COS40003 - Concurrent Programming Report I

Andrew Giannakis - 102107466

13/04/2021

Contents

Week 1 - Computing Paradigms

Within the field of having multiple machines interact, there are multiple paradigms as to how this can be accomplished each with their own pros and cons, the paradigms we will be summarising include concurrent computing, parallel computing, distributed computing, cluster computing, grid computing, cloud computing and fog/edge computing.

Concurrent Computing

Concurrent computing is simultaneous processes happening from the users perspective, originally concurrent computing was developed for single processors however it also applies to multiple processors. Concurrent computing is usually implemented through the utilization of time-sharing to perform multiple tasks on a single thread, this has the advantage of allowing the CPU to work on one task while another task is waiting such as with an I/O operation. However concurrency introduces potential problems such as race conditions and deadlocks. From the users perspective the processes seem to be executing simultaneously due to time sharing and scheduling.

Parallel Computing

Parallel computing is simultaneous processes happening from the systems perspective. Parallel computing is often distinguished from concurrent computing by specifying that each processes runs with its own local memory. Since each process can be encapsulated, parallel computing allows processes to actually run at the same time through the use of multiple cores or processors. Programs written to utilize parallel computing typically break down tasks into several subtasks that can each be processed independently from one another and can be combined afterwards.

Distributed Computing

There is no single definition of a distributed computing system, typically a distributed system is defined as several autonomous computers each with their own local memory which communicate through some form of message passing.

Cluster Computer

A Cluster Computing system is formed by multiple homogeneously located, usually low-cost devices, connected together via a fast local area network. One of the goals with a Cluster Computing system is usually to provide a high performance system by combining many low-cost devices.

Grid Computing

Grid Computing is similar to Cluster Computing except computation is performed over multiple devices that are more heterogeneous and geographically separated from one another, which results in the devices being less coupled to one another.

Cloud Computing

The goal of cloud computing is to abstract away or hide computation being performed at a server, essentially providing Software as a Service (SaaS), a Platform as a Service (PaaS) or Infrastructure as a Service (IaaS) to the end users without exposing details of how the system is implemented.

Fog/Edge Computing

Fog/Edge computing is somewhat similar to cloud computing except it can be better for systems that require sensors or Internet of Things (IoT) devices that need to communicate to servers with low-delay. Fog nodes sit between sensor devices and a central server, with fog nodes being located much closer to the devices allowing them to communicate with minimal latency. Typically Fog Nodes compute information using the sensors while the central server is used as a database so that the Fog Nodes are connected to one another forming the overall system.

Week 2 - Processes

A process is a program in action, which usually is comprised of the programs instructions and runtime memory, as well as other associated state such as the CPUs register values, program counter and stack pointer. A processes also has many associated states such as created, running, blocked, etc. Together these parts allow processes to encapsulate their state, allowing an operating system to run multiple programs concurrently even on a single processor through time-sharing and scheduling by managing the states of the various running processes.

The History Of Processes

Computers started very simple, a user typed in a command and the command performed some operation, and then immediately stopped awaiting the users next command. This was not a very efficient workflow.

Later, batch processing was invented, rather then providing the machine with each command one by one, the commands were entered as a list and executed sequentially by the machine, this list of commands was a primitive sort of program. This worked better but still had problems, for example say you wanted to run two programs A and B, program A might perform IO operations which require the machine to wait for external hardware, during this time the machine is waiting and wasting time, ideally while program A is waiting program B can begin processing to improve performance.

To implement such functionality, a few things needed to happen. For one programs at the time ran assuming they were the only thing using memory and so had no consideration for other concurrent programs placing data in the same memory. To solve this each program was allocated a chunk of memory instead of all the memory, so that each programs memory was isolated and did not interfere with each other. This was the creation of the idea of a process. A process is essentially an encapsulation of a program paired with its own chunk of memory and various hardware state values such as CPU registers. The state of the computer can be saved and restored during execution by switching between processes, and multiple processes do not interfere with each other.

