COMUNICAÇÃO E SINCRONIZAÇÃO ENTRE PROCESSOS

Escola Superior de Tecnologia e Gestão de Beja
Curso de Engenharia Informática
Sistemas Operativos

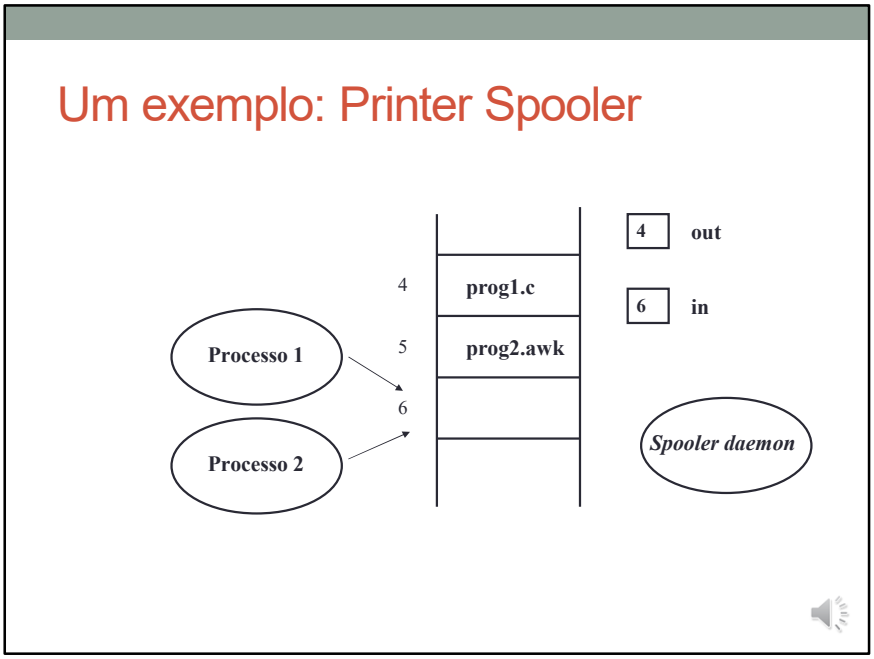
Lu

Olá! Num sistema multiprogramação os processos podem necessitar de comunicar entre si, e sincronizar as suas ações.
Nestes slides iremos apresentar os principais conceitos e mecanismos relacionados com a comunicação e sincronização entre processos.

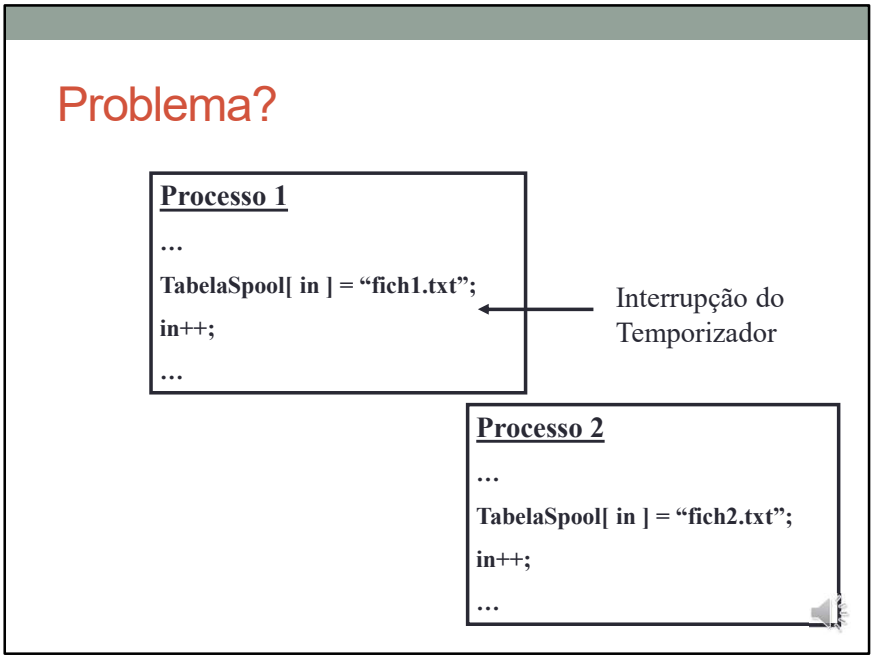
Interacção entre Processos

- Os processos num sistema multiprogramação necessitam frequentemente de interagir:
  - Partilha de recursos
  - Sincronização
  - Comunicação de dados

