25

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 address spaces across physical memory (and sometimes, disk).

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 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 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 state of T1 must be saved and the

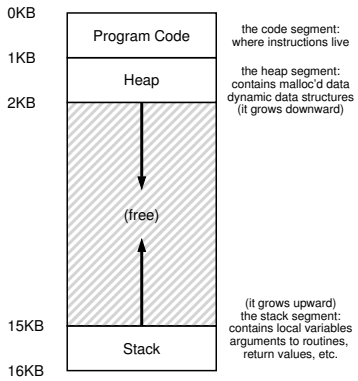

Figure 25.1: A Single-Threaded Address Space

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 25.1).

However, in a multi-threaded process, each thread runs independently and of course may call into various routines to do 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 two threads in it. In such a case, our address space looks different (Figure 25.2).

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.

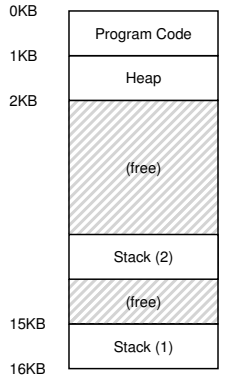

Figure 25.2: A Multi-Threaded Address Space

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).

25.1 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 printing "A" or "B". The code might look something like what you see in Figure 25.3.

The main program creates two threads, each of which will run the function `mythread()`, though with different arguments (the string `A` or `B`). Once a thread is created, it may start running right away (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 this execution

```
#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: end\n");
    return 0;
}
```

Figure 25.3: Simple Thread Creation Code

diagram, time increases in the downwards direction, and each column shows when a different thread (the main one, or T1, or T2) is running:

```
main starts running
main prints "main: begin"
main creates thread T1
main creates thread T2
main waits for T1
                            T1 runs
                            T1 prints "A"
                            T1 returns
main waits for T2
                                                  T2 runs
                                                  T2 prints "B"
                                                  T2 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 T1
                            T1 runs
                            T1 prints "A"
                            T1 returns
main creates thread T2
                                                  T2 runs
                                                  T2 prints "B"
                                                  T2 returns
main waits for T1
  (returns immediately because T1 is finished)
main waits for T2
  (returns immediately because T2 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 T1
main creates thread T2
                                                  T2 runs
                                                  T2 prints "B"
                                                  T2 returns
main waits for T1
                            T1 runs
                            T1 prints "A"
                            T1 returns
main waits for T2
  (returns immediately because T2 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.

25.2 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 threads wish to update a global shared variable. The code might look something like what is in Figure 25.4.

A few notes about the code. First, as Stevens suggests [SR05], 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 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 `counter`, and do so 10 million times (1e7) in a loop. Thus, the desired final result is: 20,000,000.

We now compile and run the program, to see how it behaves. If everything works as we might think it would, the program produces the following output:

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

Unfortunately, when we run this code, even on a single processor, we don't necessarily get the desired result. Sometimes, we might get something more like this:

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

static volatile int counter = 0;

//
// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

//
// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    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 (counter = %d)\n", counter);
    return 0;
}
```

Figure 25.4: Sharing Data: Oh Oh

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

Let's try it one more time, just to see if we've gone crazy. After all, aren't computers supposed to produce **deterministic** results? This is what you have been taught! Perhaps your professors have been lying to you? (gasp)

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
prompt>
```

Not only is each run wrong, but also yields a *different* result! A big question remains: why does this happen?

25.3 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 `counter`. In this case, we wish to simply add a number (1) to `counter`. 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 `counter` 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.

TIP: KNOW AND USE YOUR TOOLS

You should always learn new tools that help you write, debug, and understand computer systems. Here, we use a neat tool called a **disassembler.** When you run a disassembler on an executable binary, it shows you what assembly instructions make up the program. For example, if we wish to understand how the low-level code to update a counter (as in our example above), we might run `objdump` (on Linux) to see what the assembly code looks like:

```
prompt> objdump -d main
```

Doing so produces a long listing of all the instructions in the program, neatly labeled (particularly if you compiled with the `-g` flag), which includes symbol information in the program. The `objdump` program is just one of many tools you should learn how to use; a debugger like `gdb`, memory profilers like `valgrind` or `purify`, and of course the compiler itself are others that you should spend time to learn more about; the better you are at using your tools, the better systems you'll be able to build.

Let us imagine one of our two threads (thread 1) enters this region of code, and is thus about to increment `counter` by one. It loads the value of `counter` (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 `counter` 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 `counter` 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 `counter` (address 0x8049a1c). Thus, the global variable `counter` 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

```
                                                    (values AFTER inst)
OS             THREAD 1            THREAD 2          PC  %eax COUNTER
           (before crit sect)                        100   0    50
           mov 0x8049a1c, %eax                       105  50    50
           add $0x1, %eax                            108  51    50
Interrupt
(save T1's state,
restore T2)                                          100   0    50
                              mov 0x8049a1c, %eax    105  50    50
                              add $0x1, %eax         108  51    50
                              mov %eax, 0x8049a1c    113  51    51
Interrupt
(save T2's state,
restore T1)                                          108  51    51
           mov %eax, 0x8049a1c                       113  51    51
                                                             (not 52!)
```

Figure 25.5: The Problem: Up Close and Personal

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 counter is set to 1 again).

Put simply, what has happened is this: the code to increment `counter` has been run twice, but `counter`, which started at 0, is now only equal to 1. A "correct" version of this program should have resulted in `counter` 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 for those of you used to nice, RISC-like instruction sets: x86 has variable-length instructions; the `mov` instructions here take up 5 bytes of memory, 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 25.5. Assume the counter starts at value 50, and 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.

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 the book.

25.4 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, $0x1
```

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

ASIDE: **KEY CONCURRENCY TERMS**
CRITICAL SECTION, RACE CONDITION,
INDETERMINATE, MUTUAL EXCLUSION

These four terms are so central to concurrent code that we thought it worth while to call them out explicitly. See some of Dijkstra's early work [D65,D68] for more details.

