

Threading Intro

1 Why Threads?

The computing world is becoming more concurrent. Processors used to get faster by increasing the clock speed. Clock speed is the processor-wide frequency with which logic components (read: circuits) can ‘latch in’ values. Effectively, this means that the clock is the basic quantum of computation on a processor (using synchronous logic, that is). So, if you can double the clock speed on the same chunk of hardware logic, you can effectively double the number of computations per second. Programmers love this kind of hardware wizardry. We (that is, programmers) don’t have to do anything and every 2 years or so our code magically runs twice as fast!

Of course, nothing this good lasts forever. After 20+ years of intense engineering (for example, superscalar pipelined processors), clock scaling is done. The problem is that clocks are power-hungry. Once we reached 1 GHz (1 billion ticks per second), increasing clock speed meant increasing core temperature to dangerous levels. As a general rule of thumb, doubling the clock speed would square the power dissipation (e.g. it increased quadratically). Over the last 5 years, the major processor manufacturers have given up increasing clock speed much over 2.5 GHz and have started putting multiple cores on a single die (so-called multicore processors). To give you an idea why, think that if a single core dissipates 100W of power and you double the clock speed it would suddenly be dissipating 10000W (and be running the danger of melting without intense cooling). If however, you just clone the core and have two running side by side you have effectively doubled the processing power (in terms of instructions per second), but you’ve increased the power dissipation only to 200W. This is why Sun, Intel and AMD are all on the multicore bandwagon.

The problem is that multicore processors can only execute code faster if that code can take advantage of concurrently executing processors. The OS can help somewhat. If you have a dual-core machine, then the OS can run two processes at the same time. This is not unusual. Most systems will probably have two or more ready processes at any given moment. But we’ve already got 4 core desktop machines. What happens in 5 years when intel et al are producing 16+ core machines? My machine usually has two ready processes at any given moment, but it rarely has 16, 32 or more. So, simply having the OS multiplex ready processes only scales to a few cores. Past that, you need each individual process to internally exploit concurrent processing. This is commonly known as multi-threading, and used to be the domain of specialist parallel programmers. Now it looks like any marketable programmer is going to have to be able to cope with a multi-threaded world.

2 Process Interaction

At the heart of a multi-threaded program are a set of threads working on different parts of a problem simultaneously. Hopefully these threads are mostly independent, but occasionally they will need to interact with one another.

2.1 Assumptions

In dealing with concurrent code, we need to make assumptions. First, we assume that **the execution speed is unknown!** We assume that we won't be able to predict when and how fast a given thread/process is executing, however we will assume that each thread is making some progress each time it is run.

Second, we assume that **the interleaving is unknown!** That is, we don't know where other threads are in their execution. Additionally, we must assume that the current thread can also be interrupted at any point. In practice, this means that multithreaded coding is often reasoning about the worst case configuration of thread states and protecting against that.

Both these assumptions mean that a thread CANNOT assume anything about the values of shared variables. Any variables you share with other threads can change unpredictably and at ANY time!

2.2 Atomicity

Threaded code is obsessed with **atomicity**. An atomic operation is one that CANNOT be interrupted. This allows a thread to perform an operation completely without worrying about being interrupted by another thread. In reality, atomic operations are machine instructions. The hardware ensures that each instruction executes atomically, without interruption. Of course, this is of limited help because your average programmer doesn't want to design code at the machine instruction level!

2.3 Concurrent Constructs

Most programming languages are inherently serial. Parallel programming was, for a long time, the exclusive domain of supercomputing types. Most of the actual programs in the wild ran on single processor systems, and so a simpler serial language was preferable. Several ways of specifying concurrent execution have evolved over time:

1. **Procedural** - Processes/threads are specified at the function level. Some keyword (like `process`, `thread`, or `atomic`) is introduced to allow the programmer to specify chunks of code to run as threads.
2. **Parallel Operators** - Some (more exotic) languages introduce special operators to specify concurrent vs. serial execution. For example `S1 || S2` may mean execute S1 and S2 in parallel, while `S1 ; S2` may mean execute S1 before S2 (serially).
3. **Dynamic** - With dynamic threading, special functions are called to create and manipulate threads/process. For example, the `fork` and `exec` functions create new processes. With `pthread`s in C, `pthread_create` creates a new thread and `pthread_join` can be used to merge threads.

The dynamic route is favored by those bolting multithreaded functionality onto old serial languages. For example, threading in C is usually accomplished by calls into a library that handles all the jiggery-pokery of getting the OS to create new threads. `pthread` has emerged as the standard C threading library for UNIX systems. Procedural abstractions are more popular in higher-level languages. For example, transactional extensions of Java and ML use a new keyword (`atomic`) to specify blocks of concurrent code. Parallel operators are far more obscure and exist mostly in specialist languages. For the purpose of this course, we're going to be focusing on `pthread`.

3 A Bounded Buffer Example

The bounded buffer is a canonical example in concurrency circles. First assume that there are two processes, executing concurrently, that need to communicate with one another. They communicate through shared memory/variables. For example, the I/O driver for the keyboard needs to communicate the user's keystrokes to higher levels of the linux kernel. For this example, we assume that there are two kinds of process:

Producer a producer process produces values to be read. It creates values/records.

Consumer a consumer process reads produced values. It uses/reads records.

In the simplest (non-trivial) case, there is one producer and one consumer. We assume that they are mis-matched in speed and are executing asynchronously. For example, we cannot assume that the producer will always run faster and before the consumer is run.

How do these two threads communicate? Because the producer could be faster than the consumer, we would like a multi-element buffer to hold multiple values in case the producer hits its stride and generates multiple values. And, because we'd like the consumer to have enough data to work on to justify the overhead of multithreading, a buffer would be a good idea from the consumer side too. Therefore what we'd like is a shared buffer. Abstractly, we want a shared queue. The consumer plucks values off the head of the queue while the producer appends values to the tail of the queue.

