Multithreading OS

Final Report

Submitted for the BSc (or MEng) in
Computer Science

April 2016

by

Matthew Newall

Word Count: 11,088

Table of Contents

1 Introduction

The aim with this project is to develop a scheduler and process system for a C++ based primitive operating system, or PrimOS for short, that will allow it to schedule processes and their threads in the most effective way possible. This will be achieved by researching and prototyping various different algorithms, and deciding on the best algorithm to use through testing and understanding. The focus of this project will be entirely on the scheduling, multitasking, and process system, and finding the best way to implement them into PrimOS. All other work done to PrimOS is purely to support the multitasking system, and is not the focus of this project.

This report will outline the intended outcome of this aim, and to establish the objectives to achieve this aim, as well as tasks and an estimate time plan for the completion of this project. It will also give a good understanding of operating system multitasking, followed by design details, and finally a review of the project so far.

2 Aim and Objectives

The aim of this project is to research, prototype, and test multiple scheduling algorithms, and to implement the best performing algorithm onto PrimOS.

To achieve this aim, multiple different scheduler algorithms and process/thread structures will be experimented with, in order to find the best system for the task. These shall be tested by creating test processes of varying types. These processes will require a wide range of different resources and requirements, in order to cover as many situations as possible. This will allow for each scheduler to be put to the test in different ways, and will be used to decide the best algorithm to implement. The priorities for this project will be on the stability and performance of the algorithm as these are the most important parts of a scheduler. Stability is how successful the scheduler is, as well as how hard it is to break. This will be tested by running multiple different processes through it and seeing how often the scheduler breaks. It will also be tested by creating extreme situations to see how it performs under unrealistic but plausible pressure. The performance of the scheduler is how fast and efficiently it can decide on the next process to run, and swap to it, as well as how fast the results of the scheduling are. This shall be tested by calculating the average speed of the switch, and by timing how long all the given processes take to finish.

By the end of this project, multiple different systems and algorithms will be made into prototypes and tested to find the most appropriate. The fastest and most stable system found will be implemented. It will support at least 256 processes, will have full threading support, and be designed in such a way that adding features or improving upon it is simple and easy to do, however this will not be at the cost of stability or performance. It is designed to run on the x86-64 architecture, and must be as general purpose as possible.

To reach this aim, There are multiple objectives to complete.

2.1 Objective 1: Establish a Base Architecture

PrimOS and the initial base system for the multitasking are to be set up ready for the scheduler to be applied. For the base multitasking system, it will do a simple round robin schedule to confirm it is set up correctly. PrimOS will need memory management, paging, serial output for debugging, an on-screen command line for control, inputs such as the keyboard, and interrupts for making use of the PIT chip. These are not part of the project aim, but are required for the aim to be reached. The majority of these requirements have already been met, however the rest still need to be implemented, and the entire of PrimOS needs to be tested once it reaches this point to make sure it is ready. This objective needs to be completed first in order for the next two objectives to be started.

2.2 Objective 2: Implement Threading

Processes must have a threading system implemented. The minimum functionality will be to create and run threads, destroy threads, to pass arguments to the threads, and to receive a return value when a thread ends. With this functionality implemented, threads can be used to perform most if not all jobs they would be needed for. This must be done before objective three can be started as threading may have a huge impact on the best system to implement.

2.3 Objective 3: Implement Multiple Schedulers

In order to find the best scheduler to use, multiple different algorithms must be researched and implemented. This will allow for experimental algorithms to be considered, as they won't have well known advantages and disadvantages, and these can be found out by testing. Both established and experimental algorithms will be considered for this. The algorithms implemented need to be varied and unique in order to get a good spread of methods. Only a small amount of algorithms will be tested, as variations can be added once an algorithm has been chosen.

2.4 Objective 4: Evaluate the Schedulers

Once all desired algorithms have been implemented, testing must be done to decide on the best algorithm. This will consist of 3-6 tests that cover a wide range of situations that will reveal the best algorithm for PrimOS. Some heuristic thinking may be applied here, as there may be other reasons to choose an algorithm over one performing better, such as being more stable or having better potential for improvement and growth. The operating system will be used as a general purpose OS, and will have no specialised use, so the best algorithm will most likely be versatile and efficient in lots of situations. The algorithms that are not chosen will be removed from the PrimOS source code, but will be kept for future reference.

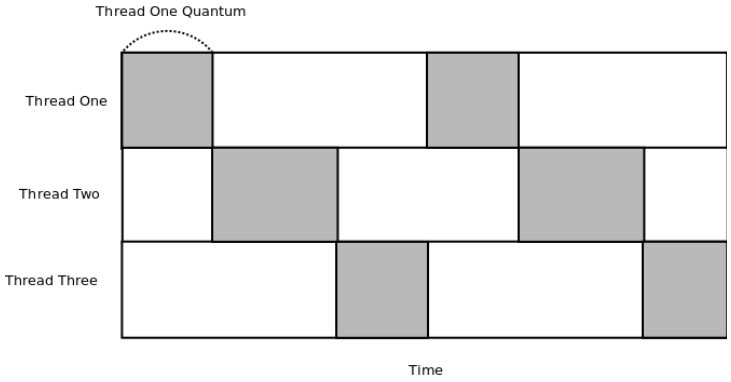
2.5 Objective 5: Research and Implement Additional Scheduler Features

Once the scheduler has been implemented, additional features can then be researched and implemented to further improve the efficiency and stability of the scheduler. Such extra features may include multicore processor support and waking processes when their IO is finished. These will be done last, as they require the previous objectives to be finished, and they are less important than the others. What extra features are implemented will be dependent on remaining time left and what the research results in. This objective is not required for the project to be complete, it is simply a way to smooth and improve on previous implemented features.

3 Background

3.1 Multitasking and Processes

Multitasking is a system which allows an operating system to run multiple process' concurrently. In this system, a process is given its own memory, code, and resource access, that is separate to other processes, as well as access to the CPU. (Tanenbaum, 2014) To achieve this, each process is given a time-slice, or quantum, to run for, and takes it in turn with other processes. These quanta are usually big enough for substantial work to get done during the process' turn, while small enough to give the impression to the user that all the processes are running at the same time.



In order for an operating system to give each process the time it needs on the CPU, it needs to be able to schedule effectively. Scheduling is the system in which the current running process is paused, and the operating system calculates which process is given control of the CPU next. There are different kinds of algorithms to achieve this, both complicated and simple. Each algorithm has its advantages and disadvantages, that need to be addressed when designing and implementing a scheduler.

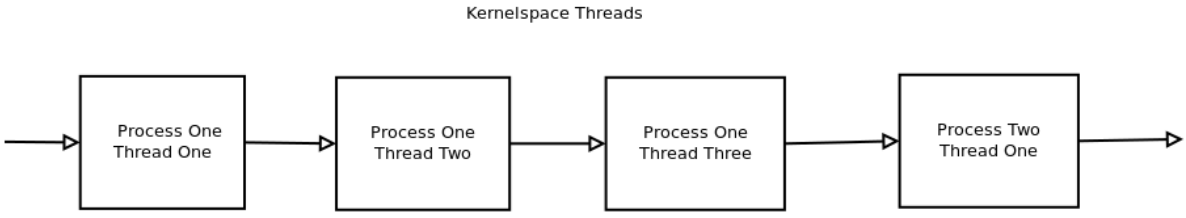
3.1.1 Threading

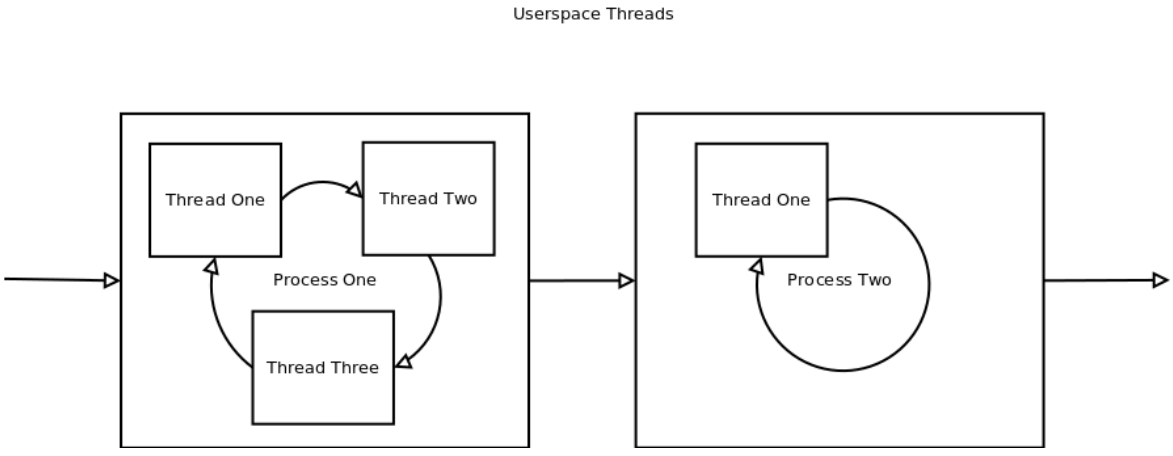
A thread is a separate execution of code that is done on behalf of a process. They are used to perform tasks in isolation to one another, but while sharing resources and memory space. Each process can create and destroy threads as needed for the situation. Each process contains at least one thread, generally accepted as the main thread. (Tanenbaum, 2014)

Using threads gives a process the ability to handle multiple things separately. For example, a process may use a thread to process data, or load files into memory, while a different thread maintains the UI to keep it responsive. Threads share the same paging system as other threads in a process, as well as external resources. (Silberschatz, Galvin, and Gagne, 2012) This means that communicating between threads in a process is a trivial issue. Other advantages of having threads include being easily created and destroyed, meaning they can be used on the fly without concern for performance, and they allow for processing to happen even when waiting on resources, such as if the process is waiting to read a file from storage, a thread can continue processing data even while the process is waiting on another thread. A thread maintains its own registers and stack separate to other threads, as threads could not function without having these alone.

