BandTec

DIGITAL SCHOOL

BandTec
DIGITAL SCHOOL

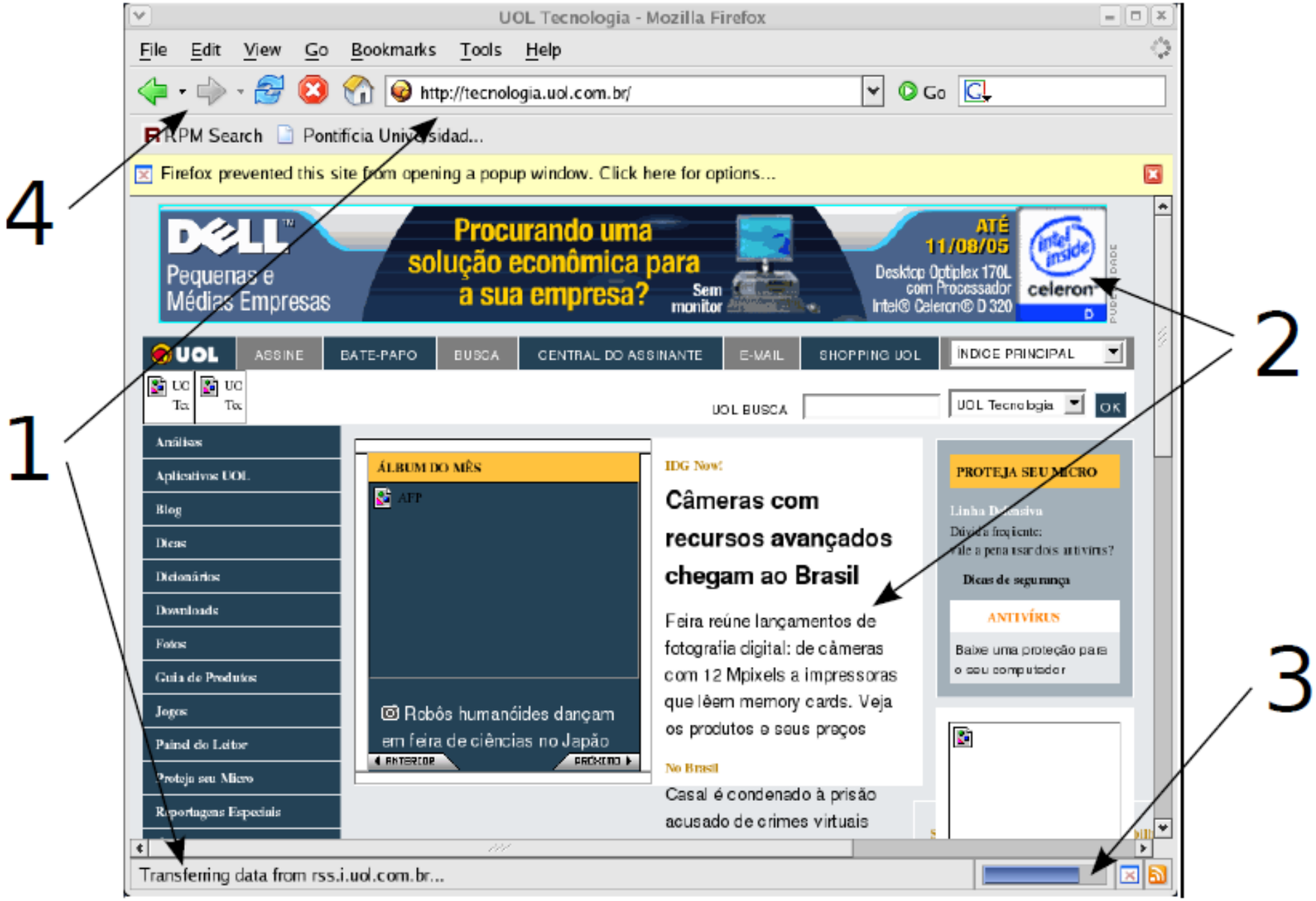
SO
Sistemas Operacionais

Aula 7 – Threads / Exclusão Mútua

Profa. Célia Taniwaki

Threads

- Tarefas relacionadas a um único aplicativo
- Exemplo: navegador 4 threads:
  - Download
  - Exibição
  - Barra de status
  - Ícones



Analogia

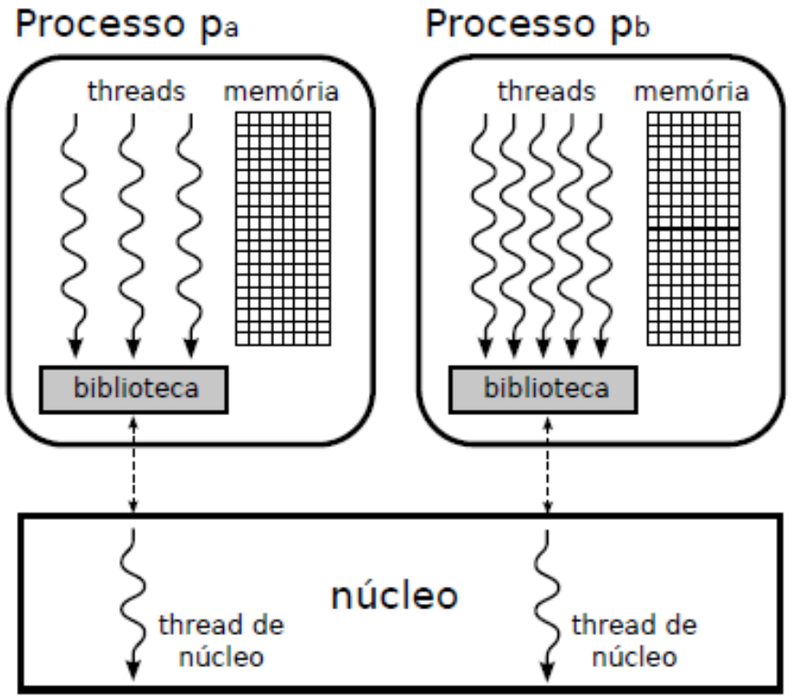
- Programa:
  - Receita do bolo
- Processo:
  - Preparo do bolo
- Threads (tarefas de um processo):
  - Preparar a massa do bolo
  - Preparar o recheio do bolo
  - Preparar a cobertura do bolo

