Sistemas Operativos I

Sincronização de Processos

Luis Lino Ferreira / Maria João Viamonte

Fevereiro de 2006

Sumário

- Problemas inerentes à gestão de recursos
- Inconsistência de dados versus sincronização de processos
- O problema das secções críticas
- Hardware de sincronização
- Semáforos
- Problemas clássicos de sincronização
- Secções críticas
- Sinais

Problemas

- *Starvation*:
  - Em consequência da política de escalonamento da UCP, um recurso passa alternadamente dum processo P1 para um outro processo P2, deixando um terceiro processo P3 indefinidamente bloqueado sem acesso ao recurso
- *Deadlock*:
  - Quando 2 processos se bloqueiam mutuamente
    - Exemplo: o processo P1 acede ao recurso R1, e o processo P2 acede ao recurso R2; a uma dada altura P1 necessita de R2 e P2 de R1
- Inconsistência/Corrupção de dados:
  - Dois processos que tem acesso a uma mesma estrutura de dados não devem poder actualizá-la sem que haja algum processo de sincronização no acesso
    - Exemplo: a interrupção de um processo durante a actualização de uma estrutura de dados pode deixá-la num estado de inconsistência

Sincronização: porquê?

- O acesso concorrente a dados partilhados pode criar situações de inconsistência desses dados
  - A manutenção da consistência de dados requer mecanismos que assegurem a execução ordenada e correcta dos processos cooperantes
- **Definição**: Condição de corrida (*race condition*)
  - Situação em que vários processos acedem e manipulam dados partilhados "simultaneamente", deixando os dados num estado de possível inconsistência
- Os processos têm, pois, de ser sincronizados para impedir qualquer condição de corrida

Paradigma Produtor/Consumidor

Estrutura dos dados Partilhados

```
#define BUFFER_SIZE 10
Typedef struct {
        . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0; //pos de escrita
int out = 0; //pos de leitura
```

Produtor

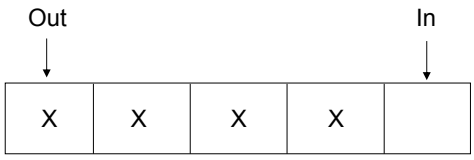
```
item nextProduced;

while (1) {
   while (((in + 1) % BUFFER_SIZE) == out)
                  ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
}
```

Consumidor

```
item nextConsumed;
while (1) {
    while (in == out)
                  ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
}
```


Paradigma Produtor/Consumidor



…
while (((in + 1) % BUFFER_SIZE) == out);
…

(4+1)%5=0

Logo esta solução apenas permite a inserção de `BUFFER_SIZE - 1` elementos no buffer!!!!

Paradigma Produtor/Consumidor

- Nova Solução

Estrutura dos dados Partilhados

```
#define BUFFER_SIZE 10
Typedef struct {
        . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0; //pos de escrita
int out = 0; //pos de leitura
// Na prática o in apenas é
 actualizado pelo produtor e o
 out pelo consumidor

int counter = 0;  //elem. no
                  buffer
```

Produtor

```
item nextProduced;

while (1) {
        while (counter == BUFFER_SIZE);
                        /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++
}
```

Consumidor

```
item nextConsumed;
while (1) {
         while (counter == 0)
                  ; /* do nothing */
         nextConsumed = buffer[out];
         out = (out + 1) % BUFFER_SIZE;
         counter--;
}
```


Paradigma Produtor/Consumidor

- Execução concorrente do programa anterior
  - As instruções `counter++` e `counter--` devem ser executadas atomicamente, i.e.:
    - sem serem interrompidas

Paradigma Produtor/Consumidor

Código máquina para:

counter++

```
register1 = counter
register1 =register1 + 1
counter = register1
```

counter--

```
register2 = counter
register2 =register2 - 1
counter = register2
```

Ordem de execução possível:

```
P1:register1 = counter = 5
P1:register1 =register1 + 1 = 6
P2:register2 = counter = 5
P2:register2 =register2 - 1
P1:counter = register1 = 6
P2:counter = register2 = 4
```

Conclusão:
Se os 2 processos tivessem sido executados correctamente, os dois processos deveriam ter chegado ao mesmo resultado – 5!!!

