COS40003 - Concurrent Programming

Learning Summary Report

Luke Rep-Baihelfer (100599155)

Report I – Learning summary report – 30%

Assess what you have learnt during the semester. This includes lecture materials/lab tasks. In your own words, you have to describe what you have learnt. One purpose of this report is to have a record of what you learnt in this unit. A record that you can refer to in future – when you work industry or research. In other words, first, write the report for yourself and then for the marks (the usefulness of the report is important and directly impact on your mark). You can also present areas that you have personally explored beyond the expectations of the unit, as well as indication of the areas where you plan to learn further on your own. Here is the submission guide for learning summary report.

Scope of the report:

Week 1 – Week 6

Submission:

(1) A single PDF to be submitted to canvas, maximum 15 pages; (more pages can be allowed after approval)

(2) Sections organized by Week 1, Week 2, … , Week 6.

Marking guideline:

(1) Completeness: whether necessary points are covered;

(2) Correctness: whether concepts are discussed correctly;

(3) Insight: explored insight will add bonus;

SWINBURNE UNIVERSITY OF TECHNOLOGY

Week 1 (Concurrent Programming Introduction)

Overview

This first week was focused on bringing us up to speed with what is concurrent programming, by exploring various types of concurrent programming with real world examples.

Concurrent Computing

The term of concurrent computing is often used from the perspective of an observer/user - where they **perceive** that a machine is performing multiple actions at the same time.

However, it may be the case that the machine only has a single core and is actually **taking turns** with the various processes running. When the processes do not take turns then this would be considered sequential computing - where a task runs from beginning to end, uninterrupted.

In both concurrent and sequential computing, there is never a case where multiple instructions are being executed **simultaneously** - just the way concurrent computing works it can make it **appear** that multiple operations are occurring simultaneously.

Parallel Computing

When a machine is physically running **multiple** instructions at the same time, this is considered parallel computing and is often utilised in multi-core processors.

The benefits of parallel computing is that they are able to break certain tasks up into smaller tasks, then have those tasks executed in parallel and finally have their results joined together to get the output.

This can provide moderate to substantial **speed benefits** depending on the task and the hardware available. A modern example of this is cryptography applications using GPUs as they have many cores.

Distributed Computing

The usage of multiple machines which have means of communicating with one another to complete a common task - similar to that of parallel computing.

This does differ from **parallel computing** where the processes have access to some form of **shared memory**, because in **distributed computing** each machine has its **own memory.**

There can be some latency issues due to the need to **send messages** between machines and in some cases there can be a large geographical gap - causing an increase in the physical network latency.

Cluster Computing

Similar to distributed computing as there are multiple physical machines (nodes) involved, however cluster computing has these nodes connected **locally** - allowing for faster communication.

The **cluster** is commonly built from **many** affordable & low powered devices - such as Raspberry Pi's and other single board computers and can be viewed/considered a single machine once setup.

Cluster computing is often how **supercomputers** are constructed as there are increasing difficulties on the limits of transistor packing. So an efficient and effective way of increasing computer power is to connect multiple machines.

Grid Computing

A form of distributed computer where multiple machines - **often more geographically distant** compared to cluster computing.

Where cluster computing is focused on all the nodes working on a common task (highly coupled), grid computing can have various nodes working on **different tasks** (loosely coupled).

Grid computing has been in a **decline** as people have been shifting towards **cloud computing**.

Cloud Computing

Exists to handle the computation behind a **cloud server** - allowing users from almost anywhere to utilise the computing powers without any of it running on their machine.

Not only used for **processing large complex tasks** but can also provide services such as (SaaS) Software as a Service, (PaaS) Platform as a Service and (IaaS) Infrastructure as a Service.

There are many providers out there which allow you to use their cloud computing services in an **on demand** manner, meaning if you suddenly have a large spike in demand - the provider will provide **additional resources** at a cost to handle the load.

Fog/Edge Computing

Technique which focuses on **moving** infrastructure/computing **resources physically closer** to the devices which are making the requests.

