). The only difference is that we have a loop that adds 5 to `result` in `main`, before the call to `sema_down`. Since the loop is *before* `sema_down`, it may execute concurrently with the loop in `thread_fn`, and both threads may therefore modify `result` concurrently. If you use the model checker in Progvis, it will quickly point out the issue.

One way to solve this issue would be to move the `sema_down` call to before the loop in `main`. This would make sure that we execute the loop in `thread_fn` first, and the loop in `main` afterwards, thereby avoiding the data race. However, imagine a situation where the numbers 2 and 5 need to be computed through some expensive computations. This may look like in `mutex-2.c` (Listing 4.11) which simulates expensive computations with a call to `timer_msleep`, which

just sleeps for the specified number of milliseconds.[11]

```
int result;
struct semaphore has_result;

int expensive_computations(int x) {
    // Simulate time-consuming computations.
    timer_msleep(1000);
    return x;
}

void thread_fn(void) {
    for (int i = 0; i < 4; i++) {
        int to_add = expensive_computations(2);
        result += to_add;
    }
    sema_up(&has_result);
}

int main(void) {
    sema_init(&has_result, 0);
    thread_new(&thread_fn);
    for (int i = 0; i < 4; i++) {
        int to_add = expensive_computations(5);
        result += to_add;
    }
    sema_down(&has_result);
    printf("result=%d\n", result);
    return 0;
}
```

Listing 4.11: The program `mutex-2.c`, which simulates time-consuming computations.

In this situation, we can imagine that the point of using an additional thread was to speed up the program. In this case, the program uses two threads to call `expensive_computations` concurrently. This means that the system may use 2 different threads to halve the execution time of the program. Therefore, the solution above (moving the call to `sema_down` to before the loop) is not ideal, since it would make `thread_fn` do all of its work first, then let the main thread do all of its work, thereby eliminating the speedup.

[11] Note that Progvis ignores this call, since you are in control of scheduling anyway.

Practice: Compile and run `mutex-2.c` on your system and measure its execution time. As before, you can compile the program using:

```
make mutex-2
```

To measure the execution time, you can run the program using:

```
time ./mutex-2
```

Measure the execution time of the program as shown in Listing 4.11. Then move the call to `sema_down` to before the loop inside `main` and see how this changes the execution time.

As you likely have seen from above, if you run the program as originally written it will take 4 seconds to run, but if we move the semaphore it will instead take 8 seconds to run.

Because of this, we have a different situation compared to before. Here, we actually want to run the two loops concurrently. However, we need to make sure that the two threads don't touch `result` concurrently. As previously mentioned, we can solve this using semaphores. We just need to view the problem in a different way compared to before. Here, the goal is to make sure that at most one thread accesses `result` at the same time. As such, if we can find a resource related to that property and use a semaphore to count it, we can solve the problem.

In this case, we can choose to count "how many additional threads may access `result`?". So we start by creating a semaphore named `access_result` to count this resource. At the start of the program no thread is accessing `result`, which means that we can allow one additional thread to access it. Therefore we initialize the new semaphore to **1** at the start of `main`.

Since we now count threads that access `result`, we need to look for places in the code that access `result` and update the semaphore accordingly. Before accessing `result`, the code needs to *consume* one instance of the result (i.e. decrease the number of additional threads that may access `result`). As such, we need to add a call to `sema_down` before the two lines `result += to_add`. Similarly, when we are done using `result` we need to *produce* an instance of the resource, since we are done accessing `result`. Therefore, we add a call to `sema_up` after the two lines `result += to_add`. Similarly, we need to add calls to `sema_down` and `sema_up` in the same way since it accesses `result`. These changes results in the code in `mutex-3.c` (Listing 4.12), which solves the issues with shared data.

Practice: Verify that the program is free from data races using the model checker in Progvis. Also verify that the program still executes the two loops in parallel by running the program outside of Progvis and verifying that it finishes in around 4 seconds rather than 8.

As mentioned previously, there are many nuances to the placement of `sema_down` and `sema_up` when working with shared data. We will examine them in further detail in the next chapter, but using locks instead of semaphores.

```
int result;
struct semaphore has_result;
struct semaphore access_result;

int expensive_computations(int x) {
    // Simulate time-consuming computations.
    timer_msleep(1000);
    return x;
}

void thread_fn(void) {
    for (int i = 0; i < 4; i++) {
        int to_add = expensive_computations(2);
        sema_down(&access_result);
        result += to_add;
        sema_up(&access_result);
    }
    sema_up(&has_result);
}

int main(void) {
    sema_init(&has_result, 0);
    sema_init(&access_result, 1);
    thread_new(&thread_fn);
    for (int i = 0; i < 4; i++) {
        int to_add = expensive_computations(5);
        sema_down(&access_result);
        result += to_add;
        sema_up(&access_result);
    }
    sema_down(&has_result);
    sema_down(&access_result);
    printf("result=%d\n", result);
    sema_up(&access_result);
    return 0;
}
```

Listing 4.12: The program `mutex-3.c` which solves the issues in `mutex-1.c` and `mutex-2.c`.

4.7 Exercises

1. In the program `mutex-3.c` we protected the final call to `printf` with calls to `sema_down` and `sema_up`. This is actually not necessary. Use the model checker in Progvis to verify that they are indeed not needed. Why is this the case?

2. What happens if you move the call to `sema_down` inside the two loops one line earlier in the program `mutex-3.c` (i.e. so that it is before the call to `expensive_computations`). Does the program still behave correctly? Does this affect the program's performance?

Chapter 5

Locks

In the previous chapter we saw how semaphores can be used to solve synchronization errors. Even though semaphores are enough to solve all concurrency issues,[1] they are not always the most convenient option. One example is the common problem of achieving *mutual exclusion* for some variables. As we saw in the previous chapter, semaphores are more than capable of doing this, but due to the versatility of semaphores it is easy to make mistakes and the code quickly gets difficult to read.

Since this problem is common, there is a separate synchronization primitive whose sole purpose is to protect variables through mutual exclusion. It is called a *lock* since we can consider it to lock some data so that only one thread may access it concurrently. Since locks are used to achieve mutual exclusion, they are sometimes called *mutexes*.

Since locks are specialized for mutual exclusion, they are able to provide help in the form of better error checking. However, as we shall see, this also means that they are less powerful than semaphores. It is thus not possible to use locks to wait for something to happen. In Chapter 8 we will add this capability to locks by introducing *condition variables* that can be used alongside locks to wait for something to happen.

5.1 Semantics

As in the previous section we start by introducing the available operations and their semantics. As before, we focus on the usage here. The full definitions are available in Chapter A for the interested reader. Again, the `lock` is an opaque data structure, and it is therefore only possible to modify it through the functions below.

A lock itself can be thought of as the lock on the door to a room that contains some data that need to be protected. A thread may then try to *acquire* the lock by entering the room and locking the door from the inside. This means that other threads who also try to acquire the lock needs to wait for their turn. A thread who has previously acquired a lock may then *release* the lock by opening the door and leaving the room. The lock ensures that at most one thread is ever inside the room. We say that the thread that is inside

[1] Unless we work with low-level programming and/or interrupts.

the room is *holding* the lock. One thing that is important to note is that locks are not special in the sense that they automatically protect some variables in the program. We must manually remember to acquire and release the relevant locks for the analogy described above to work.

```
struct lock l;
```

The line above defines a variable named `l` that contains a lock. Note: In contrast to C, Progvis uses the same namespace for types and variables (like C++ does). This means that while it is possible to define a variable named `lock`, it will shadow the type `lock`, which makes it difficult to create other locks later on in the program.

```
lock_init(&l);
```

As with semaphores, each lock variable needs to be initialized exactly once before it is used. The first and only parameter is a pointer to the lock that should be initialized.

```
lock_acquire(&l);
```

Attempt to *acquire* the lock. If the lock is not held by another thread, the calling thread will be allowed to continue and will hold the lock from this point onward. If the lock is already held by another thread, `lock_acquire` will make the calling thread sleep until the thread that is holding the lock releases it. At that point the current thread will finish acquiring the lock.

```
lock_release(&l);
```

Called by a thread that is currently holding the lock to *release* the lock. This may cause another thread that is currently waiting for the lock to wake up and acquire the lock.

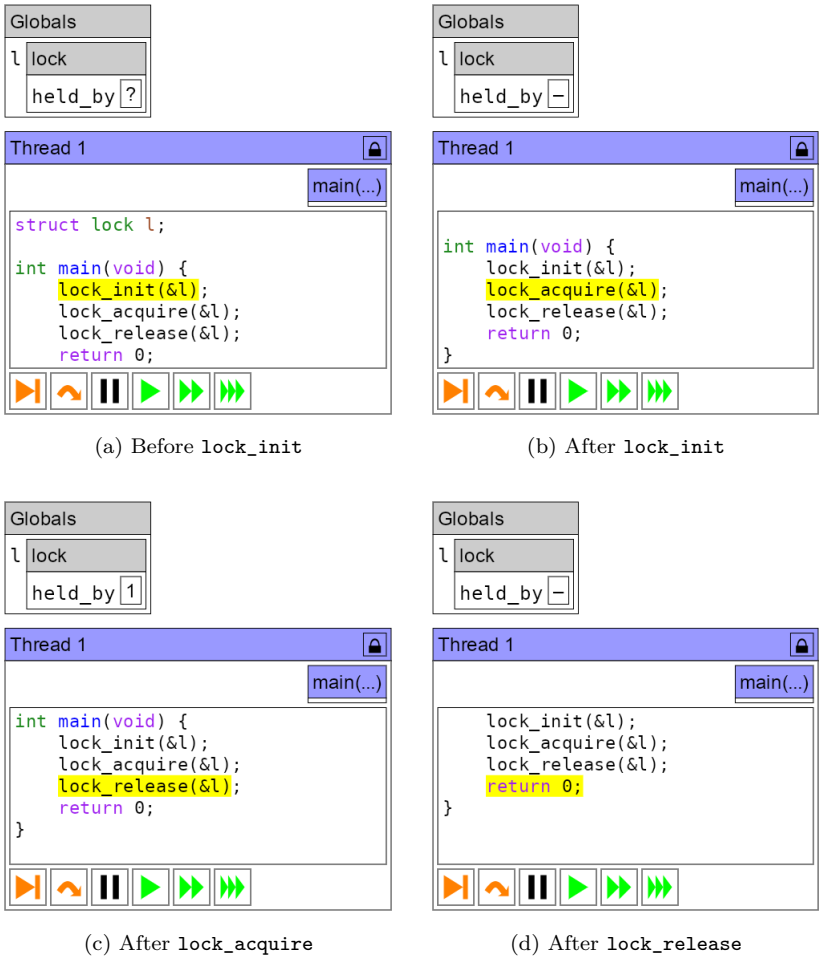
To see how locks are visualized by Progvis, we will use the sequential program `simple-semantics.c`. As we can see from Listing 5.1, the program simply initializes a lock, acquires it and releases it.

```
struct lock l;

int main(void) {
    lock_init(&l);
    lock_acquire(&l);
    lock_release(&l);
    return 0;
}
```

Listing 5.1: A program that illustrates the semantics of the program.

Figure 5.1 shows how Progvis illustrates the state of the lock throughout the program. In the first picture (Fig. 5.1a), we can see that since a lock is a complex data type, it is drawn as a box labeled *lock* to indicate the type of the data. The contains a single line named `held_by` that shows which thread is currently holding the lock. Since the lock is not yet initialized in Fig. 5.1a, the contents are shown as a question mark (?).



(a) Before `lock_init` (b) After `lock_init`

(c) After `lock_acquire` (d) After `lock_release`

Figure 5.1: Progvis' visualization of the program `simple-semantics.c`.

In the next picture (Fig. 5.1b) we have let the program execute `lock_init`. As such, the lock is initialized and ready to be used. However, no thread has yet acquired the lock, so the contents of `held_by` is shown as a dash (-) to reflect this. In the next step (Fig. 5.1c) the main thread has executed `lock_acquire` and thereby acquired the lock. We can see this by observing that the value of `held_by` is now 1. Finally, after the thread executes `lock_release` (in Fig. 5.1c) the contents of `held_by` is a dash again, sine the lock is no longer held by any thread.

5.2 Relation to Semaphores

Since semaphores are powerful enough to solve all synchronization problems, it is possible to implement a lock using a semaphore. While this rarely needs to be done in practice, it is useful to see the relation between the two to understand their semantics in more detail.

Table 5.1: List of how a lock can be implemented using a semaphore.

Lock operation	Semaphore operation
`struct lock l;`	`struct semaphore l;`
`lock_init(&l);`	`sema_init(&l, 1);`
`lock_acquire(&l);`	`sema_down(&l);`
`lock_release(&l);`	`sema_up(&l);`

Table 5.1 shows how each operation for a lock can be replaced by an equivalent operation for a semaphore. This implementation is correct for all programs that use locks correctly. However, the equivalence in Table 5.1 omits the additional error checking performed by locks.

The error checking is illustrated by the program `lock-as-sema.c` (Listing 5.2). This program is similar to the program we used in Section 4.3 to illustrate the semantics of a semaphore. The difference is that the semaphore operations have been replaced with equivalent lock operations according to Table 5.1, as indicated by the comments to the right of the relevant lines. Looking at the corresponding semaphore operations, the only difference from the code used in Section 4.3 is that the initialization is split into two steps. Since there is no operation that corresponds to `sema_init(&l, 0)`, we instead call `sema_init(&l, 1)` to initialize the semaphore to 1, and then call `sema_down` to decrease the counter to zero.

```
struct lock l;              // struct semaphore l;

void thread_fn(void) {
    lock_release(&l);       // sema_up(&l);
}

int main(void) {
    lock_init(&l);          // sema_init(&l, 1);
    lock_acquire(&l);       // sema_down(&l);
    thread_new(&thread_fn);
    lock_acquire(&l);       // sema_down(&l);
    return 0;
}
```

Listing 5.2: Trying to use a lock as a semaphore to wait for `thread_fn`.

Even though the code in Listing 5.2 would work properly if we replaced the lock with a semaphore as indicated by the comments, it does *not* work as it is currently written. This is because the lock assumes that it is protecting some data. The implementations in the library provided with this book and in Progvis both verify this, but other implementations may not give an explicit error and silently behave incorrectly instead.

Illustrates two problems that lock implementations assume, and are often able to verify. Load the program in Progvis and step the threads as described below to see the errors yourself.

1. A lock assumes that the lock is *released* by the thread that currently *holds* the lock (i.e. the thread that previously *acquired* the lock). This helps the programmer to detect cases where they have forgotten to call `lock_acquire` or multiple calls to `lock_release`.

   You can see this error if you step the main thread until it has executed `thread_new`, and then step the newly started thread so that it executes `lock_release`. In Progvis, this causes the thread to crash and Progvis informs you of the problem. The C library behaves similarly. You can cause this problem to occur by adding a call to `timer_msleep(100)` after `thread_new`.

   If a semaphore would be used instead of a lock, this would just increment the counter one additional time, meaning that the semaphore would no longer ensure mutual exclusion.

2. Locks also detect cases where one thread attempts to *acquire* a lock that is already held. The locks used in this book do not allow this to happen and report it as an error. As in the previous point, an implementation may also assume that this does not happen and do something unexpected instead.[2]

   A third option is to implement what is sometimes referred to as *recursive locks* or *recursive mutexes*. In this case, acquiring a lock that is already only causes the lock to require an additional *release* before it is actually released. This is useful in cases where two functions, say `f` and `g` both acquire the same lock, but `f` calls `g` while holding the lock.

   You can see this error if you step the main thread until the end without stepping the other thread. In the C library, you can cause this to happen by adding a call to `timer_msleep` before `lock_release` in `thread_fn`.

   If a semaphore were used instead of a lock in this situation, the extra call to `sema_down` would cause the thread to wait forever, causing a deadlock.

5.3 Using Locks for Mutual Exclusion

Now that we have seen how a lock works and how it relates to a semaphore, we can exemplify this knowledge by replacing the semaphore `access_result` in Listing 4.12 with a lock. If we just use the information in Table 5.1 to replace `sema_init`, `sema_down`, and `sema_up` with `lock_init`, `lock_acquire`, and `lock_release` we get the program in `mutex.c` (Listing 5.3). Note that we also replaced the semaphore variable `access_result` with a lock named `result_lock`.

Examining the code in Listing 5.3, we can now clearly see how locks are designed to protect shared data. Each time the program uses the variable `result`, the program has made sure to *acquire* the associated lock beforehand and to *release* the lock afterwards. This makes it possible to think about locks as if they are marking a region in the source code where the current thread has exclusive access to some variable or resource.[3] While this is often a useful way to think about locks, some care needs to be observed. For example, if the

[2] Often, this causes the thread to wait forever, but anything is allowed to happen.

[3] This idea is used for *scoped locks* in languages like C++, for example.

```
int result;
struct semaphore has_result;
struct lock result_lock;

int expensive_computations(int x) {
    // Simulate time-consuming computations.
    timer_msleep(1000);
    return x;
}

void thread_fn(void) {
    for (int i = 0; i < 4; i++) {
        int to_add = expensive_computations(2);
        lock_acquire(&result_lock);
        result += to_add;
        lock_release(&result_lock);
    }
    sema_up(&has_result);
}

int main(void) {
    sema_init(&has_result, 0);
    lock_init(&result_lock);
    thread_new(&thread_fn);
    for (int i = 0; i < 4; i++) {
        int to_add = expensive_computations(5);
        lock_acquire(&result_lock);
        result += to_add;
        lock_release(&result_lock);
    }
    sema_down(&has_result);
    lock_acquire(&result_lock);
    printf("result=%d\n", result);
    lock_release(&result_lock);
    return 0;
}
```

Listing 5.3: Replacing the semaphore in Listing 4.12 with a lock.

call to `lock_acquire` is inside an `if` statement, the lock is not always acquired and the idea of a syntactical region where it is safe to access some variable falls apart.

In the code, the connection between the `result` variable and the lock is also made clear by naming the lock `result_lock`. Note, however, that there is no formal connection between the `result` variable and the lock `result_lock`. As such, creating a lock with the appropriate name does not automatically protect variable. Since the language does not know which variables we want to protect, we need to protect them explicitly by calling `lock_acquire` and `lock_release`. Similarly, there is no way for the language to verify that we have not forgotten to protect some variable. This is why it is important to be meticulous about documenting which variable(s) the lock is intended to protect (e.g. through

naming), so that it is easy to verify that the lock is actually acquired when necessary.

5.4 Benefits of Error Checking

We have now seen how both semaphores and locks can be used to achieve mutual exclusion and thereby protect shared data from concurrent modification. Even though locks can be implemented using semaphores (as we saw in Table 5.1), they are designed to protect shared data. The benefit of this is that it makes the intent of the programmer clearer. If we see a lock in the code, we know that it protects some shared data. This is not necessarily true if we see a semaphore, since the semaphore might also be used to wait for something to happen in the program. The focus on protecting shared data also means that the lock can check for errors in using the lock. We have already seen some benefits of this in Section 5.2, but in a contrived setting. The remainder of this section will show how the error checking can help catching more realistic issues.

To illustrate this, we will use the `decrease` function in the program `decrease.c` as an example. The function is shown in Listing 5.4. It decreases the global variable `counter` with the value of the parameter `with`, but ensures that the value of `counter` does not become negative. For example, calling `decrease(14)` does nothing since the counter would become negative. Calling `decrease(2)` would subtract 2 from the counter, resulting in it containing 8 after the call.

```
int counter = 10;

void decrease(int with) {
    if (counter < with)
        return;
    counter -= with;
}
```

Listing 5.4: Implementation of the function `decrease` from the program `decrease.c`.

Now assume that we wish to call the `decrease` function from multiple threads concurrently. For this to work properly, we need to protect the `counter` variable somehow. Since the goal is to protect data (i.e. create *mutual exclusion*), we use a lock. However, imagine that we were a bit careless when adding the lock and ended up with the code in `decrease-error-1.c` (Listing 5.5). As we can see, we defined the variable `counter_lock` to protect the `counter` variable, acquire the lock at the start of `decrease` and release it at the end of `decrease`. However, we forgot to account for the fact that `decrease` may return early if it determines that `with` is larger than `counter`, and forgot to release the lock in that case.

```
int counter = 10;
struct lock counter_lock;

void decrease(int with) {
    lock_acquire(&counter_lock);
    if (counter < with)
        return;
    counter -= with;
    lock_release(&counter_lock);
}
```

Listing 5.5: Incorrect application of a lock from `decrease-error-1.c`.

Practice: Assume that a thread calls `decrease(14)` followed by `decrease(2)` using the implementation of `decrease` in Listing 5.5. What happens when `decrease(2)` is called after the call to `decrease(14)` failed to release the lock?

The `main` function in `decrease-error-1.c` calls `decrease(14)` followed by `decrease(2)`. As such, you can see what happens by running the program in Progvis.

What would happen if we use a semaphore instead of a lock to protect `counter`?

As you have noticed from above, since the implementation of `decrease` in Listing 5.5 never releases the lock when `decrease(14)` is called, the lock is already held by the current thread when `decrease(2)` is called later. The lock implementation is able to detect this, since it is always invalid for a single thread to acquire the same lock more than once.[4] If we would have used a semaphore instead, the call to `sema_down` that replaces `lock_acquire` would instead have caused a deadlock that would have been much harder to debug.

[4] Unless *recursive locks/mutexes* are used.

The program `decrease-error-2.c` contains another error that is perhaps more obvious but not uncommon. Here, the programmer has forgotten to acquire the lock at the start of the `decrease` function, and they thereby fail to protect the `counter` variable. While it is easy to spot the issue in this particular case, the same type of error would also arise if the programmer had accidentally acquired some other lock at the start of the function.

```
int counter = 10;
struct lock counter_lock;

void decrease(int with) {
    if (counter < with)
        return;
    counter -= with;
    lock_release(&counter_lock);
}
```

Listing 5.6: Another incorrect lock application of a lock from `decrease-error-2.c`.

Practice: Assume that a thread calls `decrease(2)` using the implementation of `decrease` in Listing 5.6. What happens?

The `main` function in `decrease-error-2.c` calls `decrease(2)`, so you can see what happens using Progvis.

What would happen if we use a semaphore instead of a lock to protect `counter`?