Race Condition

- Dado que a variável *counter* é manipulada pelos dois processo concorrentemente e não existe qualquer protecção, então a variável *counter* pode dar resultados inconsistentes para os dois processos – *Race Condition*

Secção Crítica

- n processos a tentar aceder a dados partilhados entre eles
- Cada processo têm um secção limitada de código em que acede a esse dados
  - Ex.: a variável counter

- Assegurar que quando um processo está a ser executado na sua secção crítica, nenhum outro processo poderá entrar na respectiva secção crítica.

Requisitos da Solução

1. Exclusão mútua
   - Se um processo está a executar código da sua secção crítica, então nenhum outro processo pode estar a executar código da sua secção crítica
2. Progressão
   - Se nenhum processo está a executar na sua secção crítica e há processos que desejam entrar na secção crítica, então a entrada destes na secção crítica não pode ser adiada indefinidamente
3. Espera limitada
   - Tem de existir um limite sobre o número de vezes que outros processos são permitidos entrar na sua secção crítica após um outro processo ter pedido para entrar na secção crítica e antes do respectivo pedido ser concedido

Solução

- Estrutura geral

```
1. do
2. {
3.       entry section
4.       …
5.       critical section
6.       exit section
7.       …
8.       reminder section
9. } while (1);
```


1ª Solução para 2 Processos

2 Processos, indexados como `i`(=0) e `j`(=1):
a variável `turn` é inicializada a 0.

Espera activa

```
1. do {
2.             While (turn != i);
3.             critical section
4.             turn = j; //passa a vez para o processo j
5.             reminder section
6. } while (1);
```

1ª Solução para 2 Processos

- Condições cumpridas:
  - Exclusão mútua:
    - Apenas um o processo pode aceder simultaneamente à secção crítica
  - Progressão:
    - O algoritmo obriga a que os processos acedam à secção crítica de forma alternada, i.e.: i,j,i,j,i…

2ª Solução para 2 Processos

2 Processo, indexados como `i`(=0) e `j`(=1):
A variável `bolean flag[2]` contém o estado de cada processo (i.e. se este está pronto para entrar na secção crítica), inicializada a `false`

```
1.do {
2.          flag[i] = true;
3.          While (flag[j]);
4.          critical section
5.          flag[i] = false;
6.          reminder section
7.} while (1);
```

2ª Solução para 2 Processos

- Condições cumpridas:
  - Exclusão mútua:
    - Apenas um o processo pode aceder simultaneamente à secção crítica
  - Progressão:
    - O algoritmo pode chegar a uma situação de bloqueio (ver slide seguinte)

2ª Solução para 2 Processos

- Situação de bloqueio

Processo i (P_i)	Processo j (P_j)
1. do {	
2. flag[i] = true;	Mudança de contexto
3.	do {
4.	flag[j] = true;
5. While (flag[j]);	
6.	While (flag[i]);
7. While (flag[j]);	
8.	While (flag[i]);
9. While (flag[j]);	
10.	While (flag[i];
	…

Tanto a `flag[j]` como a `flag[i]` podem ser simultaneamente iguais a `true` !!!

t

3ª Solução para 2 Processos

Combinando as ideias chave dos 2 últimos algoritmos temos:
Os dois processo devem partilhar as variáveis:

```
1.boolean flag[2]; //inicializadas a false
2.int turn; //inicializada a i ou j

3.do {
4.              flag[i] = true;
5.              turn = j;
6.              While (flag[j] && turn == j);
7.              critical section
8.              flag[i] = false;
9.              reminder section
10.} while (1);
```


3ª Solução para 2 Processos

- Condições cumpridas:
  - Exclusão mútua:
    - Embora a variável `turn` possa ser actualizada simultaneamente pelos dois processos, o seu valor final será ou `i` ou `j`, permitindo a entrada na zona crítica a apenas um dos processos.
    - O vector de *flags* garante que um processo apenas entra na zona crítica se o outro processo ainda não executou: `flag[i] == true`