Often setup as a layer between the end devices and a cloud system. It can **reduce latency and bandwidth** as the devices do not need to communicate all the way to the cloud for every single request.

Week 2 (Process)

Overview

This week we were introduced to what a process is and how they are created and managed.

Process States

A **process** is an **instance of a program** that is actively running - where a **program** is simply a **set of instructions**. Each process is typically assigned a process id (PID) which can be used to keep track of and manage processes.

There can be **multiple processes** running on a system and depending on the system's behaviour and architecture they may run sequentially, concurrently or in parallel.

There can be even **multiple process instances** of the **same program**, each which have their own process address space which consists of the:

- Program (program code that the process is executing)
- Stack Space (used to keep track of what is being executed such as function and system calls)
- Data Space which stores the values of variables which can be both statically defined (stored on the stack) and dynamically created (stored on the heap).

Process States

A process can exist in various states and **transition** between them as necessary. See figure 1 to understand the lifecycle of a process with the following description of each state.

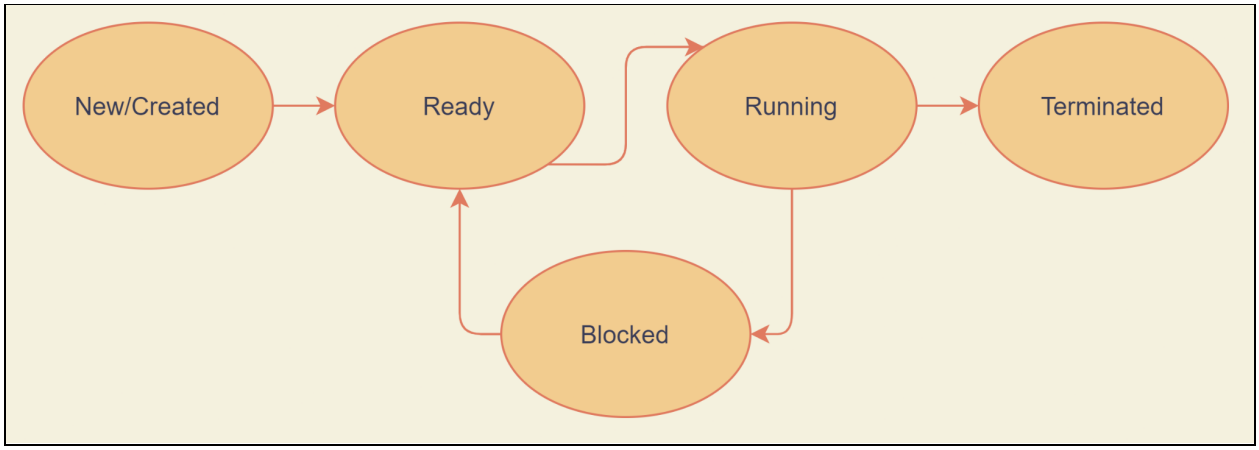

Figure 1: Process state diagram

New/Created

The state when the process is first started and is waiting to be set to the ready state.

Ready

The process is available to run, but it is not currently running for reasons such as being de-scheduled or blocked by an interrupt (user input, file handling, etc).

Running

The process is actively executing instructions.

Blocked

A process can be blocked when an interrupt needs to be handled - these interrupts will often take priority to ensure smooth interaction between the processes and also the user with the system.

Terminated

A process is considered terminated when it has executed everything it needed to, or it has been terminated by the OS/user.

The program is no longer running/executing instructions, however the process exists in what is considered a “zombie” state.

The purpose of having it work this way is to allow parent processes to receive the exit code which may indicate success, failure or some other information of the process.

Process APIs (C Programming)

fork()

Used by a process to **spawn additional instances** of itself, which may be useful to take advantage of multithreaded/parallel processing for **performance benefits** or to **segregate processes** to have their own program space, such that if a forked process crashes, it doesn’t crash the main parent process.

A forked process is able to **identify** if it is a parent or a child process, depending on the PID value of the fork() call. A PID of 0 is returned to the child and the PID of the child is returned to the process.

