UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

Fundamentos de los Sistemas Operativos
Convocatoria ordinaria - 20 de mayo de 2019

1	2	3		test	NOTA

Nombre y apellidos **DNI/NIE**

SOLUCIONES

1 (1 punto) Tenemos un proceso que genera la siguiente cadena de referencias a páginas (cada letra representa una página):

A B C B C D B E A B E A D E C A D A

El proceso tiene asignados tres marcos de página, inicialmente vacíos. Si el sistema utiliza un algoritmo LRU para el reemplazo de páginas, ¿cuántos fallos de página generará esta cadena? ¿Se podría realizar una planificación de reemplazos con un menor número de fallos? Demuéstrelo.

Este es el resultado de planificar las referencias mediante LRU:

A	B	C	B	C	D	B	E	A	B	E	A	D	E	C	A	D	A
*	*	*			*		*	*				*		*	*	*	
A	A	A	A	A	D	D	D	A	A	A	A	A	A	C	C	C	C
	B	B	B	B	B	B	B	B	B	B	B	D	D	D	A	A	A
		C	C	C	C	C	E	E	E	E	E	E	E	E	E	D	D

Los asteriscos indican fallos de página. La tabla inferior muestra el contenido de los marcos a medida que se van resolviendo las referencias. Resulta un total de **diez fallos de página**.

Es posible encontrar una planificación con menos fallos. Por ejemplo, en la referencia número seis, hemos retirado la página A. Si la hubiéramos mantenido y hubiéramos reemplazado la página C, el resultado hubiera sido mejor. En realidad, la mejor forma de demostrar que existe una mejor planificación es aplicar el algoritmo óptimo, que sabemos que es inmejorable: sustituir siempre la página que más va a tardar en accederse en el futuro. Este sería el resultado:

A	B	C	B	C	D	B	E	A	B	E	A	D	E	C	A	D	A
*	*	*			*		*					*		*			
A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
	B	B	B	B	B	B	B	B	B	B	B	D	D	D	D	D	D
		C	C	C	D	D	E	E	E	E	E	E	E	C	C	C	C

Esta planificación resulta en solamente siete fallos de página, lo cual demuestra que es posible bajar el resultado de la LRU (y además nos dice cuál es el mínimo número de fallos posible).

---

2 (2 puntos) Tenemos un sistema de memoria paginada con dos niveles jerárquicos de tablas de páginas. El tiempo de acceso a la RAM es de 50 nseg. El sistema utiliza una TLB que ofrece un tiempo de acceso muy bajo (puede considerarse despreciable). Se observa que la tasa de aciertos de la TLB es del 95%. ¿A cuánto asciende el tiempo efectivo medio de acceso a la memoria principal?

Podemos aplicar esta fórmula:

$$Tmedio = Paciertos \cdot TRAM + (1-Paciertos) \cdot 3 \cdot TRAM$$

Siendo Paciertos=95% y TRAM=50 nseg. Multiplicamos TRAM por tres porque cuando hay un fallo de la TLB tenemos primero que recorrer la tabla de páginas (dos niveles = dos accesos) para descubrir en qué dirección física tenemos que realizar el acceso a la celda deseada.

El resultado de esa fórmula es Tmedio = (95% · 50) + (5% · 3 · 50) = **55 nseg**

En este mismo sistema, las direcciones lógicas son de 48 bits, las direcciones físicas son de 32 bits y el tamaño de página es de 64 KiB. ¿Cuántas páginas lógicas puede llegar a tener un proceso? ¿Cuántos marcos físicos puede llegar a tener?

Si el tamaño de página es de 64KiB, en la dirección lógica hay que dedicar $\log_2(64Ki) = 16$ bits para el desplazamiento de la página. El resto, 48-16 = 32 bits, pueden dedicarse a seleccionar la página lógica. Por tanto un proceso puede direccionar hasta **2^{32} páginas** lógicas, que son unos cuatro millardos.

Los marcos físicos vienen limitados por la memoria física que se puede direccionar. Cada marco ocupa exactamente una página (64KiB=2^{16} bytes) y la memoria física direccionable es de 2^{32} celdas. Aplicando una simple división obtenemos cuántos marcos se pueden direccionar: $2^{32} / 2^{16} = 2^{16}$ marcos, o sea **65 536 marcos**.

¿Qué sentido puede tener que las direcciones lógicas sean más largas que las físicas?

Si se diseña un procesador así, tiene que ser por un motivo razonable. Puede ser que el fabricante quiso limitar el coste de la circuitería alrededor de la CPU: es más caro un bus de 48 bits que uno de 32 bits. Puede ser que en el momento de diseñar este procesador no hubiera disponibilidad (o necesidad) de componentes hardware que manejaran más de 32 bits, por ejemplo los chips de RAM.