3ª Solução para 2 Processos

- Condições cumpridas:
  - Progressão
    - P_i pode ficar no while enquanto `flag[j]==true && turn==i`
    - Se P_j não está pronto para entrar na secção crítica então `flag[j]==false` e o processo i pode entrar
    - Se P_j fez `flag[j]=true` e estiver também no while então a variável `turn` é igual a `j` ou a `i`, permitindo a entrada a P_j ou a P_i.
    - Se P_j entrar então na sua secção crítica, no final a sua variável `flag[j]` terá o valor `false` e caso ele prossiga novamente a sua execução até ao while, então não passará deste dado que a condição é verdadeira.
    - Concluindo: algoritmo permite a execução alternadas dos dois processos
  - Espera limitada
    - Pela exposto acima, a espera é limitada ao tempo necessário para a execução da secção crítica

Ver slide seguinte…

3ª Solução para 2 Processos

Processo i

```
1.  do {
2.        flag[i] = true;
3.
4.
5.
6.        turn = j;
7.        While (flag[j] && turn == j); //T
8.
9.
10.       While (flag[j] && turn == j); //T
11.
12.
13.
14.
15.       While (flag[j] && turn == j); //F
16.
17.
18.        //Secção Crítica
19.                          …
```

Processo j

```
do {
          flag[j] = true
          turn = i;


          While (flag[i] && turn == i); //F
          // Secção crítica

          flag[j] = false;
do {
          flag[j] = true
          turn = i;


          While (flag[i] && turn == i); //V

          …
```

Solução para n Processos

- Algoritmo de *Bakery* (padaria)
  - Ao entrar na loja cada cliente recebe um bilhete contendo um número
  - O cliente que tiver o número mais baixo será o próximo a ser servido
    - Infelizmente a sua implementação num computador não garante que todos os números são diferentes
    - O processo com menor identificador é atendido primeiro

Solução para n Processos

Estruturas partilhadas

```
boolean chosing[n];
```

- Inicializada a *false*
- Indica se o processo n está a tentar obter um bilhete

```
int number[n];
```

- Inicializada a zero
- Contêm o bilhete que permite a entrada na zona crítica a cada processo

Solução para n Processos

- Sintaxe utilizada
  - (a,b)<(c,d) é verdadeiro se:
  - a<c or (a==c and b<d)
    - Permite desempatar quando dois processos obtêm o mesmo valor para o bilhete.
    - Nota:
      - b e d correspondem ao número do processo
      - a e c correspondem ao número do bilhete obtido

Solução para n Processos

Algoritmo

2 processos podem executar este código ao mesmo tempo!!

```
1. do {
2.    choosing[i] = true;
3.    number[i] = max(number[0], number[1], …, number [n – 1])+1;
4.    choosing[i] = false;
5.    for (j = 0; j < n; j++) {
6.       while (choosing[j]); // espera que j obtenha um bilhete
7.       while ((number[j]!= 0) && (number[j],j)<(number[i],i)));
8.    }
9.    critical section
10.    number[i] = 0;
11.    remainder section
12. } while (1);
```

Solução para n Processos

- Linhas 2-4 – obtenção do bilhete
  - O P_i indica que vai obter um ticket, fazendo **`choosing[i] = true`** (linha 2)
  - O P_i determina qual o maior bilhete obtido até ao momento e fica com o maior + 1 (linha 3)
  - O P_i sinaliza que já obteve um número, fazendo **`choosing[i] = False`** (linha 4)
- Linhas 5-8 – entrada na secção crítica
  - Caso um dos processos de o até n esteja a tentar obter um bilhete, P_i espera até que este o obtenha
    `while (choosing[j]);` (linha 6)
  - P_i espera para entrar na secção crítica até ter o menor número de todos os bilhetes. Numa situação de empate entra o processo com o menor identificador.

```
while ((number[j]!= 0) &&
     (number[j],j)<(number[i],i)));
```
(linha 7)

Solução para n Processos

- Linha 10 – saída da secção crítica
  - ao fazer `number[i] = 0`, o processo i retira-se do processo de contenção
    - Note-se que na linha 7 todos os processo com `number[i] = 0` não são considerados para contenção

Exercício

1. Em que situação podem dois processos obter o mesmo número para o bilhete? Mostre a sequência de instruções que instruções que pode gerar um evento deste tipo.
2. Prove que os critérios da solução são cumpridos: exclusão mútua, progressão, espera limita