There are two methods of setting up threading: At the user-space level, and the kernel level. (Silberschatz, Galvin, and Gagne, 2012) If set up at the user-space level, a process would only have one execution of code, and a multithreaded system would be placed on that execution, allowing for the process to have threads. The advantages for this method are that it allows the process to use its own scheduling algorithm, which may be vitally important for specific programs, as well as making swapping between threads much faster, as swapping at a kernel level would either require the page to be swapped needlessly, or for the scheduler to check if both switching threads are in the same address space, which is excessive overhead for the switch. One downside to user-space level threading is there is no mechanism for pre-emptive swapping, as the timer used for timing threads is already being used for process timings. (Silberschatz, Galvin, and Gagne, 2012)

For kernel level threading, threading is an integral part of the multitasking system, as each thread is treated as its own entity. This means that rather than swapping to a process, the scheduler will swap to a new thread. This means that the swift switching between a process' threads due to not having to swap address spaces (Which is hugely costly to performance to do) isn't possible to do, as well as not being able to schedule the threads as needed, however kernel threads do not rely on cooperative swapping like user-space threading does. Another advantage to kernel threading is they work with blocking. If a user-space thread required a function that would cause it to block, it would mean the entire process losing its CPU time, as the thread would have to cooperatively swap to another thread, and it is incapable of doing that while blocked. (Silberschatz, Galvin, and Gagne, 2012) Kernel threads avoid this by being ran separately to the rest of the process. For PrimOS, kernel level threading was chosen over user-space threading, as it allows for more control and freedom in its use, as you can block and use pre-emptive scheduling, and the performance hit from page swapping can be alleviated though prioritising threads in the same address space being ran consecutively, and simply not swapping paging tables out. It is also easier to maintain overall.



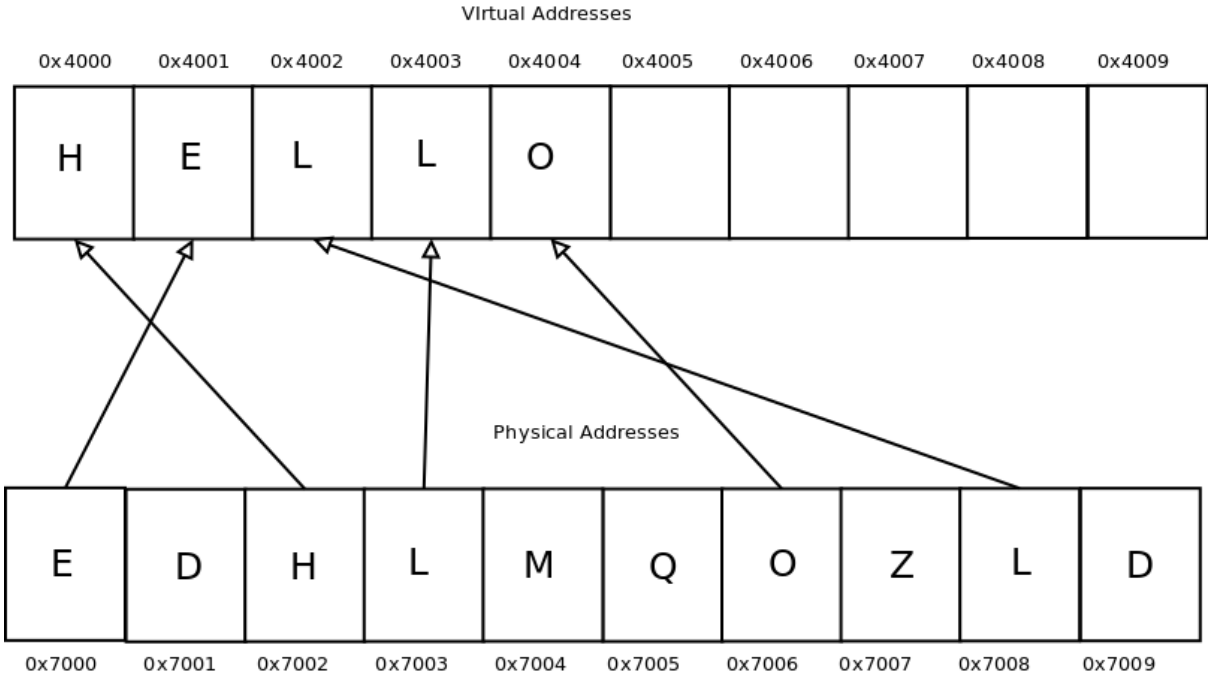


3.2 Stack

The stack is a block of memory used for storing temporary data, such as function return addresses, and local variables. (Silberschatz, Galvin, and Gagne, 2012) It is used by pushing values onto the top of the stack, or by popping the values off the stack. Values can only be moved onto or off the top of the stack, and any values lower down in the stack cannot be accessed with the push and pop instructions. Because of this limitation, values can be quickly moved on and off the stack quickly, as the instructions are very straightforward. There are two registers to handle the stack. The base register points towards the bottom of the current stack, and the stack pointer points towards the top, and is used when pushing and popping. When a call is made to a function in assembly, a new stack is setup starting at the top of the current stack. First, the current base register is pushed onto the stack. This is so after the call is done, the stack can return to its initial state. Then the base register is set to point to the stack pointer. This results in a new stack setup just above the previous stack. Once this call is done, the stack register is set to the base register, and then the base register is popped off the stack.

3.3 Virtual Memory

In order for a process to maintain its own private memory space, a system called virtual memory was created. Virtual memory is a way of separating an address from the physical memory. (Tanenbaum, 2014) In a virtual memory setup, a virtual memory address is assigned to a physical address. This means that the memory at the physical address is treated as being at the virtual address.



These are allocated in 4kB blocks usually, though larger blocks can be allocated if needed. (*Intel architecture software developer's manual*, 1997) Virtual addresses are tracked using a system of paging tables, where the root table, called the PML4, is placed in control register 3, which dictates which paging system is currently in use. These translations between virtual addresses and physical addresses are cached in the Memory Management Unit. This unit's cache greatly improves the speed at which paging works, however if the CR3, the indicator

of what address space is currently active, is altered, even if the value doesn't change, the MMU will be wiped and start from scratch. (*Intel architecture software developer's manual*, 1997) This causes a drop in efficiency until the MMU has cached translations again. Because of this, it is ideal to avoid having to swap paging tables as much as possible. Each process is given its own separate paging system, which allows for processes to use the same virtual addresses without conflicting with one another, essentially giving each process its own address space.

3.3.1 Higher Half Memory

In the paging system designed for x86-64, only 48 bits of an address can be used, with the highest 17 bits being set to the same value, similar to 2's complement. This results in a segmentation of memory, with a lower half covering from 0x0 to 0x00007FFFFFFFFFFF and the higher half covering 0xFFFF800000000000 to 0xFFFFFFFFFFFFFFFF. (*Intel architecture software developer's manual*, 1997) These are known as canonical addresses, with the empty void in between being non-canonical addresses. In most implementations, the higher half addresses are reserved for kernel software, such as the kernel or physical memory mappings.

3.4 Context Switching

In order to swap between threads safely, the exact state of a thread, such as the registers, must be saved when its execution ends, and the new thread's paging system and registers must be loaded in. This is called a Context Switch. As this is done often, making it quick and simple is a priority for good design. (*Intel architecture software developer's manual*, 1997)

The first step to context switching is to save the current state of the thread. This is achieved by pushing all the current thread's registers to the stack. This is because the stack is a quick and easy to access memory store that will not be lost when context switching. Once this has happened, the paging table address, as well as the stack pointer are saved to the thread data structure. The next thread's paging table and stack pointer are then loaded in, and all that thread's register values are loaded from the stack. (*Intel architecture software developer's manual*, 1997) Through this method, a thread will maintain its exact state through the saving and loading, and would not be affected or even know that it had been switched out.