Process States & Transitions

Processes are assigned a state by the operating system, generally the possible states a process can be in includes:

State	Description
Created	For when a process is created, the state may be held here in a real-time operating system to prevent processes oversaturation leading to an inability to meet process deadlines.
Ready	For the when the process has been loaded into memory and is available for execution, at this state the operating systems scheduler is responsible for assigning the process a time slice.
Running	For when the process is currently being executed, the process may leave the running state at any point at the descretion of the operating system.
Blocked	For when the process needs to wait for something such as an I/O operation and cannot continue execution until the operation is complete, while in this state the os can schedule other tasks to run instead until the operation is complete and more time can be scheduled for the process to continue.
Final	For when the process has either finished execution or has been killed at the discretion of the operating system.

From any given state, processes will be transitioned between states by the scheduler. For example a process in the created state will enter the ready state, a process in the ready state will enter the running state, which can enter either the blocked state if waiting on an external operation such as I/O is required or the final state if the process is complete, from blocked the process will enter back into the ready state once the I/O operation is complete, or at the whim of the operating system the process may be placed in the final state to be killed, just to name a few examples.

Process Address Space

Each process is assigned its own virtual memory address space when it is started, the virtual memory address space consists of a program located in the code segment, some static data located in the data segment, as well as a stack and a heap. Giving each process its own address space means processes will not overwrite each other in memory, it also provides security benefits as processes cannot read the memory of other processes as easily (although this is possible through operating system specific API calls).

Process Concurrency

Execution of processes is usually done via a round-robin approach on machines with a single processor to give the impression that the processes are executing in parallel, when they are in fact each running sequentially one after another (concurrent execution). How this scheduling is done will be discussed further in the next section (week 3).

Working With Processes In C On Unix Systems

Fork

Creates a new process by duplicating the current process. This is achieved by making copies of the current processes memory and kernel data structures, and adding the copied process state into the set of running processes.

```
pid_t fork(void);
```

Wait

Suspends execution of the calling processes until one of its children terminate.

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
int waitid(idtype_t idtype, id_t id, siginfo_t *infop. int options);
```

Exec

Exec replaces the current process with a new specified process.

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

Week 3 - Scheduling

Scheduling is an important part of operating systems capable of running multiple processes, the scheduler is responsible for determining which programs run, for how long they can run, and in what order, they make these decisions using a variety of different information obtained from the system, this information then is applied to a particular scheduler strategy/approach which results in the systems final scheduling behaviour.

Scheduling Criteria

Criteria often used by the scheduler to make informed decisions include CPU utilization, which describes how busy the CPU is, turnaround time which is the average time between job submission and job termination, response time which is how long a task takes to get a response, and missed deadlines which can be used to infer how under load the system is.

Priorities

Along with the commonly used scheduling criteria, the scheduler has many other options at its disposal. For example it could assign priorities to the various processes and give more or less time to each process based on what priority it is. Priorities can be assigned to processes through a variety of techniques which can be broadly categorized into two groups, static algorithms which typically assign a fixed priority to a process when it is

started, or dynamic algorithms which alter the priority of the processes during their lifetime (usually with the insight of the before mentioned scheduling criteria).

Preemptive Vs Non-Preemptive

Operating systems can prescribe to one of two techniques for handling running processes, either being "preemptive" or "non-preemptive". Preemptive systems allow the process to be interrupted at any time at the whim of the operating system, while non-preemptive systems do not allow this and instead have processes run until they change their process state or run until completion.

Scheduler Approaches

There are multiple approaches that can be taken for deciding how to schedule processes/jobs. Some of these include First In First Out (FIFO), Shortest Job First, Shortest Remaining Time, Round-Robin Scheduling, Lottery Scheduling and Multi-Level Feedback Queues.

First In First Out (FIFO)

Like a queue data structure that is also first in first out (FIFO), this scheduling approach has processes start in the order they arrive, each process is run until completion and is thus FIFO scheduling is a non-preemptive scheduling approach.

Shortest Job First & Shortest Remaining Time

Shortest job first is a simple change from the FIFO approach, different processes are submitted at which point they are ordered from shortest to longest expected runtime, there are a few methods for obtaining the expected runtime such as keeping a history of similar processes or having them be supplied by the process itself, also like the FIFO approach, Shortest Job First is non-preemptive and so each process is run until completion.

Shortest remaining time is essentially the same as shortest job first except it is preemptive meaning jobs can be paused mid-execution if a job with a shorter execution time is submitted to the scheduler.

Round-Robin Scheduling

Round robin scheduling attempts to go through each process in a circle and give each a chance to run for a small slice of time, by doing this over and over again fast enough the processes will give off the impression they are running at the same time from the perspective of the user, thus making this a good simple technique for concurrent systems.

The small slice of time given to each process is called the "time quantum", it is typically between 1ms to 100ms.

Out of all the techniques, round-robin scheduling results in the worst turn-around time, however its is good for response time.

Lottery Scheduling

Lottery scheduling is a kind of proportional share-scheduling approach that does not have much application in the real world, the simple idea is that each processes is assigned a number of "lottery tickets" and processes are selected to run at random. For example:

