Sistemas Operativos
1° Parcial 1C2024 - TM - 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. Las syscalls son la interfaz que nos brinda el sistema operativo para acceder a las funciones que él mismo provee de forma segura. Los wrappers de la biblioteca del sistema buscan abstraer el comportamiento de las primeras "envolviendo" la llamada a las mismas con lógica extra que permite simplificar su llamada por parte de los desarrolladores y brindan una mayor portabilidad.
   Cuándo usar wrappers: cuando busco simplicidad y portabilidad.
   Cuándo usar syscalls de forma directa: cuando busco un completo control sobre la configurabilidad del pedido de recursos/servicios al sistema operativo.
   Ejemplo de syscall y wrapper de la biblioteca estándar: open y fopen, read y fread, write y frite, etc.

2. Por cada **tipo** de petición se levanta al menos un proceso (es decir, 4 procesos como mínimo) ya que ***"se desea que algún error en las peticiones de algún tipo no afecten al funcionamiento del resto"***. De esta manera, cada proceso hace uso de su espacio de memoria de forma totalmente independiente imposibilitando que un error dentro de un grupo de peticiones afecte a otro grupo.
   A su vez, por cada petición nueva que llegue al servidor se creará un ULT ya que ***"desea poder planificar la atención de cada tipo bajo un algoritmo personalizado".*** De esta manera, por cada tipo de petición tendremos un proceso con una biblioteca de ULTs que se ajuste a las necesidades de planificación del tipo de petición en cuestión.
3.

	Prioridad IO bound	Cambios de contexto	Starvation
FIFO	No prioriza procesos IO bound de ninguna manera, ya que los procesos son atendidos por orden de llegada y ejecutan hasta liberar la CPU (bloqueo, exit o yield). Un proceso largo puede monopolizar la cpu, lo que es injusto para los procesos IO bound y genera un tiempo de espera impredecible.	Mínimo: es el algoritmo con menor overhead.	No genera starvation.

RR	No prioriza los procesos IO bound pero sí brinda un tiempo de espera predecible, gracias a la cota superior de tiempo de ejecución en la CPU establecida por el quantum.	Intermedio: los cambios de contexto aumentan a medida que se achica el quantum.	No genera starvation.
VRR	Tal como RR, brinda un tiempo de espera más predecible y, además prioriza a los procesos que no llegaron a aprovechar su quantum anterior colocándolos en una cola de mayor prioridad.	Mayor: se suman aún más cambios de contexto por utilizarse un quantum variable (que siempre es menor al quantum fijo).	No genera starvation.

4. A. FALSO

   Si se dan las siguientes condiciones:

   - La sección crítica es tan pequeña que el tiempo requerido para hacer los cambios de contexto inherentes al bloqueo-desbloqueo del proceso es mayor que el tiempo que el proceso necesita esperar para entrar a la sección crítica.
   - Se posee más de un procesador y los procesos que quieren acceder a la sección crítica de forma concurrente están ejecutando de forma paralela.

   Entonces, es más performante utilizar una solución de mutua exclusión con espera activa, así como lo son testAndSet y swapAndExchange. En estos casos también sería más performante utilizar semáforos con espera activa que con bloqueo, pero las soluciones de HW serían las más performantes de todas por no requerir el uso de syscalls (y los cambios de contexto inherentes a las mismas).

   B. VERDADERO

   Condiciones de Bernstein: Dos o más hilos deben estar accediendo de forma concurrente al mismo recurso y **al menos uno** de ellos debe estar modificándolo.

