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

IIC2333 — Sistemas Operativos y Redes — 1/2017

Soluciones Interrogación 1

Viernes 28-Abril-2017

Duración: 2 horas
SIN CALCULADORA

1. **[12p]** Consider el siguiente pseudocódigo (C-like) para solucionar el problema de la sección crítica entre n *threads* ejecutando en una única CPU. Las únicas instrucciones atómicas son las comparaciones y asignaciones individuales.

```
// variables compartidas
bool escogiendo[n] = {FALSE, FALSE, ... , FALSE};
int numero[n] = {0, 0, ..., 0};
```

El siguiente código es ejecutado por cada *thread* i.

```
1   do {
2     escogiendo[i] = TRUE;
3     numero[i] = max(numero[0], numero[1], ..., numero[n-1]) + 1;
4     escogiendo[i] = FALSE;
5     for (j=0; j<n; j++) {
6       while(escogiendo[j] == FALSE);
7       while(numero[j] > 0 && (numero[j] < numero[i]) );
8     }
9     /** SECCION CRITICA **/
10    numero[i] = 0;
11    /** resto del codigo **/
12  } while(true);
```

1.1) **[9p]** Argumente si esta propuesta cumple o no con cada una de las condiciones de solución al problema de la sección crítica.

R.

En la versión original de esta pregunta, la línea 6 decía `while(escogiendo[j] == TRUE);`.

La siguiente respuesta considera la línea 6 tal como fue publicada en la interrogación.

- **Exclusión mutua**. (3p) Cada *thread* se bloquea en la línea 6 al consultar por su propio estado `escogiendo`, y no puede liberarse. Hay exclusión mutua porque no hay más de un *thread* en su sección crítica.
- **Progreso**. (3p) No hay progreso. Si más de un *thread* desea entrar, ninguno puede hacerlo porque se bloquean.
- **Espera acotada**. (3p) No hay espera acotada. Ya que nadie puede entrar, el tiempo que deben esperar para entrar es infinito.

La siguiente respuesta considera que la línea 6 no bloquea a los *threads*, como era en la versión original. Es posible argumentar estas respuestas en caso que hayan utilizado el supuesto que la línea 6 decía `while(escogiendo[j] == TRUE);`

- **Exclusión mutua**. (3p) No hay exclusión mutua por el hecho que la instrucción `max` no es atómica. Eso genera que más de un *thread* pueda obtener el mismo valor en `numero[i]`. Para dos *threads* `a` y `b` con el mismo valor en `numero[a]` y `numero[b]`, ninguno se bloqueará en la segunda condición de la línea 7 y por lo tanto ambos entrarán a la sección crítica.
- **Progreso**. (3p). Sí hay progreso. Cuando más de un *thread* desea entrar, para al menos uno de ellos se deja de cumplir la segunda condición de la línea 7. Al menos uno puede entrar.
- **Espera acotada**. (3p). Sí hay espera acotada. En caso que un *thread* tenga el número más alto, deberá dejar pasar a lo más `n-1` *threads*. Cualquier que *thread* `b` que ejecute la línea 3 después de un *thread* `a` obtendrá un valor tal que `numero[b] >= numero[a]`, por lo tanto el *thread* `a` entrará antes.

Incorrecto suponer que la ejecución de `max` es atómica: 0p.

1.2) **[3p]** Indique cómo mejorar esta solución para que cumpla con todas las condiciones del problema de la sección crítica. No es necesario argumentar por las condiciones que se siguen cumpliendo.

R.

Para la versión publicada de la pregunta, se deben ejecutar al menos dos cambios: (1) (1.5p), reemplazar en la línea 6, `while(escogiendo[j] == FALSE);` por `while(escogiendo[j] == TRUE);`. Eso evita el bloqueo. Después de eso no se cumple exclusión mutua, pero sí progreso y espera acotada. El segundo cambio (2) (1.5p) implica agregar una condición a la línea 7 para desempatar en el caso que dos procesos hayan obtenido el mismo resultado al llamar a `max`. Puede quedar como `while(numero[j] >0 && (numero[j] <numero[i] || (numero[j]==numero[i] && j<i) ));`. De esta manera, el "desempate" se efectúa de acuerdo al identificador de cada *thread*, que es único, por lo tanto para exactamente uno de ellos la condición no se cumplirá y podrá avanzar a su sección crítica. Con esto sí se cumple exclusión mutua. Las otros condiciones siguen cumpliéndose en particular espera acotada porque, en el caso que todos hayan obtenido el mismo valor en `numero`, la discriminación se hará exclusivamente por el identificador de cada *thread* y el peor caso para un *thread* sigue siendo `n-1` *threads* con identificador mayor.