By running the code above, you will see that the call to `lock_release` fails since the lock was not acquired previously. If we would have used a semaphore to protect `counter`, the situation would actually have been worse than in `decrease-error-1.c`. The mistake in `decrease-error-1.c` at least led to a deadlock, which is quite noticeable. On the other hand, the mistake in `decrease-error-2.c` would not only lead to failure to achieve mutual exclusion in the `decrease` function. Since the final call to `sema_up` (which replaced `lock_release`) would increment the semaphore without previously decrementing it, it means that *all other parts* of the code that use the semaphore *correctly* would now *fail to achieve mutual exclusion* since they change the semaphore between 2 and 1 instead of between 1 and 0. As such, if we would have use a semaphore instead of a lock to synchronize `decrease-error-2.c`, the program would most likely have *appeared to work* since failure to synchronize some data is often not immediately visible. However, since the mistake breaks mutual exclusion the program is likely to exhibit strange behavior at some point in the future, which will not be fun to debug. As such, even though the error checking that can be done by locks is far from perfect, the cases it *is* able to detect can save quite a lot of debugging work down the line.

As we have seen in the two previous examples, it is easy to make mistakes when working with locks (or semaphores). Synchronization of shared data is usually one of the most tricky parts to get right, especially since mistakes in

synchronization are not always directly visible in form of incorrect behavior or a crash. Because of this, it is often useful to structure the code so that the synchronization becomes trivial to verify just by reading the code, even if it complicates the program logic to some extent. For example, if we restructure the `decrease` implementation to remove the `return` statement in the middle of the function, we get the code in Listing 5.7 (`decrease-good.c`). Here, it is trivial to verify that we always acquire and release the lock, since the `lock_acquire` and `lock_release` is at the same level of indentation (i.e. one is not in an `if` statement or a loop), and no `return` statements are between them.

```
int counter = 10;
struct lock counter_lock;

void decrease(int with) {
    lock_acquire(&counter_lock);
    if (counter >= with)
        counter -= with;
    lock_release(&counter_lock);
}
```

Listing 5.7: A good way to structure the implementation of the `decrease` function to make it trivial to verify that the synchronization is correct.

In this particular case, one can argue that it makes it easier to read the function as well. This is not always the case, as this kind of rewrite would sometimes require us to add additional variables. At the end of the day, there is a trade-off between how easy it is to follow the program logic and how easy it is to verify that the synchronization is correct. It is usually a good idea to keep the synchronization logic much simpler than the program logic, since it is *much* easier to write unit-tests to verify that the program logic is correct compared to verifying that the synchronization is correct.

5.5 Exercises

1. The program `bug-hunt.c` contains a function `update_counter` that modifies the variable `counter` according to the value of the parameter `delta`. The variable `counter` is protected using a semaphore that is used as a lock (`counter_lock`). The programmer who wrote the `update_counter` function added an "optimization" that checks if `delta` is zero and avoids synchronization in that case.

   The program is, however, not working correctly. While the program *appears* to work correctly, the model checker in Progvis (*Run* ⇒ *Look for errors...*) reports that the program behaves incorrectly. Use Progvis to understand *why* the program is incorrect (even if you see *what* the error is). Then replace the semaphore `counter_lock` with a proper lock and see what error Progvis reports when you run the program (either the model checker or normally). Did the semaphore or the lock make it easier to find the error? What was the error?

Chapter 6

Critical Sections

In Chapters 4 and 5 we have seen how both semaphores and locks can be used to avoid data races by enforcing *mutual exclusion*. That is, we make sure that at most one thread is allowed to execute code that uses some shared data and thereby avoid data races. The piece of code that at most one thread is allowed to execute is what we will call a *critical section*.

It turns out that it is usually not enough to just find all accesses to shared variables and synchronize them individually. In most cases we need to synchronize larger pieces of the code than just a single line. This is where the notion of a *critical section* is useful. It allows us to speak about sections in the code that need to be executed together as a unit, and that must not be interrupted by other code that accesses the same data.

6.1 Why Critical Sections?

To illustrate why it is useful to think in terms of critical sections we will use three versions of the program `increment` (i.e. `increment-1.c`, `increment-2.c`, and `increment-3.c`), which is similar to the program used to illustrate the problems with concurrent execution in Chapter 3. The central parts of the program `increment-1.c` is shown in Listing 6.1.

As can be seen in Listing 6.1, the program is centered around the global variable `result` that is protected with the lock `result_lock`. Two functions, `thread_a` and `thread_b`, both increment the variable 4 times in a loop by 2 and 5 respectively. The full program also contains a `main` function that initializes the lock, starts the threads, waits for them to terminate using a semaphore, and prints the value of `result`. The `main` function is not shown in the figure as it is not central to the discussion about critical sections.

```
const int times = 4;
int result = 0;
struct lock result_lock;

void thread_a(void) {
    for (int i = 0; i < times; i++) {
        lock_acquire(&result_lock);
        result += 2;
        lock_release(&result_lock);
    }
    sema_up(&done);
}

void thread_b(void) {
    for (int i = 0; i < times; i++) {
        lock_acquire(&result_lock);
        result += 5;
        lock_release(&result_lock);
    }
    sema_up(&done);
}
```

Listing 6.1: The important parts of the program `increment-1.c`.

The version `increment-1.c` currently behaves correctly. As we would expect, the variable `result` contains $(2 + 5) \cdot 4 = 28$ at the end of the program. However, remember that even though the statement `result += 2` *looks like* one operation, it is actually executed as (at least) three operations: load the value of `result` from memory, increment it by 2, and store the value back into `result`. We can illustrate this by explicitly rewriting `thread_a` as shown in Listing 6.2 (also in `increment-2.c`).

```
void thread_a(void) {
    for (int i = 0; i < times; i++) {
        lock_acquire(&result_lock);
        int tmp = result;
        tmp += 2;
        result = tmp;
        lock_release(&result_lock);
    }
    sema_up(&done);
}
```

Listing 6.2: Explicitly implementing the separate steps of `result += 2`.

This version of the code also works correctly. After all, the only change we made was to make the steps that the hardware needs to do anyway explicit in the source code. It does, however, highlight that only the first and the third steps actually access the shared variable `result`. Based on this observation, it looks like we only need to hold the lock around the lines that actually access `result`, as shown in Listing 6.3 (also in `increment-3.c`).

```
void thread_a(void) {
    for (int i = 0; i < times; i++) {
        lock_acquire(&result_lock);
        int tmp = result;
        lock_release(&result_lock);
        tmp += 2;
        lock_acquire(&result_lock);
        result = tmp;
        lock_release(&result_lock);
    }
    sema_up(&done);
}
```

Listing 6.3: Adjusting the locks to minimize the time where mutual exclusion is enforced.

The implementation in Listing 6.3 is interesting. Since the implementation makes sure to hold `result_lock` whenever `result` is accessed, the lock ensures that the two threads never access it concurrently, and therefore we have eliminated all possibilities of *data races* in the code. The program is therefore well-defined according to the C standard. However, the program is still not correct in the sense that it behaves the way we expect it to behave.

Practice: Load the program `increment-3.c` in Progvis, and use *Run* ⇒ *Look for errors...* to ask Progvis to find any errors. Does Progvis find any errors? Why/why not?

The end of the `main` function in `increment-3.c` contains a commented-out line that contains: `assert(result == times * (2 + 5));` Uncomment the line and select *Run* ⇒ *Look for errors...* again. What happens now?

As you will have noticed, Progvis will *not* find any errors in the code. This is because, as we noted earlier, the program neither crashes nor contains any data races. It is therefore a correct program according to the C standard. However, we as programmers have additional expectations about the behavior of the program. In particular, we expect that none of the additions should be "forgotten", which is why we consider the program to be incorrect. If we let Progvis know about our expectations, for example by adding an `assert` statement at the end of the `main` function, it will find the error for us since a failed `assert` statement causes the program to crash.

This example illustrates an important point. It is usually not enough to simply surround each access to shared data with `lock_acquire` and `lock_release`. Doing so does indeed eliminate data races and thereby avoids undefined behavior. However, it is usually not enough to make the program behave as we expect it to. In most cases, the program updates shared data in multiple steps, and it is important that other threads are neither able to observe nor interfere with the shared data before all steps are completed.

This is exactly the problem in Listing 6.3. The code illustrates that the `+=` operator is really executed as (at least) three steps: (1) read from memory,

(2) increment the value, and (3) write to memory. If another thread is able to observe and/or interfere with the process before it is done, the result from step (3) in one thread might be overwritten by step (3) of another another thread that read an old version of the shared data in step (1). The solution, as we have seen, is to use locks to ensure that once one thread starts step (1), all other threads have to wait until the thread is done with step (3) before accessing the shared data. This ensures that at most one thread is working on any of steps (1)–(3), and thereby that no thread is able to observe or interfere during the steps. As such, we would think of these steps as being a single critical section.

6.2 Critical Sections and Conditionals

The previous section illustrated that shared data is often updated in multiple steps, even though it might not seem to be the case initially. As such, it is important to properly determine the critical section where shared data is updated so that we can protect it with locks. It is not uncommon for updates to shared data to include conditionals (e.g. `if` statements or `for` loops) as well. In this section we will therefore return to the `decrease` program from the previous chapter to illustrate how conditionals interact with critical sections. We start with the program `decrease-1.c`, which is the same as the program we synchronized at the end of the last chapter.

As can be seen in Listing 6.4, the program contains a global variable `counter` that is protected by the `counter_lock`. The function `decrease` decreases `counter` by a specified amount after checking that the counter is large enough to not become negative when decreased (not too dissimilar from `sema_down`). To ensure that `decrease` behaves correctly when called concurrently, the function acquires `counter_lock` at the start of the function and releases it at the end of the function.

```
int counter = 10;
struct lock counter_lock;

void decrease(int with) {
    lock_acquire(&counter_lock);
    if (counter >= with)
        counter -= with;
    lock_release(&counter_lock);
}
```

Listing 6.4: The `counter` variable and `decrease` function from `decrease-1.c`.

The program also contains code that starts two threads that both call `decrease(6)`. Since `counter` starts at 10 and we call `decrease(6)` two times, one of the calls (whichever happens to acquire the lock first) decreases the counter to 4 while the other does nothing. As such, `counter` contains 4 at the end of the program.

The crucial question that remains is, how do we arrive at the solution in Listing 6.4? The first observation is, of course, that the variable `counter` is shared and all accesses to it needs to be protected. The question that remains is how we arrive at the proper size of the critical section that we need to

protect with locks. That is, do the two lines that access `counter` need to be in a single large critical section, or is it enough to have two different critical sections? The key observation here is that the line `counter -= with` assumes that `counter >= with` (i.e. the condition in the `if` statement) is still true. As such, there is a dependency between the `if` statement and decrementing `counter`.

```
void decrease(int with) {
    lock_acquire(&counter_lock);
    bool ok = counter >= with;
    if (ok) counter -= with;
    lock_release(&counter_lock);
}
```

Listing 6.5: Rewritten `decrease` function as found in `decrease-2.c`.

We can clarify this dependency by rewriting the `decrease` function as in Listing 6.5 (`decrease-2.c`). Here, we store the result of the condition in the boolean variable `ok`, and use it on the next line. The if-statement on the next line is written on a single line to illustrate that we think of it as a single unit rather than altering the control flow of the program. That is, we think of the line as "decrease `counter` if `ok` is `true`"[1] rather than "if `ok` is `false`, skip the body of the `if` statement". Since the variable `ok` is an "input" to the line, the dependency between the two lines is now quite apparent.

```
void decrease(int with) {
    lock_acquire(&counter_lock);
    bool ok = counter >= with;
    lock_release(&counter_lock);

    lock_acquire(&counter_lock);
    if (ok) counter -= with;
    lock_release(&counter_lock);
}
```

Listing 6.6: Rewritten `decrease` function as found in `decrease-3.c`.

This way of expressing `decrease` also makes it easier to see what happens if we separate the two accesses to `counter` into separate critical sections. To do this, we can simply release the lock and re-acquire it between the lines, as is done in Listing 6.6 (`decrease-3.c`).

Practice: Use the *Run ⇒ Look for errors...* option in Progvis to verify that the program `decrease-3.c` is incorrect, and why.

As you will have seen above, dividing the critical section into two causes the program to misbehave in a way that is similar to the behavior we saw in the

[1] This is actually not far-fetched. Many CPUs have an instruction called *conditional move* that can be used to implement this line without altering the control flow. If you enable optimizations, it is likely that your compiler uses such an instruction if your CPU has one.

previous chapter before we added synchronization. Even though they fail in a similar way, there is a very important difference. The non-synchronized version is undefined according to the C language, which means that anything may happen.[2] The code in Listing 6.6 on the other hand is well-defined since it avoids data races.

The issue in Listing 6.6 is that the line in the first critical section (`ok = counter >= width`) observes the value of `counter` and makes a decision based on the observation. In this case, the decision is if `counter` is large enough so that it can be decreased without becoming negative. The line in the next critical section (`if (ok) counter -= with`) then uses the value in `ok` to act on the decision. Note that the program does *not* verify that `counter` is still large enough, but blindly trusts the value in `ok` and thereby assumes that all is well. For the program in Listing 6.6, this is *not* true since it releases the lock between the two statements. Once the lock is released, other threads that operate on `counter` (e.g. another thread that calls `decrease()`) are able to acquire the thread and modify `counter`. This is why the value of `ok` may be stale when the lock is re-acquired.

This illustrates that we once again have code that updates shared data in multiple steps. In this case, the first step is that we inspect shared data to make a decision, and then act on the decision to possibly update a shared variable (which we have previously seen is multiple steps). Since it is important that the data we used when making a decision is the same as when we act on the decision, the two need to be a part of the *same* critical section. Otherwise, other threads may change the data in a way that invalidates the decision.

Finally, it is worth noting that while the rewrite in Listing 6.5 makes the dependency between the decision and the update explicit through the `ok` variable, it does *not* affect the behavior of the program. It is possible to split the critical section into two parts if the code is written as in Listing 6.4 as well, but it is less clear. It would look like Listing 6.7.

```
void decrease(int with) {
    lock_acquire(&counter_lock);
    if (counter >= with) {
        lock_release(&counter_lock);

        lock_acquire(&counter_lock);
        counter -= with;
    }
    lock_release(&counter_lock);
}
```

Listing 6.7: Split critical section without introducing additional variables. Note that the lock is no longer acquired and released at the same level of indentation, which makes it harder to determine where critical sections start and end.

[2] In particular, even if the program works for one version of a compiler, it may stop working the day you receive a small update to your compiler, or even when you change some seemingly unrelated code.

6.3 Shared Data in Multiple Functions

A critical section is not necessarily tied to a single function. If multiple functions in the same program access the same shared data, they need to use the same lock to protect the shared data. As such, the critical sections in the functions need to use the same lock since they operate on the same shared data.

```
int counter = 10;
struct lock counter_lock;

void decrease(int with) {
    lock_acquire(&counter_lock);
    if (counter >= with)
        counter -= with;
    lock_release(&counter_lock);
}

void increase(int with) {
    counter += with;
}
```

Listing 6.8: Extension of the `decrease` program to also include an `increase` function.

To illustrate this idea, we add an `increase` function to the `decrease` program we used in the previous section. It simply increases `counter` with a given value as shown in Listing 6.8. The full program `updown-1.c` also includes a `main` function that starts two threads that each call `decrease(6)` as before. It additionally calls `increase(1)` from the main thread. As such, we expect the `counter` to contain 5 at the end of the program as indicated by the `assert` at the end of the `main` program.

Practice: The program `updown-1.c` is not correct. In particular it contains a data race, even though it uses `counter_lock` to protect the shared variable `counter`. Use Progvis to find the data race.

As you have likely already seen, the issue with the program `updown-1.c` (Listing 6.8) is that the `increase` function does *not* use `counter_lock` to protect `counter`. As such, it is possible for one thread to run `increase(1)` concurrently with another thread that runs `decrease(6)`. Even though the thread running `decrease(6)` acquires the lock, the thread running `increase(1)` is allowed to continue and modify `counter` as it wishes since it does not attempt to acquire `counter_lock`.

One way to view this problem is that the `increase` function violates an assumption in the code. In this case, we assume that all threads that modify `counter` have first successfully acquired the `counter_lock` lock. This assumption is *not* validated by the programming language, which is why we will not get any errors or warnings when we compile `updown-1.c`. The assumption is, however, important to uphold. As we saw in the previous section, the

`decrease` function utilizes the assumption to ensure that no other threads modify `counter` after it has decided to decrease `counter`, but before it has actually had time to do so.[3] Additionally, we use the convention to ensure that we avoid data races when accessing `counter`. Since we fail to do so, the program currently contains a data race and is therefore undefined.

To address the issue, we need to modify the `increase` function to follow the convention of holding `counter_lock` whenever `counter` is accessed. By doing that, we properly protect `counter` with `counter_lock` throughout the program, thereby avoiding data races. In this case, we can simply modify the function as in Listing 6.8 (`updown-2.c`).

```
void increase(int with) {
    lock_acquire(&counter_lock);
    counter += with;
    lock_acquire(&counter_lock);
}
```

Listing 6.9: Updated version of the `increase` function to properly protect access to the `counter` variable.

Practice: The program `updown-3.c` (below) contains an alternative solution that uses two locks, one for `decrease` and one for `increase`. This does *not* work as intended, even though it may look like all code that uses `counter` is protected with locks. Use Progvis to find out why the program is incorrect.

```
int counter = 10;
struct lock decrease_lock;
struct lock increase_lock;

void decrease(int with) {
    lock_acquire(&decrease_lock);
    if (counter >= with)
        counter -= with;
    lock_release(&decrease_lock);
}

void increase(int with) {
    lock_acquire(&increase_lock);
    counter += with;
    lock_acquire(&increase_lock);
}
```

[3] In this particular example, increasing `counter` will never invalidate the decision made by `decrease`. However, we still have the issue that neither `-=` nor `+=` are executed as a single step, which may cause the program to behave incorrectly.

6.4 Synchronization Granularity

Up until now we have only looked at critical sections that involve a single shared variable. In real programs, critical sections are likely to involve multiple variables that together represent some state that needs to be kept consistent. As we shall see, there are different ways to synchronize such programs, each with different trade-offs. We will also see that the extent of critical sections may change based on requirements from other parts of the code. The end-goal of this section is therefore to show that it is important to consider the program as a whole when determining how it needs to be synchronized. In other words, the best way to synchronize a program often depends on how the program is structured at a larger scope than individual lines of code or even individual functions. Therefore it is important to look at the big picture before diving into details.

To illustrate the intricacies involved with synchronization of larger programs we will use different versions of the program `bank`. The first version, `bank-1.c`, is not synchronized at all and acts as the starting point for our exploration. The central parts of `bank-1.c` is shown in Listing 6.10.

```
struct account {
    int balance;
};

int accounts_count;
struct account *accounts;

bool transfer(int amount, int from, int to) {
    struct account *f = &accounts[from];
    struct account *t = &accounts[to];

    if (f->balance >= amount) {
        f->balance -= amount;
        t->balance += amount;
        return true;
    } else {
        return false;
    }
}
```

Listing 6.10: Initial, unsynchronized version of the `bank` program.

As indicated by the name, the program manages a simple fictional bank. The bank consists of a number of accounts, each represented by an instance of `struct account`. For the purposes of this example, we are only concerned about the current balance in each account (i.e. the amount of money currently in the account). All accounts are stored in the global array `accounts`. The `accounts` variable is defined as a pointer since the actual array is allocated dynamically by the `create_accounts` function (not in the figure). The variable `accounts_count` stores the number of elements in the `accounts` array. Since the array is allocated dynamically, it is possible to change the size of the array at a later point. For the time being we do, however, assume that the array is created

once and then never resized.

As shown in Listing 6.10, the program also contains a function named `transfer` that transfers money between two accounts within the bank. The function first checks that account `from` has a sufficient balance for the transfer.[4] If the balance is large enough, it then subtracts `amount` money from the balance of account `from`, and adds the corresponding balance to account `to`.

In addition to the code in Listing 6.10, the program also contains a `main` function that creates two accounts with a balance of 10. The `main` function then starts two threads that both transfer 8 units of currency from account 0 to account 1. Finally, `main` waits for both threads to complete and asserts that the balance of account 0 is nonnegative.

Practice: The program `bank-1.c` is currently not correct. First and foremost, it contains data races since access to shared data is not protected. Additionally, it is possible for the balance of account 0 to become negative.

Find the problem in the code that causes these issues (either by reasoning about the code, or by using Progvis). Then, suggest one or more critical sections in the `transfer` function that needs to be protected in order to solve the issues.

As you most likely concluded from above, the issue is similar to the issue in the `decrease` program. The `transfer` function contains an if-statement that first verifies a condition (that the balance of account 0 is large enough) and then acts on it (decreasing the balance of account 0). However, since we have not protected access to `f->balance`, other threads might have invalidated the condition that we have verified. In the case of the program above, both threads may first observe that the balance of account 0 is larger than 8, and decide to continue transferring funds. As such, both threads will decrease the balance of account 0 with 8, so that it now contains the value -6, which is not as intended.

```
bool transfer(int amount, int from, int to) {
    struct account *f = &accounts[from];
    struct account *t = &accounts[to];

    bool ok = f->balance >= amount;
    if (ok)
        f->balance -= amount;
    if (ok)
        t->balance += amount;
    return ok;
}
```

Listing 6.11: Updated version of the `transfer` function to make the dependency explicit. Available as `bank-2.c`.

As before, we can rewrite the logic in the transfer function to make the fact that adjusting the balances of both accounts depend on the check of

[4] The bank does not provide services like lending money.

`f->balance`. Initially, checking `ok` twice might seem superfluous. This does, however, become relevant when we think more carefully about the extends of the critical sections in the function.

Practice: At this point we are ready to protect the shared data in the `accounts` array with a lock called `accounts_lock`. Below are three options of possible lock placements, `bank-3-a.c`, `bank-3-b.c`, and `bank-3-c.c`. Reason about the code or use Progvis to determine which of the options most accurately protects the critical section in the `transfer` function.