   Nota: considere que um processo necessita no máximo de C_{cri} para executar a secção crítica

Hardware de Sincronização

- Problema:
  - Como evitar interrupções por outro processo durante a execução de uma secção crítica
- 1ª Hipótese:
  - Desligar as interrupções durante o execução da secção crítica
    - Permite que as instruções na zona crítica corram sem qualquer interrupção pelo escalonador
  - Complexo e demorado em sistemas multiprocessador
  - A actualização do relógio do processador deixa de ser possível enquanto o processo se encontrar na zona crítica
- **Não viável**

Hardware de Sincronização

- TestAndSet
  - Instrução em código máquina atómica
  - Pode ser definida formalmente pelo código seguinte:

```
1. boolean TestAndSet (boolean &target)
2. {
3.     rv = target //valor de retorno
4.     target = true //set
5.     return rv //retorna o valor
   anterior do target
6. }
```


Hardware de Sincronização

- Dados partilhados:

```
boolean lock = false;
```

- Processo P_i

```
do {
//Espera até que seja devolvido false
        while (TestAndSet(lock));
                critical section
        lock = false;
                remainder section
}
```

Executada atómicamente

Hardware de Sincronização

- Swap – troca de variáveis

```
void Swap(boolean &a, boolean &b) {
    boolean temp = a;
    a = b;
    b = temp;
}
```


Hardware de Sincronização

- Process P_i

```
do {
        key = true;
        while (key == true)
                Swap(lock,key);
        critical section
        lock = false;
        remainder section
}
```

Executada atómicamente

Hardware de Sincronização

- Os algoritmos anteriores não satisfazem os requisitos de Espera Limitada
  - Porquê?
- Porque o mesmo processo, caso necessite, pode ter acesso continuo à zona crítica
  - i.e. o algoritmo não determina de forma equitativa qual o próximo processo que poderá entrar na zona crítica

Hardware de Sincronização

- Dados Partilhados (inicializados a **false**):

```
boolean lock;
boolean waiting[n];

do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
            key = TestAndSet(lock); //devolve false caso a secção
                                       crítica esteja livre
   waiting[i] = false;
            Critical Section
   j = (i+1) % n;
   while ((j != i) && !waiting[j])
            j = (j + 1) % n;
   if (j == i)
            lock = false;
   else
            waiting[j] = false;
```

Hardware de Sincronização

- Vantagens
  - Aplicável a qualquer número de processos em mono ou em sistemas multiprocessador que partilhem a mesma memória principal
  - Simples e fácil de verificar
  - Permite o suporte a n secções críticas

Hardware de Sincronização

- Desvantagens
  - Espera activa consumo tempo de processador
  - Deadlock
    - Num sistema escalonado por prioridades
    - Se um processo de baixa prioridade está a aceder à secção crítica e um outro processo de mais alta prioridade também quer aceder, então o processo de mais alta prioridade retêm o processador ficando em espera activa até que a secção crítica seja libertada, o que nunca acontecerá

Semáforos

- Mecanismo de sincronização sem espera activa
- Utilização

```
do {
  wait(mutex);
     Critical Section
  signal(mutex);
} while (1);
```


Semáforos

- Variável inteira (S) acessível apenas através de operações de **atómicas** `wait(S)` e `signal(S)`. Implementação básica:

```
wait (S) {
   while (S ≤ 0);
   S--;
}

signal (S) {
   S++;
}
```

- **Atenção esta é uma implementação que ainda requer espera activa!!!**

Semáforo

- Para evitar a espera activa, um processo que espera a libertação de um recurso deve ser bloqueado – passando para o estado de *Waiting*
- Um semáforo inclui também uma fila de espera associada ao mesmo - esta fila contém todos os descritores dos processos bloqueados no semáforo
- Quando o semáforo deixa de estar bloqueado é escolhido, da fila de espera do semáforo, um processo de acordo com um critério:
  - Prioridades
  - FCFS
- O SO deve garantir que o acesso às operações sobre o semáforo são atómicas

Semáforos

- Implementação:
  - Um semáforo é definido pela estrutura:

```
typedef struct {
 int value; // valor
 struct process *L; //Lista de processos
    } semaphore;
```