Num sistema multiprogramação é comum o desenvolvimento de aplicações baseadas em vários processos, onde cada processo tem uma função específica.
Nestes casos, os vários processos poderão ter de partilhar variáveis (a partilha de recursos), ter de esperar que outro processo termine uma determinada tarefa (a sincronização), e ainda ter de enviar dados a outro processo (a comunicação).
Por isso os sistemas operativos têm de disponibilizar mecanismos que permitam estas formas de interação entre os processos.



Um exemplo, é o spooler de impressão, que gere os pedidos de impressão dos processos num sistema operativo multiprogramação.
Por um lado temos os processos que pretendem imprimir os seus ficheiros, o processo 1 e o processo 2 na figura.
Do outro lado temos o processo que trata dos pedidos de impressão, o spooler deamon.
No exemplo apresentado, os processos que pretendem imprimir um documento devem colocar o nome do ficheiro no fim da fila de impressão.
No momento representado na figura o fim da fila de impressão é a posição 6 do array - o valor indicado pela variável in.
Para imprimir o próximo documento, o spooler daemon retira o ficheiro no início da fila, na figura a posição 4 do array, o valor da variável out.



No entanto, como iremos ver, este tipo de interações, onde vários processos acedem às mesmas variáveis, podem trazer problemas inesperados.
Iremos fornecer um exemplo.
Pode suceder o processo 1 pretender imprimir o ficheiro fich1.txt, pelo que irá colocar esse ficheiro no fim da fila, ou seja na posição indicada pela variável in, que vimos ser a posição 6 do vetor.
E logo a seguir, antes de incrementar a variável in, pode acabar o seu tempo de processador, e assim ser interrompido para que outro processo possa correr.
Deve aqui notar-se que o processo colocou o seu ficheiro na posição 6 do array, a posição indicada pela variável in, mas não conseguiu incrementar esta varivável para 7.
Se a seguir o processo 2 for imprimir um ficheiro irá colocar o seu ficheiro também na posição 6, eliminando o ficheiro do processo anterior.
Nos slides seguintes iremos olhar para este problema, e discutir algumas soluções.

Regiões Criticas

- Regiões criticas são zonas do programa no qual é efectuado uma operação sobre um recurso partilhado (e.g. variável)

Região critica do exemplo anterior	**TabelaSpool[in] = "fich2.txt";** **in++;**

- **Solução:** apenas um processo pode estar na região critica (**exclusão mútua**)

O problema apresentado no exemplo anterior surge porque os dois processos alteraram de forma não sincronizada duas variáveis relacionadas, o array com a fila de impressão, e a variável que in que indica o fim de fila, e como resultado a fila de impressão fica corrompida.
Para se resolver este problema é necessário garantir que quando um processo está a alterar as variáveis partilhadas, nenhum outro o faz, até que o primeiro conclua a sua operação.
A zonas nos programas onde são utilizadas variáveis partilhadas são denominadas de regiões criticas, e deve-se garantir que apenas um processo se encontra numa determinada zona critica.
Assim, a presença de um processo na região crítica deve excluir a presença de todos os outros processos com interesse nessa região crítica, dai utilizar-se o termo garantir a exclusão mútua.

Garantir a Exclusão Mútua

- Existem diversos mecanismos que permitem garantir a exclusão mútua numa região crítica:
  - Desligar as interrupções
  - Utilizar uma flag (não é tão fácil como parece)
  - TSL (instrução Test And Set Lock)
  - Semáforos

Existem diversas formas que permitem garantir a exclusão mútua numa região crítica.
Nestes slides apresentamos algumas possíveis soluções.
Iremos nos próximos slides olhar para cada uma destas possibilidades.

