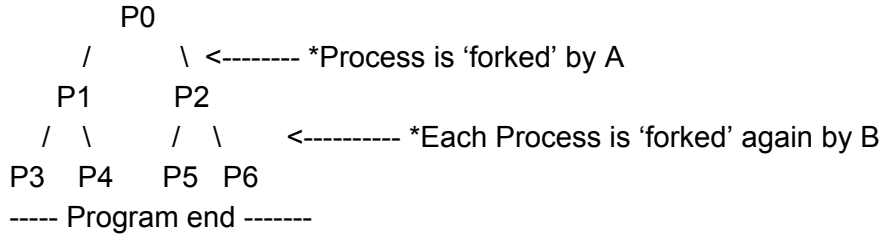
Question 1. In the following piece of C code, how many processes are created when it is executed? Explain your answer and include diagrams as appropriate.

```
int main() {
  fork();  // A
  fork();  // B
  exit(1);
}
```

Answer: 4



(P3, P4, P5, P6) = **4 processes**

You can also use 2^(# of forks) == num processes…

2^(2) = **4**

Question 2. Describe how a web server might leverage multi-threading to improve performance. Include diagrams if you feel this will make your answer clearer.

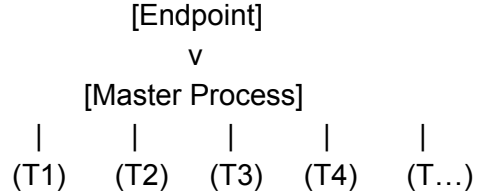
Answer

A web server could definitely benefit from multithreading as many web requests could be coming in: some simple and some more complex. You wouldn't want a thread to be blocked when a complex task comes and halts other operations from happening. That means if one person does a complex request, any request made after would have to wait until that first one was done.

So a server should create multiple threads in order to handle simultaneous/multiple requests. If a heavy process were to come in it could be sent to another thread to process.

Here's a quick diagram:

Lets define some requests that user(s) would make
R1: 12:00:00 GET csoti.xyz/api/solveworldhunger
R2: 12:00:01 GET csoti.xyz/api/helloworld



In this setup the users can request something from the endpoint (say R1 or R2) which is controlled by the Master Process -- Here Master Process (MP) intelligently (Based off of what thread is free, or has a smaller queue, or which thread is better for what) decides what thread to send the process to.

So when the user requests **R1**, MP can route the request to **T1**. Right after that it gets another request, **R2**. Rather than halting, the MP can send it to **T2**. As **R2** is pretty simple its able to respond within **2000ms**. However as **R1** is quite complex it takes about **5000ms** to finish an return. This system allowed both requests to run at the same time rather than **R1** blocking **R2.**

Total time
Singlethread: 2000ms + 5000ms = 7000ms
Multithreaded: max(2000ms, 5000ms) = 5000ms

Question 3. Briefly outline the role of the Process Control Block (PCB), listing and describing three pieces of information an Operating System might choose to store in the PCB.

Answer

A PCB's role is to hold info about each process for the OS to save/restore state (context switching). Three possible things it could hold are:

1. Process ID
2. Program Counter
3. Scheduling info

Question 4. Identify the critical section and show how the following pseudo code could be modified to avoid deadlock. Explain your answer.
Shared Variables: lockA, lockB, resourceA, resourceB

Process1 {
  aquire(lockA);
  aquire(lockB);
  modify(resourceA);
  modify(resourceB);
  release(lockB);
  release(lockA);
}
	Process2 {
	  aquire(lockB);
	  aquire(lockA);
	  modify(resourceB);
	  modify(resourceA);
	  release(lockA);
	  release(lockB);
	}

Answer

The bolded sections above are critical. To modify this code I'd write

Process1 {
  aquire(lockA);
  modify(resourceA);
  release(lockA);
  aquire(lockB);
  modify(resourceB);
  release(lockB);
 	
}
	Process2 {
	  aquire(lockB);
	  modify(resourceB);
	  release(lockB);
	  aquire(lockA);
	  modify(resourceA);
	  release(lockA);
	}

Question 5. Define the terms “race condition”, “deadlock”, and “starvation” as they relate to Operating System design and outline the relationship between deadlock and starvation.

