Sistemas Operativos
1° Parcial 1C2019 - TT - Resolución

Aclaración: 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. La syscall es el punto de entrada oficial y último para solicitar un servicio al sistema operativo. Ofrece máxima flexibilidad de uso y optimizaciones de performance, aunque carece de facilidad/claridad/portabilidad de uso.
   El wrapper (de una syscall) es una función que corre en modo usuario que se interpone entre la solicitud del servicio y la syscall real. Ofrece mayor facilidad/claridad/portabilidad en el uso, aunque restringe flexibilidad y la posibilidad de realizar optimizaciones de performance.

2. La estrategia de prevención evita el deadlock mediante el evitar que ocurra alguna de las cuatro condiciones necesarias y suficientes para que ocurre al deadlock, incurriendo en un grado de overhead bajo. Un ejemplo de uso podría ser una computadora de un avión, donde las consecuencias de la ocurrencia del deadlock podrían ser fatales.
   La estrategia de detección y recuperación no impide la ocurrencia del deadlock, sino que lo intenta resolver en caso de que el mismo ocurra, tomando un enfoque meramente reactivo y teniendo por lo tanto un grado de overhead mediano. Un ejemplo podría ser una base de datos transaccional, donde los deadlocks deberían resolverse pero es más importante la performance del sistema.

3. Región crítica es aquella porción del código donde se acceden o manipulan recursos compartidos con otros procesos/hilos, donde la no sincronización implicaría una condición de carrera entre todos los procesos/hilos que acceden a ese mismo recurso.

   Criterios para evaluar una solución al problema de la sección crítica:

1. Mutua Exclusión: Un sólo proceso/hilo a la vez. Para asegurarse que no hay condiciones de carrera, la sección crítica debe cumplir con las condiciones de Bernstein:
a. Sean *I* las variables de entrada, *O* las variables de salida, y los índices "*i,r*" dos porciones de código, los mismos son independientes si:

$I_i \cap O_r = \varnothing$, $I_r \cap O_i = \varnothing$, $O_r \cap O_i = \varnothing$

2. Progreso: Si está libre, un proceso/hilo debería poder acceder. Mientras que un proceso/hilo que está fuera de la misma, no debería impedir el ingreso.
3. Espera Limitada: Para ingresar el tiempo deber ser reducido, para evitar la inanición. Es una condición deseable.
4. Velocidad Relativa: El tiempo en que un proceso/hilo se encuentra dentro de la misma, deber ser finito, ya que corta la multiprogramación. Es una condición deseable.

Ejemplo de las funciones wait() y signal() con deshabilitación de interrupciones:

```
wait(){
        deshabilitar_interrupciones();
        semáforo --;
        if (semáforo < 0) bloquear_proceso();
        habilitar_interrupciones();
}
```

```
signal() {
        deshabilitar_interrupciones();
        semáforo ++;
        if (semáforo <= 0) despertar_proceso();
        habilitar_interrupciones();
}
```

4.
   a. F: Requiere porque wait() y signal() son syscalls, y las mismas siempre se realizan con un cambio de modo.
   b. F: Los hilos, independientemente de que sean KLTs o ULTs comparten memoria (por ejemplo, variables globales) sin intervención del sistema operativo.

5. Los algoritmos con desalojo tienen en cuenta los mismos eventos de los algoritmos sin desalojo (bloqueo del proceso por syscall y finalización del mismo), y consideran también evento que derive en un ingreso a la cola de Ready (syscalls de creación de hilos/procesos, o interrupciones que indiquen fin de ráfaga o fin de I/O).
   Los algoritmos sin desalojo descartan estos últimos eventos porque deciden no expropiar la cpu al proceso una vez que la misma fue asignada.

Práctica

1.

Peter (1 instancia)	Llenador (1 instancia)	Compañer@ matero (N instancias)

while (TRUE) {
 	
wait(yerbaPremium);
yerba = tomarDe(frascoPremium);
signal(vacioPremium);
 	
wait(mDespol);
yerbaLimpia = usar(despol, yerba);
signal(mDespol);
 	
mate = cargarCon(yerbaLimpia);
disfrutarUnosVerdes(mate);
}
	while (TRUE) {
	
	wait(vacioPremium, 20);
	wait(vacioComun, 20);
	
	llenar(frascoPremium);
	signal(yerbaPremium, 20);
	
	llenar(frasco);
	signal(yerbaComun, 20);
	}
		while (TRUE) {
		
		wait(yerbaComun);
		wait(mFrasco);
		yerba = tomarDe(frasco);
		signal(mFrasco);
		signal(vacioComun);
		
		wait(mDespol);
		yerbaLimpia = usar(despol, yerba);
		signal(mDespol);
		
		mate = cargarCon(yerbaLimpia);
		saborearUnosVerdes(mate);
		}

yerbaPremium = yerbaComun = 20; vacioPremium = vacioComun = 0; mDespol = mFrasco = 1
Nota: el semáfro mFrasco no es necesario usarlo en el proceso Llenador, dado que cuando quiere usar el recurso compartido ningún otro proceso estará queriendo usarlo (siempre y cuando esté bien sincronizada la solución)

