COS40003 – Concurrent Programming

Report 1

Weeks 1 – 6

Daniel Bowling

101616414

Week 1

Read, understand, and summarize the following computing paradigms: concurrent computing, parallel computing, distributed computing, cluster computing, grid computing, cloud computing, fog/edge computing, etc. (any more related you find?)

Concurrent Computing

Concurrent computing is a paradigm where by computations or tasks are executed concurrently. This means that two or more tasks are processed during overlapping time periods but not simultaneously, even though they may seem to occur simultaneously to the user. This can be due to limited resources such as the computer having only a single CPU. In this instance only one task will be in progress at any one time. The order with which each of these tasks is processed is dependent on the systems architecture.

Parallel Computing

Parallel computing is a paradigm where by computations or tasks are executed in parallel or simultaneously. This is achieved by breaking down tasks into smaller sub tasks and having those tasks processed independently and simultaneously. Once the processing of these sub tasks is complete, their results are combined. Parallel computing systems achieve this by sharing a single memory space and processing tasks across multiple processors. Examples of parallel computing hardware include GPU's and multicore CPU's.

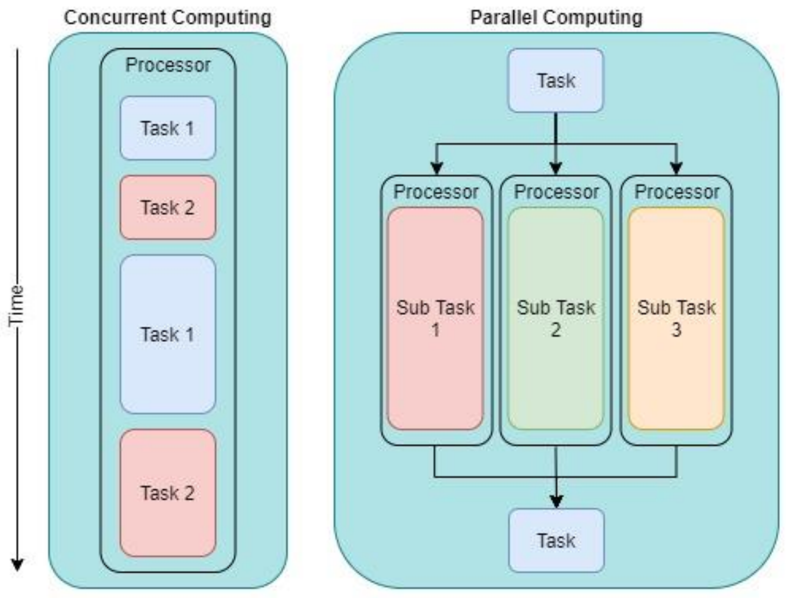

Figure 1 - Concurrent VS Parallel Computing

Distributed Computing

Distributed computing doesn't rely upon any universally agreed upon definition. This paradigm typically comprises several independent computers working in concert as a single system. Each of these independent computers has its own memory and processing resources. These computers form a distributed system via message passing between one another to carry out computations.

Cluster Computing

Cluster computing is typically although not exclusively comprised of a "cluster" of low-cost computers. Each of the computers within the cluster are connected via a fast local area network. The cluster is collocated to deliver high performance computing.

Grid Computing

Grid computing is similar to that of cluster computing apart from the fact that each computer is networked and dispersed amongst many geographic locations. Grid computing has now been superseded due to the advent of cloud computing.

Cloud Computing

Cloud computing provides its services to end users only. It completely shields users from the underlying hardware, complexity and computing tasks. These tasks are provided in three main formats.

- Software as a service (SaaS)
  Typically used to deliver applications as a service to end users via the internet
- Platform as a service (PaaS)
  Delivers a framework for developers to produce applications hosted by the platform provider.
- Infrastructure as a service (IaaS)
  Delivers infrastructure to the end user via the internet for large complex computing tasks.

Each of the service types above allows users to utilise and pay only for the services that they require.