Modelos de Threads

- Modelo de threads N:1
- Modelo de threads 1:1
- Modelo de threads N:M

Modelo de threads N:1

- Sistema Operacional não oferece suporte a threads
- Implementação dos threads: biblioteca no modo usuário (Ex: GNU Portable Threads)
- Os vários threads de um processo correspondem a um thread do núcleo

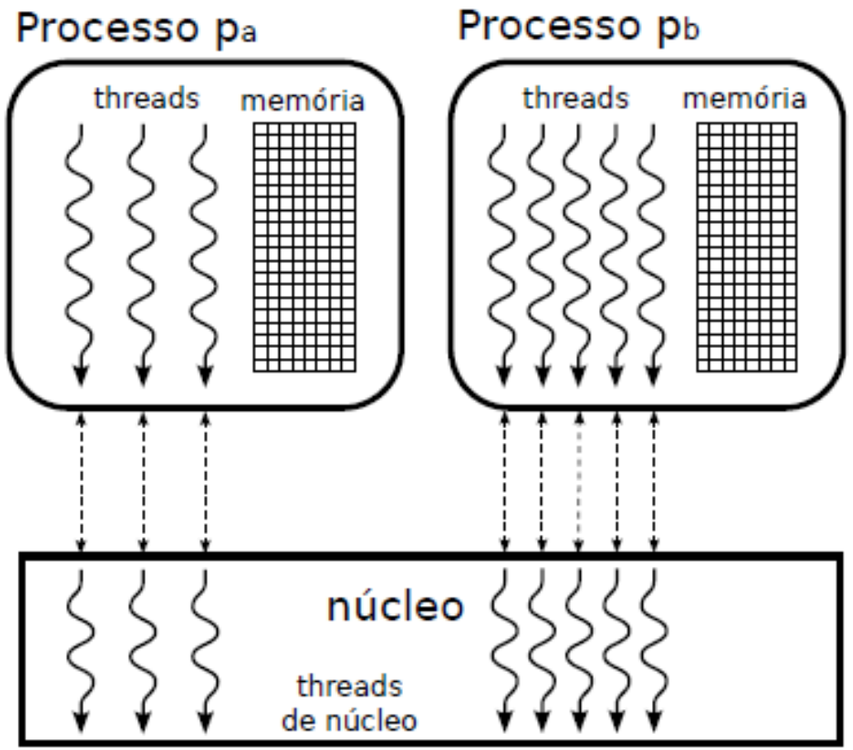


Modelo de threads N:1

- Vantagens:
  - Gerência por parte do núcleo é pequena
  - Escalabilidade (aumento dos threads do usuário) não é problema
- Desvantagens:
  - Quando um thread de um processo solicita operação de entrada/saída: bloqueia todos os demais threads do mesmo processo
  - Divisão injusta dos recursos entre as tarefas

Modelo de threads 1:1

- Sistema Operacional oferece suporte a threads
- Não há necessidade de biblioteca de threads no modo usuário
- Cada thread do usuário corresponde a um thread do núcleo
- Ex: Windows, Unix, Linux