  - Assumindo as operações:
    - **block()**: passa o processo evocador para o estado de *Waiting*
    - **wakeup(P)**: passa o processo P para o estado de *Ready*

Semáforos

- Implementação:

```
void wait (Semaphore S) {
   S.value--;
     if (S.value < 0) {
             adicionar à fila S.L;
             block();
   }
}

void signal (semaphore S) {
   S.value++;
     if (S.value <= 0) {
             remover o processo P da fila S.L;
             wakeup(P);
     }
}
```


Deadlocks

- **Deadlock** – dois ou mais processos estão à espera indefinidamente por um evento que só pode ser causado por um dos processos à espera. Exemplo:
  - Sejam P_0 e P_1 dois processos que acedem aos recursos controlados pelos semáforos S e por Q:

P_0	P_1
wait(S)	*wait*(Q)
wait(Q)	*wait*(S)
…	…
signal(S)	*signal*(Q)
signal(Q)	*signal*(S)

- **Starvation** – bloqueio indefinido; um processo corre o risco de nunca ser removido da fila do semáforo, na qual ele está suspenso

Semáforos Binários

- Semáforo cujo valor apenas pode ser 0 ou 1.
- Implementação:

```
void wait(semaphore) {
   if (s.value == 1)
        s.value =  0;
   else
   {
        adicionar à fila s.L;
        block();
   }
}
```

```
void signal (semaphore S) {
   if (s.queue is empty)
        s.value =  1;
   else
   {
        remover da fila s.L;
        wakeup(P);
   }
}
```


Utilização de Semáforos

- Um semáforo s é criado com um valor inicial - os casos mais comuns são:
  - s=0 Sincronização entre processos (por ocorrência de eventos)
  - s=1 Exclusão mútua
  - s≥0 Controlo de acesso a recursos com capacidade limitada

Problemas Clássicos

- Partilha de recursos limitados
- Sincronização de execução
- Problema do Produtor/Consumidor (*Bounded-buffer)*
- Problema dos Leitores e Escritores
- Jantar dos filósofos (*Dining-Philosophers*)
- Barbeiro Dorminhoco

Partilha de Recursos Limitados

- Problema:
  - Suponha um computador que pode utilizar 3 impressoras diferentes
  - Quando o programa quer enviar dados para um impressora utiliza a função `print(obj)`, que permite imprimir na impressora que está livre. Se nenhuma impressora estiver livre a função dá erro
- Análise
  - É necessário impedir que mais do que 3 programas estejam simultaneamente a imprimir

Partilha de Recursos Limitados

- Dados partilhados
  - `Semaphore impressora;`
- Inicialização
  - `impressora=3; // vai permitir que no máximo 3 processos utilizem as impressoras em simultâneo`

Partilha de Recursos Limitados

- Cliente

```
...
Prepara doc impressora
...
wait(impressora);
 print(doc);
signal(impressora);
...
```

Sincronização de Execução

- Suponha que cria 5 processos
- Utilize semáforos de modo a que o processo 1 escreve os números de 1 ate 200, o 2 de 201 até 400…
- Como é que garante que os números vão ser escritos por ordem?
  - O processo i+1 só deve entrar em funcionamento quando processo i já terminou a escrita dos seus números

Sincronização de Execução

- Dados partilhados
  - `Semaphore S1, S2, S3, S4; //Permite a entrada em funcionamento dos processo`
- Inicialização
  - `S1=S2=S3=S4=0`

Sincronização de Execução

- P0

```
…
Imprime os números de 1
até 200
signal(S1);
…
Termina
```

- Pi

```
…
wait(Si);
…
Impirme números de
i*200+1 até (i+1)*200
…
signal(Si+1)
…
Termina
```


Problema Produtor/Consumidor

- Problema:
  - Suponha que existem dois processos:
    - Um que produz os dados, p.e. um processo que descodifica uma sequência de vídeo o coloca num *buffer* a imagem em *bit map*
    - Outro que passa o *bit map* para o ecrã
- Analise do problema
  - O processo produtor apenas pode colocar dados no buffer se este estiver vazio
  - O processo consumidor fica à espera de dados se o buffer estiver vazio
  - O acesso aos dados deve ser exclusivo

