Semáforos

Paulo Sérgio Almeida

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho

2007/2008

Semáforos

- Semáforo é uma primitiva clássica de controlo de concorrência, inventada por Dijkstra.
- São tipicamente implementados pelo sistema operativo.
- Permitem resolver problemas de concorrência de vários tipos (e.g. exclusão mútua ou ordem de execução).
- São de relativamente baixo nível, envolvendo alguma “arte” na construção de soluções.
- Actualmente são preteridos pelos monitores como primitivas genéricas de controlo de concorrência em memória partilhada.

Definição de semáforos

- Um semáforo é um tipo de dados com dois componentes e duas operações atómicas.
- Componentes:
  - v, valor do semáforo, inteiro não negativo;
  - l, conjunto de processos bloqueados no semáforo;
  - um semáforo é iniciado com um valor v inteiro, e l vazio;
  - os componentes são manipulados apenas pelas operações;
- Operações:
  - são disponibilizadas duas operações wait e signal;
  - estas operações são vistas como atómicas;
  - wait: tenta decrementar o valor do semáforo, bloqueando o processo no semáforo caso o valor seja 0;
  - signal: se houver processos bloqueados no semáforo, liberta um arbitrariamente; senão, incrementa o valor do semáforo.

Operações sobre semáforos

- wait(S), sendo *p* o processo actual que executa:

```
if S.v > 0:
  S.v = S.v - 1
else:
  S.l.add(p)
  suspend()
```

- signal(S):

```
if S.l = {}:
  S.v = S.v + 1
else:
  q = S.l.pop()
  ready(q)
```

- Onde:
  - `add(e)`: adiciona elemento *e* ao conjunto
  - `pop()`: remove e devolve um elemento do conjunto
  - `suspend()`: bloqueia o processo corrente e escalona outro processo
  - `ready(p)`: torna o processo *p* pronto a correr

Notas sobre semáforos

- O valor do semáforo nunca é negativo.
- A operação signal nunca bloqueia.
- A operação signal liberta um processo bloqueado no semáforo.
- A operação wait pode bloquear o processo corrente.
- A implementação das operações é tal que estas têm um comportamento equivalente a serem atómicas:
  - e.g. não é possível que, com o semáforo a 1, e dois processos a fazer wait, ambos decidam decrementar o valor do semáforo e retornar, ficando este a -1;
  - neste caso, o wait retornaria num processo, o valor do semáforo ficaria 0, e o outro processo bloquearia no wait;
- As operações wait e signal, foram originalmente chamadas de P e V; são também usadas down e up ou acquire e release

Mutexes: semáforos binários

- Um caso particular importante: semáforos binários.
- O Semáforo binário pode apenas tomar os valores 0 e 1.
- As operações são as mesmas excepto signal(S), que tem um comportamento indefinido, caso o valor do semáforo seja 1:

```
if S.v = 1:
  comportamento indefinido
elif S.l = {}:
  S.v = 1
else:
  q = S.l.pop()
  ready(q)
```

- Este semáforo é também chamado de mutex, pois é apropriado para obter exclusão mútua.

Exclusão mútua com semáforos

- É trivial obter exclusão mútua com semáforos.
- Basta um semáforo binário, iniciado a 1.
- Cada processo que quer entrar numa secção crítica faz:

```
wait(S);
//
// seccao critica
//
signal(S);
```

- No primeiro processo o wait coloca o semáforo a 0 e retorna.
- Se outros processos fizerem wait, este bloqueia os processsos no semáforo.
- Quando um processo faz signal, um dos processos é desbloqueado; o valor do semáforo mantém-se a 0.
- O último processo a fazer signal coloca o semáforo a 1.

Semáforos em Java

- Classe `Semaphore`, em `java.util.concurrent`.
- O construtor leva como parâmetro o valor do semáforo.
- wait e signal têm os nomes de `acquire()` e `release()`.

```
Semaphore s = new Semaphore(1);
...
s.acquire();
// seccao critica
s.release()
```

- os semáforos de Java contêm generalizações do conceito:
  - operações `acquire(n)` e `release(n)`, para tentar decrementar / incrementar o valor do semáforo *n* vezes atomicamente;
  - operações `tryAcquire()` e `tryAcquire(n)`, que retornam imediatamente se a operação correspondente bloqueasse, devolvendo um booleano.

