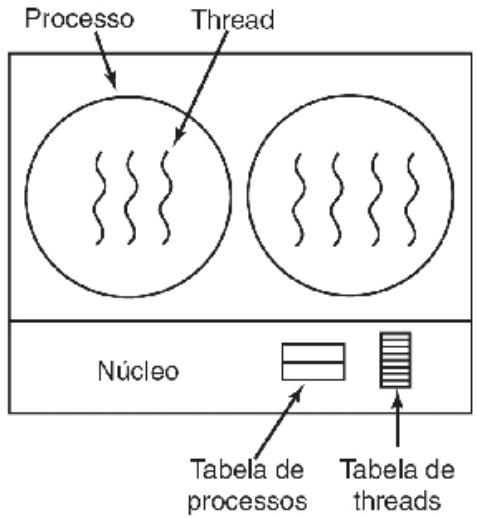
Aula 08 – Threads e Comunicação Interprocessos

Norton Trevisan Roman
Clodoaldo Aparecido de Moraes Lima

25 de setembro de 2014

Threads no Espaço do Núcleo

- Suportadas diretamente pelo SO
- Criação, escalonamento e gerenciamento são feitos pelo kernel
  - O núcleo possui tabela de threads (com todas as threads do sistema) e tabela de processos separadas
  - As tabelas de threads possuem as mesmas informações que as tabelas de threads em modo usuário, só que agora estão implementadas no kernel
  - Os algoritmos mais usados são Round Robin e Priority Scheduling

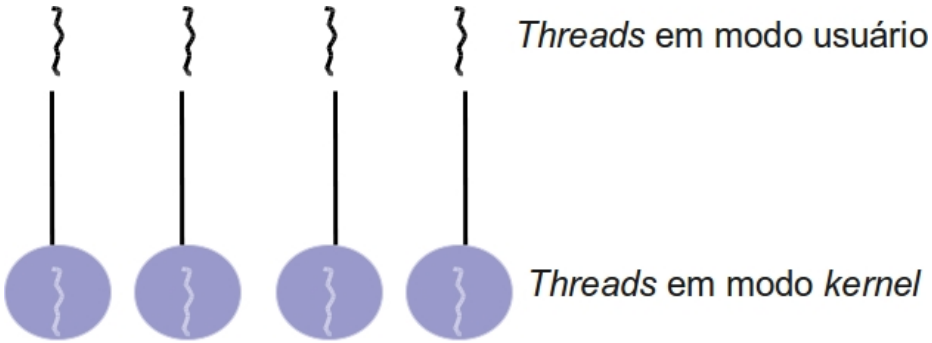


Threads no Espaço do Núcleo

- Quando uma thread deseja criar ou destruir outra, faz chamada ao núcleo
  - O núcleo atualiza a tabela de threads, dentre outras coisas
- Todas as chamadas que podem bloquear a thread são implementadas como chamadas ao sistema
  - Gerenciar threads em modo kernel é mais caro devido às chamadas de sistema durante a alternância entre modo usuário e modo kernel
  - Pode-se contudo, melhorar isso, dando preferência a threads no mesmo processo, caso sejam de igual precedência

