Operating Systems Final Exam (CS 470)

Central Washington University

Student's name (Printed): Andrew Struthers ID: 41371870

Date: 3/17/2023

Time allotted**: 120 minutes**

Instructions:

In this exam there are 20 questions. Altogether the final exam is worth 100 points. The questions' overall worth is marked in bold at the end of each question. However, for partial answers credit will be given, see the points mentioned after each particular item/notion separately.

During the written examination, no textbook, slides or other printed/handwritten/digital materials should be considered. The usage of these materials or help from a fellow student during the exam is considered cheating, and it is subject to sanctions regulated by the university policy.

If you are at home you can print the document, write on the exam sheet, scan back the document and upload it to Canvas! Or you can just simply type the document using MS Word or some other Word clones (Libre Office, Google Docs, etc.).

I prefer that you type the document (using the exam sheet and not more than the space allocated for each answer). Please submit it through Canvas as a PDF! Convention: Lastname_FinalExam.pdf

Honor Code Statement: *I pledge that this written exam is solely my work, and that I have neither given, nor received help from anyone (including textbook, Internet, slides, chat, phone, etc.)*.

Signature: Andrew Struthers

1) In which address space the memory is contiguous (1p.)? Explain why there is a need for such a concept (3p.)! (**4p**.)

   The address space that is contiguous is the virtual address space for each process. Each process has a base register and a limit register, where the base register is the value of the smallest physical address, and the limit register represents the range of logical addresses the process has access to. The memory allocated for each process is contiguous, and this is important because it allows us to do pointer arithmetic. We can access the next element of an array, or the next value in memory in general, by looking at a pointer, then incrementing the pointer by the size of the value stored in that memory location, and then reading the new value in memory. Pointer arithmetic is proof that the virtual address space for each process is contiguous.

2) Explain the concept of static library in Linux (2p.)! Explain the concept of dynamic library in Linux (2p.)! Please give a concrete example when a static library would be more appropriate to be considered and another example when the dynamic library would be more efficient (2p.) to be used in your program! (**6p**.)

   Static libraries are precompiled pieces of code that are linked to the process at compile time. When the static library is linked to the process, the process has access to all of the code included in the executable library. Dynamic libraries are different in the sense that dynamic libraries are linked at runtime to the process, and they are linked dynamically. This means that, if the process only needs one function or a handful of definitions from the dynamic library, the process will only have references to only the stuff that it needs. An example of when a static library might be better is when the program or library is very small. Smaller processes naturally use up less memory, so working with a small static library can be more efficient than using a small dynamic library. Dynamic libraries add more benefit when the program or library is very large and memory management becomes quite important.

3) What is an operating system (1p.)? Why an operating system should be layered (2p.)? (**3p**.)

An operating system is a program that serves as an intermediary between hardware and user processes. Operating systems provide many functionalities to the users such as resource allocation, execution control, I/O functionalities, process management, memory management, and storage management. An OS also provides an user interface to the users to allow the user to manipulate the hardware effectively. Layered OS is good because of security reasons and modularity. For security, if the OS is layered where each layer has an API or some sort of structure for talking to the layer below, each layer will only have access to what exactly it needs. This helps with security because the user layers won't have absolute access to every other system. Layering also helps with modularity, because instead of trying to rebuild the whole OS when one system doesn't work, you can just focus on that one module and swap it in or out as the needs change.

4) Explain why system calls are necessary from the operating system's point of view (3p.)! Give two examples for system calls (2p.)! (**5p.**)

System calls are necessary from the OS point of view because they provide an interface to services provided by the OS, without allowing the users to directly access OS specific functionality. These system calls are low level API to the OS functionalities where the OS can control the security, performance, and information given. The users don't necessarily need to understand how the system calls are implemented. A few examples of system calls are system information calls, memory management system calls like malloc() or free(), accounting operations like time(), and file read/write operations.

