

---

## Concurrency: An Introduction

Thus far, we have seen the development of the basic abstractions that the OS performs. We have seen how to take a single physical CPU and turn it into multiple **virtual CPUs**, thus enabling the illusion of multiple programs running at the same time. We have also seen how to create the illusion of a large, private **virtual memory** for each process; this abstraction of the **address space** enables each program to behave as if it has its own memory when indeed the OS is secretly multiplexing the address spaces across physical memory (and sometimes, across the disk). Finally, we have seen the abstractions of the file and the directory, virtualizing the disk to give users a way to organize and share data that they want to keep persistent.

In this note, we introduce a new abstraction for a single running process: that of a **thread**. Instead of our classic view of a single point of execution within a program (i.e., a single PC where instructions are being fetched from and executed), a **multi-threaded** program has more than one point of execution (i.e., multiple PCs, each of which is being fetched and executed from). Perhaps another way to think of this is that each thread is very much like a separate process, except for one major difference: they *share* the same address space and thus can access the same data.

The state of a single thread is thus very similar to that of a

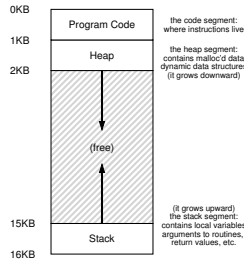


Figure 18.1: A Single-Threaded Address Space

process. It has a program counter (PC) that tracks where the program is fetching instructions from. Each thread has its own private set of registers it uses for computation; thus, if there are two threads that are running on a single processor, when switching from running one (T1) to running the other (T2), a **context switch** must take place. The context switch between threads is quite similar to the context switch between processes, as the register of T1 must be saved and the register state of T2 restored before running T2. With processes, we saved state to a **process control block (PCB)**; now, we'll need one or more **thread control blocks (TCBs)** to store the state of each thread of a process. There is one major difference, though, in the context switch we perform between threads as compared to processes: the address space remains the same (i.e., there is no need to switch which page table we are using).

One other major difference between threads and processes concerns the stack. In our simple model of the address space of a classic process (which we can now call a **single-threaded process**), there is a single stack, perhaps residing at the bottom of the address space (Figure 18.1).

However, in a multi-threaded process, each thread runs independently and of course may call into various routines to do

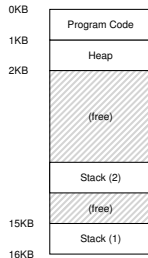


Figure 18.2: A Multi-Threaded Address Space

whatever work it is doing. Thus, instead of a single stack within the address space, there will be one for each thread. Let's say we have a multi-threaded process that has four threads in it. In such a case, our address space looks different (Figure 18.2).

In this figure, you can see two stacks spread throughout the address space of the process. Thus, any stack-allocated variables, parameters, return values, and other things that we put on the stack will be placed in what is sometimes called **thread-local** storage, i.e., the stack of the relevant thread.

You might also notice how this ruins our beautiful address space layout. Before, the stack and heap could grow independently and trouble only arose when you ran out of room in the address space. Here, we no longer have such a nice situation. Fortunately, this is usually OK, as stacks do not generally have to be very large (the exception being in programs that make heavy use of recursion).

## 18.1 An Example: Thread Creation

Let's say we wanted to run a program that created two threads, each of which was doing some independent work, in this case

```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *
mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    int rc;

    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);

    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: done with both\n");
    return 0;
}
```

Figure 18.3: Simple Thread Creation Code

printing "A" or "B". The code might look something like what you see in Figure 18.3.

The main program creates two threads, one of which will start running at function `t1`, and the other at function `t2`. Once a thread is created, it may start running (depending on the whims of the scheduler); alternately, it may be put in a "ready" but not "running" state and thus not run yet.

After creating the two threads `t1` and `t2`, the main thread calls `pthread_join()`, which waits for a particular thread to complete.

Let us examine the possible execution ordering of this little program. In the diagram, time is moving downwards.

```
main starts running
main prints "main: begin"
main creates thread 1
main creates thread 2
main waits for thread 1
                                thread 1 runs
                                thread 1 prints "A"
                                thread 1 returns
main waits for thread 2
                                thread 2 runs
                                thread 2 prints "B"
                                thread 2 returns
main prints "main: end"
```

Note, however, that this is not the only possible ordering. In fact, there are many, depending on what the scheduler decides to run at a given point. For example, once a thread is created, it may run immediately, which would lead to the following execution:

```
main starts running
main prints "main: begin"
main creates thread 1
                                thread 1 runs
                                thread 1 prints "A"
                                thread 1 returns
main creates thread 2
                                thread 2 runs
                                thread 2 prints "B"
                                thread 2 returns
main waits for thread 1
  (returns immediately because thread 1 is finished)
main waits for thread 2
  (returns immediately because thread 2 is finished)
main prints "main: end"
```

We also could even see "B" printed before "A", if, say, the scheduler decided to run it first even though thread 1 was created first. This final execution ordering is shown here:

```

main starts running
main prints "main: begin"
main creates thread 1
main creates thread 2

thread 2 runs
thread 2 prints "B"
thread 2 returns

main waits for thread 1

thread 1 runs
thread 1 prints "A"
thread 1 returns

main waits for thread 2
  (returns immediately because thread 2 is finished)
main prints "main: end"

```

As you might be able to see, one way to think about thread creation is that it is a bit like making a function call; however, instead of first executing the function and then returning to the caller, the system instead creates a new thread of execution for the routine that is being called, and it runs independently of the caller, perhaps before returning from the create, but perhaps much later.

