SAMPLE QUESTIONS FOR MIDTERM, WINTER 2021

This sample midterm contains 40 multiple-choice questions. A correct answer is worth 2 points, and incorrect answer is worth -1 points, and no answer is worth 0 points. Consequently, there is a (heavy) penalty for guessing, so make you selection based on whether you know the answer or not, otherwise you may choose to leave it blank. For each question, select the best choice (in some cases, more than one answer may seem correct, but there's only one BEST answer, and it is important that you think through each choice to figure out which is indeed best). Solutions are below.

1. The routine RestoreContext: (a) always returns 0; (b) always returns the PID of the calling process; (c) returns either 0 or the PID of the calling process; (d) does not return.

2. To yield to another process, a process MUST: (a) save the program counter last; (b) save the stack pointer before the program counter; (c) call SaveContext; (d) call RestoreContext.

3. Synchronization is used: (a) to avoid deadlocks; (b) to avoid race conditions; (c) to set an alarm; (d) to transfer data from one process to another.

4. A critical section: (a) must execute before all non-critical sections; (b) must execute before another process executes; (c) can be executed by multiple processes only on a multiprocessor; (d) can only be executed by one process at a time.

5. Given two threads, if each try to modify a variable, a race condition will not result from those modifications if the variable is declared: (a) local; (b) global; (c) static global; (d) none of the above.

6. The semaphore Signal operation increments the semaphore's value: (a) only if there are no waiting processes; (b) and if there are no waiting processes, puts the process to sleep; (c) and if there are any waiting processes, wakes one of them up; (d) only if there are waiting processes.

7. Any interprocess communication (IPC) mechanism: (a) cannot be implemented solely with semaphores and shared memory; (b) requires mechanisms for data transfer or synchronization, but not necessarily both; (c) requires mechanisms for data transfer and synchronization; (d) may be implemented solely with semaphores.

8. In a monitor, mutual exclusion is achieved by: (a) allowing only one process to be active inside the monitor; (b) forcing a process to leave immediately if another is trying to enter; (c) allowing processes to wait and signal other processes; (d) forcing a process to wait if another is trying to enter.

9. Synchronization is achieved in message passing by: (a) not allowing a process to call receive until another process has called send; (b) allowing a process to call receive only after another process has called send; (c) making a process that calls receive to wait until the message is sent; (d) making a sending process check whether another process is waiting to receive a message.

10. When evaluated based on turnaround time, Shortest Process Next (SPN): (a) is always better than First-Come-First-Served (FCFS); (b) is never worse than Shortest-Remaining-Time (SRT); (c) may be better or worse than FCFS; (d) may be better or worse than Round-Robin (RR).

11. If processes are executed in shortest-to-longest order, what is minimized (that might not happen with other orders): (a) average time spent using the CPU; (b) total time before all complete; (c) average time spent waiting for the CPU; (d) total time the CPU is idle.

12. Which is a condition for mutual exclusion: (a) semaphores must be used to guard critical sections; (b) Receive must be called before Send when using message passing; (c) a process outside a critical

section cannot cause a process about to enter a critical section to block; (d) Signal must be the last statement in a monitor procedure.

13. When a process calls Wait(sem), which of the following cannot happen: (a) a blocked process becomes ready; (b) the calling process blocks; (c) a deadlock occurs; (d) the calling process does not block.

14. In the Banker's algorithm, a safe state means: (a) deadlock can definitely be avoided; (b) there are no guarantees that deadlock can be avoided; (c) one or more resources are available; (d) all resources are currently being used.

15. Peterson's solution: (a) can't be used for mutual exclusion; (b) can be used for mutual exclusion, but requires busy waiting; (c) can be used for mutual exclusion and avoids busy waiting; (d) is more efficient than using test-and-set.

16. Monitors require programming language (compiler) support to implement: (a) producer/consumer; (b) mutual exclusion; (c) context switching; (d) deadlock prevention.

17. To support timesharing, the following must be supported by the kernel: (a) the Yield system call; (b) preemption; (c) first-come-first-served scheduling; (d) semaphores.

18. To implement user-level threads: (a) the process stack must contain a stack per thread; (b) context switching between threads must be supported by the kernel; (c) thread scheduling is done by the kernel; (d) all of the above.