5) Explain the concept of PCB (1p.)! Why such concept is necessary (2p.)? How it is implemented and how you would implement it to make the context switch efficient? (4p.)! (**7p.**)

A process control block (PCB) is a large structure that stores the information status of each process. Each process has a PCB attached to it that holds information like the process state, the program counters, the state of CPU registers and scheduling information, I/O information, and file information. PCB is important because it allows the CPU to use context switching. Context switching allows the processor to load and unload processes to execute many processes intermittently, instead of executing one process to completion before executing the next one. The Job Scheduler can load and unload processes when necessary, and this wouldn't be possible without PCBs. This is implemented in Linux by a large struct in <sched.h> that holds all the required information. Context switching requires hardware support and is time dependent on the hardware, so I would focus on improving the hardware implementation of context switching by using faster transistors or a more complex system to speed up the overhead "useless" time context switching requires.

6) What is a medium-term scheduler (1p.) and a short term scheduler (1p.)? Discuss which one is more important and why! (3p.)? (**5p.**)

A medium-term scheduler is in charge of swapping processes into or swapping processes out of long term memory. Sometimes there are so many processes in memory that there is no room for a new process, so some processes need to be dynamically written to memory for longer term storage, freeing up memory for new processes. The medium-term scheduler handles this reading/writing to storage to keep memory free. The short-term scheduler decides what process in the ready queue should be executed next and allocates the CPU to get ready for this process's execution. The short-term scheduler is invoked very frequently, whereas the medium-term scheduler is invoked only when necessary, as writing to storage is slower than writing to memory. The short-term scheduler is more important, because without the short-term scheduler the CPU would have no idea what process to execute next.

7) Explain the Many-to-One multithreading model (2p.)! Explain which thread model would allow the best compromise considering blocking and resource usage. Elaborate on your answer (3p.)! (**5p.**)

The many to one multithreading model is when many user threads get mapped to one kernel thread. The kernel thread will execute each user thread sequentially, which means that the threads not currently in execution are blocked by the current execution thread. The best model to prevent blocking and maximize the effectiveness of the resources is the two-level threading model. In the two-level model, there is a many to many model where many user threads are mapped to many kernel threads, as long as there are more user threads than kernel threads. In the two-level model, some threads also get special permissions and can have a one-to-one relationship with a kernel thread. Kernel threads take many more resources than user threads, so minimizing the number of kernel threads will decrease the resources required. But to prevent the blocking in the many to one model, mapping many user threads to many kernel threads with special permissions with a one-to-one mapping is the best balance of resource usage and minimized blocking.

8) Consider a system implementing a multilevel queue scheduling. What strategy can a computer user employ to maximize the amount of CPU time allocated to the user's process? (**5p.**)

In a multilevel queue scheduler, there are multiple queues that run different scheduling algorithms to maximize the CPU usage and minimize the time each process has to wait. To maximize the CPU time allocated to user processes, the multilevel queue system can implement round robin scheduling so that any process in the ready queue of a CPU has equal access to the CPU time. This will maximize the amount of time each process is running in the CPU. If the multilevel queue implemented priority scheduling, the user processes might have less time in the CPU than system level processes. Finding a good balance of letting system processes run when they need to, but not ignoring the user processes in the CPU can be tricky, but round robin scheduling at least guarantees that processes get equal time.

9) Explain in details the notion of process (3p.)! Compare it with the concept of thread! Elaborate your answer (2p.)! (**5p.**)

A process is a program that is being executed. Each process has its own PCB, PID, resources, and memory. A process has its own address space and is not allowed to access memory outside of its own space. Because each process has its own memory, getting processes to communicate with each other requires inter-process communication, like shared memory or network communication over sockets. Threads are a much lighter weight version of execution than a process. Each thread shares the same address space and resources as the process that created it, and threading allows multiple execution within a process with parallelization. The differences between the two are a process has completely independent execution from any other process, and threads are very dependent on the process they came from. Threads have shared resources between themselves and the process that created them, whereas processes need to have specific procedures to communicate. Processes take up more space in memory since each one has its own PCB and resources allocated to it, and they are isolated from other processes. A process never has to worry about synchronization issues with itself, because it only has one path of execution, but threads need to have careful synchronization management.

