

Semáforos

Carlos Baquero (Slides: Paulo Sérgio Almeida)

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho

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 atômicas.
- ▶ Componentes:
 - ▶ v , valor do semáforo, inteiro não negativo;
 - ▶ I , conjunto de processos bloqueados no semáforo;
 - ▶ um semáforo é iniciado com um valor v inteiro, e I vazio;
 - ▶ os componentes são manipulados apenas pelas operações;
- ▶ Operações:
 - ▶ são disponibilizadas duas operações **acquire** e **release**;
 - ▶ estas operações são vistas como atômicas;
 - ▶ **acquire**: tenta decrementar o valor do semáforo, bloqueando o processo no semáforo caso o valor seja 0;
 - ▶ **release**: se houver processos bloqueados no semáforo, liberta um arbitrariamente; senão, incrementa o valor do semáforo.

Operações sobre semáforos

- ▶ `acquire(S)`, sendo p o processo actual que executa:

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

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

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 `release(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:

```
acquire(S);  
//  
// seccao critica  
//  
release(S);
```

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

Semáforos em Java

- ▶ Classe Semaphore, em `java.util.concurrent`.
- ▶ O construtor leva como parâmetro o valor do semáforo.

```
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 inicial do semáforo S ,
- ▶ $\#release(S)$ e $\#acquire(S)$ o número de operações release e acquire concluídas sobre S ,

temos:

$$S.v \geq 0,$$

$$S.v = k + \#release(S) - \#acquire(S).$$

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

Secção crítica com semáforos

- ▶ Cada processo faz:

```
acquire(S);  
// seccao critica  
release(S);
```

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

$$\#SC = \#acquire(S) - \#release(S).$$

- ▶ Pelos segundo invariante sobre semáforos:

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

- ▶ Este algoritmo garante exclusão mútua:

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

- ▶ Este algoritmo garante ausência de deadlock:

- ▶ em deadlock estariam os processos no acquire, $S.v = 0$ 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 acquire;
 - ▶ teríamos $S.v = 0$ e $p \in S.l$;
 - ▶ como $\#SC = 1 - S.v = 1$, então q estaria na secção crítica;
 - ▶ com apenas p e q teríamos $S.l = \{p\}$;
 - ▶ o release 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 .
- ▶ Poderíamos ter uma sequência de eventos como:
 - ▶ p e q bloqueados e r na secção crítica;
 - ▶ release de r liberta q ;
 - ▶ r bloqueia novamente no acquire;
 - ▶ release de q liberta r ;
 - ▶ q bloqueia novamente no acquire;
 - ▶ voltamos à situação inicial;
- ▶ Esta sequência poderia repetir-se indefinidamente.
- ▶ 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.

Semáforos fortes (fairness)

- ▶ 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.
- ▶ `acquire(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()
```

- ▶ `release(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 (...) {  
    acquire(items);  
    x = buffer.take();  
    release(slots);  
    consume(x);  
}
```

Produtor

```
while (...) {  
    x = produce();  
    acquire(slots);  
    buffer.put(x);  
    release(items);  
}
```

- ▶ E a exclusão mútua no acesso ao buffer?

Produtor-consumidor: acesso ao buffer

Buffer como tipo abstracto 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() {  
        acquire(mut);  
        x = ...  
        release(mut);  
        return x  
    }  
    put(int x) {  
        acquire(mut);  
        ... x ...  
        release(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 (...) {  
    acquire(items);  
    x = buffer[itake];  
    itake = (itake + 1) % N;  
    release(slots);  
    consume(x);  
}
```

Produtor

```
while (...) {  
    x = produce();  
    acquire(slots);  
    buffer[iput] = x;  
    iput = (iput + 1) % N;  
    release(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 (...) {  
    acquire(items);  
    acquire(mut);  
    x = buffer[itake];  
    itake = (itake + 1) % N;  
    release(mut);  
    release(slots);  
    consume(x);  
}
```

Produtor

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

Produtor-consumidor: acesso ao buffer

Acesso directo ao buffer:

- E se a ordem dos acquire estivesse trocada?

Consumidor:

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

Produtor

```
while (...) {  
    x = produce():  
    acquire(mut);  
    acquire(slots);  
    buffer[iput] = x;  
    iput = (iput + 1) % N;  
    release(mut);  
    release(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 (...) {  
    acquire(items);  
    acquire(mutcons);  
    x = buffer[itake];  
    itake = (itake + 1) % N;  
    release(mutcons);  
    release(slots);  
    consume(x);  
}
```

Produtor

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