19. In PA2, DoSched is used to: (a) cause the kernel to make a scheduling decision at the next opportune time; (b) determine which process should run next; (c) causes SchedProc to be called immediately; (d) respond to a timer interrupt.

20. In PA2, a call to RequestCPUrate(n) MAY (i.e., it is possible that it will) be denied if: (a) $n < 100$; (b) $n > 100$; (c) $n = 100$; (d) all of the above.

21. One necessary condition satisfying mutual exclusion is: (a) no process outside its critical section may block other processes; (b) a process inside its critical section causes all other processes to block; (c) a process may cause another process in its critical section to block; (d) no process inside its critical section may block other processes outside their critical sections.

22. When comparing context switching time C to the length of a quantum L, typically: (a) $C << L$; (b) $C \approx L$; (c) $C >> L$; (d) can be any of the above.

23. Which of the following types of variables will NOT be inside an activation record: (a) only local; (b) only static local; (c) both local and static local; (d) neither local nor static local.

24. Given 4 processes with PIDs 1, 2, 3, 4, that arrive at the same time, which of the following sequences of SchedProc return values would be valid in ROUNDROBIN Scheduling: (a) 1234123412341234; (b) 1122334411223344; (c) 1111222233334444; (d) all the above.

25. In Multi-Level Feedback Queues, if a process is preempted by the arrival of a higher priority process and does not use the CPU time it was allocated, it will: (a) go to the next lower queue (assuming there is one); (b) go to the queue that it came from; (c) go to the next higher queue (assuming there is one); (d) any of the above may occur.

26. If two periodic processes A and B are being scheduled using Rate Monotonic Scheduling (RMS), where A's utilization is 30% and B's utilization is 35%: (a) all deadlines will be met; (b) no deadlines will be met; (c) can't say whether all deadlines will be met; (d) depends on the periods.

27. Which of the following is NOT true about Earliest Deadline First (EDF): (a) can achieve 100% utilization (ignoring overhead); (b) does not require preemption; (c) may require significant work, O(n) where n is the number of processes in the system, at every scheduling decision point; (d) works for aperiodic processes.

28. In PA2, a call to HandleTimerIntr: (a) will definitely lead to a call to SchedProc; (b) may occur in the middle of a quantum; (c) may cause a process to exit; (d) none of the above.

29. In PA2, Gettime can be used to implement: (a) ROUNDROBIN; (b) PROPORTIONAL; (c) FIFO; (d) none of the above.

30. In Stride Scheduling, the pass value is calculated as: (a) previous pass value + stride value; (b) L divided by the stride value; (c) 1 divided by the request value; (d) stride value + request value.

31. Say the contents of mem is 0, and an interrupt occurs during the execution of TSL(mem). After the instruction has executed, the contents of mem will be (a) 0; (b) 1; (c) cannot determine; (d) none of the above.

32. Which of the following transitions is invalid: (a) RUNNING to READY; (b) READY to RUNNING; (c) BLOCKED to READY; (d) BLOCKED TO RUNNING.

33. Which of the following causes control to transfer to the kernel: (a) executing a TSL instruction; (b) calling a procedure; (c) accessing a shared variable; (d) none of the above.

34. If a program variable is declared as a static global integer, when that program runs as a process, the memory will be allocated (a) in the kernel's stack for that process; (b) in the process's data segment; (c) in the process's stack segment; (d) in shared memory.

35. Which of the following will benefit the least from multiple CPUs: (a) a single process with multiple user-level threads; (b) a single process with multiple kernel-level threads; (c) multiple processes, each with a single user-level thread; (d) multiple processes, each with a single kernel-level thread.

36. In PA2, which of the following functions returns the process that should run next: (a) DoSched; (b) HandleTimerIntr; (c) MyRequestCPUrate; (d) SchedProc.

37. Which of the following need to be surrounded by the addition of <entry> and <exit> code to achieve mutual exclusion: (a) Wait operation in semaphores; (b) Wait operation in monitors; (c) while loop test in Peterson's solution; (d) all of the above.

38. When is a context switch NEVER allowed to occur under any circumstances whatsoever: (a) while executing in the kernel; (b) while executing in a critical section; (c) while executing in code that is expected to run atomically; (d) none of the above.

39. In Proportional Share scheduling, at each quantum, the kernel always selects the process that (a) got the least amount of CPU time so far; (b) has the maximum requested fraction of CPU time; (c) has the minimum ratio of requested fraction of CPU time to received fraction of CPU time; (d) none of the above.