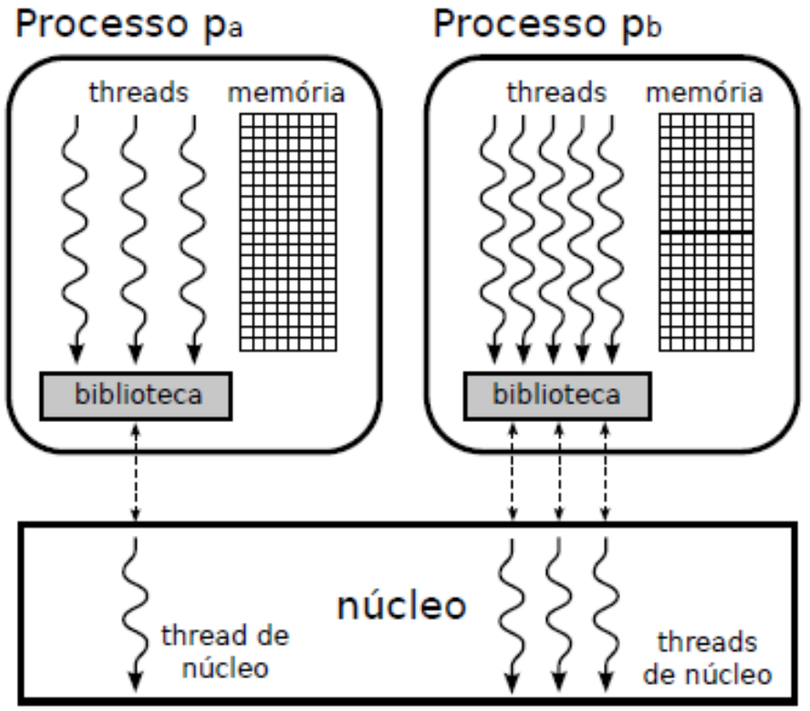
Modelo de threads 1:1

- Vantagens:
  - Quando um thread de um processo solicita operação de entrada/saída: não bloqueia todos os demais threads do mesmo processo
  - Divisão justa dos recursos entre as tarefas
  - Mais de um processador pode executar threads diferentes de uma mesma aplicação
- Desvantagens:
  - Gerência por parte do núcleo é maior do o modelo anterior
  - Escalabilidade (aumento dos threads do usuário) é problema

Modelo de threads N:M

- Sistema Operacional oferece suporte a threads
- Mais flexível do que o modelo 1:1
- Necessidade de biblioteca de threads no modo usuário
- N threads do usuário correspondem a M threads do núcleo ($1 \leq M \leq N$)
- Exemplos:

  Solaris



Modelo de threads N:M

- Vantagens:
  - Quando um thread de um processo solicita operação de entrada/saída: não bloqueia todos os demais threads do mesmo processo
  - Mais de um processador pode executar threads diferentes de uma mesma aplicação
  - Escalabilidade é possível
- Desvantagens:
  - Complexidade maior de gerência por parte do núcleo

Quadro comparativo dos modelos de threads

Modelo	N:1	1:1	N:M
Resumo	Todos os N *threads* do processo são mapeados em um único *thread* de núcleo	Cada *thread* do processo tem um *thread* correspondente no núcleo	Os N *threads* do processo são mapeados em um conjunto de M *threads* de núcleo
Local da implementação	bibliotecas no nível usuário	dentro do núcleo	em ambos
Complexidade	baixa	média	alta
Custo de gerência para o núcleo	nulo	médio	alto
Escalabilidade	alta	baixa	alta
Suporte a vários processadores	não	sim	sim
Velocidade das trocas de contexto entre *threads*	rápida	lenta	rápida entre *threads* no mesmo processo, lenta entre *threads* de processos distintos
Divisão de recursos entre tarefas	injusta	justa	variável, pois o mapeamento *thread*→ processador é dinâmico
Exemplos	GNU Portable Threads	Windows XP, Linux	Solaris, FreeBSD KSE

Sincronismo ou Coordenação entre Processos

- Como há mais de um processo ou tarefa sendo executado concorrentemente em um sistema,
  - Os processos podem estar acessando os mesmos recursos de forma concorrente (ao mesmo tempo)
- O Sistema Operacional deve garantir que:
  - Um processo possa passar informação para o outro
  - Um processo não invada o espaço do outro
  - Os processos sejam executados numa ordem adequada, para que não ocorram inconsistências:
    - Problema da condição de disputa ou de corrida
    - Problema em que um processo depende do outro (ex: produtor e consumidor)

Condições de corrida (Race Conditions) ou Condições de disputa

- Race Conditions: traduzido como condições de corrida ou condições de disputa
  - Inconsistências geradas por acessos concorrentes a dados compartilhados (pelos processos que competem pelos mesmos recursos de forma concorrente)
  - Recursos: memória, arquivos, impressora compartilhados
  - Existem somente caso uma das operações envolvidas seja de escrita (acessos concorrentes apenas de leitura não geram condições de corrida)
  - São erros dinâmicos
    - Não se manifestam no código fonte
    - Manifestam-se durante a execução, mas não sempre, dependendo do momento em que ocorrem os chaveamentos entre os processos

Condições de corrida (Race Conditions) ou Condições de disputa

- Exemplo: arquivo compartilhado
- Seja o seguinte trecho de programa:

```
Conta(      )
{
 ...
 read (arq_conta, &reg_cliente); /* lê registro do cliente do arquivo */
                                 /* reg_cliente é um struct na memória */
 scanf("%f", &valor);            /* lê valor a ser depositado ou retirado */
 reg_cliente.saldo = reg_cliente.saldo + valor;
                                /* atualiza o saldo do cliente  na memória */
 write (arq_conta, reg_cliente); /* grava registro no arquivo */
 ...
}
```

Condições de corrida (Ex: arquivo compartilhado)

- Esse trecho de programa pode ser executado num servidor que atende vários caixas bancários de forma concorrente.
- Imagine que 2 caixas estejam movimentando a mesma conta de um cliente.
  - No caixa 1, será feito um saque de 200
  - No caixa 2, será feito um depósito de 300
- Supondo que o saldo dessa conta seja inicialmente 1000:
  - Após o saque, ela teria 800,
  - E após o depósito, ela teria 1100.

