Scoring:for each question • *100% if rightt* • *-50% if wrong* • *0 if unanswered*	*UO:* *Full name:*	*Model 1*

True / False questions

IMPORTANT: This test consists of true/false questions grouped in blocks of four. However, each question value is independent from the other three. You are expected to answer each question independently, as true or false.

[System booting]

1. The ROM starter creates the data structures of the operating system.
2. The boot located in the boot sector is provided by the hardware manufacturer.
3. *The ROM starter is provided by the hardware manufacturer.
4. The hardware test is done by the boot.

[Operating system activation]

5. OS is activated whenever a process tries to access a main memory address.
6. *OS is activated whenever a process synchronizes with another one on the same machine.
7. *OS is activated whenever an execution error such as “null pointer exception” occurs.
8. *OS is activated whenever the processor receives an external interruption.

[Operating system activation]

9. *System calls are API functions that include some instruction that cause a jump to the OS.
10. *When the OS is activated the processor enters kernel mode execution.
11. *In a system call, the calling process puts the parameters in the processor registers for the OS to receive them.
12. External devices communicate with the OS through system calls.

[Interface with the operating system]

13. Programs communicate with every OS services through shell commands.
14. *Graphical interfaces provided by the systems are actually programs that perform system calls to ask services from the OS.
15. *Command line interfaces provided by the systems are actually programs that perform system calls to ask services from the OS.
16. The libraries that implement the POSIX standard contain functions that run in kernel mode.

[Operating systems evolution]

17. The first control programs, ancestors of OS, appear in the 1960s and are called "resident monitors".
18. *Multiprogramming and time sharing began to be developed in the 1960s.
19. *Ken Thompson y Dennis Ritchie designed UNIX and implemented it in C language.

20. Ada Lovelace designed an OS for the first machines of her time.

[Multiprogramming]

21. *One of the foundations of multitasking is the simultaneous execution of the processor and the I/O (interchange of information between memory and devices).
22. *One of the foundations of multitasking is sharing processor usage among several processes.
23. The Process Control Block is stored in memory inside the process image.
24. *The PCB of a "ready" process contains the values that the processor registers had the last time that process left the CPU.

[Process lifecycle] 7-state model

25. "Ready" processes will go to "blocked" if some event occurs.
26. There is only one queue with all the processes that are "blocked" waiting no matter what event.
27. *"Ready" and "blocked" process queues contain PCBs (or pointers to them).
28. Un proceso en estado bloqueado suspendido puede pasar a ejecución en cuanto lo carguen en memoria principal.

[Interruption handling] Suppose an OS executed in the process space address (in-process execution)

29. When an interruption is raised, hardware stores all the processor registers in the stack.
30. *When an interruption is raised, hardware changes the processor from user to kernel mode.
31. *When an interruption is raised, hardware substitutes the stack pointers of the user process by the control stack pointer of the operating system.
32. *When an interruption that implies a process switching is raised, the system takes all the entries contained in the control stack, and stores them in the PCB of the process that is leaving the processor.

[Context switching]

33. In an OS that runs in the user process address space, each and every interruption produces a context switch.
34. In an OS that runs in the user process address space, each and every interruption produces a process switch.
35. *In an OS that runs in the user process address space, the context switching is done only if there is a process switch.
36. *In an OS that runs in the user process address space, a context switching implies storing the processor registers values in the PCB of the process leaving the CPU, and getting the processor registers values from the PCB of the process entering the CPU.

[Threads]

37. *The threads belonging to the same process share the address space of that process.
38. *Each one of the threads belonging to the same process has its own stack.
39. The threads belonging to the same process share the values of the processor registers stored in the PCB.

40. User-level-implemented threads allows for the simultaneous execution of threads in different processors.

[Scheduling policies]

41. *In an OS supporting threads, it's each individual thread who changes state, not the process as a whole.
42. Scheduling policies based on dynamic priorities with aging can cause starvation.
43. *A system with rotation shift (round robin) for all the processes favours equality in processor use.
44. * Windows uses a policy with dynamic priorities in which processes with much I/O are favoured.

[Process scheduling]

45. The preemptive scheduling policy allows to temporarily exchange processes to secondary memory
46. *The short-term planning policy aims to optimize system performance parameters in terms of processes execution speed
47. In a system with Round Robin planning policy, a process leaves the CPU in only two cases: when the time quantum ends or when the process ends
48. A policy priority scheduling with aging attempts to prioritize processes with more I/O than processes with more CPU consumption.

[Multiprocessor scheduling]

49. *Current OS usually use symmetric multiprocessing and run in all the processors.
50. *In a multiprocessor system with unique queue, load balancing is favoured.
51. *In a multiprocessor system with several queues, processor affinity is favoured.
52. *Using a single queue in systems with a heavy load can cause a bottleneck.

[Concurrency]

53. If you need to communicate processes on the same machine, it's more efficient to use threads from the same process than from different processes.
54. If you need to communicate processes on different machines, it's more efficient to use threads from the same process than from different processes.
55. *Independent concurrent processes can compete for a common resource
56. Concurrent programming is becoming less and less necessary because of processor speed increase.

