Rui Oliveira
Departamento de Informática
Universidade do Minho

Sistemas Distribuídos

Programação Concorrente

(apontamentos originais de Paulo Sérgio Almeida)

2011~2014

Programa Detalhado

- Programação concorrente
  - Sistemas concorrentes
  - Partilha de memória vs troca de mensagens
  - Concorrência em memória partilhada
  - Secções críticas / exclusão mútua
  - Controlo de concorrência por sw e hw
  - Primitivas de CC pelo sistema operativo
  - Ordem de execução / variáveis de condição
  - Monitores

Sistemas concorrentes

- Processos correm concorrentemente num sistema:
  - Com pseudo-concorrência, partilhando tempo de CPU;
  - Com verdadeira concorrência, num multi-processador ou num sistema distribuído;
- Com concorrência os processos:
  - executam ações independentemente;
  - correm com velocidades relativas imprevisíveis; com partilha de tempo, as ações são intercaladas de um modo imprevisível;

Comunicação e sincronização

- Vários processos podem querer cooperar num objectivo comum;
- Duas necessidades se verificam: comunicação e sincronização;
- Comunicação:
  - passagem de informação entre processos;
  - Exemplos:
    - um processo produz itens que vão ser usados pelo outro;
    - um processo **servidor** recebe pedidos de **clientes** e devolve resultados;

Comunicação e sincronização

- Vários processos podem querer cooperar num objectivo comum;
- Duas necessidades se verificam: comunicação e sincronização;
- Sincronização:
  - associado à comunicação mas não necessariamente, processos podem ter que esperar antes de poder prosseguir;
  - Exemplos:
    - esperar até item estar disponível para ser consumido
    - esperar até resposta ser devolvida;
    - esperas ativas indesejáveis;

Processos e Threads

- Um processo começa com uma thread de execução; outras podem ser criadas em seguida;
- As threads dos vários processos correm concorrentemente;
- Cada processo possui o seu espaço de endereçamento privado;
- Todas as threads criadas dentro do mesmo processo:
  - partilham memória e recursos do processo;
  - mantêm o seu stack com variáveis locais privadas

Processos e Threads

- Threads podem ser vistas como processos leves que permitem cooperação eficiente via memória;
- Podem também ser pedidos ao SO segmentos de memória partilhada entre processos;
- Vamos usar frequentemente a palavra processo no sentido lato, que engloba possível partilha de memória;

Partilha de memória vs. troca de mensagens

- Consideramos duas grandes famílias de paradigmas na programação de sistemas concorrentes:
  - Sistemas com memória partilhada + primitivas de sincronização;
  - Sistemas com memória independente + troca de mensagens;

Troca de mensagens

- Diferentes primitivas:
  - Primitivas básicas tipo send/receive;
  - Cliente-servidor;
  - Broadcast/multicast;
    - Abstrações de comunicação de mais alto nível; ex. comunicação em grupo, filas de mensagens.
- Comunicação síncrona ou assíncrona;
- Comunicação orientada (ou não) à conexão;

Concorrência em memória partilhada

- As ações dos processos são intercaladas de um modo imprevisível;
- Acesso concorrente a dados partilhados pode levar a inconsistência;
- São necessárias primitivas de controlo de concorrência. Primitivas clássicas:
  - locks
  - semáforos
  - variáveis de condição
  - monitores

Corridas (race conditions)

Corrida (race condition): Quando processos manipulam concorrentemente dados partilhados e o resultado depende da ordem dos acessos.

- Exemplo:
  - Vários processos acedem a um contador partilhado e cada processo pretende incrementar o contador:

```
contador = contador + 1
```

- Suponha esta instrução traduzida para instruções máquina:

```
MOV contador, R0
ADD R0, 1
MOV R0, contador
```

- Dado que a execução dos processos pode ser intercalada de diferentes modos, o que pode acontecer ao contador?

Secções críticas

- Um segmento de código que acede a recursos partilhados é uma **secção crítica**;
- As secções críticas têm que ser submetidas a controlo de concorrência, caso contrário poderão surgir **corridas**;
- Uma secção crítica deve ser rodeada por código de entrada (que obtém permissão para entrar) e de saída;

```
...
codigo de entrada
seccao critica
codigo de saida
...
```

Problemas das secções críticas

- Uma solução para o problema das secções críticas deve garantir:
  - **exclusão mútua**: se um processo está a executar na sua secção crítica, mais nenhum o pode fazer;
  - **ausência de deadlock**: se vários processos estão a tentar entrar na secção crítica, um deles deve inevitavelmente conseguir;
  - **ausência de starvation**: se um processo tenta entrar na secção crítica, inevitavelmente conseguirá.

Problemas das secções críticas

- Outra formulação para o problema
  - **exclusão mútua**: se um processo está a executar na sua secção crítica, mais nenhum o pode fazer;
  - **progresso**: se nenhum processo estiver a executar a secção crítica apenas os processos que executam o código de sincronização podem participar na decisão de quem entra; esta decisão não poderá ser adiada indefinidamente;
  - **espera limitada**: existe um limite para o número de vezes que outros processos podem entrar passando à frente de um processo que já executou o código de sincronização.

