Sistemas Operativos

Práctica 3 bis: Razonamiento sobre programas paralelos

Notas preliminares

- Esta guía de ejercicio contiene la mayor parte de las soluciones en el apéndice. Para que tenga sentido es importante intentar resover los ejericios **y recién luego** consultar el apéndice.
- Las demostraciones que allí se muestran no son las únicas correctas, por lo que se recomienda consultar en clase las que hayan elaborado.

Parte 1 – Problemas clásicos

Ejercicio 1 (*Problema de los prisioneros*)

P prisioneros son encarcelados. Para salir de prisión se les propone el siguiente problema.

- Los prisioneros tienen un día para planear una estrategia. Después, permanecerán en celdas aisladas *sin ninguna* comunicación.
- Hay una sala con una luz y un interruptor. La luz puede estar prendida (interruptor *on*) o apagada (interruptor *off*).
- De vez en cuando, un prisionero es llevado a esa sala y tiene derecho cambiar el estado del interruptor o dejarlo como está.
- Se garantiza que *todo* prisionero va a entrar a la sala *infinitamente* seguido.
- En cualquier momento, cualquier prisionero puede declarar "*todos los prisineros hemos visitado la sala al menos una vez*".
- Si la declaración es correcta, los prisioneros serán liberados. Si no, quedarán encerrados para siempre.

El problema de los prisioneros consiste en definir una estrategia que permita liberar a los prisioneros sabiendo que el estado inicial del interruptor es *off* (luz apagada) y considerando que no todos los prisioneros tienen por qué hacer lo mismo.

Considere la siguiente estrategia de solución:

Sea $\mathcal{P} = \{1, \ldots, P\}$ el conjunto de prisioneros y $E \in \{on, off\}$ el estado del interruptor. El prisionero $p = 1$ es el encargado de contar y hacer la declaración. La estrategia es la siguiente:

- Para todo $p \neq 1$. Si $E = off$, p no hace nada. Si $E = on$, hay dos casos. Si es la primera vez que p ve el interruptor en *on* entonces lo pone en *off*. Si no, lo deja como está. Esto es, todo p cambia el interruptor exactamente una vez.
- Para $p = 1$. p mantiene un contador C, inicialmente en 0. Si $E = off$, p lo pone en *on* e incrementa el contador. Si no, no hace nada. Si el contador llega a P, p hace la declaración.

a) Demuestre que la estrategia es correcta. Para ello considere una secuencia τ de eventos que termina en la declaración y divida τ en P segmentos $\tau^1 \cdots \tau^P$, tal que el último estado de τ^i es (on, i) y el estado previo a éste es $(off, i-1)$.

Ejercicio 2 (*Turnos*)

Hay $N \geq 2$ procesos, numerados de 0 a $N-1$. Cada proceso i ejecuta la función `a()` en algún momento durante su corrida. La acción de ejecutar `a()` por el proceso i se denota a_i.

Considere la siguiente solución con semáforos que usa un array `sem[]` de N semáforos. Inicialmente, `sem`$[i]$ es 1 si $i = 0$ y 0 para todo $i \neq 0$. Cada proceso i ejecuta el siguiente código, donde los comentarios R, T, C y E corresponden a los estados del modelo de Lynch.

```
sem sem[N];

void turno(pid i) {
  // R
  // try_i
  // T
  sem[i].wait();
  // crit_i
  // C
  a(); // a_i
  // exit_i
  // E
  if (i < N-1) sem[i+1].signal();
  // signal_{i+1}
}
```

Se pide:

a) La solución presentada, ¿cumple con la propiedad de **WAIT-FREE**?
b) Pruebe que la solución presenta cumple con la propiedad de **ORDEN**. Es decir, que cualquiera sea $0 \leq i < N$, dos procesos i e $i+1$ no se pueden solapar en la ejecución de a. Esto es, que `a()` es una *sección crítica* que se ejecuta en *exclusión mutua*.
c) Probar que todo proceso i ejecuta a_i y termina (**G-PROG**).