2.

a. VRR Q=2, porque en el instante 6 se observa que se interrumpe la ejecución de KLT 2. Esto muestra que es un algoritmo con desalojo. Su ráfaga restante es de 1 unidad de tiempo, con lo cual no podría ser SJF.

b. Obligatorios: t = 0, t = 1, t = 2, t = 3, t= 4, t = 5, t = 6, t = 8, t = 9, t = 11, t = 12
Opcionales: t = 1, t = 4 (cambio de modo unicamente)

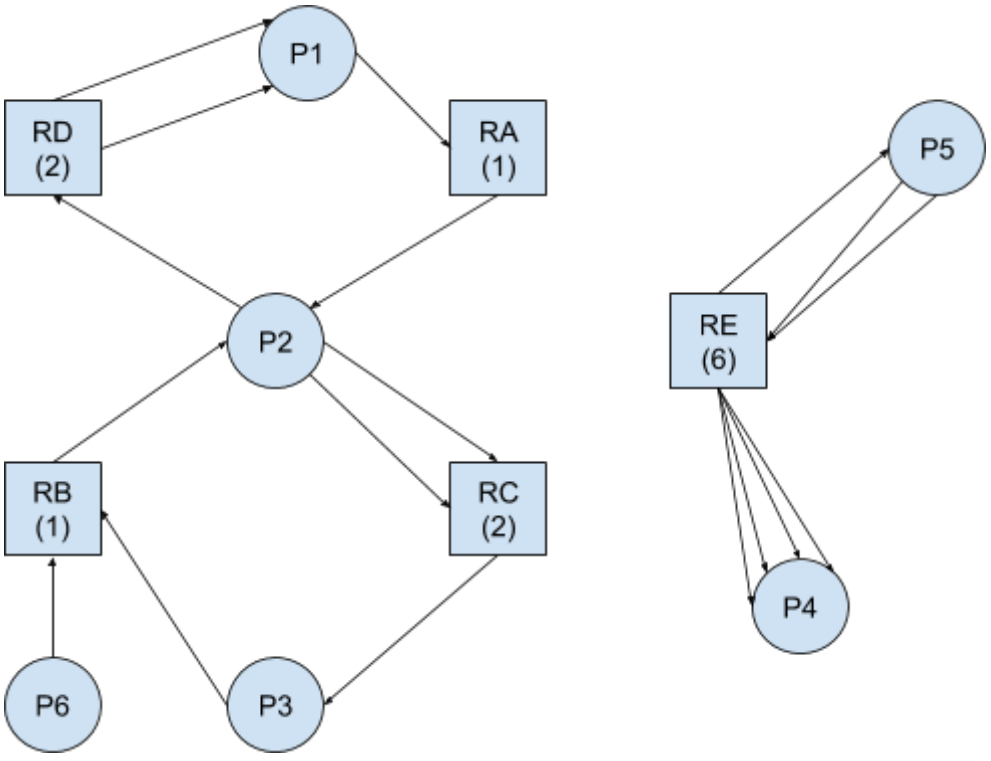
c. VRR Q=2

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P1 – KLT1				IO	IO								F				
P1 – KLT2						IO									F		
P2 – KLT3								IO									F
P3 – KLT4										IO						F	

3.

a. – P6 no tiene recursos asignados, es descartado del algoritmo
- Disponibles iniciales: 0 0 1 0 1
- P4 puede finalizar (no tiene pendientes) Nuevo disponible: 0 0 1 0 **5**
- P5 puede finalizar. Nuevo disponible: 0 0 1 0 **6**
- Nadie más puede finalizar. Existe deadlock entre P1, P2 y P3.

b.



- P1, P2 y P3: están en deadlock (espera circular visible, aunque se determina oficialmente con el algoritmo ejecutado en el punto anterior

- P4: no se encuentra bloqueado
- P5: se encuentra bloqueado temporalmente (hasta que P4 libere los recursos o finalice)
- P6: se encuentra bloqueado en starvation (dado que nunca se le asignará el recurso por existir un deadlock que involucra al mismo)

c.

- Para eliminar el deadlock hay que elegir finalizar alguno de los procesos involucrados en el mismo: P1, P2 y P3 (el enunciado explicita que solo puede eliminarse *1 proceso*)
- Por prioridades, se debería elegir a P3. El problema es que esto no elimina la espera circular que involucra a P1 y P2.
- Continuando con las prioridades, se elegiría a P2. Este sí eliminaría ambas esperas circulares y por lo tanto desaparecería el deadlock. Se verifica corriendo el algoritmo de detección nuevamente:
  - Se elimina al proceso P2
  - P6 no tiene recursos asignados, es descartado del algoritmo
  - Nuevos disponibles: **1 1** 1 0 1
  - P4 puede finalizar (no tiene pendientes) Nuevo disponible: 1 1 1 0 **5**
  - P5 puede finalizar. Nuevos disponibles: 1 1 1 0 **6**
  - P1 puede finalizar. Nuevos disponibles: 1 1 1 **2** 6
  - P3 puede finalizar. Nuevos disponibles: 1 1 **2** 2 6
  - Todos finalizaron. No existe deadlock.