

Sistemas de Computadores

Sincronização de Processos

Luis Lino Ferreira

Maria João Viamonte

Luis Nogueira

Março de 2009

Sumário

- Comunicação entre processos
- Problemas inerentes à gestão de recursos
- Inconsistência de dados versus sincronização de processos
- O problema das secções críticas
- Semáforos
- Problemas clássicos de sincronização

Comunicação entre processos

- Como é que um processo pode comunicar com outro processo?
 - Filas de Mensagens
 - Ficheiros
 - Sockets
 - **Pipes**
 - **Memória Partilhada**

Filas de mensagens

- As filas de mensagens permitem trocar mensagens entre 2 processos
- As mensagens podem ser lidas numa ordenação FIFO pelo processo receptor

Ficheiros

- Escrevendo e lendo em ficheiros também permite a comunicação entre processos.
- Mecanismos pouco eficiente dado que os dados são normalmente escritos no disco

Sockets

- Enviando mensagens entre processos diferentes, mas a correr na mesma máquina
- O funcionamento utilizada as mesmas primitivas utilizadas para a comunicação entre computadores através da rede, mas o Kernel ao detectar que a comunicação é interna efectua a troca de mensagens de maneira mais eficiente.

Pipes

- Permitem estabelecer um canal para a troca de dados
- Diferem das filas de mensagens dado que os dados podem ser lidos byte a byte

Memória Partilhada

- Um processo apenas tem acesso a uma zona privada de memória
- Para partilhar uma zona de memória um processo deve explicitamente pedi-la ao sistema operativo, e este deve configurar as áreas de memória de cada processo de modo a que ambos apontem para a mesma zona

Memória Partilhada

- As várias regiões de memória partilhadas são armazenadas pelo kernel num local especial de memória denominada ***Region Table***
- O kernel mantém um ***array*** denominado ***Shared Memory Table***, em que cada entrada possui informação referente ao nome, permissões e tamanho da região correspondente, assim como o apontador para a ***Region Table*** onde está armazenado o pedaço de memória partilhado

Memória Partilhada em Linux

- O Linux divide a memória em zonas endereçáveis linearmente chamadas *memory regions*. Caracterizadas por:
 - Endereço inicial
 - Comprimento
 - Permissões de acesso
 - O endereço inicial e final devem ser múltiplos de 4096
 - Consequência?
- Estas regiões são alocadas a um processo:
 - Na sua criação
 - Devido a ter esgotado a sua stack
 - Caso necessite carregar um programa completamente novo (se utilizar uma função `exec`)
 - Caso decida mapear um ficheiro em memória
 - Se necessitar de mais espaço de memória (`malloc()`)
 - Se quiser criar/aceder a uma região de memória partilhada

Memória Partilhada em Linux

- Como obter uma região de Memória Partilhada (MP) em Linux?
 - **Caso 1:** se a região de MP ainda **não existe**
 1. Alocar espaço para a região de memória partilhada e obter o respectivo descritor
 - Nesta operação cada região de MP é associada a uma chave
 2. Ligar a região ao espaço de endereçamento do processo
 3. Operar sobre a região
 4. Apagar ou desligar-se da região de MP

Memória Partilhada em Linux

- Como obter uma região de Memória Partilhada (MP) em Linux?
 - **Caso 2: se a região de MP já existe**
 1. Obter um descritor para a região de MP anteriormente criada
 - Utilizando a mesma chave
 2. Ligar a região ao espaço de endereçamento do processo
 3. Operar sobre a região
 4. Apagar ou desligar-se da região de MP

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 de ser sincronizados para impedir qualquer condição de corrida

Paradigma Produtor/Consumidor

Estrutura dos dados Partilhados

```
1. #define BUFFER_SIZE 10
2. Typedef struct {
3.     . . .
4. } item;
5. item buffer[BUFFER_SIZE];
6. int in = 0; //pos de escrita
7. int out = 0; //pos de leitura
```