Week 2

Read, understand and summarise "process", and put your summary into Report I

A process is an abstraction that encapsulates a single program's execution. Likewise, processes are a collection of individual programs currently in action. The benefit of processes is that they allow multiple commands or programs to execute without constant or ongoing input from the user. The way that processes allow multiple programs to execute is by assigning them resources such as memory. This is maintained by assigning each individual process its own private virtual memory space. Each and every process encapsulates the following.

- An image of the program being executed
- A virtual memory space consisting of
  - Program instructions (i.e. code)
  - Static data (i.e. global variables)
  - Heap memory
  - Stack memory
- The state of the CPU which includes
  - The state of each register
  - The state of the program counter
  - The state of the stack pointer
- The state of the operating system, such as
  - Opened files
  - Accounting statistics
  - Etc.

Processes can be in one of three main states, running, ready and blocked. The definition for these three states and their allowed transitions is as follows.

- Running
  If a process is in running state then the process is currently being executed by a processor. From this state the process can be de-scheduled to ready state by the OS or initiate an I/O operation which transitions it to blocked state.
- Ready
  If a process is in ready state it is waiting to be processed. It transitions to running state once scheduled by the OS.
- Blocked
  The process has been transitioned to blocked from running state and is now waiting for an event to occur such as an I/O request. Once received it transitions to ready state.

In addition to these three states is Initial or Created state which is when a process has been created, as well as Final or Terminate state which is when a process has been terminated and is awaiting clean up by the OS. The Final state allows the parent processes to examine whether or not a child process has terminated under fault or as expected via return codes. The following figure illustrates each of these process states and the conditions for transition.

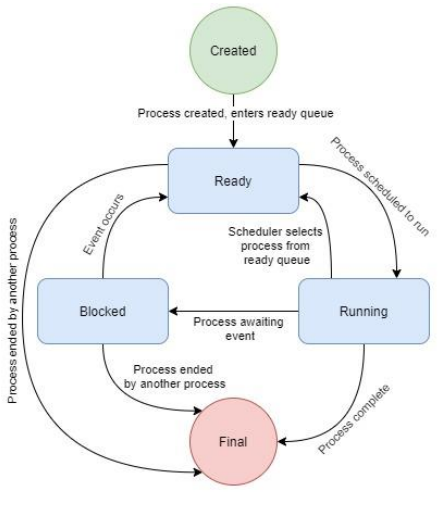
Created
Process created, enters ready queue
Ready
Process scheduled to run
Event occurs
Scheduler selects process from ready queue
Process ended by another process
Blocked
Process awaiting event
Running
Process ended by another process
Process complete
Final

Figure 2 - Process State Transition Diagram

Week 3

Read, understand, and summarize the following scheduling approaches: first come, first server (FIFO); shortest job first (SJF); pre-emptive shortest job first/shortest remaining time/shortest time-to-completion first (PSJF/STCF), round robin, lottery scheduling, multi-level feedback queue.

First come, first serve (FIFO)

The FIFO scheduling paradigm executes tasks in the order that they arrive. This is achieved by placing each task in a queue. As a task is dequeued and executed, it is allowed to run to completion before the next task in the queue is selected by the scheduler. This approach often leads to compounding delays as more and more tasks arrive in the queue. This is due to the fact that the only method for task selection is selecting the first task in the queue, ignoring any tasks expected run-time duration.

Shortest Job First (SJF)

This scheduling paradigm executes tasks in order of their expected runtime. There are two methods for executing this scheduling method. Non-pre-emptive SJF or Shortest Time to Completion First (STCF), and Pre-emptive SJF (PSJF).

- STCF
  This method inspects the expected run-time of each task in the queue, it then selects the task with shortest expect run time. As more tasks arrive in the queue, the runtime for all queued tasks is evaluated and their order sorted from shortest to longest. This method ignores the runtime of the currently running task allowing it to finish before selecting the next task from the sorted queue. This approach still leads to compounding delay times, although to a lesser extent than that of the FIFO method.
- PSJF
  This method also inspects the expected run time for each task in the queue, the difference here being that it also evaluates the remaining runtime for the currently executing task. The task with the shortest reaming runtime is selected to be executed, even if that interrupts the current task. This approach again leads to compounding delay times, although to a lesser extent than that of FIFO and STCF.