As you also might be able to tell from this example, threads make life complicated: it is already hard to tell what will run when! Unfortunately, it gets worse. Much worse.

## 18.2 Why It Gets Worse: Shared Data

The simple thread example we showed above was useful in showing how threads are created and how they can run in different orders depending on how the scheduler decides to run them. What it doesn't show you, though, is how threads interact when they access shared data.

Let us imagine a simple example where two thread wish to update a global shared variable. The code might look something like what is in Figure 18.4.

A few notes about the code. First, as Stevens suggests [S92], we wrap the thread creation and join routines to simply exit on failure; for a program as simple as this one, we want to at least

```
#include <stdio.h>
#include <pthread.h>
#include "mythreads.h"

static volatile int balance = 0;

void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        balance = balance + 1;
    }
    printf("%s: done\n", letter);
    return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (balance = %d)\n", balance);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (balance = %d)\n", balance);
    return 0;
}
```

Figure 18.4: Sharing Data: Oh Oh

notice an error occurred (if it did), but not do anything very smart about it (e.g., just exit). Thus, `Pthread_create()` simply calls `pthread_create()` and makes sure the return code is 0; if it isn't, `Pthread_create()` just prints a message and exits.

Second, you can see that instead of using two separate function bodies for the worker threads, we just use a single piece of

code, and pass the thread an argument (in this case, a string) so we can have each thread print a different letter before its messages.

Finally, and most importantly, we can now look at what each worker is trying to do: add a number to the shared variable `balance`, and do so 10 million times ( $1e7$ ) in a loop. Thus, the desired final result is: 20,000,000, as we see in this potential output:

```
prompt> gcc -o main main.c -Wall -lpthread
prompt> ./main
main: begin (balance = 0)
A: begin
B: begin
A: done
B: done
main: done with both (balance = 20000000)
prompt>
```

Unfortunately, when we run this code, even on a single processor, we don't necessarily get the desired result:

```
prompt> gcc -o main main.c -Wall -lpthread
prompt> ./main
main: begin (balance = 0)
A: begin
B: begin
A: done
B: done
main: done with both (balance = 19345221)
prompt> ./main
main: begin (balance = 0)
A: begin
B: begin
A: done
B: done
main: done with both (balance = 19221041)
prompt>
```

In fact, each run yields a *different* result! A big question remains: why does this happen?

### 18.3 The Heart of the Problem: Uncontrolled Scheduling

To understand why this happens, we must understand the code sequence that the compiler generates for the update to `balance`. In this case, we wish to simply add a number (1) to `balance`. Thus, the code sequence for doing so might look something like this (an x86 example here):

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

This example assumes that the variable `balance` is located at address `0x8049a1c`. In this three-instruction sequence, the x86 `mov` instruction is used first to get the memory value at the address and put it into register `eax`. Then, the `add` is performed, adding 1 (`0x1`) to the contents of the `eax` register, and finally, the contents of `eax` are stored back into memory at the same address.

Let us imagine one of our two threads (thread 1) enters this region of code, and is thus about to increment `balance` by one. It loads the value of `balance` (let's say it is zero to begin with) into its register `eax`. Thus, `eax=0` for thread 1. Now, something unfortunate happens: a timer interrupt goes off. This causes the OS to save the state of the currently running thread (its PC, its registers including `eax`, etc.) to the TCB for this thread.

Now something worse happens: thread 2 is chosen to run, and it enters this same piece of code. It also executes the first instruction, getting the value of `balance` and putting it into its `eax` (remember: each thread when running has its own private registers; the registers are **virtualized** by the context-switch code that saves and restores them). The value of `balance` is still zero at this point, and thus thread 2 has `eax=0`. Let's then assume that thread 2 executes the next two instructions, incrementing `eax` by 1 (thus `eax=1`), and then saving the contents of `eax` into `balance` (address `0x8049a1c`). Thus, the global variable `balance` now has the value 1.

Finally, another context switch occurs, and thread 1 resumes running. Recall that it had just executed the first `mov` instruction, and is now about to add 1 to `eax`. Recall also that `eax=0`. Thus, the `add` instruction increments `eax` by 1 (thus `eax=1`), and then the `mov` instruction saves it to memory (thus `balance` is set to 1 again).

Put simply, what has happened is this: the code to increment `balance` has been run twice, but `balance`, which started at 0, is now only equal to 1. A “correct” version of this program should have resulted in `balance` equal to 2.

