Jayven Larsen

101260364

AJ Donald

101259149

Assignment 1

SYSC-4001 section A

Dr. Gabriel Wainer

Oct 3rd 2024

Q1

a)

i) The cards have the “$” used as an identifier/ differentiator for knowing the difference between for example a card full of data and a loader function call to load the FORTRAN code into memory. The different software components needed for this would be a job control language (JCL), accompanied with a job scheduler as well as a compiler. There is also the loader function, this would be when the $LOAD command is received. Finally, when the $RUN command is received, it will move the PC to the starting address of the program

ii) If in the middle of a data punch card if there was $LOAD, the cpu would call the loader function as it cannot differentiate the fact it's in a data or command card. Although this would happen on rare occasions which is why the “$” delimiter is used. Additionally, by providing the system with a new $LOAD card instruction, we would be overriding the compilation step in job sequencing . As a result, the LOAD may be performed but without having completely compiled the program, we would not be able to perform the $RUN portion of the sequence, since there would be no executable generated

b) Kernel mode is a specific mode that grants access to all system resources and instructions. The operating system's core components, including the kernel and device drivers, run in this mode. Now the user mode is quite a bit more limited in terms of the lower level side of things. The user mode has limited access to system resources and cannot interact with the os’s core components. Now the memory protection system is in place for the user mode for obvious reasons. This protection system ensures the user cannot access or modify the kernel’s dedicated memory space. This is done through firstly, memory segmentation. This where you segment different blocks of memory and those blocks all need specific rights to be accessed. There is also access control which coincides with memory segmentation since this control defines the specific rights needed for different memory blocks. It also enforces privilege levels using control registers that can be distinguished through user and kernel mode. In the case where user mode tries to access an non user mode designated block of memory, there is trap handling which throws a hardware exception.

c) There are quite a few privileged instructions used by the cpu during use. Firstly there is the interrupt control. These commands consist of clearing input flags and setting input flags. These will be used when a peripheral device (mouse, keyboard, switch) is being toggled or used and the interrupt flag gets set. Once the interrupt has been serviced, the clear interrupt flag instruction gets run which clears the flag and allows the CPU to resume its previous tasks. These 2 commands are privileged since if a user decides to clear an interrupt flag during a critical system interrupt, it could lead to system hangs/ crashes. Given that privileged instructions are only executed in kernel mode, another key instruction that can be categorized as privileged is clearing memory. Clearing memory plays a key role in resource management and security of the system. Preventing any memory leaks or accidental writes to memory is vital to system stability. As a result, the ability to clear memory must be controlled and restricted in use from the user to prevent overwriting key memory addresses such as the Operating System. Moreover, another privileged instruction would be accessing CPU control registers. These certainly must be restricted from the user and only accessible through the kernel since they have the ability to change key settings relating to the processor such as sequencing, control flow or task switching configurations. Finally, a key protected instruction is process management. While process management is a very broad term, it relates to resource allocation such as memory allocation, input management or file access for a certain task to execute. These play critical roles in efficiency and stability of the system. As such, process management must be abstracted from the user to prevent any issues relating to memory misuse, data corruption or even exposure to security vulnerabilities.

d) While the interrupt mechanism is dependent on the hardware architecture of choice, generally speaking an interrupt requires a few key components. An interrupt is widely defined as a signal that prompts the CPU to stop the current task at hand, then deviate its resources to handle the interrupt request. Firstly, an external signal coming from some hardware is necessary, in modern scenarios an external device such as a keyboard sends the interrupt signal to the CPU using the interrupt controller. The interrupt controller helps the CPU prioritize interrupts that come from different sources, such as I/O devices. Next, once again utilizing hardware the CPU must consult the interrupt vector table which contains the addresses of interrupt service routines (ISR) for different interrupts. In terms of proceeding with an ISR, the CPU must save the current state of the process data such as the contents of key registers such as the Program Counter (PC, R15 in ARM architecture) and save it on the stack, this can be done using a device driver. Moreover, depending on the hardware of choice, a Link Register (LR, R14) which is used to store the return address of the subroutine. We then ‘jump’ to the ISR address to handle the interrupt. Moving on, software executes the ISR meaning, the ISR runs when the interrupt is being addressed. In some scenarios, interrupt could be reading data from a peripheral and then loading or storing that data in a local buffer. Additionally, upon completion of the interrupt, a hardware component like a device controller informs the device driver. The device driver will give control to other parts of the operating system. Finally, the CPU will ‘POP’ back the registers that were stored on the stack and will continue the remainder of the program execution.

e) While protected identifiers are a concept heavily used in modern technology and software engineering, it relates deeply in computer architecture and has found its way into operating systems. In fact, a system call can be interpreted as a form of this idea. System calls are a controlled mechanism designed for user-level processes to request a service from the operating system’s kernel. To elaborate, these requests are necessary to separate and make distinction between regular and privileged instructions away from users. Some examples of requests a user could make that would require the kernel are file operations or memory allocation. System calls are designed to allow the program to perform actions which are protected and require a higher privilege level than users are exposed to naturally. To draw a connection between system calls and interrupts, we can view system calls as a software-triggered interrupt that allows for transfer of control between user and kernel mode. On the hardware side, when a program makes a system call, whether it be by calling functions such as open(), read() or write(), a special instruction is required, in the ARM architecture, SVC (supervisory call) is needed. SVC as an instruction is what triggers the software interrupt, which is initiated by a program rather than an external device. Then, upon the generation of the software interrupt, the CPU’s interrupt vector table is used as an appropriate handler.

