CS3510: Operating Systems I
Finals: Autumn 2020

Soumi Chakraborty
ES19BTECH11017

Answer 1
In a batch operating system, three jobs are submitted for execution. Each job involves an I/O activity, CPU time and another I/O activity of the same time span as the first. Job JOB1 requires a total of 23 ms, with 3 ms CPU time; JOB2 requires a total time of 29 ms with 5 ms CPU time; JOB3 requires a total time of 14 ms with 4 ms CPU time. Illustrate their execution and find CPU utilization for uniprogramming and multiprogramming systems. Here CPU utilization is defined as ratio Total execution time by CPU execution time.

Total time = CPU + 2 * I/O
CPU utilization = Total time/CPU time

JOB1
Total time: 23 ms
CPU time: 3 ms
I/O total time: 23 - 3 = 20 ms
I/O time: 10 ms

JOB2
Total time: 29 ms
CPU time: 5 ms
I/Ototal time: 29 - 5 = 24 ms
I/O time: 12 ms

JOB3
Total time: 14 ms
CPU time: 4 ms
I/O total time: 14 - 4 = 10 ms
I/O time: 5 ms

Uniprogramming: sequential/serial execution
Total execution time = 23 + 29 + 14 = 66 ms
Total CPU time = 3 + 5 + 4 = 12 ms
CPU utilization = 66/12 = 5.5

Multiprogramming: a job doing I/O doesn't have to occupy the CPU and other processes ca work in the background

Time (ms)	Job	Type/Remarks
5	JOB3	
9	JOB3	CPU time done

Answer 2

a. Let's assume that the FGI only allows taking input when it is set to 0 (which implies that the INPR register is full when FGI is 1). Similarly, let's assume the FGO only allows printing when its value is set to 0 (which implies that the OUTR register is full when FGI is 1).

   Now to achieve I/O with the teletype:
   1. The CPU must check if the INPR is empty, and it confirms this by ensuring that the value of the FGI is 0.
   2. If the FGI stores a zero, the user can enter one alphanumeric symbol (and it gets converted to an 8-bit word implicitly). After this, the FGI's value is updated to 1, and the INPR is no longer empty.
   3. Now, the CPU has to print this word. To do so, this word must first be stored in the OUTR, but it can only store it here if the register is empty, and the input register is full. Hence it checks the value of FGO. If the FGO stores a 0 and the FGI stores a 1, we can transfer the word from the INPR to the OUTR.
   4. FGO is set to 1. Now the word stored in the OUTR must be changed back to it's alphanumeric symbol form and printed as output.
   5. The value of the FGO is now changed to 0 to empty the OUTR. FGI is set back to 0 which signifies that INPR is ready for its next keystroke, and hence the process repeats.

   Thus in this scenario, the CPU has to continually check the values of te FGI and FGO flags and can only proceed taking input and printing the output if the two flags are at the appropriate values.

b. If we had an IEN flag at our disposal, the I/O process would have become much faster. This is because the CPU could just move on to the next step in the process depending on the value of the IEN flag rather than having to continuously check the values of the FGI and the FGO flags. The FGI and FGO flags would send interrupts when necessary to change the value of the IEN flag, and the CPU would just have to interact with one flag.

Answer 3
4 bytes = 32 bits. I.e., half of the microprocessor (the first 32 bits) is reserved for the opcode and the other half (the last 32 bits) contain an immediate operand or an operand address.

Bits available = 64 - 32 = 32 bits
Total capacity = 2^{32} bits (2^32) = 4294967296 bits