Para la versión original de la pregunta, considerando el supuesto que `while(escogiendo[j] == TRUE);`, los 3p corresponden a la justificación (2) del párrafo anterior.

Es posible utilizar un *lock* o un semáforo inicializado en 1 para proteger la ejecución de `max`. Si está bien protegida, eso basta para asegurar exclusión mutua.

2. **[7p]** El siguiente código intenta ser replicar un comando `N` veces **en paralelo**.

```
char *command;
char *arguments;
// puede agregar mas variables aqui
while('r') {
  read_command(&command, &arguments); // lee correctamente el comando y sus argumentos
  /** completar codigo aqui **/

}
```

Complete este código, de manera que luego de ejecutar los `N` procesos en **paralelo**, el programa escriba en pantalla cuántos terminaron correctamente (esto es, su *exit code* es 0), y cuántos no.

R.

Una posible solución. Importante: debe poder haber un momento en la ejecución que permita que estén los N procesos corriendo en paralelo. No es lo mismo que *obligar* a que estén todos corriendo en paralelo, ya que eso depende del *scheduling*. Una solución que itera en N, hace `fork` y espera en cada iteración, **NO** es correcta.

```
char *command;
char *arguments;
```

```
pid_t children[N];
int ok, failed, exitStatus;
while('r') {
  read_command(&command, &arguments); // lee correctamente el comando y sus argumentos
  ok = 0; failed = 0;
  for(int i=0; i<N; i++) {
    children[i] = fork();
    if(children[i] == 0) {         // child
      exec(command, arguments);    // exit code se recupera en wait
      exit(-1);                    // si exec falla, retornamos -1. Hijo no debe continuar
      // child nunca llega aqui
    }
  }
  for(int i=0; i<N; i++) {          // parent, luego de haber creado los N hijos
    if(children > 0) {              // se asegura que el hijo es valido y no falla fork
      wait(children[i], &exitStatus);  // espera por todos
      if(exitStatus == 0)
        ok++;
      else
        failed++;
    }
    else                            // tambien es una falla si el fork falla
      failed++;
  }
  print("OK=%d, FAILED=%d\n", ok, failed);
}
```

Puntaje es la suma de (mínimo 0p):

- 3p. Crear y ejecutar exactamente `N` procesos hijo
- 2p. Esperar los `N` procesos
- 1p. Recolectar los exit code
- 1p. Contabilizar e imprimir los exit code
- -1p no considerar que `fork()` y `exec()` pueden fallar. Esto **NO es lo mismo** que suponer que el código de retorno del comando ejecutado por `exec` no va a fallar.
- -4p. No permitir que los `N` procesos estén creados simultáneamente, por ejemplo, con un `fork/wait` secuencial.

3. **[11p]** Deadlock y *scheduling*

   - **[3p]** El algoritmo del banquero tiene como premisa mantener al sistema en un estado seguro para evitar *deadlocks*. Sin embargo, un estado inseguro no es necesariamente un estado de *deadlock*. ¿Por qué se desea mantener al sistema en un estado seguro?

     R. Máximo 3 puntos sumando:

     - 0p. Explicar el algoritmo del banquero, lo que hace, o escribir el algoritmo. No es eso lo que se pregunta.
     - 1p. Mencionar que un estado inseguro *puede* convertirse en estado de *deadlock*, y que un estado de seguro *nunca* puede convertirse en estado de *deadlock*.
     - 2p. Mencionar bajo qué condiciones un estado inseguro se convierte en *deadlock*: una secuencia de *requests* de los procesos, o bien si *todos* los procesos solicitan los recursos que faltan; o bien, mencionar por qué un estado seguro *nunca* puede convertise en estado de *deadlock*: hay una secuencia en que todos pueden terminar si todos piden los recursos que les faltan.

- 1p. Explicarlo mediante un ejemplo.

- **[3p]** ¿Por qué un *scheduler* FCFS (*First-Come First-Serve)* no es apropiado para procesos interactivos? Mencione uno que sí lo sea, y justifique brevemente por qué.

