PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC2333 — Sistemas Operativos y Redes — 1/2018

Interrogación 2

Martes 8-Mayo-2018

Duración: 2 horas
SIN CALCULADORA

1. **[15p]** Sincronización

   1.1) **[3p]** Se ha mencionado que una manera de proveer exclusión mutua para un *thread* es deshabilitar las interrupciones al momento de acceder a una sección crítica, de manera que el *thread* no pueda ser interrumpido, y luego rehabilitarlas al salir de la sección crítica. ¿Qué problema presenta esta implementación en un computador moderno del punto de vista de la exclusión mutua?

   R.

   Deshabilitar las interrupciones no es suficiente en máquinas *multicore*. En un *single-core*, al deshabilitar las interrupciones, el *scheduler* pierde la capacidad de interrumpir al *thread* actual y ejecutar otro. En un *multicore*, habrá más de un *thread* ejecutando concurrentemente aún cuando se deshabilited las interrupciones. Si no se usa una primitiva de sincronización, no hay cómo impedir que dos o más *threads* se encuentren concurrentemente dentro de la sección crítica.

   1p. Mencionar el caso de *multicores*.

   2p. Argumentar que si hay dos *threads* ejecutando concurrentemente en dos o más *cores*, el hecho de deshabilitar interrupciones no impide que uno deje de ejecutar, y por lo tanto podrían estar simultáneamente en la sección crítica.

   1.2) **[12p]** Se desea simular un proceso de construcción de bicicletas. Una bicicleta está formada por, exactamente, un marco y dos ruedas, y se construye llamando a la función existente `construir()`.

   En el sistema hay múltiples *threads* constructores. Algunos construyen ruedas y otros construyen marcos. Cada vez que un *thread* ha construido una rueda, invoca la función `rueda_lista()`. Cada vez que un *thread* ha construido un marco, invoca la función `marco_listo()`. Cualquier *thread* puede llamar a la función, ya definida, `construir()` una vez que haya un marco y dos ruedas listas. Después que la función `construir()` ha terminado, exactamente una invocación a `marco_listo()` y dos invocaciones a `rueda_lista()` deben retornar.

   Escriba un código (C-like) para la función `rueda_lista()` y un código para la función `marco_listo()`, que permitan implementar el comportamiento descrito. Puede agregar las variables compartidas que considere necesarias. Puede utilizar *locks*, semáforos, y/o variables de condición, usando la API descrita.

   Su solución no debe permitir bloqueos, y debe ser libre de inanición (*starvation*).

   2p. Uso e inicialización de primitivas de sincronización (semáforos, locks, variables de condición)

   5p. Funcionamiento de `rueda_lista()`

   5p. Funcionamiento de `marco_listo()`

   El llamado a `construir()` puede producirse en cualquier de las dos funciones. En esta solución se llama en `marco_listo()` cada vez que se ha construido un marco, y que se han construido dos ruedas.

```
int contador = 0;
Semaphore semRuedas.init(0);
Semaphore semMarco.init(0);
Lock lock();
```

```
void rueda_lista() {
  lock.acquire();
  contador++;
  if (contador%2 != 0) {
    lock.release();
    semRueda.P();
  }
  else {
    semMarco.V();
    semRueda.P();
    lock.release();
  }
}
```

```
void marco_listo() {
  semMarco.P();
  construir();
  semRueda.V();
  semRueda.V();
}
```

En esta solución el contador libera a un marco solamente cuando es par (cada vez que han llegado dos ruedas más). Es importante liberar el *lock* antes de bloquearse, de lo contrario otra rueda no lo podría incrementar.

Otra solución, sin usar el contador:

```
void rueda_lista() {
  semMarco.V();
  semRueda.P();
}
```

```
void marco_listo() {
  lock.acquire();
  semMarco.P();
  semMarco.P();
  construir();
  semRueda.V();
  semRueda.V();
  lock.release();
}
```

En esta solución el *lock* es necesario en `marco_listo()` para que no entre dos marcos simultáneamente a la sección crítica.

