CS696 Concurrency

Kyle Clapper

December 7, 2023

1 Introduction

Concurrency is a notoriously tricky subject. Mentions of concurrent programs are often met with trepidation and fear because of their reputation for being difficult to program. These fears are not unwarrented because concurrent programs upset the mental model many programmers have for their programs. That being of sequential operations that happen (more or less) in the order their program specifies. These are typical sequential programs. Adding concurrency decouples the lexical order of the program from the order in which parts of the program are executed. This requires a very different mental model and a host of new systems and abstractions to manage it.

For many years programs were written almost exclusively in a sequential fasion. Programmers at the time could rely on Moores law to increase the speed of their programs without doing any additional work. Then, sometime around 2005, programmers began to realize the limits of Moores law. At some point, the "free lunch" will be over and they won't be able to rely on their programs getting faster as faster processors come out. As we've seen in recent years, the increase in CPU clock speed has started to slow and the focus is more and more on CPUs with multiple cores. With this shift in computer hardware, it's becoming increasingly necessary for programmers to write concurrent programs to take advantage of all system resources.

1.1 Concurrency vs Parallelism

A common point of confusion is the difference between concurrency and parallelism. These terms are very related, but they cannot necessarily be used interchangeably. In a similar way that a square is a rectangle but a rectangle is not necessarily a square, paralellism is a form of concurrency but concurrency does not necessarily imply parallelism.

Concurrency is when you have multiple independent threads of computation happening in a program. Parallelism is when two or more of those threads are being processed at the same time. Concurrency without parallelism often takes the form of time sharing. For example, two concurrent processes executing on a single CPU machine will appear to run at the same time, but they are

not parallel because they aren't actually running at the same time but instead sharing time slices allocated by the operating system.

For example, if two operating system processes are running at the same time on separate CPU cores, then they are running in parallel. If instead, you have a program with two functions, one that requires waiting for an I/O device and another that does not, and the non-I/O function runs while the other function is waiting, then this is concurrency but not parallelism. In this example, both functions are running on the same thread of computation, but the time is shared between them.

It's worth mentioning that distributed systems are concurrent as well. If you have a web application that's built as a collection of independent services communicating over HTTP, these services act in a concurrent fashion, and their interactions proceed concurrently. These services are likely running in parallel, but there's nothing stopping these services from both running on a single core server, in which case they would not be in parallel and would instead be taking turns on the one processor.

2 Concurrent Abstractions

Concurrency is tricky, and as programmers are want to do, many levels of abstraction have been built to make the task easier. These abstractions start at roughly the operating system level, which abstracts over an arbitrary number of hardware cores, and ends with programming languages which end up imitating an operating system in order to get a similar level of abstraction.

2.1 The Operating System

Back when computers often only had a single CPU core, it was necessary for the operating system to provide abstractions to allow multiple programs to run at once. In order to multitask, the operating system needed to give programs a 'process' abstraction, in which the program could view itself as the only running program on the machine, or rather the exclusive user of a hardware thread. With only one hardware thread and many programs, the OS switched between them quickly in order to simulate multiple threads. To users, especially of operating systems with graphical user interfaces, it appeared as though the programs were all running at once instead of time sharing.

This abstraction was extended with the widespread release of multi-core machines. Instead of time sharing over one hardware thread, the operating system could now spread the workload out over multiple. This means programs physically running at the same time, as opposed to switching back and forth. The operating system still provides process abstractions. In addition, operating systems typically also provide a 'threads' abstraction. Programs can split themselves up into multiple threads of execution, each thread having less overhead than a 'process'. These threads often share a memory space and can communicate between each other using this shared memory.

2.2 Shared State

The concurrency model of lanugages like C, C++, and Java is primarily threads with shared state or shared memory. In this model, different threads of execution communicate amoungst each other using shared memory. With this model comes additional abstractions to handle safely interacting with shared memory.

Threads are the primary abstraction in the shared state model. These threads largely line up with the concept of operating system threads. Indeed, in C and C++ these are exactly the threads that are used. In languages like Java however, they have their own concept of threads which are lighter weight that OS threads. Java, running on a virtual machine, is able to create and manage this abstraction on it's own. In that case, the Java threads are scheduled on the JVM and the JVM translates this to interactions with operating system / hardware.