Desligar as interrupções

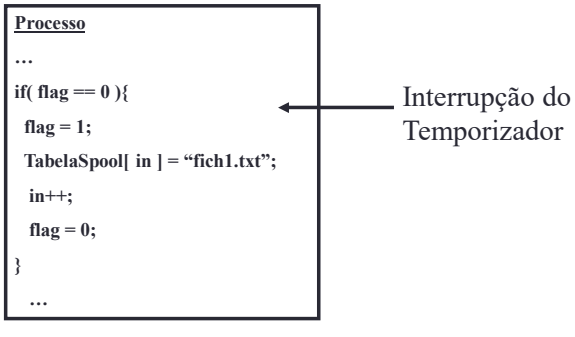
- Desligando as interrupções impede-se que o processo seja interrompido na região crítica
- Mas dar este privilégio a qualquer processo pode ser critico…

```
desligar_interrupções;
TabelaSpool[ in ] = "fich2.txt";
in++;
ligar_interrupções;
```

Uma forma simples poderia consistir em permitir aos processos poderem desligar as interrupções.
Desta forma garantia-se que os processos não seriam interrompido dentro da região crítica.
Mas permitir que qualquer processo possa realizar esta operação não é razoável num sistema operativo.
Se um processo não voltasse a ligar as interrupções, por lapso ou de forma propositada, então nunca mais seria retirado do processador, e a distribuição do processador pelos vários processos deixaria de acontecer.

Uma simples flag não é suficiente…

- Solução incorrecta.
- Outros algoritmos mais complexos solucionam este problema: algoritmo de Dekker, algoritmo de Peterson



Outra solução poderia consistir na utilização de uma variável do tipo flag, que indicasse a existência ou não, de um processo na região critica.
Esta solução parece funcionar, mas não…
Segundo esta ideia, um processo antes de entrar na região crítica deveria verificar o valor de uma variável do tipo flag.
Se a flag estivesse a zero isso queria dizer que a região crítica estaria livre, e o processo poderia avançar.
Colocando logo a seguir, a flag a um, para indicar a existência de um processo na região crítica.
Após sair da zona crítica o processo também teria de colocar a variável novamente a zero para permitir que outros processos pudessem agora entrar.
O problema é que, após verificar que a flag está a zero, um processo pode ser interrompido, e não conseguir colocar a flag a 1.
Se o processo que corre a seguir tentar entrar na região critica vai encontrar a flag a 0, e vão estar ambos na região crítica.
Existem algoritmos, como o algoritmo de Dekker, e o algoritmo de Peterson, que eliminam este problema utilizando várias variáveis do tipo flag.

A Instrução *TestAndSet*

- A Instrução *TestAndSet* que deve ser implementada pelo hardware

```
boolean TestAndSet (int i) {
          if (i == 0) {
               i = 1;
               return true;
          }
          else {
               return false;
          }
     }
```

O problema da solução anterior surge porque, entre a verificação que a flag está a 0, e a sua colocação a 1, o processo pode ser interrompido.
Isto acontece porque tratam-se de duas instruções separadas, e entre duas instruções pode surgir uma interrupção do temporizador que para um processo e inicia outro.
Mas se a verificação que a flag está a 0, e a sua colocação a 1, for realizada numa única instrução, ou seja de forma atómica, então a solução apresentada no slide anterior funciona.
Por esse motivo, os processador atuais oferecem este tipo de instrução, que verifica e coloca um valor numa variável.
Na figura mostra-se em pseudocódigo o funcionamento desta instrução no hardware.

Garantia da Exclusão Mútua com *TestAndSet*

- Esta solução garante a exclusão mútua

Processo 1

```
…
if( TestAndSet( flag ) ){
  TabelaSpool[ in ] = “fich1.txt”;
   in++;
   flag = 0;
}
   …
```

Com a instrução TesteAndSet a solução para o problema apresentado seria esta.