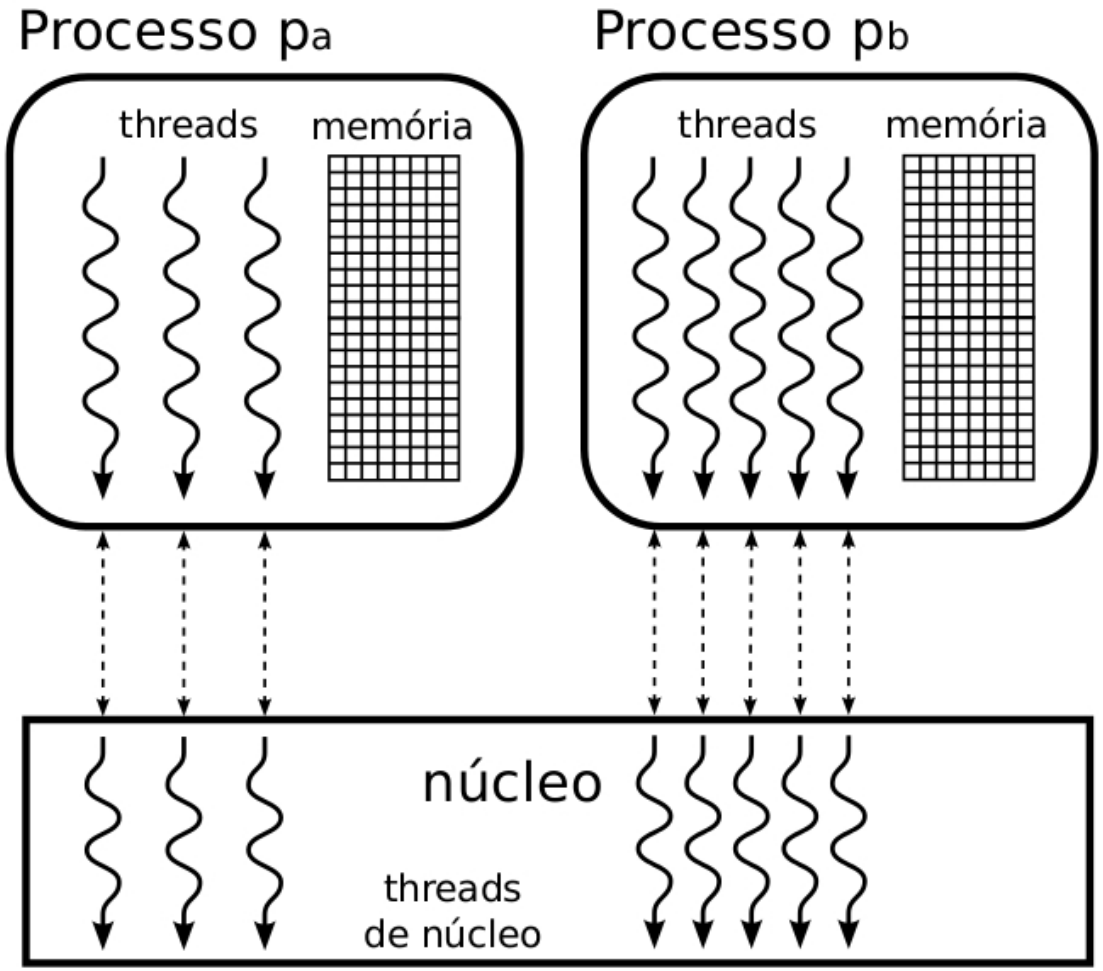
Threads no Espaço do Núcleo

- Seguem o modelo 1 para 1:
  - Chaveia para cada thread de usuário uma thread de kernel
  - Permite múltiplas threads em paralelo
  - Linux, Família Windows, OS/2, Solaris 9



Threads no Espaço do Núcleo

- 1 para 1:



Threads no Espaço do Núcleo

Reciclagem de threads

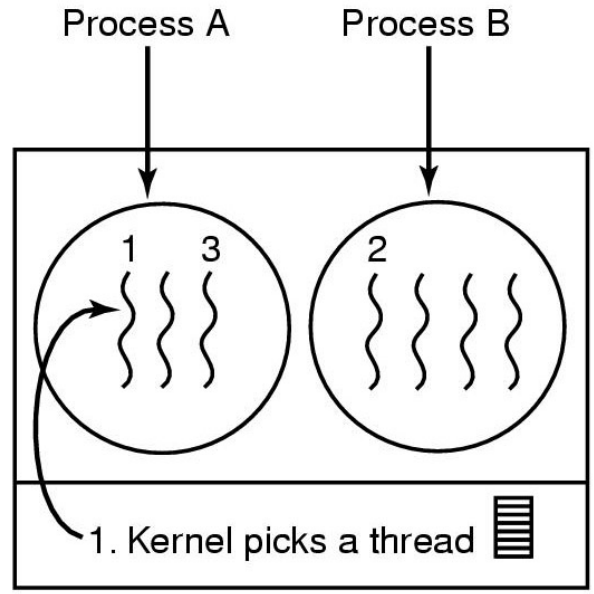
- Criar e destruir threads tem um custo alto
  - Solução: ao ser destruída, marca-se a thread como não executável, mantendo suas estruturas no núcleo
  - Ao criar uma nova thread, reativa a antiga, repopulando os dados, e economizando algum tempo de alocação
- Princípio também adotado em outras aplicações, como servidores web