Ejercicio 3 (*Turnos con espacio en* $\mathcal{O}(1)$)

Una manera de hacerlo en $\mathcal{O}(1)$ es usando un registro atómico `turno` con la operación `getAndInc()`, inicializado en 0. Esta solución tiene *busy waiting*.

```
atomic<int> turno = 0;

void turno(pid i) {
  // R
  // try_i
  // T
  while (turno < i) {}; // busy waiting
  // crit_i
  // C
  a(); // a_i
  // exit_i
  // E
  turno.getAndInc();
  // signal_{i+1}
}
```

a) Probar que esta solución satisface **ORDEN**.
b) Probar que también satisface **G-PROG**.

Parte 2 – Código

Implementación de la estrategia propuesta para el problema de los **prisioneros**:

```
#define NADIE −1

atomic<int> sala, prisionero[] = NADIE, 0;
atomic<bool> libres, declaracion, luz = false;

void guardia() {
  while (!declaracion) {}
  int cant = 0;
  for (i = 0; i < N; i++) cant += prisionero[i];
  if (cant == N) libres = true;
}

void p(int i) {
  int contador = 0;
  while (!libres) {
    // esperar a ser llevado a la sala
    while (sala.compareAndSwap(NADIE, i) != NADIE) {}
    // assert(sala == i);
    // entrar en la sala
    prisionero[i] = 1;
    if (i>0) { // no es el líder
      if (luz && ++contador == 1) luz = false;
    } else { // líder
      if (!luz) {
        luz = true;
        if (++contador == N) libres = true;
      }
    }
    // salir de la sala
    sala = NADIE;
  }
}
```

Implementación del problema de turnos con espacio en $\mathcal{O}(1)$ y sin `getAndInc()`.

```
atomic<int> turno = 0;

void turno(pid i) {
  // R
  // try_i
  // T
  while (turno < i) {}; // busy waiting
  // crit_i
  // C
  a(); // a_i
  // exit_i
  // E
  int tmp = turno; tmp += 1; turno = tmp;
  // signal_{i+1}
}
```

Parte 3 – Demostraciones

Prisioneros

Consideremos una secuencia τ de eventos que termina en la declaración. Podemos dividir τ en P segmentos $\tau^1 \cdots \tau^P$, tal que el último estado de τ^i es (on, i) y el estado previo a éste es $(\mathit{off}, i-1)$. En cada segmento τ^i, un prisionero p^i cambia el estado del interruptor. Probemos que $p^i \neq p^j$ para todo $i \neq j$. El estado inicial de τ es $(\mathit{off}, 0)$. τ^1 termina con la primera entrada del prisionero p^1 a la sala. Si algún prisionero distinto de 1 entró a la sala en τ^1, no cambió el estado del interruptor porque estaba en *off*. Entonces $p^1 = 1$. En τ^2, el prisionero p^2 cambió el estado del interruptor en algún momento. Claramente $p^2 \neq p^1$. Siguiendo este razonamiento, dado que cada prisionero distinto de 1 cambia el estado del interruptor exactamente una vez, podemos probar que para todo $1 \leq i \leq P$, $p^i \neq p^j$ para todo $1 \leq j < i$. Entonces, en τ todos los prisioneros entraron al menos una vez cada uno a la sala.