In order to keep processes from preventing other threads from having time in execution, there are two methods of triggering a context switch: Pre-emptive and cooperative. Cooperative swapping is where the currently running thread is blocked in some way, such as waiting on a file to be loaded or even simply waiting for a period of time, and thus relinquishes its control to another thread. This is in place so if a thread no longer wants or needs processing time, it can be handed to a thread that does. If a process has no reason to do this, such as if they are simply processing a large amount of data, they will exceed their quantum of time, and pre-emptive scheduling will occur. This means that no matter what happens, a process cannot hog the CPU for too long, as there are tasks that require a lot of processing, and waiting on these to relinquish the CPU will affect the performance of all other threads, as well as make the system feel unresponsive. Pre-emptive timings are handled by the timer interrupt.

3.5 Interrupts

Interrupts are blocks of code set by the kernel that are triggered and executed when an event, such as a key press on the keyboard, happens. (*Intel architecture software developer's manual*, 1997) When an interrupt occurs, the current instruction pointer is pushed onto the stack, and the interrupt code is ran. When the interrupt code has finished, the interrupt pointer from before is popped off the stack, and execution continues as it was before the interrupt occurred. Without interrupts, it would be incredibly difficult to have a operating system that could react to things that happen, as there would be no efficient or sensible way of checking during execution of code. They are numbed in order to tell which interrupt is being triggered. There are three types of interrupt: Exception, IRQ, and software.

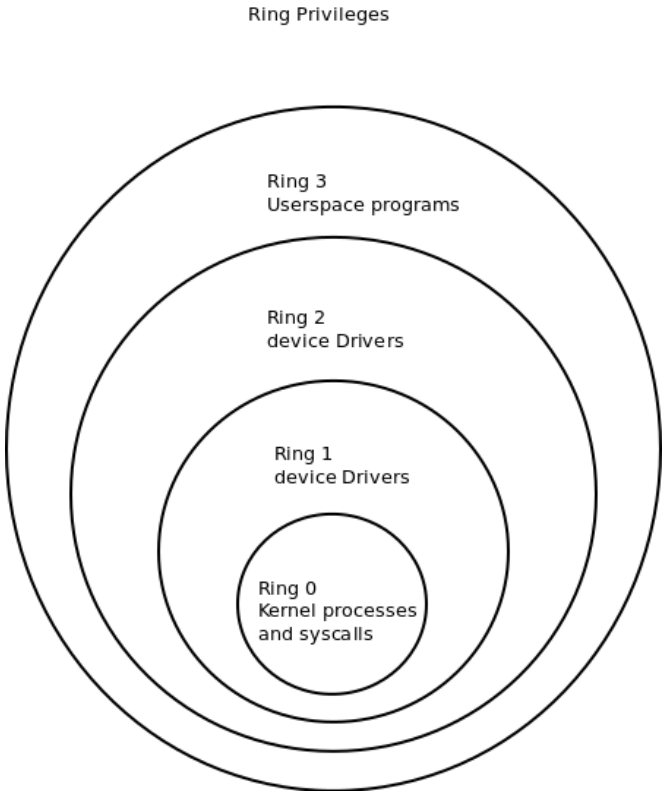
Exception interrupts are triggered when something occurs to the system. (*Intel architecture software developer's manual*, 1997) For example, a page fault exception is triggered when an attempt to read or write with an unmapped address is made. These exist to provide a mechanism to deal with problems that might come up at runtime.

The IRQ interrupts are interrupts that are generated by hardware. These interrupts are handed to the CPU by the Programmable Interrupt Controller. (*Intel architecture software developer's manual*, 1997) The timer interrupt is an IRQ interrupt that is triggered by a timer chip on the motherboard that counts down based on a value given by the operating system. This gives an easy method for setting the quantum of a running process and keeping track of time.

Lastly, the software interrupt is an interrupt that can be triggered by using the INT instruction. (*Intel architecture software developer's manual*, 1997) These are generally used for system calls. A system call is an interrupt designed to fulfil specific jobs that a program might need, such as requesting more memory or starting a thread.

3.6 CPU Rings/Privilege Level

There are a number of instructions or memory addresses that a program should not be allowed to touch, such as the clear interrupts instruction, or the kernel memory. This is because a malicious or badly written program could use these to potentially break or manipulate the system. In order to prevent this, the CPU Rings or Privilege Level system was implemented. (Tanenbaum, 2014)



This is a low level, CPU mechanism that prevents any code running under low privilege to access anything above its privilege level. There are 4 ring levels in this system. (Tanenbaum, 2014) Ring 0 is known as kernel mode or supervisor mode. This level has full access to everything in the system, and as such is reserved for only system programs and system calls. Rings 1 and 2 are slightly limited, but still retain significant privilege. Ring 3 is known as the userspace ring, as it is the least privileged ring, making it the ideal and safest level for programs to be ran on. If code attempts to access or run something that requires a higher privilege level, a general protection exception is generated. In order for a program to access higher privilege resources or instructions, it has to use a system call interrupt. (Tanenbaum, 2014) When an interrupt is defined, is is assigned a ring level that the interrupt's code is ran at. This allows a program to access higher privileged resources, but in a controlled, pre-defined way to severely hinder abuse. For instance, an interrupt may be setup to allocate more virtual memory. This would be a problem if a thread could allocate whatever memory it wanted, as it could allocate dangerous memory and edit it. However, by writing a controlled function for allocating memory, that does not allow this, and assigning it as an interrupt, you can control the thread's ability to allocate to what that function allows.

3.7 Memory Allocation

In order for a process and thread to function correctly, it must be able to allocate memory spaces to store data. This data must be organised in order to avoid corrupting data and requesting more memory than needed. To achieve this, a memory allocation system is implemented.

One problem that algorithms aim to address is memory fragmentation. This is where there are sections of memory in-between allocated memory that are unable to be used due to being too small. (Kasper, 2016) Over time, this fragmented memory adds up to a significant amount of memory wasted. Reducing the amount of fragmented memory will drastically increase the performance of a system at high memory usage. There are two kinds of

fragmentation: Internal and External. (Kasper, 2016) Internal fragmentation is where too much memory is allocated for a set of data, and there is excess memory that the data doesn't need. External fragmentation is the sections of memory between allocations. Another issue of memory allocation is speed. As allocating and deallocating memory is done often by most if not all programs, the speed of the algorithm will affect the overall speed and efficiency of the system. Overhead memory usage is also a significant issue, however as memory capacity increases, this issue becomes less of a priority over the previous two.

There are multiple methods for handling memory allocation. One method of allocating memory is to use a bitmap. (*Memory management from the ground up 2: Bitmap allocator*, 2010) Every bit in the bitmap represents a unit of memory that is allocated or free. Another is to make memory into a linked list. (Golick, 2013) This means having each block of allocated or free memory covered with a header. This makes searching for free space much faster, but is more liable to large overhead, especially with small allocations. (Golick, 2013)

A system that allows for small and precise allocation block sizes is more at risk of external fragmentation over time, which can lead to memory usage creep. Larger and less precise allocations take up more space and have much more internal fragmentation, but have insignificant external fragmentation.

3.8 Multitasking Problems

There are multiple problems that come up when multitasking is introduced. The most prominent of these are race conditions. A race condition is a situation where a system can break due to two or more threads trying to access a resource at the same time. For instance if two threads wished to append something to the end of a list, the first one may start by finding the address that needs to be written to. Pre-emptive scheduling may coincidentally force the running thread to stop there in between finding the first empty address in the list and writing to the address, and load in the second thread that finds the same address, and then goes on to write to that address. Upon returning to the first thread, it will write to that same address, thinking it's still the first empty address, overwriting the second thread's data. While there is a very small chance this could happen, this needs to be fixed regardless as it can still plausibly happen. In order to remove the chance of a race condition, a separate boolean variable can be used to state that the memory in question is in use. (Robbins, President, and Technologies, 2000) This system is called a mutex.

3.8.1 Mutexes

A mutex is a system designed to allow only one thread at a time to access a resource. It does this by marking whether a resource is in use at a given time, using the previously mentioned method of a bool. (Robbins, President, and Technologies, 2000) The problem with this system alone is that changing the bool to signify it is in use can also be a race condition. In order to set the resource to being in use safely, checking and setting the in-use boolean needs to be done without the possibility of a context switch happening in-between. One method of doing this could be to disable interrupts, thus stopping pre-emptive switching for the duration of the checking and setting. While this method works fine, it needs to be ran as kernel call, due to disabling and enabling interrupts being off limits to users. A much faster and sturdy method is to use an atomic instruction. (*OSDev.org • view topic - Mutex implementation*, 2007) An atomic instruction is an instruction that does multiple things in one go, and cannot be interrupted midway. The compare-and-swap operation can be used for this. (*OSDev.org • view topic - Mutex implementation*, 2007) It means that the thread can

check the in-use variable, and set it if it's false, without anything possibly happening in between.