If not handled correctly or intentionally **maliciously crafted**, you can have your process create a “fork bomb” which essentially r**ecursively forks** the process such that it consumes **all** the system’s resources.

wait()

Used to wait for certain forked processes to complete execution.

exec()

Used to replace the existing process with a process running a **different program**.

The **PID** remains the **same**, but what is being **executed** is **different** - including the program space as this needs to be able to **hold the new program** being loaded.

Week 3 (Scheduling)

Overview

This week was focused on scheduling of processes, covering various different techniques and understanding when some are better than others.

Also covered preemptive vs non-preemptive, where a preemptive system is one which is able to suspend/interrupt tasks, to perform other actions/run other processes which comes in very useful for scheduling.

First-Come First-Served (FIFO)

A very **simple** scheduling technique which accepts jobs in a **queue** like fashion, where they are executed in the **order** they were received. It is a non-preemptive scheduling algorithm which means jobs are executed from beginning to end.

There are some downfalls to this algorithm, due to that fact that this is a non-preemptive algorithm if a job gets **stuck** or runs for a **long period** of time, it will be **delaying** all the other jobs in the queue.

Also with the queue system, even if there are many small jobs these all need to wait until all the tasks in front of them complete - causing big problems if you are trying to **poll** a device or handle **user input**.

Shortest Job First (SJF)

Another simple non-preemptive scheduling technique, where the scheduler selects the next job to run based on the smallest estimated/expected **execution time** of the jobs in the pool.

When the current job has finished, the scheduler will look at its pool of available jobs and select the one that has the **shortest** execution time. If multiple jobs have the **same** shortest execution time, they will usually pick the one that came **first**.

This provides the benefit **minimising** the delay for shorter jobs as they will not need to wait for a potentially **long** queue to finish executing before they are executed. However, it can have an **impact** on **long jobs** in the case where many **frequent smaller** jobs are being added to the pool - leaving the longer jobs starved.

Preemptive Shortest Job First (PSJF)/Shortest Remaining Time

A preemptive version of SJF, where jobs are able to be **suspended** in order to jump/switch to shorter jobs.

This technique heavily **favours** short jobs and with that comes the issue of **starvation** again, as a long job can be constantly interrupted with shorter jobs being added to the pool.

Round Robin

This technique focuses on “fairness” by **cycling** between the various jobs at a set **time slice**/period. By doing so it also assists with reducing/preventing **starvation** but at the cost of giving **equal processing time** to each job.

There are some **inefficiencies** with this technique which are noticeable when there are a **large** number of jobs, due to the cyclic nature of Round Robin and also the **overhead** of **context switching** when the jobs are preempted so a small time quantum will **increase** the response time but **reduce** the frequency of context switches which makes it a complicated technique to **balance**.

Lottery

A technique which is based around “random” selection which utilises a **ticket** idea to give fair weighting to various jobs based on the **proportion** of tickets they have assigned.

The scheduler will draw a random ticket number at a certain time quantum and whichever job holds that ticket is executed for a set period of time before another ticket call is processed.

It avoids guaranteed starvation by giving each job at least 1 ticket, as the job will always have a chance of being picked - even if the chance is very slim.

Multi-Level Feedback Queue (MLFQ)

Works to solve the issue that SJF and PSJF have, as they do not know how long a certain job will run for.

The basis of this technique uses **multiple queues** with **different priorities** to keep **track** of jobs and ensure short tasks are executed **promptly** and longer tasks do not get **starved** via a priority boost mechanism.

Jobs are **first** added to the **highest** priority queue and if they are not complete by the end of their time quantum, they are **dropped** down to the next priority queue. If multiple jobs exist in the **same** priority queue, they are executed in a round robin fashion.

After a while longer tasks will eventually be **dropped** to the lowest priority queue and will be **vulnerable** to **starvation**, this is where the priority boost mechanism comes in and after a **set time** the **low** priority tasks will be sent back up to the high priority queue.

Priority Scheduling (Researched)

This exists in both preemptive and nonpreemptive versions. The premise is that the scheduler will have various **priorities** assigned to jobs and will **focus** on executing the **highest priority** jobs in a round robin fashion.