Turnos

ORDEN Probamos que cualquiera sea $0 \leq i < N$, dos procesos i e $i+1$ no se pueden solapar en la ejecución de a, esto es, que `a()` es una *sección crítica* que se ejecuta en *exclusión mutua*. Procedemos por el absurdo. Supongamos que existe una ejecución en la que i e $i+1$ están en C al mismo tiempo. Más formalmente, existe una secuencia $\tau_0 \to \tau_1 \dots$ y un k tal que $\tau_k(i) = \tau_k(i+1) = C$. Los estados de esa secuencia guardan información de lugar dónde cada proceso i está con respecto al modelo (esto es, R, T, C o E) y además el valor del semáforo `sem[`i`]`. Retomando el hilo de la prueba, se desprende entonces que existe un estado previo, digamos $\tau_{k'}$, $k' < k$, en el cual $\texttt{sem}[i] = \texttt{sem}[i+1] = 1$. Formalmente, deberíamos escribir: $\tau_{k'}(\texttt{sem}[\text{i}]) = \tau_{k'}(\texttt{sem}[\text{i+1}]) = 1$, pero lo omitimos para no complicar demasiado la prueba. Dado que en el estado inicial τ_0 el valor del semáforo $\texttt{sem}[i+1]$ es 0 puesto que $i+1 > 0$, necesariamente tiene que haber ocurrido signal_{i+1} previamente. Más formalmente, tiene que existir $k'' < k'$ tal que $\tau_{k''} \xrightarrow{\mathit{signal}_{i+1}} \tau_{k''+1}$. Por lo tanto, a_i tiene que haber terminado antes de k'' y por lo tanto, antes de k, dado que a_i ocurre necesariamente *antes* que signal_{i+1}. En términos formales, existe un $k''' < k''$ tal que $\tau_{k'''} \xrightarrow{a_i} \tau_{k'''+1}$. Por lo tanto, $\tau_k(i) \neq C$, lo que contradice la hipótesis. $\square$

Observemos que la demostración de **ORDEN** no prueba que la ejecución ordenada existe, sino sólo que no puebe haber ejecuciones desordenadas. La propiedad **ORDEN** se cumple aunque el conjunto de ejecuciones sea vacío. De hecho, en ningún momento se usa la propiedad que `sem[0]` $= 1$, sino sólo que `sem[`i`]` $= 0$ para todo $i > 0$.

La demostración siguiente prueba que existe una corrida en la que los a se ejecutan (en orden). Pero para hacerlo, necesitamos asumir que `a()` se ejecuta en un tiempo finito. La demostración consiste en *construir* una secuencia de manera inductiva, esto es, construyendo un prefijo y extendiéndolo.

G-PROG Suponemos que `a()` termina para todo i. La prueba es por inducción. Es trivial para $i = 0$ dado que inicialmente $\texttt{sem}[0] = 1$. Más formalmente, existe un prefijo (esto eso, una subsecuencia finita), llamémoslo $\tau_{<0>}$ que termina con la transición signal_1. Supongamos que vale para n. Esto es, existe un prefijo $\tau_{<n>}$ que termina con la transición signal_{n+1}. Tenemos que considerar dos casos.[1]

1. En el estado final de $\tau_{<n>}$ el proceso $n+1$ está en T, i.e., $\tau_{<n>}(n+1) = T$. Esto es, la transición try_{n+1} ocurre en $\tau_{<n>}$.

2. En el estado final de $\tau_{<n>}$ el proceso $n+1$ no está en T, i.e., $\tau_{<n>}(n+1) \neq T$. Entonces, el prefijo $\tau_{<n>}$ se puede extender con una secuencia finita de estados cuya última transición es try_{n+1}.

Entonces, el prefijo $\tau_{<n>}$ se puede extender a una subsecuencia $\tau_{<n>}\tau'$ en cuyo último estado la transición crit_{n+1} puede ocurrir. Esto es así dado que `sem[`$n+1$`]` $= 1$, puesto que ocurrió signal_{n+1} en el prefijo $\tau_{<n>}\tau'$. Por lo tanto, en algún momento en el futuro el proceso $n+1$ está en C. Dado que, por hipótesis, `a()` termina, entonces a_{n+1} y signal_{n+2} ocurren. Esto es, $\tau_{<n>}$ se puede extender a $\tau_{<n+1>}$ que termina con la transición signal_{n+2}. $\square$

[1] Es útil remarcar aquí que se asume que el *scheduler* subyacente es justo.

Observemos que la propiedad **WAIT-FREE** no se satisface porque la terminación de un proceso depende de la terminación de otro.