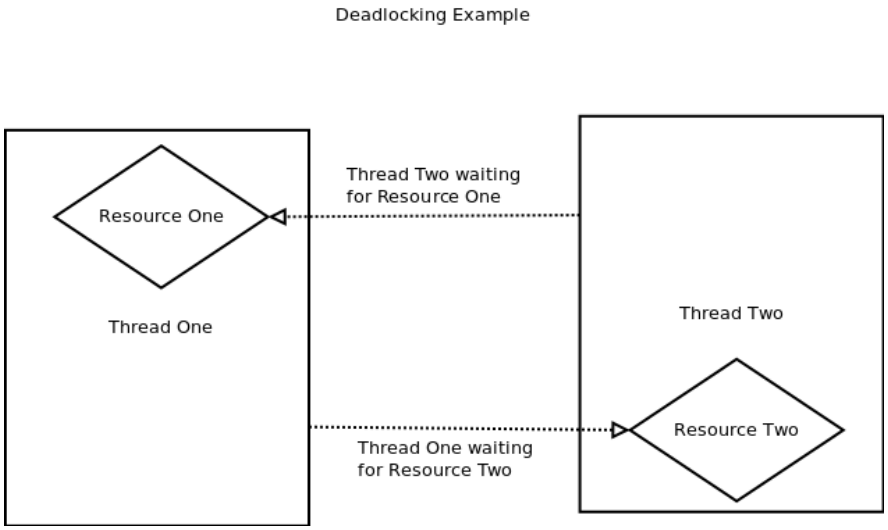
If the check is performed, and in-use is true, the thread will block, and wait until the resource is free, before setting in-use and continuing with the work it needs to do. This generally forms a queue for the resource. If in-use is false, it will set in-use to true and use the resource.

3.8.2 Deadlocking

Another significant problem within multitasking is deadlocking. This is a situation where two or more threads require a resource that one of the others have, effectively stopping the whole group from progressing. For instance, if Thread 1 has resource A and needs resource B, and Thread 2 has resource B and needs resource A, they will be deadlocked, as they both require the resource that the other thread is using. In order for a deadlock to happen, four conditions must be met: Mutual Exclusion, Hold and Wait, No preemption, and Cirtual Wait. (Silberschatz, Galvin, and Gagne, 2012) Mutual Exclusion means that the resources being contested must be unsharable. This means that it is not possible for the deadlock to be fixed by simply sharing one of the resources. Hold and Wait means that all threads involved in the deadlock must be holding one resource, and waiting on another resource. No Preemption means that the resources held by a thread cannot be taken away forcefully. Circular Wait means that every thread involved must be waiting on another thread's resource. Once these four conditions are met, a deadlock is formed. (Silberschatz, Galvin, and Gagne, 2012)

This can be fixed by having resource requests time out after an arbitrarily random period if a thread blocks too long. This stops a deadlock from persisting, and the randomness of the period reduces the chance of the threads deadlocking again straight away to a minimum. (*Deadlock Solutions*, 2012)

Another common method is to leave handling the deadlocks to the application developers to deal with. This allows a deadlock to be handled in a way more fitting to the applications involved, for instance if a certain resource is wanted but not really required, it could drop waiting for the resource entirely. (*Deadlock Solutions*, 2012)



4 Technical Development

4.1 Initial State

At the beginning of this project, PrimOS had a very basic and simple architecture. It has been tested and ran on a virtual machine called BOCHS, which emulates a fully functional computer, as this is easier to work and test with than a second computer. When the operating system is started, the GRUB bootloader is started up by the machine, which loads PrimOS into memory, and transfers control to the OS. PrimOS then puts the computer into 64 bit long mode, and begins to set itself up. This setup involves creating the physical memory manager using the memory information provided by GRUB, creating the Interrupt Descriptor Table and filling it with the required exceptions, setting up the serial port for debugging, initialising the system virtual memory paging, and finally setting up the process multitasking and passing control to the first process. This setup is very straight forward, and most of it has been left unchanged as it isn't within the scope of this project of writing a scheduler.

When a paging table is first setup or changed, accessing the tables required them to be mapped to a virtual address. As stated previously, switching virtual memory spaces has a significant impact on the performance of the system. In order to improve on this, a new system that didn't require the memory space to change was required.

Processes at this point consisted of a single thread with its own paging tables. While functional, kernel level threading is to be implemented. It was also running these threads with kernel level priority, which is a massive security flaw. In order to improve this, two steps were needed to be taken: Adding ring privileges, and implementing threads. These will be implemented early as they form the base of the multithreading system. Doing so will make jobs easier for the process to handle, as it can separate the processing to different threads.

The system also had a timer setup. This timer was used to keep track of how long a process was running and pre-emptively pause it when it ran over its allotted quantum. As this was a relatively simple system, no changes needed to be made to it.

In order to receive outputs from the operating system, a feature in BOCHS that writes serial outputs to a text file was utilised. This feature will be used to communicate test results back from each scheduler.

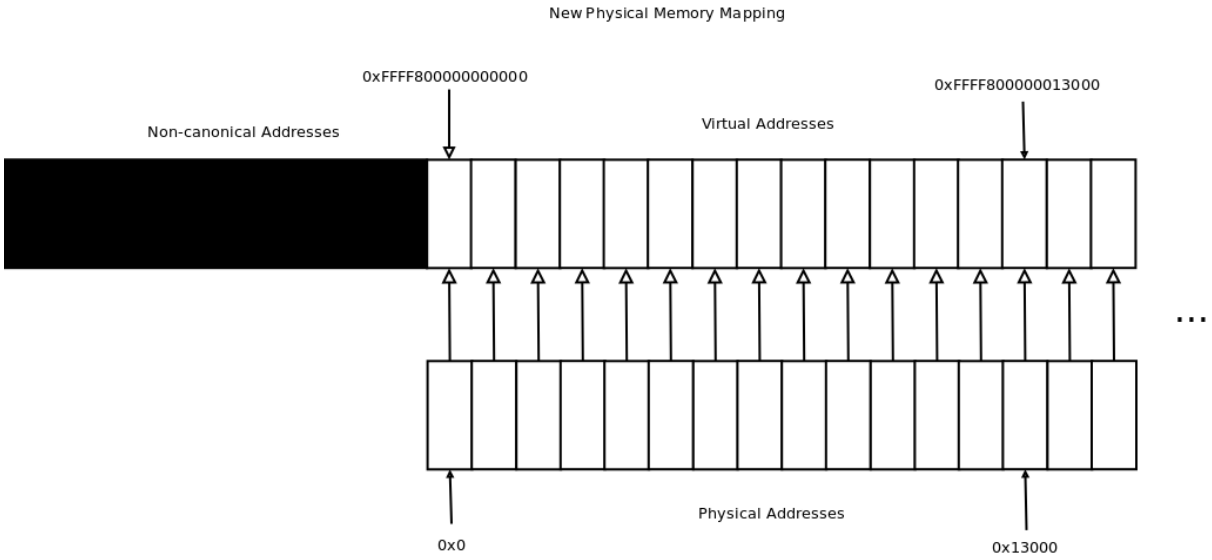
The exception interrupts are set to lock the system up if one is triggered. This is because PrimOS currently has no real variable parts, and is running very deterministically. This means that if an exception does occur, there is a problem with the system, thus this is the best course of action for the purposes of this project. This is helpful with testing, as information regarding the problem can be quickly passed through the serial port, and the system will show itself locking up in the BOCHS debug console.

4.2 Establish the Base

In order to begin work on the scheduling, the basic tools and systems needed to be implemented. Over the course of the project, it was found that the required parts were different than first anticipated. The ability to take keyboard input, as well as a rudimentary command line on-screen was not needed, as all testing is automated, and any user input needed in the tests is better simulated in order to provide consistency and fairness to the tests. Implementing a graphics system was found to be highly useful to provide an extra layer of complexity to the testing, as writing to the graphics buffer is considerably slow. This base was required as working without these tools would make testing and usage of the operating system hard.

4.2.1 Virtual Memory Allocation

Firstly the memory management and paging system was improved. When a process wanted to allocate more virtual memory, it needed to write to its paging tables, however paging tables were not mapped. This was addressed by creating a paging system that mapped all physical addresses to their virtual equivalent. This is known as an identity mapping. To use this required swapping the current paging table for the identity map table, changing the paging tables of the process as needed, and then swapping back. Is also required a new stack to be used for the duration of the swap. This system was buggy due to GCC not being prepared for the stack and virtual memory being changed. It was also considerably slow, as swapping to a new paging table resets the MMU cache, causing all memory accesses to go through the paging tables again. In order to improve this, a new system was implemented. During the operating system's setup, a map of the whole physical memory was created at the start of the higher half memory. This mapping is then added to every paging table created. This results in a less buggy, and faster way to map addresses, as the current paging table doesn't need to change temporarily. As it is stored in the process' virtual memory, it will require protection in order to stop it being utilised wrong.