Threads no Espaço do Núcleo

- Quando uma thread bloqueia, o kernel pode rodar outra thread do mesmo processo, ou uma de outro processo
  - No nivel do usuário, o sistema só consegue rodar threads do próprio processo, até que o kernel tire seu uso da CPU
- Vantagem:
  - Processo inteiro não é bloqueado se uma thread realizar uma chamada bloqueante ao sistema (ex: E/S)
- Desvantagem:
  - Exige chaveamento de contexto
    - Tarefa cara (pode inviabilizar grandes servidores Web e simulações de grande porte)

Threads no Espaço do Núcleo

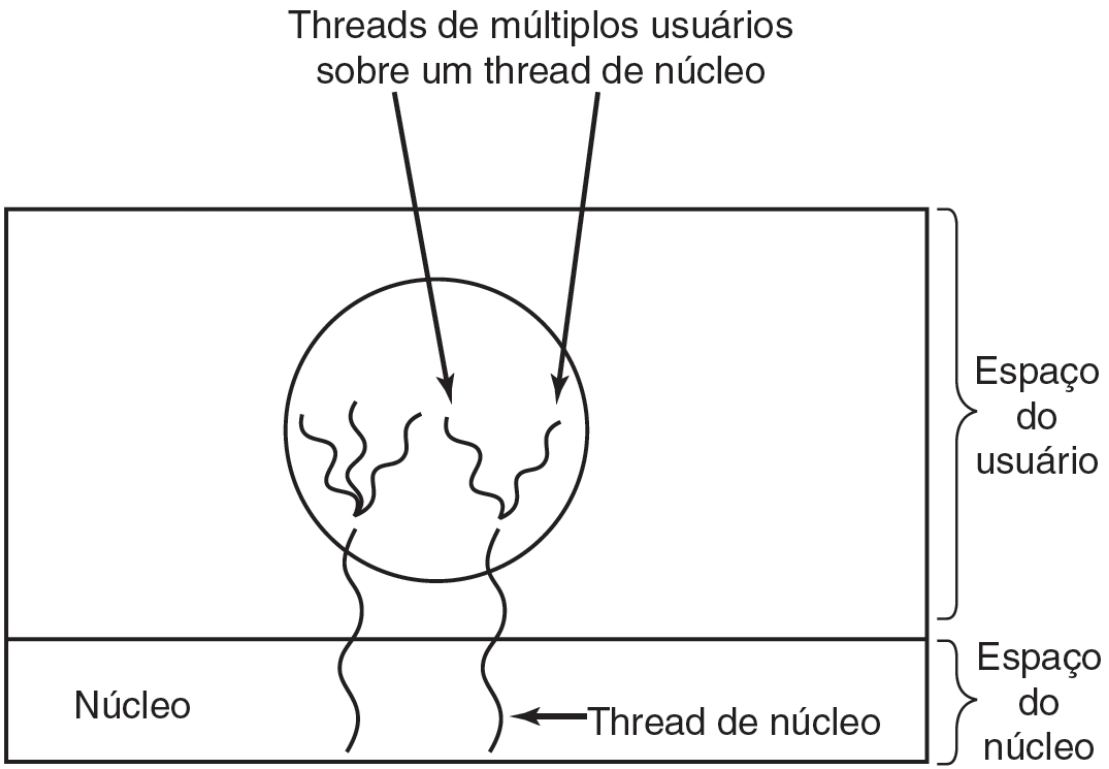
- Escalonamento:
  - O núcleo escolhe a thread diretamente
  - A thread é quem recebe o quantum, sendo suspensa se excedê-lo
  - Thread bloqueada por E/S não bloqueia o processo



Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3

Threads Híbridas

- É possível também fazer implementações híbridas
  - O núcleo sabe apenas sobre as threads de núcleo

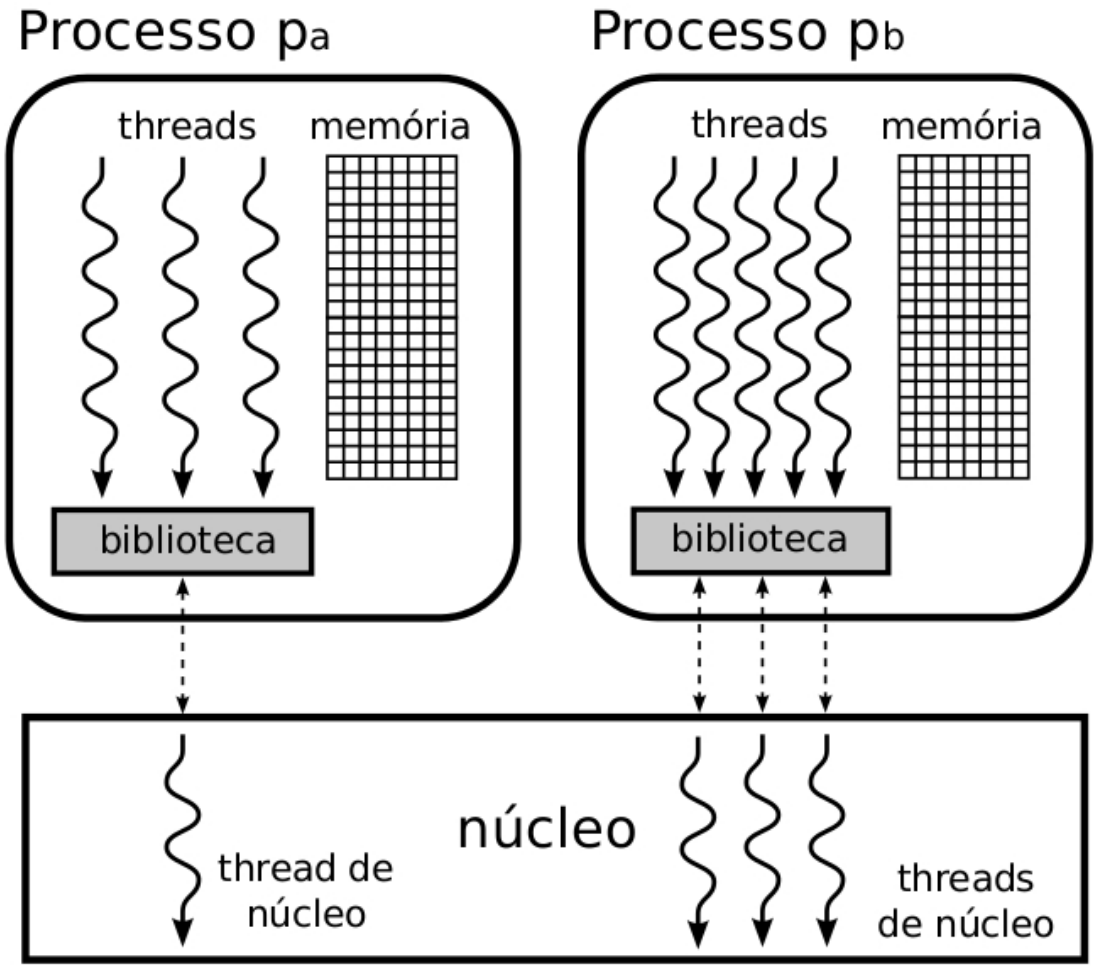


Threads Híbridas

- Seguem o modelo N para M:
  - N threads de usuário são mapeadas em M $\leq$ N threads de núcleo
  - Uma biblioteca gerencia um conjunto de threads de usuário (dentro do processo)
  - Esse processo é mapeado para uma ou mais threads do núcleo
    - No núcleo, há geralmente uma thread para cada tarefa bloqueada, e mais uma para cada processador disponível
    - Número de threads ajustado dinamicamente, conforme necessidade da aplicação
  - Solaris até versão 8, HP-UX, Tru64 Unix, IRIX