40. When a process enters the kernel, we say that there is an "amplification of power" in that that process now has access to all the kernel data structures and can execute both non-privileged and

privileged CPU instructions (whereas outside the kernel, the process can only execute non-privileged instructions). Which of the following causes such an amplification in power: (a) switching from user mode to kernel mode; (b) executing a TRAP instruction; (c) responding to a hardware interrupt; (d) all of the above.

Solutions

1. The routine RestoreContext: (a) always returns 0; (b) always returns the PID of the calling process; (c) returns either 0 or the PID of the calling process; (d) does not return.

(d) RestoreContext restores the value of the PC to a saved value. Once the PC is loaded, a jump occurs to whatever location was saved, which has no relationship to how RestoreContext was called. Returning makes no sense for RestoreContext.

2. To yield to another process, a process MUST: (a) save the program counter last; (b) save the stack pointer before the program counter; (c) call SaveContext; (d) call RestoreContext.

(d) This is a very subtle question that requires careful thought. It is RestoreContext that actually causes control to transfer to another process. The other answers may be part of a yield operation, but the MOST IMPORTANT part is RestoreContext. In fact, if a process did not care whether it was ever resumed (such as if it were to exit), yield could ONLY do a RestoreContext. RestoreContext is the only operation that MUST be done to transfer control.

3. Synchronization is used: (a) to avoid deadlocks; (b) to avoid race conditions; (c) to set an alarm; (d) to transfer data from one process to another.

(b) The only way to avoid race conditions is to have one process wait for another, which is synchronization, so that they are not in critical sections at the same time.

4. A critical section: (a) must execute before all non-critical sections; (b) must execute before another process executes; (c) can be executed by multiple processes only on a multiprocessor; (d) can only be executed by one process at a time.

(d) A critical section identifies a region of code where indeterminate results can result if its execution by multiple processes is not serialized (executed by one after another). Having it executed by one process at a time ensures mutual exclusion.

5. Given two threads, if each try to modify a variable, a race condition will not result from those modifications if the variable is declared: (a) local; (b) global; (c) static global; (d) none of the above.

(a) A local variable exists only during the time between when the function in which it is declared is called and when it returns. Each call has a separate activation record, so if each thread calls the function, they will have their own copies of the local variable and there can be no race condition.

6. The semaphore Signal operation increments the semaphore's value: (a) only if there are no waiting processes; (b) and if there are no waiting processes, puts the process to sleep; (c) and if there are any waiting processes, wakes one of them up; (d) only if there are waiting processes.

Both (a) and (c) are acceptable, as each correspond to a different (correct) implementation of semaphores as given in the lecture notes.

7. Any interprocess communication (IPC) mechanism: (a) cannot be implemented solely with semaphores and shared memory; (b) requires mechanisms for data transfer or synchronization, but not necessarily both; (c) requires mechanisms for data transfer and synchronization; (d) may be implemented solely with semaphores.

(c) Again, straight from the notes: interprocess communication mechanism requires mechanisms for data transfer and synchronization.

8. In a monitor, mutual exclusion is achieved by: (a) allowing only one process to be active inside the monitor; (b) forcing a process to leave immediately if another is trying to enter; (c) allowing processes to wait and signal other processes; (d) forcing a process to wait if another is trying to enter.

(a) Mutual exclusion means only one process active (running or able to run) at the exclusion of all others.

9. Synchronization is achieved in message passing by: (a) not allowing a process to call receive until another process has called send; (b) allowing a process to call receive only after another process has called send; (c) making a process that calls receive to wait until the message is sent; (d) making a sending process check whether another process is waiting to receive a message.

(c) Synchronization (arranging for events to occur at the same time) is effected by having one process wait for another. Since a process cannot receive a message that was not yet sent, it must wait, and only the operating system can have it wait until the message is sent.

10. When evaluated based on turnaround time, Shortest Process Next (SPN): (a) is always better than First-Come-First-Served (FCFS); (b) is never worse than Shortest-Remaining-Time (SRT); (c) may be better or worse than FCFS; (d) may be better or worse than Round-Robin (RR).