4.2.2 Ring Privileges

One issue with the system as it is, which is made worse with the previous change, is that processes have full control of the system. For instance, any process can run the cli instruction, which turns off interrupts, including the timer interrupt. This means a process could accidentally or purposely stop preemptive scheduling, effectively taking full control of the system. The physical memory mapping created in the previous change is also fully available to processes, meaning programs can change other program's data. This massive security flaw is fixed by implementing privilege levels. All threads will be ran at ring 3, the lowest priority level. This means that when the system is running at a privilege level below 0, no dangerous instructions can be ran, and will trigger a General Protection exception when attempted. Virtual addresses can also be assigned minimum privilege levels, in order to stop low privileged accessing memory it shouldn't be, such as the aforementioned physical memory map.

If a process needs to access higher privilege, it can be achieved by using interrupts. Since interrupts are set up by the kernel itself, and a process cannot change the code, it enables processes to use high priority functionality, without the control to cause problems with malicious or badly written software. For this, two interrupts have been set up. One interrupt has been set up to yield the cpu for cooperative swapping. This interrupt is encapsulated in a function called YieldCPU(). The other interrupt is a syscall handler, allowing for multiple different functions, such as creating a process or mapping virtual memory, based on a flag contained in %RAX. The cooperative swap is confined to a separate interrupt in order to make it as efficient as possible, as the syscall handler will be slightly slower due to required arguments and performing a switch.

4.2.3 Graphics

A graphics system was setup to allow processes to draw to the screen. This was set up using two buffers. Each process can setup a buffer to contain the frame currently being drawn. Once it has drawn the frame completely, it can then use the syscall interrupt to copy the process buffer to the main buffer. The graphics driver then draws this buffer to the graphics memory. This system is done to prevent half drawn frames to be drawn to the screen, also known as screen tearing. Accessing the main buffer is gated behind a mutex to prevent the graphics driver drawing the buffer to the screen while a process is still copying to it.

4.2.4 Mutexes

As mentioned in the previous paragraph, a mutex system has been developed. This was achieved by using the atomic instruction builtin __sync_bool_compare_and_swap, which compares a given memory address with a value, and changes it, within one instruction. This is to ensure there are no race conditions with the mutex. Alongside a bool representing if the resource is in use, a second variable was added to show if the mutex is being edited. This is because two threads could attempt to change the mutex structure at the same time, and create a problem.

If a mutexed resource is in use, the current thread will block itself, and add itself to the queue to wait for its turn. When the thread using the mutexed resource releases it, it will automatically unblock the next thread in the queue. This results in a FIFO system for deciding the next thread to receive a resource. A similar system has been set up for threads waiting on other threads to end, however this system will unblock all threads rather than just

the first in line. If two threads attempt to edit the mutex itself at the same time, the first one will gain access to, while the second will simply yield the CPU and wait, as editing the mutex is a small job.

4.3 Implementing Threading

In order to implement a multitasking system, threads must be first created. Currently, the process structure controls the execution of code. This would be fine for implementing userspace threading, however for kernel threading this will have to be restructured to move code execution to the thread structure, while process would simply control the group of threads. Threads under a single process must have access to the same memory space, but they must have separate stacks, and they must be unable to access another thread's resources.

4.4 The Schedulers

Four scheduling algorithms were decided on and implemented in this project. Two of the scheduling algorithms, namely round robin and priority scheduling, are the standard, unmodified algorithms, while the multilevel queue and shortest time remaining have been modified to fit the role better.

4.4.1 Round Robin

Round Robin is a simple algorithm, that works by having each running thread point to another running thread. This creates a ring of threads, that pass control to the next thread In the ring once the current thread is blocked or reached its quantum. (*Operating system scheduling algorithms*, 2016) When a blocked thread begins to operate again, it is simply added into the ring at any point, making sure to maintain the ring. The most important factor of this algorithm is that it guarantees every thread to have processing time. However this method is incapable of prioritising different threads which will cause threads like UI threads to appear sluggish.

This algorithm consists of a simple linked list, that simply passes control of the CPU onto the next thread in the list. This is one of the simplest algorithm possible. One advantage to this algorithm is that it guarantees every thread to have a CPU timeslot. (*Scheduling (computing)*, 2016) However treating every thread equally could cause inefficiencies with threads that require more CPU time than others. This algorithm is relatively common, and is very useful with light operating systems that can operate all threads equally.

4.4.2 Shortest Time Remaining

This algorithm simply calculates how long each thread has left remaining and runs the thread with the shortest time remaining. (*Operating system scheduling algorithms*, 2016) This algorithms is very good at handling numerous, short threads in good time, however this ends up starving larger threads, which when used in a general purpose operating system, which has UI threads that go on indefinitely, will be problematic. (*Operating system scheduling algorithms*, 2016)

To rectify this, the algorithm has been modified. The algorithm has been merged with the round robin algorithm, whereby “quick” threads and “normal” threads are declared separately, and run in different schedules. These separate schedules take it in turns to run, which will allow for quick, short threads to be ran for half the time, and normal threads to be executed the other half. By doing this, we maintain the advantages of shortest time remaining for use on quick tasks, while being able to use the normal threads for the main tasks. If during the quick schedule's turn there are no quick threads to execute, the turn is relinquished to the normal schedule.

4.4.3 Priority Scheduling

Priority Scheduling assigns every thread a priority value, which is used to decide which thread is to be executed next. (*Operating system scheduling algorithms*, 2016) This is used to give threads the appropriate amount of CPU time. However this alone would cause low priority threads to starve if a thread above them never ends or blocks. (*Scheduling (computing)*, 2016) In order to avoid this, threads slowly gain priority over time, until they have the highest priority of all threads. Once they have executed until they block or are pre-emptively swapped out, their priority is reset to its original value. Thus, this algorithm is capable of guaranteeing all threads eventual execution, while maintaining a dynamic but effective order and priority. However, this requires that each thread is assigned a starting priority level, which may be hard to decide on. (*Operating system scheduling algorithms*, 2016)

4.4.4 Multilevel Queue

Multilevel Queue is a form of priority based scheduling, where instead of assigning each and every thread a priority value, is it split into varying groups. These groups have different priority levels, scheduling algorithms, and quantum sizes. (*Operating system scheduling algorithms*, 2016) Doing this allows for better control of different types of threads, while keeping the scheduling both powerful and simple. However, since a groups priority cannot just be raised like in Priority Scheduling, the problem of thread starvation returns. In order to alleviate this problem, this algorithm was implemented using enforced turns, similar to the Shortest Time Remaining. Each level is assigned a maximum run duration for the whole level, and once the level has reached this duration, threads from that level cannot be chosen to run until there are no other valid threads. The higher levels are checked first to see if there is both a thread available and duration left on the level, and runs a thread from that level if both are true. If not, it moves to the next level to check. If it fails to find any thread, all durations on levels are reset and a search is tried again. If no thread is found still, it will default to the idle thread.

For the implementation in PrimOS, three levels were created: UI, Normal, and Background. UI threads are used for UIs, and anything related to appearances and responsiveness. Background threads are used for anything low priority, that doesn't need to be ran or checked often, and normal threads cover anything that doesn't fit these categories. These are all scheduled using round robin, as the threads are already segregated into priority groups so ideally all the threads on each level require the same amount of CPU time.

4.5 System design

After the chosen scheduler was decided, the system design was finalised. Processes are stored in a 256 large table. This is a good amount of processes for an operating system of