  R.

  - 0p. Definir FCFS, o proceso interactivo. No es lo que se pregunta.
  - 1p. Justificación: no se puede predecir el tiempo de espera en un FCFS; o bien ejemplificarlo con un caso malo (proceso largo llega antes que uno más corto); o bien otra explicación correcta
  - 1p. Ejemplo: RR, MLFQ, Prioritario si dan una política para la cola principal, ó algún otro. Puede ser también, O(1), CFS, BFS. Incorrectos: SJF, EDF, RM, SJN, Proportional, Lottery
  - 1p. Mencionar que el algoritmo elegido tiene tiempo de espera bajo, o acotado, para un proceso nuevo o que vuelve de *waiting*. No es necesario explicar todo el algoritmo.

- **[5p]** De los siguientes elementos, ¿cuáles deben ir en un PCB, y cuáles no? (solo mencionarlo): variables locales de funciones, ubicación de la tabla de páginas, prioridad de *scheduling*, código binario del proceso, ubicación de TLB.

  R.

  - 1p. NO. Variables locales de funciones. Van en *stack*.
  - 1p. SI. Ubicación de la tabla de páginas. Va en registro PTBR.
  - 1p. SI. Prioridad de *scheduling*.
  - 1p. NO. Código binario. Va en *code*.
  - 1p. NO. Ubicación de TLB. TLB es única para todos. Va en memoria del Sistema Operativo.

4. **[30p]** Considere un sistema computacional para soportar hasta 1024 procesos concurrentes, todos con un comportamiento similar. El *hardware* está diseñado para soportar hasta 8GB de memoria física, y el sistema operativo configurado para manejar páginas de 16KB. El espacio de direcciones virtuales utiliza 46 bit.

   4.1) **[7p]** Diseñe un sistema de direccionamiento de memoria usando tabla de páginas de un nivel. Especifique, describiendo claramente su cálculo, la cantidad de bit para *offset*, número de página, número de *frame*, tamaño en memoria de la tabla de páginas, y cantidad total de memoria virtual direccionable.

   Incluya al menos 3 bit de metadata, indicando cuáles son.

   Considere además que la arquitectura requiere que cada entrada en la tabla de páginas (PTE) debe ser de un tamaño que sea múltiplo de 1 Byte (8 bit). Esto significa que puede necesitar agregar bits adicionales (*padding* o alineamiento).

   R. Se puede hacer con un diagrama. Se deben indicar los siguientes datos:

   - 1p. Bit para *offset*. Páginas de $16\text{KB} = 2^{14}\text{B}$. 14 bit para *offset*.
   - 1p. Bit para #página. $46 - 14 = 32$
   - 1p. Bit para #*frame*. $8\text{GB} = 2^{33}\text{B}$. 33 bit para dirección física. $33 - 14 = 19$ bit para #*frame*
   - 3p. Tamaño tabla de páginas. Número de páginas: 2^{32} páginas. Tamaño de PTE: 19 bit (frame), 3 bit de metadata. Hasta ahí son 22 bit. Con 2 bit más se llega a 24 bit = 3B. Tamaño de tabla de páginas: $2^{32} \times 3\text{B} = 12\text{GB}$.
   - 1p. Bit de metadata pueden ser: present/absent, valid, dirty, protected, RW/RO, reference. Puede haber más de un bit de reference.

   4.2) **[7p]** Modifique el diseño de la tabla de páginas utilizando paginación multinivel, de manera que cada tabla en cada nivel quepa **completamente en una página**.

   Especifique la cantidad de bit y los tamaños de tabla en cada nivel, e indique la cantidad mínima de memoria física necesaria para poder hacer una traducción.

   R. Se puede hacer con un diagrama. Se debe indicar cada nivel. Se necesitan **al menos** 3 niveles. Se puede hacer con más, pero también requiere escribir más.