[Concurrency] Let there be two threads using a shared variable to store information that both of them need, such that one of them shouldn't read the information until the other has written the data on it.

57. This is a "readers and writers" problem.
58. *This is a "producer-consumer" problem.
59. This problem is a "client-server" problem.
60. This problem is a "shared resource, or philosophers", problem.

[Concurrency]

61. *The race conditions problem that can arise in some concurrent programming situations should be avoided by means of proper synchronization between the involved threads or processes.
62. *The deadlock problem that can arise in some concurrent programming situations should be avoided by removing any of the 4 necessary conditions for deadlock.
63. Critical section is a code fragment shared by all the processes or threads that cooperate.
64. *Mutual exclusion implies executing the critical section of each process or thread in an atomic manner.

[Communication mechanisms]

65. Pipes and shared memory are communication mechanisms suitable for processes located in different machines.
66. *There is no strict need of a communication mechanism managed by the OS if all the involved threads belong to the same process.
67. *Mutexes and semaphores (on their own) are useful for synchronizing but not for communicating.
68. *Conditional variables can avoid deadlocks.

[Semamphores and mutexes]

69. *If a thread executes a lock(m) on a mutex initially set to 0, the OS changes the thread to the blocked state.
70. If a process executes a wait(s) on a semaphore initially set to 1, the OS changes that process to the blocked state.
71. A thread blocked in the queue of a mutex can be un-blocked by a thread belonging to a different process.
72. *A process blocked in the queue of a semaphore can be un-blocked by another process which have access to that semaphore.

[Signals]

73. *POSIX standard defines signals as a synchronization mechanism but Windows API does not.
74. *Signals can be synchronous or asynchronous.
75. *Kill is a system call that sends a signal to a process.
76. Signals are useful for both communication and synchronization.

[Pipes]

77. *Pipes are useful to solve producer/consumer problems
78. Pipes are useful only to communicate information, but an additional synchronization mechanism is needed.
79. Pipes are useful to synchronize processes in different machines.
80. Pipes are useful to communicate processes in different machines.

[Scheduling policies] Pure round robin, Q=2, process A starts at 0 and needs 6, B 1/3, C 3/8, D 5/2, E 7/3 (notice that starting times are not correlative)

81. *Te for process C is 19.

82. Tw for process B is 6.
83. *Tw for process E is 8.
84. *Te for process A is 13.

[Context switching] Given the situation in the accompanying sheet:

85. *Some interruption is being handled right now.
86. *Process number 18721 is going to be executed.
87. * Process 377 was interrupted when its PC was at position 003884.
88. This OS runs in independent kernel mode.

[System calls for process creation] Given this code with fork and exec calls, and a process P that runs this code

```
int count = 5;
if (fork() == 0)
{
    exec("executable file");
    printf("%d ", count);
}
else
{
    for (int i=0; i<2; i++)
       {
       fork();
       count ++;
       printf("%d ", count);
       }

}
```

89. *From the process P that runs this code, 4 more process are generated (not counting P)
90. A possible console output is "5 6 6 7 7 7 7"
91. All the processes generated are children of P
92. *A possible console output is "6 7 7 6 7 7"

[Concurrency] We want to synchronize two processes so that one writes "Hello, how are you?" in a file, and the other reads this text and prints it on the console.

```
fd file;
char buffer[128];

file = fopen("r+", "file.dat");
if (fork() != 0){
       wait(sem1);
       buffer = "Hello, how are you?";
       write(file,buffer);
       signal(sem1);
       signal(sem2);
}
else{
       wait(sem2);
       read(file,buffer);
       printf(buffer);

}
```

93. *We should initialize sem2 semaphore to 0, and sem1 to 1, in order to get the desired result.
94. *If we delete the sem2 semaphore, and initialize sem2 to 0, we get the desired result.

95. The child process needs to add signal(sem1) at the end for the synchronization to be correct.
96. *If we substitute the file for a pipe, and the read operation reads from the pipe, and the write operation writes on the pipe, we would need no semaphore to achieve the synchronization.

[Concurrency] Given, these two processes in an assembly line for a cider factory (executing in two different threads):

```
void fillBottle(){
       repeat{
              wait(lane);
              bottle = new Bottle();
              bottle.fillContents();
              corkLane.put(bottle);
              signal(filled);
       }
}
void corkBottle(){
       repeat{
              wait(filled);
              bottle = corkLane.getBottle();
              bottle.putCork();
              tagLane.put(bottle);
              signal(lane);
       }
}
```

97. *If both processes have an initial value of 0, then deadlock happens
98. If both processes have an initial value of 0, then deadlock happens
99. If lane is initialized to 10, and filled to 0, until the first 10 bottles are corked, no more bottles can be filled.
100. *If lane is initialized to 10, and filled to 0, there will be a maximum of 10 filled bottles without cork