2301871196

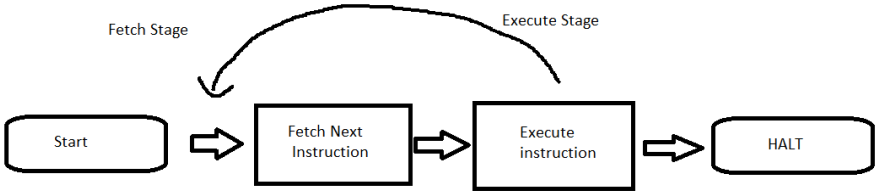
Matthew Brian

Operating Systems

I. Essay

1. Computer and O/S Overview
   a) The **program counter (PC)** will hold the address of the next instruction to be fetched, this present in a basic instruction cycle where after starting, the cycle will fetch next instruction, then execute that instruction. If the cycle is not in HALT, then the stage will be executed and the stage will be fetched again. Unless instructed otherwise, the processor always increments the PC after each instruction fetch so it will fetch the next instruction in sequence.

   Its relationship with an **instruction register(IR)** is that the fetched instruction gained in the basic instruction cycle is that the fetched instruction is loaded into the IR. The IR will read the bits of information and specify the actions that the processor is going to take. The processor than interprets the actions read by the IR and does the written action.



   In computing, many factors can contribute to the interruption of a process by another process. Whether it be when a specific task needs access to disk storage, or even when a process of high priority comes in the 'ready state'. In the case of the latter, the execution of the running process should be stopped, and the higher priority process should be given the extra disk space or even be given the CPU for its execution.

   During this sudden interruption, the register that must be performed in order to successfully switch from one process or task to another is a context switch. When a context switch is performed, this involves the switching of the CPU in order to focus on the task with the most priority. In this phenomenon, the execution of the process that is present in the running state is suspended by the kernel and another process that is present in the ready state is executed by the CPU.

   Many factors come into effect when context switching, but the main feature and use it has is that context switching will save all context of the previous process the CPU was switched out of. Without this present, the CPU will have to resume the previous process from the start opposed to starting where it was left out before the switch. A context is the contents of a CPU's registers and program counter at any point in time of the previous process, this may also include a

CPU's instruction register. The program counter is a special-purpose register that is used by the processor to hold the address of the next instruction to be executed.

b) In modern operating systems, these objectives can be summarized in three points as according to the three main objectives of operating system(convenience, efficiency, and the ability to evolve) these objectives can be achieved through operating system as a user/computer interface, operating system as resource manager, and ease of evolution of an operating system. In order, these will in then make an OS more convenient to use, resources of an OS will be used in an efficient way and emphasize the construction of an OS that permits the effective development, testing, and introduction of new system functions and features.

The operating System as a User/Computer Interface can be viewed as a layered fashion. This will allow the end user to view the interfaces in forms of an application set, this can be expressed in a programming language created by an application programmer. Those applications will then make use of these provided services to further promote the objective of convenience: program development, program execution, access to I/O devices, controller access to files, system access, error detection and response, accounting, instruction set architecture(ISA), and application programming interface(API).

The Operating System as Resource Manager is where resources get managed, such as I/O, main and secondary memory, and processor execution time. The main resources managed by an OS will include its kernel, which contains one of the most frequently used functions of an OS, and that is why resources must be used efficiently.

Ease of Evolution of an Operating System embodies the last point of an operating system objective due to how flexible an operating system can be. With its nature an OS is built to withstand a long amount of time due to easy it is to maintain and update. This will also require the OS to be built in modular fashion.

2. Process Description and Control
   a) Before finding out how LINUX/UNIX handles a zombie and orphan process. We must first find out what those processes are and how they differentiate from one another.

   A zombie process happens in a thread where a process has terminated but a parent did not yet invoke wait(). It is then due to the parent not reading its exit() status a zombie process will then be made from the terminated child. LINUX/UNIX handles this situation by implementing a wait call to the parent, so that a parent must wait for its children to execute its exit() call and the parent can take back its child.

An orphan process is created when a child is in the middle of executing a command, but its parent exits abruptly. This results in the process having no parent and in turn makes them orphans. LINUX/UNIX handles this situation by adopting the orphan process by assigning a 'init' process as the parent of the orphan. The 'init' process periodically invokes wait(), allowing the exit status of an orphan to be processed and terminated without its original parent.

b) A fork() happens in process creation, this function returns three types of values a negative, positive, and a zero value. The functions being performed are represented in the points below.
   - A slot being allocated in the process table for a new process
   - The child process will get assigned a unique process ID
   - A process image copy of the parent will be made
   - Any counter owned by the parent will get incremented
   - Child process be in a ready state
   - ID number of the child will be returned

   The code snippet below will show simple use of the fork() function:

```
main(){

int parentID;

parentID = fork();

printf ("%d \n", parentID);

}
```

3. Multiprocessor, Multicore and Embedded System
   a) Fine-grained parallelism is a far more complicated application of parallelism than threads. Despite the fact that a lot of work has been done on highly parallel applications, it is still a specialized and fragmented field with a lot of diverse techniques.

      In a typical multiprocessor that deals with coarse-grained or independent synchronization granularity, it is obvious that each individual processor should be able to switch between a number of tasks in order to achieve high utilization and hence higher performance. The issue is less obvious for medium-grained applications operating on a multiprocessor with several processors.

      When a large number of processors are available, it is no longer necessary for each processor to be as active as feasible. Rather, we are focused with providing the optimum performance for the applications on average.

   b) A D/A converter is a device that converts a precise number, usually a fixed-point binary integer into a physical amount. When converting finite-precision time series data to a continuously fluctuating physical signal, D/A converters are frequently utilized.