2. **[15p]** Considere un sistema con páginas de 16KB, direcciones virtuales de 48 bit, direcciones físicas de 30 bit.

   2.1) **[2p]** Indique la cantidad máxima, en Byte, de memoria virtual que puede utilizar cada proceso, y la cantidad máxima, en Byte, de memoria física que puede direccionar el sistema.

   R.

   1p. Cantidad máxima de memoria virtual: $2^{48}\text{B} = 2^{8}\text{TB} = 256\text{TB}$

   1p. Cantidad máxima de memoria física: $2^{30}\text{B} = 1\text{GB}$.

   2.2) **[4p]** Calcule el tamaño, en Byte, de una tabla de páginas de 1 nivel para este sistema. Considere que se utilizan 4 bit para metadata: *valid*, *reference*, *dirty*, y *RW*. ¿Cabe la tabla de páginas en la memoria física?

   R.

   Tabla de páginas de 1 nivel. Dirección virtual: 48 bit. Tamaño de página: $16\text{KB} = 2^{4}\text{KB} = 2^{14}B$. Bit para *offset*: 14. Bit para número de página: $48 - 14 = 34$. Bit para número de *frame*: $30 - 14 = 16$.

   1p. Cantidad de PTEs: 2^{34} PTEs.

   1p. Tamaño de PTE: $16 + 4 = 20$ bit.

   1p. Tamaño de tabla de páginas: $2^{34} \times 20\text{bit} = 2^{31} \times 20\text{B} = 2 \times 20\text{GB} = 40\text{GB}$

   1p. La tabla de páginas (40GB) no cabe en una página de memoria física (16KB)

2.3) **[6p]** Proponga un esquema de paginación multinivel con la mínima cantidad de niveles necesarios, en el cual para cada nivel, una tabla de ese nivel cabe completamente en una página de memoria. Para justificar su solución debe indicar el tamaño de una tabla (en Byte) de cada nivel y dibujar el esquema de direccionamiento con las dimensiones de cada tabla. Puede suponer que la metadata se encuentra únicamente en la última tabla.

R.

Con un nivel, la tabla es de tamaño $40\text{GB} > 16\text{KB}$

Con dos niveles, se pueden usar 17 bit para cada nivel. La tabla del primer nivel es de tamaño $2^{17} \times 17\text{b} = 17 \times 2^{14}\text{B} = 17 \times 16\text{KB} > 16\text{KB}$. Por lo tanto, no es posible con dos niveles.

Con tres niveles, se puede usar 12 bit para un nivel y 11 bit para los otros dos. La tabla más grande podría ser la del último nivel, de tamaño $2^{12} \times 20\text{b} = 20 \times 2^{9}\text{B} = 10\text{KB} < 16\text{KB}$, que sí cabe en una página.

2.4) **[3p]** Calcule el tamaño, en Byte, de una tabla de páginas invertida de 1 nivel para este sistema. Suponga que los procesos utilizan 8 bit para almacenar su identificador.

R.

1p. Cantidad de PTEs: 2^{16}

1p. Tamaño de PTE: $34 + 4 + 8 = 46$ bit.

1p. Tamaño de tabla de páginas invertida: $2^{16} \times 46\text{bit} = 2^{13} \times 46\text{B} = 368\text{KB}$

3. **[15p]** Memoria y algoritmos de reemplazo.

3.1) **[3p]** ¿Qué aspecto del problema de direccionamiento de memoria se intenta resolver con el multinivel?

R.

Se intenta resolver el problema que las tablas de página que se producen en un esquema de un nivel no caben completamente en una página.

También se puede argumentar que no todas la memoria se usa simultáneamente y por lo tanto no se necesita mantener el registro de todas las páginas al mismo tiempo, por lo que tiene sentido dividir la tabla de páginas.

NO es correcto decir que es para direccionar más memoria que en un nivel.

