Teoría

1) Relacione los conceptos de: cambio de modo - cambio de contexto - cambio de proceso

Siempre que hay un cambio de proceso, debemos realizar dos cambios de contexto, uno para guardar el estado del proceso reemplazado y otro para setear el del nuevo proceso de usuario. A su vez, es el planificador el que realiza este cambio de procesos, es decir, entonces hay también dos cambios de modo de u->k y luego de k->u al setear el nuevo proceso.
Sin embargo, las relaciones al revés no siempre se cumplen. Por ejemplo, podemos tener dos cambios de contexto sin que haya cambio de proceso; esto puede pasar cuando un proceso accede al SO o si éste último tiene que atender algún evento(interrupción) que le hace cambiar el contexto de ejecución, pero que luego vuelve a setear al mismo proceso para ejecutar (por ejemplo el planificador elige al mismo proceso).
Por último, también pueden ocurrir dos cambios de contexto sin que haya un cambio de modo. Un ejemplo de lo mismo es cuando las interrupciones se atienden en forma anidada. El SO se encuentra atendiendo una interrupción, luego llega otra de mayor prioridad por lo que hay que atenderla. Como se pasa a ejecutar otra cosa, y luego hay que continuar la atención de la interrupción anterior, es necesario hacer un cambio de contexto; sin embargo, el procesador ya se encontraba en modo kernel, por lo que no hay cambio de modo.

2) Escriba el pseudo-código de dos hilos cooperativos, incluyendo (y señalando) dónde se referencia su stack, su sección de datos y su heap.

```
int GLOBAL = 1; ← Data
void main() { ← Stack
  char* msj = pedir_mem(10); ← Heap
  crear(hilo_1, msj);
  crear(hilo_2, msj);
}
```

```
void hilo_1(char* msj) {
  concat(msj, “hilo1”);
}
```

```
void hilo_2(char* msj) {
  concat(msj, “hilo2”);
}
```

3) Indique qué similitudes y diferencias hay entre el algoritmo del banquero y el de detección de deadlocks. ¿Por qué se dice que la técnica de evasión es “pesimista”?

Ambos algoritmos consideran una matriz de recursos asignados y vectores de recursos totales y disponibles. Ambos también realizan una simulación de asignación de recursos, la diferencia radica que en el caso de detección, el resultado dice si en dicho momento el sistema se encuentra o no en deadlock mientras que con el algoritmo del banquero se determina si el sistema está en estado seguro.
Otra diferencia es que el algoritmo del banquero utiliza las peticiones máximas, que pueden ocurrir o no, mientras el de detección sólo conoce peticiones que se realizan en un momento en particular.
La técnica de evasión es pesimista ya que considera el peor de los casos: ante cada petición, sólo la satisface inmediatamente si la misma deja al sistema en un estado que aún si todos los procesos piden lo máximo que podrían pedir, se pueden asignar correctamente los recursos (en una secuencia segura).

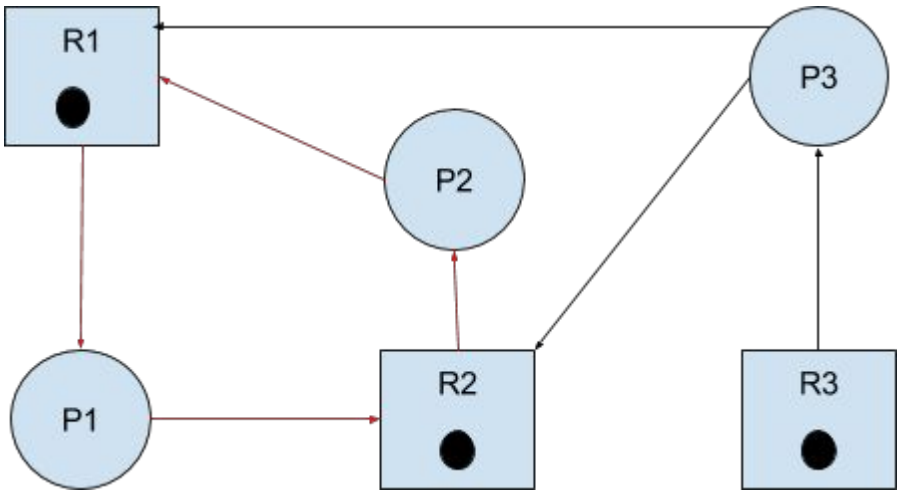
4) V o F. En un esquema en que se utiliza un algoritmo de planificación por prioridades, si dos procesos de diferente prioridad comparten un mismo recurso y utilizan mutex con espera activa se puede llegar a generar una situación en la que ninguno de los dos pueda ejecutar. Justifique

Verdadero, ver inversión de prioridades

5) Explique cuál podría ser el problema de utilizar semáforos sin espera activa en procesos que utilizan ULTs.
Si los semáforos son bloqueantes, y bloquean todo el ULT, podrían nunca continuar.

