Auxiliar 2

28 de octubre de 2011

1. Sincronización

1.1. Barreras

Supongamos que existe un cálculo complejo que queremos realizar a través de varios threads. El problema es que a veces se requiere que todos los threads alcancen un punto común en el desarrollo del cálculo antes de seguir. Para ello, queremos implementar una función `barrier()` que detenga a los threads en ese punto de su ejecución y que sólo los deje continuar cuando todos hayan alcanzado el "punto de reunión". En código "Pintos-like", se tendría en cada thread algo como:

```
super_calculo() {
        thread *me;
        int my_id;

        me = thread_current();
        my_id = me->tid;

        parte_1_del_calculo();
        barrier(my_id);
        parte_2_del_calculo();
}
```

(para efectos del problema supondremos que las *id* de los threads son `int` en vez de `tid_t`)

Se pide implementar la función `void barrier(int thread_id)` utilizando sólo semáforos (es decir, ninguna otra variable global ni mecanismo de sincronización). Suponga que existen `NUM_THREADS` threads ejecutándose, que `barrier()` sólo se invoca una vez en el cálculo de cada *thread* y que los valores de id van desde 0 a `NUM_THREADS` $-$ `1`.

1.2. ¿Tírate un qué?

Tenemos un paso cordillerano de una sola vía. Vienen *autos-threads* por ambos lados y demoran un tiempo en pasar. Usamos, entonces, monitores/condiciones para implementar los "bandereros" del camino:

```
struct paso {
        struct lock lock;
        struct condition esta_vacio;
        int n_desde_E, n_desde_O;
}
void entrar_por_O(struct paso *paso) {
        lock_acquire(&(paso->lock));
        while(paso->n_desde_E)
                cond_wait(&(paso->esta_vacio), &(paso->lock));
        paso->n_desde_O++;
```

```
        lock_release(paso->mon);
}
void salir_desde_O(struct paso *paso) {
        lock_acquire(&(paso->lock));
        if(!(--(paso->n_desde_O)))
                cond_broadcast(&(paso->esta_vacio), &(paso->lock));
        }
        lock_release(&(paso->lock));
}
//funciones para E son análogas
```

Explique por qué esta solución puede generar hambruna. Luego modifique el código, implementando el siguiente algoritmo:

- Partimos por dejar pasar autos desde alguna dirección (por ejemplo, desde O). Mientras lleguen autos de la dirección elegida, deja pasar autos.
- Si llega alguien por el otro lado (por ej., desde E):
  - Si no queda gente en el paso, los deja pasar, cambiando la bandera de color.
  - Si queda gente en el paso, se mantiene igual hasta que hayan pasado al menos 5 seguidos en la dirección que ya se tenía (desde O). Luego bloquea esa entrada, espera a que se vacíe el paso y luego deja pasar desde el otro lado (E).

¿Resuelve esto la hambruna?

1.3. Muchos Threads (Propuesto)

Considere un sistema que tiene dos tipos de *threads, A* y *B*. Existe una gran cantidad de cada tipo, y todos están intentando invocar a `foo`, que se define así:

```
enum {A, B} thread_type;
semaphore sem_x;
semaphore sem_y;

sems_init() {
        sema_init(&sem_x, 0);
        sema_init(&sem_y, 0);
}

void foo(thread_type type) {
        if (type == A) {
                sema_down(&sem_x);
                sema_up(&sem_y);
        }

        if(type == B) {
                sema_up(&sem_x);
                sema_up(&sem_x);
                sema_down(&sem_y);
        }
        bar(); // Esta fn hace algun trabajo
}
```

1. ¿Qué se puede decir sobre los threads que se encuentran ejecutando `bar`()?
2. Reimplemente `foo`() utilizando un monitor (es decir, a través de condiciones).

2. Estrategias de Scheduling

2.1. Conceptos

Para las preguntas conceptuales puede ser útil tener frescos algunos conceptos como hambruna (la denegación eterna de recursos a ciertos procesos), fairness (que eventualmente todo proceso tenga acceso a todo recurso que pida), throughput (número de procesos terminados por unidad de tiempo), tiempo de respuesta (la demora entre input del usuario y el poder ver el efecto de éste), tiempo de espera (tiempo que se pasa en la ready queue) y turnaround time (tiempo para completar el proceso). También sirve recordar que existe I/O.

- ¿Cuál es el objetivo de "First come, first served"? ¿Cuáles son sus pro y contra?

FCFS es una respuesta "default" al problema de scheduling. La simplicidad de implementación es una de las ventajas de esta estrategia. Otra es que minimiza el tiempo de respuesta: una vez que un proceso toma la CPU, responderá de forma prácticamente instantánea. Los problemas que tiene es que las otras métricas sufren: los procesos pueden demorar mucho en obtener la CPU, puede haber hambruna ante un proceso que no quiera terminar, etc.

- ¿Cuál es el objetivo de Round-robin? ¿Cuáles son sus pro y contra?

Round-robin apunta a resolver de forma simple el problema de la hambruna, ya que todo proceso eventualmente será elegido para tener la CPU. Un problema desagradable se produce cuando se tienen muchos procesos que duran lo mismo, ya que el turnaround time de cada proceso se comienza a aproximar al total de ejecución de todos éstos (¡el peor caso posible!),

- ¿Cuál es el objetivo de "Shortest job first"? ¿Cuáles son sus pro y contra?