`bank-3-a.c`

```
bool transfer(int amount, int from, int to) {
    lock_acquire(&accounts_lock);
    struct account *f = &accounts[from];
    struct account *t = &accounts[to];
    lock_release(&accounts_lock);

    bool ok = f->balance >= amount;
    if (ok)
        f->balance -= amount;
    if (ok)
        t->balance += amount;
    return ok;
}
```

`bank-3-b.c`

```
bool transfer(int amount, int from, int to) {
    lock_acquire(&accounts_lock);
    struct account *f = &accounts[from];
    struct account *t = &accounts[to];

    bool ok = f->balance >= amount;
    if (ok)
        f->balance -= amount;
    if (ok)
        t->balance += amount;
    lock_release(&accounts_lock);
    return ok;
}
```

`bank-3-c.c`

```
bool transfer(int amount, int from, int to) {
    struct account *f = &accounts[from];
    struct account *t = &accounts[to];

    lock_acquire(&accounts_lock);
    bool ok = f->balance >= amount;
    if (ok)
        f->balance -= amount;
    if (ok)
        t->balance += amount;
    lock_release(&accounts_lock);
    return ok;
}
```

As you have likely realized, the program in `bank-3-a.c` is incorrect. Even though the data protected by the lock `accounts_lock` is in the `accounts` array, it is not enough to just protect the lines in the code that mention `accounts`. In this case, the lines protected in `bank-3-a.c` compute pointers to the data in `accounts`, and the code that actually modify shared data appears later on in the function. As we discussed above, the relevant part is the code that inspects and modifies the `balance` member of the individual accounts.

As such, we can conclude that `bank-3-b.c` works better. This is true, `bank-3-b.c` does indeed behave correctly. Since we acquire the lock before we touch the `accounts` variable or modify any other accounts, and release it after we have finished all modifications to the `balance` of all accounts, it is not possible for one thread that executes `transfer` to interrupt another thread that also executes `transfer`. As such, the issues we found in `bank-1.c` and `bank-2.c` can not happen.

The remaining question is, what about `bank-3-c.c`? The difference between `bank-3-b.c` and `bank-3-c.c` is that the latter does *not* include the lines below in the critical section that is protected by the lock:

```
struct account *f = &accounts[from];
struct account *t = &accounts[to];
```

Perhaps surprisingly, these lines do *not* need to be included in the critical section. To understand why, we need to think about what the lines above represents and what `account_lock` is actually protecting.

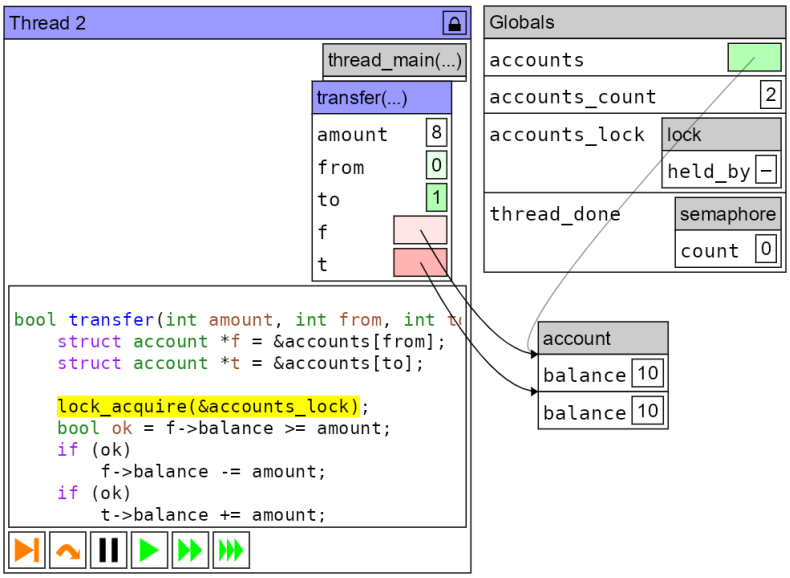


Figure 6.1: Progvis illustration of how the lines not protected in `bank-3-c.c` affects memory.

As a starting point, we can see how Progvis illustrates the memory accesses performed by the two lines. We can do this by opening the program `bank-3-c.c`

and stepping the program until thread 2 is started and reaches the point in Fig. 6.1. As usual, Progvis colors memory locations the program has read from in green, and locations the program has written to in red. The strong colors represent locations accessed by the last statement, and more faded ones represent locations accessed since the previous synchronization operation (e.g. starting threads, acquiring or releasing locks, etc.). In particular, note that `accounts` is colored green, both `t` and `f` are colored red, but the two accounts' `balance` is *not* colored at all. What this means is that the two lines *did not* access the balances[5] of *any* of the accounts. Since the lock `account_lock` is used to protect the *contents* of the array, this is our first hint that `bank-3-c.c` is indeed correct.

To understand *why* the lines do not access the contents of the array, we need to consider what the lines above actually mean. Remember that arrays are both represented as and handled through pointers (see Section 2.6). As such, the syntax used above is just syntactic sugar for the following:

```
struct account *f = accounts + from;
struct account *t = accounts + to;
```

That is, each line reads the pointer stored in the variable `accounts` and increments it with the value that is stored inside `from` and `to` multiplied with the size of `struct account`. Remember that the underlying representation of a pointer is just an integer value, the special semantics of pointers is something that C adds to help us remember what type of data the pointer refers to. From the rewrite above, it hopefully becomes clear why these lines do not access the contents of the `accounts` array. It is, however, important to remember that the code still reads from the `accounts` variable itself (i.e. the pointer, not the contents of the array). As such, the reason we do not have to include these lines in the critical section is that we assume that `accounts` is initialized once and then never changes.

It is worth noting that this phenomenon is not unique to arrays. The same thing happens when we use the address of operator (`&`) to compute the address of other things as well. For example, if we wished to compute the pointer to the `balance` variable in the `from` account, we could write: `int *f_balance = &f->balance;`. Just as when we computed the address to the `from` account, this line only needs to read the variable `f`. It does not touch the `balance` in the account that `f` points to, since we requested the address of the variable rather than the integer stored there. As you can see, correctly reasoning about what parts of a function that needs to be included in a critical section requires a good understanding of the semantics of the programming language we use. This is one reason why Progvis emphasizes an accurate and detailed representation of the memory and how the code interacts with the memory!

The conclusion is that the program `bank-3-c.c` is correct since the `account_lock` only protects the *contents* of all accounts, and the lines that compute the address to the `from` and `to` accounts do not access any data of any account. Furthermore, the program `bank-3-c.c` is slightly better than `bank-3-b.c`, since it does not disallow two threads from computing the address to their respective accounts concurrently. Even though the difference is negligible in this case, it is good practice to find the minimal critical section and only protect that

[5] Or any other part for that matter.

using a lock. When learning concurrency and synchronization, this mindset challenges us to refine our understanding of the program we synchronize and learn more nuances.

Furthermore, in "real" multithreaded programs, the main bottleneck is often the critical sections. Since only one thread at a time is allowed to execute code in a critical section, the performance of critical sections will *not* scale with the number of available CPU cores. As such, there are tangible benefits to structuring programs so that critical sections can be minimized or avoided altogether.

6.4.1 Improved Parallelism

Our current implementation of the bank program (e.g. `bank-3-c.c`) uses a single lock to protect all accounts, and thereby only allows *a single* thread to transfer money between accounts. This is overly restrictive, since there would be no issues allowing one thread to transfer money from account 0 to account 1 while another thread transfers money from account 1 to account 2.

Practice: The file `bank-4.c` contains a copy of `bank-3-c.c`, but with a different main program. This program starts two threads as described above. One thread transfers money from account 0 to account 1, while the other thread transfers money from account 2 to account 3. Open the program in Progvis and observe the behavior of the program:

1. The way the program is currently written, note that the two threads need to wait for each other since both acquire `accounts_lock` inside the `transfer` function.
2. What happens if we remove the lock? From before, we know that this causes issues when two or more threads operate on the *same* accounts concurrently. However, the main program here always operates on *different* accounts. Use *Run* ⇒ *Look for errors...* to see if the removal of the lock causes any issues in this particular case.

As you have likely noticed from above, using Progvis we can see the problem in `bank-4.c`. In Fig. 6.2, we can see that the thread labeled Thread 2 has acquired `accounts_lock` and has started to transfer money from account 0 to account 1. We can also see that the thread labeled Thread 3 also tried to acquire `accounts_lock`, and therefore has to wait until Thread 2 is done. This happens even though Thread 3 is trying to transfer money between different accounts (from 2 to 3, as shown by the arrows).

The reason for this is that we have been too coarse when determining our critical sections. In Section 6.4, we treated the `accounts` array as one large unit. This was initially intuitive since the data we needed to protect was stored in that variable. However, as we saw above, this gives us an implementation that seems to impose stricter restrictions than the program actually needs to fulfill its requirements. As we shall see, the synchronization will depend on our requirements of the system as a whole. Therefore it is a good idea to start by clarifying our requirements of the system. So far, we only need to focus

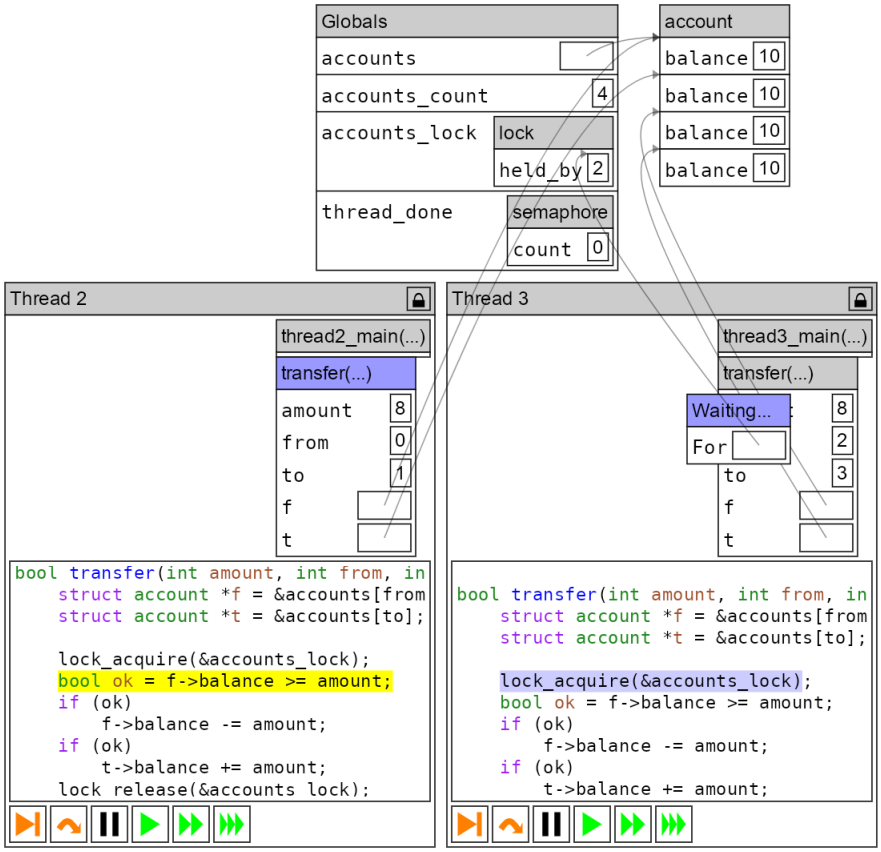


Figure 6.2: Even though the two threads are trying to operate on different accounts, the implementation in `bank-4.c` requires the threads to wait for each other.

on the `transfer` function (`create_accounts` is only called once to initialize everything). Our requirements are that `transfer` should...

1. ...ensure that the balance of the `from` account remains non-negative by refusing to transfer money if that would be the case.
2. ...ensure that a transfer from some account should not be overwritten by another transfer from the same account.
3. ...ensure that a transfer to some account should not be overwritten by another transfer to the same account.

From our analysis before we noted that requirements 1 and 2 above are linked, since the check for a sufficient balance needs to be performed in the same critical section as the removal of the funds. Furthermore, we can observe that requirements 1 and 2 are related to the `from` account, while requirement 3 is related to the `to` account. Since it is always acceptable to *add* money to an account in our system (there is no maximum balance, for example), this requirement can be fulfilled separately from the other two.

Since none of the requirements speak about more than one account at a time, we can conclude that there is no need for us to guarantee a consistent state for the *entire array* of account. Rather, it is enough for us to treat *individual accounts* separately, and ensure consistency for individual accounts in isolation. Since the program `bank-4.c` uses a single lock to protect the entire account, we ensure that only one thread accesses the entire array at any one time, and therefore we can guarantee a consistent state for the entire array at the cost of not being able to perform multiple transfers in parallel. However, we have concluded that we do not need that property, and we would instead like to trade some consistency with increased parallelism.

To achieve our goal, we need to re-consider our lock placement. A good starting point is to declare the lock together with the data that should be protected. In this case, our goal is to protect the `balance` of individual accounts. Therefore, we remove `accounts_lock` and instead add a lock inside `struct account` that we call `balance_lock` to remind ourselves that it protects `balance`, as shown below:

```
struct account {
    int balance;
    struct lock balance_lock;
};

int accounts_count;
struct account *accounts;
```

Now that we have determined at what level we wish to synchronize the data (i.e. one lock for each account rather than one lock for *all* accounts), we need to re-visit the implementation of `transfer` to find the critical sections that need to be protected. To do this, we will start over from the implementation in `bank-2.c` below:

```
1  bool transfer(int amount, int from, int to) {
2      struct account *f = &accounts[from];
3      struct account *t = &accounts[to];
4
5      bool ok = f->balance >= amount;
6      if (ok)
7          f->balance -= amount;
8      if (ok)
9          t->balance += amount;
10     return ok;
11 }
```

We previously concluded that lines 2 and 3 do not need to be a part of any critical section since we are just computing the address of the accounts that will be used in the remainder of the function. Furthermore, lines 5–7 need to be a part of the same critical section, since line 5 checks that the balance of the `from` account is sufficient and line 7 acts on that decision. These lines address requirements 1 and 2, and are all related to the balance in the `from` account. As such, this critical section needs to be protected by the `balance_lock` in the `from` account.

The only other part of the function that accesses shared data is line 9. Line 9 needs to be protected by a lock to fulfill requirement 3. However, even though

line 9 is only executed if `ok` is `true`, it is not important that line 9 is a part of the *same* critical section as the modifications to the `from` account. This is because while we only wish to add money to the `to` account if we removed money from the `from` account, adding money to the `to` account can always be done and does not require that the state of either account remains in the state it was previously. As such, line 9 is a separate critical section that needs to be protected by the `to` account's `balance_lock`.

Now that we have determined the critical sections and what locks need to be used to protect them, we can simply add calls to `lock_acquire` and `lock_release` to protect the critical section. This ends up looking like the code in `bank-5.c`, also depicted in Listing 6.12.

```
bool transfer(int amount, int from, int to) {
    struct account *f = &accounts[from];
    struct account *t = &accounts[to];

    lock_acquire(&f->balance_lock);
    bool ok = f->balance >= amount;
    if (ok)
        f->balance -= amount;
    lock_release(&f->balance_lock);

    if (ok) {
        lock_acquire(&t->balance_lock);
        t->balance += amount;
        lock_release(&t->balance_lock);
    }
    return ok;
}
```

Listing 6.12: The implementation of `transfer` in `bank-5.c`.

Practice: Use Progvis to verify that the proposed synchronization in `bank-5.c` is indeed correct. The main program in `bank-5.c` is a combination of the two previous ones. It creates three threads in addition to the main thread. Thread 2 transfers money from account 0 to account 1, Thread 3 transfers money from account 0 to account 2, and Thread 4 transfers money from account 2 to account 3. In particular, verify that:

- Transfers between different pairs of accounts can occur without having to wait for each other (i.e. Thread 2 never has to wait for Thread 4, but Thread 3 may cause both to wait).
- Transfers do not cause concurrency errors (e.g. by using *Run* ⇒ *Look for errors...*).

From above (and Listing 6.12) we can make an additional important observation. Since both the `from` and `to` parts of the code uses `balance_lock` (for different accounts), it is possible that the critical section for the `from` account

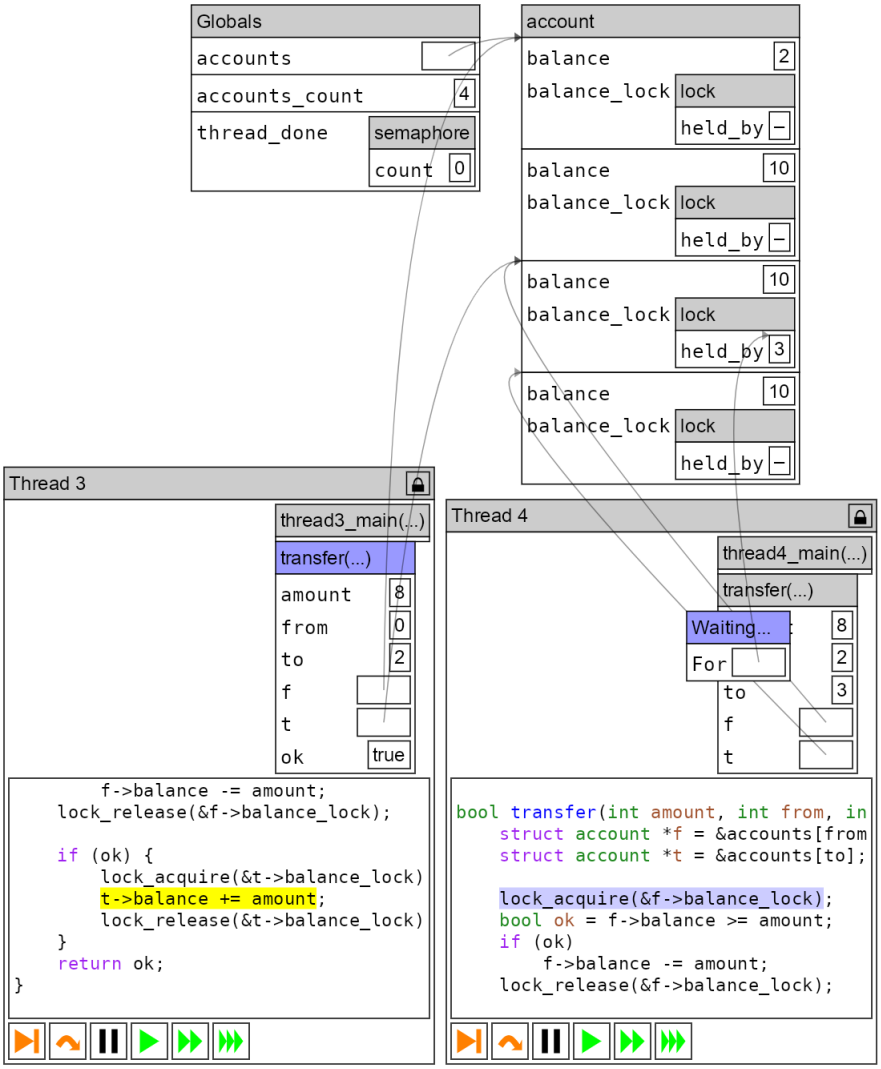


Figure 6.3: Using the same lock for different critical sections means that a thread executing one part of the code blocks another thread that wishes to execute another part of the code.

interferes with the critical section for the `to` account in another thread. For example, when Thread 4 acquires the lock before reading the balance of the `from` account, it may need to wait for Thread 3 to update its `to` account as depicted in Fig. 6.3.

Practice: We can avoid one critical section blocking another critical section by using two locks for each account, as is done in `bank-5-incorrect.c` (depicted below). As the name of the program implies this is, however, not correct. Why? You can use *Run ⇒ Look for errors...* in Progvis to find an example of the issue.

```
struct account {
    int balance;
    struct lock balance_from_lock;
    struct lock balance_to_lock;
};

int accounts_count;
struct account *accounts;

bool transfer(int amount, int from, int to) {
    struct account *f = &accounts[from];
    struct account *t = &accounts[to];

    lock_acquire(&f->balance_from_lock);
    bool ok = f->balance >= amount;
    if (ok)
        f->balance -= amount;
    lock_release(&f->balance_from_lock);

    if (ok) {
        lock_acquire(&t->balance_to_lock);
        t->balance += amount;
        lock_release(&t->balance_to_lock);
    }
    return ok;
}
```

As you have likely noted from above, the behavior illustrated in Fig. 6.3 is exactly what we want. Since both critical sections may access `balance` in the same account, we want to protect the shared data to avoid data races. Otherwise it is possible for one thread to add funds to an account at the same time as another thread removes funds. Just as we have seen when two or more threads execute `+=` concurrently, it is possible that one of the operations will be overwritten by another thread and we thereby either create or destroy money from the bank.

This example illustrates a good rule of thumb when synchronizing code: it is usually a good idea to place the synchronization close to the variable or variables that need to be protected. As we have seen from this example, this means that we sometimes need to think about at what granularity we operate on data. For example, we found it overly restrictive to treat the entire array as a big unit, and ended up protecting accounts individually. This is, however, a rule of thumb. Sometimes it is desirable to lump data together and synchronize them as a bigger unit. One way to find such cases is to start by synchronizing individual variables and then pay attention to which locks always need to be acquired together. Locks that always need to be acquired

together can then be merged into a single lock that protects multiple variables without any performance loss.

6.4.2 Extending the Functionality of the Bank

As we noted in the previous section, the extent of the critical sections we need to protect in the program depends on the requirements of the program *as a whole*. As such, if the requirements for the program changes, we might need to revisit parts of the code we have synchronized earlier.

To illustrate this, the program `bank-sum-1.c` adds a new function `accounts_total` to the bank program. The function computes the total balance of all accounts in the bank. The initial version of the implementation is not synchronized at all, as shown in Listing 6.13.

```
int accounts_total(void) {
    int total = 0;
    for (int i = 0; i < accounts_count; i++) {
        total += accounts[i].balance;
    }
    return total;
}
```

Listing 6.13: The initial version of the function `accounts_total`.

Since the function accesses the `balance` of all accounts in the bank, we need to protect the access with the `balance_lock` as before. Since the function only reads each account's `balance` once in sequence, it appears to be enough to treat the body of the loop as a critical section that should be protected by the current account's `balance_lock`. As such, we add `lock_acquire` and `lock_release` to the function as shown in Listing 6.14 and `bank-sum-2.c`.

```
int accounts_total(void) {
    int total = 0;
    for (int i = 0; i < accounts_count; i++) {
        lock_acquire(&accounts[i].balance_lock);
        total += accounts[i].balance;
        lock_release(&accounts[i].balance_lock);
    }
    return total;
}
```

Listing 6.14: First attempt at synchronizing the `accounts_total` function in `bank-sum-2.c`.

