ECSE 427 Written Assignment 1

By:
Priscilla Ip - 260 665 764
Ezz Aboulezz - 260 677 463
Saifullah ElSayed - 260 733 168

Question 1:
Interrupts can occur on devices that do not include an OS. An interrupt service vector points out the interrupt and the CPU stops the currently executing program and carries out the task immediately (I/O interrupts). A system call involves a switch from user mode to kernel mode, which is not necessary in interrupts. The system call allows programs to request services from the kernel such as accessing devices and memory which would not be possible without switching to kernel mode.

A similarity between the two is that in some operating systems, interrupt service vectors and the interrupt handler as a whole is managed by the kernel.

Question 2:
I/O device access is usually restricted due to admin/organization policies. These restrictions can apply to some users using the workstation. Access to a scanner or a printer can be restricted to certain groups (ex:guests) and the organization admin needs to enforce those policies in order to deny user programs access to the devices.

Question 3:
From the following instructions, all of them should only be allowed to execute in kernel mode except instruction b. (Read the time-of-day clock).

Question 4:
A trap instruction is a procedure call that synchronously transfers the control. It is a software interrupt generated by the user program or by an error when the operating system is needed by it to perform the system calls or an operation. In operating systems, the instruction is used to switch from user mode to kernel mode of the system.

Question 5:
Two main functions provided by memory management essential for multiprogramming are memory locking and reserving/freeing physical and virtual memory.

Memory locking is one way to ensure that a process stays in main memory and is exempt from paging. In a realtime environment, a system must be able to guarantee that it will lock a process in memory to reduce latency.

Virtual memory is a technique that allows for the execution of processes that are not completely in memory.

Question 6:
Microkernel OS would be more reliable for the following reasons:
1. The micro-kernel has less responsibilities; It can be developed and verified with more reliance than with larger monolithic systems.
2. If there is a bug in one of the user processes, the system will not crash since the operating system is split into modules which all run as user processes except for the micro-kernel. In summary, the system won't crash unless the microkernel crashes.

However, microkernels can be more complex due to the need to construct complex communication interfaces. This can make the microkernel less reliable and favor the monolithic OS in some cases.

Question 7
If a single thread is used, the best request rate occurs when the request is serviced from the cache and the worst request rate occurs when the request is serviced from the disk. The best would take a 200ms and so the request rate would be 5 requests/second. The worst would take 1s and so the request rate would be 1 request/second. In the case of multi-threading, the best case would be to assign back to back cache access to a second thread while the first thread is servicing the disk access request. In 1s, five requests could be serviced (1 disk access request plus 4 cache requests). This would result in a rate of 5 requests/second.

Question 8
There are 15 processes that run the *run_morecompute()* function.

Iteration	Processes at start	Processes running *run_morecompute()*
1	1	1
2	2	2
3	4	4
4	8	8
Total		15

Question 9
The contents of tempt2.txt would be:

Message from A
Message from B

or

Message from B
Message from A

Message from B *Message from B*

This is because the parent process would add the “Message from B” text into the file and the child would add both the “Message from A” and “Message from B” since the latter print statement isn’t in an else condition.

To ensure that the “Message from A” text does not appear in the file, the code should be modified by inserting a close(1) before the print statement for “Message from A”. This solution would also remove the second “Message from B” from the file as well.

To ensure that “Message from A” shows up on the standard output, the code would need a few modifications. First, the standard output should be copied before the *close(1);* by inserting a *stdout = dup(1)*. Then, before the *printf(“Message from A”);* line, insert a *close(1);* and *dup(stdout);* to set 1 back to the standard output. This solution would also show the second “Message from B” text on the standard output as well.

Question 10

The speedup from running the application on the four-core machine is 800s/290s = 80/29.

To calculate expected runtimes with a varying number of cores, we must calculate the speedup and therefore the serial portion (S) of the application

To calculate S:
Parallel portion runtime for 4 cores = (800s - 290s)/(4-1) = 170s
S = (290s - 170s)/800s = 0.15 or 15%

To verify, we calculate speedup for 4 cores:
Speedup = 1/(S + (1-S)/N) = 1/(0.15 + 0.85/4) = 80/29

Number of Cores	Speedup	Expected Runtime
2	1/(0.15 + 0.85/2) = 40/23	800s / (40/23) = 460s
4	1/(0.15 + 0.85/4) = 80/29	800s / (80/29) = 290s
8	1/(0.15 + 0.85/8) = 160/41	800s / (160/41) = 205s
16	1/(0.15 + 0.85/16) = 64/13	800s / (64/13) = 162.5s

Question 11:

```
fd = open(file)
while (! end of stdin) {
  buffer = read(stdin)
  write(stdout, buffer)
  write(fd, buffer)
}
close(fd)
```

Question 12:

The exec() system call may fail if one of the following conditions is met:

1. When interpreting the specified path name, too many symbolic are encountered.
2. The maximum recursion limits is reached.
3. The new command or process file does not exist.
4. A process needs more virtual memory than allowed by the limit.
5. The absolute pathname is not valid.
6. There is an input or an output error when reading from the file.
7. The size of the arguments list that is used by the process or command exceeded the limit.
8. The new command or process file do not have the correct permissions.

The fork() system call may fail if one of the following conditions is met:

1. No sufficient memory in the system for the fork process to allocate.
2. The limit on PIDs is reached.
3. The limit for the number of threads and processes for the kernel is reached.
4. The fork process is not supported on the current platform.