- A **critical section** is a piece of code that accesses a *shared* resource, usually a variable or data structure.
- A **race condition** arises if multiple threads of execution enter the critical section at roughly the same time; both attempt to update the shared data structure, leading to a surprising (and perhaps undesirable) outcome.
- An **indeterminate** program consists of one or more race conditions; the output of the program varies from run to run, depending on which threads ran when. The outcome is thus not **deterministic**, something we usually expect from computer systems.
- To avoid these problems, threads should use some kind of **mutual exclusion** primitives; doing so guarantees that only a single thread ever enters a critical section, thus avoiding races, and resulting in deterministic program outputs.

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 an "atomic update of B-tree" instruction? Probably not, at least in a sane instruction set.

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 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.

This is the problem we will be studying in this section of the book. 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. So keep working until your head hurts; you then know you are headed in the right direction.

THE CRUX:
HOW TO PROVIDE SUPPORT FOR SYNCHRONIZATION
What support do we need from the hardware in order to build useful synchronization primitives? What support do we need from the OS? How can we build these primitives correctly and efficiently? How can programs use them to get the desired results?

25.5 One More Problem: Waiting For Another

This chapter has set up the problem of concurrency as if only one type of interaction occurs between threads, that of accessing shared variables and the need to support atomicity for critical sections. As it turns out, there is another common interaction that arises, where one thread must wait for another to complete some action before it continues. This interaction arises, for example, when a process performs a disk I/O and is put to sleep; when the I/O completes, the process needs to be woken so it can continue.

Thus, in the coming chapters, we'll be not only studying how to build support for synchronization primitives to support atomicity but also for mechanisms to support this type of sleeping/waking interaction that is common in multi-threaded programs. If this doesn't make sense right now, that is OK! It will soon enough, when you read the chapter on **condition variables**. If it doesn't by then, well, then it is less OK, and you should read that chapter again (and again) until it does make sense.

25.6 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 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()`

TIP: USE ATOMIC OPERATIONS

Atomic operations are one of the most powerful underlying techniques in building computer systems, from the computer architecture, to concurrent code (what we are studying here), to file systems (which we'll study soon enough), database management systems, and even distributed systems [L+93].

The idea behind making a series of actions **atomic** is simply expressed with the phrase "all or nothing"; it should either appear as if all of the actions you wish to group together occurred, or that none of them occurred, with no in-between state visible. Sometimes, the grouping of many actions into a single atomic action is called a **transaction**, an idea developed in great detail in the world of databases and transaction processing [GR92].

In our theme of exploring concurrency, we'll be using synchronization primitives to turn short sequences of instructions into atomic blocks of execution, but the idea of atomicity is much bigger than that, as we will see. For example, file systems use techniques such as journaling or copy-on-write in order to atomically transition their on-disk state, critical for operating correctly in the face of system failures. If that doesn't make sense, don't worry – it will, in some future chapter.

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 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; thus, OS designers, from the very beginning of the introduction of the interrupt, had to worry about how the OS would update internal structures; an untimely interrupt causes all of the problems described above. Thus, page tables, process lists, and virtually every kernel data structure has to be carefully accessed, with the proper synchronization primitives, to work correctly.

References

[D65] "Solution of a problem in concurrent programming control"
E. W. Dijkstra
Communications of the ACM, 8(9):569, September 1965
Pointed to as the first paper of Dijkstra's where he outlines the mutual exclusion problem and a solution. The solution, however, is not widely used; advanced hardware and OS support is needed, as we will see in the coming chapters.

[D68] "Cooperating sequential processes"
Edsger W. Dijkstra, 1968
Available: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF
Dijkstra has an amazing number of his old papers, notes, and thoughts recorded (for posterity) on this website at the last place he worked, the University of Texas. Much of his foundational work, however, was done years earlier while he was at the Technische Hochshule of Eindhoven (THE), including this famous paper on "cooperating sequential processes", which basically outlines all of the thinking that has to go into writing multi-threaded programs. Dijkstra discovered much of this while working on an operating system named after his school: the "THE" operating system (said "T", "H", "E", and not like the word "the").

[GR92] "Transaction Processing: Concepts and Techniques"
Jim Gray and Andreas Reuter
Morgan Kaufmann, September 1992
This book is the bible of transaction processing, written by one of the legends of the field, Jim Gray. It is, for this reason, also considered Jim Gray's "brain dump", in which he wrote down everything he knows about how database management systems work. Sadly, Gray passed away tragically a few years back, and many of us lost a friend and great mentor, including the co-authors of said book, who were lucky enough to interact with Gray during their graduate school years.

[L+93] "Atomic Transactions"
Nancy Lynch, Michael Merritt, William Weihl, Alan Fekete
Morgan Kaufmann, August 1993
A nice text on some of the theory and practice of atomic transactions for distributed systems. Perhaps a bit formal for some, but lots of good material is found herein.

[SR05] "Advanced Programming in the UNIX Environment"
W. Richard Stevens and Stephen A. Rago
Addison-Wesley, 2005
As we've said many times, buy this book, and read it, in little chunks, preferably before going to bed. This way, you will actually fall asleep more quickly; more importantly, you learn a little more about how to become a serious UNIX programmer.