Sistemas Operativos
1° Parcial 1C2019 - TM - Resolución

<u>Aclaración</u>: La mayoría de las preguntas o ejercicios no tienen una única solución. Por lo tanto, si una solución particular no es similar a la expuesta aquí, no significa necesariamente que la misma sea incorrecta. Ante cualquier duda, consultar con el/la docente del curso.

Teoría

1. HRRN elige procesos cuyas próximas ráfagas sean cortas, pero ponderando también su espera en la cola de listos, para evitar la inanición. Ambos SJF apuntan a elegir al más corto, ya sea su ráfaga completa o bien su remanente. HRRN y SJF sin desalojo, no desalojan procesos en ejecución. El hecho de que HRRN calcule un response ratio para elegir procesos hace que sea el que mayor overhead presenta. Por otro lado, entre ambos SJF, la variante con desalojo produce mayor overhead porque compara a cualquier proceso que ingrese a la cola de listos con el que está en ejecución.

2. Tenemos tres tipos de semáforo:
   - Contadores: representan instancias de un recurso. Pueden tener cualquier valor, siempre y cuando sean números enteros.
   - Binarios: representan dos estados (cero o uno, libre u ocupado). Se utilizan para permitir o prohibir a un proceso avanzar.
   - Mutex: son un caso particular de los binarios. Se utilizan para garantizar mutua exclusión sobre un recurso o región crítica.

   Para un productor consumidor infinito necesitaríamos un semáforo mutex que proteja el buffer compartido y un semáforo contador que lleve un registro de la cantidad de mensajes en el buffer. De esa forma, un consumidor no intentará retirar un mensaje si no hay ninguno aún, y el productor lo incrementará ante cada nuevo mensaje.

3.
   a. La forma correcta es realizarlo a través de una syscall, le "pedimos" al SO que realice la operación por nosotros, realizando el correspondiente cambio de modo
   b. La forma incorrecta sería intentar ejecutar la instrucción privilegiada en directamente. Esto fallaría ya que dicha instrucción sólo puede ser ejecutada en modo kernel.

4. V o F
   a. F. Es una solución "barata" a nivel de no tener que determinar cuál es más conveniente pero no siempre soluciona el problema. Además, el impacto es muy alto ya que se pierde el avance de todos los procesos a diferencia de finalizar sólo uno o desalojar recursos.

b. F. Justamente evasión asegura que siempre se esté en estado seguro, por lo que nunca podría ocurrir un deadlock. Al ser pesimista siempre considera ese caso (que todos pidan el máximo) a la hora de ver si le asigna o no un recurso a un proceso.

5. Puede ocurrir que estemos modificando una variable compartida entre dos procesos y que en medio de la operación (que a nivel instrucción se descompone en varias instrucciones) por causa de una interrupción el planificador decida ejecutar otro proceso. Esto da resultado la condición de carrera. Una forma de solucionarlo es sincronizando la región crítica con instrucciones atómicas como test_and_set, que pueden ser utilizadas en entornos multiprocesador.

Práctica

1. Como el enunciado no nos especifica si las E/S son manejadas por la biblioteca o se llaman directamente a través del SO, se analizan ambas soluciones y se consideran válidas una u otra.

E/S por Biblioteca

a.

K1																			
K2					I/O	I/O	I/O												
K3U1		I/O	I/O																
K3U2								I/O	I/O										
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	

b.

i) En T=11, en RR debería ejecutar K1 en vez de K3 que fue elegido por estar en la cola de prioridad de VRR. (En T=3 también pero en RR ejecutaría igual ya que K3 llega antes que K1)
ii) En T=1, si la biblioteca implementara jacketing seguiría ejecutando ULT3.2 ya que el KLT no se bloquearía al realizar la I/O.

E/S por SO

a.

K1																			
K2					I/O	I/O	I/O												
K3U1		I/O	I/O																
K3U2											I/O	I/O							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	

b.
i) En T=6, se corta la ejecución de KLT3 por quedarle 2 unidades de tiempo para ejecutar, en RR ejecutaría todo el quantum.
ii) En T=1, si la biblioteca implementara jacketing seguiría ejecutando ULT3.2 ya que el KLT no se bloquearía al realizar la I/O.

2.

a) Primero miramos dos cosas: total de recursos asignados, y el máximo a nivel peticiones máximas (necesidad + asignaciones)

Vector de asignados = 2 2 3 4 -> lo mínimo que necesito par que todo eso pueda estar realmente asignado
Vector de máximos de peticiones máximas = 3 1 2 3 -> lo mínimo que necesito para que las peticiones máximas sean válidas

Entonces, en principio, el vector mínimo de totales puede ser = 3 2 3 4 (máximo entre ambos) => recursos disponibles 1 0 0 0
Sin embargo, falta un requerimiento más, que es que como sabemos que se utiliza evasión, dicho sistema tiene que estar en estado seguro, es decir que todas las necesidades máximas deben poder ser satisfechas en algún orden.

Para poder satisfacer alguno de los pedidos, necesito al menos 2 recursos más dependiendo de si elijo primero a P1, P2 o P4 (para P3 necesito 4 más)

Vamos por P1 -> por lo que iniciamos con recursos disponibles 2 0 1 0
Finaliza P1 -> 3 1 1 0 -> puedo ejecutar P4
Ejecuto P4 -> 3 1 2 2
Ejecuto P2 -> 4 1 4 2
Ejecuto P3 -> 4 2 4 4

Encontramos la secuencia P1 - P4 - P2 -P3
Recursos totales mínimo : 4 2 4 4 (pueden existir otras opciones si se avanza por P2 o P4)

b) Ahora debemos revisar si el vector de recursos que definimos se banca los siguientes requerimientos
Recursos disponibles = 2 0 1 0
P1 1 R1 -> sabemos que se lo asigna ok
En este caso no hay problema, ya que luego podría seguir ejecutando P1 y continuar con la secuencia de a)

Recursos disponibles = 1 0 1 0
P4 1 R2 -> vemos que si o si necesitamos 1 recurso disponible de R2 =
Recursos disponibles = 1 1 1 0
Se lo asigno quedando = 1 0 1 0 -> puedo ejecutar P1 .. y el resto de la secuencia de a)

P4 1 R3 -> no se lo asinga
Probamos -> recursos disponibles 1 0 0 0
Ninguno puede ejecutar => OK

Entonces, a nuestros recursos totales de a) debemos agregarle una instancia de R2

3.

Cliente (N instancias)	Cajero (2 instancias)

dirigirse_al_super()
 	
wait(canastos)
agarrar_canasto()
cargar_canasto()
esperar_en_cola()
signal(gente_esperando)
 	
wait(llamado)
id_cajero = recibir_llamado_cajero()
entregar_productos()
signal(productos[id_cajero])
 	
wait(ticket[id_cajero])
recibir_ticket()
pagar()
signal(pagado[id_cajero])
 	
signal(canastos)
	wait(gente_esperando)
	llamar_cliente()
	signal(llamado)
	
	wait(productos[get_id()])
	pasar_productos_por_lectora()
	entregar_ticket()
	signal(ticket[get_id()])
	
	wait(pagado[get_id()])
	saldo_actual += recibir_dinero();
	
	if(saldo_actual > 10000){
	wait(mutex_caja_fuerte)
	depositar_en_caja_fuerte(saldo_actual)
	signal(mutex_caja_fuerte)
	}

Canastos = 20
Gente_esperando = 0
Llamado = 0
Productos = {0,0}
Ticket = {0,0}
Pagado = {0,0}
Mutex_caja_fuerte = 1