Semáforos

- Utilização de uma variável especial (e.g. S), denominada semáforo, para sinalização
- O semáforo é uma variável que contém um valor inteiro
  - Pode ser inicializado com um valor não negativo
  - A operação *Down* decrementa o semáforo, mas o valor do semáforo não pode baixar de zero
  - Um *Down* num semáforo a zero bloqueia o processo
  - A operação *Up* incrementa o valor do semáforo
- Semáforos binários
  - Apenas assumem dois valores: 0 ou 1

Outro mecanismo de mais alto nível, que permite garantir a exclusão mútua, são os semáforos.
Neste mecanismo, o semáforo é uma variável especial que pode ser inicializada com o valor zero, ou outros valores positivos.
E oferece duas operações possíveis.
A operação Down que decrementa o valor do semáforo se este for positivo, mas que bloqueia o processo se o semáforo estiver a 0.
A operação Up incrementa o valor do semáforo, e nunca bloqueia um processo.
Os semáforos binários são semáforos que podem conter apenas os valores 0 ou 1.

Garantia da Exclusão Mútua com Semáforos

- Inicializando o semáforo S com o valor 1 garante-se a exclusão mútua

Processo 1

```
Down( S )
TabelaSpool[ in ] = "fich1.txt";
in++;
Up( S )
```

Processo 2

```
Down( S )
TabelaSpool[ in ] = "fich2.txt";
in++;
Up( S )
```

Neste exemplo mostra-se como pode ser utilizado um semáforo binário para a solução do problema do spooler de impressão.
A solução consiste em inicializar o semáforo s a 1, e obrigar todos os processos a realizar um down no semáforo antes de avançarem para a região crítica.
Se um processo encontrar o semáforo s igual a 1 então conseguirá entrar para a região crítica, mas deixará o semáforo igual a 0 após o Down.
Se entretanto surgir um processo a querer entrar na região crítica, tentará realizar o Down, mas como o semáforo estará igual a zero, ficará bloqueado, à espera.
Quando o processo na região crítica sair, realizará um Up no semáforo que o colocará novamente a 1.
Isso permitirá que outros processos, em espera ou não, possam realizar os respetivos Downs para aceder à região crítica.

Sincronização entre Processos

- Inicializando o semáforo S com o valor zero garante-se a ordem pela qual os processos escrevem no ecrã

Processo 1

```
for( i = 0; i < 10; i++ )
   printf( "%d\n", i );
Up( S )
```

Processo 2

```
Down( S )
for( i = 10; i < 20; i++ )
   printf( "%d\n", i );
```

O mecanismo dos semáforos também permite sincronizar processos.
No exemplo apresentado neste slide coloca-se o processo 2 à espera que o processo 1 conclua uma tarefa, neste caso a escrita dos números de 1 a 9 no ecrã.
O processo 2 é obrigado a esperar pois realiza um down num semáforo s que é inicializado a 0.
Por isso fica bloqueado.
Só o processo 1 poderá libertar o processo 2, realizando um Up nesse semáforo.
Como podemos verificar, depois de terminar a escrita dos números, o processo 1 realiza um Up no semáforo s, que desbloqueará o processo 2, e permite que este escreva os números de 10 a 19 no ecrã.

Mensagens

- Além da sincronização entre processos, estes também podem necessitar de trocar informações.
- As operações básicas deste mecanismo:
  - send (destinatário, mensagem)
  - receive (remetente, mensagem)
- O mecanismo de mensagens também permite sincronizar processos

Além da sincronização entre processos, pode também ser necessário os processos trocarem informações entre si.
O mecanismo de mensagens é uma das formas de comunicação entre processos mais comuns.
Este mecanismo disponibiliza normalmente duas funções aos processos, uma para o envio de mensagens, a função send, e outra para a receção de mensagens, a função receive.
Neste slide apresenta-se um exemplo destas funções.
Para se enviar uma mensagem com esta função send deve indicar-se o processo destinatário, e a seguir a mensagem a enviar.
O processo que pretende receber uma mensagem deve chamar a função receive, indicando o processo remetente, e uma variável onde ficará armazenada a mensagem recebida.
O mecanismo de mensagens também permite sincronizar processos como veremos a seguir.

