CSCE 313
Quiz 1
Jeffrey Xu
08/27/20

1. The executable image of a program must be loaded into the main memory first before executing

True. The PC register points to the next instruction in main memory and if the program isn't loaded into the main memory then the CPU doesn't know what instruction to run next.

2. An Operating System (OS) does not trust application programs because they can be either buggy or malicious

True. If the OS gave application programs access to priviledged instructions, then the programs could go and change the OS code and consequently crash the OS and lose data.

3. There was no concept of OS in first generation computers

True. The first generation of computers used card readers and programmers had to manually code and compile their programs.

4. The PC register of a CPU points to the next instruction to execute in the main memory

True. The PC register determines the program flow that the CPU uses. Without the PC, the CPU wouldn't know what instruction to execute.

5. Second generation computers still executed programs in a sequential/batch manner

True. The second generation computers automated the process of loading, translating and executing instructions and allowed for multiple programs to be run but it still performed these tasks in a sequential manner.

6. Time sharing computers gave a fixed time quantum to each program

True. Time sharing forced the CPU to rotate between programs (programs kicked out whenever the time quantum is up).

7. An OS resides in-between the hardware and application programs

True. The OS manages resources for the application layer and provides abstraction of hardware resources.

8. The primary goal of OS is to make application programming convenient

True. The goal of the OS is to provide virtualization to the programmer and program. This allows for each program to think it has all of the CPU resources and allows the program to not worry about physical memory management.

9. Context switching does not contribute much to the OS overhead

False. If we are doing to many context switches, then that would contribute a lot to the OS overhead since it has to handle the interrupt everytime you are switching.

10. Multiprogramming cannot work without Direct Memory Access (DMA) mechanism

True. If multiprogramming didn't have DMA, then it couldn't load and kick out programs whenever a program required IO or CPU time.

11. Interrupts are necessary for asynchronous event handling in a CPU

True. The way asynchronous pipelining works is by kicking out programs and loading them back in. Without interrupts, we wouldn't be able to kick out programs from the CPU.

12. A program can be kicked out of a CPU when it requests I/O operation, or when another Interrupt occurs

True. In multiprogramming, whenever an interrupt occurs for a program, it can kick that program out of the CPU and handle the IO or another interrupt while another program uses the CPU.

13. A program error can kick a program out of CPU

True. If a program error causes a program to crash, then the program can be kicked out of the CPU and another program can use the CPU.

14. Interrupts are necessary to bring a program back to CPU if it was previously kicked out

True. Interrupts allow for programs to be kicked out of the CPU. If a program was previously kicked out and another program takes its place, then if the previously kicked out program needs the CPU, an interrupt must be sent to the program that is currently using the CPU.

15. The "Illusionist" role of the CPU allows a programmer write programs that are agnostic of other programs running in the system

True. Each program has its own program frame allowing it to run in the illusion that it has full access to the CPU.

16. Modern operating systems come with many utility services that are analogous to the "Glue" role of the OS

True. The OS connects the application layer and the hardware layer.

17. Networking service is not a core OS part, rather a common service included with most OS

True. The OS really only concerns itself with resource management and memory allocation, not network services but most OS do provide functionality for networks.

18. Resource allocation and Isolation are not part of the core OS, rather common services included with OS

False. The main purpose of the OS is to manage allocation and isolation of resources.

19. Efficiency is the secondary goal of an OS

False. Efficiency is the main goal of an OS. If an OS was slow, it would not be able to compete with other OS making it useless.

20. After handling a fault successfully, the CPU goes (when it does go back) to the instruction immediately after the faulting one

True. Faults are exceptions that arre unintentional but ultimately recoverable. If the OS is able to recover from the fault, control will be given back to the program and it will execute the next instruction normally.

21. Interrupts are asynchronous events

True. There are synchronous and asynchronous exceptions and asynchronous exceptions are also known as interrupts since they do not depend on sequential states of the program but rather something external.

22. Memory limit protection (within a private address space using base and bound) is implemented in the hardware instead of software

True.

23. Memory limit protection checks are only performed in the User mode

True. The CPU checks all memory access in user mode.

24. Divide by 0 is an example of a fault

False. A divide-by-0 zero is an example of an abort since it is unrecoverable. Faults can be recovered from it the OS is able to properly handle it.

25. Multiprogramming can be effective even with one single-core CPU

True. Multiprogramming allows multiple programs to be run asynchronously instead of sequentially which can dramatically increase throughput since when one program needs the IO or doesn't need the CPU, the OS can swap that program out of the CPU and put another program that needs the CPU to run. This allows the CPU to not be idle as much as it isn't waiting for programs to run sequentially and can run programs out of order.

26. [10 pts] Define multiprogramming. How is this better than sequential program execution?

Multiprogramming allows for asynchronous pipelining. This allows programs to be run out of order instead of the order than they come in (sequentially). Multiprogramming is much better than sequential program execution because it makes the CPU as busy as possible resulting in a higher throughput.

27. [10 pts] Define time-sharing. Can you combine time-sharing with multiprogramming?

Time-sharing allows each program some fixed amount of time with thee CPU. The time limit each program has is called a time-quantum. After the time-quantum is reached, the current program saves its state and gets kicked out while the next program gets loaded and ran. Time-sharing can be implemented with multiprogramming.

28. [10 pts] Say you are running a program along with many other programs in a modern computer. For some reason, your program runs into a deadlock and never comes out of that. How does the OS deal with such deadlock? How about infinite loops? How does the OS detect, if at all, such cases?

