Operating Systems Exam – UFCFWK-15-2

Ashley Pearson – 20001030

ANSWER SHEET.

1a.

Hardware is the physical component of a computer, the low level, least abstracted element containing the CPU, memory and I/O devices, these components form the physical computer that to the average user are widely unusable without a form of interface. Operating Systems (OS's) provide the 'middle man' function both to the user and user applications and allows for communication between high level software and hardware in a controlled fashion using API's, ABI's and syscalls. They provide a consistent experience through GUI's, CLI's or both and control what system functions a user has access to using permissions. User applications are processes started and stopped by the user and do not run for the entire runtime unlike the OS. A user may for example, open a word processor, create a file using a keyboard, save the file to disk and then close the application, this will require multiple syscalls via API's to access kernel level functions such as open(), write() or exit(), though the user will never see this due to the abstraction provided by the OS.

1b.

Security.
Authenticate and then control any given users permissions/access.
Segregate individual processes to avoid accidental or intentional accessing of each other's memory space.

Resources.
Control and time access to shared resources.
Manage RAM efficiently.
Use drivers to communicate with different devices.

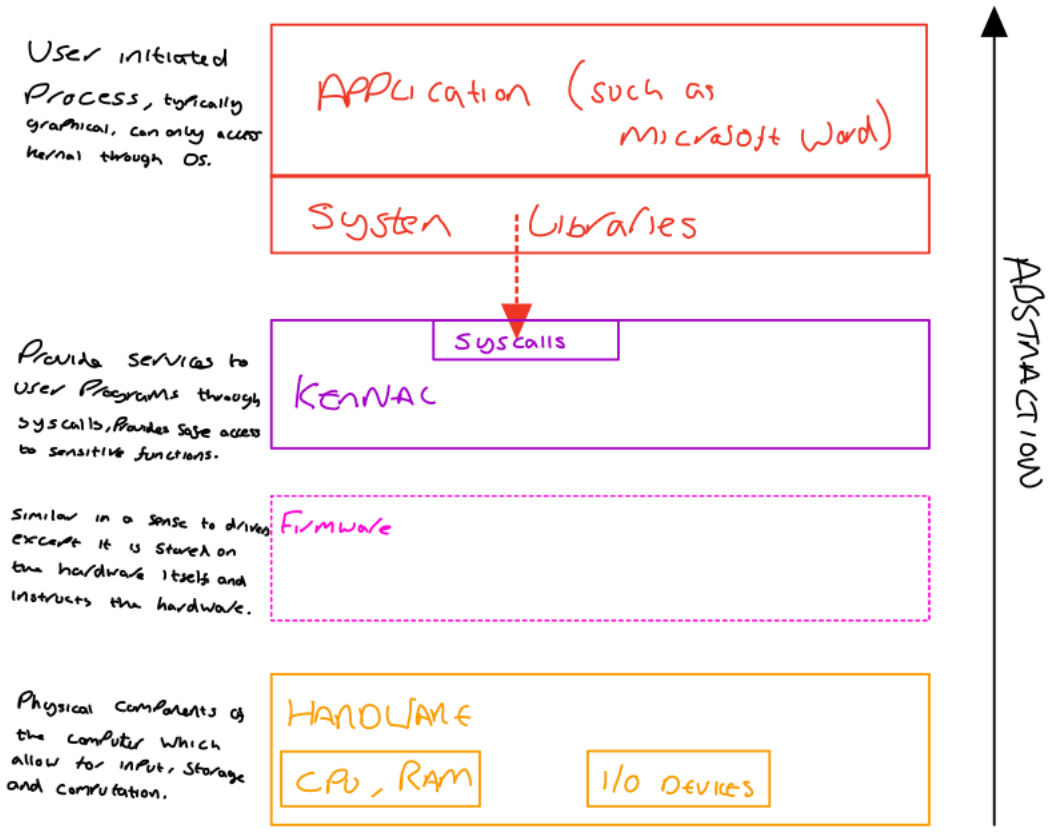
Application layer.
Software in ring 3, user space has no direct hardware access.
Use inter-process communication (IPC) to share data.
Schedule execution of applications.