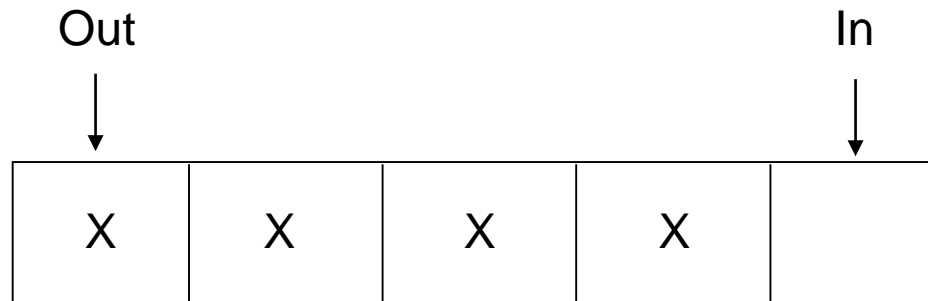
Produtor

```
1. while (1) {
2.     item nextProduced;
3.     while (((in + 1) % BUFFER_SIZE) == out)
4.         ; /* do nothing */
5.     buffer[in] = nextProduced;
6.     in = (in + 1) % BUFFER_SIZE;
7. }
```

Consumidor

```
1. item nextConsumed;
2. while (1) {
3.     while (in == out)
4.         ; /* do nothing */
5.     nextConsumed = buffer[out];
6.     out = (out + 1) % BUFFER_SIZE;
7. }
```

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

```
1. #define BUFFER_SIZE 10
2. typedef struct {
3.     . . .
4. } item;

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

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

Produtor

```
1. item nextProduced;
2. while (1) {
3.     nextProduced = ...
4.     while (counter == BUFFER_SIZE);
5.         /* do nothing */
6.     buffer[in] = nextProduced;
7.     in = (in + 1) % BUFFER_SIZE;
8.     counter++;
9. }
```

Consumidor

```
1. item nextConsumed;
2. while (1) {
3.     while (counter == 0)
4.         ; /* do nothing */
5.     nextConsumed = buffer[out];
6.     out = (out + 1) % BUFFER_SIZE;
7.     counter--;
8. }
```

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:

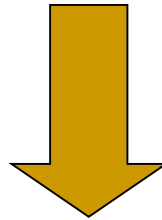
Se os 2 processos tivessem sido executados correctamente, os dois processos deveriam ter chegado ao mesmo resultado – 5!!!

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

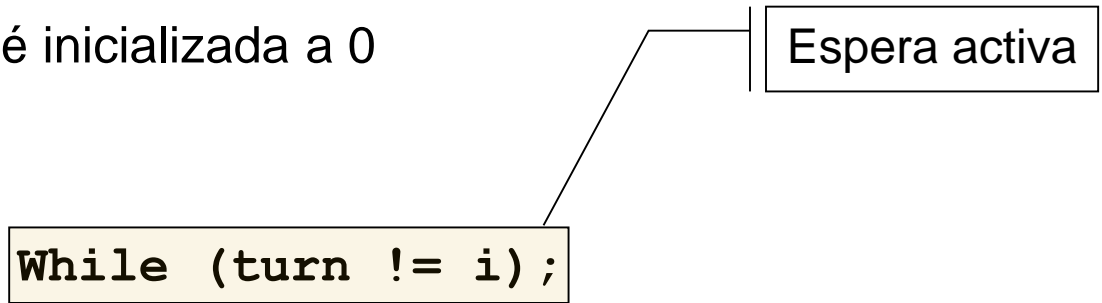
2 Processos, indexados como $i(=0)$ e $j(=1)$:

a variável `turn` é inicializada a 0

1.do {

2.

While (turn != i);



A diagram consisting of a rectangular box with a black border containing the text 'Espera activa'. A thin black line extends from the bottom-left corner of this box, angled downwards and to the left, ending at the right side of the yellow-highlighted box containing the code '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 `boolean 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)

```
1.  do {  
2.      flag[i] = true;  
3.  
4.  
5.      While (flag[j]);  
6.  
7.      While (flag[j]);  
8.  
9.      While (flag[j]);  
10.
```

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

Processo j (P_j)

```
Mudança de contexto  
do {  
    flag[j] = true;  
    While (flag[i]);  
    While (flag[i]);  
    While (flag[i]);  
    While (flag[i];  
    ...
```

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 choosing[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 $(a == c \text{ and } b < d)$
 - 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