This approach does indeed make sure that no data races are possible. Since we have made sure to hold `balance_lock` every time we access the `balance` of an account, the lock ensures that two threads are not able to access `balance` of the same account at the same time. This is good, since it makes our program well-defined according to the C standard. However, as we have seen before, all well-defined programs are not necessarily deterministic, nor do they always behave as we want them to. This is the case for `bank-sum-2.c`.

Practice: The main program in `bank-sum-2.c` illustrates a situation where the behavior of the program differs from our expectations. Since we only allow transferring money within the bank, we would expect `accounts_total` to always return the same value once the bank is initialized.

To verify this, the main program in `bank-sum-2.c` initializes a bank with 2 accounts with a balance of 10. It then creates two threads that transfer 5 money from account 0 to account 1. While the threads running, the main thread calls `accounts_total` and verifies that the total is 20.

Use Progvis to find a situation where `accounts_total` does *not* return 20. You can use *Run* ⇒ *Look for errors...* to do this. As the program is written, `accounts_total` may return either 10, 15, 20, 25, or 30. Can you find interleavings that cause this to happen? (Hint, with a slight modification to the `assert` statement, you can use *Run* ⇒ *Look for errors...* to find the interleavings for you!)

As you have likely noticed from above, there are two main problems with the program `bank-sum-2`. They are illustrated in Figs. 6.4 and 6.5. Note, that they only illustrate the key insights in the interleavings that cause `accounts_total` to return 10 and 30. The interleavings where `accounts_total` returns 15 and 25 are simply less extreme versions of the interleavings shown here.

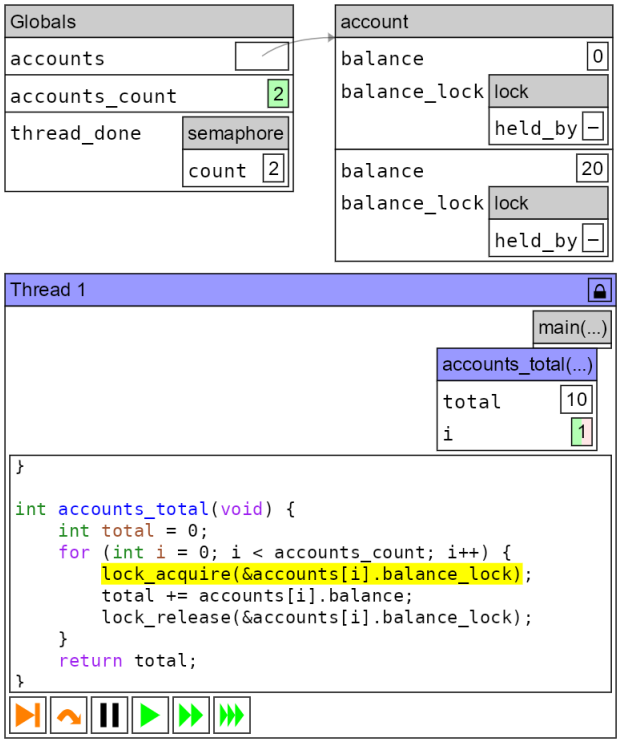


Figure 6.4: The first main issue in `bank-sum-2.c`.

The first issue is illustrated in Fig. 6.4. Here, Thread 1 executed the first

iteration of the loop inside `accounts_total` before Thread 2 and 3 transferred any money from account 0. As such, `total` was increased to 10 and Thread 1 continued to its second iteration. Before Thread 1 acquired the lock for account 1, the two other threads finished their transactions and terminated. This is the state shown in the figure. As we can see, `total` in Thread 1 is 10, but the 10 money from account 0 has already been transferred to account 1 (i.e. the balance of account 0 is 0 and the balance of account 1 is 20). As such, when Thread 1 continues, `total` will be increased to 30 and then returned.

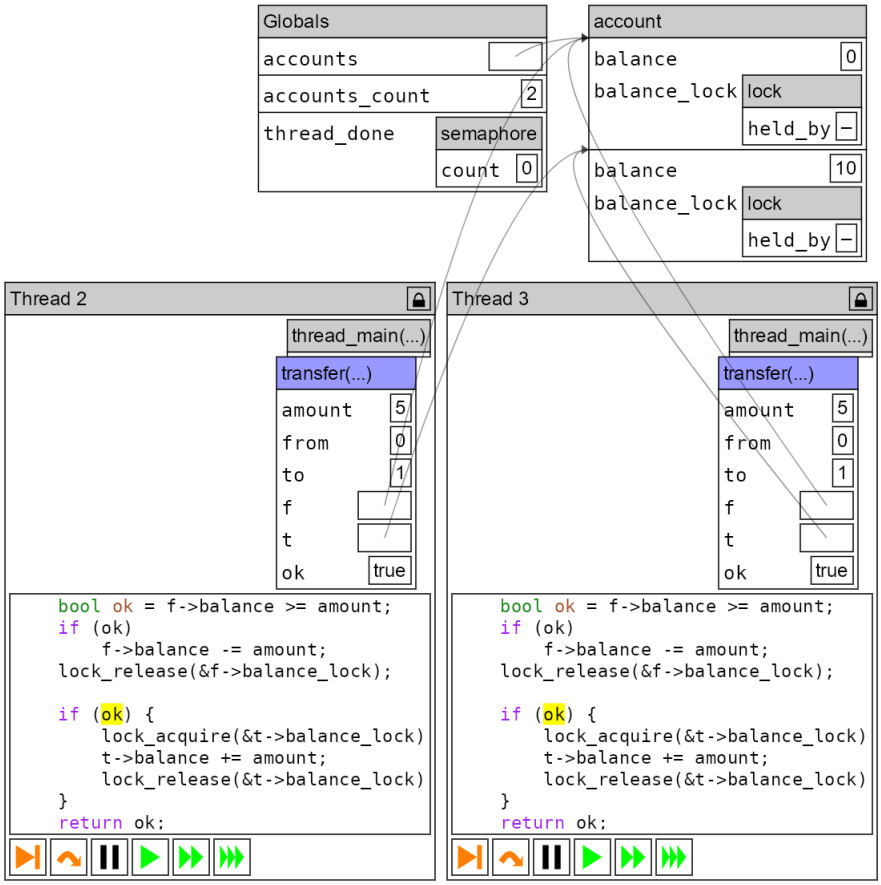


Figure 6.5: The second main issue in `bank-sum-2.c`.

The second issue is illustrated in Fig. 6.4. Here, Thread 2 and 3 have both started to transfer money. As we can see in the figure, both threads are partway through the transfer. They have both deducted 5 from the balance of account 0, but they have not yet increased the balance of account 1. If Thread 1 (not in the figure) were to execute `accounts_total` in its entirety at this point it would return 10 since the total balance of the two accounts is $0 + 10$ at this point in time.

This shows, again, that even though the program is free from data races, `accounts_total` may produce surprising results when used in concurrent programs. If we would have used `transfer` and `accounts_total` in a sequential

program (i.e. a program with only one thread), then `accounts_total` would always return the same value since a transfer would never occur concurrently with the summation.

At this point we need to take a step back and define our expectations of the `accounts_total` function in terms of requirements on the program. Most importantly, we need to determine whether we need `accounts_total` to provide the same guarantees as it would have in a sequential program, or if inaccurate results are acceptable. This very much depends on the context in which the program or the data structure is being used. In this case, we decide that it *is* important that `accounts_total` always gives an accurate result. Both in order to avoid surprising the developers that will use the functions down the line, but also since we are working with money, and losing money due to race conditions feels like a very bad idea. As such, we add a requirement to our previous list of requirements. As such, our requirements now look like this:

1. The `transfer` function should ensure that the balance of the `from` account remains non-negative by refusing to transfer money if that would be the case.

2. The `transfer` function should ensure that a transfer from some account should not be overwritten by another transfer from the same account.

3. The `transfer` function should ensure that a transfer to some account should not be overwritten by another transfer to the same account.

4. The `accounts_total` function should behave as if no transfers are in progress while it is being called. That is, when money is being transferred, it should always include money "in flight" as well as counting newly transferred money twice or not at all.

The new requirement (number 4) implies that `accounts_total` needs stronger guarantees regarding the consistency of multiple accounts. As we saw, the first problem arose because there are places in the code where `accounts_total` holds *no* lock. This means that other threads are able to interrupt it and shuffle money around in the bank (we saw that money "appeared" by transferring from low to high account numbers, but money can also "disappear" by transferring from high to low account numbers). Similarly, the second problem arose because `bank_transfer` releases the first lock before it acquires the second lock. This means that `accounts_total` is free to observe a state where money has "disappeared" since some thread is in the middle of a transfer.

Since we are trying to behave *as if* the program is sequential, the easiest way to achieve that is of course to go back to using a single lock for all accounts, as was the case in `bank-2-c.c`. This makes it easy to ensure that `accounts_total` does not execute during a transfer simply by acquiring the global lock. This solution is implemented in the program `bank-sum-slow.c` and depicted in Listing 6.15.

While this is a valid solution to the problem, we are back to the situation where we do not allow transfers between separate pairs of accounts concurrently. As such, we try to fix the issues in `bank-sum-2.c` instead. We start by addressing the first issue we found, that `accounts_total` can be interrupted by other threads that call `transfer` and thereby accidentally count the same

```
bool transfer(int amount, int from, int to) {
    struct account *f = &accounts[from];
    struct account *t = &accounts[to];

    lock_acquire(&accounts_lock);
    bool ok = f->balance >= amount;
    if (ok)
        f->balance -= amount;
    if (ok)
        t->balance += amount;
    lock_release(&accounts_lock);
    return ok;
}

int accounts_total(void) {
    int total = 0;
    lock_acquire(&accounts_lock);
    for (int i = 0; i < accounts_count; i++) {
        total += accounts[i].balance;
    }
    lock_release(&accounts_lock);
    return total;
}
```

Listing 6.15: Solution to the problems with `accounts_total` using a single lock in `bank-sum-slow.c`.

money more than once. In other words, `accounts_total` needs to see a consistent state of the balance in *all* accounts. Therefore, we can solve the problem by having it acquire the `balance_lock` for all accounts, and then release each lock after it has read the `balance`. This is implemented in `bank-sum-3.c` and depicted in Listing 6.16.

```
int accounts_total(void) {
    int total = 0;

    for (int i = 0; i < accounts_count; i++)
        lock_acquire(&accounts[i].balance_lock);

    for (int i = 0; i < accounts_count; i++) {
        total += accounts[i].balance;
        lock_release(&accounts[i].balance_lock);
    }

    return total;
}
```

Listing 6.16: Solving the first problem by acquiring *all* accounts' locks in `bank-sum-3.c`.

Practice: Use Progvis to verify that the program `bank-sum-3.c` solves the first problem. This means that it is no longer possible for `accounts_total` to return a value that is too large (i.e. only 10, 15, and 20 are possible). As before, you can use *Run* ⇒ *Look for errors...* to help you by modifying the assert statement.

As we have seen, the code in `bank-sum-3.c` does not solve the second problem, namely that `transfer` releases `balance_lock` for the `from` account before acquiring the `balance_lock` for the `to` account. As such, `accounts_total` is allowed to sum all accounts after `transfer` has removed money from the `from` account, but before it has returned money to the `to` account.

To avoid this, we simply need change `transfer` so that it acquires both locks at the same time. What happened here is that since `accounts_total` needs to be able to have a consistent view of all accounts, it changes the critical sections in `transfer`. Before we introduced `accounts_total` we concluded that `transfer` could treat the `from` and `to` accounts as separate entities. That was correct at that time. However, since `accounts_total` needs to be able to see the state of all accounts at they are either before or after a `transfer`, it means that `transfer` needs to hold the locks from when it starts accessing the first account until it is done with the second account. Essentially, the additional requirements introduced by `accounts_total` made the critical section in `transfer` larger. This again highlights that we need to consider the system *as a whole*, and we can not only focus on individual functions in isolation! Luckily, if we utilize abstraction to modularize our programs, it is usually sufficient to consider individual modules at a time. This is covered in more detail in Chapter 9.

Based on these observations, we can now solve the second problem by moving the calls to `lock_acquire` and `lock_release` as depicted in Listing 6.17. The final program is available as `bank-sum-4.c`.

```
bool transfer(int amount, int from, int to) {
    struct account *f = &accounts[from];
    struct account *t = &accounts[to];

    lock_acquire(&f->balance_lock);
    lock_acquire(&t->balance_lock);

    bool ok = f->balance >= amount;
    if (ok)
        f->balance -= amount;
    if (ok)
        t->balance += amount;

    lock_release(&t->balance_lock);
    lock_release(&f->balance_lock);
    return ok;
}
```

Listing 6.17: Solving the second problem by moving `lock_acquire` and `lock_release` as done in `bank-sum-4.c`.

At this point it is useful to take a step back and compare `bank-sum-4.c` to `bank-sum-slow.c`. The program `bank-sum-slow.c` does indeed look quite a bit simpler than `bank-sum-4.c`. However, `bank-sum-4.c` allows transfers between different pairs of accounts to occur concurrently. Note that neither of the two solutions allow two threads to run `accounts_total` concurrently: `bank-sum-4.c` acquires the locks from all accounts, while `bank-sum-slow.c` acquires the global lock. As such, which of the two is the best depends on how the code is used. If the program using the code calls `transfer` from multiple threads very frequently, but only calls `accounts_total` rarely, then the extra complexity in `bank-sum-4.c` is probably worthwhile. However, if `accounts_total` is called about as frequently as `transfer`, then the additional complexity in `bank-sum-4.c` is probably not worth it. Both regarding to the additional memory and CPU overhead involved in managing a larger set of locks, but also with regards to the complexity of the code. This is another example of why it is relevant to consider the system as a whole when working with concurrency, even though this particular aspect does not directly affect the correctness of the program.

Finally, it is worth noting that `bank-sum-4.c` *still* contains problems. We will look closer at these in the next chapter.

6.5 Working with Critical Sections

This section has been dedicated to the concept of critical sections. Determining what code needs to be a part of a critical section, and which lock should protect the critical section is not easy. However, as a part of illustrating the reasoning behind critical sections, we have also seen a strategy for finding and protecting the critical sections throughout the chapter. While it is not extremely formal and requires a good amount of creative thinking, it can be summarized as follows:

1. Start by formulating your expectations of the program or module that you need to synchronize as requirements.
2. Find variables that are shared between threads. For all variables that are modified outside of the initialization code, create a lock that protects the variable and add calls to `lock_acquire` and `lock_release` to protect accesses to the variable.
3. Use the requirements from step (1) and try to find valid uses of your program or module that behave incorrectly. This is probably the hardest step, and requires the ability to scrutinize one's own work, even though it is believed to be correct.
4. Based on the outcomes of step (3), either refine the synchronization (e.g. enlarge critical sections, merge different locks into one, etc.) or refine the requirements.
5. Repeat steps (3) and (4) until no further problems are found.

Of course, once you get more familiar with concurrent programming you can likely skip writing the first version of the program from step (2), and start with

code that already addresses the obvious issues (e.g. starting not synchronizing the if-statement and the decrement of the balance as two separate steps initially). However, the main takeaway from the steps above still remains. That is, write a solution, *critically analyze your solution*, and refine it accordingly. As mentioned above, trying to find problems in a solution you believe is correct is probably the hardest step of them all. Hopefully, this book and the help of the tools in Progvis will help you get in the right mindset at least!

6.6 Exercises

Chapter 7

Deadlocks

In the previous chapter we analyzed and synchronized a program that manages accounts in a fictional bank. At the end of the chapter, the program had two functions `transfer` and `accounts_total` that manipulated accounts with seemingly proper synchronization. Indeed, the final version of the program was free from data races and behaved according to the synchronization. However, it has the potential to *deadlock*.

> **Practice:** The program `bank-1.c` contains the code of the final version of `bank-sum` from the previous chapter, but with a different main program. In `bank-1.c`, the main program spawns one thread that transfers money from account 1 to account 0, while the main thread itself transfers money from account 0 to account 1.
>
> Use Progvis to find the problem in the program. Progvis will tell you when the program is in a deadlock. You can use *Run* ⇒ *Look for errors...* to help you. Think about *why* the program ended up in a deadlock, and why the program is not able to continue.

If you open `bank-1.c` in Progvis and select *Run* ⇒ *Look for errors...*, it will quickly state that it found a *deadlock* in the code. In visualization that shows the problem ends up in the state shown in Fig. 7.1. First and foremost we can see that both threads are waiting for something. Progvis shows this both by highlighting the relevant line in the source code blue, and by adding the box labeled *Waiting...* in the call stack. If we follow the arrow in the *Waiting...* box from Thread 1, we can see that it is waiting for `balance_lock` in account number 1 to be released, which is held by Thread 2. If we follow the arrow in the *Waiting...* box in Thread 2, we can see that it is waiting for `balance_lock` in account 0 to be released. However, that lock is currently held by Thread 1, which as we saw before is waiting for Thread 2 to release its lock.

This is a typical illustration of a deadlock: two or more threads wait for each other in a cycle. Since they are all waiting, none of them are able to continue executing code and thereby release any of the locks. As such, the typical symptoms of a deadlock is that some part of the program simply stops. Since deadlocks do not cause programs to crash, they are not always easy to find and debug. Similarly to other parts of concurrent programming, it

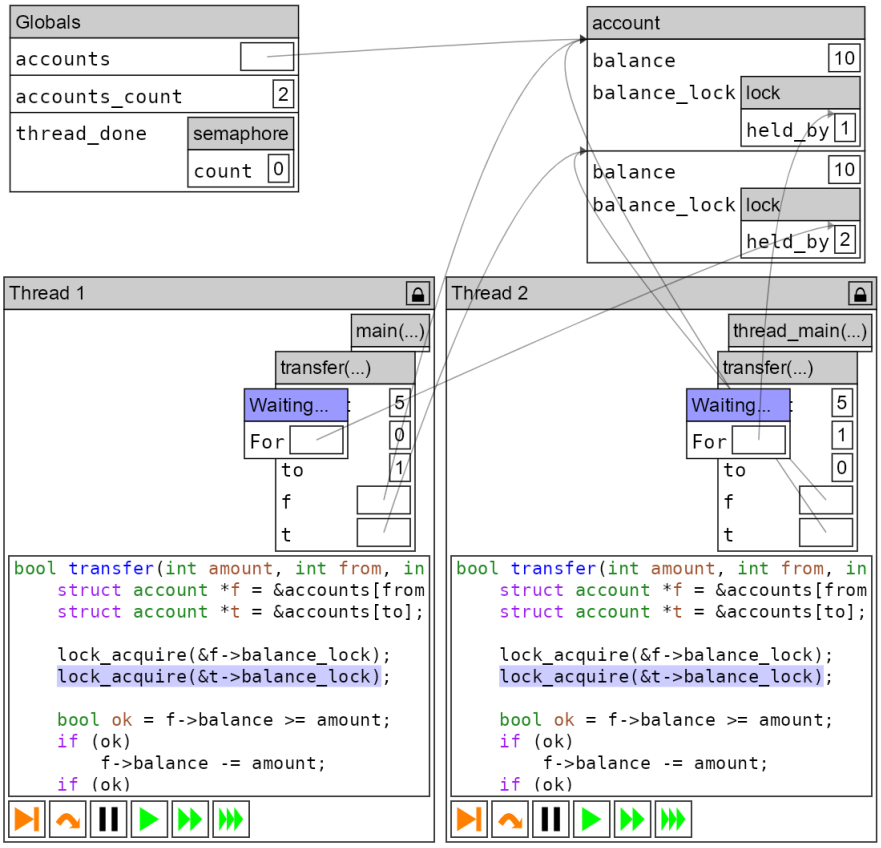


Figure 7.1: Deadlock found in `bank-1.c` by Progvis.

is therefore highly beneficial to address the issue up-front, when designing programs. Luckily, there are several approaches to ensure that deadlocks are not possible.

Before we dive deeper into deadlocks, it is perhaps relevant to mention how Progvis detects and reports deadlocks. Whenever program execution is paused (i.e. when Progvis shows the current state of the program), Progvis checks if all threads are waiting for something. If they are, Progvis reports that the program is in a deadlock. This means that even if a deadlock has occurred in some part of the program, it will not be reported until all other threads have terminated. This is not a problem when using *Run ⇒ Look for errors...*, since Progvis will find the issue eventually, but it is worth noting when stepping the program manually. Another implication of this behavior is that Progvis will report a single thread calling `sema_down` on a semaphore initialized to zero as a deadlock. While this is of course an error, it is not typically considered a deadlock in the literature.

7.1 What is a Deadlock?

For a deadlock to occur, the following 4 conditions all need to be fulfilled. They are sometimes called *the 4 conditions of deadlock*:

1. **Mutual exclusion**

   There are resources (e.g. variables or other data) in the program that threads need exclusive access in order to access. In other words, there are critical sections in the program that we protect using locks to ensure that only one thread access the resources at a time.

2. **No preemption**

   Once a thread has gained exclusive access to some resource, only the thread itself can voluntarily relinquish the exclusive access. For example, once a thread has successfully acquired a lock, the thread will continue to hold the lock until the thread itself releases the lock by calling `lock_release`. Other threads can not force the thread to release the lock prematurely.

3. **Hold and wait**

   A thread that has gained exclusive access to some resource requests exclusive access to another resource and needs to wait. For example, the thread T currently holds lock A and calls `lock_acquire` to acquire lock B. However, another thread holds lock B, so T needs to wait. As such, T holds lock A and waits for lock B.

4. **Circular wait**

   Two or more threads fulfill *hold and wait* so that they form a cycle. For example as we saw in the example above: we have two threads, 1 and 2. Thread 1 holds lock A and waits for lock B. Thread 2 waits for lock B and holds lock A.

The first three conditions (1–3) are usually referred to as *necessary conditions* for deadlock, while the last one (4) is referred to as the *sufficient condition* for deadlock. The necessary conditions stipulate properties that are required in order for circular wait to arise (i.e. the sufficient condition), but does not necessarily mean that deadlock *will* arise. The sufficient condition is, as the name implies, sufficient for a deadlock to arise. If a circular wait has occurred, then a deadlock is happening, and the necessary conditions have to be fulfilled.