This problem can be solved with time-sharing. If time-sharing was never implemented, then if an infinite loop or deadlock occurs, then that program would run forever and no operation can stop the program. However, since time-sharing only allows each program a limited amount of time with the CPU, infinite loops and deadlocks can be broken when the time-quantum is up for that specific program. A deadlock can be detected by forming a resource-allocation graph and if there exists a cycle within the graph then there will be a deadlock. To terminate a deadlock, the OS can kill the processes in the deadlock and continue normal program function. For infinite loops, since not all infinite loops can be detected, the best the OS can do is provide resources in the case an infinite loop occurs. The best an OS can do is terminate the program after some specific amount of time.

29. [5 pts] Which of the following are privileged operations allowed only in Kernel mode?

a) Modifying the page table entries:

This is a privileged instruction since it involves modifying data in memory. If the user programs were allowed access to modify memory, it could harmfully delete or add memory in places it shouldn't be allowed to.

b) Disabling and Enabling Interrupts:

Disabling and enabling interrupts is an instruction that can only be performed in kernel mode. Therefore it is a privileged instruction.

c) Using the "trap" instruction:

A trap instruction is just an instruction to switch between user and kernel mode. It can be called in user mode, therefore it isn't a privileged instruction.

d) Handling an Interrupt:

Handling an interrupt requires running code in the interrupt handler which is stored in the kernel. Therefore only the kernel can handle an interrupt making interrupt handling a privileged instruction.

e) Clearing the Interrupt Flag:

Clearing the interrupt flag has the same protection as disabling and enabling interrupts as both deal with interrupt flags. Therefore clearing the interrupt flag is a kernel-mode only instruction making it a privileged instruction.

30. [25 pts] In a single CPU single core system, schedule the following jobs to take the full advantage of multiprogramming. The following table shows how the jobs would look like if they ran in isolation. [Use the attached pages from W. Stallings book to solve this problem]

	Job 1	Job 2	Job 3
Type of Job	Full CPU	Only I/O	Only I/O
Duration	5 min	15 min	10 min
Memory Required	50 MB	100 MB	75 MB
Needs Disk?	No	No	Yes
Needs terminal?	No	Yes	No

a. What is the total time of completion for all jobs in sequential and multi-programmed model?

The time to run jobs sequentially is the sum of the time it takes to run each job since only one job can be run at one time. Therefore the total time for sequentially running of all the jobs is $5 + 10 + 15 = 30$ min.

In multi-programming, the time it takes to run a sequence of jobs is the job that takes the most time since multiprogramming allows the CPU to run multiple jobs asynchronously. Therefore the total time to run the three jobs in multi-programming would be $max(5, 10, 15) = 15$ min.

b. Fill out the multiprogramming column in the following table (i.e., when the jobs are scheduled in multiprogramming). Assume that the system's physical memory is 256 MB.

The total running time for multi-programming is 15 min, and since only job 1 is using the CPU, the average processor use is $5/15 = 33.33\%$. The average memory for multi-programming is calculated the same way as for sequential except the total run time is decreased by a factor of 2. This means that for multi-programming the memory usage would be double; $(5 * 50 + 15 * 100 + 15 * 75)/(15 * 256) = 65.1\%$. The average disk use is 10 min$/15$ min$= 66.67\%$. The average terminal use is 15 min $/15$ min$= 100\%$.

Average Resource Use	Sequential	Multiprogramming
Processor	5/30 = 16.67%	33.33%
Memory	32.55%	65.1%
Disk	33.33%	66.67%
Terminal	50%	100%

31.a. [10 pts] The following are steps in a "sequential" Interrupt handling. What changes would you make in the steps below so that "nested" Interrupts can be handled?

Hardware does the following:

Mask further interrupts
Change mode to Kernel
Copy PC, SP, EFLAGS to the Kernel Interrupt Stack(KIS)
Change SP: to the KIS (above the stored PC, SP, EFLAGS)
Change PC: Invoke the interrupt handler by vectoring through the Interrupt Vector Table (i.e., overwrite PC with handler PC)
Interrupt Vector Table (i.e., overwrite PC with the handler PC)

Software(i.e., the handler code) does the following:

Stores the rest of the general-purpose registers being used by the interrupt process
Does the rest of the interrupt handling operation?

In order for us to be able to handle nested interrupts, all interrupts must be prioritized in order for us to know which interrupt to deal with if a nested interrupt occurs. We also cannot mask other interrupts as that would make our implementation sequential. Therefore, before we start to copy all the memory to the KIS, we need to assign a priority to the interrupt and put all of the

current interrupts into a priority queue. When the Interrupt Handler is called, we then handle the interrupts in teh priority queue in the given order that the priority queue assigns them to. This way if we are handling an interrupt and another interrupt occurs, we simply put that interrupt into the priority queue and pull the next element from the queue (if a low-priority interrupt is interrupted by a high-priority interrupt, the high-priority interrupt will run and if a high-priority interrupt is interrupted by a low-priority interrupt, the high-priority interrupt will be handled first; interrupts with the same priority will be handled by time order).

b. [3 pts]: Can you interchange steps 2 and 3? Why or why not?

Steps 2 and 3 can be interchanged. This is because step 3 doesn't depends on the completion of step 2. We can copy the memory values for PC, SP and the flags into the KIS despite the mode because the copying is done in the hardware.

c. [2 pts]: Can we interchange step 1 with step 2? Why or why not?

Step 1 and step 2 in the hardware process should be able to be interchanged, although it is kind of redundant to switch them. Masking the interrupt can be performed in any mode, user or kernel. Therefore if we first switch to kernel mode, we can still mask further interrupts, though typically once an interrupt has happened, we probably want to mask further interrupts as soon as possible to prevent too much overhead.