What would this actually look like? At the lower levels of the system, dynamic allocation is complicated (and slower than statically sized data). So, we're going to have to restrict the queue to have a fixed size (this is the bounded part of the bounded buffer). Additionally, at this level, pointers are an extravagance, so how can we implement a circular queue without pointers? The answer is to have a fixed size array of records (of maximum length N), with two integers on the side. One, `first` holds the index of the head of the queue. The other, `len` holds the number of unprocessed records in the queue.

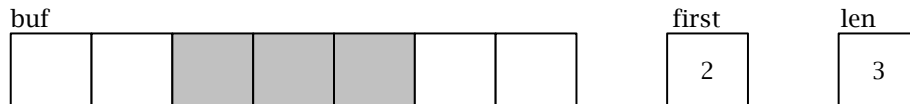


Figure 1: A bounded buffer example

The operations possible on such a queue are as follows:

Add: Producer inserts a new record at $\text{buf}[(\text{first} + \text{len}) \% N]$ and sets $\text{len} = (\text{len} + 1) \% N$, if $\text{len} < N$. Waits, otherwise.

Remove: Consumer removes a record at $\text{buf}[\text{first}]$ and sets $\text{len} = \text{len} - 1$, $\text{first} = (\text{first} + 1) \% N$, if $\text{len} > 0$. Waits, otherwise.

Already we're on tricky ground. What happens if there's nothing in the queue for the consumer to read (i.e. the producer has yet to run, or is running slowly)? What happens if the consumer is too slow and the producer has filled the buffer? The short answer is, we'd like the consumer to wait until there's something in the buffer to read in the first case, and we'd like the producer to wait until the consumer has processed some records in the second. Getting this right is harder than it sounds, and it usually involves something known as condition variables (which we'll get into later).

4 Interleaving

When one talks about threads, one starts reasoning about **interleavings**. An interleaving is a mixture of instructions from multiple threads. Even though each thread's instructions are executed in program order, the thread may be interrupted at any time. Therefore, between any two atomic instructions an arbitrary number of instructions from other threads may have executed. Critically important are instructions to read and write from memory. Consider the following C code (where A and B mark different threads):

```
A: len--;
```

```
...
```

```
B: len++;
```

These C one-liners will compile down to multiple machine instructions. On a RISC-style load/store architecture, the code may look something like:

```
A1: LOAD r1, [len]
A2: SUB r1, 1, r1
A3: STOR r1, [len]
```

```

...
B1: LOAD r2, [len]
B2: ADD r2, 1, r2
B3: STOR r2, [len]

```

Recall that the hardware only guarantees atomic execution of single instructions, so threads A and B can be interrupted between each individual instruction. That means that even this short example has many interleavings. Consider just the interleavings of the memory instructions (LOAD, STOR).

Instruction Sequence	Final value of len
A1 ; B1 ; A3 ; B3	len + 1
B1 ; A1 ; A3 ; B3	len + 1
B1 ; A1 ; B3 ; A3	len - 1
A1 ; B1 ; B3 ; A3	len - 1
A1 ; A3 ; B1 ; B3	len
B1 ; B3 ; A1 ; A3	len

This is where multithreaded programming starts to get really tricky. Because you can't even assume that a one-liner in C will execute atomically!

This example reveals some of the pitfalls of concurrency, and lets us define two new vocabulary terms:

Race Condition - a situation in multithreaded code where the outcome depends unpredictably on the execution order of the threads. So called because the threads are all "racing" for access to the shared variable.

Critical Section - a block of code where interleavings must be prevented (usually to avoid races). In general, a critical section is a chunk of code that needs to execute atomically in order for the program to be correct.

5 Synchronization

So, at this point we know that threaded code can interfere with itself in unpleasant ways. Race conditions should be avoided, so code that manipulates shared data should be organized into critical sections. Fine, but how do you actually prevent other threads from executing their critical sections when your thread is in the middle of its critical section? As often in systems, there are several ways to accomplish this, and each with its own advantages and disadvantages.

Conditional - The thread waits until some condition becomes true.

Mutual Exclusion - Critical sections are guarded by mutex's, and only the thread that holds the mutex may execute the critical section (all other threads must wait).

We've already encountered examples of conditional synchronization. When a process asks the OS for a file to be read off disk, the OS has the process wait until the file data is available in memory. Mutual exclusion is trickier. A thread must claim a mutex in order to execute a critical section, so the protocol for acquiring and releasing mutexes must be interleaving-proof (i.e. it must either not be interrupted, or immune to interference in the presence of interruptions).

At a high level of abstraction, a thread with a critical section looks like:

```
...
NCS1 //non-critical code
Entry1 //start of critical section 1
CS1 //actual code of critical section
Exit1 //leaving critical section
NCS2 //non-critical code
Entry2
CS2 //Second critical section
Exit2
NCS3
...
```

A mutex must ensure that only one thread is executing CS1 at any given time, and a separate mutex must ensure that only one thread is executing CS2 at any given time. That is absolutely essential. However, it is desirable for a mutex to be:

Efficient Entry and Exit should be fast, and acquiring a mutex for CS 1 or 2 shouldn't impact other threads executing the non-critical code (i.e. only code that's executing critical sections should pay the cost of acquiring a mutex).

Progress If multiple threads are attempting to execute a critical section, at least one of them gets to (i.e. some work is always getting done).

No Starvation Every contending thread eventually executes its critical section (i.e. no thread is starved of mutexes).

The challenge, of course, is implementing a mutual exclusion primitive that is correct, efficient, progress-ensuring and fair all at the same time. This is, as you may imagine, quite difficult.