Next, the context switching occurs, where the CPU switches from user mode to kernel mode, giving the necessary privileges to perform kernel-level tasks. Lastly, the system call handler will execute the call, which will access protected resources such as memory, files or devices. Upon completion, the CPU will revert back to user processing, swapping back to user mode and program execution.

f) A time sharing operating system allows a multitude of users to interact with a single computer system at the same time, all while just using a single CPU. This is achieved by rapidly switching between users and tasks giving a type of illusion that all the processes are running concurrently. Here are a couple of the functions needed for the time sharing operating system. Firstly there are Scheduled algorithms, these algorithms will determine the order in which processes are given CPU time. It ensures an equal share of the CPU for all users. Secondly there is context switching, which is when the system switches from one process to another. It saves the current state of the running process. These would include registers, memory pointers etc. Then it proceeds to load the context of the next process. This enables paused processes without the loss of data or progress. Finally there is the multi-tasking factor. Although this time sharing system only runs on one CPU, the very rapid switching between processes makes it appear as they are executing simultaneously. This works in collaboration with the scheduled algorithms.

g)

1. For a process to move from the ready state (waiting to be executed) to the end state (process termination), several events occur:
   1. Process completion: This is when the process receives its last instruction and gives a single saying it has entered the final lifecycle. In the OS, there is a Kernel reaction when a process completes its execution. It triggers a termination routine which performs cleanup operations. This includes deallocating memory, closing file descriptors and removing the process from the systems process table. Finally after the routine is completed the process is moved to its “END” state.
   2. Kill or abort signal: This happens when a process might be forcibly terminated due to user intervention (could be via kill command or system error such as a segmentation fault for example). The kernel reaction in this case will be to handle this via an interrupt handler. It will receive a signal to terminate the process. Similar to normal termination, it will go through the termination routine and go through the steps previously mentioned.

h)

To start, we have a terminal which is 40 char’s wide and 20 lines of height

* 40 x 20 = 800 characters

The time to display 1 character is 1 millisecond and the interrupt processing time is 50 microseconds.

Secondly, we have a high resolution screen which is 1000 pixels wide by 400 pixels in height.

* 1000 x 400 = 400000 pixels total

The time to display 1 pixel is 1 microsecond and interrupt processing time is 50 microseconds.

1. **The Terminal** (800 characters total). Since each character will trigger an interrupt, there are a total of 800 interrupts. If each character takes around 1 millisecond to display then the total time to display the 800 char’s will be 800 ms. The time to process an interrupt is 50 ms, so for 800 characters the interrupt time will be 40ms (800×50µs=40,000µs). So in total, the time to display all characters on the screen including interrupt processing time will be 840 ms.
2. **The high resolution screen** is a total of 400,000 pixels meaning a much larger amount of interrupts of 400,000. Each pixel takes 1µs to display meaning it takes 400 ms to display the whole screen. Now, each interrupt takes 50µs to process. So for 400,000 interrupts it will take a staggering 20 seconds (400,000×50µs=20,000,000µs=20,000ms=20 seconds.) And in total the time to display the entire high resolution screen will be a whopping 20.4 seconds.

With the results from the high resolution screen, there is evidently a problem. A significant performance issue comes to life because of the very long interrupt processing time. With the interrupts taking 20 seconds of processing time, this screen would be very impractical and almost impossible to use for real time display purposes. To combat this long processing time, a use of batch processing could be viable. Instead of processing the interrupts one at a time for every pixel, implementing some system buffers would help reduce processing time. This will make it so while the CPU is processing the longer interrupts, the buffer will work in parallel with the CPU to process them. These are referred to or known as PU’s which are smaller and much cheaper CPU’s. Once these PU’s have completed their processing, they will signal the CPU to hand off the data. This will reduce the frequency of interrupts and will increase the throughput since the CPU will be working at all times. For example, if there was a buffer line of 100 pixels, The number of interrupts for the high resolution screen reduces to 4000 interrupts. (400,000 pixels/ 100 pixels/interrupt = 4000 interrupts) This means the processing time is now 200 ms (4000 x 50µs = 200,000µs = 200 ms). Knowing the time to display each pixel is 400ms, the total time to display the screen with interrupt processing time is reduced to 600 ms. With even bigger buffer lines, (e.g 1000 pixels) the total processing time can be reduced even further down to 420 ms.