(d) SPN is the best possible non-preemptive scheduling algorithm. FCFS is also non-preemptive, but if all processs arrive in order of shortest to longest, FCFS will do just as well as SPN, so: (a) is incorrect (since it says SPN is ALWAYS better). However, SPN can never do worse than FCFS, so (c) is incorrect. SRT is the best possible preemptive scheduling algorithm, and so SPN, which is non-preemtive, may in fact do worse, so (b) is incorrect. Since RR is preemptive and can therefore adapt to new arrivals better than SPN, it may do better; or, it may do worse, making (d) the correct answer.

11. If processes are executed in shortest-to-longest order, what is minimized (that might not happen with other orders): (a) average time spent using the CPU; (b) total time before all complete; (c) average time spent waiting for the CPU; (d) total time the CPU is idle.

(c) Shortest-to-longest minimizes average turnaround time, where turnaround time is completion or departure time - arrival time, part of which is time using the CPU and part is waiting. Since the time using the CPU cannot be changed, it must be the waiting time that is being reduced. Note that total time stays the same no matter what order, so in a sense, all total times are minimal; but since the question says "that might not happen with other orders", (c) is the best choice.

12. Which is a condition for mutual exclusion: (a) semaphores must be used to guard critical sections; (b) Receive must be called before Send when using message passing; (c) a process outside a critical

section cannot cause a process about to enter a critical section to block; (d) Signal must be the last statement in a monitor procedure.

(c) This is right out of the lecture notes or book. The other statements are relevant to synchronization, but only (c) is a condition for mutual exclusion.

13. When a process calls Wait(sem), which of the following cannot happen: (a) a blocked process becomes ready; (b) the calling process blocks; (c) a deadlock occurs; (d) the calling process does not block.

(a) Calling wait cannot possibly wake up a blocked process. Calling wait CAN possibly cause the calling process to block (if sem = 0), CAN possibly cause a deadlock (c) (if sem = 0 and Signal(sem) is never called), or CAN possibly cause the calling process to not block (if sem > 0).

14. In the Banker's algorithm, a safe state means: (a) deadlock can definitely be avoided; (b) there are no guarantees that deadlock can be avoided; (c) one or more resources are available; (d) all resources are currently being used.

(a) This is the definition of a safe state, and none of the other choices make sense.

15. Peterson's solution: (a) can't be used for mutual exclusion; (b) can be used for mutual exclusion, but requires busy waiting; (c) can be used for mutual exclusion and avoids busy waiting; (d) is more efficient than using test-and-set.

(b) Peterson's solution is a software solution to mutual exclusion, and explicitly uses busy waiting to test its state variables.

16. Monitors require programming language (compiler) support to implement: (a) producer/consumer; (b) mutual exclusion; (c) context switching; (d) deadlock prevention.

(b) Calling a monitor procedure causes the monitor lock to be obtained, the point of which is to support mutual exclusion. This occurs automatically by simply calling the procedure, and therefore it must be the compiler that generates the instructions because only the compiler recognizes the monitor programming language construct.

17. To support timesharing, the following must be supported by the kernel: (a) the Yield system call; (b) preemption; (c) first-come-first-served scheduling; (d) semaphores.

(b) Timesharing means that each process repeatedly gets a slice of time, or quantum. At the end of the quantum, it's time for another process to get a turn, which means the CPU needs to be taken away from the current process. This is what preemption is, forcibly taking away a resource.

18. To implement user-level threads: (a) the process stack must contain a stack per thread; (b) context switching between threads must be supported by the kernel; (c) thread scheduling is done by the kernel; (d) all of the above.

(a) Each thread needs its own stack to maintain activation records of its pending procedure calls. These stacks must be somewhere. Since these are user-level threads, the kernel doesn't know anything about them, and so (b) and (c) cannot be true, and the stacks must be in the process's memory.

19. In PA2, DoSched is used to: (a) cause the kernel to make a scheduling decision at the next opportune time; (b) determine which process should run next; (c) causes SchedProc to be called immediately; (d) respond to a timer interrupt.

(a) DoSched will cause the kernel to make a scheduling decision at the next opportune time, at which point SchedProc will be called to determine which process to select.

20. In PA2, a call to RequestCPUrate(n) MAY (i.e., it is possible that it will) be denied if: (a) n < 100; (b) n > 100; (c) n = 100; (d) all of the above.