5. A. FALSO

   Matriz de peticiones máximas -> Evasión

   B. FALSO

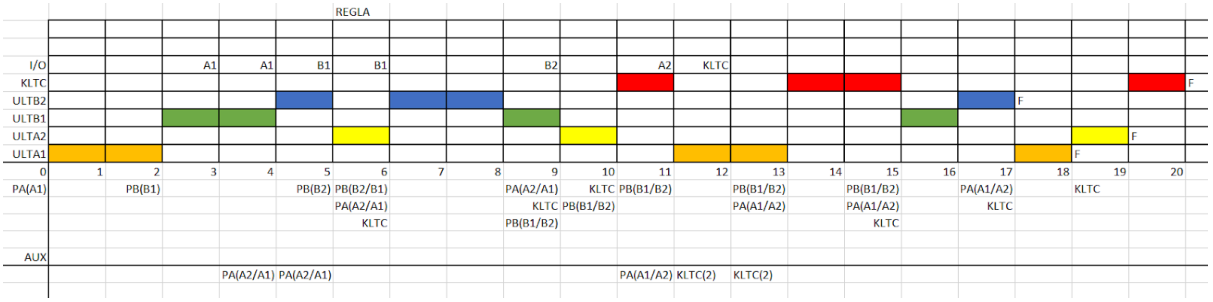
   En caso de elegir como estrategia de recuperación de deadlocks la eliminación de un proceso entonces debe posteriormente volverse a correr el algoritmo de detección para verificar la efectiva recuperación y elegir otra víctima en caso de que el deadlock no se haya resuelto.

   (También es válido que ejemplifiquen con un caso concreto donde solamente finalizando un proceso no es suficiente para resolver el deadlock).

Práctica

1.

a)



b) Cambiaría en el instante 4 en que se bloquearía el KLT.

2.

stockMaximo = 33; **stock** = 0; **accesoAlmacen** = 2
entrega[10] = {0, 0, 0 ...}; **mutexEntrega**[10]= {1, 1, 1 ...}

Fábrica	**Distribuidor (5 instancias)**	**Asentamiento (10 instancias)**

while(true) {
caja = fabricarMuniciones()
 	
wait(stockMáximo)
wait(accesoAlmacen)
depositar(almacén, caja)
signal(accesoAlmacen)
signal(stock)
}
	while(true) {
	wait(stock)
	wait(accesoAlmacen)
	caja = retirar(almacén)
	signal(accesoAlmacen)
	signal(stockMáximo)
	
	id_asent = getDestino(caja)
	wait(mutexEntrega[id_asent])
	entregar(lider[id_asent], caja)
	signal(mutexEntrega[id_asent])
	signal(entrega[id_asent])
	}
		while(true){
		id_asent = getID()
		wait(entrega[id_asent])
		wait(mutexEntrega[id_asent])
		caja = recibir(lider[id_asent])
		signal(mutexEntrega[id_asent])
		
		cargarArmas()
		}

3.

a)

	W(mutexA)		A++				A++	W(mutexB)		Bloqueado por mutexB			
P1													
				W(mutexB)		B++				B++	W(mutexA)		Bloqueado por mutexA
P2													

b) Cambiando el planificador de corto plazo a uno sin desalojo como FIFO, el deadlock no ocurriría en este caso ya que los semáforos no se tomarían de manera alternada, sino que ejecutaría un proceso completo, y luego el otro. Otra opción podría ser cambiar el Quantum a un valor muy grande para que finalicen antes del primer fin de Q.

c) Hay varias opciones, por ejemplo liberando los mutex inmediatamente luego de modificar cada variable, otra opción puede ser tomar los mutex en orden, en cualquier caso no podría darse **espera y retención** y por lo tanto no habría **espera circular**.

P1	P2
Wait(mutexA) contadorA++ **Signal**(mutexA) **Wait**(mutexB) contadorB++ **Signal**(mutexB)	**Wait**(mutexB) contadorB++ **Signal**(mutexB) **Wait**(mutexA) contadorA++ **Signal**(mutexA)

P1	P2
Wait(mutexA) **Wait**(mutexB) contadorA++ contadorB++ **Signal**(mutexA) **Signal**(mutexB)	**Wait**(mutexA) **Wait**(mutexB) contadorB++ contadorA++ **Signal**(mutexB) **Signal**(mutexA)

P1	P2
Wait(mutexA) contadorA++ **Wait**(mutexB) contadorB++ **Signal**(mutexA) **Signal**(mutexB)	**Wait(mutexA)** contadorA++ **Wait(mutexB)** contadorB++ **Signal**(mutexB) **Signal**(mutexA)