Sincronização de processos

António Pinto
apinto@estg.ipp.pt

Escola Superior de Tecnologia e Gestão

Outubro, 2016

Sumário

Conceitos introdutórios

- *Threads* e processos podem partilhar o mesmo espaço de endereçamento, segmentos de memória adicionais, ficheiros, . . . (processos **cooperativos**)
- Acesso a recurso partilhado por mais do que um processo/*thread* em simultâneo
  - → **Situação de competição** (ou *race condition*)
- Situações de competição resolve-se com coordenação da execução dos processos/*threads* que competem pelo recurso
  - → **Sincronização** da execução dos processos

Conteúdos

Situações de competição

Situação de competição

Situação em que o resultado da execução de vários processos que operam sobre um mesmo recurso é diferente dependendo da ordem de execução dos vários processos.

Situações de competição

Exemplo: Classes *Race* e dados

```
import java.io.*;
class dados {
  private String str;
  public dados(String s) {str=s;}
  public void setDados(String s) {str=s;}
  public String getDados() { return str;}
  }
public class Race implements Runnable {
  public dados d;
  public Race(dados newd) {d=newd;}
  public void run() {
    String myname=Thread.currentThread().getName();
    d.setDados(myname);
    System.out.println("["+myname+"] str:"+d.getDados());
    }
  public static void main(String args[]) {
    dados d=new dados("Inicial");
    (new Thread(new Race(d),"50")).start();
    (new Thread(new Race(d),"86")).start();
    }
}
```

Situações de competição

Execução da classe *Race*

Resultado esperado:

```
aap@~/src $java Race
[50] str:50
[86] str:86
aap@~/src $
```

Situações de competição

Execução da classe *Race*

Resultado esperado:

```
aap@~/src $java Race
[50] str:50
[86] str:86
aap@~/src $
```

Resultado conseguido:

```
aap@~/src $java Race
[50] str:86
[86] str:86
aap@~/src $
```

Incongruência de dados!

Conteúdos

Problema da secção crítica

- Quando existem duas ou mais *threads* que usam ou alteram o valor de um objeto partilhado entre elas, diz-se que está numa **secção crítica** do seu código
- Se as *threads* executarem a secção critica em simultâneo, dá-se a incongruência de dados
- Situações que levem à incongruência devem ser evitadas

Solução para secção crítica

Requisitos

- **Exclusão mútua**: Se já existir uma *thread* a executar a secção crítica do código, mais nenhuma o poderá fazer
- **Progresso**: Se nenhuma *thread* está a executar a secção crítica, mas há várias que o querem fazer, então apenas estas últimas podem participar na seleção da *thread* que irá executar a secção crítica naquele momento
  - Esta seleção não pode ser adiada indefinidamente
- **Espera limitada**: Deve existir um limite máximo para o número de vezes que se impede uma *thread* de aceder à secção crítica.

Problemas possíveis

- Com a implementação de condicionantes à execução decorrentes da existência de secções criticas, podem surgir 2 problemas

Problemas possíveis

- Com a implementação de condicionantes à execução decorrentes da existência de secções criticas, podem surgir 2 problemas

Impasse (*deadlock*)

Bloqueio de um conjunto de processos que aguarda por um evento que apenas pode ser gerado por um processo do conjunto.

Problemas possíveis

- Com a implementação de condicionantes à execução decorrentes da existência de secções criticas, podem surgir 2 problemas

Impasse (*deadlock*)

Bloqueio de um conjunto de processos que aguarda por um evento que apenas pode ser gerado por um processo do conjunto.

Míngua (*starvation*)

Bloqueio de um processo enquanto espera indefinidamente pelo acesso a um recurso que não é libertado.

Soluções possíveis

- Soluções de *software*
  - Programador codifica a sua própria solução.
  - Recorrem a testes constantes de condições (**espera ativa**)

Soluções possíveis

- Soluções de *software*
  - Programador codifica a sua própria solução.
  - Recorrem a testes constantes de condições (**espera ativa**)
- Soluções de *hardware*
  - Utilização direta de instruções especiais do CPU
  - Cria dependência do CPU

Soluções possíveis

- Soluções de *software*
  - Programador codifica a sua própria solução.
  - Recorrem a testes constantes de condições (**espera ativa**)
- Soluções de *hardware*
  - Utilização direta de instruções especiais do CPU
  - Cria dependência do CPU
- Soluções de sistema operativo
  - Sistemas operativos dispõem de funcionalidade avançadas para gerir secções críticas
  - Dependente do sistema operativo
  - Não obrigam à espera ativa