(d) all of the above. It should definitely fail for n > 100. And while n < 100 and n = 100 are potentially valid requests (as long as n > 0), if allowing such a request causes the CPU to be over-allocated, the call will fail.

21. One necessary condition satisfying mutual exclusion is: (a) no process outside its critical section may block other processes; (b) a process inside its critical section causes all other processes to block; (c) a process may cause another process in its critical section to block; (d) no process inside its critical section may block other processes outside their critical sections.

(a) This is one of the four conditions for mutual exclusion. The others are not.

22. When comparing context switching time C to the length of a quantum L, typically: (a) C << L; (b) C ≈ L; (c) C >> L; (d) can be any of the above.

(a) Typically, C is at most 1% of L, otherwise the overhead of switching starts to become too high.

23. Which of the following types of variables will NOT be inside an activation record: (a) only local; (b) only static local; (c) both local and static local; (d) neither local nor static local.

(b) If a variable is declared static local, it goes into the data area of a process and not the stack area, whereas a local variable goes in the stack area where activation records are located.

24. Given 4 processes with PIDs 1, 2, 3, 4, that arrive at the same time, which of the following sequences of SchedProc return values would be valid in ROUNDROBIN Scheduling: (a) 1234123412341234; (b) 1122334411223344; (c) 1111222233334444; (d) all the above.

(a) This is the only sequence that gives every process a chance to run within a 4-quantum cycle.

25. In Multi-Level Feedback Queues, if a process is preempted by the arrival of a higher priority process and does not use the CPU time it was allocated, it will: (a) go to the next lower queue (assuming there is one); (b) go to the queue that it came from; (c) go to the next higher queue (assuming there is one); (d) any of the above may occur.

(b) If a process is preempted because it did not get to use what it was given, it is neither promoted or demoted.

26. If two periodic processes A and B are being scheduled using Rate Monotonic Scheduling (RMS), where A's utilization is 30% and B's utilization is 35%: (a) all deadlines will be met; (b) no deadlines will be met; (c) can't say whether all deadlines will be met; (d) depends on the periods.

(a) RMS says that periodic processes will definitely meet all deadlines if the utilizations sum to n*(2^(1/n) - 1), and the SMALLEST this bound can be is ~69%. Since the utilizations in this case sum to 65%, all deadlines will be met.

27. Which of the following is NOT true about Earliest Deadline First (EDF): (a) can achieve 100% utilization (ignoring overhead); (b) does not require preemption; (c) may require significant work, O(n) where n is the number of processes in the system, at every scheduling decision point; (d) works for aperiodic processes.

(b) EDF DOES require preemption. At a process's deadline, its new deadline must be considered as to whether it is now the earliest deadline, and if it is, the currently running process would need to be preempted. All the other selections are true.

28. In PA2, a call to HandleTimerIntr: (a) will definitely lead to a call to SchedProc; (b) may occur in the middle of a quantum; (c) may cause a process to exit; (d) none of the above.

(d) Selections (a), (b) and (c) are all false. For (a), HandleTimerIntr does NOT necessarily lead to a call to SchedProc; for (b), HandleTimerIntr is called at the END of a quantum; for (c), only a call to Exit or reaching the end of its code (which causes an implicit call to Exit), cause a process to exit, and NOT the calling of HandleTimerIntr.

29. In PA2, Gettime can be used to implement: (a) ROUNDROBIN; (b) PROPORTIONAL; (c) FIFO; (d) none of the above.

(d) Gettime is a system call, and system calls cannot be called from within the kernel, so it cannot be used (and there is no reason to use it, even if it could be used).

30. In Stride Scheduling, the pass value is calculated as: (a) previous pass value + stride value; (b) L divided by the stride value; (c) 1 divided by the request value; (d) stride value + request value.

(a) After a process has received a quantum, its pass value is increased by the stride value. The others make no sense.

31. Say the contents of mem is 0, and an interrupt occurs during the execution of TSL(mem). After the instruction has executed, the contents of mem will be (a) 0; (b) 1; (c) cannot determine; (d) none of the above.

(b) TSL(mem) ALWAYS sets the contents of mem to 1, regardless of its previous value, and since it is atomic, an interrupt does not affect it.

32. Which of the following transitions is invalid: (a) RUNNING to READY; (b) READY to RUNNING; (c) BLOCKED to READY; (d) BLOCKED TO RUNNING.

