# Resolución propuesta - Parcial 09/05/2015 - K3013/ K3113

## Teoría

1) El cambio de modo se realiza de modo usuario a modo kernel, guardando el contexto del proceso y actualizando el bit de modo en el procesador. De esta forma se pueden ejecutar instrucciones privilegiadas. Para realizar el cambio de modo kernel a modo usuario, se vuelve a colocar el contexto del proceso que deba ejecutar y luego se actualiza el bit de modo nuevamente.
Si llegaran dos interrupciones mientras ejecuta un proceso, se debe pasar a modo kernel, atender ambas, y luego se vuelve a modo usuario para que el proceso siga ejecutando.

2) Hay varias opciones válidas. Aquí va una: se podría usar un algoritmo multinivel con una cola para llamadas (FIFO), otra para notificaciones y SMS (FIFO), y otra para aplicaciones en pantalla y aplicaciones en segundo plano (podría ser VRR, para que las aplicaciones en pantalla tengan mayor prioridad pero no sufran de inanición las de segundo plano). Los procesos entrarían en la cola correspondiente cuando son creados y cuando vuelven de E/S, por lo que no habría “feedback”. La prioridad sería llamadas > notificaciones > aplicaciones, y tendríamos desalojo entre colas (porque probablemente, si llega una llamada, no quiera esperar a que el quantum de la aplicación termine).

3) Deshabilitar las interrupciones
- Ventajas: no tiene espera activa
- Desventajas: lento para sistemas multiprocesador. El S.O. pierde el control durante un tiempo. Si un proceso se apoderara de la CPU no se lo podría desalojar

Test & Set / Swap & Exchange
- Ventajas: el S.O. nunca pierde el control.
- Desventajas: tiene espera activa

4)
a. Falso. Expropiar no es sencillo. Para hacerlo necesito un mecanismo que me permita volver a los procesos a un estado anterior (rollback) y luego reasignar los recursos. Matar procesos es más sencillo (aunque no necesariamente sea lo mejor, lo más performante, etc).
b. Verdadero. Previene la espera circular

5) Los procesos son mejores en términos de estabilidad y seguridad. Para el caso del web browser, podemos usar un proceso por pestaña si nos enfocamos en la estabilidad (si se cuelga una pestaña no se cuelgan las otras) o bien un proceso por dominio si nos enfocamos en la seguridad (esperamos que www.utn.so/faq no tenga problemas de seguridad con www.utn.so/campus por ej).

Bonus) Nuestras variables “clave” y “logueado” son variables locales y probablemente estén cercanas en el stack, Por otro lado, nuestra lectura usando el scanf no pone un límite, por lo que, si el stack no estuviese correctamente protegido, podríamos ingresar SSSSSSSSSS… y eventualmente sobreescribiríamos el contenido de “logueado”.

## Ejercicio 1

a)

Matriz de maximos

| | R1 | R2 | R3 |
|---|---|---|---|
| P1 | 2 | 2 | 0 |
| P2 | 2 | 0 | 2 |
| P3 | 2 | 2 | 1 |

Matriz de asignacion

| | R1 | R2 | R3 |
|---|---|---|---|
| P1 | 0 | 1 | 0 |
| P2 | 2 | 0 | 1 |
| P3 | 1 | 1 | 1 |

Matriz de necesidad

| | R1 | R2 | R3 |
|---|---|---|---|
| P1 | 2 | 1 | 0 |
| P2 | 0 | 0 | 1 |
| P3 | 1 | 1 | 0 |

DISP: 0, 0, 1

P2 fin.: 2, 0, 2

Nadie más puede finalizar = Estado inseguro

b) Puede haber varios ejemplos al respecto. Uno sería:

Caso donde ocurriría deadlock:

- P2 finaliza -> disponibles: 2, 0, 2
- P1 pide 1 R2 -> se bloquea
- P3 pide 1 R2 -> se bloquea (generando deadlock con P1)

Caso donde no ocurriría deadlock:

- P2 finaliza -> disponibles: 2, 0, 2
- P3 libera 1 R2 -> disponibles: 2, 1, 2
- P1 finaliza -> disponibles: 2, 2, 2
- P3 finaliza -> disponibles: 3, 2, 3

## Ejercicio 2

| | | Q=3 | | | | Q=3 | | | Q=3 | | | Q=3 | | | Q=3 | | | | | Q=3 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | | | | | | |
| A | C | C | C | | C | W | W | W | W | W | | | | W | W | W | | | W | W | W |
| B | | | | W | ■ | ■ | ■ | ■ | ■ | | T | P | P | | | | P | P | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |

C = Crear P = Procesar T = Turno W = While ■ = E/S

Esperas activas: Todos los ciclos while

Cambios de modo: 0 y 1 (proceso nuevo)
3, 7, 10, 13, 16, 21 (fin de Q)
9 (fin de E/S)
18 (fin de proceso B)

No hay tiempo ocioso. El proceso A sigue en un ciclo infinito

## Ejercicio 3

turnos [idCarrera][nro] = todos en 0 (o podrían ser 5 colas)
hayLugar[5] = {30,30, 30, 30,30};
mutex[5] = {1,1,1,1,1}
hayPendientes [5] = {0,0,0,0,0}

<table>
<tr><td>“Corredor” N</td><td rowspan="2">struct ticket {<br>  int idCarrera;<br>  int numero;<br>};<br>Nota: puede usar las variables de este tipo.</td></tr>
<tr><td>ticket t = generarTicket(datos);<br>wait(turnos[t.idCarrera][t.numero]);<br>entregarPlanilla();<br><br>signal(planilla[t.idCarrera])<br><br>wait(remera[t.idCarrera][t.turno])<br>probarseRemera();</td></tr>
<tr><td>Terminal de tickets (4)</td><td>Empleado (5)</td></tr>
<tr><td>while (true) {<br><br>datos = obtenerDatos();<br>ticket t = generarTicket();<br><br>wait(hayLugar[t.idCarrera])<br>wait(mutex[t.idCarrera])<br>encolarTicketEnColaCorrespondiente(t)<br>;<br>signal(mutex[t.idCarrera])<br><br>signal(hayPendientes[idCarrera])<br><br>}</td><td>int idCarrera = midCarrera();<br><br>while (true) {<br><br>wait(hayPendientes[idCarrera])<br>wait(mutex[idCarrera])<br>  ticket t = obtenerTicketDeColaCorrespondiente();<br>signal(mutex[idCarrera])<br><br>signal(hayLugar[t.idCarrera])<br><br>signal(turnos[t.idCarrera][t.numero]);<br>wait(planilla[t.idCarrera])<br><br>  revisarPlanilla();<br><br>  entregarRemera();<br>signal(remera[t.idCarrera][t.turno])<br><br>}</td></tr>
</table>