Soluções por software para secções críticas

- Será possível encontrar uma solução para o problema sem ajuda de hardware?
- Admitamos que podemos fazer esperas activas:

```
...
While(...)
  ;
// Secção crítica
...
```

- Poderemos obter o código de entrada e saída da secção crítica manipulando variáveis partilhadas de um modo cuidadoso?

Soluções por software para secções críticas

- Na solução, uma variável é partilhada:

```
int autorizacao = 1;
```

- O processo i executa:

```
while (autorizacao == 0)
  ;
  autorizacao = 0
//
// secção crítica
//
  autorizacao = 1;
```

- Não é uma solução, pois não?

Solução de Peterson para as secções críticas

- Exemplo clássico de solução por software;
- Restrito ao caso de 2 processos, P0 e P1;
- Na solução, duas variáveis são partilhadas:

```
int vez = 0;
boolean entrar[2] = [0,0];
```

- O processo i executa:

```
entrar[i] = true;
vez = 1 - i;
while (entrar[1-i] && vez == 1-i)
  ;
//
// secção crítica
//
entrar[i] = false;
```

Solução de Peterson para as secções críticas

- Problemas
  - Restrita a 2 processos;
  - Envolve espera activa;
  - Pode não funcionar em arquiteturas de hardware modernas devido a cache e coerência de caches;
- A solução pode ser estendida a n processos*

* Some Myths About Famous Mutual Exclusion Algorithms, K. Alagarsamy, 2003

Soluções por hardware para secções críticas

- Um padrão geral de solução pode ser obtido com locks:

```
adquirir lock
secção crítica
libertar lock
```

- O problema **pode** ser passado para a implementação dos **locks**, com ajuda de hardware
- Diferentes possibilidades:
  - inibir interrupções
  - instruções **atómicas**: test-and-set,swap,...

Instrução atómica Test-and-set

- Instrução que atomicamente:
  - coloca o valor de uma flag a verdadeiro e
  - devolve o valor que a flag tinha previamente

- Faz atomicamente o equivalente a:

```
boolean TestAndSet(boolean *pt) {
   boolean tmp = *pt;
   *pt = true;
   return tmp;
}
```

Exclusão mútua com test-and-set

- Usando test-and-set é fácil implementar exclusão mútua;
- Flag `lock` começa a `false`;
- Cada processo faz:

```
while (TestAndSet(&lock)== true)
       ;
//
// seccao critica
//
lock = false;
```

- A solução satisfaz todos os requisitos?

Exclusão mútua espera limitada via test-and-set

- Solução para N processos, usa como itens partilhados, iniciados a `false`:

```
boolean espera[N];
boolean lock = false;
```

- O processo i executa:

```
espera[i] = true;
while (TestAndSet(&lock) && espera[i]
   ;
espera[i] = false;
// seccao critica
j = (i + 1) % N;
while (j != i && !espera[j])
   j = (j + 1) % N;
if (j == i)
   lock = false;
else
   espera[j] = false;
```

Esperas ativas e o sistema operativo

- As soluções anteriores ainda envolvem esperas ativas;
- Estas podem prolongar-se enquanto outros processos executam as suas secções críticas – inaceitável;
- Solução: o sistema operativo disponibiliza as primitivas (ex. adquirir e libertar lock) para rodear as secções críticas;

```
acquire(lock);
//
// secção crítica
//
release(lock);
```

Implementação de primitivas pelo SO

- As primitivas de controlo de concorrência disponibilizadas pelo sistema operativo **eliminam as esperas ativas**;
- O kernel pode fazer um processo P passar ao estado bloqueado, não sendo escalonado e não consumindo tempo de processador;
- Mais tarde um processo Q invoca o libertar lock que trata de mudar o estado do processo P para pronto;
- O processo P pode então entrar na secção crítica da próxima vez que for escalonado.

Exclusão mútua versus ordem de execução

- Na resolução de problemas de controlo de concorrência é útil distinguir duas situações:
  - **exclusão mútua**: quando vários processos concorrem no acesso a recursos partilhados:
    - caso particular muito comum; processo apenas é impedido de prosseguir temporariamente;
  - **ordem de execução**: quando existem padrões de cooperação e dependência entre acções de processos:
    - um processo pode não poder prosseguir até uma dada ação de outro processo;
    - processos bloqueiam-se voluntariamente e são libertados explicitamente por outros.