Both STCF and PSJF also have the added disadvantage of further compounding the delay time of any tasks which have a greater expected runtime than any other that arrives in the queue.

Round Robin

This scheduling method introduces a time quantum or time slice for the purpose of allocating tasks CPU time. A time slice is a discrete period of time, typically within the range of 1 to 100ms. It also introduces a ready list for queuing tasks as they arrive. Each task in the ready list is assigned a time slice, the first task in the queue is allocated that time slice to execute before the scheduler moves onto to the next task in the ready list. When the end of the ready list is reached, the scheduler returns to the front of the ready list to perform these operations over again in the same fashion. If a task is completed during a time slice, it is removed from the ready list. This method of scheduling leads to greater delay times than any other method discussed thus far. This occurs as the number of tasks in the queue increases, the longer each task has to wait before the scheduler returns to its place in the ready list. In addition, as the time quantum increases, so too does the time it takes to return to each task in the ready list leading to longer response times. A reduction in the time quantum does lead to faster response times although this requires greater overheads. The one advantage that the round robin method has over the others mentioned thus far is that it equitably allocates CPU usage time to each task ensuring that no one task is disadvantaged due to its expected runtime.

Lottery Scheduling

Lottery scheduling allocates CPU time by allocating a "lottery ticket" to each task in the queue. The scheduler then selects a lottery ticket at random, the winning task is scheduled. Any task can be given a number of lottery tickets, each ticket that a task has when assessed against the total number of tickets allocated to all tasks represents its probability for being selected by the scheduler.

Multi-level feedback queue (MLFQ)

This scheduling method introduces a number of queues as rows, each with a different level of priority. It also includes the time slice exhibited by the round robin method. The method for which each task is assigned to a queue and ultimately scheduled follows the following rules.

- Rule 1: Tasks are scheduled in order of priority.
- Rule 2: Any tasks that share the same priority level are scheduled in a round robin fashion. This ensures that multiple tasks in the same priority level are given equal CPU time.
- Rule 3: Each task enters the system at the highest priority level.
- Rule 4a: If a task utilises an entire time slice during execution its priority level is reduced. This allows shorter tasks to be completed quicker than longer tasks.
- Rule 4b: If a task releases the CPU before the end of a time slice, it remains in the current priority level. This ensures that quick response tasks such as I/O operations do not have their priority reduced leading to longer response times.

The above rules ensure that shorter response time tasks remain as such and that longer runtime tasks are sacrificed at their expense falling in priority. This does however lead to problems, the two key problems being Starvation and Gaming the Scheduler.

- Starvation
  This occurs when multiple short interactive tasks such I/O operations continuously arrive. These tasks are assigned to the highest priority level and allowed to execute at the expense of lower priority tasks. These short I/O tasks are typically completed within one time slice meaning they stay at the highest priority level. This means that lower priority tasks are starved of CPU time.
- Gaming the Scheduler
  Longer runtime tasks can be designed so that their processing ends before the end of each time slice. This would ensure that it stays at the current priority level and given more CPU time than that of lower priority tasks.

These problems have led to the revision of Rules 4a and 4b as well as the introduction of a 5th rule.

- Rule 4: Once a task has used up its time allotment at a given priority level, regardless of how frequently it releases the CPU within a time slice, its priority level is reduced. This eliminates the ability of a task to game the scheduler.
- Rule 5: After some number of time slices all tasks are moved to the highest priority level. This relieves the issue of starvation.