7.1.1 Implications of the Four Conditions

To better understand the implications of the four conditions, we will first take a closer look at each of the four conditions of deadlock. Both at the relevance of each of them, but also their implications on programs.

As mentioned above, *mutual exclusion*, essentially means that the program uses some form of synchronization to ensure mutual exclusion when accessing some resource. This is most commonly done using locks, and we will therefore use locks as an example in the remainder of this chapter. However, it is important to remember that locks are simply one of multiple possible means to

achieve mutual exclusion. Deadlock may just as well happen if semaphores or other synchronization primitives are used. The key insight is that regardless of *how* mutual exclusion is enforced, there are times when threads need to wait for some other thread to finish using the shared resource. The waiting is the problematic part, not the locks themselves.

The name *no preemption* refers to the fact that it is not possible to *preempt* a thread *from its exclusive access to a resource*. It is worth noting that the word *preemption* is also used to describe schedulers. A *preemptive* scheduler is able to force the currently running thread to stop executing on the CPU and give other threads the opportunity to execute (as discussed in Chapter 3). In contrast, a *non-preemptive* or *cooperative* scheduler is only allowed to switch between whenever the running thread explicitly allows it.

The same word is used for condition 2 above, because it describes the same situation. However, the key difference is that condition 2 speaks about whether threads can be forced to give up exclusive access to a resource or not. As described above, *no preemption* means that the only way a thread can give up exclusive access to a resource is to give it up voluntarily. In terms of locks, this means that if thread T acquires a lock by calling `lock_acquire`, the thread itself must call `lock_release` to release the lock. There is no other way a second thread can forcefully take over the lock from T without T calling `lock_release`. As you can imagine, writing correct code in a system where this is possible would be borderline impossible. The whole point of a lock is to protect a critical section so that other threads are unable to interfere with shared data. If other threads would be able to take over locks, this is no longer true and most benefits of mutual exclusion disappear.[1]

One important observation from above is that without the ability to preempt locks from threads, the only way a thread can release a lock is by calling `lock_release` itself. That is, the thread needs to execute code in order to release a lock. As such, if a thread is waiting for something else, it is unable to release any locks it is currently holding.[2]

The last of the necessary conditions, *hold and wait*, combines the key insights of the two previous conditions. That is, *mutual exclusion* means that threads may need to wait for something, and *no preemption* means that threads that wait are unable to release resources. If we combine them, we get *hold and wait*, which is the observation that if a thread, T, currently holds a lock, A, and attempts to acquire another lock, B, the T will be unable to release A until it acquires B. One way to think of this is that T creates a link between A and B. In this case, A can not be acquired until the thread that holds B releases it.

The final condition, *circular wait*, essentially uses *hold and wait* to create a cycle of threads waiting for each other. We just saw that we could use hold and wait to link threads together. If we have two threads, T and U, and assume that T holds A while U holds B. Then if T attempts to acquire B, we get a situation similar to in the previous paragraph. Lock A can not be released until

[1] One possibility is to use *transactional memory*, but the hardware support required for that is scarce at the time of writing.

[2] For example, this means that even if you implement a mechanism that allows threads to ask the thread currently holding a lock to release its lock, we still have *no preemption*. Since the other threads *ask* rather than *take* the lock, the thread holding the lock still needs to realize that it has been asked to release the lock, perhaps finish a part of what it is doing, and then release the lock.

thread T successfully acquires lock B, which requires that thread U releases it. However, if instead of releasing lock B, thread U attempts to acquire lock A we get a problem. At this time, we have essentially created two links: lock A will not be released until lock B is released, and lock B will not be released until lock A is released. Since we have created these links in a circle, we can conclude that none of the locks will be released (due to *no preemption*), and therefore that neither thread T nor thread U will ever acquire their second lock.

7.1.2 Resource Allocation Graphs

As you have most likely realized from the textual descriptions above, it is difficult to get an overview of even relatively small systems without a good representation of the state of the system. Resource allocation graphs is representation that is useful for representing threads in relation to mutually exclusive resources.

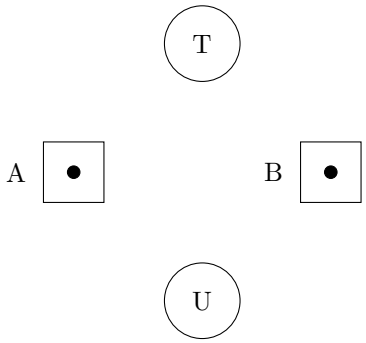


Figure 7.2: The initial state of a system that contains two threads (T and U) and two shared resources (A and B).

We will use the scenario above to demonstrate how resource allocation graphs can be used to illustrate the state of a concurrent system. As a starting point, a program contains two threads, T and U, and two shared resources that require mutually exclusive access. This initial state is depicted in Fig. 7.2. The threads are represented as circles that contain the name of the thread. Each resource is represented as a dot. The dots are in turn grouped into squares with a label. All dots within the same box are treated as equivalent and interchangeable, and as such only the boxes are labeled. In this example, we imagine that the resources A and B correspond to locks that each protect a shared variable. Therefore, each box contains a single dot (we can think of the dot as corresponding to "the ability to access the shared variable").

If a box contain more than one dot, all dots are treated as equivalent. This represent cases when a certain thread will need one instance of a resource but does not care which one it gets, since they are equivalent. As an example, imagine a thread that needs to store temporary data on a disk connected to the computer. If the system has multiple disks, the program does not care *which* one it gets to use, as long as it gets exclusive access to that disk. We could also model RAM usage in the same way by letting each dot represent 1 MiB of RAM for example. When working with examples in the scope of this book, these situations will be rare, since none of the standard synchronization

primitives support this use-case out of the box (i.e. there is no version of the lock that acquires one out of many resources, but it is possible to implement one if we need one as we will see in Section 8.4).

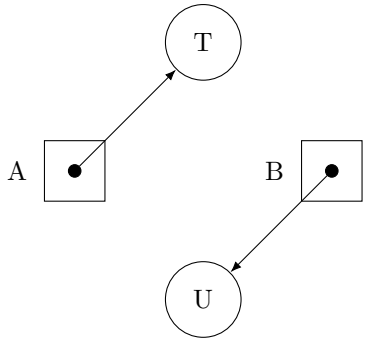


Figure 7.3: Thread T has acquired one instance of resource A and thread U has acquired one instance of resource B.

The next step in our scenario is that thread T acquires one instance of resource A, and thread U acquires one instance of resource B. As shown in Fig. 7.3, we illustrate this by drawing an arrow from the dot in the resource to the thread that has access to it. One way to think of the representation is that the arrow indicates that a particular dot is moved to the thread itself. In this case, that the threads have the ability to access the variables protected by locks A and B.

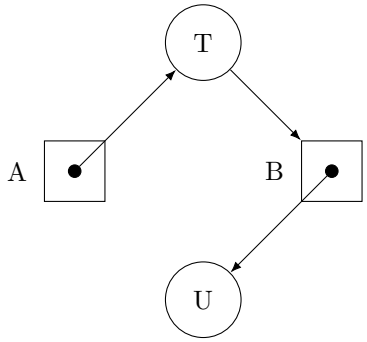


Figure 7.4: Thread T has acquired one instance of resource A and thread U has acquired one instance of resource B.

After T has successfully acquired an instance of resource A (e.g. acquired the lock), it attempts to acquire an instance of resource B. As we can see from Fig. 7.3, this is not possible, since the only instance of resource B is currently held by thread U. As such, thread T needs to wait. Again, we represent this as an arrow, but this time from the thread to the box that represent the resource(s) the thread is waiting for, as shown in Fig. 7.4. The reason we draw the arrow to the box rather than one of the dots in the box is to make it clear that the thread is waiting for *one* of the possible multiple equivalent resources that are in the box.

At this point we can start to see the benefits of the graphical notation. By following the arrows, we can see that resource A is acquired by thread T, that thread T is waiting for an instance of resource B, that is in turn acquired by thread U. As such, resource A is currently indirectly blocked by thread U, as it can not be released until thread U releases B.

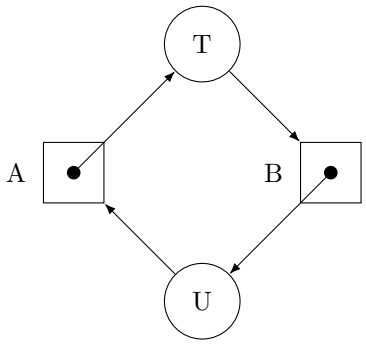


Figure 7.5: Thread T has acquired one instance of resource A and thread U has acquired one instance of resource B.

Finally, thread U attempts to acquire an instance of resource A. Again, no instance of resource A is available, so thread U needs to wait. As before, we indicate this by adding a line from thread U to the box labeled A. This results in the representation in Fig. 7.5. In the figures we can see that the lines form a cycle. This means that resource A is held by thread T, which is waiting for resource B, which is held by thread U, which in turn waits for resource A. As such, the two threads are indirectly waiting for themselves, which is clearly visible in the resource allocation graph.

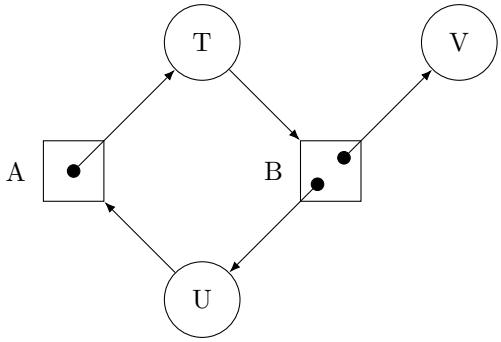


Figure 7.6: An example of a resource allocation graph with multiple instances of resource B.

It is worth noting that some care needs to be taken when working with resources that have multiple instances. For example, resource B in Fig. 7.6 has two instances. One instance is acquired by thread U and the other by thread V. In this case, even though the graph *does* contain a cycle, the threads are actually not currently in a deadlock. Since one instance of B is held by thread V, and thread V is currently not waiting for anything, V can release its instance of resource B, which means that thread T can acquire it and resolve

the deadlock, which results in the situation in Fig. 7.7. As such, for a deadlock to have occurred in graphs where there are multiple instances of some resource the graph needs to contain cycles for *all* instances.

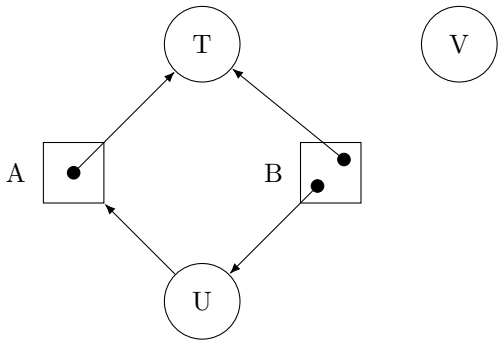


Figure 7.7: The situation in Fig. 7.6 after thread V released its instance of resource B.

It is worth noting that the situation in Fig. 7.6 may eventually lead to a deadlock. For example, if thread V would have attempted to acquire an instance of resource A rather than releasing its instance of resource B then both instances of resource B would have been part of a cycle, and the system would have been in a deadlock.

7.2 Preventing Deadlocks

The 4 conditions for deadlock (Section 7.1) can also be used to prevent deadlocks from occurring. The idea is simple, we know that all conditions need to be fulfilled for a deadlock to occur. This means that we can avoid deadlocks by ensuring that at least one of the conditions can never be fulfilled.

As such, we will discuss how we can break each of the conditions below. Then we will see how we can apply them to prevent deadlocks from occurring in the `bank-1` program.

1. **Mutual exclusion**

   Since deadlocks can not occur without mutual exclusion, we can avoid deadlocks by avoiding mutual exclusion. As we have seen earlier in this book, we typically need mutual exclusion *somewhere* in a concurrent program in order to avoid data races, so it is often not possible to avoid mutual exclusion altogether.

2. **No preemption**

   Another option to avoid deadlocks would be to allow threads to take over locks that other threads have acquired. This is often not a viable approach either, since this more or less eliminates the benefits of locks altogether.[3]

[3] At least without mechanisms like *transactional memory*.

3. **Hold and wait**

   Sometimes it is possible to structure the program so that it never holds a lock while waiting for something else. One way to achieve this is to make sure to never hold more than one lock at a time. Doing this means that it is impossible to build "links" between two resources, and thereby circular wait can not occur.

4. **Circular wait**

   Even when the other conditions are fulfilled, it is still possible to avoid a circular wait. That is, implementing some way to make sure that cycles in the resource allocation graph can not form. A simple way to achieve this is to ensure that all threads acquire their locks according to some total order.

It is worth examining the idea for how circular wait can be broken in more detail. Consider a system with 5 resources and 5 threads as depicted in Fig. 7.8. As can be seen in the figure, the resources are labeled R1–R5 and the threads are labeled T1–T5. The system is also in a deadlock since the edges in the graph form a cycle that involves all threads and all resources.

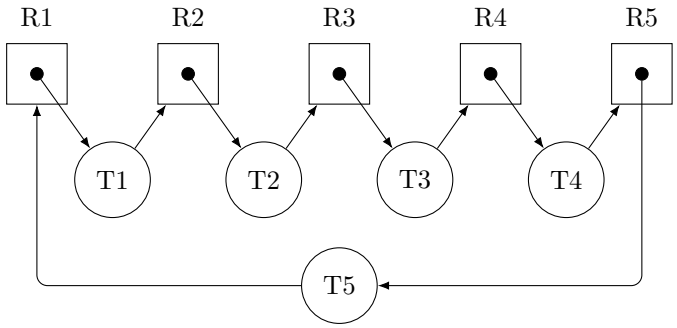


Figure 7.8: Generalization of a circular wait in a resource allocation graph.

One important observation from the graph is that for all threads except T5, thread Tn has acquired resource Rn and waits for resource R$n+1$. That is, thread T1 has acquired R1 and then attempted to acquire R2, but had to wait since R2 was already acquired by another thread. In particular, this means that each of the threads hold resources that have lower numbers than the resource they are waiting for (i.e. the arrows go to the right in the figure). This is, however, *not* true for T5, which holds R5 and waits for R1 (i.e. the arrows go to the left in the figure). As such, T5 needs to have acquired R5 before it attempted to acquire R1.

The key observation here is that we need threads like T5 that "break the rule" in order to form a cycle. Without T5, all edges from resources in the figure would go from left to right, and thereby we can not form a cycle. We need at least one edge from a resource that goes right to left to be able to close the cycle. If T5 would follow the same pattern as the other threads, that is it acquires the lower numbered resource before the higher numbered resource, either T1 or T5 would have to wait for R1 before attempting to acquire its second resource. For example, if T1 acquired R1 before T5, the resource allocation graph would

look like Fig. 7.9. Importantly, the rule that multiple resources need to be acquired according in the same order prevents cycles from forming since the highest numbered resource will always be acquired last.

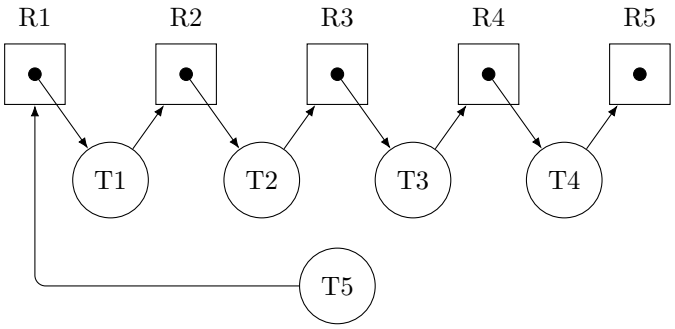


Figure 7.9: Generalization of a circular wait in a resource allocation graph.

Practice: In the example above, threads acquire resources with numbers adjacent to each other (i.e. R1 and R2, R2 and R3, etc.). Of course the idea works regardless of exactly which resources need to be acquired, and even the number of resources that should be acquired. To illustrate this, consider the scenario below. First, draw one resource allocation graph where threads acquire the resources in the order they are listed below. Then, draw another resource allocation graph where threads instead acquire resources in a fixed order (e.g. increasing or decreasing). Remember that the thread stops execution once it needs to wait for a resource that is acquired by another thread!

T1: R2, R4

T2: R1, R3, R4

T3: R5, R2

T1 (after T3 is done): R5

If you have done the diagrams correctly, you will see that the system enters a deadlock if you follow the order above, but the deadlock is avoided when locks are acquired in a specific order.

Finally, it is worth noting that the numbering of the threads does not matter at all. It just makes it easier to see what happens in the example. The numbering of the locks is not important. What matters is that all locks that are used together are always acquired in the same order. For example, taking the locks above in the order 4, 2, 3, 1, 5 will still avoid deadlock. This shows that the important part is that there is *a* consistent order, not *which* order is used.

7.3 Deadlock Avoidance in Programs

With knowledge of both what a deadlock is and how deadlocks can be avoided, it is time to apply the knowledge to the `bank` program. As we saw in Section 7.2, we have two options. We either break *hold and wait* or *circular wait* (as we have seen, we need *mutual exclusion* since there is shared data). Breaking *hold and wait* is possible by replacing the per-account `balance_lock` with a single lock for all accounts, which gives us the program `bank-sum-slow.c` covered in the previous chapter (see Listing 6.15). However, as mentioned in the previous chapter this means that it is not possible for multiple threads to transfer money between different pairs of accounts concurrently. As mentioned previously, this might be a reasonable solution depending on how `transfer` and `accounts_total` are used. After all, the implementation of `bank-sum-slow.c` is simpler than that of `bank-1.c`, and since it does not have *hold and wait*, it can never cause a deadlock.

As we saw above, breaking *hold and wait* sometimes requires compromising on the available parallelism.[4] As such, we will instead try to break *circular wait* and hope that it leads us to a solution that avoids deadlock while allowing multiple transactions to happen concurrently. The easiest way to achieve that is to use the idea outlined at the end of Section 7.2, namely to make sure that threads acquire locks according to some total ordering. One way to do this is to give each lock a number and make sure to acquire locks in increasing numerical order (at least conceptually, we do not necessarily need to write it down other than "always acquire lock X before lock Y"). In the case of `bank-1.c`, this is quite easy. The only locks we have in the program are the `account_lock`s for each account. Since the accounts are already numbered, we can simply use the account number to order the locks.

> **Practice:** Apply the idea outlined above to solve the deadlock we discovered in `bank-1.c` at the start of the chapter. The file `bank-1-larger-test.c` contains the same code as `bank-1.c`, but has a main program that also calls `accounts_total` so that you can verify that `transfer` and `accounts_total` work properly together. You can use *Run* ⇒ *Look for errors...* to verify your solution.

As you have likely realized above, the idea of utilizing account numbers to ensure threads acquire the `account_lock`s in the same order can be applied quite easily. One way is the approach implemented in `bank-2.c`. As can be seen in Listing 7.1, we simply check which account number is larger before acquiring the locks, and acquire the lock for the account with the lowest number first. Note that even though it may seem natural to release the locks in the opposite order we acquired them in, the order does not matter at all since `lock_release` never needs to wait.

Even though we have focused on `transfer` so far in the chapter, it is important to note that it is important that *all* parts of the program adhere to

[4] This depends on the program as a whole. For example, the program `bank-3-c.c` from the previous chapter does not have *hold and wait* even though it uses one lock for each account. The same is true for `bank-sum-2.c`, even though that program has other odd behaviors that are probably undesirable.

```
bool transfer(int amount, int from, int to) {
    struct account *f = &accounts[from];
    struct account *t = &accounts[to];

    if (from < to) {
        lock_acquire(&f->balance_lock);
        lock_acquire(&t->balance_lock);
    } else {
        lock_acquire(&t->balance_lock);
        lock_acquire(&f->balance_lock);
    }

    bool ok = f->balance >= amount;
    if (ok)
        f->balance -= amount;
    if (ok)
        t->balance += amount;

    lock_release(&t->balance_lock);
    lock_release(&f->balance_lock);
    return ok;
}
```

Listing 7.1: Modifications to `bank-1.c` to avoid deadlocks.

the order in which locks should be acquired. In this case, we also need to make sure that `accounts_total` acquires locks in the same order as `transfer`. By accident, `bank-2.c` happens to be correct simply because it was natural to acquire the lock for the account with the lowest number first in both cases.

Practice: Change the condition in the `if` statement from `from < to` into `from > to` (available as `bank-2-error.c`), and use *Run* ⇒ *Look for errors...* to see what happens when `transfer` acquires locks in a different order compared to `accounts_total`.

7.4 Exercises

- Resource allocation graphs.
- See if there is a risk for deadlock in some programs.
- Solve deadlock in some programs.

Chapter 8

Condition Variables

Up to this point we have introduced semaphores and locks. As we have seen, semaphores are powerful enough to solve any concurrency issue while locks are specialized for mutual exclusion. This makes locks easier to use to protect shared data, but also limits their capabilities. In particular, it is *not* possible to use locks to wait for some event to occur (i.e. we can *not* implement a semaphore using only one or more locks).

This chapter introduces *condition variables*, which can be used in conjunction with locks to make it possible to waiting for events. As such, condition variables can be viewed as an extension of the capabilities of locks to make them as powerful as semaphores (i.e. we can implement a semaphore using a lock and a condition variable). Even though we do not gain any additional power in terms of problems that are solvable by introducing condition variables, they work differently from semaphores. Therefore it is easier to solve some synchronization problems with locks and condition variables than with semaphores, and vice versa.

8.1 Semantics

As in Chapter 5, we start by introducing the available operations and their semantics. Again, we focus on how the operations are used. The full definitions are available in Chapter A for the curious reader.

A condition variable can be thought of as containing a set of zero or more threads that are currently waiting. Just as semaphores and locks, a condition variable is an opaque data structure that is only possible to modify by calling the operations below.

