Final de Sistemas Operativos

Sergio Yovine

2 de agosto de 2016

1 Sincronización

Ejercicio 1.A

Considere el modelo de memoria compartida.

1. Dé dos primitivas de sincronización.
2. Dé implementaciones para cada una de ellas. Explique y justifique todo lo que asuma.
3. Explique ventajas y desventajas de cada una.

Ejercicio 1.B

Considere el siguiente problema de sincronización. Hay $N \geq 2$ procesos, numerados de 1 a N. Cada proceso i ejecuta la secuencia $a_i b_i$.

1. Implemente un protocolo de sincronización tal que las acciones b se ejecuten globalmente en orden $b_1 \dots b_n$.
2. Pruebe que todo proceso termina.
3. Puede resolver el problema en $\mathcal{O}(1)$ de memoria? Justifique.

2 Consenso

Sea dado un conjunto de $N \geq 2$ procesos y un conjunto V de valores. Cada proceso dispone de dos funciones, $in()$ y $decide()$. El problema de *consenso* está definido por las siguientes propiedades:

Inicio Todo proceso i empieza con $in(i) \in V$

Acuerdo Para todo $i \neq j$, se cumple que $decide(i) = decide(j)$

Validez Existe i, tal que $in(i) = decide(i)$

Terminación Todo i decide en un número finito de transiciones (*wait-free*)

El *número de consenso* de un objeto atómico es la cantidad de procesos para la cual un objeto atómico puede resolver el problema de consenso.

Ejercicio 2.A

Sea Stack una pila con dos operaciones atómicas push(x), que inserta x en la pila, y pop(), que devuelve el último elemento insertado en la pila. Pruebe que Stack puede resolver consenso para $N = 2$, es decir, su número de consenso es al menos 2. Ayuda: Tome $V = \{0, 1\}$ e implemente las funciones $in()$ y $decide()$ usando un objeto atómico de tipo Stack.

Ejercicio 2.B

Considere el problema de elección de líder en un modelo de memoria compartida con $N \geq 2$ procesos, numerados de 1 a N. Inicialmente, todos los procesos quieren ser líderes, in(i) = i.

1. Explique qué condiciones tiene que cumplir N para que se pueda resolver el problema de consenso definido arriba con un objeto atómico de tipo *test-and-set*.

2. En caso de ser posible, asumiendo que se cumplen las condiciones apropiadas, implemente un protocolo que resuelva el problema anterior. Argumente por qué la solución propuesta no funciona si no se cumplen las condiciones.

3. Explique usando argumentos teóricos si es posible resolver el problema sin condiciones sobre N. En caso afirmativo, implemente un protocolo.

Solución 1.A

Orden N

Una solución es usar un array `sem` de N semáforos. Para simplificar, asumimos que los arrays están indexados de 1 a N. Inicialmente, `sem`$[i]$ es 1 si $i = 1$ y 0 para todo $i \neq 1$.

```
1  sem sem[N];
2
3  void proceso(pid i) {
4    a(i);
5    // T
6    sem[i].wait(); // wait_i
7    // C
8    b(i);
9    // E
10   if (i < N) sem[i+1].signal(); // signal_{i+1}
11 }
```

Tenemos que probar que: 1) garantiza el orden pedido (**ORDEN**) y que todo proceso termina (**G-PROG**).

ORDEN Probamos que dos procesos i e $i+1$ no se pueden solapar en la ejecución de b. Procedemos por el absurdo. Supongamos que existe una ejecución en la que i e $i+1$ están en C al mismo tiempo. Entonces, existe un estado previo en el cual `sem`$[i]$ = `sem`$[i+1] = 1$. Dado que en el estado inicial `sem`$[k]$ es 1 si y sólo si $k = 1$, tiene que haber ocurrido $signal_{i+1}$. Por lo tanto, b_i tiene que haber terminado, lo que contradice la hipótesis. La prueba se completa por inducción: para todo i, si b_i ocurre, entonces ocurrieron b_j para todo $j < i$. Es trivial para $i = 1$. Supongamos que vale para todo $i \leq n$. Si ocurre b_{n+1} entonces `sem`$[n+1] = 1$ en algún estado previo. Por lo tanto, ocurrió $signal_{n+1}$ con anterioridad. Dado que b_n precede a $signal_{n+1}$ y por hipótesis inductiva b_i precede a b_n para todo $i < n$, concluimos que b_i precede a b_{n+1} para todo $i < n+1$.