This technique does suffer from **starvation** in the case of many high priority tasks **flooding** the scheduler - not allowing **sufficient** time for lower priority jobs to run.

Week 4 (Threads)

Overview

This week we focused on the idea of threads, what they are and how to manage them correctly.

Threads

Threads are what run beneath the hood of a program. All programs have **at least one t**hread, but they can utilise many more to **improve** interactivity/response time, performance or to just simply do more things at the same time.

While process scheduling allows multiple programs to appear to be running at the same time, thread scheduling allows threads within a program to **appear** to be running at the same time. Modern processors now also support multi-threading which allows these threads to run in parallel at the **hardware** level.

If a program only had a single thread, it would not be able to handle user input while rendering a video at the same time - **without** significant interruptions/delays.

Threads also share the same resources and memory addresses as its parent process.

Thread Pooling

A technique which can reduce **overhead** by setting up a pool of threads beforehand in anticipation of future demand/need for those threads. The main benefit is not needing to **wait** for the threads to be **instantiated** before being used - as they are already instantiated and can just be handed their task and run.

Setting up the correct pool size is a tricky task to get correct because having too few can **increase** the likelihood of **starvation** and having too **many** will increase the amount of system resources **required**.

C APIs

pthread_create

This creates a new thread and returns the thread ID. Also accepts a parameter for a routine to be executed.

pthread_exit

Terminates/ends a given thread

pthread_join

Tells a specific thread to wait until another given thread completes, so that they “join” up time wise.

pthread_yield

Gives up the CPU to be put back into the queue - this can be useful if a thread is taking a **long** time to finish.

Java APIs

Extending Thread class

When implementing the Thread class you must set up/override the run() method and when this ends the thread ends.

You make the thread ready to run by calling .start() on it.

Something to note with Java is that you can only extend a single class - so if you wanted to have a class extend another class **and** the Thread class, this would not be possible.

Implementing Runnable

Likewise you extend the Thread class you must override the run() method and then get the thread running by calling .start() on it.

Thread Safety & Atomicity

Issues can occur when utilising non-thread safe classes as they do not handle things well regarding atomicity.

Atomicity is ensuring that a block of code runs from beginning to end, uninterrupted and that the end result is the expected result. This can be particularly difficult/complex when there are various classes and instances running - interacting with shared resources/variables.

Parts of code which may be vulnerable are called “critical sections” and are to be handled with caution.

Comments/Thoughts

While I have had some brief interactions with threading in other units, they did not go into such details about threads, how they work and are managed. After experimenting with threads it becomes quite obvious that programs can get quite complex and as a result debugging can become tedious.

Week 5 (Locks - Usage)

Overview

This week was focused on locks, when to use them and why to use them, alongside various terms things you should be mindful of when dealing with multiple threads.

Critical Section

Is a section of code which interacts with some resource/variable which may be shared across multiple instances of a class. This can include statically defined variables which are shared across all instances and also class data which can be used by multiple threads of a single instance.

Race Condition

This occurs when multiple threads are trying to access a critical section simultaneously, this can lead to undefined behaviour.

An example of this occurring might be if two threads are trying to increment a variable. The steps to increment a variable is to first read in the value, increment it and then replace/update the value in the variable.

In our example if x = 2 and we have two threads trying to update it, we would expect x = 4 at the end.

Since the threads are running at their own pace, it is possible for both of them to reach the critical section of code which increments x.

Both of them then read in x as 2, increment the value they have taken so they both have 3 stored and then they update x to equal 3 - leading to an undesired outcome as we should have expected x = 4.

Indeterminate

A program/system is considered indeterminate when there are multiple race conditions which lead to unexpected results after every execution/run.

Mutual Exclusion

The implementation of mutual exclusion is a way to prevent race conditions/indeterminates by only allowing a single thread to access any shared variable/resource.

This can often be done by utilising some form of a lock on the resource in question - restricting only a single thread to interact it at any given moment.

Atomicity

Atomicity is ensuring that a block of code runs from beginning to end, uninterrupted and that the end result is the expected result.

