

Introdução à Programação Concorrente

Paulo Sérgio Almeida

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho



- 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 **acções** independentemente;
 - em modelos simplificados, acções são consideradas **atómicas**
 - correm com velocidades relativas imprevisíveis;
 - com partilha de tempo, acções são intercaladas imprevisivelmente;
 - com paralelismo verdadeiro, acções decorrem simultaneamente;
- Exemplos de acções:
 - leitura de posição de memória
 - escrita em posição de memória
 - envio de mensagem
 - recepção de mensagem



- Vários processos podem querer cooperar num objectivo comum;
- Duas necessidades se verificam: comunicação e sincronização;
- Comunicação:
 - passagem de informação entre processos;
 - processo produz itens que vão ser usados por outro;
 - processo **servidor** recebe pedidos de **clientes** e devolve resultados;
- Sincronização:
 - processos podem ter que esperar antes de poder prosseguir;
 - normalmente associada à comunicação;
 - esperar até item estar disponível para o consumir;
 - esperar até resposta ser devolvida;
 - esperar que todos tenham terminado respectivas tarefas;
 - esperar que todos os componentes estejam iniciados;
 - **esperas activas indesejáveis**



- 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 (variáveis globais e heap) e recursos globais;
 - mantêm o seu stack com variáveis locais privadas;
- 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;



- Duas grandes famílias de paradigmas existem:
 - Com memória partilhada + primitivas de sincronização;
 - Com troca de mensagens;



- Diferentes primitivas:
 - Primitivas básicas tipo send/receive;
 - Cliente-servidor;
 - Broadcast/multicast;
 - Abstracções de mais alto nível; e.g., comunicação em grupo.
- Comunicação síncrona ou assíncrona;
- Comunicação orientada à conexão;
- Formalismos; e.g. CSP – Communicating Sequential Processes

(Deixaremos concorrência baseada em mensagens para mais tarde)



- As acções dos processos intercaladas de modo imprevisível;
- Acesso concorrente a dados partilhados gera inconsistência;
- É necessário primitivas de **controlo de concorrência**;
- Primitivas clássicas:
 - semáforos
 - monitores
- Monitores particularmente importantes – onde são baseadas primitivas de muitas linguagens modernas;
- Monitores levam a tipos abstractos de dados concorrentes



- Intercalamento e “memória partilhada” é um modelo simplificado, mas suficiente para ilustrar muitos problemas;
- Na realidade, temos multicores / multiprocessadores;
- Temos hierarquias de memória: registos, caches, RAM;
- Temos optimizações dos compiladores;
- Variável partilhada pode estar temporariamente em registo;
- “Memória partilhada” pode não reflectir escritas;
- É necessário primitivas de **controlo de concorrência**:
 - não só para sincronização;
 - mas para dar visibilidade das escritas às outras threads;
- Modelo de coerência de memória especifica visibilidade;
- Estudaremos brevemente um modelo de memória mais realista;



Race condition – corridas

Quando processos manipulam concorrentemente estrutura de dados partilhada, e o resultado depende da ordem dos acessos

Exemplo:

- Vários processos acedem a um contador partilhado;
- Cada processo tenta incrementar o contador com:

```
contador = contador + 1
```

- Esta instrução é traduzida para instruções máquina que usam registos, e.g.

```
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?



- 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 temos race conditions;
- As secções críticas são por recurso: podemos ter concorrência envolvendo recursos diferentes;
- Uma secção crítica deve ser rodeada por código de **entrada** (que pede permissão para entrar) e **saída**;

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



Cada processo quer aceder um número arbitrário de vezes em momentos arbitrários a secções críticas sobre um dado recurso.

Uma solução para o problema das secções críticas deve garantir:

- exclusão mútua** se um processo está a executar uma secção crítica, mais nenhum o pode estar;
- ausência de deadlock** se vários processos estão a tentar entrar numa secção crítica, um deles deve conseguir;
- ausência de starvation** se um processo tenta entrar numa secção crítica, inevitavelmente vai entrar.



Outra formulação para o problema:

exclusão mútua se um processo está a executar uma secção crítica, mais nenhum o pode estar;

progresso se nenhum processo estiver numa secção crítica e alguns processos quiserem entrar, apenas os processos que executam o código de sincronização podem participar na decisão de quem entra; esta decisão não pode 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 pediu entrada.



- Será possível arranjar solução para o problema sem ajuda de hardware?
- Suponhamos modelo simplificado com intercalamento de acções atómicas (leituras e escritas) e visibilidade imediata;
- Suponhamos que podemos fazer esperas activas:

```
...  
while(...)  
    ...  
    // seccao critica  
    ...
```

- Poderemos obter o código de entrada e saída da secção crítica manipulando variáveis partilhadas de um modo cuidadoso?
- Pensemos no caso particular de 2 processos, P_0 e P_1 ;



Solução de Peterson para as secções críticas

- Exemplo clássico de solução por software;
- Restrito ao caso de 2 processos, P_0 e P_1 ;
- Na solução, dois itens são partilhados:

```
int vez;  
boolean entrar[2];
```

- Processo i executa:

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



- Exemplo clássico de solução por software;
- Restrita a 2 processos;
- Envolve espera activa;
- Assume implicitamente exclusão mútua de mais baixo nível:
 - atomicidade de acções de leitura/escrita na memória;
 - e.g., dois processos tentarem escrever em vez ao mesmo tempo;



Algoritmo *bakery* de Lamport para exclusão mútua

- Para N processos;
- Nenhuma variável é escrita por dois processos;
- Não assume atomicidade de leituras/escritas: quando leitura e escrita se sobrepõem, assume resultado arbitrário da leitura;

```
boolean choosing[N];  
int number[N];
```

- Processo i executa:

```
choosing[i] = true;  
number[i] = 1 + reduce(max, number); // maximo do array; varias leituras  
choosing[i] = false;  
for j in [0..N-1]:  
    while (choosing[j]) ;  
    while (number[j] != 0 &&  
           (number[j], j) < (number[i], i)) ; // ordem lexicografica  
//  
// seccao critica  
//  
number[i] = 0;
```



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

```
adquirir lock  
seccao critica  
libertar lock
```

- O problema é passado para a implementação dos locks ...
- Com ajuda de hardware temos diferentes possibilidades:
 - inibir interrupções (em uniprocessadores);
 - instruções atômicas: test-and-set, swap, ...



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

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



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

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

- Solução satisfaz todos os requisitos?



- Instrução que atômicamente troca o valor de duas posições de memória;
- Faz atômicamente o equivalente a:

```
void Swap(boolean *a, boolean *b) {  
    boolean tmp = *a;  
    *a = *b;  
    *b = tmp;  
}
```



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

```
key = true;
while (key == true)
    Swap(&key, &lock);
//
// seccao critica
//
lock = false;
```

- Solução satisfaz todos os requisitos?



Exclusão mútua com espera limitada via test-and-set

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

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

- 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;
```



- As soluções anteriores ainda envolvem esperas activas;
- Estas podem demorar tempo considerável, enquanto outros processos executam as suas secções críticas – inaceitável;
- Solução: o sistema operativo disponibilizar as primitivas (e.g. adquirir e libertar lock) para rodear as secções críticas;

```
acquire(&lock);  
//  
// seccao critica  
//  
release(&lock);
```

- Código do utilizador invoca primitivas de controlo de concorrência disponibilizadas, não contendo esperas activas;
- O problema foi passado para a implementação das primitivas ...



- Na implementação das primitivas de concorrência, depois de uma curta espera activa, o kernel pode fazer um processo P passar ao estado bloqueado;
- O processo P , no estado bloqueado, não é escalonado, não consumindo tempo de processador;
- Mais tarde um processo Q invoca o libertar lock;
- O código desta primitiva 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.



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;
 - com apenas um processo, este corre sempre sem problemas;
 - vulgarmente papeis simétricos;
- **ordem de execução**: quando existem padrões de cooperação e dependência entre acções de processos:
 - um processo não pode prosseguir até acção de outro;
 - processos bloqueiam-se voluntariamente;
 - processos são libertados por outros;
 - exige normalmente mais do que um processo;
 - vulgarmente papeis diferentes (e.g., produtor e consumidor);



Exemplo de exclusão mútua: jantar dos filósofos

- Problema clássico de programação concorrente
- Cinco filósofos alternam entre pensar e comer esparguete, sentados à volta de uma mesa redonda:
 - 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 ou pousar um garfo de cada vez.
- Este é um problema de exclusão mútua envolvendo vários recursos, os garfos.
- Uma solução tem que também garantir:
 - segurança: cada filósofo só pode comer tendo dois garfos;
 - ausência de *deadlock*;
 - ausência de *starvation*;
 - eficiência na ausência de contenção.



Exemplo de ordem de execução: produtor-consumidor

- Outro problema clássico.
- 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 (*bounded buffer*).
- 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 (neste caso na manipulação do buffer).



Existem dois tipos de propriedades:

- **Segurança**: determinada propriedade (invariante de estado) é **sempre** verificada:
 - exemplo: não estão dois processos na secção crítica;
 - exemplo: um filósofo só pode estar a comer com dois garfos;
 - a correcção diz respeito a certos estados **nunca** serem atingíveis.
- **Animação**: determinada propriedade será **inevitavelmente** (mais tarde ou mais cedo) verificada:
 - se processo quer entrar na secção crítica, acabará por entrar;
 - se filósofo quer comer, acabará por comer;
 - animação diz respeito a certos estados serem atingidos.



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 efectuar uma dada acção, acaba por o conseguir;
 - exemplo: um processo só pode ser ultrapassado numa espera um número **limitado** de vezes.