G-PROG Asumimos que a y b terminan para todo i. La prueba es por inducción. Es trivial para $i = 1$ dado que inicialmente `sem`$[1] = 1$. Supongamos que vale para todo $i \leq n$. Por hipótesis inductiva n termina, entonces ocurre $signal_{n+1}$. Si $n+1$ está en T entonces en algún momento en el futuro está en C. Dado que b_{n+1} termina, concluimos que $n+1$ termina.

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

Orden 1

Una manera de hacerlo en $\mathcal{O}(1)$ es usando un registro RW atómico `turno`, inicializado a 1. Estas soluciones tienen *busy waiting*.

```
1  atomic<int> turno = 1;
2
3  void proceso(pid i) {
4    a(i);
5    // T
6    while (turno < i) {}; // busy waiting
7    // C
8    b(i);
9    // E
10   if (i < N) { tmp = i+1; turno = tmp; } // signal_{i+1}
11 }
```

Si disponemos de registros con `getAndInc()` podemos hacer:

```
10   if (i < N) turno.getAndInc() // signal_{i+1}
```

Solución 2.A

La solución consiste en inicializar la pila con dos valores `WIN` y `LOOSE`, y un array `proposed` de dimensión 2 en los que se guardan los valores propuestos por cada uno en `in()`. El array `proposed` se inicializa en -1. Si un proceso obtiene `WIN` decide lo que propuso él mismo; si no, decide lo que propuso el otro.

```
1  #define WIN 1
2  #define LOOSE 0
3  atomic<Stack> stack;
4  int[2] proposed = {-1, -1};
5
6  void init() {
7   stack.push(WIN);
8   stack.push(LOOSE);
9  }
10
11 void in(value v) { proposed[pid()] = v; }
12
13 int decide() {
14   if (WIN == stack.pop()) return proposed[pid()];
15   else return proposed[(pid() + 1) % 2];
16 }
```

Inicio Trivial.

Acuerdo Si los dos proponen el mismo valor es trivial. Supongamos que proponen valores diferentes. Sin pérdida de generalidad, supongamos que el proceso i propone i, $i = 0, 1$ y que el proceso 0 obtiene `WIN`. Dado que in precede $decide$, tenemos que $decide_0$ = `proposed`[0] = 0. Por consiguiente, el proceso 1 obtiene `LOOSE` y decide el valor de `proposed`[0]. En este caso, 1 desempiló después que 0, esto es, pop_0 precede a pop_1. Como in_0 precede a pop_0, concluimos que in_0 precede a pop_1, con lo cual `proposed`[0] = 0.

Validez Hay dos casos.

- El valor obtenido de la pila es `WIN`. Entonces $decide_i = proposed_i = in_i$ dado que in_i precede a $decide_i$.
- El valor obtenido de la pila es `LOOSE`. Entonces, $decide_i = proposed_j$, $i \neq j$. En este caso, j desempiló antes que i. Como in_j precede a $decide_j$, concluimos que in_j precede a $decide_i$, con lo cual $decide_i = in_j$.

Terminación Se garantiza porque la pila es wait-free e inicialmente se empilan dos valores.

Solución 2.B

Dado que TAS tiene número de consenso 2, N tiene que ser igual a 2. El problema de la elección de líder es un problema de consenso que se puede resolver de manera *wait-free* para $N = 2$ con TAS como sigue:

```
1  atomic<bool> b = false;
2  int[2] proposed = {-1, -1};
3
4  void in() { proposed[pid()] = pid(); }
5
6  int decide() {
7   if !b.testAndSet() return proposed[pid()];
8   else return proposed[(pid() + 1) % 2];
9  }
```

La prueba es similar a la del ejercicio 2.A. Esta solución no funciona para $N > 2$ porque si TAS devuelve *true*, no se sabe qué proceso fue el que ganó, es decir, el que puso el registro TAS en *true*. Observemos que usar una variable compartida para guardar el *pid* del proceso ganador tampoco funciona porque no se puede garantizar que el ganador la escriba *antes* de que los demás la lean.

El problema de elección de líder para un N dado se puede resolver de manera *wait-free* con una primitiva cuyo número de consenso sea al menos N. Para cualquier N, se puede resover con `compareAndSwap` porque tiene número de consenso infinito. La solución es la siguiente:

```
1  atomic<int> leader = -1;
2  int[N] proposed = {-1, -1};
3
4  void in() { proposed[pid()] = pid(); }
5
6  int decide() {
7    if pid() == leader.compareAndSwap(-1, pid())
8      return proposed[pid()];
9    else return proposed[leader];
10 }
```