COMP 346 – ASSIGNMENT 1 THEORY PART

SUBMITTED BY ANTOINE ASSAL 40022745

Question #1

I. **What is an operating system? What are the main purposes of an operating system?**
A system software that manages the computer's hardware, it acts like a middleman between the user and the hardware. It is the lowest level software (closest to physical hardware) and is considered as the resource manager of the computer.

II. **Define the essential properties of the following types of operating systems:**
- **Batch:** This type of OS keeps multiple processes in memory. By organizing similar jobs into batches to be processed at a time, and always having something for the CPU to execute (keeping it busy with buffers, spooling and multi-programming), it increases utilization of the CPU. So instead of waiting on additional X time to finish a process, the OS tells the CPU to switch to another process. Which keeps the user satisfied because typically they want to run multiple programs at the same time.
- **Time sharing:** This type supports interactive users because it ensures that the response times are fast enough. It uses $multiprogramming$ and $CPU\ scheduling$ to economically stay busy. Instead of having a spool like $\boldsymbol{Batch}$, each program is loaded into memory and executed, there is no idle time because the CPU rapidly switches between processes.
- **Dedicated:** This type of OS is designed to be used in specific systems with limited resources specific to a given machine which makes it very efficient by nature.
- **Parallel:** Computation is scattered among different processors. The processors run at same time and have shared memory and clock.
- **Multi-programming:** Multi-user operating system that uses time-sharing.

III. **Under what circumstances would a user be better of using a time-sharing system rather than a PC or single-user workstation?**

When there are few other users, the task is large, and the hardware is fast, time-sharing makes sense. The full power of the system can be brought to bear on the user's problem. The problem can be solved faster than on a personal computer. Another case occurs when lots of other users need resources at the same time, with time-sharing 10 users could be sharing the same computer instead of having 10 PCs. A personal computer is best when the job is small enough to be executed reasonably on it and when performance is sufficient to execute the program to the user's satisfaction.

Question #2

Consider a computer system with a single-core processor. There are two processes to run in the system: $P1$ and $P2$.

Process $P1$ has a life cycle as follows: CPU burst time of $15\ units$, followed by I/O burst time of $minimum\ 10\ units$, followed by CPU burst time of $10\ units$.

Process $P2$ has the following life cycle: CPU burst time of $10\ units$, followed by I/O burst time of $minimum\ 5\ units$, followed by CPU burst time of $15\ units$. Now answer the following questions:

a) **Considering a $single\ programmed$ operating system, what is the minimal total time required to complete executions of the two processes? You should explain your answer with a diagram.**

In a $single\ programmed$ system we fully execute one process at a time, sequentially. So, the minimal time required will be the sum of the time units it takes to fully process both processes' cycles. $(15 + 10 + 10) + (10 + 5 + 15) = 65\ units$.

b) **Now considering a $multiprogrammed$ operating system, what is the minimal total time required to complete executions of the two processes? You should explain your answer with a diagram.**

In a $multiprogrammed$ system we optimize CPU utilization by switching between processes to stay busy when waiting. So, the CPU will not have to waste time waiting for $I/O\ operation$ to complete. As soon as it stops at the I/O stage of $P_1$the system will start executing P_2. The execution time is less than the sum of the cycles. $15 + 10 + 10 + 15 = 50\ units$.

c) **$Throughput$ is defined as the number of processes (tasks) completed per unit time. Following this definition, calculate the throughputs for parts a) and b) above. How does multiprogramming affect throughput? Explain your answer.**

$$Throughput_a = \frac{2}{65}$$

$$Throughput_b = \frac{2}{50}$$

The system using $multiprogramming$ is clearly more efficient because it's using the CPU the whole time with no idle waiting time wasted.

Question #3

I. What is the performance advantage in having device drivers and devices synchronize by means of device interrupts, rather than by polling (i.e., device driver keeps on polling the device to see if a specific event has occurred)? Under what circumstances can polling be advantageous over interrupts?

Device drivers and devices synchronize by means of device interrupts are more efficient than polling because they allow the CPU to process other operations until it gets interrupted by a process instead of wasting time polling the devices.

Polling can be faster than interrupts (but wastes more resources) so it could be advantageous if the system does not have many time sensitive operations to run.

Polling can be advantageous for devices that interrupt the CPU very frequently anyways or when the latency of a device is very small, since it would take a lot less time than a context switch.

II. Is it possible to use a DMA controller if the system does not support interrupts? Explain why.

Yes, it is, and it would be better. The $DMA\ Controller$ would take the bus away from the CPU and save time. Even if the system does not support interrupts, when the DMA is transfeering data the CPU will be free to process other operations instead of waiting for any interrupts.

III. The procedure $ContextSwitch$ is called whenever there is a $switch$ in context from a running program A to another program B. The procedure is a straightforward assembly language routine that saves and restores registers and must be atomic. Something disastrous can happen if the routine $ContextSwitch$ is not atomic.

a) Explain why $ContextSwitch$ must be atomic, possibly with an example.

$ContextSwitch$ must be atomic to maintain the consistency of processed data. If it were not atomic then it would be possible to context switch anywhere during an operation. Which would process only parts of the data without guarantee of accuracy and negative side effects. For example, consider the programming assignment. The server thread always had to end after the client thread. This is required because if the server thread ended before the client thread, we wouldn't be able to guarantee successful processing.

b) Explain how the atomicity can be achieved in practice.

During a context switch, disable all interrupts which will ensure that nothing else can interrupt the context switch operation until it completes the given process and turns interrupts back on. We can achieve atomicity also through synchronization.