2a.

The computing stack describes the levels of a computer and the layered structure in which they exist and work together. At the top of the stack is the application layer, sitting in user space this is where the average user will spend most of their time, with compiled applications such as word processors accessing lower levels using system libraries, including syscalls. The operating system will then assist in performing the required function if the request is valid and safe, drivers are used to communicate between the hardware at the

bottom of the stack such as memory or I/O devices like keyboards, once the driver has retrieved the requested information from the device it is then returned through the BIOS to the OS and in turn, the user application.

2b.



3.

PC (Program Counter) points to the next instruction, as such the next instruction to be executed is Instruction A.

I believe instruction C may print (“E”).

4.

As in the diagram in 2b, the three primary hardware components in a computer are the CPU, memory, and I/O (Input/output) devices.

The CPU is a central component in computers and is responsible for taking instruction from software and performing the Fetch, Decode and Execute cycle, whilst modern chips are extremely complex, they are ultimately extremely fast calculator capable of basic arithmetic.

Memory is primarily composed of both temporary memory, random access memory (RAM) and permanent storage (SSD, HDD). Temporary memory is volatile and data is lost when the computer is powered down, but allows near instant access for use by the CPU. Permanent storage on the other hand is persistent and non-volatile, it is used to store data, user applications and the OS itself, the trade-off being a considerably slower write speed, especially on HDD.

I/O devices are the devices used by the operator (often human) to get data into and out of the system, a keyboard for example is used to input data into the system, where as a monitor is used to display the current state of the system, including data inputted using the aforementioned keyboard.

5.
To process an instruction the CPU follows the below cycle -

Fetch.
The CPU uses the program counter to fetch the address to be executed next and stored in the IR.

Decode.
The instruction is then decoded, whereby the CPU decides what needs to be done with the instruction, such as moving values to different registers.

Execute.
The command is then executed, potentially utilising the Arithmetic logic unit (ALU) if math is required, the CPU then loops round to fetch and continues indefinitely.

6.

File/data abstraction – An OS presents files simply to the user, requiring only its path (such as the desktop) to allow manipulation of the file, if users were required to navigate actual memory, it would be almost impossible for typical users, directory files are used to help navigate.

Memory abstraction – Due to the multitude of memory management techniques available such as paging, virtual memory etc using abstract memory can help, a process has its own address space separate to any other process and all further memory management is managed by the MMU.

Thread abstraction – Through abstraction a process may be executing multiple threads concurrently, each with its own set of instructions and variables. Despite this, when running a single multithreaded process multiple threads run individually allowing for thread context switching, which is cheaper on the CPU than process switching.

7.

A thread can be viewed as a small unit inside of a process that can be run in parallel with one another as they share the same virtual address space ‘owned’ by the process they belong to, context switching amongst threads is considerably ‘cheaper’ than switching between processes.

8a.

Context can be thought of as the status of a process, containing the process state, CPU registers, data segmentation and I/O information. If it is necessary to interrupt a process or switch to another, the data is stored in the Process Control Block (PCB).

8b.

Context needs to be saved when switching processes to ensure that when the process is resumed the memory context can be restored and the process can continue from its previous state without re-treading all previously executed steps.

9.

Processes are slower with regards to creation, context switching and termination. As threads share memory, they can also share information quicker than processes and multiple threads are treated as a single task by the OS, if the CPU is working on a process, multiple threads inside of it can be working concurrently as opposed to processes which can result in blocking.

10a.

As message passing does not used shared memory, monitors would be more appropriate, especially as semaphores are atomic, meaning they cannot be stopped nor interrupted by other processes.

10b.