```
Task A gets 50 tickets.
Task B gets 15 tickets.
Task C gets 35 tickets.
There are 100 tickets outstanding.
```

Multi-Level Feedback Queue

Multi-Level feedback queues (MLFQ) is a scheduling approach invented by Fernando Jose Corbato, turning award winner in 1990 for his pioneering work in organizing the concepts and leading the development of the general-purpose, large-scale, time-sharing and resource sharing computing systems. The purpose of MLFQ is to optimize turnaround time and minimize response time.

MFLQ function by assigning processes to different queues representing different priority levels, and by following a few basic rules processes are transitioned between the queues/priorities to optimize for response time and turn around time.

A poorer implementation of MLFQ can suffer from problems such as starvation or allow for gaming the scheduler, the former occurs when there are too many short running or interactive jobs and they consume too much CPU time, not giving longer running processes a chance to run and thus "starving" them. The latter issue occurs

1. If Priority A > Priority B, A runs before B.
2. If Priority A = Priority B, A and B run in Round Robin.
3. When a job enters the system, it is placed at the highest priority (topmost queue).
4. Once a job uses up its time allotment at a given priority, regardless of how many times it has given up the CPU, its priority is reduced (i.e. it is moved down a queue). This is done to address starvation.
5. After a given time period, move all the jobs in the system to the topmost queue. This is done to address gaming the scheduler.

MLFQ approximate task length by assuming at the start that a job is short and then progressively lowering the priority as the task runs.

MLFQ can also be tuned in a variety of ways, such as changing the number of queues, the size of the time slice (time quantum) given to each process, or modifying how often priorities are boosted. Unfortunately there is no "correct" value for any of these variables, however there are some generally accepted rules for choosing them. For example, Most MLFQ implementations allow for varying time-slice lengths across different queues, where high-priority queues are usually given short time slices while lower-priority queues are given longer time slices.

Week 4 - Threads

While processes are concurrent programming at the operating system level, threads allow for concurrent programming at the program level. Threads allow a program to

overlap computation and blocking operations such as I/O, allowing more performance to be squeezed out of a process, on top of this threads are usually lower overhead then processes including the cost of communication between threads due to the fact they share memory.

The History Of Threads

Continuing on from processes, the desire to make applications more real-time continued, for example if the computer is processing some task and the user presses a button, the user may have to wait for the computer to finish processing before it can respond to the button press, this is not ideal and thus threads were invented. Each thread can run an independent sub-task, however they are working in the same working space.

Differences Compared To Processes

Threads, like processes, have their own CPU context and stack memory. However unlike Processes which have different address spaces and a higher cost to communicate with other processes, threads instead use the same address space as their parent process and have a much lower communication cost with other threads due to the fact that they share memory.