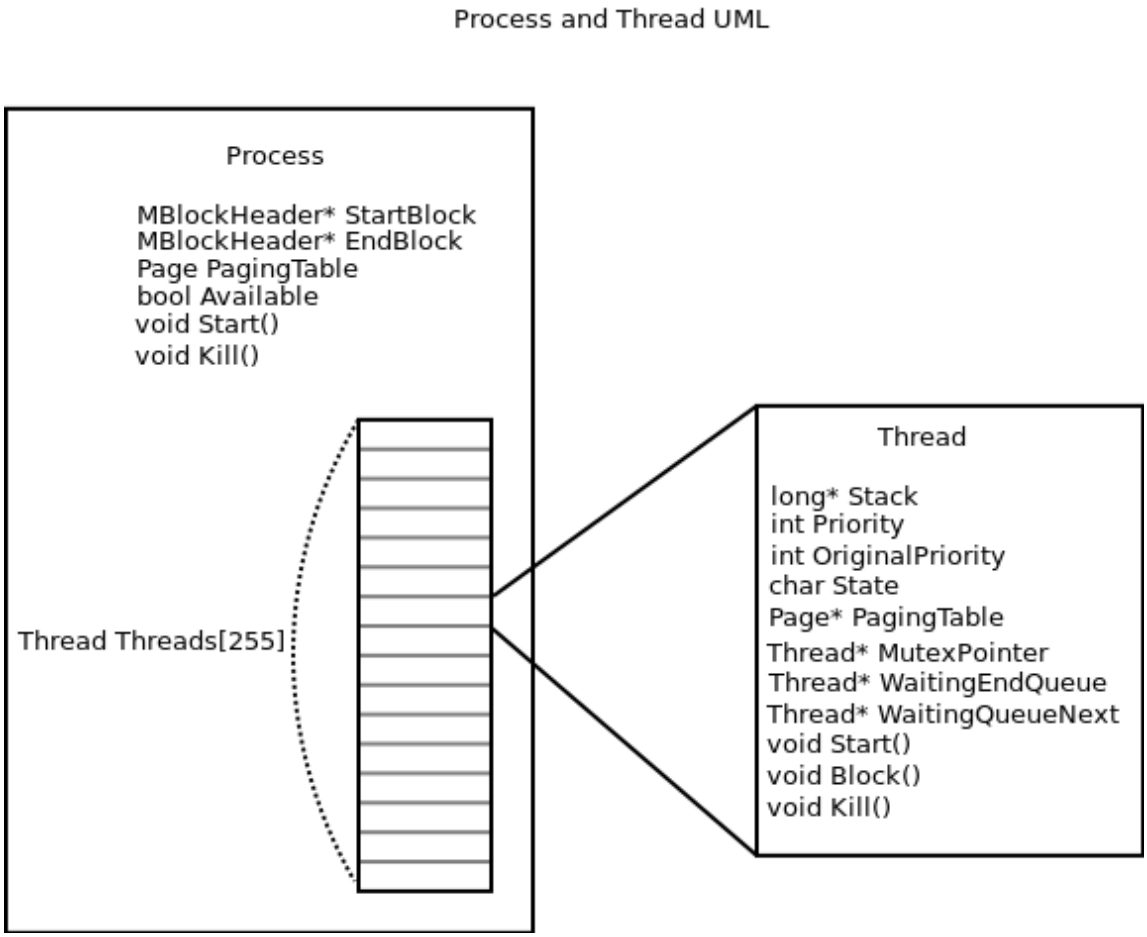
this scale, as it allows for a significant amount of programs to be ran, while keeping the size of the table small, as each process entry is of substantial size with potential growth. Inside each process object, the paging table address is stored, as well as a 256 large thread table. Having the process table, and the thread tables within these tables, as static tables rather than dynamic allocations was due to this being faster for looking up details of a process or thread. Both the total size of the process and thread tables can be changed at will at compilation, making the operating system scalable depending on memory size, however it cannot be changed while running, and too small a size may stop the operating system from being able to function correctly.

Each thread object contains a pointer for holding the stack pointer during the time the thread is swapped out, as well as pointers to other threads for use when waiting for a thread to finish, or with the Mutex system. This works by forming a linked list queue, whereby threads joining the queue simply add themselves to the linked list, with the first thread in the queue being pointed to by a pointer in the respective object. This makes it easy to quickly queue up threads, as it only requires adding a pointer to the last thread in the queue. The thread also contains its priority and original priority. The original priority is the base starting priority a thread has, when a thread's priority increases, the priority variable is the one to be increased, so a current priority and original priority are kept separate for when the current priority needs to be reset to the original.

Thread stacks are located at the top end of lower memory space, starting at 0xFF0000000, and each being 0x100000 in size. The stack was placed high up in order to allow for a process to maximise its memory usage as much as possible without limiting the stack size. None of this space is mapped initially, as this would be a waste of space, the memory is allocated as needed. As there is an extreme excess of virtual memory space, there is no loss to designating a large area for the stack, and having a large stack allows for large temporary objects to be created if need be. A portion of these stacks are set aside for use inside system calls, as they require a separate stack to function with.

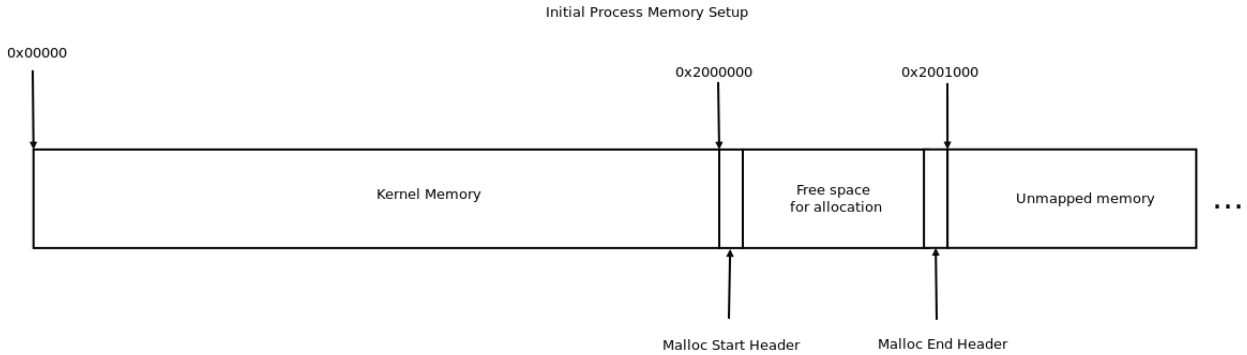
The memory allocation for the system involves a linked list setup, where the start of memory features a starting header that designates how large the first block is, and if it's free or not. At the other end of this block is a second header, containing both these same details of the previous block, and the details of the following block. This continues onwards until you reach the end of the memory space, where a final header designates the end. These headers can be traversed by simply adding the header size and the block size to the current position, to give you the position of the next header. This was chosen as speed of functionality is more important than reducing overhead, and overhead is only really a problem with small allocations, which can be accounted for by implementing further additions.

The development of this project followed an incremental development model. Everything was split into smaller groups and developed one at a time. This was the best method because each part of the scheduler can be neatly split off. For instance, the mutex system could be developed in almost total isolation to the scheduler, and then implemented afterwards. This allowed the system to be pieced together over time, while allowing for bugchecking to be done with little complications due to multiple components being involved. It also allowed any issues that were found with the components to be brought up before the entire system was under development.



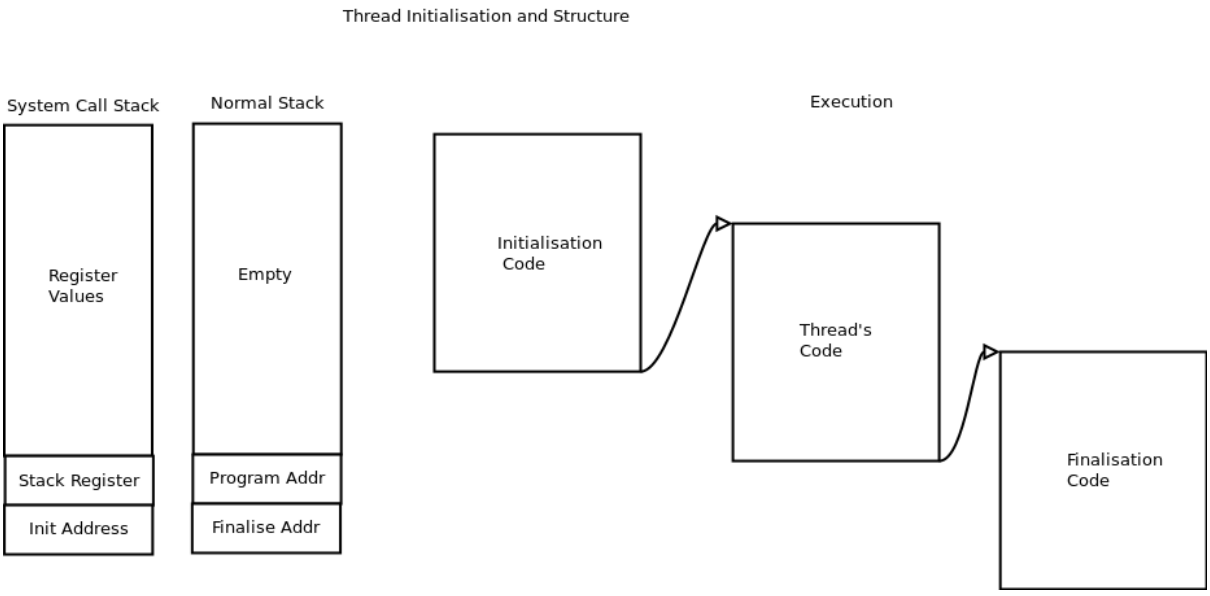
4.5.1 Setup

When a process is created, it is placed at the earliest free slot in the process table. It will start by setting up a paging system, which will contain the kernel space at < 0x2000000, and the physical memory map in the upper memory. It will then map a block of memory at 0x2000000 - 0x2001000, which serves as the start of the process' memory space. This block will be initialised with the malloc setup. All memory after this is left unmapped. This is because if the malloc function cannot find a block of memory big enough, it can request more memory to be mapped. After doing this, it will create a thread at index 0 in the thread table that will function as the process' main thread.