Condições de corrida (Ex: arquivo compartilhado)

- Mas, no servidor haverá dois processos (ou threads) Conta sendo executados, cada um atendendo um dos caixas.
- Imagine o que aconteceria se o chaveamento entre os processos acontecesse justamente antes da instrução write (veja a simulação):

Caixa	Instrução	Saldo no arquivo	Valor	Saldo na memória
1	read (arq_cliente, &reg_cliente);	1000	*	1000
1	scanf (“%f”, &valor);	1000	-200	1000
1	reg_cliente.saldo=reg_cliente.saldo + valor;	1000	-200	800
2	read (arq_cliente, &reg_cliente);	1000	*	1000
2	scanf (“%f”, &valor);	1000	+300	1000
2	reg_cliente.saldo=reg_cliente.saldo + valor;	1000	+300	1300
1	write (arq_cliente, reg_cliente);	800	-200	800
2	write (arq_cliente, reg_cliente);	1300	+300	1300

Condições de corrida (Ex: arquivo e memória compartilhada)

- Dessa forma, se o chaveamento entre os processos ocorrer justamente antes da instrução write, a execução desses processos gerará uma inconsistência:
  - O saldo final da conta é 1300, sendo que o correto seria 1100.

- Outro exemplo: memória compartilhada
- Imagine 2 processos A e B, que estejam compartilhando o acesso a uma variável X.
- O processo A incrementa a variável X:
  - X = X + 1;
- O processo B decrementa a variável X:
  - X = X - 1;

Condições de corrida (Ex: memória compartilhada)

- Em Assembly, o código correspondente a essas instruções seriam:
- Processo A :

```
LOAD Ra, X
ADD Ra, 1
STORE X, Ra
```

Processo B:

```
LOAD Rb, X
SUB Rb, 1
STORE X, Rb
```

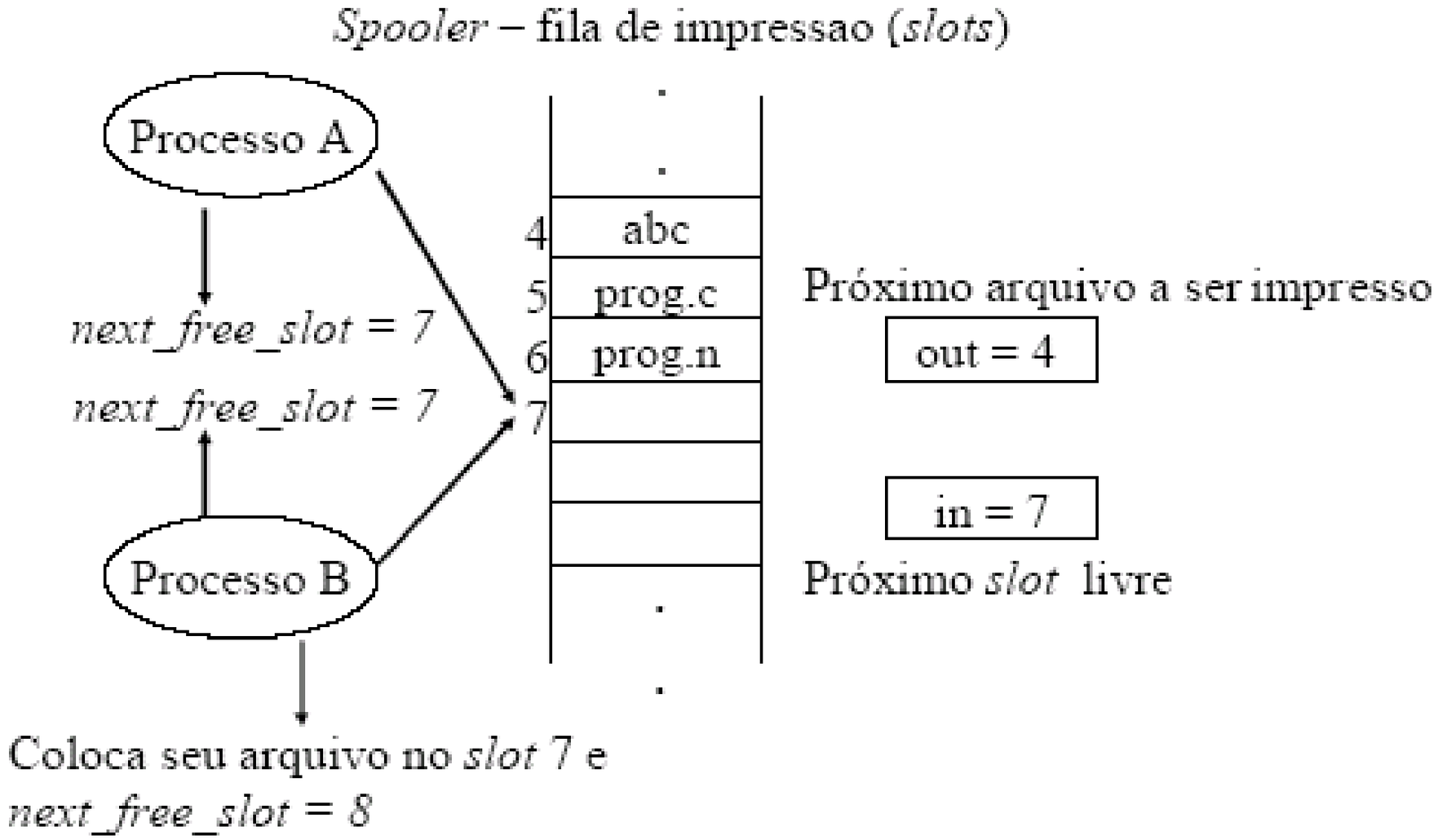
- Supondo que inicialmente a variável X tenha o valor 2, e que o chaveamento entre os processos ocorra justamente antes da instrução STORE