Here is a pictorial depiction of what happened and when in the example above. Assume, for this depiction, that the above code is loaded at address 100 in memory, like the following sequence (note: as it turns out, x86 has variable-length instructions, and the `mov` instructions here take up 5 bytes whereas the `add` takes only 3):

```
100 mov    0x8049a1c, %eax
105 add    $0x1, %eax
108 mov    %eax, 0x8049a1c
```

With these assumptions, what happens is seen in Figure 18.5. Trace through this example to make sure you understand what is going on.

What we have demonstrated here is called a **race condition**: the results depend on the timing execution of the code. With some bad luck (i.e., context switches that occur at untimely points in the execution), we get the wrong result. In fact, we may get a different result each time; thus, instead of a nice **deterministic** computation (which we are used to from computers), we call this result **indeterminate**, where it is not known what the output will be and it is indeed likely to be different across runs.

Because multiple threads executing this code can result in a race condition, we call this code a **critical section**. A critical section is a piece of code that accesses a shared variable (or more generally, a shared resource) and must not be concurrently executed by more than one thread.

OS	THREAD 1	THREAD 2	BALANCE
	mov 0x8049a1c, %eax (eax=0)		0
Interrupt			0
Save T1			0
(pc=105, eax=0)			
Restore T2			0
(pc=100, eax=0)			
Run T2			0
		mov 0x8049a1c, %eax (eax=0)	0
		add \$0x1, %eax (eax=1)	0
		mov %eax, 0x8049a1c (eax=1)	1
Interrupt			1
Save T2			1
(pc=113, eax=1)			
Restore T1			1
(pc=105, eax=0)			
Run T1			1
	add \$0x1, %eax (eax=1)		1
	mov %eax, 0x8049a1c		1 (not 2!)

Figure 18.5: The Problem: Up Close and Personal

What we really want for this code is what we call **mutual exclusion**. This property guarantees that if one thread is executing within the critical section, the others will be prevented from doing so.

Virtually all of these terms, by the way, were coined by Edsger Dijkstra, who was a pioneer in the field and indeed won the Turing Award because of this and other work; see his paper on “Cooperating Sequential Processes” [D68] for an amazingly clear description of the problem from 1968! We’ll be hearing more about Dijkstra, of course, in this section of notes.

## 18.4 The Wish For Atomicity

One way to solve this problem would be to have more powerful instructions that, in a single step, did exactly whatever we needed done and thus removed the possibility of an untimely interrupt. For example, if we had a super instruction that looked like this:

```
memory-add 0x8049a1c, %eax
```

which added a value to a memory location, the hardware would guarantee that this would execute **atomically**; when the instruction executed, it would perform the update as desired. It could not be interrupted mid-instruction, because that is precisely the guarantee we receive from the hardware: when an interrupt occurs, either the instruction has not run at all, or it has run to completion; there is no in-between state. Hardware can be a beautiful thing, no?

Atomically, in this context, means “as a unit”, which sometimes we take as “all or none.” What we’d like is to execute the three instruction sequence atomically:

```
mov 0x8049a1c, %eax
add %eax, %eax
mov %eax, 0x8049a1c
```

As we said, if we had a single instruction to do this, we could just issue that instruction and be done. But in the general case, we won’t have such an instruction. Imagine we were building a concurrent B-tree, and wished to update it; would we really want the hardware to support a generic “atomic update of B-tree” instruction? Probably not<sup>1</sup>.

Thus, what we will instead do is ask the hardware for a few useful instructions upon which we can build a general set of what we call **synchronization primitives**. By using these synchronization primitives, in combination with some help from

---

<sup>1</sup>If you do, you are in what is called the **CISC** camp of computer architecture (the opposite of **RISC**). There is little hope for you then, alas.

the OS, we will be able to build multi-threaded code that accesses critical sections in a synchronized and controlled manner, and thus reliably produces the correct result despite the challenging nature of concurrent execution.

#### CRUX OF THE PROBLEM

What support do we need from the hardware in order to build useful synchronization primitives? What support do we need from the OS?

This is the problem we will be studying in this section of the notes. It is a wonderful and hard problem, and should make your mind hurt (at least a little). If it doesn't, then you don't understand it.

## 18.5 Summary: Why in OS Class?

Before wrapping up, one question that you might have is: why are we studying this in OS class? “History” is the one-word answer. Simply put, the OS was the first concurrent program, and thus most of these techniques (such as those invented by Dijkstra) arose due to the need for them *within* the OS. Later, as multi-threaded programs became popular, application programmers also had to consider such things.

For example, imagine the case where there are two processes running. One calls into the kernel to open a file, say using the `open()` system call. At about the same time, another process does the same thing. Now say that there is a counter within the OS called `opencount`, which simply is incremented each time `open` is called. Because an interrupt may occur at any time, the update to the shared variable `opencount` is a critical section, as described above. Thus, OS designers, from the very beginning of the introduction of the interrupt, had to worry about how the OS would update any internal structures; an untimely interrupt would cause all of the problems that we describe above.

## References

[D68] "Cooperating sequential processes"

Edsger W. Dijkstra, 1968

Available: <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF>

[S92] "Advanced Programming in the UNIX Environment" W. Richard Stevens

Addison-Wesley, 1992

*As we've said many times, buy this, use it, live it.*