CSCE 313 QUIZ 1 Spring 2021

Student Name: Arthur Chen
UIN: 327003368

Student Score / 100

True/False Questions **[25 pts]**

1. The executable image of a program must be loaded into the main memory first before executing T
2. An Operating System (OS) does not trust application programs because they can be either buggy or malicious T
3. There was no concept of OS in first generation computers T
4. The PC register of a CPU points to the next instruction to execute in the main memory T
5. Second generation computers still executed programs in a sequential/batch manner T
6. Time sharing computers gave a fixed time quantum to each program T
7. An OS resides in-between the hardware and application programs T
8. The primary goal of OS is to make application programming convenient F
9. Context switching does not contribute much to the OS overhead F
10. Multiprogramming cannot work without Direct Memory Access (DMA) mechanism T
11. Interrupts are necessary for asynchronous event handling in a CPU T
12. A program can be kicked out of a CPU when it requests I/O operation, or when another Interrupt occurs T
13. A program error can kick a program out of CPU T
14. Interrupts are necessary to bring a program back to CPU if it was previously kicked out T
15. The “Illusionist” role of the OS allows a programmer write programs that are agnostic of other programs running in the system T
16. Modern operating systems come with many utility services that are analogous to the “Glue” role of the OS T
17. Networking service is not a core OS part, rather a common service included with most OS T
18. Resource allocation and Isolation are not part of the core OS, rather common services included with OS T
19. Efficiency is the secondary goal of an OS T
20. After handling a fault successfully, the CPU goes (when it does go back) to the instruction immediately after the faulting one F
21. Interrupts are asynchronous events T
22. Memory limit protection (within a private address space using base and bound) is implemented in the hardware instead of software T
23. Memory limit protection checks are only performed in the User mode T
24. Divide by 0 is an example of a fault F
25. Multiprogramming can be effective even with one single-core CPU T

Short Questions

26. **[5 pts]** Define multiprogramming. How is this better than sequential program execution?
Multiprogramming is the solution introduced in the third generation of computer system, which have several programs ready to execute and keep several programs in memory. For sequential program execution, when a job requests for I/O, CPU has to be idle, which is a waste of compute power of CPU. On the other hand, multiprogramming allows other jobs to be executed during the idle time of CPU, which increases the overall efficiency. Note that multiprogramming requires direct memory access to achieve.

27. **[5 pts]** Define time-sharing. Can you combine time-sharing with multiprogramming?
Job scheduling was an issue that emerges from the innovation of multiprogramming, and time-sharing makes the process scheduling much more sophisticated. Time sharing assign each job a specific time that a job can own the CPU. When the assign time quant is up, the program will be kicked out of the CPU and wait behind other programs in line. Fortunately, time sharing can combine with multiprogramming. However, we need a service that can bring the program back to the CPU when CPU is ready, which is called interrupts. Interrupts helps the OS to pause the current program and resumes it later.

28. **[5 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?
In short, halting problem, which is proved by Alan Turing in 1936, indicates that OS does not have the ability to determine whether a program will end eventually or continue executing forever because OS does not have the ability to analyzes the internal logics in our code. There are several famous ways of how OS handles a deadlock. One of the methods is process termination. The time quant of each processes severs as a watchdog for the OS to handle them. Since the deadlock does not end when the time quant is used up, the OS will terminate the deadlock.

Reference:
Halting Problem, available at: https://en.wikipedia.org/wiki/Halting_problem

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

a) Modifying the page table entries
b) Disabling and Enabling Interrupts
c) Using the "trap" instruction
d) Handling an Interrupt
e) Clearing the Interrupt Flag

a, b, d, e

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]

	JOB1	JOB2	JOB3

Type of job	Full CPU	Only I/O	Only I/O
Duration	5 min	15 min	10 min
Memory required	50MB	100MB	75MB
Needs disk?	No	No	Yes
Needs terminal?	No	Yes	No

a. What is the total time of completion for all jobs in a sequential and multi-programmed model?
   Sequential: 5+15+10=30mins
   Multiprogramming: according to the utilization histograms in the attached page of W. Stallings, three programs can be executed at the same time. Thus the time needed is max(5,15,10) = 15 mins.

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 256MB.

Average Resource Use	Sequential	Multiprogramming
Processor	5/30 = 16.67%	5min/15min=33.33%
Memory	32.55%	5min*(50MB+100MB +75MB)+5min*(100MB+75MB)+5min*(75MB) / (15minx256MB) = 61.8489%
Disk	33.33%	10mins/15mins=66.66%
Terminal	50%	15min/15min=100%

Memory usage is computed as follows: (5minx50MB + 15minx100MB + 10minx75MB) / (30minx256MB) = 32.55%
Other resources are fully utilized during the time they are utilized. So, you compute utilization only based on the duration they are used.

31. **[15 pts]** The following are steps in a "sequential" Interrupt handling. These steps are in the user-to-kernel direction, while the steps in the opposite direction are simply reversed.

Hardware does the following:

---

1. Mask further interrupts
2. Change mode to Kernel
3. Copy PC, SP, EFLAGS to the Kernel Interrupt Stack (KIS)
4. Change SP: to the KIS (above the stored PC, SP, EFLAGS)
5. Change PC: Invoke the interrupt handler

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

---

1. Stores the rest of the general-purpose registers being used by the interrupted process
2. Performs the rest of the interrupt handling operation

Now, answer the following questions:

(a) [7 pts] What changes would you make in the steps below so that "nested" Interrupts can be handled?

Two steps are required to handle a nested interrupt. Nested interrupts are interrupts that happen with other interrupts. Therefore, we need to check for priority among interrupts. Adding step 0(before step 1 of HW): check if the current running interrupt has higher or equal priority, if yes, do nothing and return, if no, continue to step 1. Adding step 4.5(between step 4 and 5 of HW): Right before we invoke the interrupt handler, we need to enable interrupt again so that nested interrupt will work accordingly.

(b) [3 pts] For sequential interrupt handling, can you interchange steps 2 and 3? Explain.
No, because Kernal Interrupt Stack cannot be modified while not in Kernel mode.

(c) [2 pts] For sequential interrupt handling, can we interchange step 1 and 2? Explain.
Doable, however, not encouraged. What if there is another interrupt after switch to Kernel mode?

(d) [3 pts] For sequential interrupt handling, can we interchange step 1 and 3? Explain.
No, for the same reason as problem c, if there is another interrupt after we copied data to KIS, then KIS will have wrong data if there is further interrupt.

32. [15 pts] Assuming each page is 4KB, how many page faults or faults of any kind the following program will generate? Explain your answer.

```
int size = 4 * 1024 * 1024; // size = 4 megabytes
char * memory = new char [size]; // allocate 4 mega bytes
__int64_t * a = (__int64_t *) memory;//__int64_t is 64-bit or 8-byte integer
for (int i=0; i< (100 * 1024) ; i++)
    a [i] = 0;
```

unlike data type int, __int64_t is 8 bytes each. Therefore, by the end of the for loop, the code will try to access 8*100*1024 bytes, which is much greater than page size, only 4 KB, thus leads to 8*100*1024 / 4 KB = 200 page faults. Since 8*100*1024 = 800KB is still within 4MB of int size, no seg fault will occur.