Processo	Instrução	X	Ra	Rb
A	LOAD Ra, X	2	2	*
A	ADD Ra, 1	2	3	*
B	LOAD Rb, X	2	3	2
B	SUB Rb, 1	2	3	1
A	STORE X, Ra	3	3	1
B	STORE X, Rb	1	3	1

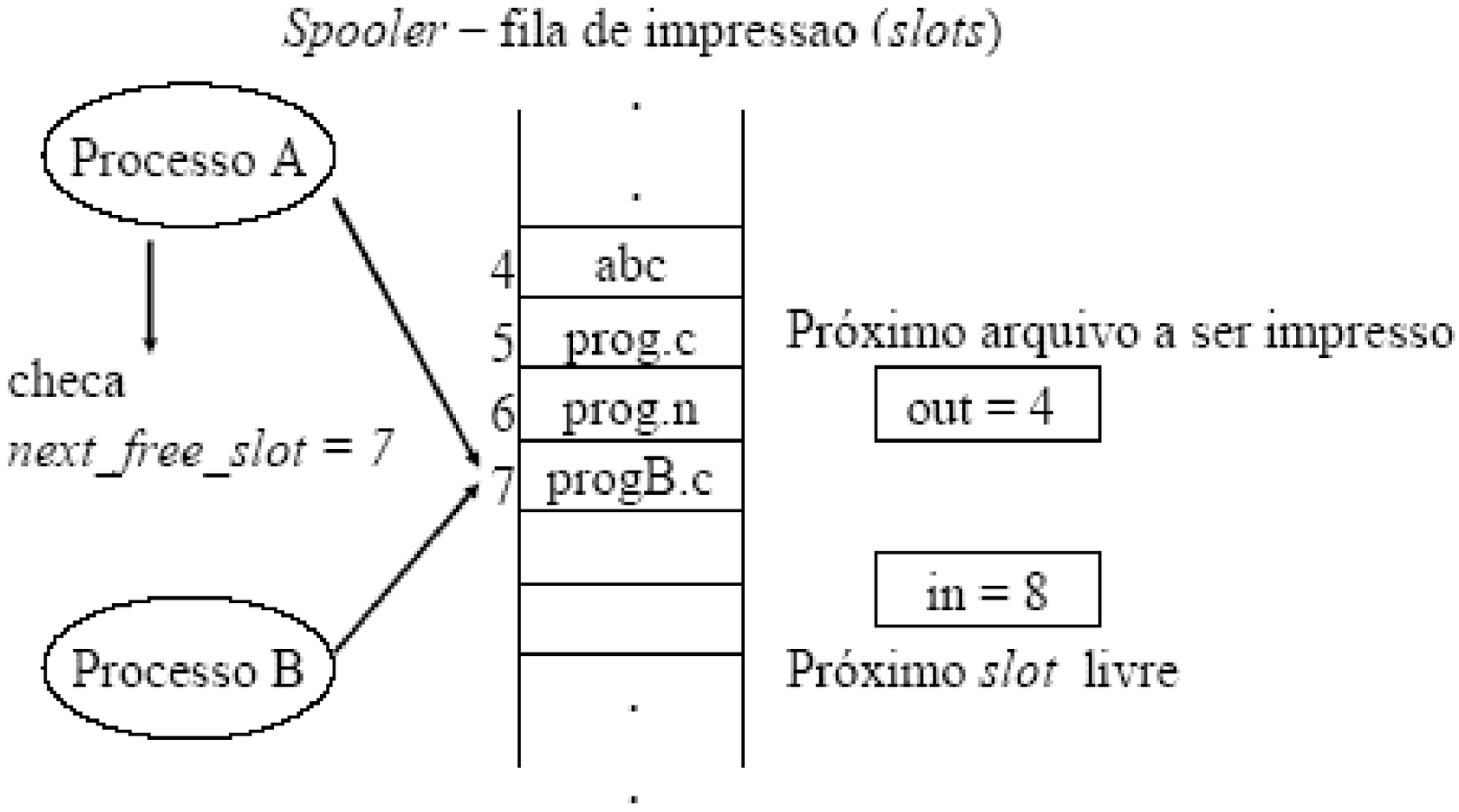
Condições de corrida (Ex: memória e impressora compartilhada)

- Dessa forma, se o chaveamento entre os processos ocorrer justamente antes da instrução STORE, a execução desses processos gerará uma inconsistência:
  - O valor final da variável X é 1, sendo que o correto seria 2.
- Outro exemplo: impressão
  - Processo que deseja imprimir um arquivo, coloca arquivo em uma área chamada spooler.
  - Outro processo (printer spooler) verifica se tem arquivo a ser impresso
    - Se tiver, imprime o arquivo e retira-o do spooler
  - Se 2 processos querem imprimir ao mesmo tempo:

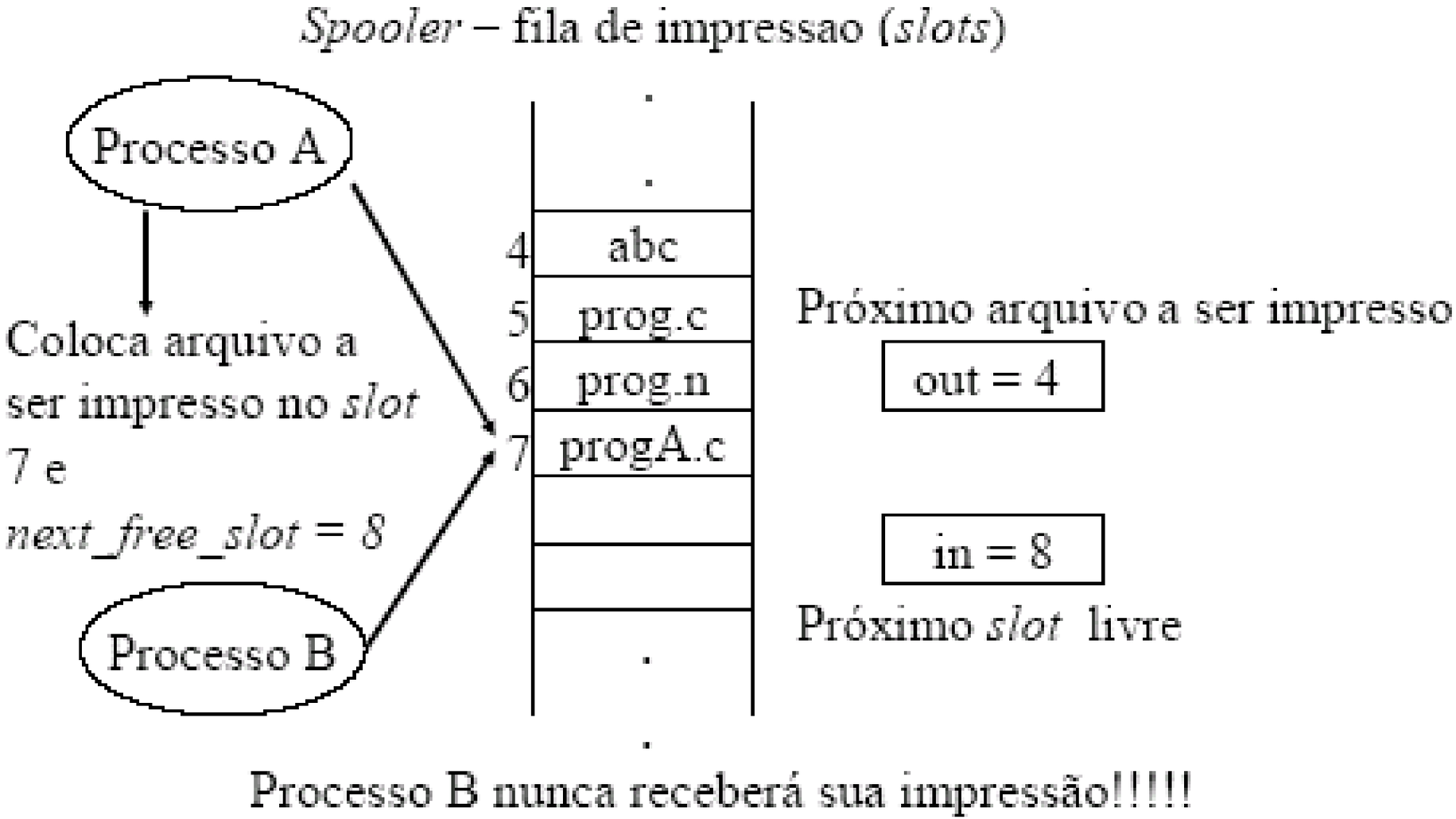
Condições de corrida (Ex: impressora compartilhada)



Condições de corrida (Ex: impressora compartilhada)



Condições de corrida (Ex: impressora compartilhada)