It is also useful to consider the condition variable to be associated with a lock that is used to protect some variables that we wish to wait until they have a particular value. To mirror the semantics in Pintos[1] we consider the condition variable itself to *not* be threadsafe, and the lock to protect the condition variable as well. While this strict view is not always necessary (many implementations of condition variables *are* threadsafe), it is useful since it avoids sporadic wakeups, and some implementations (e.g. pthreads) still require that

[1] Which is the source of the nomenclature for the synchronization primitives used in this book.

each condition variable is used with one lock. As such, thinking of the condition variable as not being threadsafe and thereby needing protection from a lock makes it easier to reason about the condition variable and the behavior of the program.

```
struct condition c;
```

The line above defines a variable named `c` that contains a condition variable. As we can see, the type is named `struct condition`.

```
struct lock l;
```

To use the condition variable, we also need a lock. Each condition variable should be associated with exactly one lock (i.e. each condition variable should be protected by one lock, but using one lock to protect multiple condition variables is fine). For this reason it is useful to think of the condition variable as a shared variable that needs to be protected by the associated lock, even if this is not strictly necessary in most implementations.

```
cond_init(&c);
```

Each condition variable needs to be initialized before it is used. As with locks, the initialization function does not require any additional parameters apart from a pointer to the condition variable to initialize. Of course we need to initialize the lock as well, but we omit the initialization since this chapter focuses on condition variables.

```
cond_wait(&c, &l);
```

Put the current thread to sleep and store the thread in the condition variable.[2] The function assumes that the current thread holds the lock (`l`). It makes sure to release the lock just before the thread is put to sleep, and to re-acquire the lock once the thread wakes up again. As such, the calling thread needs to hold the lock before calling `cond_wait` and will still hold the lock afterwards. However, it is important to remember that the lock is released during the call to `cond_wait`, since other threads may have acquired the lock and changed the variables protected by the lock.

```
cond_signal(&c, &l);
```

Wakes *one* of the threads that is currently sleeping and stored inside the condition variable `c`. Requires that the lock `l` is held by the current thread. If no threads are stored in the condition variable, nothing happens. If more than one thread is sleeping, an arbitrary thread is picked to wake up.

```
cond_broadcast(&c, &l);
```

Wakes *all* of the threads that is currently sleeping and stored the condition variable. Requires that the lock `l` is held by the current thread.

It is worth noting that many implementations of condition variables do not require that the lock (`l`) is passed to `cond_signal` and `cond_broadcast`.

[2]The condition variable typically stores some form of reference to the thread (e.g. a thread identifier) rather than the thread itself, but conceptually we can consider it to store the thread.

They also do not require that the lock associated with the condition variable is held by the current thread when `cond_signal` and `cond_broadcast` is called. However, as mentioned above, holding the lock while calling `cond_signal` and `cond_broadcast` is still good practice as it eliminates certain forms of sporadic wakeups. Furthermore, in most cases the thread that calls `cond_signal` and `cond_broadcast` needs to hold the lock just before the call anyway, so the call to `lock_release` can simply be placed after `cond_signal` and `cond_broadcast` rather than acquiring the lock again.

8.1.1 Using `cond_wait` and `cond_signal`

Now that we have seen the operations provided by a condition variable, we start by observing them in practice in Progvis. For this, we will use the program `semantics-signal.c` which is shown in Listing 8.1. The program declares two global variables, a condition named `cond` and a lock named `cond_lock`. Note that we name the lock `cond_lock` to remind ourselves that we use the lock to protect the variable `cond`, just as we have done with other variables previously. After initializing the lock and the condition variable, the `main` function starts a new thread that executes `thread_fn`, acquires the lock, calls `cond_signal`, and then exits. The second thread simply acquires the lock, calls `cond_wait`, and releases the lock.

```
struct condition cond;
struct lock cond_lock;

void thread_fn(void) {
    lock_acquire(&cond_lock);
    cond_wait(&cond, &cond_lock);
    lock_release(&cond_lock);
}

int main(void) {
    cond_init(&cond);
    lock_init(&cond_lock);

    thread_new(&thread_fn);

    lock_acquire(&cond_lock);
    cond_signal(&cond, &cond_lock);
    lock_release(&cond_lock);

    return 0;
}
```

Listing 8.1: The program `semantics-signal.c` used to illustrate the semantics of condition variables.

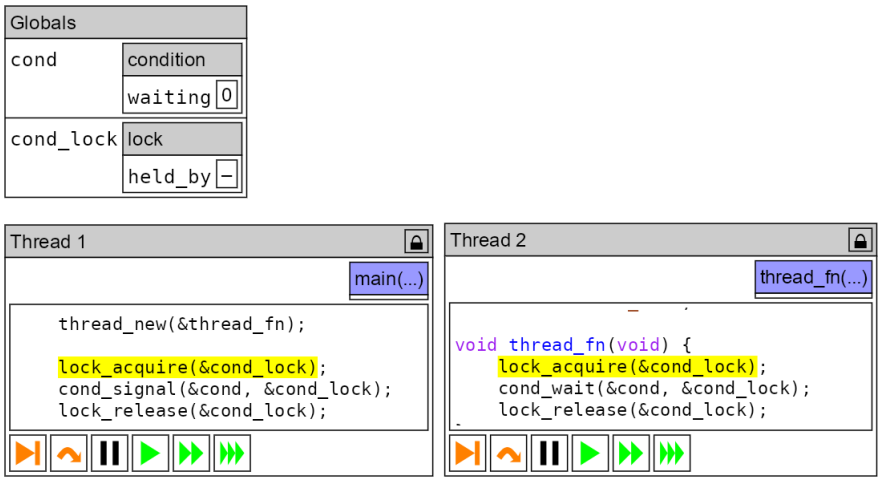


Figure 8.1: The program `semantics-signal.c` after initialization.

If you open the program in Progvis and step Thread 1 to the end of the initialization, you will see a figure similar to the one in Fig. 8.1. Note that Progvis represents a condition variable as a box that contains a value named *waiting* whose value is currently zero. This is how Progvis represents the number of threads that are currently waiting on the condition variable.

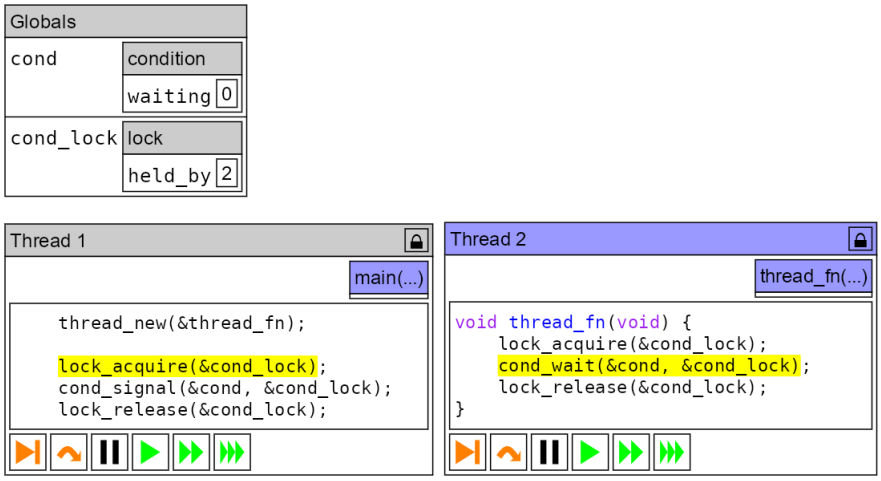


Figure 8.2: The program `semantics-signal.c` after Thread 2 has acquired the lock.

If we step Thread 2 once, the program will reach the state shown in Fig. 8.2, where Thread 2 has just acquired the lock `cond_lock`. As we have seen before, Progvis represents this by displaying the number 2 next to the text *held_by* in the box for the lock.

If we step Thread 2 once more, it will call `cond_wait`. As the name implies, this puts the thread to sleep. Progvis illustrates this as in Fig. 8.3. First and foremost, we can see that Thread 2 is sleeping, both since the highlighted line

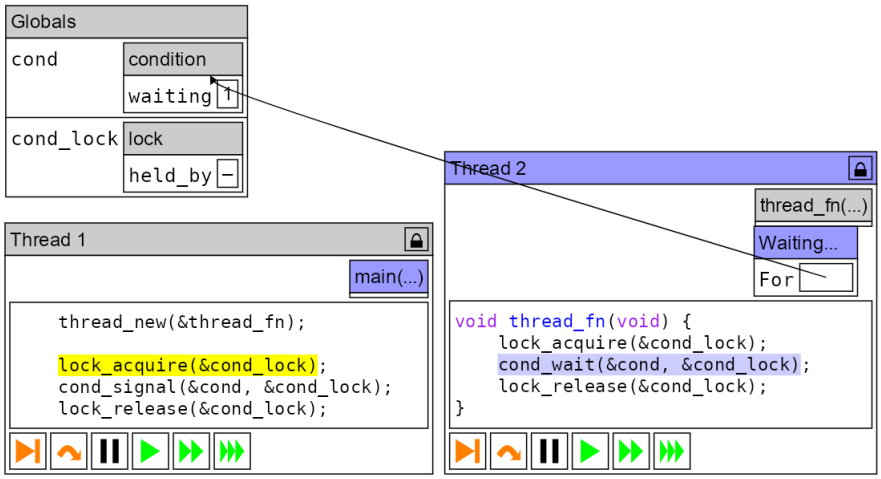


Figure 8.3: The program `semantics-signal.c` after Thread 2 has called `cond_wait`.

is blue, and due to the box labeled *Waiting...* on the thread's call stack that points to the condition variable. We can also see that the condition variable in the *Globals* box says that one thread is waiting. Another important detail that is easy to miss is that Thread 2 *no longer* holds the lock. We can see this by observing that *held_by* is a dash, which means that the lock is not held by any thread. This is fortunate, since Thread 1 needs to acquire the lock to call `cond_signal`.

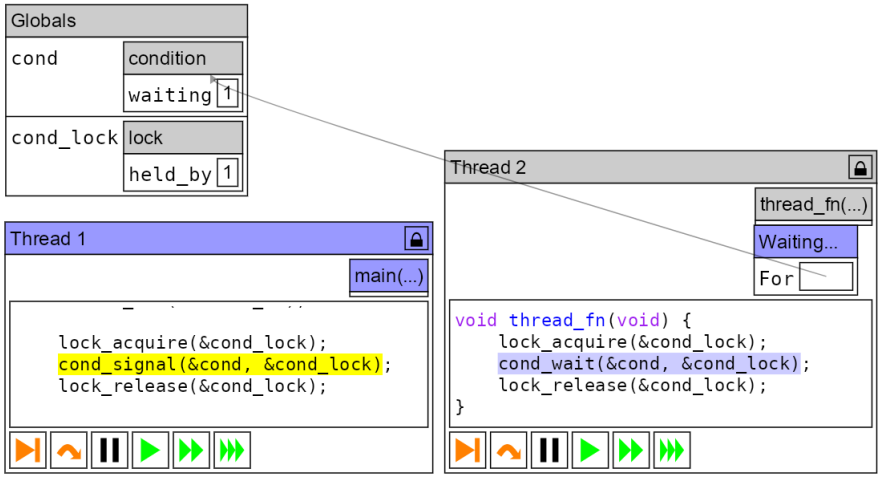


Figure 8.4: The program `semantics-signal.c` after stepping Thread 1.

At this point, our only option is to step Thread 1. If we do that, we will end up in the state depicted in Fig. 8.4, where Thread 1 has acquired the lock.

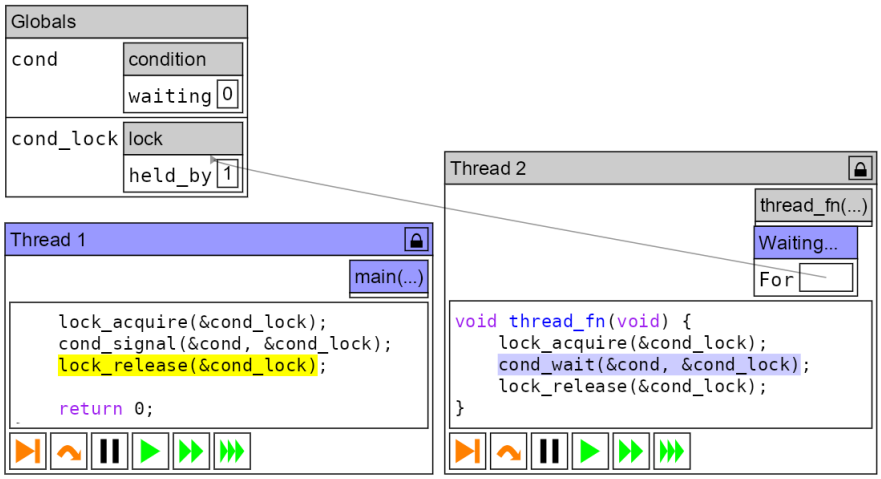


Figure 8.5: The program `semantics-signal.c` after stepping Thread 1 yet again, just after it has returned from the call to `cond_signal`.

If we step Thread 1 once more, it will call `cond_signal`. This instructs the condition variable to wake *one* of the threads that are currently waiting. In this case, Thread 2 is the only option, so the condition variable wakes Thread 2. After this, `cond_signal` returns and the program ends up in the state in Fig. 8.5. Interestingly, Thread 2 is still waiting for something. If we follow the arrow from the *Waiting...* box in Thread 2's call stack, we will see that the thread is indeed no longer waiting for the condition variable, but for the lock! Remember that `cond_wait` does three things: (1) it releases the lock, (2) puts the thread to sleep, and (3) once the thread wakes up it re-acquires the lock. What happened here[3] is that the thread was woken up and then immediately attempted to re-acquire the lock, which is still held by the thread that called `cond_signal` (Thread 1 in this case).

If we step Thread 1 a final time, it will call `lock_release` and thereby allow Thread 2 to continue. As shown in Fig. 8.6, Thread 2 now holds the lock again. This might initially seem strange. However, if we consider `lock_acquire` and `lock_release` as markers for the start and end of a critical section, it is sensible to let `cond_wait` denote a temporary break in the critical section. That way, the intuition that `lock_acquire` and `lock_release` denotes the start and end of a critical section remains, and we do not have to remember if `cond_wait` is replaces the start or the end of the critical section.

[3]This is indeed a common case, especially in our implementation where it is necessary to hold the lock when calling `cond_signal`. Because of this, many implementations optimize this case by not actually waking the thread just for it to immediately sleep again since the lock is still held by the current thread. Instead, they simply move the thread from the waiting queue for the condition variable to the waiting queue for the lock.

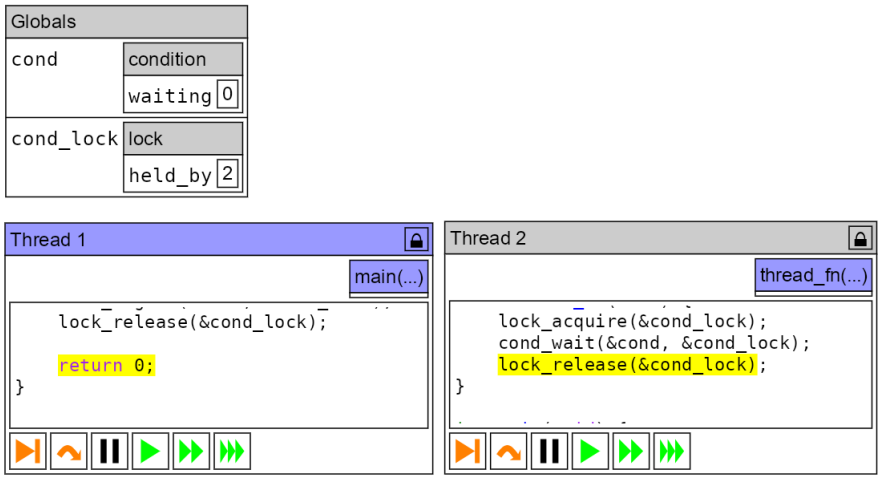


Figure 8.6: The program `semantics-signal.c` after Thread 1 releases the lock.

> **Practice:** The example above only illustrated what happens if Thread 2 calls `cond_wait` *before* Thread 1 calls `cond_signal`. Use Progvis to investigate what happens if the threads execute the other way around!

As you have likely noticed above, if Thread 1 calls `cond_signal` before Thread 2 has called `cond_wait`, no threads are waiting for the condition when `cond_signal` is called, and thereby no threads are woken up. This is not considered an error, so Thread 1 simply continues executing. However, once Thread 2 calls `cond_wait`, no thread will wake it up and it will therefore sleep forever. Progvis will report this as a deadlock once Thread 1 terminates. However, as discussed in Chapter 7, this situation is not what is usually meant by the term *deadlock*, even if the symptoms are similar to a cycle of threads waiting for each other.

8.1.2 The Difference Between `cond_signal` and `cond_broadcast`

The example above only used `cond_signal`. Before we move on to examine how programs are used we first take a closer look at the difference between `cond_signal` and `cond_broadcast`. For this, we use the program `semantics-broadcast.c`, which is the same as `semantics-signal.c` except that the `main` function is replaced with the implementation in Listing 8.2.

```
int main(void) {
    cond_init(&cond);
    lock_init(&cond_lock);

    thread_new(&thread_fn);
    thread_new(&thread_fn);
    thread_new(&thread_fn);

    lock_acquire(&cond_lock);
    cond_signal(&cond, &cond_lock);
    lock_release(&cond_lock);

    lock_acquire(&cond_lock);
    cond_broadcast(&cond, &cond_lock);
    lock_release(&cond_lock);

    return 0;
}
```

Listing 8.2: The `main` function of `semantics-broadcast.c`.

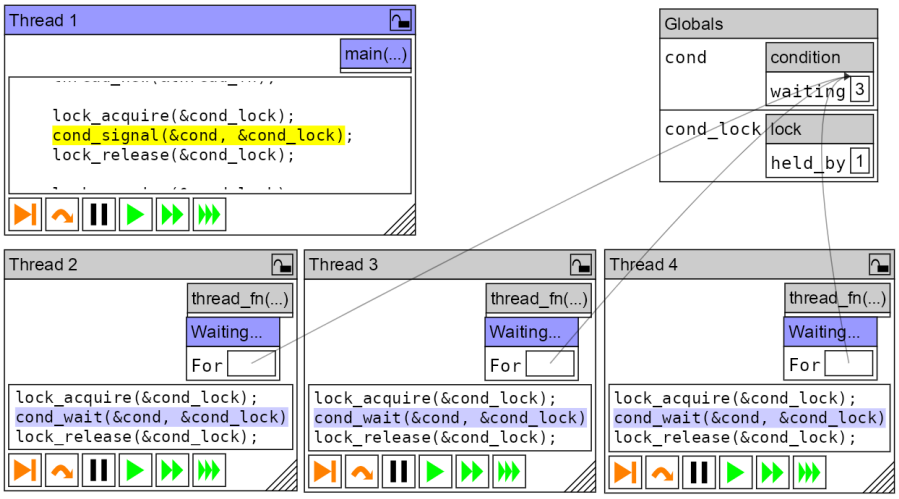


Figure 8.7: The program `semantics-broadcast.c` once threads 2–4 are waiting for the condition and Thread 1 is about to call `cond_signal`.

We will start from the state in Fig. 8.7, where threads 2–4 are waiting on the condition inside `cond_wait`, and Thread 1 is about to call `cond_signal`. To reach the state, start by stepping Thread 1 until it has called `thread_new` three times. Then step threads 2–4 until they all wait inside `cond_wait`. Finally, step thread 1 until it reaches the line `cond_signal` in Fig. 8.7.

At this point, three threads are waiting on the condition. If we step Thread 1 it will call `cond_signal` and we will be able to see how it behaves. As noted before, `cond_signal` wakes *one* of the threads that wait for the semaphore (as with other synchronization primitives, it is not well-defined which one, so it is not possible to rely on first-in first-out). In this case, Thread 2 is woken, and

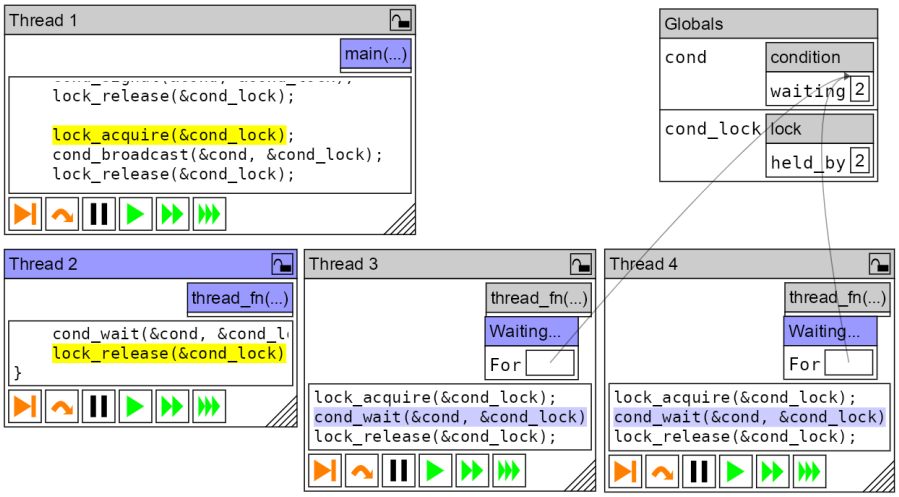


Figure 8.8: The program `semantics-broadcast.c` after `cond_signal` wakes one of the threads.

as before Thread 2 will immediately start waiting for the lock that is currently held by Thread 1. Once we step Thread 1 once more, it releases the lock and thereby allows Thread 2 to acquire the lock and continue, as shown in Fig. 8.8.

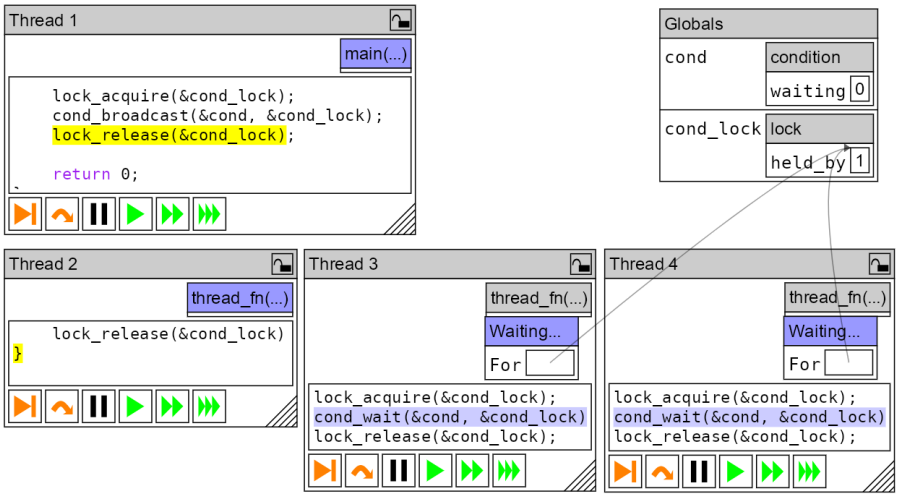


Figure 8.9: The program `semantics-broadcast.c` just after `semantics-broadcast.c` after `cond_broadcast` has been called.