Soluções de *software*

Algoritmo de Peterson[1]

```
//Variaveis partilhadas por 2 processos
bool flag[2] = {false, false};
int turn;
```

```
flag[0] = true;
turn = 1;
while (flag[1] && turn == 1){
//espera ativa
}
// Seccao critica
// ...
// Fim seccao critica
flag[0] = false;
```

```
flag[1] = true;
turn = 0;
while (flag[0] && turn == 0){
//espera ativa
}
// Seccao critica
// ...
// Fim seccao critica
flag[1] = false;
```

Cumpre com os requisitos das secções críticas para dois processos num sistema mono-processador

[1] https://en.wikipedia.org/wiki/Peterson%27s_algorithm

Soluções de Sistema Operativo

Semáforos

- Mecanismo de sincronização que está presente na generalidade dos sistemas operativos
- Funcionalidade resume-se a um número e 2 funções atómicas
  - *wait()*: obtém permissão e avança, senão bloqueia
  - *signal()*: concede autorização
- *wait()* e *signal()* são executadas em exclusão mútua
- Número retrata a quantidade de processos que podem obter permissão em simultâneo

Soluções de Sistema Operativo

Semáforos - Impasse

- Assumindo que Sem1 e Sem2 são dois semáforos com o seu valor iniciado a 1
- Exemplo seguinte pode gerar um impasse (*deadlock*)

P_0	P_1
...	...
wait(Sem1);	wait(Sem2);
wait(Sem2);	wait(Sem1);
...	...
signal(Sem1);	signal(Sem2);
signal(Sem2);	signal(Sem1);

Conteúdos

Almoço dos filósofos

Descrição

- Cinco filósofos estão sentados à mesa
- Cada filósofo têm um prato de arroz
- Filósofo usa 2 paus para comer
- Há 1 pau entre cada 2 pratos
- Filósofo pensa e come em períodos alternados de tempo
- Com fome, tenta pegar nos 2 paus
- Se conseguir, come
- Quando acabar volta a pensar

Almoço dos filósofos

Problema

- Assumindo que se pretende traduzir o comportamento de cada filósofo num programa
- Como implementar o filósofo de forma a que lhe seja permitido comer e pensar sem ficar encravado?

Almoço dos filósofos

Solução 1

```
#define N 5                     //Numero filosofos
void philosopher(int i) {       // Filosofo i
 while (TRUE) {
  think();                      //Pensar
  take_fork(i);                 //Pegar garfo esq.
  take_fork((i+1)%N);           //Pegar garfo dir.
  eat();                        //Comer
  put_fork(i);                  //Pousar garfo esq.
  put_fork((i+1)%N);            //Pousar garfo dir.
  }
 }
```

Almoço dos filósofos

Solução 1

```
#define N 5                     //Numero filosofos
void philosopher(int i) {       // Filosofo i
 while (TRUE) {
  think();                      //Pensar
  take_fork(i);                 //Pegar garfo esq.
  take_fork((i+1)%N);           //Pegar garfo dir.
  eat();                        //Comer
  put_fork(i);                  //Pousar garfo esq.
  put_fork((i+1)%N);            //Pousar garfo dir.
  }
 }
```

- E se todos tentar pegarem o garfo esquerdo em simultâneo?

Almoço dos filósofos

Solução 1

```
#define N 5                     //Numero filosofos
void philosopher(int i) {       // Filosofo i
 while (TRUE) {
  think();                      //Pensar
  take_fork(i);                 //Pegar garfo esq.
  take_fork((i+1)%N);           //Pegar garfo dir.
  eat();                        //Comer
  put_fork(i);                  //Pousar garfo esq.
  put_fork((i+1)%N);            //Pousar garfo dir.
  }
 }
```

- E se todos tentar pegarem o garfo esquerdo em simultâneo? $\rightarrow$ **Impasse**
- Nenhum consegue garfo direito

Almoço dos filósofos

Solução 2

```
#define N 5                     //Numero filosofos
void philosopher(int i) { // Filosofo i
 while (TRUE) {
  EAT=0;
  think();                      //Pensar
  while(EAT=0) {
   take_fork(i);                //Pegar garfo esq.
   if(check_fork((i+1)%N)==AVAILABLE) {
    take_fork((i+1)%N);         //Pegar garfo dir.
    eat();                      //Comer
    put_fork(i);                //Pousar garfo esq.
    put_fork((i+1)%N);          //Pousar garfo dir.
    EAT=1; }
   else {
        put_fork(i);            //Pousar garfo esq.
        sleep(10);              //Esperar
}}}}
```

Almoço dos filósofos

Solução 2

```
#define N 5                     //Numero filosofos
void philosopher(int i) { // Filosofo i
 while (TRUE) {
  EAT=0;
  think();                      //Pensar
  while(EAT=0) {
   take_fork(i);                //Pegar garfo esq.
   if(check_fork((i+1)%N)==AVAILABLE) {
    take_fork((i+1)%N);         //Pegar garfo dir.
    eat();                      //Comer
    put_fork(i);                //Pousar garfo esq.
    put_fork((i+1)%N);          //Pousar garfo dir.
    EAT=1; }
   else {
        put_fork(i);            //Pousar garfo esq.
        sleep(10);              //Esperar
}}}}
```

- E se todos iniciarem ao mesmo tempo?