In order to distribute CPU allocation among tasks within varying priority levels, the widely accepted rule is to vary the duration of the time slice depending on the level of priority. The higher the priority level the shorter the time slice, the lower the priority level the longer the time slice. In assigning priority to each task, the MLFQ scheduling method observes the execution of each task rather than each task implicitly demanding a high priority level. All of these mechanics results in high performance for short tasks such as interactive I/O operations and fair to moderate performance for longer runtime tasks. These features see MLFQ implemented in many OS's such as BSD, UNIX, Solaris and Windows.

Week 4

Read, understand, and summarize thread and related material in Lecture 4.

Just as a process is a running program, a thread is a running subtask within a process. A process can encapsulate one or multiple threads. An example of this in action is a Graphical User Interface (GUI) designed to play videos. The user interacts with the GUI by pressing the play button, stop button, etc. In this instance the video runs in one thread, whilst the user's input is run in another, i.e., the user presses the play button and the video plays, they press stop and the video stops. So just as processes allow OS's to multitask, so to do threads allow processes to multitask. This dynamic sees processes as the basic unit of memory allocation, with threads the basic unit of CPU scheduling.

Sub tasks could be handled by multiple processes rather than threads, but doing so would require greater resources as each process requires its own memory space and sharing data between these memory spaces can lead to data corruption as registers are updated. Breaking a process into multiple threads allows them to handle their own subtasks as a stream of instructions. These threads operate within the same memory address space hosted by the encapsulating process. This enables them to more readily share data and the fact that these threads inhabit the same memory address means that data is shared quickly and the integrity of the data is better ensured. These characteristics means that threads have a lower communication cost when compared to that of processes. Threads are best suited to applications where communications and blocking needs to be overlapped. Returning to the video GUI example, in this instance threads best handle communications and blocking as one thread can be paused to allow another to run. Say for instance the user presses pause and then drags the scroll bar to another point in the video. Here the pause button thread has blocked the video thread, the scroll bar thread then allows the video to resume at a different point. All this achieved quickly and with low overheads thanks to the threads. If a multicore system is introduced to the system, then multiple threads can be run in parallel leading to even faster response times.

In order to facilitate multiple threads within the one process, each thread inhabits its own stack space within the parent process' memory address space. In addition, multiple CPU contexts need to be created such as the Program counter, Stack Pointer, Register States and as mentioned Stack. As such threads introduce control flow into the system. This means that in practice a thread is a processor context and stack. Each thread is scheduled for CPU time and executed within the parent process' memory address space.

In order to maintain multiple threads each has its own private or per-thread state, as well as a shared thread state to maintain all threads encapsulated within a process. The information that each of these states maintains is as follows.

Private per-thread state

- CPU context
  - Program Counter
  - Stack Pointer
  - Registers
  - Stack, which contains local variables, etc.
  - Scheduling properties (i.e. priority).

Shared Thread State

- User ID, Group ID, Process ID
- Memory Address Space
  - Program instructions
  - Static data (i.e. global variables)
  - Heap memory which contains dynamic data
  - Stack memory
  - Currently open files, sockets, locks, etc.

In practice threads split programs into routines that can execute in parallel, either true parallelism when dealing with multiprocessors or pseudo parallelism when dealing with single processors. These threads can then be scheduled in a variety of manners such as MLFQ, round robin, etc. There are two key strategies for dealing with threads. They are;

- Manager/Worker strategy
  The manager thread handles I/O and assigns work to the worker threads. These worker threads can either be created dynamically of assigned from a thread pool.
- Pipeline strategy
  Each thread works on an "assembly line". Each thread is assigned and completes a task before handing it off to the next thread in the assembly line.

Threads have both their advantages and their disadvantages. Threads allow for the overlapping of I/O operations and computation. They also provide cheaper context switching and better mapping to multicore processors. With this said they do add complexity to processes. They can be difficult to debug, can add undesired complexity within multithreaded programs which can lead to issues with backwards compatibility, they can introduce undesirable thread interactions, and without the use of lock or synchronise functionality can lead to race conditions.

Week 5

Read, understand, and summarize lock and related materials in Lecture 5.