If we continue stepping Thread 1 until it has called `cond_broadcast` (which requires stepping Thread 2 until it releases the lock), we see the difference between `cond_signal` and `cond_broadcast`. While `cond_signal` only wakes *one* of possible multiple threads that wait on the condition variable, `cond_broadcast` wakes *all* of them. We can see this in Fig. 8.9, since both Thread 3 and Thread 4 are now waiting for the lock rather than the condition. Just as before, this is because `cond_wait` re-acquires the lock when threads wake up,

before it returns to the caller.

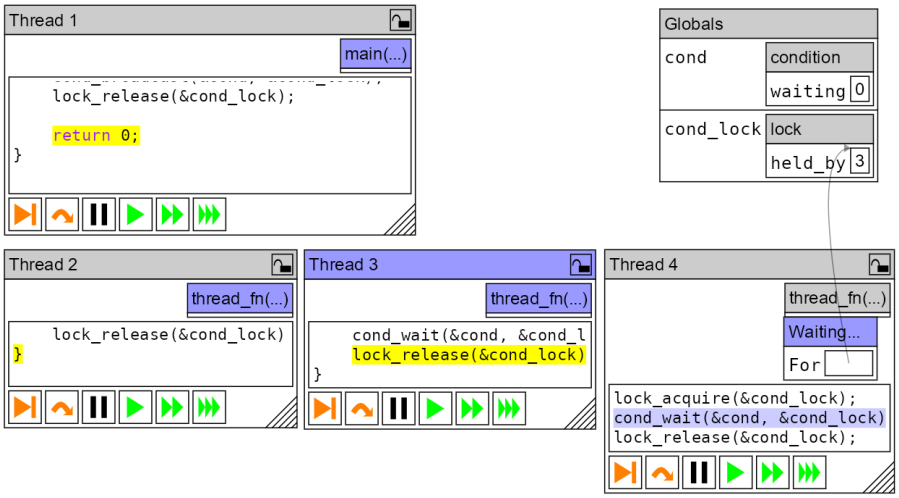


Figure 8.10: The program `semantics-broadcast.c` after Thread 1 releases the lock, which allows Thread 3 to acquire the lock.

If we let Thread 1 execute another step, it will release the lock. This allows *one* of the two threads to acquire the lock and continue execution. In Fig. 8.9, this happens to be Thread 3 (again, just as with other synchronization primitives, the order is typically not well-defined so we can not rely on some particular ordering even though Progvis follows first-in first out).

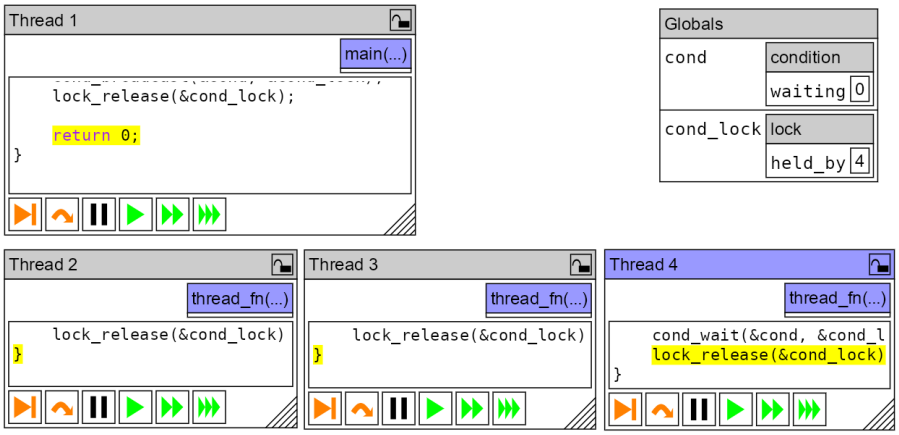


Figure 8.11: The program `semantics-broadcast.c` after Thread 3 releases its lock.

At this point, it might seem like there is no real difference between `cond_signal` and `cond_broadcast` since the lock only allows one thread to continue anyway. The key difference is, however, that Thread 4 is currently waiting for the lock rather than the condition variable (in contrast to Fig. 8.8). This

means that once Thread 3 releases the lock (which it does if we step it once), Thread 4 will be allowed to acquire the lock and thereby continue. As such, `cond_broadcast` allows all threads to continue, but the lock makes them start one by one in some order.

We could of course achieve the same behavior by calling `cond_signal` two times in Thread 1. However, since it is not possible to ask a condition variable if any threads are currently waiting on it, we must either provide such a mechanism ourselves, or call `cond_signal` enough times, which is likely at least a few too many. Neither of these solutions are very convenient, so `cond_broadcast` is a good alternative here, especially since `cond_broadcast` is a fairly common operation.

8.2 Condition Variables and Semaphores

The two programs we used above had the same issue. If `cond_signal` or `cond_broadcast` was executed before `cond_wait`, some threads would sleep forever. This was not a problem we encountered when using semaphores! Why is this an issue with condition variables and not semaphores?

```
int result;

struct semaphore sema;

void thread_fn(void) {
    result = 10;


    sema_up(&sema);

}

int main(void) {

    sema_init(&sema, 0);

    thread_new(&thread_fn);


    sema_down(&sema);


    printf("%d\n", result);
    return 0;
}
```

```
int result;
struct condition cond;
struct lock l;

void thread_fn(void) {
    result = 10;

    lock_acquire(&l);
    cond_signal(&cond, &l);
    lock_release(&l);
}

int main(void) {
    cond_init(&cond);
    lock_init(&l);

    thread_new(&thread_fn);

    lock_acquire(&l);
    cond_wait(&cond, &l);
    lock_release(&l);

    printf("%d\n", result);
    return 0;
}
```

Figure 8.12: Comparison between using a semaphore and a condition variable to wait until a thread is finished. The left program is `compare-sema.c` and the right is `compare-cond.c`.

To illustrate the difference, consider the two programs in Fig. 8.12. The program on the left uses a semaphore to wait for the thread running `thread_`

`fn` to set `result` to 10. As you can probably easily conclude by now, the program works as expected. The program on the right attempts to use `cond_wait` and `cond_signal` to do the same thing. As you can immediately see, the approach that uses a condition variable involves more code. More problematic, the program does not always work as expected. Remember that `cond_signal` only wakes the threads that are waiting on the condition variable when `cond_signal` is called. As such, if the thread running `main` reaches `cond_wait` *after* the other thread calls `cond_signal`, the main thread will never wake up.

> **Practice:** Verify the problem in `compare-cond.c` using Progvis. Using *Run* ⇒ *Look for errors...* you can see a visualization of the error. As noted above, `compare-sema.c` does *not* have the same issue, even though the thread running `thread_fn` may still call `sema_up` before the thread running `main` calls `sema_down`. What difference between the two makes `compare-sema.c` work correctly while `compare-cond.c` misbehaves sometimes?

As you have likely noticed from above, the crucial difference between semaphores and condition variables is that condition variables simply do nothing if `cond_signal` is called and there is no thread to wake. As such, if a thread later calls `cond_wake`, it will still have to wait. In contrast, the semaphore remembers that `sema_up` was called by incrementing its internal counter. That way, if a thread later calls `sema_down`, it does not have to wait since the counter is at 1. One way to think of this difference is that a semaphore maintains more state than a condition variable (the semaphore also needs to keep track of the number of threads waiting, it is just not shown in Progvis).

To compensate for the fact that condition variables lack the state that semaphores have, we need to maintain that state ourselves. Since this state will be represented as some form of shared data, we need a lock to protect that data. This is the reason why a condition variable needs to be paired with a lock — we need to maintain some external state to use the condition variable correctly anyway. Maintaining this state manually does of course mean it takes a bit more effort to use condition variables. The great benefit, however, is that we as users of the condition variable get to choose how this additional state is represented and how it is updated. If we use a semaphore, the state has to be a counter that we can only increment and decrement. If we instead use a condition variable, the state can be whatever we want and we can update it however we want! This is what makes condition variables much easier to use than semaphores for certain problems.

8.3 Waiting Until a Condition is True

Now that we know that we have to maintain additional state to use condition variables correctly, the next step is to examine how we maintain the state properly. For once, there is actually a fairly straight-forward method that dictates how to maintain this state. One way to describe this method is to convert a non-synchronized program that utilizes while-loops to wait into a properly synchronized program that uses locks and condition variables. We will use the programs `wait-for-two-`x`.c` to illustrate this process by example, and then summarize it into four concrete steps at the end of the section.

```
int result_a;
int result_b;
int threads_running;

void thread_fn_a(void) {
    result_a = 2;
    threads_running--;
}

void thread_fn_b(void) {
    result_b = 3;
    threads_running--;
}

int main(void) {
    threads_running = 2;

    thread_new(&thread_fn_a);
    thread_new(&thread_fn_b);

    while (threads_running > 0)
        ;

    assert(result_a == 2);
    assert(result_b == 3);
    return 0;
}
```

Listing 8.3: The program `wait-for-two-1.c`.

Listing 8.3 shows the program `wait-for-two-1.c`, which is the initial version of the program. The `main` function launches two threads that run `thread_fn_a` and `thread_fn_b` respectively, and then waits for both of them to complete using a `while` loop. Each of the threads set `result_a` and `result_b` respectively, and tell the main thread that they are done by incrementing `threads_running`. As you can see, the program is currently not synchronized, and since data is shared without properly protecting it, it contains data races. At this point, you are probably confident enough to synchronize the program using semaphores. However, note that doing this requires some creativity, since it is not possible to simply convert `threads_running` to a semaphore and replace `threads_running--` with `sema_down`. As we shall see, using locks and

condition variables does not require altering the program logic at all.

The first step is to identify shared data and the associated critical sections. By examining `wait-for-two-1.c`, we find that the variables `result_a`, `result_b`, and `threads_running` are shared between threads. However, assuming that the logic `main` uses to wait for the two threads to complete works correctly (which is not the case currently, but we will fix it soon), only `threads_running` will actually be accessed concurrently since it ensures that both threads have written to `result_a` and `result_b` by the time the main thread reads from them.

The critical sections for `threads_running` are straight-forward to find. The program contains a total of three critical sections. Two of them are around the two occurrences of the line `threads_running--`. The third and final one surrounds the `while` loop that waits until `threads_running` is zero.

```
int result_a;
int result_b;
int threads_running;
struct lock threads_running_lock;

void thread_fn_a(void) {
    result_a = 2;
    lock_acquire(&threads_running_lock);
    threads_running--;
    lock_release(&threads_running_lock);
}

void thread_fn_b(void) {
    result_b = 3;
    lock_acquire(&threads_running_lock);
    threads_running--;
    lock_release(&threads_running_lock);
}

int main(void) {
    lock_init(&threads_running_lock);
    threads_running = 2;

    thread_new(&thread_fn_a);
    thread_new(&thread_fn_b);

    lock_acquire(&threads_running_lock);
    while (threads_running > 0)
        ;
    lock_release(&threads_running_lock);

    assert(result_a == 2);
    assert(result_b == 3);
    return 0;
}
```

Listing 8.4: The program `wait-for-two-2.c`, where the critical sections are protected with locks.

We then protect the critical sections using locks. Since we only have one variable to protect, we only need one lock. As before, we name the lock `threads_running_lock` to remind ourselves that it protects the variable `threads_running`. Then we simply insert calls to `lock_acquire` and `lock_release` around the three critical sections we found. This results in the code in `wait-for-two-2.c` showed in Listing 8.4.

Interestingly, this version of the program might seem worse than the original version. In particular, if the main thread reaches the `while` loop before `threads_running` is zero, the program will get stuck in the loop. The original version at least had the merit that it *appeared* to work properly even in this case. However, as we saw in Chapter 3 the original program contains data races and may thereby behave as poorly as `wait-for-two-2.c`. The synchronization simply makes this issue readily visible by making the program well-defined.

You can see this issue in Progvis, either by manually stepping the main thread to the `while` loop without touching the other threads. Note that Progvis will not report the error explicitly, but you will note that the main thread gets stuck in the while loop. Progvis does, however, detect and report the error if you select *Run* ⇒ *Look for errors...*. In this case, Progvis identifies the issue as a *livelock*, which is when one or more threads do work but do not make any progress in program execution. When Progvis shows you the cycle, you can click *Close* in the green bar at the top of the window to take control of program execution if you wish to experiment.

The reason for this behavior is, as you might already have realized, that the main thread holds the lock across the entire `while` loop. This means that neither of the two other threads may acquire `threads_running_lock` and thereby they can not modify the `threads_running` variable. This means that the main thread has prevented `threads_running` from changing, even though it is waiting for it to change!

```
lock_acquire(&threads_running_lock);
while (threads_running > 0)
    cond_wait(&threads_running_cond,
              &threads_running_lock);
lock_release(&threads_running_lock);
```

Listing 8.5: Adding `cond_wait` to the loop to avoid livelock, available as `wait-for-two-3.c`.

We can solve this issue using a condition variable. We define a new condition variable next to the lock and name it `threads_running_cond` to remind ourselves that we should signal the condition variable when `threads_running` changes. Once we have a condition variable, we can insert a call to `cond_wait` into the body of the loop that waits for `threads_waiting` to become 0. This makes the loop look like in Listing 8.5 (also in `wait-for-two-3.c`).

This solves the livelock issue we saw before since `cond_wait` releases the lock while the thread is waiting. One way to look at this is to consider `cond_wait` as a pause in the critical section that is delimited by `lock_acquire` and `lock_release`. Since the lock is released, other threads are allowed to acquire the lock and modify `threads_running`. However, we are still not done. If we use *Run* ⇒ *Look for errors...* in Progvis at this time, it will report a deadlock.

What it found was that if the main thread reaches the while loop before both threads are done, it will call `cond_wait`. Since no other thread calls `cond_signal` or `cond_broadcast`, it will never wake up again.

This illustrates that we have forgotten to signal the condition variable when the program modifies `threads_running`. In this case, this happens at the end of the functions executed by the two threads. To solve the issue we can thereby add a call to `cond_signal` or `cond_broadcast` after the line `threads_running--` in both functions. Since `cond_wait` is called in a loop (which is good practice regardless, as we shall see), calling `cond_broadcast` is always a safe option. However, doing so might be wasteful as it wakes *all* threads. If we know that we only need to wake up *one* thread, and it does not matter which one (i.e. we do not care which thread the implementation selects), we can use `cond_signal` instead. In this case, the main thread is the only thread that will wait on the condition variable, so `cond_signal` is sufficient. This gives us the code in `wait-for-two-4.c`, which works correctly.

```
void thread_fn_a(void) {
    result_a = 2;
    lock_acquire(&threads_running_lock);
    threads_running--;
    cond_signal(&threads_running_cond,
                &threads_running_lock);
    lock_release(&threads_running_lock);
}
```

Listing 8.6: Adding `cond_signal` to wake the main thread. Available as `wait-for-two-4.c`.

Note that we want to call `cond_signal` or `cond_broadcast` right after we have modified one or more variables that some thread uses the condition to wait for. Since the variables are shared, they need to be protected by the same lock that protects the condition variable. Because of this, the thread that calls `cond_signal` or `cond_broadcast` will always hold the lock just before the call anyway. As such, it is a good idea to keep holding the lock during the call to `cond_signal` and `cond_broadcast` (and required in the library provided with this book), since it means that no other thread will invalidate our assumptions before signaling the condition.

Practice: In `wait-for-two-4.c` we have retained the `while` loop around the call to `cond_wait`. This might seem unnecessary since it means that the main thread will check the condition again after waking up. We can avoid this by replacing the `while` loop with an `if` statement as shown below and in `wait-for-two-error.c`. This will, however, cause the program to fail. Why?

```
lock_acquire(&threads_running_lock);
if (threads_running > 0)
    cond_wait(&threads_running_cond,
              &threads_running_lock);
lock_release(&threads_running_lock);
```

You can use *Run* ⇒ *Look for errors...* to find an interleaving that illustrates the error. Is there another way in which we can solve the issue in `wait-for-two-error.c`?

As you have seen from above, the issue is that the two threads always calls `cond_signal` when they have decremented `threads_running`. This means that the first one will decrease `threads_running` to 1 and then call `cond_signal`. If the main thread has reached `cond_wait` at this time, this causes it to wake up even though `threads_running` is not yet zero. This is sometimes called a *spurious wakeup*, since the thread was woken even though the condition was not yet fulfilled. If we use an `if` statement instead of a `while` loop, the main thread will simply continue its execution when this happens. If you try the same interleaving in `wait-for-two-4.c` in Progvis,[4] you will see that the main thread wakes up, but rather than continuing it checks the condition again, realizes that it is not yet time to continue, and goes back to sleep by calling `cond_wait` again.

In this case, we could avoid the issue altogether by only calling `cond_signal` when `threads_running` reaches zero. However, this means that we need to duplicate the condition that the main thread is waiting for, which makes it more difficult to change the condition in the future. As such, this is only possible to do if a condition variable is used to wait for *one* thing. Furthermore, not calling `cond_wait` in a loop also means that it is not always safe to call `cond_broadcast` rather than `cond_signal` to wake threads, which is not always possible,[5] but also breaks the expectation that it is safe to replace `cond_signal` with `cond_broadcast` without affecting the correctness of the program. Furthermore, evaluating a condition is typically very cheap in comparison to waking a thread anyway, so the cost of the loop is often negligible. For all of these reasons, it is good practice to always call `cond_wait` in a loop. In fact, condition variables in C++ have a member function that checks the condition (provided as a lambda function) in a loop for you, since the pattern is so common.

For the curious reader, the file `wait-for-two-duplicated.c` contains an implementation that avoids waking the main thread more than once, and

[4] That is: (1) step the main thread until it reaches `cond_wait`, (2) let one of the other threads finish, and (3) continue stepping the main thread.

[5] For example, it is not possible to wake *a particular* thread without waking all of them and have the threads themselves figure out which one should continue

thereby does not need the loop around `cond_wait`. However, as mentioned above and by the comment in the program, note that this type of optimization is often only necessary in "hot code", that is known to be the bottleneck of a concurrent program.

8.3.1 Summary of the Method

The method above is actually general enough to be applicable to arbitrary problems. As such, we can summarize what we did above into four general steps that we can use to turn a non-synchronized program that utilizes loops to wait into a properly synchronized program that uses locks and condition variables.

1. Examine the program and identify (1) the shared data, (2) the critical sections that manipulate the shared data, and (3) loops that wait for a condition to become true. Note that the condition will involve shared data, and thereby examine the state of the shared data.

2. Use one or more locks to protect the critical sections. Note that all data that are used in the same conditions need to be protected by the same lock (this will likely be the result anyway, but it is worth considering as it may help the implementation).

3. Add a condition variable for each set of variables that the program is waiting for, and call `cond_wait` in the loop that waits until the condition becomes true. Note that the call to `cond_wait` needs to be a part of the same critical section that checks the condition.

4. Find all places where the program changes variables associated with each condition. Call `cond_broadcast` or `cond_signal` after they are updated, but before the lock is released. Using `cond_broadcast` is always safe, but may wake up too many threads. If only one thread needs to wake, and the implementation does not care *which* out of all threads that wait on the same condition, then `cond_signal` can be used.

8.4 A More Complex Condition

To illustrate some more nuances with the method above, we will next look at the program `seats-`x`.c`, which contains a more complex condition. The first version of the program (`seats-1.c` and Listing 8.7) is again not synchronized at all. The program manages a number of seats (e.g. seats at a cinema). The variable `seat_taken` is an array with `num_seats` elements that stores whether each seat is currently occupied or not. The function `seat_acquire` searches the `seat_taken` array to find a seat that is not occupied, marks it as taken, and returns the index of the seat. If no seat is available, the function waits until one becomes available. Once the user of the seat is finished, they are expected to call `seat_release` to mark it as free again.

The program additionally contains a variable `seat_used_count` that the `thread_fn` (Listing 8.8) function is used. Once it has acquired a seat, it increases the value for that seat, to keep track of how many times each seat has been used. Since `seat_acquire` and `seat_release` have semantics similar to

a lock, `thread_fn` uses them to manage access to the used count for the seats, and this variable therefore does not need additional protection. You may have noted that this program essentially implements a lock that manages access to multiple instances of a resource that are treated as equivalent as discussed in Section 7.1.2.

The rest of the code in the main program initializes the arrays to contain 2 seats that are both initially not taken. It then starts two threads that both call `thread_fn` and then calls `thread_fn` itself. As such, a total of three threads will attempt to acquire a seat concurrently, which means that one of them will have to wait. After the call to `thread_fn`, the main thread waits for all threads to finish and verifies that the two elements in `seat_used_count` sums to 3.

```
bool *seat_taken;
int *seat_used_count;
int num_seats;

int seat_acquire(void) {
    int taken = -1;
    while (taken == -1) {
        for (int i = 0; i < num_seats; i++) {
            if (seat_taken[i] == false) {
                seat_taken[i] = true;
                taken = i;
                break;
            }
        }
    }
    return taken;
}

void seat_release(int id) {
    seat_taken[id] = false;
}
```

Listing 8.7: The initial version of the `seats` program.

```
    int seat = seat_acquire();
    seat_used_count[seat]++;
    seat_release(seat);
```

Listing 8.8: The code that uses `seat_acquire` and `seat_release` in `seats-1.c`. Note that they have similar semantics to a lock.

Even though we can formulate what `seat_acquire` is waiting for (i.e. it is waiting for one seat to be available), the condition is perhaps not immediately apparent in Listing 8.7. To help our understanding, we start by refactoring the code as shown in Listing 8.9. In particular, we move the `for` loop in `seat_acquire` into its own function that we call `grab_seat`. We mark the function as `static` to remind ourselves that the function is not supposed to be called outside of `seat_acquire`.

```
static bool grab_seat(int *taken) {
    for (int i = 0; i < num_seats; i++) {
        if (seat_taken[i] == false) {
            seat_taken[i] = true;
            *taken = i;
            return true;
        }
    }
    return false;
}

int seat_acquire(void) {
    int taken = -1;
    while (!grab_seat(&taken))
        ;
    return taken;
}

void seat_release(int id) {
    seat_taken[id] = false;
}
```

Listing 8.9: Refactoring of the `seats` program to make it easier to see what the program is waiting for.