Almoço dos filósofos

Solução 2

```
#define N 5                     //Numero filosofos
void philosopher(int i) { // Filosofo i
 while (TRUE) {
  EAT=0;
  think();                      //Pensar
  while(EAT=0) {
   take_fork(i);                //Pegar garfo esq.
   if(check_fork((i+1)%N)==AVAILABLE) {
    take_fork((i+1)%N);         //Pegar garfo dir.
    eat();                      //Comer
    put_fork(i);                //Pousar garfo esq.
    put_fork((i+1)%N);          //Pousar garfo dir.
    EAT=1; }
   else {
        put_fork(i);            //Pousar garfo esq.
        sleep(10);              //Esperar
}}}}
```

- E se todos iniciarem ao mesmo tempo? $\rightarrow$ **Míngua**
- Nenhum consegue comer

Almoço dos filósofos

Solução 2

- Solução 2 resolve o problema do impasse, mantendo o problema da míngua
- Problema desta solução é o tempo de espera ser constante
- Problema pode ser minimizado com tempos de espera aleatórios
- Continua a não resolver o problema da míngua, embora o minimize

Almoço dos filósofos

Solução 3

```
#define N 5                         //Numero filosofos
void philosopher(int i) {           // Filosofo i
 while (TRUE) {
  think();                          //Pensar
  wait(sem);
  take_fork(i);                     //Pegar garfo esq.
  take_fork((i+1)%N);               //Pegar garfo dir.
  eat();                            //Comer
  put_fork(i);                      //Pousar garfo esq.
  put_fork((i+1)%N);                //Pousar garfo dir.
  signal(sem);
  }
 }
```

Almoço dos filósofos

Solução 3

```
#define N 5                         //Numero filosofos
void philosopher(int i) {           // Filosofo i
 while (TRUE) {
  think();                          //Pensar
  wait(sem);
  take_fork(i);                     //Pegar garfo esq.
  take_fork((i+1)%N);               //Pegar garfo dir.
  eat();                            //Comer
  put_fork(i);                      //Pousar garfo esq.
  put_fork((i+1)%N);                //Pousar garfo dir.
  signal(sem);
  }
 }
```

- Não tem problemas (impasse, míngua)

Almoço dos filósofos

Solução 3

```
#define N 5                         //Numero filosofos
void philosopher(int i) {           // Filosofo i
 while (TRUE) {
  think();                          //Pensar
  wait(sem);
  take_fork(i);                     //Pegar garfo esq.
  take_fork((i+1)%N);               //Pegar garfo dir.
  eat();                            //Comer
  put_fork(i);                      //Pousar garfo esq.
  put_fork((i+1)%N);                //Pousar garfo dir.
  signal(sem);
  }
 }
```

- Não tem problemas (impasse, míngua)
- Quanto filósofos podem comer de cada vez?

Almoço dos filósofos

Solução 3

```
#define N 5                         //Numero filosofos
void philosopher(int i) {           // Filosofo i
 while (TRUE) {
  think();                          //Pensar
  wait(sem);
  take_fork(i);                     //Pegar garfo esq.
  take_fork((i+1)%N);               //Pegar garfo dir.
  eat();                            //Comer
  put_fork(i);                      //Pousar garfo esq.
  put_fork((i+1)%N);                //Pousar garfo dir.
  signal(sem);
  }
 }
```

- Não tem problemas (impasse, míngua)
- Quanto filósofos podem comer de cada vez? $\rightarrow$ 1, não é uma solução **ótima**

Almoço dos filósofos

Solução 4 (parte 1)

```
#define LEFT (i-1)%N  // Vizinho esquerdo
#define RIGHT (i+1)%N // Vizinho direito
#define THINKING 0    // Pensar
#define HUNGRY 1      // Com fome
#define EATING 2      // Come
int state[N];
semaphore mutex=1;    // Exclusao mutua
semaphore s[N];
void philosopher(int i) {
 while (TRUE) {
  think();
  take_forks(i);
  eat();
  put_forks(i);
  }
}
```