A thread is created by mapping its assigned stack addressing to physical memory, and then inputting the initial stack values on. Register are placed onto the system call stack, as

context switches use these stacks. These values allow the thread to be started through being swapped in as if it was already running. With exception to the FLAG register, all of the registers are set to 0x0. It also contains the address of the initialisation function, and the starting position of the regular stack pointer. The normal stack is empty except for the addresses of the thread's code and the finalisation function. Threads are set up to start at an initialisation function, before "returning" to the main code, which is turn will "return" to finalising function, which cleans up the thread, sets it as an available slot, and yields the cpu. This is done to allow any future initialisation and finalisation jobs have a space to be executed, such as the things mentioned before. If the 0 index thread returns or crashes, it is treated as if the process has ended and all other threads in that process are terminated. Currently, no memory space is needed for the thread code, as all threads are currently compiled into the kernel for the purposes of this project.



The mutex system works by using __sync_bool_compare_and_swap to guarantee only one thread can edit a resource at a time. The mutex is a class that can be inherited by any other class, in order to gain the mutex system. In order to lock the mutex, first the Editing variable is set in the mutex to confirm that no other thread is currently editing the mutex. This is different to using the resource the mutex protects, as this concerns changing the mutex object itself. The code will then check if the mutex is in use by attempting to set InUse. If it isn't in use, the thread claims the resource, and the resource Is set to in use until the thread unlocks the resource later. If the thread is already in use, the thread will then add itself to the linked list that starts with a pointer in the mutex object, and then block. Then when the resource is released, the thread running will then check the start of the queue, unblock the thread, assign the resource to be in use by that thread, and then remove it from the queue. By doing this, you can maintain a record of which threads are waiting for the resource, and what order they requested it in. A similar system is used when threads are waiting for another to end, except in that system, all the threads in the queue are unblocked, not just the first in the queue.

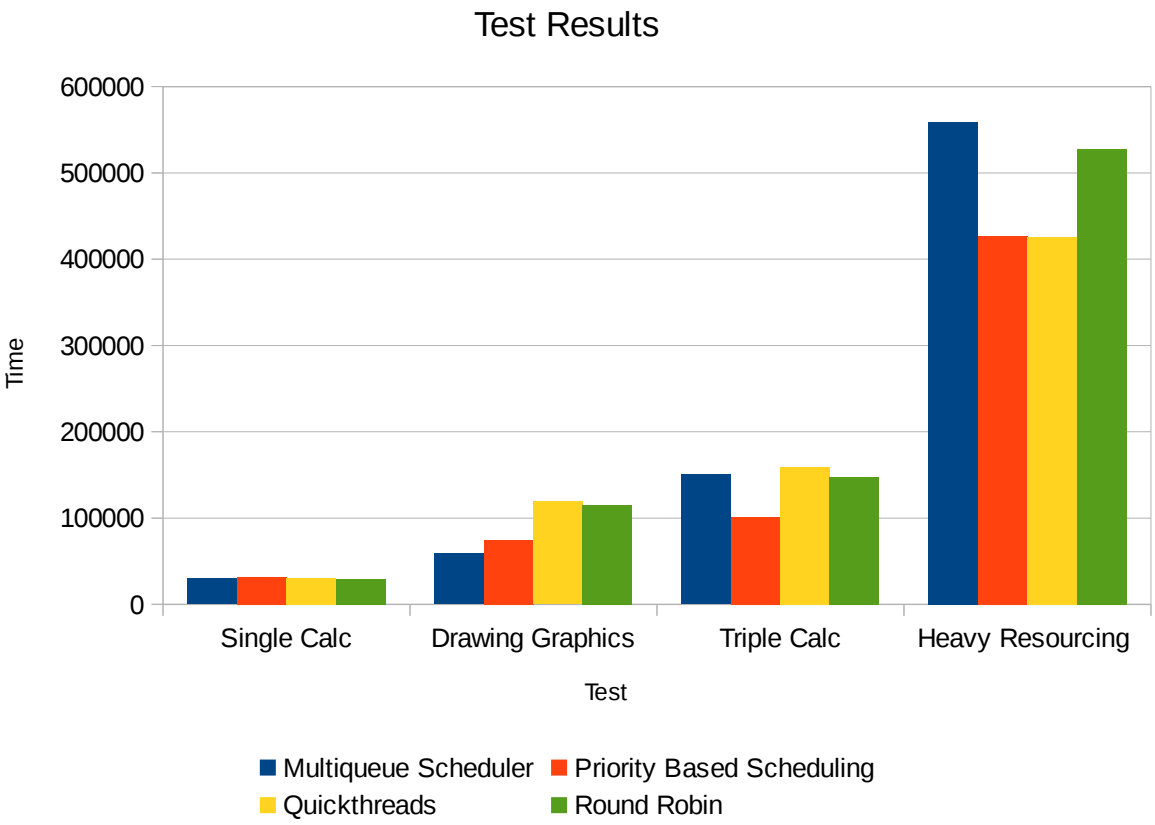
4.6 Testing

Testing consisted of four tests being ran automatically over each algorithm, each test being timed to see how long each respective algorithm took to complete. These results are then written to the serial port which is output to a text file on the host PC. These results have been compared and compiled into a table and graph. The tests cover single calculation, drawing to the video buffer, running three calculation threads at the same time, and running one hundred threads that require access to a single mutexed resource, in order to test how the algorithms handle lots of threads contesting mutexes, as well as an extreme amount of threads. These tests cover most basic situations without being too specific. This is to make sure the results give a general idea of performance without too much bias towards the test itself. Testing was ran multiple times in order to confirm the results were correct.

The single calculation test was designed to show the scheduler’s ability to handle a simple situation, and will mostly show a scheduler’s ability to swap tasks effectively. The drawing graphics test will assess a scheduler’s ability to draw to the screen. This is a crucial part of a modern day operating system, and it must be efficient at handling this task. The triple calculation is similar to the single calculation test, however it also tests the scheduler’s ability to handle multiple threads in the best method possible. The hundred threads competing for a single resource, named the Heavy Resourcing test, was designed to push the scheduler to an extreme, forcing most of the hundred threads to wait in the resource’s mutex queue. A good scheduler would be able to cope with the heavy load and finish the test in a quick manner.

4.7 Results

Test one, the single calculation test, gave similar results for all algorithms. This is most likely due to the simplicity of the test, not allowing much variation in the algorithm. Test two, the graphics test, priority based scheduling and multi queue had the clear advantage, due to being able to prioritise certain threads over others. As the other two have no functionality for handling UI threads, they struggled a lot more. Test three, the triple calculation test, and test four, the heavy resourcing test, both had similar results, with priority based scheduling being significantly faster than the other algorithms due to it's dynamic prioritising. Round robin and multi queue both did poorly due to both of them working in a similar way for these tests, as multi queue does not utilise the background and UI priority levels in these tests, with multi queue doing slightly worse due to the large overhead when switching. Quick scheduling had strange results on these tests, as it performed at a similar level in test four to priority scheduling, but did the worst in test three.



Based on the results of the tests, as well as other considerations, priority based scheduling was implemented. As well as having the best speed over all the tests, it provides the strongest level of control over threads out of the schedulers. However, the biggest problem is requiring priority levels to be set for each thread. However further modification and additions to the scheduler may be able to fix this.

4.8 Debugging

In order to make the system stable, it must be as bug free as possible. As the multitasking system of an operating system fills a significant core role, by which it could not run without it, it can be considered a critical part. This means that it must be checked for bugs as thoroughly as possible, as one error could cause a system crash.

Firstly, the system was stress tested. This included tests such as creating a large amount of threads and processes, as well as making a large demand for a single resource. Rapidly creating and deleting processes and threads was also tested. Throughout these tests, the system continued to function correctly, as expected. However during testing a bug regarding processes being deleted and created quickly was shown. During round robin scheduling, when a thread was deleted, it would remain in the round robin ring until its turn came up, where it would be skipped over, however if a thread is deleted, and a thread is created in the same slot, this created two entries in the ring, which caused a smaller loop to occur, cutting out threads outside this loop. This wasn't a problem in the final scheduler, as it contains no round robin element, however future improvements may include one. The mechanisms involved in blocking require a linked list in a similar manner, and so the bug was fixed in these.

The security of a system was also tested, to make sure it is not vulnerable to being manipulated. These tests mostly covered checking ring priority rights. The idea of the tests were to cause the system to break. Tests include attempting to access different areas with and without priority, and attempting to disrupt the scheduler and cause it to stop functioning. All of these tests passed without problems. Currently it is possible for a program to manipulate the stack and start running kernel code, however this is limited by the program's privilege level. Once the kernel and userspace programs are separated, this can be fixed.

The unit test table has been included under the Appendix A.

5 Evaluation

The aim of this project was to develop and implement a scheduler for PrimOS. In order to do this, the objectives were defined as key points to be achieved in order to consider the project a success.

