Teoría

1. Explique detalladamente las operaciones implicadas en un process switch ocurrido al terminar el quantum

Cuando finaliza el quantum:

1. Se lanza una interrupción por fin de quantum a nivel de HW
2. Al finalizar la instrucción actual se chequean si existen interrupciones pendientes.
3. Se cambia de modo U -> K
4. Se guarda el PC y el PSW (estado del proceso actual)
5. Se determina a qué se debe la interrupción. Se identifica cuál es el interrupt handler adecuado y se carga su instrucción en el PC
6. Se guarda el resto del estado del proceso actual y se lo coloca al final de la cola ready (se modifica su estado de running -> ready)
7. El planificador de corto plazo elige al siguiente proceso a ejecutar
8. Se pasa al proceso a modo running
9. Setea el contexto del procesador con los valores del nuevo proceso. Dentro de ellos el PSW (eso hace pasaje de K -> U) y el PC
10. La siguiente instrucción a ejecutar es la del nuevo proceso

2. Compare los algoritmos Virtual Round Robin y Round Robin usando los siguientes criterios: atención de procesos I/O bound, inanición y tiempo de respuesta promedio.
    - IO Bound: Virtual Round Robin los favorece, ya que cuando los procesos vuelven de E/S les otorga el quantum remanente y mayor prioridad. Round Robin trata a todos por igual.
    - Inanición: ninguno tiene. RR está basado en FIFO por lo que no sufre de inanición. VRR a pesar de tener una cola de mayor prioridad, a los procesos que se encuentran en la misma les da un quantum que tiende a cero, es decir, les da la parte de su quantum anterior que no consumieron, por lo que en algún momento se les acabará su remanente y serán desalojados a la cola RR de menor prioridad.
    - T.Rta promedio: Virtual Round Robin favorece a los IO Bound, con lo que el tiempo de respuesta mejora respecto a Round Robin.

3. Dé un ejemplo en pseudocódigo donde sea necesario utilizar el stack, el heap y la sección de datos del proceso. Indique la relación entre estos elementos y el código que escribió.
    - Hay varias alternativas. Una podría ser:

```
int global = 1; // datos
int main() { // va al stack
```

```
        int local = 2; // va al stack
        char* nombre = (char*) malloc(sizeof(char) * 10); // al heap
}
```

4. Elija dos condiciones necesarias para que ocurra deadlock e indique dos formas para prevenirlas (una para cada una). ¿En cuál situación sería preferible utilizar técnicas de Prevención en lugar de técnicas de Evasión?

Retención y espera: Para evitar esta condición, hay que garantizar que un proceso que está reteniendo recursos no pida nuevos y en el caso que pida, que no retenga otros recursos.
Una opción: Un proceso al iniciar, debe pedir todos los recursos que va a necesitar durante su ejecución (de esta forma al principio va a esperar todos sus recursos pero sin retener, y luego va a retener todos, pero sin pedir nuevos). Obviamente esto hace un mal uso de los recursos.
Otra opción: Un proceso cuando quiere pedir nuevos recursos, primero deberá liberar todos los que retiene. Entonces, si uno de los recursos que estaba usando lo sigue necesitando pero a su vez necesita uno nuevo, deberá liberar todos y volver a pedir el liberado + el nuevo.

Sin desalojo:
Una opción: cuando un proceso pide un recurso y el mismo no está disponible, se le expropian todos los recursos (ya que va a tener que esperar, que sus recursos puedan ser utilizados por los otros procesos)
Otra opción: cuando un proceso pide un recurso y el mismo no está disponible, si ese recurso está retenido por un proceso que está esperando por otro recurso, entonces se le expropia el mismo al segundo proceso y se le asigna al primero.

Espera circular:
Si se le asigna un orden numérico creciente a los recursos (siguiendo el orden normal en el que se suelen pedir) y luego se piden los mismos siguiendo dicho orden, se rompe con la espera circular.

Mutua exclusión:
Es la más difícil de evitar. Si un recurso se está utilizando únicamente en modo lectura, entonces no es necesario garantizar mútua exclusión (pueden tener un uso compartido).

Sería preferible utilizar Prevención frente a evasión en un entorno en el cual no se pueda agregar mucho overhead. En sistemas específicos en los que se sabe cómo se van a utilizar los recursos es sencillo definir una política de prevención que limite el acceso a los mismos permitiendo que nunca ocurra deadlock.
Si se utiliza evasión, se deberá correr el algoritmo del banquero con cara petición, lo cual generará mucho overhead (procesamiento extra por parte del SO).

5. Escriba el pseudocódigo de las operaciones wait y signal (bloqueantes), indicando en dónde podría ocurrir una condición de carrera y resolviéndola a través de algún mecanismo que sea idóneo para sistemas multiprocesador.

```
wait(Semaphore s) {
        while(t_a_s(&rc)==false){};
        s.contador --;
        if(s.contador < 0) {
                bloquear();
        }
        rc = 1;
}
```

```
signal(Semaphore s) {
        while(t_a_s(&rc)==false){};
        s.contador ++;
        if(s.contador <= 0) {
                despertar(s.cola[0]);
        }
        rc = 1;
}
```

La variable semáforo “s” podría estar siendo modificada por varias operaciones. Por ende sería necesario garantizar mutua exclusión usando test&set (llamadas “t_a_s” en el código), y siendo rc inicializada en 0 (cero)

Bonus

a. A diferencia de usar una variable global, se aloca memoria dinámicamente cuando se necesita (al iniciar la función) y se dealoca dinámicamente cuando se deja de necesitar (al finalizar la función). A diferencia de usar malloc, todo ello ocurre automáticamente sin llamar a malloc() ni a free(), por lo tanto es menos propenso a errores.
b. El error común podria ser un “stack overflow” (manifestado en linux típicamente como un segmentation fault). En la función recursiva ocurriría por llegar a un nivel de recursión muy alto. En el caso del vector de tamaño variable ocurriría si el tamaño dinámicamente calculado es exageradamente grande.