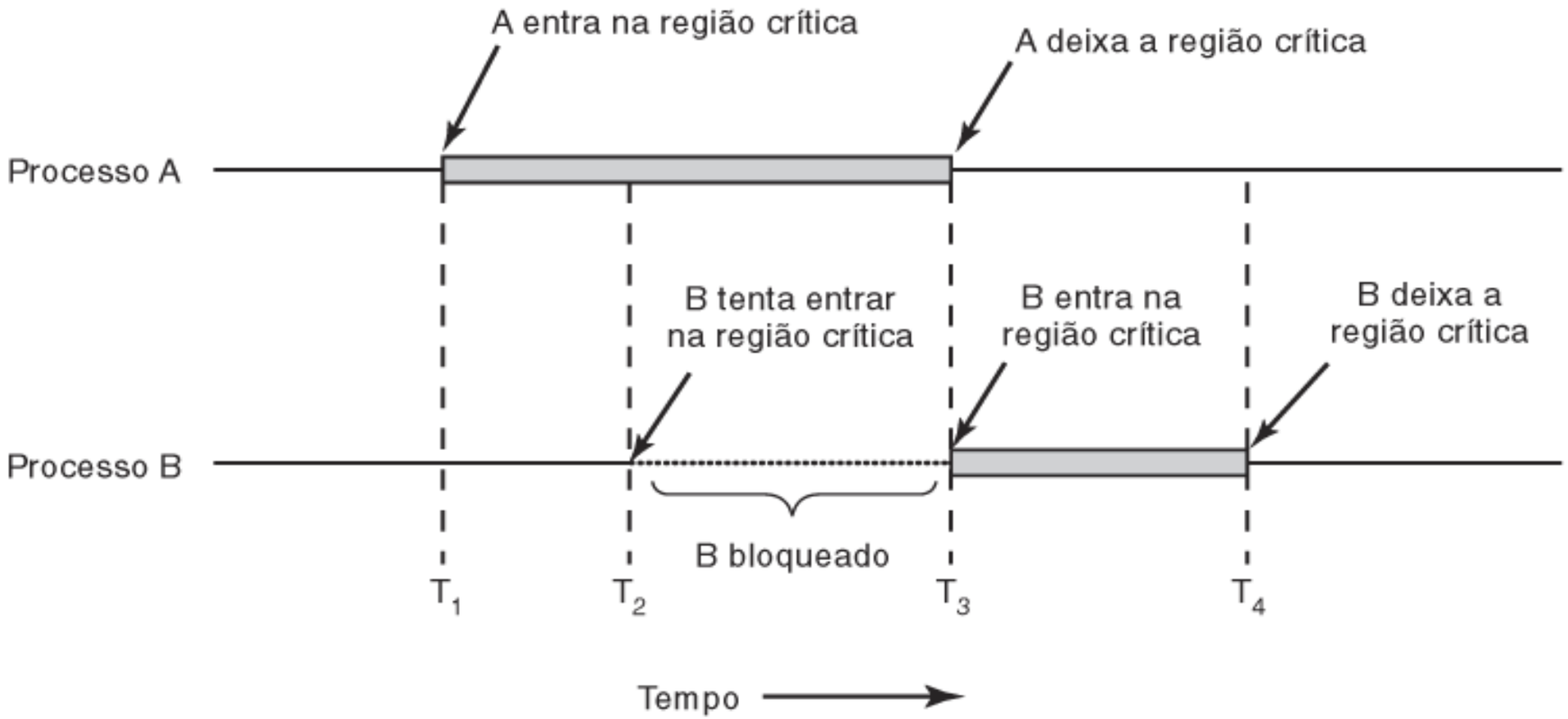


Região crítica (ou Seção crítica) Exclusão Mútua

- Como prevenir os problemas de Condição de Disputa?
  - Identificar a Região Crítica ou Seção Crítica
    - Trecho do programa que acessa e altera o valor do recurso compartilhado
  - Não permitir que mais de um processo execute sua região crítica relacionada ao mesmo recurso ao mesmo tempo.
- Exclusão mútua:
  - Mecanismo ou técnica que garante que apenas um processo ou tarefa execute sua região crítica num determinado instante.

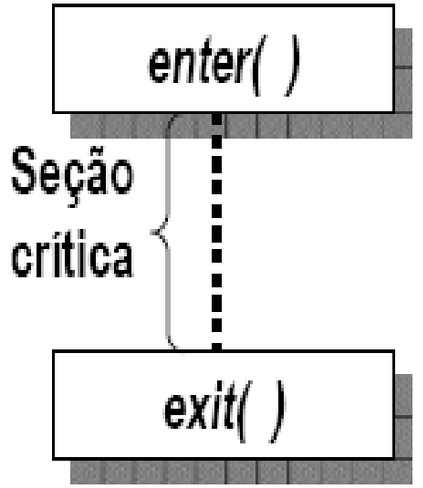
Exclusão Mútua

- Enquanto um processo está executando sua região crítica, outro processo que deseja entrar na região crítica fica bloqueado até que o primeiro deixe a região crítica.



Implementação da Exclusão Mútua

- Implementar a Exclusão Mútua envolve:
  - Identificar a seção crítica de cada processo
  - Implementar as funções:
    - enter( ), chamada antes da seção crítica
      - Posso executar a seção crítica?
      - Sim, sinaliza que está dentro da seção crítica e a executa
      - Não, então aguarda
    - exit( ), chamada após a seção crítica
      - Sinaliza que está deixando a seção crítica, liberando outros processos que queriam executar a seção crítica.



Exclusão Mútua
Condições para uma boa solução

- Boa solução deve satisfazer essas 4 condições:
  1. Exclusão mútua: Dois processos não podem estar simultaneamente em suas regiões críticas.
  2. Nenhuma restrição deve ser imposta com relação à CPU.
  3. Processos que não estão em regiões críticas não podem impedir os processos de entrarem em suas regiões críticas.
  4. Processos não podem esperar indefinidamente para entrarem em suas regiões críticas.

Soluções de Exclusão Mútua

- Soluções para Exclusão Mútua:
  - **Inibição de interrupções**
  - Espera ocupada
  - Primitivas Sleep/Wakeup
  - Semáforos
  - Monitores
  - Passagem de mensagem

Inibição de interrupções

- Processo desabilita as interrupções ao entrar na região crítica e habilita essas interrupções ao sair da região crítica.
- Com as interrupções desabilitadas, a CPU não realiza o chaveamento entre os processos (garante que somente um processo esteja na região crítica), mas:
  - Não satisfaz a condição 2
- Não é uma solução segura, pois um processo pode não habilitar novamente suas interrupções
  - Não satisfaz a condição 4

Soluções de Exclusão Mútua

- Soluções para Exclusão Mútua:
  - Inibição de interrupções
  - **Espera ocupada**
  - Primitivas Sleep/Wakeup
  - Semáforos
  - Monitores
  - Passagem de mensagem