Locks allow atomic actions to be executed, that is they allow a task to execute and complete otherwise the task does not run at all. This guarantees mutual exclusion between threads by locking them once they have entered a critical section. This locking is achieved by allowing multiple threads to possess one shared lock. If a thread possesses the lock no other thread is allowed to execute until that lock is released at which point another thread can retrieve and hold the lock. This is typically achieved on a first come, first serve basis and the threads try to retrieve and possess the lock as they enter a critical section. An analogy for this is that of a phone booth. If the phone booth is empty, then one person can enter and use the phone. If the phone booth is occupied, no one else can enter until the occupant exits the booth. Here the phone booth represents the lock, and each person a thread attempting to enter a critical section.

In practice locks are achieved via two different means with two different locking functionalities. A Lock object can be initialised or by using the keyword *Synchronize*.

Lock

Lock variables are declared as defined by the appropriate header file. In C they are defined as a struct and in Java as an object. The lock object knows which thread possesses the lock and maintains a queue of those threads awaiting lock acquisition. The critical section of code is encapsulated within a try block. The lock method is called before `try` and the unlock method is within the `finally` block. See Java example below.

```
import java.util.concurrent.locks.ReentrantLock; //Lock header file
private ReentrantLock lock = new ReentrantLock(); //New Lock object
    public void run() {
        lock.lock(); //Try to acquire lock
        try {
            //Critical section of code
        }finally {
            lock.unlock(); //Release lock
        }
    }
```

Synchronized

All objects in Java have an intrinsic lock that can be exploited by using the *Synchronize* keyword. The lock is initially unowned. As long as any object owns the lock, no other lock can possess it, if it tries it is blocked. This is achieved in two ways. The first is by declaring *Synchronize* before the return type of a method declaration. This forces the current thread to attempt to acquire the objects intrinsic lock before entering the function.

```
     public synchronized void run() {
        //Critical section
    }
```

Here everything within the method declaration is treated as a critical section meaning the intrinsic lock is only released once the entire function has completed execution. The second method includes using *Synchronize* and casting to it the current object before the start of a separate block of code within a function.

```
//Synchronize method 2
     public void run() {
        //Some code
        synchronized(this){
            //Critical section
        }
    }
```

Here only the code noted as the critical section within the *synchronized* block above is treated as such.

In both instances above, whenever a synchronised method exits, it automatically establishes a "happens before" relationship with any subsequent invocations of a synchronised method for the same object. This ensures that changes to the state of an object as a result of a *Synchronize* method are visible to all threads. Having one lock that maintains all critical sections that appear within a process could lead to vast delays in execution. As such, both the lock and synchronise methods allow for multiple locks. This allows different data and data structures to be protected and maintained.

From the code snippets above it can be seen that using the *Synchronize* keyword is far simpler than utilising a lock object. That said Locks are preferred over the *Synchronize* keyword. This is because the Lock object delivers more functionality through additional lock object methods.

- `Lock.tryLock()` which returns a Boolean can be used to see if a lock can be acquired. If not then something else can be done without having to wait for the lock. `tryLock()` can also have parameters passed to it defining the amount of time to wait for the lock before moving onto to another task if it is not acquired (i.e. `lock.tryLock(50, TimeUnit.SECONDS).`
- `Lock.lockInterruptibly()` which is a void function can be used to interrupt a thread under user defined circumstances.

Week 6

Read, understand, and summarize lock II and related materials in Lecture 6.

Static variables are shared by all instances of a class, as such locks do not provide mutually exclusive access to static variables by default. This can be achieved however by declaring the lock object as static OR by using *Synchronized* and casting the class of the object. I.e.:

- `Static ReentrantLock Lock = new ReentrantLock()`
- `Synchronized(ClassName.class){//Critical Section...}`

There are multiple ways of implementing lock functionality. However, any lock functionality should aim to satisfy the following criteria

- Mutual Exclusion/Correctness
  This prevents multiple threads from accessing the critical section and corrupting the underlying data.
- Fairness
  This ensures that all threads are given equal access (or as close to as possible) to the lock.
- Performance
  This ensures that locks do not unnecessarily increase overheads whilst decreasing system performance.