Produtor/Consumidor

- Dados Partilhados

```
Semaphore full; //nº de posições ocupadas no buffer
Semaphore empty;//nº de posições livres no buffer
Semaphore mutex;//Permite o acesso exclusivo à secção crítica
```

- Valores iniciais:

```
full = 0, empty = n, mutex = 1
```


Produtor/Consumidor

Produtor:

```
do {
   …
   produce an item
    …
   wait(empty);
   wait(mutex);
    …
   add item to buffer
    …
   signal(mutex);
   signal(full);
} while (1);
```

Consumidor:

```
do {
   …
   wait(full);
   wait(mutex);
    …
   remove an item from buffer
    …
   signal(mutex);
   signal(empty);
    …
   consume the item
    …
} while (1);
```

Problema dos Leitores e Escritores

- Problema:
  - Suponha que existe um conjunto de processos que partilham um determinado conjunto de dados
  - Existem processos que lêem os dados (mas não os apagam!)
  - Existem processos que escrevem os dados
- Analise do problema
  - Se dois ou mais leitores acederem aos dados simultaneamente não existem problemas
  - E se um escritor escrever sobre os dados?
    - Podem outros processo estar a aceder simultaneamente aos mesmos dados?

Problema dos Leitores e Escritores

- Solução
  - Os escritores apenas podem ter acesso exclusivo aos dados partilhados
  - Os leitores podem aceder aos dados partilhados simultaneamente
  - A solução proposta é simples mas pode levar à *starvation* do escritor
- Dados partilhados
  - `int readcount //número de leitores activos`
  - `Semaphore mutex //protege o acesso à variável readcount`
  - `Semaphore wrt //Indica a um escritor se este pode aceder aos dados`
  - Incialização: `mutex=1`, `wrt=1`, `readcount=0`

Problema dos Leitores e Escritores

- Escritor

```
wait(wrt);
…
writing is performed
…
signal(wrt);
```

- Leitor

```
wait(mutex);
readcount++;
if (readcount == 1)
      wait(wrt);
signal(mutex);
…
reading is performed
…
wait(mutex);
readcount--;
if (readcount == 0)
      signal(wrt);
signal(mutex);
```


Problema dos Leitores e Escritores

- Sugestão de trabalho
  - Procurar outras soluções para o problema
  - Desenvolver uma solução que atenda os processo pela ordem de chegada

Problema dos Leitores e Escritores

- Outra solução:
  - Quando existir um escritor pronto escrever este tem prioridade sobre todos os outros leitores que cheguem entretanto à secção crítica
- Dados partilhados
  - `Readcount //número de leitores`
  - `Writecount //número de escritores, apenas pode estar um escritor de cada vez a aceder aos dados partilhados`
  - `mutex1 //protege o acesso à variável readcount`
  - `mutex2 //protege o acesso à variável writecount`
  - `mutex3 //impede que + do que 1 leitor esteja a tentar entrar na secção crítica`
  - `w //Indica a um escritor se este pode aceder aos dados`
  - `r //Permite que um processo leitor` **`tente`** `entrar na sua secção crítica`

Problema dos Leitores e Escritores

- Inicialização
  - `readcount = 0`
  - `writecount = 0`
  - `mutex1 = 1`
  - `Mutex2 = 1`
  - `Mutex3 = 1`
  - `w = 1`
  - `r = 1`

Problema dos Leitores e Escritores

- Escritor

```
wait(mutex2);
  writecount++;
  if (writecount == 1)
    wait(r);
wait(mutex2);
wait(w)
…
   Escrita
…
signal(w)
wait(mutex2);
  writecount--;
  if (writecount == 0)
     signal(r);
signal(mutex2);
```