1. The first objective was to establish a base system by adding required features to PrimOS. All features that were needed have been successfully added, however a few of the features originally considered to be required ended up being unnecessary. These features will most likely be added once the project has been completed.
2. The next objective was to implement threading. At the beginning of the project, PrimOS only had processes, without any such threading. Essentially they were single threaded, with the processes themselves getting scheduled. To consider threading to be implemented, each process must have its own table of threads, with the threads being scheduled instead of the processes. There must also be some features added to allow for thread interaction and manipulation. These were all implemented successfully, with processes becoming a group of threads that share memory space. It is also possible for threads to be created and started, as well as for threads to wait for other threads to finish.
3. The third objective was to implement a number of schedulers to be tested. Four schedulers were successfully implemented for this objective, which was enough for the project to progress.
4. Objective four was to test and decide on the scheduler to use. This was achieved by running four simple test programs that give a good range of situations in order to find the fastest scheduler, as well as taking into account pros and cons of each. The conclusion was that priority based scheduling would be the most appropriate. As this objective, as well as all previous objectives were completed fully, the project can be considered a success.
5. The last objective was an extra objective that aimed to enhance and extend the system implemented. It was not considered required for the project, but it was there to allow time for the system to be improved on beyond the requirements of the project. Multicore support was suggested as a possible enhancement, however this would've taken a significant amount of time to implement. However numerous other enhancements have been implemented, such as the ring privilege system. While this objective was not needed for the project to be successful, it has been beneficial for the project overall, providing extra tools and stability/security to the operating system.

Overall the project was a success. While there were issues brought up, these were easily managed and kept to a minimum. The system has been given a scheduler that works well, and is in a position to move onto another stage of implementation for the operating system.

During the start of the project, tasks took longer than was expected, which put the schedule behind. This was made up for by shortening the time spent on later tasks, as well as dedicating more hours per week to the project. In doing so, the project was brought back on track. This delay may be caused the earlier tasks having significantly more depth to them and required more time than initially expected, while the later tasks were based on the work of the previous tasks, and so didn’t require as much work.

While the project didn’t have any significant risks, it still had some which needed to be taken into account. As the biggest risk to the project is data loss, everything has been constantly backed up to an SVN in case of any problems. The work load was underestimated, however it was early on so the project was easily able to manage.

5.1 Further Improvements

Following the completion of this project, there are numerous improvements that can be added to PrimOS. Deciding on specified initial priority levels for different uses, such as assigning priority level 9 for UI threads, would be a good starting point. Another useful change may be to modify the scheduler to take on advantages of other schedulers. For instance implementing the quick threads may be beneficial to the scheduler. Optimising the scheduling code would also be a big bonus, as a significant amount of time is spent deciding the next thread to be ran.

The scheduler as currently implemented works as intended, however there are a number of improvements and changes that can be made to increase the speed and stability of the scheduling. One improvement may be to decide on set priority levels for different jobs. For instance, UI threads could be assigned a higher base priority to background threads, similar to the multilevel queue algorithm.

Substantial amounts of the OS overall were developed prior to adequate understanding of the problem. Because of this, one of the first steps following this project will be to redesign and improve every section of PrimOS. This will result in a stabler, faster, more usable system overall. Following this, the next step will be to separate the userspace programs from the kernel. Currently all userspace programs ran on the OS are compiled into the kernel. This means that the software has access to variables and data in the kernel that it wouldn't usually be able to access so easily. To achieve this, I will first be setting all the programs currently compiled into the kernel, to be compiled into a separate object file, before then merging into the kernel. This will mean that the programs do not have implicit access to kernel variables, and would have to request data through the system call interrupt alone. Once this has been achieved, the kernel would then need to be capable of loading a program into userspace memory, and execute it. Once this has been achieved, there will be full separation of kernel and userspace programs.

Other sections of the operating system can be substantially improved. For instance, the malloc system works fine, however it can be improved through making block sizes more specific. One idea could be to make all block sizes powers of two, starting with a substantial size. This would effectively remove all external fragmentation, as all the blocks would piece together well, and the smallest gap possible would be a usable size.

Another improvement to be added is user interactivity and usability. Currently, PrimOS is capable of taking keyboard inputs in, however there is no interaction between the user and the operating system. In order for the multitasking to be of use, the user must be able to run and use software. For this, one possible step would be to implement a previous suggestion that was not needed for this project, a command line. This command line could be then used to run, and possibly even create, programs for the operating system to use.

6 Conclusion

The goal of this project was to research, test, and implement a scheduler into an operating system that is currently under development, nicknamed PrimOS. Achieving this goal has meant implementing multiple different systems in order to create a stable, and functioning operating system. This project has reached a stage where it can be considered to have met this goal. However meeting this goal, and subsequently completing the project, does not mean the end of development, as the operating system still requires a lot before it can be considered useful, such as having user input, permanent storage, etc.

Appendix A: Unit Testing

Test Name	Expected Result	Actual Result	Status
Create a process and execute	Process executes correctly	Process executes correctly	pass
Create and execute a thread inside a process	Thread executes correctly	Thread executes correctly	pass
Create multiple threads and multiple processes	Threads and processes execute correctly	Threads and processes execute correctly	pass
Utilise a mutex resource	Mutex is set to in use and then unset afterwards	Mutex is set to in use and then unset afterwards	pass
Attempt to utilise a mutex resource when it is in use	Current thread is put into the queue for the resource	Current thread is put into the queue for the resource	pass
Create large number of processes	Scheduler works fine	Scheduler works fine	pass
Create large number of threads in a process	Scheduler works fine	Scheduler works fine	pass
Create large demand for a single resource	Resource queues threads fine	Resource queues threads fine	pass
Remove all but the idle thread	Scheduler repeatedly runs the idle thread	Scheduler repeatedly runs the idle thread	pass
Quickly created and deleted threads and processes	Scheduler works fine	Scheduler works fine	pass
Utilise a syscall	Syscall executes correctly	Syscall executes correctly	pass
Attempted to access the physical memory map without a syscall	General Protection exception should trigger	General Protection exception triggers	pass
Attempted to map virtual addresses with a syscall	Virtual address should be mapped to an empty physical address	Virtual address mapped to an empty physical address	pass

Appendix B: Timeplan

		University Calender Weeks																														
No.	**Task Name**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**
1	Implement PrimOS Functionality																																	
2	Implement Base Scheduler																																	
3	Implement Round Robin																																	
4	Test Base System																																	
5	Research/Implement Threading																																	
6	Research/Theorise Multitasking Systems																																	
7	Interim Report																																	
8	Prototype Useful Systems																																	
9	Implement Best System																																	
10	Test Final System																																	
11	Research/Theorise Additional Features																																	
12	Implement Additional Features																																	
13	Final Report																																	

Appendix C: Risk Analysis

Risk	Severity (L/M/H)	Likelihood (L/M/H)	Significance	How to Avoid	How to Recover
Data loss	H	L	HL	Use two forms of SVN, including the university SVN.	Recover from the backups.
Underestimating workload	H	M	HM	Review time plan often to be aware of problem early.	Invest more time into the project.
Badly timed critical error	M	L	ML	Test early and often, use SVN to track changes.	Revert to previous version that worked.
Faults in PrimOS	M	M	MM	Test and secure PrimOS as the first priority.	Find and correct the fault.
Invalid results from comparing and evaluating prototypes	M	M	MM	Run tests multiple times to confirm results.	Redo the tests.
Insufficient information to include in reports	M	L	ML	Make notes throughout project, and tabulate data well.	Go through project to recover information.

References

QUOTE Tanenbaum, A. (2014). Modern operating systems. 4th ed. Englewood Cliffs, N.J.: Prentice Hall.

Silberschatz, A., Galvin, P.B. and Gagne, G. (2012) *Operating system concepts*. 9th edn. United Kingdom: John Wiley & Sons.

Intel architecture software developer's manual (1997) Intel.

Scheduling Algorithms

Operating System Scheduling algorithms. Available at: http://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm (Accessed 3 May. 2016).

Wikipedia, (2016). *Scheduling (computing)*. Available at: https://en.wikipedia.org/wiki/Scheduling_%28computing%29 (Accessed 3 May. 2016).

Deadlocks

Deadlock Solutions (2012) Available at: https://courses.engr.illinois.edu/cs241/sp2012/lectures/27-deadlock-solutions.pdf (Accessed: 4 May 2016).

Mutexes

Robbins, D., President and Technologies, G. (2000) *Common threads: POSIX threads explained, part 2*. Available at: http://www.ibm.com/developerworks/library/l-posix2/ (Accessed: 4 May 2016).

OSDev.org • view topic - Mutex implementation (2007) Available at: http://forum.osdev.org/viewtopic.php?t=14261 (Accessed: 4 May 2016).

Memory Allocation

Golick, J. (2013) *Memory Allocators 101*. Available at: http://jamesgolick.com/2013/5/15/memory-allocators-101.html (Accessed: 4 May 2016).

Memory management from the ground up 2: Bitmap allocator (2010) Available at: https://eatplayhate.me/2010/09/04/memory-management-from-the-ground-up-2-foundations/ (Accessed: 4 May 2016).

Kasper, E. (2016) *What is internal & external memory fragmentation?* Available at: http://everydaylife.globalpost.com/internal-external-memory-fragmentation-28851.html (Accessed: 4 May 2016).