10) Explain in details the priority based scheduling mechanism (2p.)! Explain why this method is better/worse than some other scheduling mechanisms (2p.)! (**4p.**)
Priority scheduling is when each process in the ready queue has a priority assigned to it, and then the short-term scheduler will queue up the process with the highest priority (priority values are inversely proportional to the true priority, so smaller priority values are more important). The process with the highest priority will get all of the CPU resources until it is done executing, then the short-term scheduler moves on to the next process. This scheduling is very good for super important processes like low level system and management processes, but it can leave low level user processes out of the CPU for quite a long time relatively, and some very low priority processes might never execute. Priority scheduling can be improved with an aging mechanism, where every time delta each process in the ready queue has their priority increased, so that no process gets left behind permanently.

11) Explain in details the critical section problem! How would you solve this problem? (**2p.**)

The critical section problem is a problem that happens in parallel programming where multiple threads or processes are trying to access shared resources, known as the critical section. Having multiple processes or threads accessing the same memory can lead to race conditions that will lead to incorrect data processing. If multiple threads are accessing the same integer in memory with no safeguards in place, some unexpected and unpredictable stuff could happen. To solve this problem, we must make sure to use mutex or atomic variables to prevent multiple threads from accessing the same part of memory at the exact same time. Once we start forcing the threads to synchronize before accessing the same values in memory, this problem is minimized. With mutex or other synchronization techniques, we can force only one thread to operate on memory at any given time.

12) Explain the concept of soft affinity (2p.) and hard affinity (2p.)! How this influences (3p.) the CPU scheduling? (**7p**.)

Affinity describes the desire of a process to remain on the CPU or core that it is currently on after a job scheduler performs load balancing. Soft affinity means that the process doesn't really care if it gets reallocated to a different processor or core during balancing. Hard affinity is when a process must stay at the core that it was originally allocated to. This influences CPU scheduling because schedulers seek to balance the load of all processes evenly between each of the cores in the computer. With soft and hard affinity, the load balancer needs to not only evenly allocate resources to each core but do so while respecting the affinity of each process. If we created many processes that all have a hard affinity for core 0, for example, the job scheduler would have a hard time load balancing all processes evenly among the cores because most processes must stay on the first core.

13) If you would need to install a Linux on your computer, which method would you choose a) native, b) virtual, c) container and d) other? Please motivate in details your choice! (**5p.**)
I would install a container on my computer because containers are very lightweight builds of the OS with very good performance and little overhead. A native install would be more performant and provide more functionality than a container, but I would have to give up my beloved Windows environment. I would not install a virtual image of Linux because virtual images are quite a bit slower than a container. Even though I could work with a GUI in a virtual install, the performance hit with a virtual install compared to a container would make the ecosystem frustrating to deal with. The preferred solution that combines the benefits of all these installs would be setting up a dual boot and a partitioned hard drive for the second OS, then I could boot into either Windows or Linux depending on my current needs. A dual boot gives the benefits of a native install without losing the option of using Windows and without the slow down and performance decrease of a virtual install.

14) In the SJF CPU scheduling exponential averaging is playing an important role. How the parameter alpha is influencing the nature of the estimation? (**5p**.)
In SJF scheduling, predicting the runtime of the process from previous estimations helps the scheduler to sort and prioritize the proper job. The basic prediction model relies on an alpha parameter, where the alpha parameter dictates the weight of the actual previous length of the CPU burst and the weight of the predicted previous CPU burst. The alpha parameter lets us weight what the scheduler should consider more, the predicted execution time or the actual execution time. Balancing this parameter to get the best prediction is quite hard, but oftentimes an estimate is good enough. Because the alpha parameter represents a weight as a percent, alpha must be between 0 and 1 (0% and 100%).

15) What is the external fragmentation for a hard drive (1p.)? Propose and explain a method to mitigate this problem (4p.)! (**5p.**)

External fragmentation of the hard drive is when the total memory space exists in the hard drive for a new file write operation, but the space required is not contiguous. This occurs when the OS allocates slightly larger space in the hard drive than the file requires, or when files get deleted and the hard drive memory starts filling with “holes”. One method to reduce this problem is to use compaction. Compaction shuffles the contents of the hard drive and brings all the files in to one large block. This will allow all of the “holes” to be removed, because all of the blank space will get moved to the end of the space.