Invariantes sobre semáforos

Sendo:

- k o valor inivial do semáforo S,
- $\#signal(S)$ e $\#wait(S)$ o número de operações signal e wait concluídas sobre S,

temos:

$$\begin{aligned} S.v &\geq 0, \\ S.v &= k + \#signal(S) - \#wait(S). \end{aligned}$$

- O primeiro é trivial pela definição de semáforos.
- O segundo porque:
  - se alguma operação termina mudando $S.v$, preserva o invariante,
  - se signal liberta um processo, também termina um wait, $S.v$ fica na mesma, mas também o lado direito da equação.

Secção crítica com semáforos

- Cada processo faz:

```
wait(S);
// seccao critica
signal(S);
```

- Sendo $\#SC$ o número de processos na secção crítica:

$$\#SC = \#wait(S) - \#signal(S).$$

- Pelos segundo invariante sobre semáforos:

$$\#SC + S.v = 1.$$

- Este algoritmo garante exclusão mútua:

$$\textit{Como } S.v \geq 0\textit{, temos que } \#SC \leq 1.$$

- Este algoritmo garante ausência de deadlock:
  - em deadlock estariam os processos no wait, $S.v = 0$ e $\#SC = 0$;
  - isto contradiz $\#SC + S.v = 1$.

Secção crítica com semáforos: starvation

- E quanto à ausência de starvation?
- Para 2 processos, p e q:
  - um processo p em starvation estaria bloqueado no wait;
  - teriamos $S.v = 0$ e $p \in S.l$;
  - como $\#SC = 1 - S.v = 1$, então q estaria na secção crítica;
  - com apenas p e q teriamos $S.l = \{p\}$;
  - o signal de q libertaria p, que entraria na secção crítica;
  - portanto, para 2 processos não há starvation.
- E para N processos?

Secção crítica com semáforos: starvation

- Para $N > 2$ processos, nomeadamente p, q e r.
- Poderiamos ter uma sequência de eventos como:
  - p e q bloqueados e r na secção crítica;
  - signal de r liberta q;
  - r bloqueia novamente no wait;
  - signal de q liberta r;
  - q bloqueia novamente no wait;
  - voltamos à situação inicial;
- Esta sequência poderia repetir-se indefenidamente.
- Conclusão: p poderia esperar indefinidamente.
- Para $N > 2$ processos pode haver starvation.
- Problema: não há garantia de qual dos processos bloqueados no semáforo é libertado.

Algoritmo de Udding sem starvation

- Semáforos G1(1), G2(0); inteiros nG1=0, nG2=0;

```
wait(G1);
nG1 = nG1 + 1;
signal(G1);
wait(G1);                  // primeira gate
nG1 = nG1 - 1;
nG2 = nG2 + 1;
if (nG1 > 0)
  signal(G1);
else
  signal(G2);
wait(G2);                  // segunda gate
nG2 = nG2 - 1;
//
// seccao critica
//
if (nG2 > 0)
  signal(G2);
else
  signal(G1);
```

- Será que garante ausência de starvation?

Algoritmo de Udding sem starvation – versão correcta

- Semáforos G1(1), G2(0), M(1); inteiros nG1=0, nG2=0;

```
wait(G1);
nG1 = nG1 + 1;
signal(G1);
wait(M);
wait(G1);                  // primeira gate
nG1 = nG1 - 1;
nG2 = nG2 + 1;
if (nG1 > 0)
  signal(G1);
else
  signal(G2);
signal(M);
wait(G2);                  // segunda gate
nG2 = nG2 - 1;
//
// seccao critica
//
if (nG2 > 0)
  signal(G2);
else
  signal(G1);
```

