Lucas Bellido - 260804057
Kamy Moussavi - 260807441
Paul Attara - 260807280

2019-10-6
ECSE 427

Written Assignment #1

1. Explain the key differences between the operations performed in system call processing and interrupt servicing. What are the similarities between the two?

System calls are usually accessed in programs via a high level programming language. They are in fact pre-written segments of code, which are made available to the programmers by the OS, and are used to perform the most basic functions (such as read from I/O, write to I/O, etc…) In consequence, the programmer doesn't need to know how to implement these functions, only how to call them. When the subroutine is called from a program, the respective segment of instructions is fetched from a system call interface, which executes the code in the kernel and on completion, returns to the user application.

At the root of interrupt servicing are device drivers, these are computer programs which are able to control different types of input/output devices attached to a computer. Linux, for example is an OS which comes with built-in drivers for Hardware. When an I/O device fires an interrupt, the driver code on how to handle the device is taken from the kernel space and put onto the CPU, halting any existing program which is currently being executed.

The key similarities between both originate from the fact that drivers and system calls are both predefined segments of Instructions which reside in the kernel space, in both cases the OS is responsible for fetching the instructions and putting them on the CPU for execution

On the other hand, the main difference between both comes from what "calls" the segment of instructions. For example, interrupts will be fired from I/O devices, whereas system calls will be fired from programs and applications.

2. Instructions related to accessing I/O devices are typically privileged instructions, that is, they can be executed in kernel mode but not in user mode. Give a reason why these instructions are privileged.

First, privileged instructions are instructions which can only be executed by the operating system in a Kernel Mode. They are used in order to ensure protection and security of the system and to ensure operations are performed correctly. In general, not at users may be allowed to access I/O devices. In general users will have different permissions for the use of different I/O devices. These permissions are specified and implemented in kernel mode so that user applications can not change or modify them.

3. Which of the following instructions should be allowed only in kernel mode?

a. Disable all interrupts
b. Read the time-of-day clock
c. Set the time-of-day clock
d. Change the memory map

A, C, D

4. What is a trap instruction? Explain its use in operating systems.

A trap instruction switches the CPU mode from user to kernel mode. This allows a user to invoke kernel functions.

5. Multi-programming enables an operating system to run multiple programs concurrently. What are the two important functions provided by memory management that are essential for multiprogramming?

Job scheduling and Message passing (IPC).
This allows multiple programs to run at the same time (or at least give the illusion that they are) without overriding each other memory sections.

6. Provide reasons why micro-kernel OS would be more reliable than a monolithic OS. Provide reasons why monolithic OS would be more reliable than a micro-kernel OS.
Advantages of micro-kernel:

- One service failing does not lead to an entire system failure because most of the system's services are outside the kernel space.

Advantages of monolithic:

- Faster execution due to it being a single process in kernel space.
- Less error prone, i.e. less interprocess communication between services tends towards less errors.

7. Consider a web server. Each incoming request is processed as follows: input-side processing (done on all incoming request) – 100ms, disk access – 900 ms, cache access – 100ms. The input side processing determines whether a request can be served by the cache or disk access is required. Assume that the web server has a single disk that serves the requests one after the other. The cache will hold popular requests after warming up. If the web server uses a single thread (kernel level), what are the best and worst request rates achievable by the server? Suppose we use two threads in the web server, what is the best request rate we could achieve?

Single Thread **(Best case)** - Cache access(100ms) + input-side processing(100ms) = **200ms**

Single Thread **(Worst case):** Disk access(900ms) + input-side processing(100ms) = **1000ms**
Double Threads **(Best case)**: parallel cache access and input processing (100ms) = **100ms**

8. Consider the following C code fragment. How many processes run the run_morecompute() function?

```
for (i = 0; i < 4; i++)
{ pid = fork();
if (pid == 0)
        run_compute(i);
}
void run_compute(int i)
{
int cpid = fork();
if (cpid == 0)
        run_morecompute();
}
```

There are 40 processes that run the function.

9. Consider the following code fragment.

```
close(1);
fd = open("temp2.txt", …);
if (fork() == 0)
printf("Message from A\n");
printf("Message from B\n");
```

What will be the contents of the temp2.txt file after executing this fragment?

Message from B
Message from A
Message from B

OR

Message from A
Message from B
Message from B

Let's say you don't want "Message from A" in your temp2.txt file. What modifications would you do to the above code fragment?

Close stdout in the child process, **close(1)**, before printing the message to the file in order to close the file

What modifications could you do to the code fragment to ensure that the output of "Message from A" shows up on the standard output? Note: no need to create a working program. Your modification should be valid in UNIX/Linux.

```
if (fork() == 0)
        printf("Message from A\n");
close(1);
fd = open("temp2.txt", …);
printf("Message from B\n");
```

10. An application took 800s to run in a single core machine. It took 290s on a four-core machine. What is the speedup you got when running the application on the four-core machine? What the expected runtimes in machines with 2, 8, 16 cores? What portion of the application is actually parallel?

Speedup obtained when running on four-core machine
= 800 seconds/290 seconds
= 2.76 times faster

Expected runtime with 2 cores
Amdahl's Law is used here:
N = 2 and S = 0.15
= 1/(0.15 + (1-0.15)/2)
= 1.74 times faster
= 460 Seconds

Expected runtime with 8 cores
Using same formula with N = 8 and S = 0.15
= 3.9 times faster
= 205 Seconds

Expected runtime with 16 cores
Using same formula with N = 16 and S = 0.15
= 4.92 times faster
= 163 Seconds

Serial portion
Using same formula and solving here for s

1/(S+ ((1-S)/4)) = 2.758
S= 0.15
15% represents serial portion

Parallel Portion
This means that parallel portion is 85%.

11. Operating systems have a tee command that allows a copy of the redirected pipe contents to be logged into a file. That is, you can get a copy of the output passed by a process to the other process logged in a file. Briefly explain how you would implement (give a high level pseudo code) the tee command and how it would work with the pipe redirection.

The tee command enables the user to redirect the output to a file as well as stdout.
Typically, a series of commands follow a linear pipeline. This would redirect the output of a command to the input of another command. To write the output of a command to a file, redirection is needed. Multiple pipes require multiple redirection operations. Multiple redirect operations would at some point prevent piping to another command.

The solution to this problem is through the use of the tee command. The tee command can read stdin and can write to one/multiple files and stdout by being introduced in the pipeline.

Pseudocode

```
// Read input command
In ←getline()
// Store stdout in out
out ← stdout

// Loop through files to output into copies out1...
For File fl: files :
    out1 ← out
    // open file
    fl_content ← fopen(fl)
    // write out to file
    print out1

// Display output in stdout
print out
```

12. When would an exec() system call or its variation that you might have used in the shell program fail? When would a fork() system call fail?

The exec system call runs an executable file in the context of an already existing process, replacing the previous executable. So, it returns a process image. As a result, the callee does not return anything to the caller program. Anything returned indicates that an error occurred, caused by an invalid path passed in, or insufficient memory to run the process image. It can also be caused by an argument list that is too long, exceeding the allowed amount of arguments. It can also be caused by a locking violation on the file to be accessed

A fork system call can fail if there is a limit that was set for the amount of processes that can run at a time or if there is not enough swap space for a new process.