When you have multiple threads of execution accessing the same memory at the same time, you run into new classes of issues not present in sequential programs. One such class of issues is race conditions, where two threads access a shared resource at the same time. In this scenario, the threads may have different effects on that resource resulting in conflicting updates to the shared state. For example, if two threads access a counter variable at the same time, they both might read the same value from memory, but if they go to update the value with different numbers, only one of them will actually be stored and the other will be overwritten.

2.2.1 Locks

Locks are an abstraction that is primarily used to resolve race conditions and other problems arising from multiple threads trying to access the same resource at the same time. In it's simplest form, a lock on a resource can be acquired by a single thread, which prevents other threads from accessing that resource until the lock is released. In this way, locks allow for mutual exclusion when accessing shared resources (and this kind of lock is often called a mutex). There are various types of locks, for example a read-write lock is a lock that allows either multiple read-only locks on a resource or a single write lock. This is useful because the value of a shared resource can be read by multiple threads simultaneously without issue so long as the value is not changed. If the value is going to be changed, it must happen under mutual exclusion to ensure there is no confusion over it's value.

Locks help solve race conditions in shared memory concurrency environments, but they can introduce their own problems. Consider the scenario where multiple threads need to access a shared resource to make progress. If one of the threads aquires a mutual exlusion lock on the resource then fails to halt or to release the lock, then the other threads are stuck in a blocked state and will fail to make progress. If the thread with the lock fails to halt or needs to access a lock held by one of the blocked threads, then the whole program will hang. This situation is called a deadlock. The resources needed to make progress are

held under locks that will not be released.

Similarly, consider the scenario where two threads need access to two shared resources in order to make progress. Consider that each thread may try to place a lock on differing resources. If they then try to place a lock on the other resource, they may find it's being held and so they release their lock then try again from the beginning. In this scenario, it's possible that both threads keep trying to lock the shared resources but are unable and so release their locks, constantly oscillating between locked and unlocked and unable to make progress. This situation is called live lock.

2.2.2 Critical Regions

Critical regions are sections of code where access to shared resources may introduce race conditions or other concurrency bugs. While the programmer can use locks to prevent these bugs, some languages provide features that enable them to instead simply specify a critical section and leave it up to the runtime to implement the concurrency. An example of this is the synchronized keyword in java. This language abstraction provides the programmer a means to specify a region of code and provides a guarantee that code will always run under mutual exclusion. Meaning, while many threads of computation may want to execute it at the same time, only one of them can and the others must wait their turn.

This kind of critical region abstraction allows programmers to control access to shared resources by specifying regions of code that cannot run at the same time. While that code may not directly place locks on the shared resources, by controlling when it runs, the programmer can achieve the desired effect.

2.2.3 Atomicity and Optimistic Concurrency

Another class of concurrency abstractions are atomic operations or data structures. These abstractions provide guarantees that their operations will either happen completely, or not happen at all. They ensure the user will not see half completed operations. With these guarentees, the programmer is free to use them as a shared resource amoung multiple threads without fear of race conditions.

Atomic operations, along with other concurrency abstractions, have driven the adoption of certain hardware features in modern CPUs. One such feature is the compare-and-swap operation which checks if a value in memory is the same as a given value and if it is, swaps it with a new value. The hardware guarantees this operation will happen atomically, which means it can be used to prevent concurrency bugs and as the basis for other concurrency abstractions.

Indeed, the compare and swap operation is the basis of a technique called optimistic concurrency. In this technique, a thread tries to mutate a shared resource but monitors whether or not it was successful. If it was not successfull, the thread tries again. The compare-and-swap instruction gives the programmer the ability to implement this technique easily. They can ready the new value for the shared resource, then try to compare and swap. If the swap doesn't succeed,

they simply keep trying until it does. While this technique leaves the thread 'spinning', it avoids the potential overhead of using explicit locks and may in some cases be easier to implement or less error prone.

2.2.4 Use Cases

The shared state concurrency model and it's associated abstractions have a wide variety of use cases. It's one of the most basic concurrency models and is often used to implement higher level concurrency abstractions. These shared state abstractions often have lower memory or computational overhead compared with message passing or actors, so they're often used to implement high performance applications such as operating systems or compilers.