3.2) **[3p]** ¿Qué relación hay entre el concepto de *working set* y el fenómeno de *thrashing*?

R.

El *working set* se utiliza para determinar una cantidad apropiada de *frames* que el proceso necesita mantener en memoria.

Si el sistema operativo asigna menos páginas que las necesarias (las indicadas por el *working set*, el proceso empezará a hacer *thrashing*, esto es, reemplazar páginas repetidamente para poder continuar su ejecución.

3.3) **[3p]** ¿Qué ventaja presenta la segmentación respecto a paginación? ¿Qué método se usa en los sistemas modernos?

R.

1,5p. Segmentación permite evitar fragmentación interna mejor que paginación, ya que el segmento se asigna del tamaño necesario. Esto se hace al costo de un sistema de traducción más complejo que el de las páginas ya que además de la dirección de inicio del segmento se debe almacenar su tamaño.

1,5p. Los sistemas modernas utilizan "segmentos paginados". Esto es, el espacio de direcciones del proceso se divide en segmentos, y cada segmento se almacena en páginas.

3.4) **[3p]** Se ha mencionado que LRU es un algoritmo que aproxima bastante bien la decisión óptima de qué *frame* conviene reemplazar. ¿Qué dificultades prácticas tiene la implementación de LRU? Mencione otro algoritmo que se implemente en lugar de LRU y por qué su implementación es mejor (si bien no entregue siempre un mejor resultado que LRU).

R.

1p. Dificultades de LRU: luego de cada acceso a una página es necesario recordar no solo el hecho que fue accedido (lo que se puede lograr con un *reference bit*), sino que también se debe almacenar el tiempo (*timestamp*) en que se accedió. Si un *timestamp* utiliza 8 Byte, se debe guardar eso junto con la tabla de páginas (aumento de tamaño) y actualizarlo en cada acceso (poco eficiente porque requiere interrupciones sobre una página que ya está en memoria). Aún si se hace por *hardware*, después hay que comparar todos los *timestamps*.

1p. Otro algoritmo puede ser: el de segunda oportunidad, el del reloj, el que usa un *reference Byte* (en lugar de bit) o *aging*, o la combinación de *dirty bit* con *reference bit*.

1p. Mencionar, dependiendo del algoritmo elegido, que requiere menos comparaciones, o que su actualización se puede efectuar por *hardware*, o que guarda menos información, o que requiere menos interrupciones.

3.5) **[3p]** Describa el significado y el uso de *valid bit*, *dirty bit* y *reference bit*. Si corresponde, incluya para qué algoritmo se usan.

R.

1p. *Valid bit*. Se usa para indicar si una entrada de tabla de páginas (o de TLB) está actualizada, o para indicar que la página está asignada a un *frame* en memoria física. Se usa para determinar si debe gatillarse una falta de página (*page fault*).

1p. *Dirty bit*. Se usa para indicar que una página se encuentra modificada respecto a la última vez que fue cargada en memoria (o respecto a su imagen en disco si la tiene. Se usa para como complemente al algoritmo de reloj, o junto con el *reference bit* al momento de priorizar una página que puede ser *swappeada*.

1p. *Reference bit*. Se usa para indicar que una página ha sido accedida. Puede haber sido modificada o no. Se usa para en algoritmo del reloj, o en el de segunda oportunidad, o en implementaciones que aproximan LRU (y en el mismo LRU).

4. **[15p]** Discos

4.1) **[3p]** ¿Cuál es la ventaja de utilizar un *buffer* a nivel del sistema operativo para comunicarse con un sistema de I/O (respecto a no tener *buffer*, o tenerlo solo a nivel de usuario)? Explique brevemente su respuesta.

R.

Al estar el *buffer* en *kernel space*, el usuario puede reutilizar esa memoria. No hay riesgo de que el espacio sea *swapped out* y después tenga que restaurarse (*swap-in*) para seguir leyendo el *buffer*.

(Máx 3p)

1,5p. Mencionar que el espacio puede ser reutilizado por el usuario.