Semáforos fortes

- Garantir ausência de starvation pode ser complexo.
- Variantes de semáforos podem dar mais garantias.
- Semáforos fortes têm uma fila de processos bloqueados (em vez de um conjunto); estes são libertados por ordem de chegada.
- Tal permite, por exemplo, evitar starvation na solução simples para secções críticas com *N* processos.
- wait(S), sendo *p* o processo actual que executa:

```
if S.v > 0:
  S.v = S.v - 1
else:
  S.l.append(p)          // acrescenta no fim da fila
  suspend()
```

- signal(S):

```
if S.l = []:
  S.v = S.v + 1
else:
  q = S.l.pop(0)         // remove e devolve o primeiro da fila
  ready(q)
```

Exemplo: produtor-consumidor com bounded-buffer

- Dois semáforos: items e slots.
- Contam os itens no buffer e as posições livres.

Consumidor:

```
while (...) {
  wait(items);
  x = buffer.take();
  signal(slots);
  consume(x);
}
```

Produtor

```
while (...) {
  x = produce():
  wait(slots);
  buffer.put(x);
  signal(items);
}
```

- E a exclusão mútua no acesso ao buffer?

Produtor-consumidor: acesso ao buffer

Buffer como tipo asbtracto de dados com controlo de concorrência:

- Exclusão mútua é garantida internamente pelo buffer.
- Buffer contém mutex interno.
- Operações put e take adquirem e libertam mutex.

```
class Buffer {
  Semaphore mut(1);
  int take() {
    wait(mut);
    x = ...
    signal(mut);
    return x
  }
  put(int x) {
    wait(mut);
    ... x ...
    signal(mut);
  }
}
```

Produtor-consumidor: acesso ao buffer

Acesso directo ao buffer:

- Controlo de concorrência no produtor e consumidor.
- Será possível não ter semáforo para exclusão mútua?
- Variáveis: int buffer[N], itake=0, iput=0;

Consumidor:

```
while (...) {
  wait(items);
  x = buffer[itake];
  itake = (itake + 1) % N;
  signal(slots);
  consume(x);
}
```

Produtor

```
while (...) {
  x = produce():
  wait(slots);
  buffer[iput] = x;
  iput = (iput + 1) % N;
  signal(items);
}
```

- Funciona para 2 processos?
- Funciona para $P > 2$ processos?

Produtor-consumidor: acesso ao buffer

Acesso directo ao buffer:

- Controlo de concorrência no produtor e consumidor.
- Solução genérica com semáforo extra para exclusão mútua.

Consumidor:

```
while (...) {
  wait(items);
  wait(mut);
  x = buffer[itake];
  itake = (itake + 1) % N;
  signal(mut);
  signal(slots);
  consume(x);
}
```

Produtor

```
while (...) {
  x = produce():
  wait(slots);
  wait(mut);
  buffer[iput] = x;
  iput = (iput + 1) % N;
  signal(mut);
  signal(items);
}
```

Produtor-consumidor: acesso ao buffer

Acesso directo ao buffer:

- E se a ordem dos waits estivesse trocada?

Consumidor:

```
while (...) {
  wait(mut);
  wait(items);
  x = buffer[itake];
  itake = (itake + 1) % N;
  signal(mut);
  signal(slots);
  consume(x);
}
```

Produtor

```
while (...) {
  x = produce():
  wait(mut);
  wait(slots);
  buffer[iput] = x;
  iput = (iput + 1) % N;
  signal(mut);
  signal(items);
}
```

- Haverá uma versão correcta que permita mais concorrência?

Produtor-consumidor: acesso ao buffer

Acesso directo ao buffer:

- Controlo de concorrência no produtor e consumidor.
- Dois semáforos para exclusão mútua.
- Um para produtores e outro para consumidores.

Consumidor:

```
while (...) {
  wait(items);
  wait(mutcons);
  x = buffer[itake];
  itake = (itake + 1) % N;
  signal(mutcons);
  signal(slots);
  consume(x);
}
```

Produtor

```
while (...) {
  x = produce():
  wait(slots);
  wait(mutprod);
  buffer[iput] = x;
  iput = (iput + 1) % N;
  signal(mutprod);
  signal(items);
}
```