Question #4

I. **If a user program needs to perform I/O, it needs to $trap$ the OS via a system call that transfers control to the kernel. The kernel performs I/O on behalf of the user program. However, systems calls have added overheads, which can slow down the entire system. In that case, why not let user processes perform I/O directly, without going through the kernel?**

Letting the processes perform I/O operations would give them power over the OS and is a dangerous security risk. If I/O instructions are entrusted to users, they may misuse them and if there's a malicious intent then one can ruin the data on disk causing the system to crash. That's why the OS offers this function through appropriate system calls.

II. **Consider a computer running in the user mode. It will switch to the monitor mode whenever an interrupt or trap occurs, jumping to the address determined from the interrupt vector.**

a) **A smart, but malicious, user took advantage of a certain serious loophole in the computer's protection mechanism, by which he could make run his own user program in the monitor mode! This can cause disastrous effects. What could have he possibly done to achieve this? What disastrous effects could it cause?**

Let the application have root access ($monitor\ mode$) thinking that it'd be best since it can make the I/O operations run faster. He probably bypassed the system calls which gave the user program permission to access otherwise inaccessible information.

b) **Suggest a remedy for the loophole.**

Use $system\ calls$ or $message\ passing$ and let the OS perform the I/O operations on behalf of the process so it can stay in $user\ mode$.

Question #5

Suppose that a $multiprogrammed$ system has a load of N processes with individual execution times of $t_1, t_2, \dots, t_N$. Answer the following questions:

a) **How would it be possible that the time to complete the N processes could be as small as: $maximum\ (t_1, t_2, \dots, t_N)$?**

By taking advantage of multiprogramming techniques like Time and Space multiplexed. The CPU runs multiple programs at the same time while sharing memory and time, so we end up running N processes in $maximum\ (t_1, t_2, \dots, t_N)$. It is worth noting that this improves the overall system performance but not a single processes' performance and requires additional security and fairness measures since running processes on shared memory raises security concerns. Consider the example of these 3 processes and their life cycles.

$$P_1 = 10\ units\ CPU + 10\ units\ I/O\ + 5\ units\ CPU$$
$$P_2 = 5\ units\ CPU$$
$$P_3 = 5\ units\ CPU$$

When P_1 reached it's I/O block and waits for resources, the system will finish both P_2 and P_3. So total execution time is $25\ units$, which is equal to $maximum(25,5,5)$.

b) **How would it be possible that the total execution time, $T > (t_1 + t_2 + \dots + t_N)$? In other words, what would cause the total execution time to exceed the sum of individual process execution times?**

This can happen on any system that has an overhead associated with the context switch operation. If the processes are not allowed to run in parallel or need to context switch.

Consider the example of these 2 processes and their life cycles.

$$P_1 = 5\ units\ CPU + 1\ units\ I/O\ + 5\ units\ CPU\ +\ 1\ units\ I/O$$
$$P_2 = 5\ units\ CPU + 1\ units\ I/O\ + 5\ units\ CPU\ +\ 1\ units\ I/O$$
$$context\ switch = 1\ units$$

Therefore, in this scenario the system will context switch 3 times and the total execution time is

$$T = (5 + 1 + 1 + 5 + 1 + 1 + 5 + 1 + 1 + 5 + 1) > (5 + 1 + 5 + 1) + (5 + 1 + 5 + 1)$$
$$27 > 24$$

Question #6

Which of the following instructions should be privileged? Explain why.

i. **Read the system clock: Non-Privileged –** Reading the system clock doesn't change anything related to the OS or processes. It is not a dangerous operation.

ii. **Clear memory: Privileged –** Clearing memory modifies information used by the OS. Deleting the information stored in $memory$ (used by OS or other processes) will cause issues and can crash the system.

iii. **Reading from user space: Privileged –** I/O operations are only allowed in $kernel\ mode$. It is dangerous because it can interfere with other processes so the appropriate $system\ call$ will allow it once it checks the validity of the operation.

iv. **Writing to user space: Privileged –** I/O operations are only allowed in $kernel\ mode$. It is dangerous because it can interfere with other processes so the appropriate $system\ call$ will allow it once it checks the validity of the operation.

v. **Copy from one register to the other: Privileged –** Allowing any process to copy between registers makes no sense because it would allow any user level process to access all information related to OS or any other process. Only $kernel\ mode$ should have access to the registers.

vi. **Turn off interrupts: Privileged –** Turning off interrupts will allow the process to control the CPU. It will be able to run indefinitely until it decides to hand over control of the CPU, in the case of a malicious exploit it can just keep looping and monopolize the CPU.

vii. **Switch from user to monitor mode: Privileged –** Switching from $user$ to $monitor\ mode$ gives the user too much control over the system (root). It is the whole point of having Dual mode in OS; to protect the system from user level processes accessing its information or each others'. Which is why the OS forces any process to use system resources through a provided $system\ calls\ interface$.

Question #7

Assume you are given the responsibility to design two OS systems, a $Network\ Operating\ System$ and a $Distributed\ Operating\ System$. Indicate the primary differences between these two systems. Additionally, you need to indicate if there any possible common routines between these systems? If yes, indicate some of these routines. If no, explain why common routines between these two systems do not make sense.

Network operating system will have multiple computers connected on a network to operate together, managed by a server. This would allow a user to use a program that is located on another computer. The computers don't necessarily have to be running on the same operating system. A distributed OS is the same OS running on different computers with their own independent memories and clocks. There are no common routines between both systems. In a network OS, when running a routine, it will call the host and then perform it. On a distributed system there is no host to call the OS is running there.