Java Locks

Locks in Java utilise the lock() and unlock() methods to acquire and release a lock. There are various ways to lock critical sections and ensure mutual exclusion.

ReentrantLock

The ReentrantLock class interfaces the Java Lock interface.

You would call the .lock() method on it before entering a critical section and call .unlock() once finished.

ReentrantLock usage should be wrapped in a try, finally block where the finally block is where .unlock() is called so that regardless of success or failure the lock is given up.

ReentrantLock also has various other options/features such as a tryLock() where you can provide a ‘timeout’ value where the code will wait for that duration to try and perform the lock.

Synchronized Keyword

The synchronized keyword can be used with a generic object or a specific object/resource where using the synchronized keyword will ensure that only one thread is handling the code.

If two threads are attempting to execute a function which is defined as synchronized, it will only allow one thread to execute while the others are suspended.

When synchronized is used, there is a happens-before relationship which ensures that changes to the object managed by the lock is seen in all the threads.

It should also be noted that constructors cannot be marked as synchronized and handles the lock, try, finally and unlock sections mentioned in the ReentrantLock. Making Synchronized a little easier to use, but at the cost of not having as many features/controls over the locks.

Week 6 (Locks - Implementation)

Static Variables

Due to static variables being linked together across all instances of a class, they are not able to be guaranteed mutual exclusion by simply having their calls wrapped in the earlier methods.

The solution is to utilise the Synchronized keyword or a Static ReentrantLock.

Building a Lock

A well designed and implemented lock is not an easy task to perform as there are many considerations necessary which do not come to mind initially.

They need to consider the hardware and operating system at hand (meaning that it might not be cross-platform compatible).

Also need to ensure mutual exclusion while also balancing fairness and performance.

Controlling Interrupts

This is an implementation of a lock system which is of **very little benefit** in most cases where the premise is to **disable interrupts** on the system such that context switching is **prevented** and only a single thread can **access** the critical section.

There are various downfalls and issues with this technique and as a result it is not often seen in usage.

Loads/Stores

Another basic locking technique which does not work well. The premise is that a flag (essentially just a variable) is checked and updated when entering and exiting a critical section.

The issue here, similar to that of multiple threads incrementing a variable is that multiple threads may read the flag as being available before the first one sets it as “locked”. Resulting in multiple threads operating in the critical zone at the same time.

Test-and-Set

The test-and-set technique is setup in such a way that it atomically updates the value of the lock and returns the old/current value - not allowing for context switching to occur during this operation.

As a result a while loop is required to constantly check this atomic operation so that the locks which were waiting can finally execute the code.

This unfortunately means that this technique will only work on preemptive processors because if it were non-preemptive it would just constantly be in the spin/wait cycle.

Compare-and-Swap

Similar to that of the Test-and-Set however it utilises another check to reduce the need for assignments when the current value is equal to the expected value.

Spin Ticket with Fetch-And-Add

Uses “ticket” and “turn” variables to build a spin lock system.

The premise is that the current owner of the thread’s lock is assigned a ticket, while the other threads are awaiting their assignment of a ticket. Where each progressive ticket has its number incremented so they are unique.

Each thread will then wait until the lock’s “turn” variable is equal to their ticket number and then once the thread is done executing the critical section it will increment the lock’s turn number so it will allow the next thread to execute.

This ensures fairness such that all threads will be able to execute and progress, however the performance is still poor on single processor systems due to the need to spin when waiting.

Yield

Yielding solves the issue of wasted performance due to threads spin waiting, by having the thread yield itself so that it deschedules itself to allow other threads to run and progress.

Simply invoking the appropriate thread yield call instead of spin waiting will make this work.

Queues

With all previous approaches there was no real way of controlling/determining which thread will take control of the lock after it has been released.

Utilising a queue along with support from the operating system we will be able to track threads and their IDs to be able to sleep and and wake up threads.

Calls such as park() and unpark(id) allow threads to park themselves and then have the current thread call unpark on the next thread in the queue - further improving fairness.