Shortest job first minimiza el tiempo de espera promedio y maximiza el throughput de procesos, pero no necesariamente el turnaround time. Además, como no se puede saber a priori la duración del proceso, es necesario hacer estimaciones. Otro problema, más crítico, es la presencia de hambruna para procesos largos ante la llegada de mucho procesos de corta duración.

- ¿Cuál es el objetivo del multilevel-feedback queueing? ¿Cuáles son sus pro y contra?

MLFQ busca minimizar el tiempo de respuesta de los procesos intensivos en I/O y que usan poca CPU, pues posiblemente son procesos que interactúan con el usuario. Se busca también evitar la hambruna mediante la modificación de prioridades. Una cualidad de esta estrategia, además de los objetivos perseguidos, es que se adapta a la situación del sistema. Sin embargo, un proceso inteligente puede engañar al scheduler y alterar su prioridad; además, puede producirse inversión de prioridades al no ser posible donarlas.

- ¿Cuál es el objetivo del Lottery scheduling? ¿Cuáles son sus pro y contra?

Lottery scheduling resuelve el problema de la hambruna, ya que todo proceso tiene una probabilidad de obtener la CPU. Además, es capaz de realizar donación de prioridades (evitando la inversión de prioridades) y controlar la cantidad promedio de CPU que recibe cada proceso (evitando el aprovechamiento por parte de procesos astutos). Ciertas variantes ayudan a mejorar otros aspectos como utilización de I/O. Un problema es que, al ser un algoritmo probabilístico, se producen latencias impredecibles.

2.2. Multilevel Feedback Queues Scheduling

Considere la siguiente variante de MLFQS:

- Cuando un proceso nace, lo hace al final de la cola de mayor prioridad.
- Si un proceso usa toda su tajada de tiempo, su prioridad disminuye en 1.
- Si un proceso cede la CPU antes de usar toda su tajada de tiempo, su prioridad queda igual.

- Si el proceso no usa toda su tajada por tener que utilizar I/O, su prioridad crece en 1.
- En general, a los procesos con mayor prioridad se les dan tajadas de tiempo más pequeñas.

Supongamos que tenemos un sistema que utiliza esta estrategia de scheduling. Suponga que existen 8 colas, teniendo la de más alta prioridad una tajada de tiempo de 10 ms; la segunda, 20 ms; la tercera, 40 y así. La máquina tiene una CPU que ejecuta 10 MIPS (millones de instrucciones por segundo) y el disco responde en 15 ms. Tenemos dos procesos A y B ejecutándose en el sistema. Analice los siguientes casos:

- A es intensivo en I/O, leyendo un bloque de disco para luego trabajar en el bloque por 25 ms; B es intensivo en CPU, pero cede la CPU cada 100 instrucciones que ejecuta pues se le obliga a acceder al disco (está haciendo lo que conocerán como *trashing*). ¿Qué sucede? Explique.

En régimen permanente, el proceso A oscilará entre las colas de 20 y 40 ms, pudiendo usar todo el slice de 20 ms (y pasando a la cola de 40 ms) pero siempre teniendo que hacer operaciones de I/O antes de terminarse el slice de 40 ms (por lo tanto subiendo a la cola de 20 ms). El proceso B se mantiene en la primera cola, ya que usa $\frac{100}{10^7} = 10^{-5}$ *s de su slice, es decir, 0.01 ms. de su slice de 10 ms antes de ser obligado a hacer I/O.*

Noten dos cosas. Si bien B es intensivo en CPU, de todas formas debe hacer operaciones de I/O (está mandando y recuperando páginas a/de swap, como verán más adelante) por lo que el scheduler lo trata como intensivo en I/O. Por otro lado, aunque B tenga siempre mayor prioridad que A, este último sí logra obtener tiempo de ejecución y avanzar.

- A trabaja 100 % en la CPU, mientras que B se ve obligado a cederla cada 100 instrucciones que ejecuta (de nuevo, *trashing*). ¿Qué sucede? ¿Cuál proceso utiliza la CPU durante mayor tiempo?

De nuevo, B se mantiene como el proceso de máxima prioridad, quedándose en la cola de 10 ms. El proceso A siempre usará toda su slice de tiempo, por lo que caerá hasta la cola de menos prioridad, siguiendo las reglas del scheduler.

¿Cuál proceso usa más tiempo de CPU? Si bien B tiene mayor prioridad que A, sólo alcanza a ejecutarse por 0.01 ms antes de ceder la CPU. Al no haber más procesos, correrá A, entregándosele una tajada grande de tiempo de procesador. Así, A terminará por usar más la CPU que B.

- ¿Qué defectos puede tener esta variante?

Un proceso que conozca la especificación del scheduler puede engañarlo, por ejemplo, intercalando operaciones inútiles de I/O cerca del final de su slice. Así, se mantendrá con una alta prioridad.

Se pueden señalar también detalles concernientes a MLFQ en general. Por ejemplo, existe la imposibilidad de realizar donación de prioridades.