Almoço dos filósofos

Solução 4 (parte 2)

```
void test(int i) {
 if(state[i]==HUNGRY && state[LEFT]!=EATING
    && state[RIGHT]!=EATING) {
  state[i]=EATING;
  signal(s[i]);
  }
}
void take_forks(int i) {
 wait(mutex);    // entrar seccao critica
 state[i]=HUNGRY;
 test(i);
 signal(mutex); // sair seccao critica
 wait(s[i]);     // bloqueia se nao conseguir os garfos
}
void put_forks(int i) {
 wair(mutex);    // entrar seccao critica
 state[i]=THINKING;
 test(LEFT);
 test(RIGHT);
 signal(mutex); // sair seccao critica
}
```

Almoço dos filósofos

Solução 4

- Solução ótima
- Resolve ambos os problemas (impasse e míngua)
- Permite que 2 filósofos comam em simultâneo

Leitores e escritores

Descrição

- Controlo de acesso a dados partilhados
- Dados partilhados pode ser um segmento de memória, um ficheiros, ...
- Acessos podem ser de leitura ou escrita de dados
- Podem coexistir múltiplos leitores em simultâneo
- Acesso de escrita tem de ser em exclusivo

Leitores e escritores

Problema

- Como implementar o leitor e o escritor de forma a manter a eficiência e a evitar incongruência de dados?

Leitores e escritores

Solução - Leitor

```
semaphore mutex=1;      // Garantir exlusao variavel rc
semaphore db=1;         // Garantir acesso BD
int rc=0;               // Num processos em leitura
void reader(void) {
 while (TRUE) {
  wait(mutex);
  rc=rc+1;                  // Mais 1 leitor
  if (rc==1) wait(db);      // Primeiro leitor?
  signal(mutex);
  read_data_base();
  wait(mutex);
  rc=rc-1;
  if (rc==0) signal(db);    // Ultimo leitor?
  signal(mutex);
  use_data_read();
 }
}
```

Leitores e escritores

Solução - Escritor

```
semaphore db=1;          // Garantir acesso BD
void reader(void) {
 while (TRUE) {
   think_up_data();
   wait(db);
   write_data_base();
   signal(db);
   }
}
```

Barbeiro adormecido

Descrição

- Numa barbearia existe uma cadeira de barbeiro e algumas cadeiras de espera
- Na ausência de clientes, o barbeiro senta-se na cadeira de barbeiro e adormece
- Cliente, quando chega, tem de acordar o barbeiro
- Clientes seguintes sentam-se nas cadeiras de espera
- Se não houver cadeiras livres, clientes vão embora

Barbeiro adormecido

Problema

- Como implementar tanto o cliente como o barbeiro sem que surjam situações de competição, míngua ou impasse?

Barbeiro adormecido

```
#define CHAIRS 5
semaphore customers=0;
semaphore barbers=0;
semaphore mutex=1;
int waiting=0;
```

Barbeiro

```
void Barber(void) {
 while (TRUE) {
  down(customers);
  down(mutex);
  waiting=waiting-1;
  up(barbers);
  up(mutex);
  cut_hair();
 }



}
```

Cliente

```
void Customer(void) {
 down(mutex);
 if (waiting<CHAIRS) {
  waiting=waiting+1;
  up(customers);
  up(mutex);
  down(barbers);
  get_haircut();
 }
 else {
  up(mutex);
 }
}
```

Barbeiro adormecido

Solução - Barbeiro

```
#define CHAIRS 5          // Numero cadeiras espera
semaphore customers=0;
semaphore barbers=0;
semaphore mutex=1;
int waiting=0;
```

```
void Barber(void) {
 while (TRUE) {
  down(customers);      // Sem clientes, adormece
  down(mutex);          // Exclusividade acesso
  waiting=waiting-1;    // Decrementa cliente atual
  up(barbers);          // Inicio corte cabelo
  up(mutex);            // Liberta acesso exclusivo
  cut_hair();
 }
}
```

Barbeiro adormecido

Solução - Cliente

```
void Customer(void) {
 down(mutex);                    // Exclusividade acesso
 if (waiting<CHAIRS) {
  waiting=waiting+1;
  up(customers);                 // Acorda barbeiro (se necessario)
  up(mutex);                     // Liberta acesso exclusivo
  down(barbers);                 // Esperar por barbeiro
  get_haircut();
 }
 else {
  up(mutex);                    // Liberta acesso exclusivo
 }
}
```

Bibliografia

- Baseado na bibliografia da unidade curricular.