The following examples detail different methods of implementing lock functionality and assesses their effect against the above criteria.

Controlling interrupts

Controlling interrupts by disabling them in the instance of lock, and enabling in the instance of unlock is a poor method for implementing lock functionality as it places far too much trust in applications. Any program could request a lock and take control of the CPU or send all processors into an endless loop. Disabling interrupts also will not work on multiple processors. If a thread disables interrupts on one processor, all other processors are enabled by default allowing other threads to run on them. Disabling interrupts for an extended period of time can also lead to catastrophic system problems. For example, if a CPU misses the completion of a disk read request, the OS will not know when to wake the process waiting for the response. Finally, this method is an extremely inefficient method of implementing lock functionality.

Load and Store

The load and store method utilises flags to maintain a record of which thread possesses the lock.

```
//Load and store
typedef struct _lock_t{int flag;} lock_t;

void init (lock_t *mutex){
    //0 -> lock is available, 1 -> lock is held
    mutex->flag = 0;
}

void lock(lock_t *mutex){
    while(mutex->flag == 1); //Test flag
    //Spin/Wait
    mutex->flag = 1//Set flag
}

void unlock(lock_t *mutex){
    mutex->flag = 0//Set flag
}
```

The issue with this method is that it does not ensure mutual exclusion. This is because the process of testing the flag via the while loop and then setting the flag as seen in the `lock()` method is not atomic. This means that the flag can be set to 1 by other threads before the current thread has had the opportunity to do so.

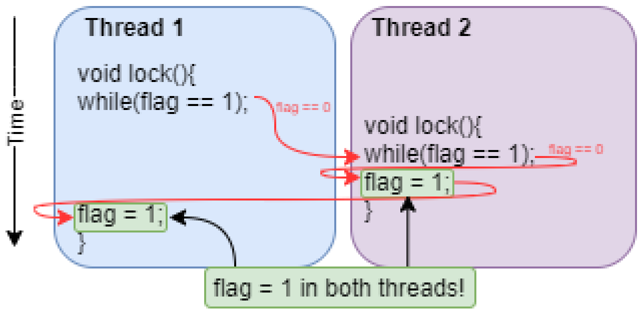

Figure 3 - Load and Store mutual exclusion failure

It is also extremely inefficient as resources are constantly utilised checking if the flag is set via the while loop. This is referred to as spin waiting.

Test and Set

This is an extension of the Load and Store method with the addition of a test and set method. Test and set ensures that the operation of checking and then setting the flag is a single atomic action.

```
//Test and set
int TestAndSet(int *old_ptr, int new){
    int old = *old_ptr; //Retrieve value at old_ptr
    *old_ptr = new; //Replace value at old_ptr with new
    return old;//Return old value
}
```

It achieves this by reading and checking the old value at the memory location pointed to by `old_ptr` whilst updating that same memory location simultaneously. This updating of the old value occurs regardless of whether or not it was already set.

```
//Implementation
typedef struct _lock_t{int flag;} lock_t;

void init (lock_t *lock){
    //0 -> lock is available, 1 -> lock is held
    lock->flag = 0;
}

void lock(lock_t *lock){
    while(TestAndSet(&lock->flag,1) == 1);
    //Spin/Wait
}

void unlock(lock_t *lock){
    lock->flag = 0//Set flag
}
```

The above implementation does ensure mutual exclusion as only one thread can possess the lock at any one time. However, it is not an appropriate solution for non-pre-emptive processors due to the executing thread always spinning and waiting. It is also an inefficient method of locking as each thread will occupy an entire time slice on the CPU spinning and waiting although this can be alleviated if the number of processors is approximately equal to the number of threads.