Answer

Race Condition: Where two tasks “race” and happen to access/change the same shared space

Deadlock: Where tasks are all blocked as each one is holding a resource that the other resources want, leading the system to halt

Starvation: A thread waits forever

Deadlock => Starvation (Deadlock leads to/means there **will** be starvation)
Starvation X=> Deadlock (Starvation does **not** mean deadlock)

Question 6.
(a) In a multiprogrammed environment with 16MB of memory where all processes require 1MB of unshared memory and spend 60% of their time in I/O wait, calculate how much memory will remain unused when approximately 99% CPU utilization is achieved.

(b) In the same multiprogrammed environment, if each process now requires 3MB of unshared memory, calculate the maximum achievable CPU utilization. Show all your work.

Answer

a) CPU Utilization = 1 - p^n

p^n = Probability that all n processes are waiting for I/0

.99 = 1 - (.6)^n
- .01 = - (.6)^n
.01 = (.6)^n
log.6(.01) = n
N = 9.01 => 9

9 processes x 1MB = 9MB
16MB total - 9MB used = 7MB unused

7MB unused

b) CPU Utilization = 1 - p^n

P = 16MB / 3MB each = 5.33 => 5 processes
N = .60

CPU = 1 - (.6)^5

CPU = .92224

CPU Utilization = **92%**

Question 7. If we assume that when processes are interrupted they are placed in a queue containing all non-running processes not waiting for an I/O operation to complete, briefly describe two strategies the Operating System might adopt to service that queue. One-word answers will not suffice.

Answer

...

Question 8. Can the “priority inversion” problem outlined in section (3) of the background information to Lab 2 occur if user-level threads are used instead of kernel-level threads? Explain your answer.

Answer

No I don't think this can happen with user-level threads. The user threads are invisible to the OS so a thread cant be preempted (no preemption possible) so this “priority inversion” could never happen

Question 9. List and describe the necessary conditions for deadlock.

Answer

- Mutual exclusion
    - Where at least one resource does not support simultaneous use // the resources are unshareable
- No preemption
    - A resource cannot be seized/taken from the holder
- Hold and wait
    - At least one process must hold a resource and be waiting on another different resource
- Circular wait
    - A waiting cycle that a chain of process are waiting for a process that the next process has
    - EX: A → B → C → D → A

Question 10. Explain how quantum value and the time taken to perform a context switch affect each other in a round robin process-scheduling algorithm.

Answer

Quantum value is how long the a process will execute before performing a context switch (which also takes time) and putting that process at the back of the queue

There are two scenarios:

- The **process** finishes before the quantum: the process is done and the next is loaded
- The **quantum** expires before the process -- the process is not done and the next one is loaded

This means RR succeeds when **time_to_finish < quantum** and has bad performance when all the jobs take the **same time** (meaning time is wasted to switch contexts)

Now this means you must find the right quantum value and also try to reduce the expensive context change.

- If “q” is too **big** it’s basically the same as first come first served (all tasks will finish before the q time)
- If “q” is too-too **small,** the time to context switch will basically be the same -- then the overhead is too big (more time switching rather than real work)

Thus both values affect each other, and the right “q” is needed for the processes such that the overhead of the context switch is worth it in Round Robin

Question 11. Describe a mechanism by which counting semaphores could be implemented using the minimal number of binary semaphores and ordinary machine instructions. Include C code snippets if you feel this will make your answer clearer and/or more concise

Answer

```
// 2 locks needed???
sema lock_a;
sema lock_b;

// down
P_counting(int c){
        if(c > 0)
                // call down??
        Else
            // ...
}

// up
V_counting(int c){
        if(c <= 0)
                // call up??
        Else
            // ...

}
```

Question 12. Five threads, A through E, arrive in alphabetic order at a scheduling queue one second apart from each other. Estimated running times are 10, 6, 2, 4, and 8 seconds, respectively. Their externally determined priorities are 3, 5, 2, 1, and 4, respectively, 5 being the highest priority. For each of the following scheduling algorithms, determine the mean turnaround time and mean waiting time. Assume thread switching is effectively instantaneous.