Por otro lado, si un proceso puede manejar más memoria lógica que física, esto facilita el diseño de sistemas de memoria virtual, en la que el proceso esté manejando un espacio de memoria mucho mayor que el que físicamente tiene concedido en la RAM. El sistema puede apoyarse en el almacenamiento secundario para guardar páginas poco accedidas, tal como hemos visto en el Tema 4.

Nombre y apellidos	DNI/NIE

3 (1'5 puntos) En un aeropuerto se han dispuesto dos surtidores de líquido para los aviones: uno de combustible (fuel) y otro de aceite lubricante. Los aviones se acercan a los surtidores y repostan ambos líquidos, mediante la operación `repostar()` que se define más abajo. Regularmente, una empresa de suministros viene con un camión cuba y repone uno de esos líquidos. Para ello se han implementado las operaciones `reponer_fuel()` y `reponer_aceite()`.

Sobre el sistema operan las siguientes restricciones:

- La operación `repostar()` solo se puede realizar si hay suficiente cantidad de suministro en ambos surtidores. Si alguno de los depósitos es insuficiente, el avión debe esperar hasta que se reponga una cantidad suficiente.
- Mientras se está reponiendo alguno de los líquidos, ningún avión puede repostar.
- Si un avión está repostando, por seguridad ningún otro proceso puede operar con los surtidores: ni otro avión puede repostar, ni el camión cuba puede reponer.

TAREA. Complete las operaciones `repostar()`, `reponer_fuel()` y `reponer_aceite()` de manera que se cumplan todas las restricciones y que se preserve la integridad de los datos. Utilice mutex y variables condición para establecer la sincronización entre los procesos.

NOTAS. Suponga que los depósitos de los surtidores tienen capacidad ilimitada (nunca se llenan). Puede asumir que solamente existe un camión cuba (no hay que sincronizar varios camiones).

```
// Cantidad de líquido en cada surtidor
float deposito_fuel = XXX;
float deposito_aceite = XXX;

// Operaciones de los surtidores

void repostar (float litros_fuel, float litros_aceite)
{
  // ... acciones físicas de repostaje
  deposito_fuel -= litros_fuel;
  deposito_aceite -= litros_aceite;
}

void reponer_fuel (float litros_fuel)
{
  // ... acciones físicas de reponer combustible
  deposito_fuel += litros_fuel;
}

void reponer_aceite (float litros_aceite)
{
  // ... acciones físicas de reponer aceite lubricante
  deposito_aceite += litros_aceite;
}
```

La solución necesita un mutex para proteger las variables compartidas deposito_fuel y deposito_aceite. Podemos tener una variable condición que actúe como cola de espera para los aviones que se encuentren con que no hay suficiente líquido. El camión, cuando termina de reponer líquido, ejecuta un *broadcast* sobre la variable condición, para avisar a todos los aviones que están en espera.

Un posible algoritmo se muestra a continuación. En este algoritmo, el mutex se usa también para asegurar que solamente un proceso (avión o camión) está trabajando con los surtidores.

Tal y como está diseñado el algoritmo, es importante que el desbloqueo en la variable **cola** se haga con operaciones **broadcast**, dado que puede haber varios aviones esperando y algunos seguirán teniendo que mantenerse bloqueados, porque algún depósito no tiene cantidad suficiente. Si se utilizara un **signal**, podría desbloquearse uno de esos aviones y el sistema dejaría bloqueado indefinidamente a algún otro avión que sí podría repostar.

Este algoritmo no resuelve la posible inanición de aquellos aviones que necesiten una gran cantidad de líquido y que se pueden ver postergados indefinidamente, si siempre llegan aviones con menores necesidades.

```
float deposito_fuel = XXX;
float deposito_aceite = XXX;

// Para bloquear a los procesos cuando las condiciones
// no les permitan avanzar
Mutex mutex;
Condition cola = new Condition(mutex);

// OPERACIONES

void repostar (float litros_fuel, float litros_aceite) {
   mutex.lock();
   while ( deposito_fuel < litros_fuel ||
           deposito_aceite < litros_aceite ) {
      cola.wait();
   }

   // ... acciones físicas de repostaje

   deposito_fuel -= litros_fuel;
   deposito_aceite -= litros_aceite;
   mutex.unlock();
}

void reponer_fuel (float litros) {
   mutex.lock();
   // ... acciones físicas de reponer fuel
   deposito_fuel -= litros;
   cola.broadcast();
   mutex.unlock();
}

void reponer_aceite (float litros) {
   mutex.lock();
   // ... acciones físicas de reponer aceite
   deposito_aceite -= litros;
   cola.broadcast();
   mutex.unlock();
}
```