Processes make operating systems multitasking, while threads make a process multi-tasking. Processes do not share resources in an OS while threads of a process share resources and memory. Processes are the basic units of resource allocation of operating systems while threads are the basic units of CPU scheduling of operating system.

Shared State Among Threads

As mentioned before, threads share the memory of their parent process, however in order for multiple sub-tasks to be performed within a single process there is some state that needs to be stored for each individual thread alongside the state they share.

Each thread is given its own CPU context, including a program counter, stack pointer and register states, they are usually also given certain operating system specific information such as user ID, group ID and parent process ID. In regards to memory, threads share an address space, however each thread is assigned its own stack. Apart from this the threads share the same code segment, data segment and heap, as well as any resources such as opened files, network sockets and synchronization structures such as locks.

Thread Pools

Threads are overall relatively cheaper then processes, however this does not mean they are cheap. Creating and destroying thread contexts are in fact expensive operations, especially in a hot section of code such as a loop. To reduce this cost thread pools emerged. Thread pools are a collection of threads that are initialized once, and kept around for as long as they are needed, and finally destroyed when they no longer are. Jobs can be submitted to a thread pool and work is put into a queue and completed by any threads currently not in use. This effectively removes the overhead of threads as existing threads are re-used instead of creating and destroying threads constantly.

Common Threading Approaches

There are multiple patterns or strategies commonly employed when using threads in code, these include the manager/worker approach and pipeline approach.

Manager / Worker

With this approach, a primary manager thread is responsible for handling I/O or other resources that cannot easily be threaded (for example an OpenGL context), the manager thread will then assign work in the form of small easily parallelizable tasks to a collection of worker threads, typically in the form of a thread pool. The manager thread will then do other work and/or wait for the worker threads to complete before combining and using the results.

Pipeline

In a pipeline, each thread handles a different stage of an "assembly line" passing along the results of one step of processing to the next thread. Threads typically hand work off to each other in a producer-consumer relationship. This works well when the job is a little harder to parallelize and there is enough work that the entire "assembly line" can be utilised, if work is trickled into a pipeline each thread ends up waiting on each other and thus the system is not being used as effectively as it could be.

Working With Threads In C On Unix Systems

Threading was not added to the C standard library until C11, so before that most software used the Posix standard threading API for implementing threaded code. To this day the C11 standard libraries threading is lacking in features and is not as widely supported as the Posix implementation. However if you only need basic threading functionality the C11 standard does implement all of the essential features, for anything more advanced it is probably better to use the Posix implementation instead. C11 threads are cross-platform while Posix threads where designed primarily for Unix systems however there are implementations for windows and other platforms available.

Some of the C11 standard threading function declarations:

```
int thrd_create(thrd_t thr, thrd_start_t func, void *arg);
void thrd_exit(int res);
int thrd_join(thrd_t thr, int *res);
void thrd_yield(void);
```

The Posix equivalent function declarations:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void *(*start_routine)(void *), void *arg);
void pthread_exit(void *retval);
int pthread_join(pthread_t thread, void **retval);
int pthread_yield(void);
```

Working With Threads In Java

Threads can be used in Java by either extending the `Thread` class or by implementing the `Runnable` interface for your own types. An example of extending the `Thread` class

looks something like:

```
class MyThread extends Thread
{
    public void run()
    {
        // ...
    }
}
// ...
Thread a = new MyThread();
a.start();
```

While implementing the `Runnable` interface looks like:

```
class MyRun implements Runnable
{
    public void run()
    {
        // ...
    }
}
// ...
Thread b = new Thread(new MyRun());
b.start();
```

Advantages of using runnable interface is that Java does not have multiple inheritance and so sometimes you may want you class to extend something else but still be usable within threads.

Advantages & Disadvantages Of Threads

Threads allow programs to mix I/O with computation, allowing work to be done instead of having the process enter a blocked state. Switching between thread contexts incurs less overhead then switching between processes, and overall the use of threads can result in generally faster software for modern multiprocessor CPUs.

Threads also have a number of downsides however, such as introducing complexity to a codebase and any debugging that needs to take place, introducing non-deterministic thread interactions due to data-races and badly implemented shared state, it also may not be compatible with existing code that was not written with multithreading in mind.

Week 5 - Lock

Locks are synchronization primitives that allow programs to limit access to shared data when there are multiple threads, normally such situations would cause race conditions to emerge however through the proper use of locks race conditions can be prevented.

Data Races

Data races occur when two threads attempt to mutate some shared data at the same time, leaving that data in an invalid state. To give an example, lets imagine two threads

attempting to increment a shared integer `n`. Assuming the instruction `n++` which would expand into `n = n + 1` translates roughly into the following instructions:

```
give me the value of n.
add 1 to that value.
store that value in n.
```

since the two threads could be executing in parallel, this can lead to a scenario such as:

```
Thread A:                   Thread B:
give me the value of n.
                            give me the value of n.
add 1 to that value.
store that value in n.
                            add 1 to that value.
                            store that value in n.
```

if `n` was 10, and two threads both added 1 to `n`, we would expect `n` to become 12, however since the broken down operations happened in the order specified above and both threads are working with their own local copy of the data, in this example thread B actually overwrote the work thread A performed and so the final answer would be 11, which is not what we expected. To make things worse, this result is non-deterministic (indeterminate) and can differ each time this program is run.

Critical Section

A critical section is a section of code that accesses a shared resource, usually a variable or data structure. Shared resource could be global/static or simply shared among different threads such as a field on an instance that is shared among the threads. If care is not taken to properly synchronize the critical section data races can occur.

Race Conditions

A race condition is when multiple threads of execution enter the critical section at roughly the same time, creating the conditions required for the possibility of data races to occur, Once race conditions are possible multiple threads may attempt to mutate shared data in parallel, leading to data races which are an undesirable outcome.

Mutual Exclusion

To avoid problems such as data races, threads should use some kind of mutual exclusion primitives such as `Mutexs` or `ReadWriteLocks`. Doing so guarantees that only a single thread can enter the critical section at a time which allows programs to avoid the possibility of data races from occurring, resulting in a deterministic program.

Atomicity

Means all or none, either it has not run at all or it has run to completion. Atomicity refers to the "smallest possible unit" of a thing (thus atomic terminology), which in relation to concurrent primitives refers to the fact that an atomic operation can not be broken down into further steps.

Locks In Java

There are multiple classes in Java that implement the Lock interface, including `ReentrantLock`, `ReentrantReadWriteLock.ReadLock`, and `ReentrantReadWriteLock.WriteLock`.

Each java object has an associated intrinsic lock. The lock is initially unowned, and as long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock. The `synchronized` keyword forces the current thread to obtain the objects intrinsic lock.

```
public synchronized void MyFunction()
{
    // object implicitly locked
    // ...
    // object implicitly unlocked
}
```

since the entire object is locked, it is not possible for multiple synchronized method calls to either the same or different methods to execute in parallel. Synchronized blocks can also be used:

```
public void addName(String name)
{
    // ...
    synchronized(this)
    {
        lastName = this;
        nameCount++;
    }
    // ...
}
```

Using synchronized blocks we can create multiple locks by utilizing objects intrinsic locks.

```
public class Demo
{
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1()
    {
        synchronized(lock1)
        {
            c1++;
        }
    }

    public void inc2()
    {
        synchronized(lock2)
```

```
        {
            c2++;
        }
    }
}
```

The question may arise as to which is better to use, the Lock interface or Javas synchronization keywords? The answer is that the Lock interfaces should generally be preferred as they provide more benefits. For example, the lock interface has the option of calling methods such as `tryLock()` so that you can do something else instead of blocking the waiting thread, you can also specify timeouts when attempting to lock.