In a shared memory IPC system to ensure both processes can safely read and write a shared resource it is necessary to implement semaphores to stop multiple processes trying to access the space at the same time resulting in a race condition. Using a binary semaphore either locks, or unlocks a resource, making it accessible or not depending on its status, this way only one process can access the resource at once, with the other(s) being made to wait.

11.

The primary difference between pre-emptive and non-pre-emptive scheduling is the way in which the CPU allocates resources. In pre-emptive, a process can be interrupted by the OS if it is a priority and the existing process's state will be switched to 'waiting'. Shortest remaining time first (SRTF) is a pre-emptive algorithm where the task with the smallest amount of time to completion runs first, if 3 processes are in the que but a 4th arrives with a shorter completion time, it will be processed with priority.

In non-pre-emptive however the process cannot be interrupted and will run until termination, or a context switch. First come First Served (FCFS) is a non-pre-emptive algorithm works in a chronological order, whereby the first process in the que is processed first, before moving to the next process in an orderly fashion.

12.

Both methods are valid approaches, however a common problem with pre-emptive scheduling is that low priority processes often get starved of CPU time. Due to the allowed switching and interruption, low priority processes (such as long processes in SRTF) can often be continuously put behind other processes, not allowing them the time to execute.

13a.

Fragmentation is the splitting of files across the physical memory due to processes being loaded in and out of memory if a file grows in size beyond its allocated space, the additional data must be stored in a different physical location, this results in a single file having multiple locations.

13b.

Fragmentation causes several problems but ultimately the issue is performance cost and slow down on spinning disk HDD's, the slow physical way in which these disks operate require the drive to scan across the entire physical space to locate the entirety of a file, this effects the computer at a system level but also at a user level, loading of documents and files may take longer than is required due to the fragmented nature of files. Pre-emptively allocating chunks of data for files to grow into is one way in which files can be kept 'geographically' together. After the fact, defragmentation can be used to reorder files and make them contiguous, though this method should not be used on flash storage as no physical movement is required to locate file locations, it can have a negative impact by performing unnecessary writes to the drive, reducing lifespan.

14a.

Paging is a method to mitigate the issues caused by fragmentation in memory, memory is divided into predefined smaller chunks that do not need to be set out in a contiguous way. As processes must be able to 'see' memory in a linear state it is necessary to move data in

such a way that it is physically ‘in order’, though the data is split up it is still in a linear fashion when it needs to be accessed by a process.

14b.

Paging offers advantages as outlined above, it improves allocation times as the OS has to calculate the number of pages required to hold a process as opposed to allocating in detail the process to individual addresses, likewise the smaller page numbers stored in the data table are easier to read than full memory addresses. As pages can be swapped as whole items it also reduces swap times as a 4kb page file can be swapped, regardless of the size of its content, this also means however that within the 4kb page file, if this space is not fully utilised it will be wasted, a 4kb file can hold a 1kb file and be moved around as if it was a 4kb file. Also, if the number of pages grows large the page table can become quite large and despite eliminating external fragmentation, internal fragmentation within the pages themselves can still be an issue.

15a.

When RAM is running low, to avoid a crash or hang-ups, software and hardware can be used in conjunction to ‘create’ virtual memory, secondary storage, SSDs or HDD’s are addressed as though they are part of the systems primary memory, abstracted from the processes, the MMU has to translate the virtual address used by the program to a physical address, this abstraction however allows a process to view memory as ‘infinite’ in the sense there is always memory available to expand into, meaning no process will get killed due to the lack of available space.

15b.

Virtual memory allows the OS to allocate memory addresses to a process that they view as purely their own, allowing the process to grow as required without being immediately limited by physical memory available, especially useful in a multiprogram environment as new processes can be instantiated without either process hanging or being terminated.

As mapping is used for the processes it also eliminates the security risk of multiple processes trying to access/use a single address in memory, which could lead to corruption, security risks or system crashes.

16.