2.3 Message Passing

In the shared state concurrency model, abstractions are created to manage several threads reading and writing memory at the same time. Another concurrency model obviates the need for these mechanisms by removing the shared memory all together. Instead, threads communicate by "message passing" or sending each other copies of data. Without any shared state, many of the concurrency bugs described above cannot occur.

A programmer needs additional abstractions to build a message passing concurrency system. First, the system needs some concept of threads. These threads need to have their own state and thread of execution. Further, these threads need some system to communicate and share data. There may be explicit channels to send data through or thread ids to send messages. The data sent between threads needs to be copied or otherwise only under the control of a single thread. This means that if a thread sends some data to another, a copy of that data is either created at the other thread or it is otherwise unnaccessible from the first.

Since processes on a machine cannot share memory (or at least it's difficult for them to do so), message passing is the default strategy for interprocess communication. An example is the use of standard input and output on linux machines. When one process pipes it's output to another, it's passing a message to another concurrent process. It's not necessary for the first process to finish before the second process starts consuming it's output.

Similarly, machines interacting over the internet are using message passing. In a traditional client server model, for example a web server and a client browswer, both machines are running at the same time (concurrently). When the client makes a request to the server, it sends it as a message over the internet using the HTTP protocol. The client can continue doing work while it waits for a response from the server, which itself responds by sending an HTTP message. Neither machine shares any physical resources or memory, and so their concurrent interaction is achieved entirely through message passing.

2.3.1 Actors

Actors are a further abstraction built on the concept of message passing. They can be implemented in any language with basic concurrency primitives, but languages such as Swift have them built in. Actors combine a thread of computation along with it's data/state and communicate with other actors by sending messages. Each actor has an "inbox" queue where it recieves and processes messages from other actors. This queue is an abstraction that allows other actors to concurrently send at the same time. While it has no guarantee on the order the messages are recieved, it guarantees that each message is correctly recieved. Messages cannot be shared between actors, so when one actors sends data to another, they both have their own copies.

This pattern prevents bugs associated with shared state by ensuring that nothing is shared between actors. Each actor is only responsible for a single thread of computation and since it's the only thread accessing it's data, can ensure that data access is handled correctly. The actor abstraction allows the programmer to focus very little on how the actual concurrency is handled. The actor system provides the implementation of the "inbox", message passing, and actor thread scheduling.

2.3.2 Use Cases

Message passing and actors are high level concurrency abstractions that allow programmers to much more easily access concurrency in their programs, often with performance benefits. These types of abstractions are well suited for systems that need to scale a high degree of scalability or fault tolerance. The isolated nature of actors or threads with message passing makes it easier to handle failures gracefully. A good example of this is the Erlang programming language. This language was built with actors and concurrency at it's heart and was used by the company Ericson to built highly concurrent and fault tolerant telephony systems.

2.4 Recreating the Operating System

As higher and higher level abstractions are built to handle concurrency in programs, the programming language starts to resemble an operating system itself. For example, the Racket language has various high level abstractions to handle concurrency. One of the aims of these abstractions is to safely allow threads to access and share resources while retaining the ability to kill child threads as needed. These design goals have led the Racket language to adopt abstractions that are rather similar to the abstractions that an operating system provides the programs running on it.

In the Racket language, each thread of computation has it's own set of resources including I/O, memory, and control flow. These resources are governed by custodians, which are responsible for reclaiming these resources after they're done been used or the thread is terminated. These custodians can be hierarchical. If one thread spawns new threads, each will have their own sub-custodian

under the orign thread's custodian. If a custodian reclaims it's resources, it forces all of it's sub-custodians to reclaim their resources as well.

In this way the Racket language concurrency system starts to look similar to an operating system. A main thread (the OS) can spawn several other threads (processes). These threads themselves can spawn threads (posix threads), and if a thread is killed, the threads it spawned are killed as well (a terminated process also terminates any threads it created). The similarity is also shown in the way threads are allocated their own set of resources which are reclaimed when a thread is killed.

2.4.1 Use Cases

These concurrency abstractions in the Racket language where inspired by their applicability towards building Dr. Racket, a Racket IDE with a built in REPL. Dr. Racket is built with Racket, and the language's use of custodians and threads allows it to evaluate Racket programs and have a REPL while retaining a reponsive UI and the ability to kill either when necessary.