```
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);
```

2 processos podem executar
este código ao mesmo tempo!!

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 0 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 pode gerar um evento deste tipo
2. Prove que os critérios da solução são cumpridos: exclusão mútua, progressão, espera limitada

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

Semáforos

- Mecanismo de sincronização sem espera activa
- Utilização

```
do {  
    down (mutex) ;  
        Critical Section  
    up (mutex) ;  
} while (1) ;
```


Semáforos

- Variável inteira (S) acessível apenas através de operações de **atômicas** `down(S)` e `up(S)`. Implementação básica:

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

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

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

Semáforos

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

void up (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
$down(S)$	$down(Q)$
$down(Q)$	$down(S)$
...	...
$up(S)$	$up(Q)$
$up(Q)$	$up(S)$

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

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
 - $s \geq 0$ 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

■ Clientes

```
/*inicializa semáforos */  
Impressora = 3  
...  
Prepara documento para imprimir  
...  
down (impressora);  
    print (doc);  
up (impressora);  
...
```

Sincronização de Execução

- Suponha que cria 5 processos (1-5)
- Utilize semáforos de modo a que o processo 1 escreva os números de 1 até 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 S2, S3, S4, S5; //Permite a entrada em funcionamento do processo i (2-5)

- Inicialização

- S2=S3=S4=S5=0

Sincronização de Execução

■ P1

...

Imprime os números de 1
até 200

`up(S2);`

...

Termina

■ Pi (2-5)

...

`down(Si);`

...

Imprime números de $(i-1)*200+1$ até $i*200$

...

`up(Si+1)`

...

Termina

Problema Produtor/Consumidor

■ Problema:

□ Suponha que existem dois processos:

■ Um que produz os dados, por ex:

- um processo que descodifica uma sequência de vídeo e coloca num *buffer* a imagem em *bit map*

■ Outro que periodicamente coloca o *bit map* na placa gráfica

■ Análise

- O processo produtor apenas pode colocar dados no *buffer* se este tiver posições livres

- 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; //n° de posições ocupadas no buffer`

`Semaphore empty; //n° 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  
    ...  
    down(empty);  
    down(mutex);  
    ...  
    add item to buffer  
    ...  
    up(mutex);  
    up(full);  
} while (1);
```

Consumidor:

```
do {  
    ...  
    down(full);  
    down(mutex);  
    ...  
    remove an item from  
    buffer  
    ...  
    up(mutex);  
    up(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

■ Análise

- ❑ 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 no slide seguinte 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`
- ❑ **Inicialização:** `mutex=1, wrt=1, readcount=0`

Problema dos Leitores e Escritores

■ Escritor

```
down(wrt);  
...  
writing is performed  
...  
up(wrt);
```

■ Leitor

```
down(mutex);  
readcount++;  
if (readcount == 1)  
    down(wrt);  
up(mutex);  
...  
reading is performed  
...  
down(mutex);  
readcount--;  
if (readcount == 0)  
    up(wrt);  
up(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

■ Escriitor

```
down(mutex2);
    writecount++;
    if (writecount == 1)
        down(r);
up(mutex2);

down(w)
...
    Escrita
...
up(w)

down(mutex2);
    writecount--;
    if (writecount == 0)
        up(r);
up(mutex2);
```