By moving the `for` loop into `grab_seat` the condition that the program waits for becomes clearer, since we can put the function call in the header of the `while` loop inside `seat_acquire`. Note that we do not need to refactor the code in this way to apply condition variables. It just makes it easier to see what is happening, and helps making the discussion in this section clearer. We will provide a non-refactored version of the solution at the end of the section in addition to the refactored version.

Now that we have refactored the code, we can start apply the steps from Section 8.3.1. The first steps asks us to identify (1) the shared data, (2) the critical sections, and (3) loops that wait for a condition to become true. Our answer to (1) is that the shared data is the elements of the `seat_taken` array, and that (2) they need to be protected in the if-statement in `grab_seat` as well as in `seat_release`. Finally, (3) the program waits for at least one seat to be available in the `while` loop in `seat_acquire`.

Using this information, we can then move on to step 2 from Section 8.3.1 and protect the shared data using locks. Since our initial impression is that seats can be treated separately, we define an array of locks so that we can use one lock for each individual seat. This gives us the program `seats-3.c` (Listing 8.10).[6]

[6]You might have noticed that this disregards an important detail. This is intentional to illustrate *why* this detail is important. We will get back to it!

```
static bool grab_seat(int *taken) {
    for (int i = 0; i < num_seats; i++) {
        lock_acquire(&seat_taken_lock[i]);
        bool take = seat_taken[i] == false;
        if (take) {
            seat_taken[i] = true;
            *taken = i;
        }
        lock_release(&seat_taken_lock[i]);
        if (take)
            return true;
    }
    return false;
}

int seat_acquire(void) {
    int taken = -1;
    while (!grab_seat(&taken))
        ;
    return taken;
}

void seat_release(int id) {
    lock_acquire(&seat_taken_lock[id]);
    seat_taken[id] = false;
    lock_release(&seat_taken_lock[id]);
}
```

Listing 8.10: Protecting the critical sections using an array of locks.

Practice: As we would expect, the program does not yet work as intended since we still have 2 more steps to follow. However, if you run the program (either in the terminal or in Progvis) you will see that the program actually behaves correctly at this point. In particular, the third call to `seat_create` does not cause a deadlock even if all seats are occupied.

Investigate the behavior of the program using Progvis to understand *why* `seats-3.c` does not deadlock even though `wait-for-two-2.c` caused a deadlock in the same situation.

The program is not correct, however. If you select *Run* ⇒ *Look for errors...* in Progvis, it will tell you that the program uses *busy wait* (it needs to investigate around 54 000 transitions to reach this conclusion, so it takes a while). What is the problem?

As you have noted above, the current version of the program surprisingly works as expected, even though the same stage of the previous program (`wait-for-two-2.c`) deadlocked in certain situations. The reason for this is that `seats-3.c` does *not* hold a lock throughout the *entire* loop it uses to wait for a seat to become free. Since it makes sure to release the lock periodically, other threads are still able to release their seats. This was not the case in `wait-for-two-2.c`, which did not release the lock inside the loop. We could cause `wait-for-two-2.c`

to behave the same way by inserting `lock_release` directly followed by `lock_acquire` in the body of the `while` loop.

Even though the program does work as expected, it is of course not ideal. As Progvis reports, the program uses *busy wait* to wait for the condition to become true. In this case, if a thread finds that no seats are available, it will keep checking for an available seat until one becomes available, thereby essentially wasting CPU time to do nothing. While this is fine to do for a short amount of time,[7] it is problematic if it is done for a longer time or at multiple places in the program. Since the scheduler does not know which threads do useful work and which threads are just wasting CPU time to wait, it is unable to prioritize the threads that actually do useful work. As such, threads that busy wait for a considerable amount of time reduce the available CPU time for the threads that do useful work,[8] which means that it will take longer for them to finish their work. Because of this, it is good practice to use synchronization primitives to wait. They contain logic to inform the scheduler that the thread is waiting, which allows it to not waste CPU time in this way.

To fix this issue, we move to step 3 from Section 8.3.1 and add a condition variable to help us wait efficiently. We would like to call `cond_wait` in the `while` loop in `seat_acquire`. However, this is not currently possible as we do not hold a lock in the while loop. We also need a lock to protect the condition variable itself.[9] One way to solve the issue is to add a separate lock that protects the condition variable and allows us to call `cond_wait`. We call the condition variable `seat_free_cond` since we wait for a seat to become free, and we call the new lock `seat_free_lock` to help us associate the two together. With these additions we can insert the call to `cond_wait` in `seat_acquire`. While we are at it, we can insert a call to `cond_signal` in `seat_release` as instructed by step 4 as well. This gives us the code in `seats-4.c` (Listing 8.11).

Practice: The program `seats-4.c` is still not correct. There are situations that cause the program to deadlock. Use Progvis to find the issue. If you use *Run* ⇒ *Look for errors...*, note that it needs to investigate around 100 000 transitions to find the issue, so checking will take some time.

[7] For example, one implementation of a lock is a *spinlock*, which utilizes busy wait. It is, however, best suited for situations where we know that threads never have to wait for a significant amount of time.

[8] Or prevent the CPU from becoming idle and thereby consume less energy.

[9] This is of course an indication that we have done something wrong, as we shall see.

```
int seat_acquire(void) {
    int taken = -1;
    while (!grab_seat(&taken)) {
        lock_acquire(&seat_free_lock);
        cond_wait(&seat_free_cond, &seat_free_lock);
        lock_release(&seat_free_lock);
    }
    return taken;
}

void seat_release(int id) {
    lock_acquire(&seat_taken_lock[id]);
    seat_taken[id] = false;
    lock_release(&seat_taken_lock[id]);
    lock_acquire(&seat_free_lock);
    cond_signal(&seat_free_cond, &seat_free_lock);
    lock_release(&seat_free_lock);
}
```

Listing 8.11: The `seat_acquire` and `seat_release` functions after adding a condition variable. Available as `seats-4.c`.

As you have probably noted, it is possible for the program to deadlock since `cond_wait` is in a *different* critical section to the code that checks if any seats are available. Since the program does not hold any locks at the end of `grab_seat`, a seat may become available after `grab_seat` returns but before `cond_wait` is called. In particular, assume that all seats are occupied an thread A calls `seat_acquire`. Thread A calls `grab_seat` which returns `false` since no seats are available. Before it acquires `seat_free_lock`, thread B calls `seat_release` which marks one of the seats as available and signals the condition variable. When Thread A continues again, it acquires the lock and calls `cond_wait`. However, since it called `cond_wait` while a seat was available, the condition variable will not be signaled again, and the thread may have to wait forever.

This observation shows that it is necessary to treat condition variables as any other variables when we look for critical sections and consider which locks to use when protecting shared data. As we saw above, `cond_wait` needs to be called in the same critical section where we verified that the condition was false. The same is often true for calls to `cond_signal` and `cond_broadcast`,[10] even if they are forgiving since an extra wakeup rarely hurts correctness, only performance. This is the reason why we want to treat condition variables *as if* they are not thread-safe, and protect them with a particular lock. This also shows that we should treat the condition variable as a part of the condition we wish to wait for, which is the reason why step 2 in Section 8.3.1 states that "Note that all data that are used in the same conditions need to be protected by the same lock."

[10] In particular, it is only important if we do not always call `cond_signal` or `cond_broadcast`. The logic that determines if we need to signal the condition variable likely depends on shared state, and if we break the critical section, other threads may have modified the state before we inspect it.

As such, since we have one condition variable that waits for one of multiple seats to become available, we need to revise our synchronization to use a single lock for all seats, rather than separate locks for each seat. This way we can use the same lock to protect the condition variable and all data involved in checking the condition. In a way, this is similar to what happened when we added `accounts_total` to the `bank` program in Section 6.4.2. Since `accounts_total` required stronger guarantees from the rest of the program, it required us to revise our synchronization approach. Here, the requirement existed from the start but it was not immediately obvious until we added the condition variable.

To fix the problem, we make `seat_taken_lock` into a single lock instead of an array and remove `seat_free_lock` since it is no longer necessary. We then revise the critical sections in `seat_acquire` with the assumption that we need to have a consistent view of *all* seats, not just individual seats at a time. If we do this, we will end up with the program in `seats-5.c` (Listing 8.12), which works as intended.

```
bool *seat_taken;
int *seat_used_count;
int num_seats;
struct lock seat_taken_lock;
struct condition seat_free_cond;

static bool grab_seat(int *taken) {
    for (int i = 0; i < num_seats; i++) {
        if (seat_taken[i] == false) {
            seat_taken[i] = true;
            *taken = i;
            return true;
        }
    }
    return false;
}

int seat_acquire(void) {
    int taken = -1;
    lock_acquire(&seat_taken_lock);
    while (!grab_seat(&taken))
        cond_wait(&seat_free_cond, &seat_taken_lock);
    lock_release(&seat_taken_lock);
    return taken;
}

void seat_release(int id) {
    lock_acquire(&seat_taken_lock);
    seat_taken[id] = false;
    cond_signal(&seat_free_cond, &seat_taken_lock);
    lock_release(&seat_taken_lock);
}
```

Listing 8.12: Correct synchronization of the `seats` program. Available as `seats-5.c`.

As mentioned previously in the section, it is not necessary to refactor the program to add condition variables. The refactoring does, however, make it easier to see what part of the code is the condition, and what part is the loop that waits for the condition to become true. For reference, a non-refactored version of the program is available as `seats-no-refactor.c` and the relevant parts are shown in Listing 8.13. Note that it is significantly harder to follow the control flow of the program to mentally verify that it is correct.

```
int seat_acquire(void) {
    int taken = -1;
    lock_acquire(&seat_taken_lock);
    while (taken == -1) {
        for (int i = 0; i < num_seats; i++) {
            if (seat_taken[i] == false) {
                seat_taken[i] = true;
                taken = i;
                break;
            }
        }
        if (taken == -1)
            cond_wait(&seat_free_cond, &seat_taken_lock);
    }
    lock_release(&seat_taken_lock);
    return taken;
}
```

Listing 8.13: Correct synchronization of the non-refactored version of the `seats` program. Available as `seats-no-refactor.c`.

8.5 Exercises

1. The file `bank.c` contains the final version of the bank program used in Chapters 6 and 7. One shortcoming of the program is that `accounts_total` can not be called from multiple threads concurrently, even though there is no problem in doing so since it only reads from the accounts. Modify the code so that both (1) multiple threads can call `transfer` concurrently and (2) multiple threads can call `accounts_total` concurrently. However, note that `accounts_total` can not be executed concurrently with `transfer`, so your solution must make threads wait for each other if that is the case.

   You can verify that your solution behaves correctly using *Run* ⇒ *Look for errors...*. This does not verify that `transfer` and `accounts_total` can be executed concurrently. You need to do that manually. Note that the sample solution needs to explore around 170 000 transitions, so verification takes a while, especially if you solution is more complex than the sample solution.

Chapter 9

Abstractions and Concurrency

- Speak about different things to think about.
- What does *thread-safe* mean?
- What *invariants* are important?
- What expectations are common? Which are not achievable?
- ...
- Some fun example where we use multiple data structures that acquire locks, so that we get a deadlock at a "large scale".

9.1 Exercises

- In Section 7.1.2, we mention "it is possible to build a lock that acquires one out of many available resources". Implement such a synchronization primitive! If we have it, make a reference to it from Section 7.1.2. It is also discussed in Section 8.4, so we could focus on modularization of the solution here.
- Another interesting idea is to build a "lock" that acquires multiple "locks" at the same time. Perhaps better suited for condition variables.

Chapter 10

Atomics

- Remember to cover things like busy-waiting and livelocks (= lack of *progress*), and also the requirements on locks/semaphores/etc (i.e. mutual exclusion, progress, bounded waiting). These may come naturally when speaking about condition variables?

10.1 Basics

10.2 Advanced Usage

Advanced usage (e.g. acquire, release semantics, etc.).

Appendix A

Threading Library Reference

This chapter contains reference to all threading-related functions introduced in this book. All of the functions are introduced in more detail where they are first used. The goal of this chapter is therefore to list all functions in one place to make them easier to find.

The functionality described here is available both in Progvis and in C through the threading library provided with this book. The names and semantics of the synchronization primitives are inspired from their appearence in Pintos.

A.1 Thread Management

```
#include "thread.h"
```

These functions and types are defined in the file `thread.h`.

`tid_t`

A type that is used to store thread identifiers. A thread identifier may be any type, so it is only safe to assume that it is possible to compare two `tid_t` values with the `==` and `!=` operators. For the purposes of this book you can mostly ignore this type. The book mostly uses `thread_new` below to start threads. It is seldom necessary to keep track of threads by their identifier.

```
tid_t thread_new(thread_func *fn, void *data, ...);
```

Creates a new thread. The new thread will execute the function passed as `fn`. Note that it is not well-defined *when* the new thread will start execute `fn` in relation to when the call to `thread_new` returns. The function returns the thread-id of the newly created thread.

For convenience, this function is implemented specially to allow a variable number of parameters to be passed to the function in the thread. As such, you will not find the above definition in the header files in the threading library.

The function can be used as follows:

```
void zero(void);
void one(int *x);
void two(int *x, struct my_data *y);

int main(void) {
    int a = /* ... */;
    struct my_data b = /* ... */;

    thread_new(&zero);
    thread_new(&one, &a);
    thread_new(&two, &a, &b);

    return 0;
}
```

Note that `thread_new` accepts a variable number of parameters. However, the implementation requires that all parameters are pointers. As such, passing `b` by value to a function would fail.

```
tid_t thread_current(void);
```

Retrieve the identifier of the current thread.

```
void thread_yield(void);
```

Asks the scheduler to let other threads run for a while. This is just a hint, and it is entirely up to the scheduler to determine which other threads to run and for how long.

```
void timer_msleep(int ms);
```

Pause execution of the current thread for *at least* the specified number of milliseconds. Note that it is acceptable for the implementation to wait for *more* than the specified time.

```
void prevent_optimization(void);
```

A utility function that does nothing but preventing the compiler from performing certain kinds of optimizations. If you look at the function you will see that it is empty. The reason it works is that it is implemented in another *translation unit*, which means that the compiler cannot see what the implementation actually does, and therefore has to make pessimistic assumptions.[1] This is not a general way of preventing optimizations that break programs. It is only used in this book to force the compiler to take certain decisions and illustrate certain problems. It is not a replacement for proper synchronization.

[1] Unless link-time optimizations are used.

Progvis-specific Notes:

- `thread_new` is implemented as a special form. It is thereby possible to pass an arbitrary number of parameters to functions in Progvis. Progivis is also not restricted to pointer types, but can pass arbitrary data types. This also means that the calls are properly type-checked. However, be aware of the limitations in the C version noted above if you aim to write portable code!
- `tid_t` is implemented as a special type. It is only possible to compare them to other `tid_t` variables.
- `thread_yield` and `timer_msleep` have no effect at all. These functions are used to make certain problematic interleavings more likely to occur when running programs outside of Progvis. Since Progvis allows single stepping anyway, they provide no benefit inside Progvis.

C-specific Notes:

- `thread_new` supports up to 5 parameters. Passing more than 5 parameters will fail with an error message. Also note that parameters are not type-checked since all parameters are cast to `void *`. Make sure that the parameter count and types passed to `thread_new` match the parameter count and types accepted by the function!

The C version also contains the following functions that might be useful to force interesting thread behaviors:

```
void thread_exit(void);
```

Immediately terminates the current thread. This is not used in the book, as we can simply return from the entry-point of the thread instead.

A.2 Synchronization Primitives

```
#include "synch.h"
```

These functions and types are defined in the file `synch.h`.

```
struct semaphore;
```

Datatype that represents a semaphore. The datatype is fully defined in the header file, so it can be stack-allocated and used inside other data structures. Even though the data structure is available in the header file, it should be considered as a "black box", and it should only be accessed through the functions below. Note that it is generally not safe to copy a semaphore once it has been initialized, so make sure to pass the semaphore or the data structure that contains it by pointer.

```
void sema_init(struct semaphore *s, int value);
```

Initialize the semaphore `s` with a counter of `value`. Assumes that `value` ≥ 0.

```
void sema_down(struct semaphore *s);
```

Try to decrement the counter in the semaphore `s`. If the counter would become negative, the calling thread waits until another thread calls `sema_up` and decrements the counter when it wakes up.

```
void sema_up(struct semaphore *s);
```

Increments the counter inside the semaphore `s`. This may wake one thread that is waiting inside a call to `sema_down`. Note that it is not well-defined *which* thread wakes up if multiple threads are waiting.

```
void sema_destroy(struct semaphore *s);
```

Deallocates any resources allocated by `sema_init`. This is not crucial to proper operation of the program, but leads to resource leaks on some platforms. Progvis does not consider it essential to deallocate semaphores, so this book considers it to be optional to destroy semaphores.

```
struct lock;
```

Datatype that represents a lock. The datatype is fully defined in the header file, so it can be stack-allocated and used inside other data structures. The datatype is fully defined in the header file, so it can be stack-allocated and used inside other data structures. Even though the data structure is available in the header file, it should be considered as a "black box", and it should only be accessed through the functions below. Note that it is generally not safe to copy a semaphore once it has been initialized, so make sure to pass the semaphore or the data structure that contains it by pointer.

```
void lock_init(struct lock *l);
```

Initialize the lock `l`.

```
void lock_acquire(struct lock *l);
```

Try to acquire the lock `l`. If another thread currently holds the lock, the calling thread will wait until the lock is released.

```
void lock_release(struct lock *l);
```

Release the lock `l`. Assumes that the current thread currently holds the lock (i.e. the current thread has previously called `lock_acquire` for the same lock). If one or more threads are waiting to acquire the lock, this will cause one of them to wake up eventually. Note that it is not well-defined *which* thread wakes up.

```
void lock_destroy(struct lock *l);
```

Deallocates any resources allocated by `lock_init`. This is not crucial to proper operation of the program, but leads to resource leaks on some platforms. Progvis does not consider it essential to deallocate semaphores, so this book considers it to be optional to destroy semaphores.

```
struct condition;
```

Datastructure that represents a condition variable. The datatype is fully defined in the header file, so it can be stack-allocated and used inside other data structures. Even though the data structure is available in the header file, it should be considered as a "black box", and it should only be accessed through the functions below. Note that it is generally not safe to copy a semaphore once it has been initialized, so make sure to pass the semaphore or the data structure that contains it by pointer.

This implementation of condition variables is *not* thread-safe in and of itself (in contrast to e.g. pthreads). As such, each condition variable needs to be protected with *one* lock. As such, the implementation expects that this lock is held when `cond_wait`, `cond_signal`, or `cond_broadcast` is called, and that the lock is passed as the parameter `l`.

```
void cond_init(struct condition *c);
```

Initialize the condition variable `c`.

```
void cond_wait(struct condition *c, struct lock *l);
```

Releases the lock `l` and causes the current thread to wait until another thread calls `cond_signal` or `cond_broadcast`. When the thread resumes, it acquires `l` before returning from `cond_wait`.

```
void cond_signal(struct condition *c, struct lock *l);
```

Causes *one* of the threads that are currently waiting inside a call to `cond_wait` for the condition variable `c` to wake up. Nothing happens if no threads are currently waiting for `c`. Note that it is not well-defined *which* thread wakes up. Assumes that `l` is held.

```
void cond_broadcast(struct condition *c, struct lock *l);
```

Causes *all* threads that are currently waiting inside a call to `cond_wait` for the condition variable `c` to wake up. Nothing happens if no threads are currently waiting for `c`. Assumes that `l` is held.

```
void cond_destroy(struct condition *c);
```

Deallocates any resources allocated by `cond_init`. This is not crucial to proper operation of the program, but leads to resource leaks on some platforms. Progvis does not consider it essential to deallocate semaphores, so this book considers it to be optional to destroy semaphores.

A.3 Atomic Operations

The atomic operations below are generic in the sense that they are able to operate on any integer-like type (i.e. integers and pointers). The functions are listed alongside with their semantics, expressed as a regular function. Do remember, however, that the actual implementation uses *compiler intrinsics* to let the compiler and hardware know that the operations should be atomic.

```
#include "atomics.h"
```

These functions and types are defined in the file `atomics.h`.

```
T atomic_read(T *value) {
    return *value;
}
```

Read the value pointed to by `value` atomically.

```
void atomic_write(T *value, T write) {
    *value = write;
}
```

Write the value passed to `write` to the location pointed to by `value` atomically.

```
int test_and_set(int *value) {
    int old = *value;
    *value = 1;
    return old;
}
```

Read the value pointed to by `value` and set it to 1. Return the old value pointed to by `value`.

```
int atomic_swap(int *value, int replace) {
    int old = *value;
    *value = replace;
    return old;
}
```

Read the value pointed to by `value` and swap it the value passed as `swap`. Return the old value pointed to by `value`. Note that only the contents of `value` is accessed atomically.

```
int compare_and_swap(int *value, int compare, int swap) {
    int old = *value;
    if (old == compare)
        *value = swap;
    return old;
}
```

Read the value pointed to by `value`. Compare it to the value passed as `compare`. If they were equal, replace the value pointed to by `value` with the value passed as `swap`. Regardless of the check, return the old value pointed to by `value`. Note that only the contents of `value` is accessed atomically.

```
int atomic_add(int *value, int add) {
    int old = *value;
    *value += add;
    return old;
}
```

Read the value pointed to by `value`, add one to it, and return the original value.

```
int atomic_sub(int *value, int add) {
    int old = *value;
    *value -= add;
    return old;
}
```

Read the value pointed to by `value`, subtract one from it, and return the original value.

```
void atomics_busy_wait();
```

Function used to mark loops that use atomic operations to wait for something. By default, the option *Run* ⇒ *Look for errors...* looks for places in the code where busy-wait may occur. Since there is no way to wait "properly" using only atomic operations, we need to tell Progvis that we are aware that some particular loop contains a busy wait, and that we do not think it is a problem. By calling `atomics_busy_wait` inside such loops we inform Progvis about this, and thereby it will ignore that particular busy wait.

Appendix B

Limitations in Progvis

TODO: Outline the limitations of Progvis
In particular:

- No `void` pointers!
- No type casts
- Includes are not parsed – just open all files you need
- A `return` statement *is* needed inside `main`
- No initializers for structs