An ideal D/A converter extracts abstract numbers from a series of impulses, which are then processed via interpolation to fill in the gaps between them. A traditional D/A converter converts the data into a piecewise constant function composed of a series of rectangular functions and modelled using the zero-order hold.

A D/A converter reconstructs original signals so that its bandwidth meets certain requirements. With digital sampling comes quantization errors that create low-level noise which gets added to the reconstructed signal. The minimum analogue signal amplitude that can bring about a change in the digital signal is called the Least Significant Bit (LSB), while the (rounding) error that occurs between the analogue and digital signals is referred to as quantization error.

For the A/D converter, the amplitude of the analogue signal is broken up and sampled into discrete intervals by the A/D converter, which are subsequently translated into digital values. The number of bits is often used to represent the resolution of an analogue to digital converter.

4. Threads
   a) In general, this will make the process faster due to how a thread is able to schedule and organize a processes priority. Without a thread multiple processes are able to use the same resource and may in end result in a deadlock.

   b) We can use semaphore in this case due to the reason that both require the same resource. In this case thread 1 will see the mutex and if the mutex is being used the thread will wait. But in this case thread 1 will update first so since it is first the thread will decrement the mutex. After updating the thread will send a signal to increment the mutex so that thread 2 will get out of waiting.

5. Process Scheduling
   a)

Highest Response Ratio

Process	CPU Burst Time	Arrival Time	Finish Time	Turnaround Time	Waiting Time	Response Time
A	5	0	5	5	0	0
B	18	2	23	21	3	3
C	1	7	24	17	16	16
D	3	10	27	17	14	14
E	3	12	30	18	15	15

	A	B	C	D	E
0	5	23	24	27	30

Queue:

B(18)	C(1) D(3) E(3)	D(3) E(3)	E(3)	(-)

First Ratio	C	16	+	1	/	1	=	17
	D	13	+	3	/	3	=	5.3
	E	11	+	3	/	3	=	4.6
Second Ratio	D	14	+	3	/	3	=	5.6
	E	12	+	3	/	3	=	5

Average Turnaround Time: 5 + 21 + 17 + 17 + 18 = 15,6
Average Waiting Time: 0 + 3 + 16 + 14 + 15 = 9,6
Average Response Time: 0 + 3 + 16 + 14 + 15 = 9,6

Round Robin

Process	CPU Burst Time	Arrival Time	Finish Time	Turnaround Time	Waiting Time	Response Time
A	5	0	9	9	4	0
B	18	2	30	28	10	2
C	1	7	10	3	2	2
D	3	10	17	7	4	4
E	3	12	20	8	5	5

	A	B	A	C	B	D	E	B	B	B
0	4	8	9	10	14	17	20	24	28	30
	B(18)	A(1)	C(1)	B(14)	D(3)	E(3)	B(10)			
		C(1)	B(14)	D(3)	E(3)	B(10)				

Average Turnaround Time: 9 + 28 + 3 + 7 + 8 = 11
Average Waiting Time: 4 + 10 + 2 + 4 + 5 = 5
Average Response Time: 0 + 2 + 2 + 4 + 5 = 2,6

With this done we can now see that round robin has less turnaround time and less waiting time. This means that round robin has better efficiency.

6. Synchronization and Deadlock
   a) The Dining Philosophers Problem happens when a group of Philosophers are placed in a round table with a fork placed in between each one of them and a plate of spaghetti is placed in front of them. The catch is each philosopher may only eat their respective plates of spaghetti when two forks are available for them to use. This will indicate the forks as their resources and a philosopher must not eat when a fork next to them is in use, in other words, the Philosopher must think(wait).

   A solution to this problem may be presented when the philosophers are given an integer to indicate the total number of philosophers, this will also indicate the unique forks, then each philosopher should start of by executing a 'semaphorewait[chopsticks(i)]', 'i' indicating the chopstick number a philosopher is about to use, this will then decrement the total number so that when another philosopher wants to eat, he/she will have a unique number to decrement. After decrementing 'i', the number of chopsticks must be modulo 'ed by the total number of philosophers so that the philosopher may be able to check whether the chopstick is in use or not.

   If all goes well and a chopstick is available, the philosopher wanting to use the resource will proceed with eating the spaghetti. This will put any other philosopher requesting to eat with a fork being used on wait and will not go forward with eating the spaghetti. After done eating, the philosopher that was using a resource(fork) will then go on with a 'signal()', this will increment the

resource that was being used so that when a philosopher wants to check if that fork is available, that philosopher may go on with eating.

b)

Current Allocation

	S	T	U	V
A	1	2	1	1
B	2	0	2	1
C	1	1	0	1
D	2	1	0	1

Current Requirements

	S	T	U	V
A	1	3	2	4
B	3	1	6	2
C	1	2	1	1
D	3	1	0	2

Existing resources: S = 6, T = 5, U = 5, V = 4

Need table

	S	T	U	V
A	0	1	1	3
B	1	1	4	1
C	0	1	1	0
D	1	0	0	1

Total Allocated	6	4	3	4
System Has	6	5	5	4
Available	0	1	2	0

Result

Grant 1	C	0 1 2 0	+	1 1 0 1	=	1 2 2 1
Grant 2	D	1 2 2 1	+	2 1 0 1	=	3 3 2 2
DEADLOCK						

With the given result above final, we can assume the process grants C followed by D due to it sufficing the available space. But after D the system goes into **DEADLOCK** due to insufficient available space.