1,5p. Mencionar utilidad de tener un *buffer*.

1,5p. Mencionar que el sistema operativo protege el acceso.

1,5p. Mencionar que permite que la transmisión, o escritura en el disco, ocurre de manera asíncrona.

NO es correcto decir únicamente que "es más eficiente" sin argumentar alguna de las situaciones mencioadas.

4.2) **[6p]** Para un conjunto de N discos, cada uno con capacidad C. Compare los arquitecturas RAID-0, RAID-1, RAID-5 y RAID-6, en cuanto a:

a) Capacidad máxima de almacenamiento

R.

2p. La mayor capacidad se obtiene con RAID-0. Luego vienen RAID-5 y RAID-6, y el que deja menos espacio es RAID-1. La respuesta está correcta mientras mantenga ese orden. Estrictamente hablando: RAID-0 $>$ RAID-5 $>$ RAID-6 $>$ RAID-1.

b) Cantidad máxima de discos que pueden fallar sin interrumpir el funcionamiento del sistema

R.

2p. RAID-1 permite que fallen hasta $N/2$ discos (si tenemos que fallen copias distintas), pero al menos permite que falle 1 disco. RAID-5 permite que falle 1. RAID-0 no puede permitir ninguna falla. RAID-6 permite que fallen 2. Definitivamente RAID-6 $>$ RAID-5 $>$ RAID-0. RAID-1 puede ir con RAID-5, o sobre RAID-6. Ambas son correctas si está mencionado por qué (aunque sea con una nota breve), y también podría aparecer dos veces.

c) Velocidad para escritura

R.

2p. RAID-0 es el más rápido ya que requiere escribir en un único disco. RAID-1 es el siguiente ya que requiere escribir en dos discos (lo que afecta las escrituras concurrentes). RAID-5 y RAID-6 son más lentos porque requieren escribir en dos discos además de calcular la paridad. Entre ellos, RAID-5 es más rápido en escrituras que RAID-6.

4.3) **[6p]** Considere la siguiente secuencia ordenada de solicitudes a cilindros del disco: 20, 44, 40, 4, 118, 12, 60. El disco posee 120 cilindros (numerados de 0 a 119), la cabeza lectora se encuentra posicionada en el cilindro 40, y antes de ello se había leído una posición en el cilindro 42. Para los siguientes algoritmos de scheduling de disco, determine la secuencia de lecturas y el desplazamiento total de la cabeza lectora

a) SSTF (*Shortest Seek Time First*)

R.

3p. Secuencia de lecturas: $40, 44, 60, 20, 12, 4, 118$

3p. Desplazamiento: $(60 - 40) + (60 - 4) + (118 - 4) = 190$

b) SCAN

R.

3p. Secuencia de lecturas: $40, 20, 12, 4, 0, 44, 60, 118$

3p. Desplazamiento: $40 + 118 = 158$

c) C-LOOK

R.

3p. Secuencia de lecturas: $40, 20, 12, 4, 118, 60, 44$

3p. Desplazamiento: $(40 - 4) + (118 - 4) + (118 - 44) = 36 + 114 + 74 = 224$

API para sincronización

`lock.acquire()`, `lock.release()`. Toma y libera un *lock*.

`semaphore.init(N)` inicializa un semáforo a `N`

`semaphore.P()`, ó `semaphore.down()` decrementa el contador del semáforo

`semaphore.V()`, ó `semaphore.up()` incrementa el contador del semáforo

`condition.wait(l)`, espera incondicionalmente en el *lock* `l`

`condition.signal()`, desbloquea a un *thread* que esté esperando en `condition`, si lo hay

Tantos los semáforos como las variables de condición implementan una cola FIFO para mantener a los *threads* bloqueados.

i	2^i B
6	64 B
7	128 B
8	256 B
9	512 B
10	1024 B

i	2^i B
10	1 KB
20	1 MB
30	1 GB
40	1 TB