Answer 4

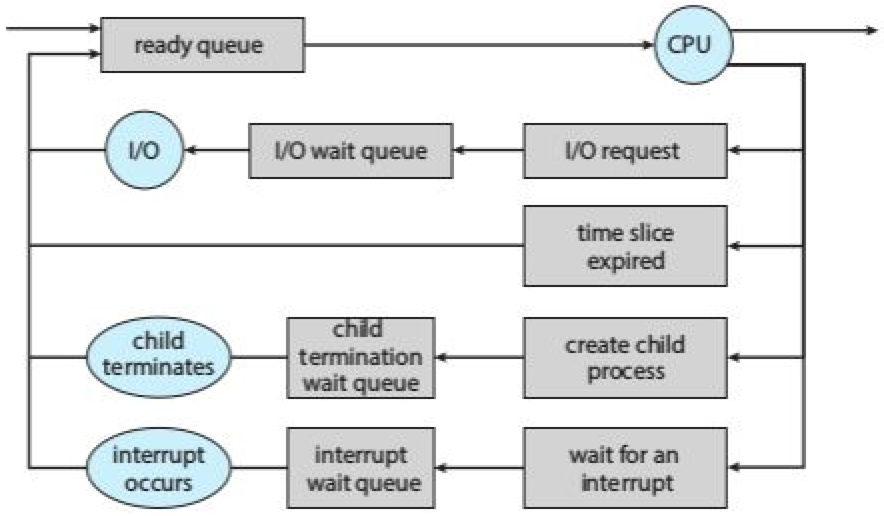

Figure 3.5 Queueing-diagram representation of process scheduling.

a. Yes it is possible to allow a process to wait on more than one event at the same time. For example, if we were to take a website with a form under consideration, which expects multiple I/O operations from the user. Each part of the form is a separate I/O operation, but the user cannot access the next step/page without filling up the entire form. So in such a case, a process would have to wait in multiple wait queues for each field in the form.
b. Queuing structure: for this particular example, we have multiple I/O wait queues. In other examples, it may be a different type of queue. Hence, as a general example, we can either put loops around each event or add counters to the queueing diagram. Instead of sending a process back to the ready queue after just one iteration, we'll update the mentioned counters, and send the control back to where it came from for the next event. Once every counter's value is up to the required value, we send the process back to the ready queue as usual.

Answer 5
This question can be solved using Amdahl's law.

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

Given, S = 20% = 0.2.

a. N = 4

   speedup <= 1/(0.2 + (1 − 0.2)/4)

   speedup <= 2.5

   Therefore, max speedup will be 2.5 times.

b. N = 8

   speedup <= 1/(0.2 + (1 − 0.2)/8)

   speedup <= 3.33

   Therefore, max speedup will be 3.33 times.

Answer 6

Thread pools limit the maximum number of threads a process can create and thereby reduce the risk of all the resources getting exhausted by the threads. Moreover, a thread pool removes the overhead which would otherwise slow down the process due to having to create numerous threads.

Thread pooling might be useful in an application which requires a lot of parallel computing. For example, if an application were to search through a large database, doing this parallely rather than concurrently will be much more efficient, and a process can create separate threads to search through different parts of the database (i.e., data parallelism not task parallelism). If we use a thread pool for this, once a particular thread is done searching through it's part of the database, another part can reuse this thread because of thread pooling. The advantages of this are:

- Each section of the database won't have to create its own thread, hence it will be faster
- Thread pooling will prevent the application from going overboard with the threads and eating up all the resources

Answer 7

If the algorithm was using multithreading (excluding the many-to-one type) on a multiprocessor system to do its calculations, it would definitely be more efficient than it would be without threads because it would make full use of the multiple processors and speed up the calculation. However, if we had a single-processor system or had implemented the many-to-one type of multithreading, we would be better off by not using threads at all. This is because in both the mentioned cases, the threads do nothing whatsoever to speed up the process and moreover, they add to the overhead because creating new threads just waste precious time and give nothing in return.

Hence, the answer to this question is in the case of a single processor system, we would benefit from not using threads.

Answer 8

12 msec => 2/3rd of the time (hits)

12+75 = 87 msec => 1/3rd of the time (misses)

1) **Single Threaded**
Average time for a request = (⅔) * 12 + (⅓) * 87 = 37 msec

Time	Number of Requests
37 msec	1
1 msec	1/37
1 sec	1000/37 = 27.027027027

Thus, in the single threaded case, the server can handle at least **27 requests per second.**

2) **Multi Threaded**
In this scenario, the time that the thread sleeps doesn't need to be factored into the time taken for one request because all the threads sleeping at the same time when one of them requires a disk operation (or ALL of them requiring a disk operation at the same instant) is not something that is very plausible. Hence, we don't need to account for the additional 75ms here.

Time	Number of Requests
12 msec	1
1 msec	1/12
1 sec	1000/12 = 83.3333333333

Thus, in the multi threaded case, it's safe to assume that the server can handle approximately **83.33 requests per second.**