(d) When a block process is woken up, this is done because the condition it was waiting for (e.g., a resource becoming available) becomes true (which causes the process to become eligible to get the CPU), which is separate from a scheduling decision (which is what causes a process to actually get the CPU).

33. Which of the following causes control to transfer to the kernel: (a) executing a TSL instruction; (b) calling a procedure; (c) accessing a shared variable; (d) none of the above.

(d) All of the other choices do not involve the kernel (and in particular, don't confuse TSL with TRAP, where the latter does cause transfer to the kernel).

34. If a program variable is declared as a static global integer, when that program runs as a process, the memory will be allocated (a) in the kernel's stack for that process; (b) in the process's data segment; (c) in the process's stack segment; (d) in shared memory.

(b) The data segment in user space is where all variables that are either global, or static, or both, because the variable must exist for the entire lifetime of the process (in contrast to variables that exist solely during the lifetime of a procedure/function, and thus could be allocated on the stack).

35. Which of the following will benefit the least from multiple CPUs: (a) a single process with multiple user-level threads; (b) a single process with multiple kernel-level threads; (c) multiple processes, each with a single user-level thread; (d) multiple processes, each with a single kernel-level thread.

(a) If the kernel does not support threads (and so threads are implemented at user level), since the kernel is "unaware" of them, it cannot assign them to different CPUs.

36. In PA2, which of the following functions returns the process that should run next: (a) DoSched; (b) HandleTimerIntr; (c) MyRequestCPUrate; (d) SchedProc.

(d) This is simply the definition of SchedProc, which when called, will return which process to schedule next.

37. Which of the following need to be surrounded by the addition of <entry> and <exit> code to achieve mutual exclusion: (a) Wait operation in semaphores; (b) Wait operation in monitors; (c) while loop test in Peterson's solution; (d) all of the above.

(a) The body of Wait in semaphores is a critical section requiring mutual exclusion, because if two processes were to try to update the semaphore value within Wait, the results are non-deterministic. In monitors, the monitor lock guarantees atomicity of the body of the monitor, and since the Wait operation is inside the body, it already runs atomically. In Peterson's solution, the entry code that contains the while loop makes no assumptions about running atomically; in fact, it is what is used to implement atomicity.

38. When is a context switch NEVER allowed to occur under any circumstances whatsoever: (a) while executing in the kernel; (b) while executing in a critical section; (c) while executing in code that is expected to run atomically; (d) none of the above.

(d) The key principle here is, IF a context switch will not cause a problem, we never want to prevent it from happening. Consider each choice: if a process is executing in the kernel, and there were another process that could be switch to and would be executing outside the kernel, there's no reason to prevent the context switch. Even IF that other process were to enter the kernel, as long as the code it executes doesn't interfere (i.e., no race conditions) with the first process, it should be allowed to run. Same idea with a process executing in a critical section; if another process is allowed to run code that is independent of the first process's critical section, it can run without any problems, so we should allow it. Same for a process running code that is expected to run atomically; if another process is allowed to run code that is independent of the atomic code the first process is running, it can run without any problems, so we should allow it.

39. In Proportional Share scheduling, at each quantum, the kernel always selects the process that (a) got the least amount of CPU time so far; (b) has the maximum requested fraction of CPU time; (c) has the minimum ratio of requested fraction of CPU time to received fraction of CPU time; (d) none of the above.

(d) What matters in Proportional Share is meeting the requests of all processes, and so at each quantum, the kernel checks which process has gotten the LEAST of what it expects, i.e., got the minimum ratio (compared to all other requesting processes) of received (actual) fraction of CPU time to requested fraction of CPU time (minimum RECEIVED to REQUESTED, rather than minimum ratio of REQUESTED to RECEIVED, that's why (c) is incorrect). This is best corrected by giving that process the CPU for one quantum, and re-evaluating.

40. When a process enters the kernel, we say that there is an "amplification of power" in that that process now has access to all the kernel data structures and can execute both non-privileged and privileged CPU instructions (whereas outside the kernel, the process can only execute non-privileged instructions). Assuming a process is running its own code (not kernel code), which of the following causes such an amplification in power: (a) anything that causes a switch from user mode to kernel mode; (b) executing a TRAP instruction; (c) responding to a hardware interrupt; (d) all of the above.

(d) All of the choices cause a process to enter the kernel, which results in an amplification in power.