Sincronização no Envio e Recepção das Mensagens (1)

- O emissor e receptor podem ou não ficar bloqueados à espera da mensagem
- Envio bloqueante, Recepção bloqueante
  - O emissor e receptor ficam ambos bloqueados até à entrega da mensagem
  - Chamado um *rendezvous*

Quanto à forma como os processos se sincronizam para o envio e receção das mensagens, existem três variantes.
Na primeira, ambos os processos bloqueiam até à entrega da mensagem.
O processo que envia, bloqueia na função send, e o processo que recebe, bloqueia na função receive.
Quando ambos os processos chegarem a estas funções, trocam a mensagem, e continuam o seu programa.

Sincronização no Envio e Recepção das Mensagens (2)

- Envio não bloqueante, Recepção bloqueante
  - O emissor continua o processamento…
  - O receptor é bloqueado até à chegada da mensagem
- Envio não bloqueante, Recepção não bloqueante
  - Nenhuma das partes tem de esperar

A segunda possibilidade consiste num envio não bloqueante, e numa receção bloqueante.
Ou seja, o processo que envia chama a função send, e mesmo que o outro processo não esteja a realizar o receive, prossegue o seu programa.
O processo que recebe, se realizar um receive antes da mensagem ser enviada, ficará bloqueado à espera que tal aconteça.
Esta variante é bastante utilizada pois o processo que recebe a mensagem necessita normalmente dos dados contidos na mensagem para prosseguir o seu processamento, e por isso faz sentido deixar o processo bloqueado à espera que tal aconteça.
A terceira possibilidade consiste em termos um envio e uma recção não bloqueantes.
Nesta variante nenhuma das partes tem de esperar.
É a forma mais flexível de utilização do mecanismo das mensagens, mas tal irá acarretar também mais algum esforço de programação.
Por exemplo, se num dado momento o processo recetor tentar receber a mensagem e esta ainda não tiver chegado, poderá ir realizar outra tarefa, mas mais tarde terá de tentar receber a mensagem.

Endereçamento da Origem e Destino da Mensagem (1)

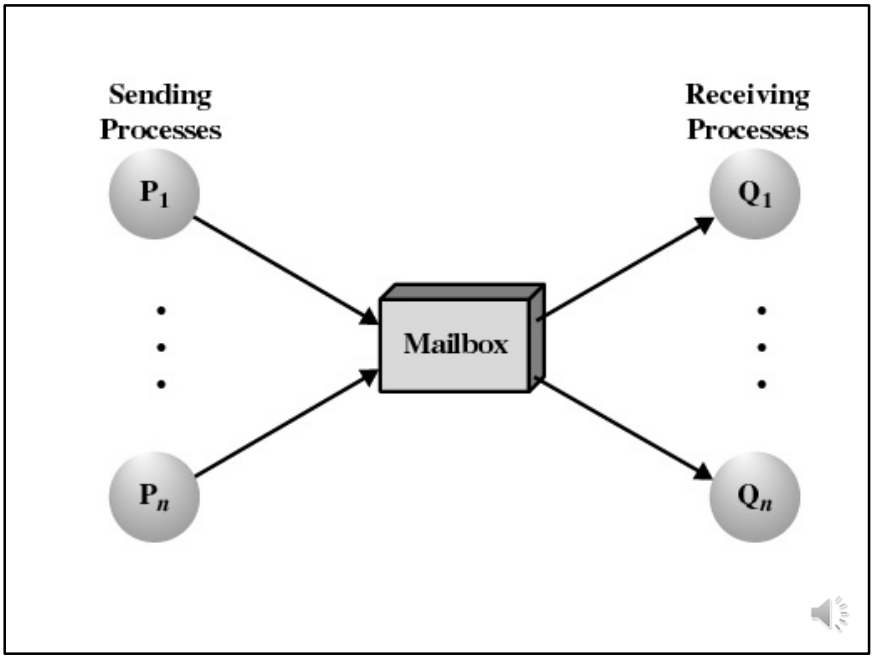
- Endereçamento directo
  - As primitivas *send* contêm um parâmetro que especifica o identificador do processo destino
    - `send (destinatário, mensagem)`
  - As primitivas *receive* contêm um parâmetro que especifica o identificador do processo origem
    - `receive (remetente, mensagem)`

