UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

Fundamentos de los Sistemas Operativos
Examen parcial - 9 de abril de 2018

1	2	3	test	extra	NOTA

Nombre y apellidos **DNI/NIE**

SOLUCIONES

DURACIÓN: *Dispone de dos horas para realizar el examen.*

Lea las instrucciones para el test en la hoja correspondiente.

1 (1'25 puntos) A un planificador de CPU llegan tres procesos, según el cuadro adjunto. Aplique las dos políticas SRTF (SJF expulsivo) y Round Robin (Q=2) y, para cada una de ellas, obtenga lo siguiente:

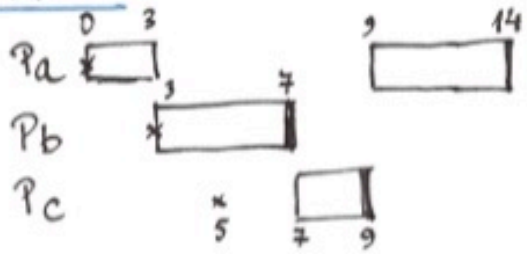
proceso	llegada	duración
Pa	0	8
Pb	3	4
Pc	5	2

- Diagrama de Gantt o similar con la planificación.
- Tiempo de espera y de retorno de cada uno de los procesos.
- Número de cambios de contexto realizados durante la planificación.

Ver modelo de respuesta en la próxima página

SRTF

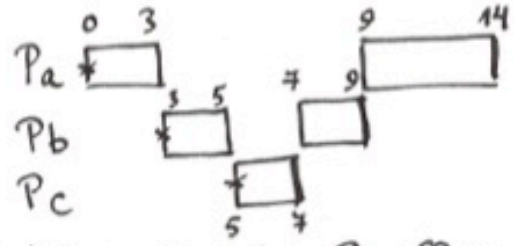
ⓐ



	tesp	tret
Pa	6	14
Pb	0	4
Pc	2	4

cambios contexto = 3

ⓑ



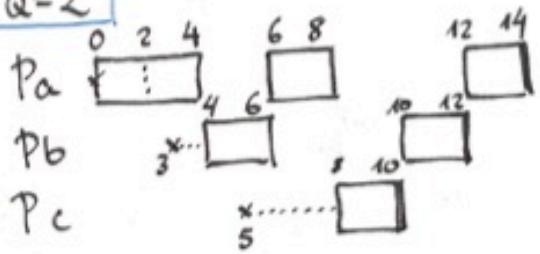
	tesp	tret
Pa	6	14
Pb	2	6
Pc	0	2

cambios contexto = 4

- La diferencia entre ⓐ y ⓑ es qué ocurre cuando llega Pc, en t=5. a Pb le quedan 2 u.t., que es la misma duración de Pc.

RR Q=2

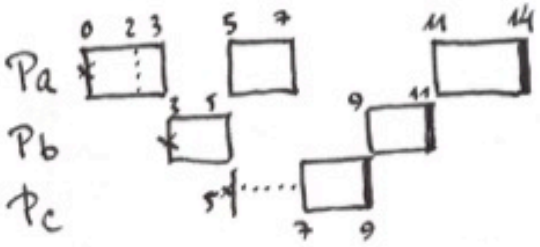
ⓐ



	tesp	tret
Pa	6	14
Pb	5	9
Pc	3	5

cambios contexto = 5

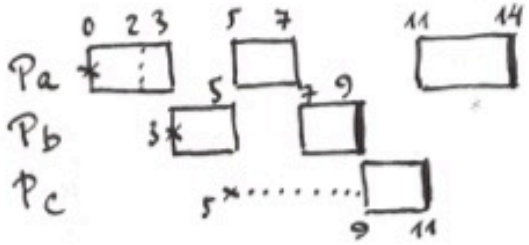
ⓑ



	tesp	tret
Pa	6	14
Pb	4	8
Pc	2	4

cambios contexto = 5

ⓒ



	tesp	tret
Pa	6	14
Pb	2	6
Pc	4	6

cambios contexto = 5

- en ⓑ y ⓒ se considera que Pa está en 'prórroga' por haber agotado Q (t=2) y no haber ningún otro proceso en espera. En t=3 se le expulsa para que entre Pb. La diferencia entre ⓑ y ⓒ es si consideran que Pc llega antes o después del momento en que se agota Q.

Nombre y apellidos	DNI/NIE

2 (1'50 puntos) Responda a estas dos cuestiones, empleando hasta 150 palabras para cada una:

a) ¿Cuál es la diferencia entre un sistema de tiempo compartido y un sistema de tiempo real? Ponga algún ejemplo práctico y realista en el que un sistema de tiempo compartido no puede resolver un requisito de tiempo real.