Práctica

1.a) Disponibles (0,0,0) → No puede ejecutar ninguno
1.b) En rojo se ve un deadlock. P3 se ve en inanición. El algoritmo de detección no detecta procesos en inanición, si tienen recursos asignados.



2)

U1														X				
U2															X			
U3												X						
K2																		X
K3																	X	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

T = 0
K1 = 2 / K2 =3 / K3 = 1 (por dato) => elijo K3

T = 2
Elijo K1 .
Todos los ULTs llegaron juntos, por lo cual el RR será mayor para el que tenga menor ráfaga, en este caso U3

T = 3
Solo está K2 listo

T = 4
K3 vuelve de bloqueado. Su nueva estimación es = 1´+ 2 / 2 = 1,5.
K2 ya ejecutó una unidad, y como su estimado es de 3 , estimamos que le quedan 2 unidades más.

Por eso elegimos desalojarlo y ejecutar K3.
T = 5
K1 vuelve de bloqueado. Su nueva estimación es = 2 + 1/ 2 = 1,5
K3 ya ejecutó una unidad, y como su estimado es de 1,5, estimamos que le queda 0,5.
Por eso dejamos seguir ejecutando a K3, ya que estimamos que le queda menos que a K1.

T = 7
Est K1 = 1,5
Est K2 = 2
Elijo a K1

RR1 = 7 + 3 / 3 = 10/3 = 3,3..
RR2 = 7 + 2 / 2 = 9 / 2 = 4,5
RR3 = 2 + 1 / 2 = 2
Elijo a U2 porque tiene el mayor RR

T = 8
K3 vuelve de bloqueado. Su nueva estimación es = 1,5 + 3/ 2 = 2,25
K1 ya ejecutó una unidad, y como su estimado es de 1,5, estimamos que le queda 0,5.
Por eso dejamos seguir ejecutando a K1, ya que estimamos que le queda menos que a K3.

T = 9
Est K2 = 2
Est K3 = 2,25
Elijo a K2

T = 10
Est K3 = 2,25
Est K1 = 1,5 + 2 / 2 = 1,75
Elijo a K1
RR 1 = 10 + 3 / 3 = 13/ 3 = 4,3..
RR 2 = 0 +1 / 1 = 1
RR 3 = 5 + 1 / 1 = 6
Elijo a U3

T = 11
K2 vuelve de bloqueado. Su nueva estimación es = 3 + 2/ 2 = 2,5
K1 ya ejecutó una unidad, y como su estimado es de 1,75, estimamos que le queda 0,75.
Por eso dejamos seguir ejecutando a K1, ya que estimamos que le queda menos que a K2.
RR 1 = 11 + 3 / 3 = 14 / 3 = 4, 6…
RR 2 =1 + 1 / 1 = 2
Elijo a U1

T = 15
Est K2 = 2,5
Est K3 = 2,25

Elijo a K3
Preguntas:

A. Si usamos jacketing las io que realicen los ults de k1 no van a ser bloqueantes por lo que la biblioteca de hilos de usuario va a poder planificar otro. De todas formas, esto va a generar que la ráfaga de k1 sea más larga, por lo que su siguiente estimación será mayor, y probablemente se lo deje para el final para ejecutar.
B. Si tenemos dos CPUs vamos a poder ejecutar dos klts en paralelo. Por lo tanto, los procesos probablemente terminarán antes.
C. Si los ults llaman directamente al SO para hacer una io lo que ocurrirá es que al bloquearse y guardar su contexto, en el tcb de k1 se guardará la última instrucción del ult que realizó la io. Por lo tanto, al volver a elegir a dicho klt seguirá el mismo ult, ya que la biblioteca no tendrá forma de planificar

3) mut_revisar = mut_ok = 1; deploy=0; max_commits=10;

Programador (4 instancias)	Peter (1 instancia)	Tester (1 instancia)

while(1) {
 	
wait(mut_revisar);
if(commits_a_revisar > 0) {
~~signal(mut_revisar);~~
 	
revisar_commit(1);
signal(max_commits);
 	
wait(mut_ok);
commits_ok++;
signal(mut_ok);
 	
~~wait(mut_revisar);~~
commits_a_revisar--;
signal(mut_revisar);
} else {
signal(mut_revisar);
programar();
 	
wait(max_commits);
commitear(1);
 	
wait(mut_revisar);
commits_a_revisar++;
signal(mut_revisar);
}
}
	while(1) {
	
	wait(mut_ok);
	if(commits_ok > 10) {
	commits_ok = 0;
	signal(mut_ok);
	
	deployar();
	signal(deploy);
	
	} else {
	signal(mut_ok);
	
	super_programar();
	wait(max_commits, 5);
	commitear(5);
	
	wait(mut_revisar);
	commits_a_revisar += 5;
	signal(mut_revisar);
	}
	}
		while(1) {
		wait(deploy);
		testear_deploy();
		}

Nota: el programador tiene dos semáforos tachados. A simple vista uno pensaría en ponerlos… esa parece ser la forma de encarar la mutua exclusión. Pero como commits_a_revisar se decrementa recién al final, nos puede traer inconsistencias.