- 3p. Último nivel debe tener máximo 12 bit. Tabla de páginas con 12 bit en el último nivel ocupa $2^{12} \times 3\text{B} = 12\text{KB}$. Si se usan 13 bit, la tabla de páginas del último nivel ocupa $2^{13} \times 3\text{B} = 24\text{KB}$.
- 1p. Solo el último nivel debe considerar los bit de metadata.
- 2p. Cantidad mínima de memoria física: depende de cuántos niveles se usaron. Es la suma de una página en cada nivel. Ejemplo: niveles 6, 12, 12. Tabla en primer nivel: 2^6 PTEs, y cada PTE ocupa 12 bit. Padding con 4 bit para completar 2B. Tabla de páginas nivel 1: $2^6 \times 2\text{B} = 128\text{B}$. Tabla en segundo nivel: 2^{12} PTEs, y cada PTE ocupa $12 + 4$ bit $= 2\text{B}$. Tabla de páginas nivel 2: $2^{12} \times 2\text{B} = 8\text{KB}$. Sumando una página por cada nivel: $128\text{B} + 8\text{KB} + 12\text{KB} = 20\text{KB} + 128\text{B}$.
- 1p. Justificación: cálculo correcto de páginas en cada nivel.

4.3) **[4p]** ¿Cuánto espacio ocuparía en memoria utilizar una tabla de páginas invertida?

R.

- 1p. 19 bit para #*frame*. 2^{19} *frame* posibles. Tabla de páginas tiene 2^{19} entradas.
- 1p. Cada entrada requiere: 10bit para PID (máximo 1024 procesos), 32 bit para #página, 3 bit de metadata.
- 1p. Hasta ahí van 45 bit. Con 3 bit de alineamiento se llega $48\text{bit} = 6\text{B}$.
- 1p. Tabla invertida ocupa $2^{19} \times 6\text{B} = 3\text{MB}$

4.4) **[3p]** De acuerdo a los bit que especificó en la primera parte, ¿qué algoritmo de reemplazo de páginas utilizaría, y por qué?

R.

Depende de los bit de metada declarados. El único prohibido es MIN (u OPT). Algunas combinaciones válidas:

- *reference* bit. LRU usando reloj, o segunda oportunidad.
- Más de un *reference* bit. LRU usando *Aging*.
- *dirty bit*. LRU usando reloj con criterios aumentados (reference + dirty).
- Se puede mencionar algoritmo *random* si agrega algún supuesto relativo a localidad de referencia.
- Otra combinación correcta.
- -1p. Mencionar LRU sin ninguna implementación real. LRU no se implementa directamente porque es costoso.

4.5) **[3p]** Suponga que el $90\,\%$ de los accesos a memoria corresponde a un conjunto de 10 páginas. En el esquema de una tabla con un nivel, ¿de qué tamaño (en Byte) debería ser el TLB para tener un $90\,\%$ de *hit rate*? Justifique su respuesta. Recuerde que cada entrada debe ser del tamaño de un múltiplo de 1 Byte.

R.

Se puede especicar con un diagrama.

- 1p. Al menos 10 entradas en TLB.
- 0.5p. Cada PTE en el esquema de un nivel: 3B
- 0.5p. Bit para el #página: $32\text{bit} = 4\text{B}$
- 1p. Tamaño TLB: $10 \times 7\text{B} = 70\text{B}$.

4.6) **[3p]** Suponiendo que una consulta al TLB toma Ans, y un acceso a memoria toma Bns, ¿cuánto sería el tiempo de acceso promedio a memoria usando el TLB de la pregunta anterior?

R.

$T_{\text{prom}} = 0{,}9 \times (A + B)\text{ns} + 0{,}1 \times (A + 2B)\text{ns}$

4.7) **[3p]** ¿Cuál debería ser un criterio para el *long term scheduler* al momento de decidir si permite aceptar un proceso para ejecución?

R.

Si un proceso ocupa repetidamente 10 páginas de 16KB cada una, entonces su *working set* requiere al menos 160KB. Si hay 160KB libres, debe aceptar al proceso, si no puede generar *thrashing*.

API de procesos

- `pid_t fork()` retorna 0, en el contexto del hijo; retorn *pid* del hijo, en el contexto del padre.
- `int exec(char *command, char *argumentos)` recibe como parámetro un *string* con la ruta del archivo a ejecutar y sus argumentos. Si hay error retorna -1. De lo contario, no retorna.
- `pid_t wait(pid_t p, int *exitStatus)` espera por el proceso `p`, y guarda el estado de salida de `p` en `exitStatus`. Si `p` es -1, espera por cualquiera. Retorna el `pid` del proceso que hizo `exit`.

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

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