El objetivo central de un sistema de tiempo compartido es repartir la CPU entre los procesos de manera que den *una experiencia interactiva a los usuarios*, mientras que un sistema de tiempo real trata de *garantizar que cada proceso finalice en un plazo de ejecución determinado*. La cuestión es que un sistema de tiempo compartido no está concebido para *garantizar* plazos de finalización: no da prioridad a los procesos que tienen más urgencia por finalizar.

En esta pregunta se puede responder con ejemplos en los que un sistema de tiempo compartido fracasa con una exigencia de tiempo real, si el sistema está muy cargado y un proceso de tiempo real no disfruta de suficientes rodajas de CPU para poder acabar a tiempo. Por ejemplo en un Round Robin con Q=1 milisegundo, una tarea que debe finalizar en 2 milisegundos se mete en la cola de preparados junto con otros 5 procesos. Esta tarea no tiene garantizado finalizar en el plazo marcado. Podríamos aplicar este ejemplo general a múltiples situaciones reales: reproducción de archivos multimedia, ejecución de un proceso crítico en respuesta a un evento de E/S (ej. alarma contra incendios, en un automóvil explosionar el airbag ante una deceleración brusca, en un portátil pasar a modo hibernación cuando la batería alerta de que está a punto de agotarse, etc.).

b) Describa cómo puede protegerse la zona de memoria del núcleo mediante los registros base y límite.

Aquí hay que explicar el mecanismo de los registros base y límite y cómo se puede aplicar para delimitar la zona de memoria a la que puede acceder un proceso de usuario. En la respuesta hay que dejar claro que la CPU distingue un modo privilegiado de un modo usuario en el cual se controlan los accesos a memoria; que el control se realiza mediante hardware; y que el sistema debe dar valores a estos registros para que se obligue al proceso de usuario a acceder a una zona determinada.

3 (1'25 puntos) Se propone el siguiente algoritmo para resolver el problema de sección crítica para dos procesos:

```
// Variables compartidas
bool flag[2] = { false, false };
```

```
// Código del proceso 0
while (true) {

      ... código de sección no crítica

      flag[0]=true;
      while (flag[1]) {
            flag[0]=false;
            sleep(1);
            flag[0]=true;
      }

      ... código de sección crítica

      flag[0]=false;
}
```

```
// Código del proceso 1
while (true) {

      ... código de sección no crítica

      flag[1]=true;
      while (flag[0]) {
            flag[1]=false;
            sleep(1);
            flag[1]=true;
      }

      ... código de sección crítica

      flag[1]=false;
}
```

¿Se trata de una solución válida para el problema? ¿Por qué? Justifíquelo en relación con las propiedades que debe cumplir un algoritmo válido para el problema de la sección crítica.

¿Este algoritmo puede mejorarse de alguna forma?

En este algoritmo cada proceso «i» que quiere entrar en sección crítica declara primero que nada su intención de acceder, poniendo a TRUE la variable flag[i]. Cuando abandona la sección crítica, vuelve a poner flag[i] a FALSE. El bucle sirve para detener al proceso «i» si este observa que su compañero tiene intención de entrar en sección crítica: esto se hace observando el valor de flag[] para el otro proceso. Si observa que el compañero quiere entrar en sección crítica, el proceso «i» mantiene durante un segundo su flag[i] a FALSE y al cabo de ese tiempo vuelve a intentar entrar.

El algoritmo planteado asegura la **exclusión mutua** entre ambos procesos, ya que ninguno de los dos avanza a su sección crítica si observa que el otro ha declarado su interés en entrar. Sin embargo, cuando los dos procesos quieren entrar al mismo tiempo, puede ocurrir que ambos se queden atascados en un bucle infinito (*livelock*), poniendo a FALSE y TRUE sus respectivos flags y observándose mutuamente como interesados en entrar en la sección crítica. Por tanto no se garantiza el **progreso** del algoritmo, aunque la situación descrita es poco probable: en un sistema real, lo más probable es que alguno de los dos procesos acabe entrando.

Por último, hay que destacar que el algoritmo no garantiza la **espera limitada** de un proceso. Supongamos que el proceso 0 está dentro de la sección crítica y que el proceso 1 quiere entrar. El proceso 1 se mantiene en el bucle de espera. Supongamos que mientras el proceso 1 está en el «sleep», con su flag a FALSE, el proceso 0 abandona la sección crítica y vuelve de inmediato a solicitar entrar en sección crítica: podrá entrar, ya que flag[1] está a FALSE. Si el proceso 1 se despierta y vuelve a evaluar flag[0], observará que está a TRUE y volverá a esperar. Esta situación, teóricamente, se puede dar una y otra vez, sin garantía de que el proceso 1 vuelva a entrar en sección crítica.

El algoritmo se podría mejorar de muchas formas, por ejemplo obligando a que el sleep() fuera aleatorio (disminuiría la probabilidad de un atasco eterno o *livelock*), o bien utilizando una variable que deshaga el empate si los dos procesos están interesados en entrar (como ocurre en el algoritmo de Peterson), etc.