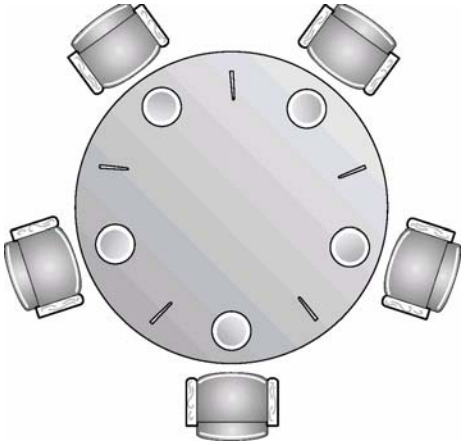
- Leitor

```
wait(mutex3);
 wait(r);
  wait(mutex1);
   readcount++;
   if (readcount == 1)
        wait(w);
  signal(mutex1);
 signal(r);
signal(mutex3);
…
Leitura dos dados
…
wait(mutex1);
 readcount--;
 if (readcount == 0)
   signal(w);
signal(mutex1);
```


Problema dos Leitores e Escritores

- O primeiro escritor a passar por `wait(r)` vai impedir novos leitores de progredir para além da segunda linha (`wait(r)`)
- Se existirem leitores na zona crítica o escritor não entra dado que o semáforo `w` foi colocado a zero pelo primeiro processo leitor, e só será novamente colocado a 1 quando não existirem processos leitores

Jantar dos Filósofos



Jantar dos Filósofos

- Considere 5 filósofos que passam a vida a comer e a pensar
- Partilham um mesa circular, com um tacho de arroz ao centro
- Na mesa existem 5 pauzinhos, colocados um de cada lado do filósofo
- Quando um filósofo fica com fome pega nos dois pauzinhos mais próximos, um de cada vez e come até ficar saciado
- Quando acaba de comer pousa os pauzinhos e fica de novo a pensar

Jantar dos Filósofos

- Representa o conjunto de problemas referentes a como alocar vários recursos a vários processo evitando *deadlocks* e *startvation*

Jantar dos Filósofos

- Solução
  - Cada pauzinho é representado por um semáforo:

```
semaphore chopstick[5]; //Inicializados a 1
```

  - A operação de pegar num pauzinho é emulada através de uma operação de wait() sobre o respectivo semáforo

Jantar dos Filósofos

```
Filósofo i
   do {
        wait(chopstick[i])
        wait(chopstick[(i+1) % 5])
         …
        eat
         …
        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);
        …
        think
        …
   } while (1);
```


Jantar dos Filósofos

- Problemas da solução:
  - a situação em que todos os filósofos levantam ao mesmo tempo o pauzinho da mão direita leva a um deadlock. Como resolver?
    - Permitir que apenas 4 filósofos se sentem à mesa
    - Permitir que um filósofo pegue nos pauzinho apenas se se encontrarem os dois disponíveis
    - Usar uma solução assimétrica. I.e. um filósofo par pega primeiro no pauzinho da direita e um impar no da esquerda
  - Qualquer solução deve garantir que nenhum filósofo morra à fome

O Barbeiro Dorminhoco

- A barbearia consiste numa sala de espera com n cadeiras mais a cadeira do barbeiro
- Se não existirem clientes o barbeiro fica a dormir
- Ao chegar um cliente:
  - Se todas as cadeiras estiverem ocupadas, este vai-se embora
  - Se o barbeiro estiver ocupado, mas existirem cadeiras então o cliente senta-se e fica à espera
  - Se o barbeiro estiver a dormir o cliente acorda-o e corta o cabelo

O Barbeiro Dorminhoco

- Dados partilhados:
  - `semaphore cliente`: indica a chegada de um cliente à barbearia
  - `semaphore barbeiro`: indica se o barbeiro está ocupado
  - `semapho mutex`: protege o acesso à variável `lugares_vazios`
  - `int lugares_vazios`
- Inicialização:
  - `barbeiro=0; mutex=1; lugares_vazios=5; cliente=0`

O Barbeiro Dorminhoco

- Barbeiro

```
do {
 wait(cliente); //barbeiro a
   dormir
 ...
 Corta cabelo
 ...
 signal(barbeiro)
} while (1)
```

- Cliente

```
wait(mutex)
 if (lugares_vazios==0)
   exit();
 else
   lugares_vazios--;
signal(mutex);
signal(cliente);

wait(barbeiro); //corta cabelo
wait(mutex)
 lugares_vazios++;
signal(mutex)
```


Sistemas Operativos I

Sincronização de Processos

Luis Lino Ferreira / Maria João Viamonte

Abril de 2006