Ejercicios de Sincronización de Procesos

Sergio Yovine

31 de agosto de 2016

1. 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. Se pide:

1. Implementar un protocolo de sincronización tal que las acciones a_i se ejecuten globalmente en orden $a_0 \dots a_{N-1}$ (**ORDEN**).
2. Probar que todo proceso i ejecuta a_i y termina (**G-PROG**).
3. Resolver el problema en $\mathcal{O}(1)$ de memoria.

1.1. Solución con semáforos

Una solución es usar 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, dónde los comentarios R, T, C y E corresponden a los estados y $wait_i$ y $signal_{i+1}$ corresponden a las acciones *try* y *rem* del modelo de Lynch.

```
sem sem[N];

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

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 $\tau_0 \to \tau_1 \ldots$ 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 $\texttt{sem[}i\texttt{]}$. 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: $\tau_{k'}(\texttt{sem}[\mathrm{i}]) = \tau_{k'}(\texttt{sem}[\mathrm{i+1}]) = 1$, 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 $\tau_{k''} \xrightarrow{signal_{i+1}} \tau_{k''+1}$. 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 vaío. De hecho, en ningún momento se usa la propiedad que `sem[0]` $= 1$, sino sólo que $\texttt{sem[}i\texttt{]} = 0$ 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 Asumimos 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 $wait_{n+1}$ 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 $wait_{n+1}$.

Entonces, el prefijo $\tau_{<n>}$ se puede extender a una subsecuencia $\tau_{<n>}\tau'$ en cuyo último estado la transición $enter_{n+1}$ puede ocurrir. Esto es así dado que $\texttt{sem[}n+1\texttt{]} = 1$, puesto que ocurrió $signal_{n+1}$ en el prefijo $\tau_{<n>}\tau'$. Por lo tanto,

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

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 $signal_{n+2}$. $\square$

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

1.2. Solución en $\mathcal{O}(1)$

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

```
atomic<int> turno = 0;

void turno(pid i) {
  // R
  // try_i
  // T
  while (turno.compareAndSwap(i, -1) != i) {}; // busy waiting
  // enter_i
  // C
  a();
  // exit_i
  // E
  turno.getAndInc();
  // signal_{i+1}
}
```

También es posible resolver el problema sólo usando registros atómicos *read-write*, es decir, registros cuyas únicas operaciones atómicas son la lectura y la escritura.

```
atomic<int> turno = 0;

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

Se deja como ejercicio probar que estas soluciones son correctas, es decir, satisfacen **ORDEN** y **G-PROG**.