Práctica

Planificación

Un SO planifica sus procesos utilizando 3 colas de Ready.
Los procesos nuevos ingresan a una cola de prioridad intermedia que utiliza RR con Q = 3. Cuando los mismos son desalojados de dicha cola por fin de Q se mueven a la cola de menor prioridad que utiliza FIFO.
Cuando un proceso no consume todo su Q, al volver de IO se colocará en una cola de mayor prioridad en donde se le dará un Q' igual a lo que no ejecutó de su Q anterior. En caso de ser desalojado por fin de Q´ en dicha cola, se lo llevará al final de la cola RR de prioridad media.
La cola FIFO será desalojada si un proceso llega a una cola de mayor prioridad. Sin embargo, las colas RR son con desalojo entre ellas únicamente con respecto al quantum.

	Arribo	CPU	IO	CPU	IO	CPU
A	0	5	1	2	-	-
B	2	1	2	1	1	1
C	5	2	2	2	-	-
D	6	6	-	-	-	-

A																				FIN		
B															FIN							
C																FIN						
D																						FIN
ARRIBO	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
	A		B			C	D															
RR Q'	A		B			C	B			C	CB	B	B									
RR Q 3							D		D				C									
FIFO				A		A	A		A	A	A	AD	AD		AD							

Deadlock

	R1	R2	R3	R4		R1	R2	R3	R4
P1	1	1	0	0	P1	2	0	0	1
P2	1	0	2	1	P2	0	0	2	0
P3	0	0	1	0	P3	1	0	1	0
P4	1	0	0	1	P4	0	0	0	1
Recursos asignados					Peticiones actuales				

a. Teniendo en cuenta las matrices de **Recursos Asignados** y **Peticiones Actuales** indique cuál sería el vector **Recursos Totales** mínimo para que no haya deadlock (la suma de las instancias de todos los recursos debe ser la mínima).
b. Utilizando la matriz de **Recursos asignados.** interpretando en esta ocasión a la matriz de **Peticiones Actuales** como **Necesidad** (recursos que todavía podría pedir) y su vector de **Recursos Totales** obtenido en a) proponer una petición de recursos válida por parte de un proceso que no sea aprobada por estado inseguro.

a-

Se comienza con Disp= 0 0 0 1 (mínimo necesario para atender a un proceso- P4)
P4 - Disp 1 0 0 2
No se puede atender a ninguno más. Para atender a P1 o a P3 necesito una instancia más (de R1 y R3 respectivamente). Por ahora da igual ir por cualquier camino, vamos por P1, entonces comenzamos con 1 instancia más de R1.

Disp = 1 0 0 1
P4 - 2 0 0 2
P1 - 3 1 0 2
Para atender a otro proceso debería agregar otra instancia (ya iríamos 3) por lo que probamos con la otra opción, atendiendo a P3

Dis = 0 0 1 1
P4 - 1 0 1 2
P3 - 1 0 2 2
P2 - 2 0 4 3
P1 - 3 1 4 3

Como mínimo se necesitan 2 instancias disponibles = 0 0 1 1

Con este vector se pueden ejecutar todos los procesos en el orden: P4 - P3 - P2 -P1
Entonces, los totales serían: 3 1 4 3

Nota: otras posibles soluciones son disp: 1010, disp: 0020

b-
Teniendo como disponibles: 0 0 1 1
Y considerando a la segunda matriz como Matriz de Necesidad (lo que declaró que puede llegar a pedir menos lo que ya tiene asignado):

P1 pide 1 R4 -> 0 0 0 1

Es menor a su necesidad? Si
Es menor a los disponibles? SI

Existe una secuencia segura?

Primero simulamos la asignación -> disp : 0 0 1 0
P1 ahora necesita 2 0 0 0
No se puede satisfacer la necesidad de ningún proceso => Estado inseguro

Sincronización

Homero es un buen padre que juega con sus hijos alternadamente para que no se pongan celosos. Estableció que primero juega con maggie, luego con lisa y luego con bart (para luego volver con maggie, y repetir el ciclo). Cada tanto sus hijos deciden usar el sofá, para el cual solo hay dos lugares. Finalmente, si la cantidad de veces que Homero jugó es mayor a 100, los hijos se compadecen y deciden irse a dormir.
Dado el siguiente pseudo-código:

Bart	Lisa	Maggie	Homero

while(true){
 	
w(bart)
w(hijos)
 	
s(homero)
 	
jugar_con_homero()
 	
s(maggie)
 	
w(sofa)
sentarse_en_sofa()
s(sofa)
 	
if (juegos>100) {
exit();
}
 	
}
	while(true){
	
	w(lisa)
	w(hijos)
	
	s(homero)
	
	jugar_con_homero()
	
	s(bart)
	w(sofa)
	sentarse_en_sofa()
	s(sofa)
	
	if (juegos>100) {
	exit();
	}
	
	}
		while(true){
		
		w(maggie)
		w(hijos)
		
		s(homero)
		
		jugar_con_homero()
		
		s(lisa)
		
		w(sofa)
		sentarse_en_sofa()
		s(sofa)
		
		if (juegos>100) {
		exit();
		}
		
		}
			while(true){
			w(homero)
			
			jugar_con_hijo()
			
			juegos++
			
			s(hijos)
			}

sofa = 2
lisa = bart = homero= 0
hijos = maggie = 1