Week 6 - Lock II

The previous week introduces locks, which allowed us to resolve data races by preventing race conditions in the critical sections of a codebase, however there is still an underlying question of how these locks are implemented so that they can grantee mutual exclusion, that is the topic covered in this section.

How Lock Is Implemented

A good lock implementation should have mutual exclusion so that multiple threads are prevented from entering a critical section at the same time, it should be fair in that each thread attempting to obtain the lock should have an opportunity to do so, the threads should not be starved, finally a good lock should ideally have as low overhead as possible to ensure good performance.

Controlling Interrupts

A bad implementation of lock could operate by disabling interrupts while locked and enabling them once unlocked again. This however has several very negative side effects. For one it requires too much trust in applications using the locks, allowing applications to arbitrarily disable interrupts could have security implications, especially since interrupts being disabled for extended periods of time can result in serious systems problems. On top of this, the approach of disabling interrupts would not work on a system with multiple processors. Finally this approach at implementing a lock is simply inefficient, however that is a minor point compared to the other issues an interrupt based lock would introduce.

Overall, this approach is only used in very limited contexts such as an operating system disabling interrupts to guarantee atomicity when accessing its own data structures.

Hardware Instructions

Alternative methods for implementing locks involve hardware supported instructions that are atomic so that the locks can ensure they are mutually exclusive. Some of these instructions include Test-and-Set, Compare-and-Swap, and Fetch-and-Add.

Test-and-set works by making both the test of the old value and the assignment of the new value a single atomic operation, which ensures atomicity however this will not work on a non-preemptive processor as the process will be stuck spin-waiting. Test-and-set is correct however it does not guarantee fairness nor does it have great

performance, the performance is acceptable for multiprocess processors where the number of threads match the number of cores, for other scenarios such as a single processor the performance is bad as there is a lot of spin-waiting involved causing processes to hold their time-slices when they cant do anything.

Compare-and-swap is similar to test-and-set, except it takes an additional argument which is the expected value, it will only perform the "set" if the actual value is equal to the additional expected value. This approach also requires spin-waiting to obtain the lock, and has the same fairness and performance problems as Test-and-set.

Fetch-and-swap takes a reference to a value, increments it and returns the old value (value before it was incremented) within a single atomic instruction. This can be used to create Ticket locks where the lock has a ticket and a turn, while spin-waiting the lock will obtain a ticket number and wait for its turn. This fixes the fairness problem as obtaining the lock becomes first-come first-serve preventing starvation. However the performance is still not great on single processors.

Yield

The problem with all the hardware instruction methods alone is that they all require spin-waiting which inevitably leads to poor performance especially on single-threaded machines.

The solution to this is an operating system primitive that allows the thread to surrender its CPU time and allow another thread to run, this operating system primitive is Yield.

Yield is implemented into a program by modifying the typical spin-waiting approach to utilize it. Instead of continually checking the condition in a loop wasting CPU time, you instead check the condition, if you need to wait call yield, then loop so that when the processor gives you more time you check the condition again and repeat. By letting the operating system know when you don't need to be using the processor there can be significant performance gains.

Queues

While the ability to yield was a significant improvement over wasting time-slices, it is still not perfect. Imagine there are 100 threads contending for a lock, in this scenario after one of the threads obtains the lock there are still 99 threads that will be assigned a time-slice that will be yielded anyway, constantly yielding time-slices can still eat up performance that could otherwise go towards jobs capable of progressing.

To address this, we need a little more support from the operating system that will allow us to implement queues. This is achieved through calls to `park()` which puts the calling thread to sleep, and `unpark(threadID)` which will wake up the thread with the associated ID. Together these calls can be used to build a lock that puts threads to sleep when they request a lock and they can be awoken when it is their turn to acquire the lock.