Espera Ocupada

- Espera ocupada (Busy Waiting)
  - Consiste em testar continuamente uma condição ou valor que indique se a seção crítica está livre ou ocupada

- Há várias implementações possíveis:
  - Variável de travamento (*Lock*)
  - Estrita alternância (Strict Alternation)
  - Solução de Peterson
  - Instrução TSL (Test and Set Lock)

Espera Ocupada – Variável de travamento

- Antes de entrar na seção crítica, processo deve consultar a variável *lock*:
  - Se *lock* = 0 (zero), nenhum processo está na seção crítica
  - Se *lock* = 1 (um), existe algum processo na seção crítica
- Implementação:

```
while (lock != 0) ;      /* loop enquanto lock diferente de zero */
lock = 1;                /* altera valor de lock para 1 */
```

Seção crítica

```
lock = 0;                /* altera valor de lock para 0 */
```

Espera Ocupada – Variável de travamento

- Problema: ocorre condição de disputa em relação à variável *lock*
  - Imagine que um processo A consulte o valor de *lock* que tem o valor 0 (então o processo A deve alterar o valor de *lock* para 1 e entrar na sua seção crítica)
  - Mas antes que o processo A altere o valor de *lock* para 1, suponha que ocorre o chaveamento, e o processo B seja escalonado
  - B também deseja entrar na sua seção crítica, consulta *lock* (que é zero), altera o valor de *lock* para 1 e entra na sua seção crítica. Antes que B saia da seção crítica, termina seu quantum
  - Então, o processo A é escalonado novamente, altera valor de *lock* para 1 e entra na sua seção crítica
  - Resultado: ambos os processos estão na seção crítica
    - Transgride a condição 1

Soluções de Espera Ocupada

- Todas as soluções apresentadas que utilizam espera ocupada:
  - Processos ficam num loop consultando um valor, até que possam utilizar a região crítica
  - Consome tempo de processamento da CPU
  - Podem ocorrer situações inesperadas, dependendo do momento em que ocorre o chaveamento entre os processos
- Solução melhor
  - Enquanto o processo não puder entrar na região crítica, o processo é suspenso (muda do estado executando para estado suspenso), não consumindo tempo de CPU

Soluções de Exclusão Mútua

- Soluções para Exclusão Mútua:
  - Inibição de interrupções
  - Espera ocupada
  - **Primitivas Sleep/Wakeup**
  - Semáforos
  - Monitores
  - Passagem de mensagem

Primitivas Sleep/Wakeup

- Solução para o problema do consumo da CPU pela espera ocupada
- Primitiva Sleep ("Dormir")
  - Chamada de sistema que suspende o processo que a chamou, até que outro processo o "acorde"
- Primitiva Wakeup ("Acordar")
  - Chamada de sistema que "acorda" um determinado processo
- Ambas possuem 2 parâmetros:
  - Processo sendo manipulado
  - Endereço de memória para realizar a correspondência entre uma primitiva Sleep e sua correspondente Wakeup

Soluções de Exclusão Mútua

- Soluções para Exclusão Mútua:
  - Inibição de interrupções
  - Espera ocupada
  - Primitivas Sleep/Wakeup
  - **Semáforos**
  - Monitores
  - Passagem de mensagem

Semáforos

- Idealizados por E. W. Dijkstra (1965)
- Variável inteira que armazena o número de sinais wakeup enviados
- Um semáforo pode ter valor 0 quando não há sinal armazenado ou um valor positivo correspondente ao número de sinais armazenados
  - Se Semáforo = 0, então não pode prosseguir
  - Se Semáforo > 0, então pode prosseguir
- Duas primitivas ou chamadas de sistema: *down* e *up*
- Originalmente: P (down) e V (up) em holandês

Semáforos – Down e Up

- Primitivas down e up ocorrem como ações atômicas:
  - Ação atômica – garante que nenhum outro processo terá acesso ao semáforo enquanto a operação não finalizar ou não for suspensa
- Down
  - Suspende o processo se semáforo = 0
  - Decrementa semáforo se semáforo > 0
- Up
  - Incrementa semáforo
  - (caso algum processo esteja suspenso nesse semáforo, ele é "acordado" para poder terminar de executar sua operação *down*)

Semáforo Mutex ou Binário

- Semáforo Mutex ou Binário:
  - Assume dois valores: 1 e 0 (zero)
  - Garante a exclusão mútua, não permitindo que os processos acessem a região crítica ao mesmo tempo
- Utilização para controle de exclusão mútua:

Down (Semáforo)

Se Semáforo = 0,
Então suspende o processo
Decrementa Semáforo

Seção crítica

Up (Semáforo)

Incrementa Semáforo
(Libera processo suspenso)

Bibliografia

- Esse material foi elaborado com base nos livros:
  - Sistemas Operacionais Modernos. Tanenbaum, Andrew. 3ed. Pearson.
  - Sistemas Operacionais: Conceitos e Mecanismos. Maziero, Carlos. Disponível em: http://dainf.ct.utfpr.edu.br/~maziero/doku.php/so:livro_de_sistemas_operacionais

Obrigada!

BandTec DIGITAL SCHOOL

Em caso de dúvidas, entre em contato com:
celia.taniwaki@bandtec.com.br