* Puede quedarse con este impreso.

* Rellene sus datos personales en la hoja de respuestas múltiples, incluyendo expresar “con marcas” el DNI.

* Responda en la hoja de respuestas asignando **A=VERDADERO; B=FALSO.**

* **Puntución:** 3,5 puntos en total. Hay 35 item, cada acierto vale 0,1 puntos; cada fallo penaliza con 0,05 puntos (la mitad de un acierto); no contestar no penaliza.

* Si necesita hacer algún comentario a algún item, puede expresarlo en la cara en blanco por detrás de la hoja de respuestas múltiples, o en un folio en blanco.

Supuesto 1) Tenemos estos tres script bash de nombres s1, s2 y s3:

s1	s2	s3

`lim=1000000`
`for ((C=1;C<lim;C++));do`
 	
`## lanzamos la ejecucion de1 script`
`## llamado calculo1 que realiza un`
`## calculo aritmetico que dura 1 seg`
 	
`calculo1 ##`
`done`
`echo Fin s1 con pid $$`
	`lim=1000000`
	`for ((C=1;C<lim;C++));do`
	
	`  sleep 1`
	
	`done`
	`echo Fin s2 con pid $$`
		`lim=1000000`
		
		`sleep $lim`
		
		`echo Fin s3 con pid $$`

En relación a estos tres script tenemos los siguientes enunciados (1 a 7):

1. V	El proceso resultante de ejecutar s1 es un proceso “limitado por CPU”
2. V	El proceso resultante de ejecutar s2 es un proceso “limitado por E/S” `La orden sleep provoca el bloqueo del proceso, por lo que éste tiene ráfagas cortas.`
3. F	El proceso s3 tiene ráfagas de CPU cortas.
4. F	Si como únicos procesos tenemos varias ejecuciones simultáneas de **s1**, la cola de ejecutables estará vacía la mayor parte del tiempo. `Un proceso resultante de ejecutar s1 no se bloquea nunca, por tanto siempre existe como proceso ejecutable`
5. V	Si como únicos procesos tenemos varias ejecuciones simultáneas de **s3**, la cola de ejecutables estará vacía la mayor parte del tiempo.
6. V	Si como únicos procesos tenemos varias ejecuciones simultáneas de **s3 con distintas prioridades**, el hecho de que tengan distintas prioridades no va a repercutir visiblemente en cómo van progresando en su ejecución.
7. F	Si como únicos procesos tenemos varias ejecuciones simultáneas de **s1 con distintas prioridades**, el hecho de que tengan distintas prioridades no va a repercutir visiblemente en cómo van progresando en su ejecución.

SOBRE IMPLEMENTACION EN LINUX

8. V	En el kernel 2.6 una tarea con la política de planificación FIFO podría ser expulsada por otra de mayor prioridad (mayor importancia).

9. F	***task_tick*** provoca incondicionalmente la ejecución de ***schedule***
10. V	Cuando se va a hacer una apropiación en modo usuario, es seguro que todas las estructuras del núcleo están en un estado coherente, visible, no a mitad de una actualización.
11. F	En un kernel no apropiativo podemos expulsar a un proceso en cualquier punto de su ejecución, tanto si está ejecutando en modo usuario o en modo kernel
12. V	Cuando una tarea entra en estado zombie ya no puede volver a estar nunca en estado ejecutable.
13. F	La filosofía CFS no se puede implementar en ningún sentido si hay procesos de distintas prioridades.
14. F	Los valores de preempt_count de distintos procesos podrían ser simultáneamente > 0
15. V	Todo proceso cuando termina pasa por el estado EXIT_ZOMBIE
16. V	El valor de retorno que un proceso hijo pasa al padre se almacena en la task_struct del hijo

17. V	El siguiente fragmento de código dentro del kernel muestra en pantalla el pid del proceso actual y de sus ancestros:
	`struct task_struct *task;`
	`for (task = current; task != &init_task; task = task->parent)`
	`{printk("%s[%d]\n", task->pid}`

18. F	En el anterior fragmento de código dentro del kernel, al terminar task apunta a la task_struct del proceso actual.
19. V	Cuando un proceso lanza varias hebras, se crea una task_struct para cada una de ellas.
20. F	Si un proceso padre que aún tiene hijos existentes llega a su fin, se produce la terminación automática de cada uno de sus hijos.
21. F	Justo en el instante en que el flag TIF_NEED_RESCHED cambia a estado establecido se produce una llamada a la función ***schedule*** para que se estudie si cambiar la asignaciónde CPU a otro proceso.
22. V	El método de planificación periódico de la clase de planificación a que corresonde el proceso actual es activado por el planificador periódico scheduler_tick.
23. F	Dentro de las funciones definidas en un programa, cada vez que se retorna de una función (en el espacio de usuario) se chequea el flag TIF_NEED_RESCHED
24. F	En CFS, a mayor valor de latencia hay un mayor coste de tiempo de CPU en realizar cambios de contexto.
25. V	El valor de vruntime de todos los procesos de la clase CFS se actualiza siempre que haya que reelegir un nuevo proceso.
26. F	En un kernel apropiativo, si llega un proceso con más preferencia que el actual para disfrutar de CPU, se retira la CPU al actual esté haciendo lo que esté haciendo
27. F	Mientras un proceso está ejecutando código de usuario podría tener el flag preempt_count a un valor >0.
28. V	En SMP es crucial minimizar el coste de migrar procesos de una CPU a otra.
29. V	Cuando un proceso realiza una operación de E/S que implica bloqueo se ve sometido a la transición que en la figura aparece entre TASK_RUNNING y TASK_INTERRUPTIBLE (o TASK_UNINTERRUPTIBLE)
30. F	Un proceso privado de memoria estaría en el estado TASK_RUNNING
31. F	Situándonos en el diagrama de transiciones entre procesos en Linux, cuando un proceso invoca a una llamada al sistema que no suponga bloqueo no sufre ninguna transición que se muestre en dicho diagrama.
32. F	Un proceso en estado TASK_INTERRUPTIBLE o TASK_UNINTERRUPTIBLE puede pasar directamente a estado zombie.
33. V	Como consecuencia de la ejecución de una interrupción software se crea un marco nuevo en la pila kernel.
34. V	Como consecuencia de la ejecución de una interrupción hardware se crea un marco nuevo en la pila kernel.
35. V	El planificador principal va recorriendo las clases de planificación en orden de mayor a menor importancia encontrando la primera que no esté vacía, y el método de planificación asociado determina el siguiente proceso a ejecutar