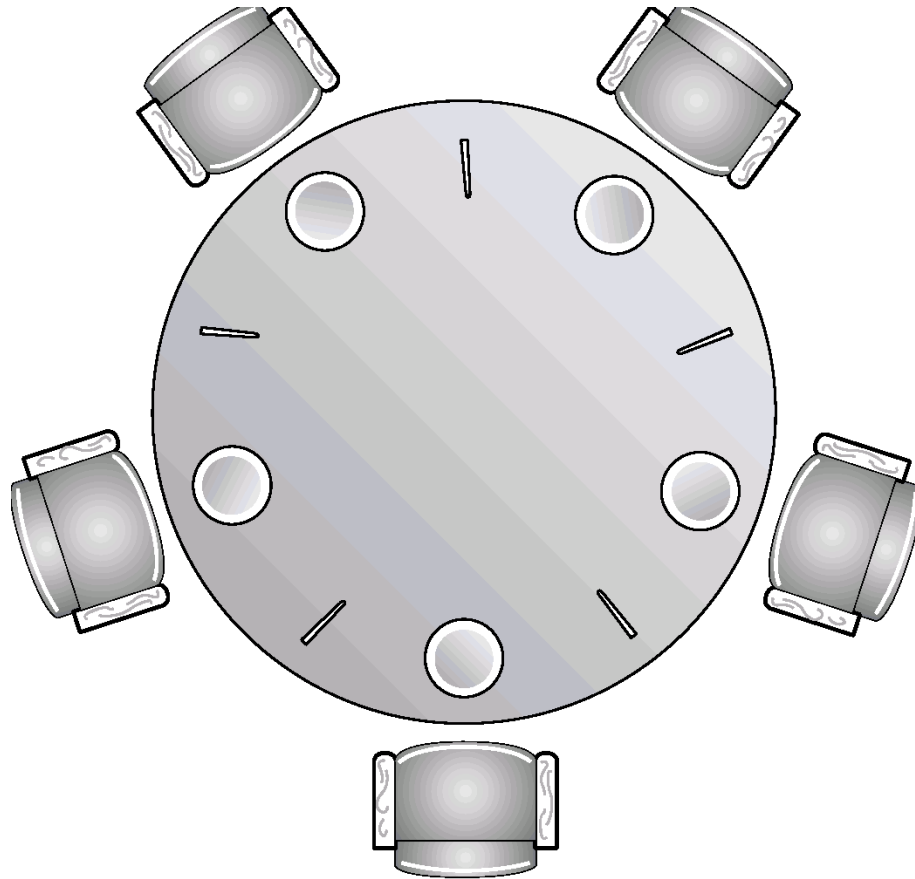
■ Leitor

```
down(mutex3);
    down(r);
    down(mutex1);
    readcount++;
    if (readcount == 1)
        down(w);
    up(mutex1);
    up(r);
up(mutex3);
...
Leitura dos dados
...
down(mutex1);
    readcount--;
    if (readcount == 0)
        up(w);
up(mutex1);
```

Problema dos Leitores e Escritores

- O primeiro escritor ao passar por $\text{down}(r)$ vai impedir novos leitores de progredir para além da segunda linha ($\text{down}(r)$)
- 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 uma mesa circular, com um tacho de arroz ao centro
- Na mesa existem 5 pauzinhos, colocados um de cada lado de um 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 *starvation*

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 `down()` sobre o respectivo semáforo

Jantar dos Filósofos

Filósofo i

```
do {  
    down(chopstick[i])  
    down(chopstick[(i+1) % 5])  
    ...  
    eat  
    ...  
    up(chopstick[i]);  
    up(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 ímpar 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
- `semaphore mutex`: protege o acesso à variável `lugares_vazios`
- `int lugares_vazios`

■ Inicialização:

- `barbeiro=0; mutex=1; lugares_vazios=5; cliente=0`

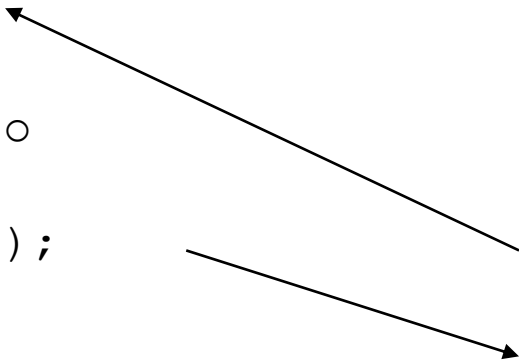
O Barbeiro Dorminhoco

■ Barbeiro ≈ Servidor

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

■ Cliente

```
down(mutex)  
    if (lugares_vazios==0)  
        up(mutex); exit();  
    else  
        lugares_vazios--;  
up(mutex);  
up(cliente);  
  
down(barbeiro); //corta  
    cabelo  
down(mutex)  
    lugares_vazios++;  
up(mutex)
```



Bibliografia

- Silberschatz, Galvin and Gagne, “Operating Systems Concepts 7th Edition”, John Wiley and Sons, Inc

Sistemas de Computadores

Sincronização de Processos

Luis Lino Ferreira

Maria João Viamonte

Luis Nogueira

Março de 2009