Much of the reasoning behind the stack growing downward originate from historical implementations of using positive offsets from the stack pointer. However, growing the stack and heap in opposite directions allows them to share the same space and grow accordingly. When space was a much stricter constraint, starting at the end and growing downward into space allowed expansion, if the stack grew upward toward a fixed limit, once that limit was reached it would have nowhere to grow to.

17a.

A Memory Management Unit (MMU) is a translation unit which as mentioned previously is capable of mapping virtual addresses from processes to the hardware addresses actually assigned in memory, it is typically located on the CPU but can be software based if not provided in the hardware, though this is less than optimal. The MMU uses a page table to store the 'maps' for the processes to allow for seeking out the correct physical address from the provided virtual address.

17b.

A system could use paging without a MMU theoretically, however it would not be able to use the paging table outlined above for converting virtual addresses to physical addresses as there is no software or hardware in place to map the two. To mitigate this the program would need to directly use hardware addresses, this leaves no room for errors as they could lead to system-wide errors with memory crashes as the system loses the memory protection provided by having an interim capable of mapping fragmented memory.

18a.

Process memory is divided into code (text) and data areas and segmented in memory, primarily due to the different permission requirements of the two for security reasons. Process code should have the ability to be read and executed to allow the program to execute, data however should contain variables which can be read and wrote to at run time but not executable.

18b.

Allowing process data to be wrote to and then executed means it would be possible for potentially malicious code to be inserted and executed, potentially compromising the system. Code is kept separate In a read only segment so that it does not accidentally have its contents altered leading to potential instability, some constant variables that will never change may be allowed here however.

19a.

Buffer overflows, when used maliciously overwrite the memory of a program to insert executable code in place of the intended code, this is typically done by inserting data into the stack larger than it can hold, 'overflowing' and overwriting the functions return address. The malicious insert can then replace the return pointer with a new pointer that returns to the actors malicious code, compromising the system as theoretically any code can now be executed, this is often exploited when a variables length has not been validated, and data of for example 5 characters is inserted into a memory allocation (a) of only 4 bytes, the data will overflow into the next variable (b)'s memory, overwriting what was initially stored here.

19b.

To mitigate this, checks should be put in place to validate data entered into the system. In the 19a example of data being copied of an invalid size, if the program was written in C and strcpy() was used, the data would be copied in with no checks, allowing an overflow to take place, maliciously or otherwise. Strlcpy() however requires an additional argument in addition to the source and destination variables, length. Rather than using if statements to validate the inputted data strlcpy() the function will only copy up to the given length, in the above example if 4 bytes is set as the max length, anything beyond that will not be copied, this removes the risk of a buffer overflow as the data can physically not exceed the specified memory size, it can however pose other risks such as data loss.

20.

In this example the C function printf() will be followed on a Linux system.

A user program will initially add the required arguments for the syscall to the stack to be passed forward to the kernel space from the user space. When the printf() function (which is contained in the C system library) reaches the stack an interrupt is executed (using int $0x80 on an x86 CPU), this calls the syscall handler and stores the system call number to be called (in this case, write() will have a defined syscall number) alongside arguments if appropriate. The syscall handler will then select the corresponding syscall using the number provided and point to the memory address defined in the handler to be executed, write() in this instance then takes the passed arguments, including a file descriptor telling it to print to the screen (or given I/O device), device drivers will interface this request and execute the necessary commands to produce the required output. If correctly executed the stored registers and address to user level will be followed to context switch to user space and the output will be reflected on screen, otherwise one of several defined errors will be shown.

21a.

The primary performance issue with syscalls is the need to create a kernel thread to execute the call, which in turn requires the kernel to be interrupted. When called it is necessary for the system to context switch, ‘saving’ the current status of the user program so that it can be safely returned to upon completion of the systems time in kernel space, at which point it will again need to context switch back to the user program.

21b.

For stability reasons and computational overhead reasons it would be good practice to only use system calls when absolutely required, using 1 printf() statement for example as opposed to 3 will reduce the number of system calls made proportional to the reduced number of printf() statements.

///End of Answer sheet.