(a) First Come First Served
(b) Round Robin
(c) Preemptive Priority Scheduling
(d) Preemptive Shortest Job First

For (b), assume the system is multi-programmed with a quantum of 4 seconds. In all cases, show your work and include diagrams/charts/tables as appropriate.

FCFS

	Time to finish	Arrival	Mean waiting time = (0-0) + (10-1) + (16-2) + (18-3) + (22-4) / 5 = MWT = 0 + 9+ 14 + 15 + 22 / 5 = 60 / 5 = 12 **MWT = 12** Mean turnaround time = (0-0) + (16-1) + (18-2) + (22-3) + (30-4) / 5 = MTT = 0 + 15 + 16 + 19 + 26 / 5 = 76 / 5 = 15.2 **MTT = 15.2**
T1	10	0	
T2	6	1	
T3	2	2	
T4	4	3	
T5	8	4	

Round Robin

Time 0: T1(10)
1: T1(9) T2(6) // T2 joins
2: T1(8) T2(6) T3(2) // T3 joins
3: T2(6) T1(7) T3(2) T4(4) // 4s of quantum reached - next process turn
4: T2(5) T1(7) T3(2) T4(4) T5(8) // T5 joins
5: T2(4) T1(7) T3(2) T4(4) T5(8)
6: T2(3) T1(7) T3(2) T4(4) T5(8)
7: T1(7) T3(2) T4(4) T5(8) T2(2) // quantum reached
8: T1(6) T3(2) T4(4) T5(8) T2(3)
9: T1(5) T3(2) T4(4) T5(8) T2(3)
10: T1(4) T3(2) T4(4) T5(8) T2(3)
11: T3(2) T4(4) T5(8) T2(3) T1(3) // quantum reached
12: T3(1) T4(4) T5(8) T2(3) T1(3)

13: T4(4) T5(8) T2(3) T1(3) // T3 finishes
14: T4(3) T5(8) T2(3) T1(3) // Still time to quantum so T4 starts to process
15: T5(8) T2(3) T1(3) T4(2) // Quantum
16: T5(7) T2(3) T1(3) T4(2)
17: T5(6) T2(3) T1(3) T4(2)
18: T5(5) T2(3) T1(3) T4(2)
19: T2(3) T1(3) T4(2) T5(4) // quantum
20: T2(2) T1(3) T4(2) T5(4)
21: T2(1) T1(3) T4(2) T5(4)
22: T1(3) T4(2) T5(4) // T2 finishes so T1 starts
23: T4(2) T5(4) T1(2) // quatum
24: T4(1) T5(4) T1(2)
25: T5(4) T1(2) // T4 finishes, T5 starts
26: T5(3) T1(2)
27: T1(2) T5(2) // quantum
28: T1(1) T5(2)
29: T5(2) // T1 finishes
30: T5(1)
31: --- done --- //T5 finishes and quantum reacched

Mean waiting time = (0-0) + (3-1) + (11-2) + (13-3) + (15-4) / 5 =
MWT = 0 + 2 + 9 + 10 + 11 / 5 = 32 / 5 = 6.4
MWT = 6.4

Mean turnaround time = (29-0) + (22-1) + (13-2) + (25-3) + (31-4) / 5 =
MTT = 29 + 21 + 11 + 22 + 27 / 5 = 110 / 5 = 22
MTT = 22