Compare and Swap

Unlike test and set, this method does not set the flag every time it is called. Rather this method compares the actual value against the expected value and only if they match is the flag set.

```
//Compare and swap
int CompareAndSwap(int *ptr, int expected, int new){
    int actual = *ptr;
    if(actual == expected)
        *ptr = new;
    return actual
}
```

This makes compare and swap a more powerful method than test and set, although its efficiency still falls short when used on single processor systems.

```
//Implementation
void lock(lock_t *lock){
    while(CompareAndSwap(&lock->flag,0,1) == 1); //Test flag
    //Spin/Wait
}
```

Fetch and Add

Fetch and add simply adds to the value pointed to by `ptr`. The result is a “ticket” and “turn” variable that implements the locking functionality.

```
//Fetch and add
int FetchAndAdd(int *ptr){
    int old = *ptr;
    *ptr = old + 1; //Increment ticket number for next thread
    return old; //Return current ticket number
}
```

The above function allocates tickets to threads. The current thread is allocated the current ticket value before it is then incremented for the next thread.

```
//Implementation
typedef struct _lock_t{
    int ticket;
    int turn;
} lock_t;

void init(lock_t * lock){
    lock->ticket = 0;
    lock->turn = 0;
}

void lock(lock_t *lock){
    int myturn = FethAndAdd(&lock->ticket);
    //when ticket == turn, acquire lock
    while(lock->turn != myturn);
    //Spin
}

void unlock(lock_t *lock){
    //Increment turn after every unlock
    lock_turn = lock->turn + 1;
}
```

The unlock function increments the turn variable each time it releases the lock. The lock function then checks the value of the threads ticket against the turn value. If they match, then that thread acquires the lock. Whilst this method delivers correctness and fairness it still falls short of delivering on performance especially on single processors.

Whilst all of the methods mentioned thus far deliver locking functionality with test and set, compare and swap as well as fetch and add even providing fair performance when used on multiprocessors, the issue of context switching still remains. In each of these examples the thread that possesses the lock could endlessly utilise the critical section starving other threads and causing them to endlessly spin. All of these methods have employed hardware alone to implementing locking functionality. However, to deliver locking functionality that provides greater efficiency, some functionality only provided by the OS is required.

Yield()

It is assumed that the OS has a primitive that allows threads to yield control of the CPU.

```
void lock(lock_t *lock){
    while(TestAndSet(&lock->flag,1) == 1);
    yield(); //Release CPU
}
```

`Yield()` essentially deschedules the thread from running state to ready, promoting the next thread to running in the process. Whilst this approach alleviates some performance concerns an issue still remains. If the thread that holds the lock is pre-empted by the system before releasing it, then each of the remaining threads will then call lock only to find that it is currently held by another thread. This isn't of great concern if only dealing with a small number of threads, but if there are a large number of threads, then the CPU will be occupied by a large number of threads calling lock whilst it is held by another thread leading to delays.

Queues

Using a queue allows threads waiting for a lock to be queued on a first come first serve basis as they attempt to acquire the lock. Additional methods provided by the OS also allow threads to sleep rather than spin.

- `park()` puts a thread to sleep
- `unpark(threadID)` wakes the thread designated as `threadID`

These two calls can then be used to put a thread to sleep anytime it tries to call an occupied lock and wake it once the lock is available.

The characteristics when assessed against the locking criteria for each of the locking implementations mentioned herein is summarised in the table below.

	Correctness	**Fairness**	**Performance**
Controlling Interrupts	No	No	No
Load and Store	No	No	No
Test and Set	Yes	No	Bad for single processor. Ok for multiprocessor.
Compare and Swap	Yes	No	Better than test and set but still bad for single processor. Ok for multiprocessor.
Fetch and add	Yes	Yes (First come first serve)	Better than test and set but still bad for single processor. Ok for multiprocessor.
Yield() (in concert with other methods i.e. Test and Set)	Yes	Yes	Yes (although a large number of threads can cause delays if a thread is pre-empted before releasing the lock)
Using queues (in concert with other methods i.e. Test and Set)	Yes	Yes	Yes (with additional logic and methods (i.e. `park()`, `unpark()`))