As funções de send e receive podem referenciar diretamente qual o processo destinatário e qual o processo remetente da mensagem.

Endereçamento da Origem e Destino da Mensagem (2)

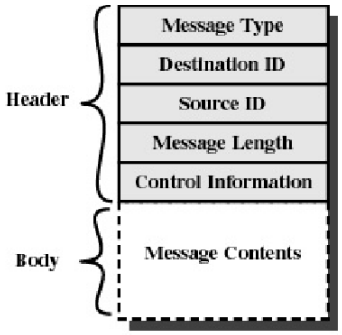
- Endereçamento indirecto
  - As mensagens são enviadas para uma estrutura partilhada denominada de caixa de correio (mailbox)
  - Um processo envia um mensagem para uma caixa de correio e outro processo retira a mensagem da caixa de correio

Ou então podem referenciar um objeto externo, uma caixa de correio, à qual um ou mais processos podem aceder para o envio e recção das mensagens.

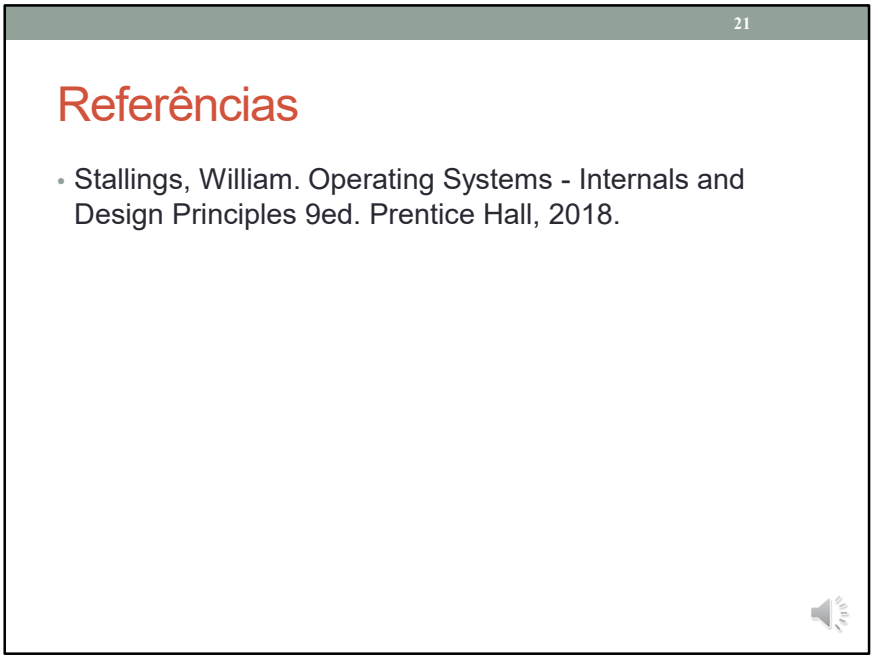


Esta solução da caixa de correio permite que vários processos colaborem na produção e envio das mensagens, assim como na receção, e processamento dos dados recebidos.

Formato da Mensagem



Além do conteúdo, as mensagens contém um cabeçalho com informação que permite fazer chegar a mensagem ao destinatário sem erros.
Na figura apresenta-se um exemplo da informação que pode existir no cabeçalho de uma mensagem.



Neste slide apresenta-se a referência utilizada para a construção destes slides.