16) How the size of a sector on a disk is influencing the access speed? (**4p.)**

The sector is the smallest unit of data that can be written or read, and usually the sector size is in bytes. Larger sector sizes can result in faster access speeds because the disk head needs to perform fewer operations when reading or writing to large sectors. If the sector size of a system was 1024 bytes, and there was a 10 kB read operation, the disk head would only need to perform 10 reads, whereas if the sector size was smaller there would need to be more read operations. However, the sector size can’t just be set to 10GB for example, because if the sector size is larger than the data being written to the drive, the disk will need to allocate more size than the data takes up, creating holes of inaccessible memory on the disk. If there are many holes on the disk, and the data being read is a lot smaller than the sector size, the disk head will spend more time reading the sector than necessary, reducing the read/write speed. Balancing the sector size so that there are few read/write operations without creating a lot of holes and inaccessible or wasted space is necessary to optimize the disk access speed.

17) The output of this C code is presented below! Please explain this output? (**6p**.)

```
#include<stdio.h>
#include<pthread.h>
#define N 5
void * Hello(void * p)
{
    int Value = (int)p;
    printf("The value in the thread is:%d\n",Value);
}
int main()
{
    pthread_t myThreads[N];
    for(int i = 0; i<N; i++)
        pthread_create(&myThreads[i],NULL,Hello,(void*)i);
}
```

Output:

```
The value in the thread is:2
The value in the thread is:4
The value in the thread is:0
```

The output of this code seems out of order from what we would expect. If this was a serial program, we would expect the code to print out 0, 1, 2, …, N, instead of in a seemingly random order. The reason that this occurs is because, when we create the threads, each thread is entered into a job scheduler and the scheduling algorithms determine which thread executes next. The threads run in parallel and can potentially run on different cores that have different jobs waiting in their respective ready cores. This means that it is very difficult to determine the exact order of execution of each thread and is why stuff like thread synchronization is important. Parallel code rarely executes in the exact order that a serial counterpart would execute, which is why the for loop that iterates from 0 to N doesn't necessarily result in thread functions that output in order.

18) Assuming a 1-KB page size, what are the frame numbers and offsets for the following virtual address references (provided as decimal numbers)! (**4p.**)

a) 312
b) 4095

Let's assume that this is a 32 bit system. Because there is a 1kB page size and 1024 = 2^10, we know that the lower 10 bits of the address is the page offset, and the remaining 22 (32 – 10) bits are the frame number. Since 312 = 100111000b, which is an 8 bit number, and the lower 10 bits are frame offset and the remaining 22 bits are all 0, we know that 312 maps to frame 0, offset 312. Using this same process for b), we know that 4095d = 111111111111b, which is 12 bits. The lower 10 bits 1111111111b = 1023d, and the remaining 22 bits are 11b = 3d, therefore we know that 4095 maps to frame 3, offset 1023.

19) In a C/C++ program how would you prove that different threads run inside the same process! (**7p.**) [No need to write functional code! Just explain the concept!]

Each thread running the thread function assigned at thread creation could use the system call getpid() to get the process ID, then print out this process ID. We will see that each thread has the same process ID, meaning that all the threads that process created are all a part of the same process, and thus run inside the same process. Another way that we could prove that each thread runs inside the same process is by having each thread do an operation on a global variable. Because processes can't inherently access other processes memory, operating on a global variable within each thread without throwing a segmentation fault would prove that each thread is inside of the process and has access to the process's resources.

20) Consider a disk queue with requests for I/O to blocks on cylinders: 71, 146, 32, 90, 45, 24, 161, 75 and 63. If the disk head initially is at cylinder 83 list in which order the SSTF algorithm would visit these cylinders! (**6p**.)

SSTF is Shortest Seek Time First, meaning that this algorithm will select the next request based on the shortest distance between the current disk head position and the next. This means that, starting at position 83, 75 is the closest next request, so the head would move to 75. From 75, the head would move to 71 because that's the next closest position from 75. This process would continue until all the requests are processed, and the final order for requests would be 83, 75, 71, 63, 45, 32, 24, 90, 146, 161