Exemplo de exclusão mútua: jantar dos filósofos

- Cinco filósofos alternam entre pensar e comer, sentados à volta de uma mesa:
  - existem 5 pratos na mesa;
  - existem 5 garfos na mesa, um entre cada dois filósofos;
  - um filósofo necessita de dois garfos para poder comer;
  - um filósofo só pode pegar/pousar um garfo de cada vez.
- Este é um problema de exclusão mútua envolvendo vários recursos, os garfos.
- Uma solução tem que garantir:
  - segurança: cada filósofo só pode comer tendo dois garfos;
  - ausência de deadlock;
  - ausência de starvation;

Ex. de ordem de execução: produtor-consumidor

- Existem dois tipos de processos:
  - produtor: produz itens de dados;
  - consumidor: consome itens produzidos.
- Itens são produzidos e consumidos para um buffer partilhado, de tamanho limitado.
- Uma solução tem que garantir que:
  - se o buffer está vazio, um consumidor não pode prosseguir, tendo que ficar bloqueado;
  - se o buffer está cheio, um produtor não pode prosseguir, tendo que ficar bloqueado.
- Estes problemas também podem envolver exclusão mútua (manipulação do buffer).

Variáveis de Condição

- As VC são uma primitiva de sincronização disponibilizada pelo SO que permite a um processo **bloquear-se voluntariamente**.
- Os processos testam um predicado sobre variáveis de estado e decidem se bloqueiam. A uma VC é **sempre associado um lock**.
- As VC não têm um valor que se leia ou escreva; o termo “variável” vem do aspecto sintáctico da declaração. São também conhecidas como “condition queues”.

Variáveis de Condição

- Associamos às VC nomes sugestivos das condições pelas quais o processo se decide bloquear, ex.:
  - cheio/vazio, garfo_ocupado, escritor_ativo
- Utilizamos normalmente primitivas **wait** e **signal** para manipular as VC:

```
Lock lock;
Condition cheio;

acquire(lock);
while (lugares == 0)
   wait(cheio, lock);
...
...
signal(cheio)
```

Associação de um lock à VC

Variáveis de Condição

- A cada VC é associada uma fila f de processos que estão bloqueados na VC.
- Sendo o processo p a invocar o wait:

```
wait(cond, lock){
  cond.f.append(p);
  release(lock);
  suspend();
  acquire(lock);
```

- A primitiva signal seria:

```
signal(cond){
  if cond.f != []
     q = cond.f.pop();
     ready(q);
```

Ex.: bounded buffer bloqueante

```
Lock lock;
Condition vazio, cheio;
int a[N], nitems, ...;

int tirar(){
  acquire(lock);
  while (nitems == 0)
     wait(vazio, lock);
  x = ...
  nitems--;
  signal(cheio);
  release(lock);
  return (x);
}

int por(int x) {
  acquire(lock);
  while (nitems == N)
     wait(cheio, lock);
  ...
  nitems++;
  signal(vazio);
  release(lock);
}
```

Monitores

- O Monitor é uma primitiva estruturada de controlo de concorrência, de maior nível de abstração.
- Oferece um tipo abstrato de dados com controlo de **concorrência implícito em todas as operações** com exclusão mútua.
- **Associa automaticamente** os dados e operações ao código de controlo de concorrência.
- Disponibiliza **variáveis de condição** implicita e/ou explicitamente.

Ex.: bounded buffer bloqueante com monitor

```
Monitor Buffer {
    Condition vazio, cheio;
    int a[N], nitems, ...;

    int tirar(){
        while (nitems == 0)
            wait(vazio);
        x = ...
        nitems--;
        signal(cheio);
        return (x);
    }

    int por(int x) {
        while (nitems == N)
            wait(cheio);
        ...
        nitems++;
        signal(vazio);
    }
}
```

Exclusão mútua implícita

Características de implementação de Monitores

- O acesso aos métodos está sujeito a **starvation**
- Os locks são "re-entrantes" (**ReentrantLock**)
- Em monitores modernos (ex.: Java, Pthreads) quando é executado signal:
  - o processo que invoca signal continua a sua execução
  - depois pode executar o processo acordado **ou qualquer outro processo**
  - os testes de predicados do estado deverão por isso utilizar **while** e não apenas if

Correção de programas concorrentes

- Existem dois tipos de propriedades:
  - **Segurança**: determinada propriedade (invariante de estado) é sempre verificada:
    - ex: não estão dois processos na secção crítica;
    - ex: um filósofo só pode comer com dois garfos;
    - a correcção corresponde a certos estados nunca serem atingíveis.
  - **Animação**: determinada propriedade será inevitavelmente verificada:
    - ex: se alguém quer entrar na secção crítica acabará por fazê-lo;
    - ex: se um filósofo quer comer, acabará por comer;
    - a animação diz respeito a certos estados acabarem por ser atingidos.

Algumas propriedades de animação

- Casos particulares importantes de propriedades de animação:
  - **Ausência de deadlock**: nunca é atingido um estado do qual não haja saída (não possa ser feito progresso para um estado desejável);
  - **Ausência de livelock**: nunca é atingido um de vários possíveis estados dos quais não haja saída (para um estado desejável).
  - **Ausência de starvation**: quando um dado processo tenta continuamente efetuar uma ação acaba por o conseguir.
    Ex: um processo só pode ser ultrapassado numa espera um número limitado de vezes.