Threads Híbridas

- N para M:



Multithreading – Comparação de Modelos

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

Note que multithreading leva a 2 níveis de paralelismo: processos e threads

Comunicação Interprocessos

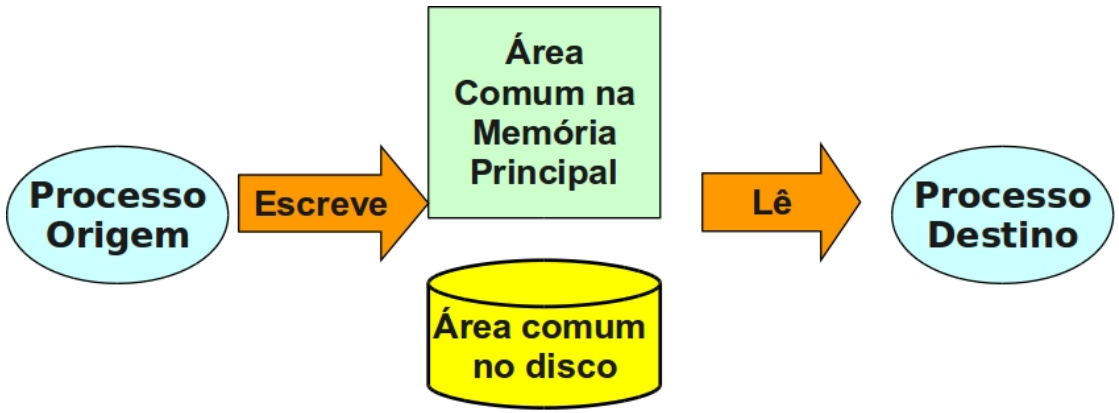
Comunicação Interprocessos

- Frequentemente processos precisam se comunicar
  - A comunicação entre processos é mais eficiente se for estruturada e não utilizar interrupções.
- Três aspectos importantes:
  - Como um processo passa informação para outro processo?
  - Como garantir que processos não invadam espaço uns dos outros, nem entrem em conflito?
  - Qual a sequência adequada quando existe dependência entre processos?

Comunicação Interprocessos

Condição de Corrida (Race Condition)

- Em alguns Sistemas Operacionais os processos se comunicam através de alguma área de armazenamento comum.
- Esta área pode estar na memória principal (ex: variáveis em comum a 2 threads) ou pode ser um arquivo compartilhado.



Comunicação Interprocessos

Condição de Corrida (Race Condition)

- Também chamada "Condição de Disputa"
- Situação onde dois ou mais processos acessam recursos compartilhados concorrentemente
  - Há uma corrida pelo recurso
  - Ex: quando estão lendo ou escrevendo algum dado compartilhado e o resultado depende de quem processa no momento propício
- Depurar programas que contém condições de corrida não é fácil, pois não é possível prever quando o processo será suspenso.

Comunicação Interprocessos

Condição de Corrida (Race Condition)

- Um exemplo: Print Spooler
- Spooling
  - Originalmente:
    - *System Peripheral Operations OffLine* (Batch)
  - Hoje
    - *Simultaneous Peripheral Operation On Line*
    - Consiste em colocar jobs em uma área da memória ou de um disco onde um dispositivo pode acessá-los quando estiver preparado.
    - Possibilita que a leitura de processos seja feita diretamente do disco: assim que um termina, o sistema operacional aloca outro, direto do disco.

Comunicação Interprocessos

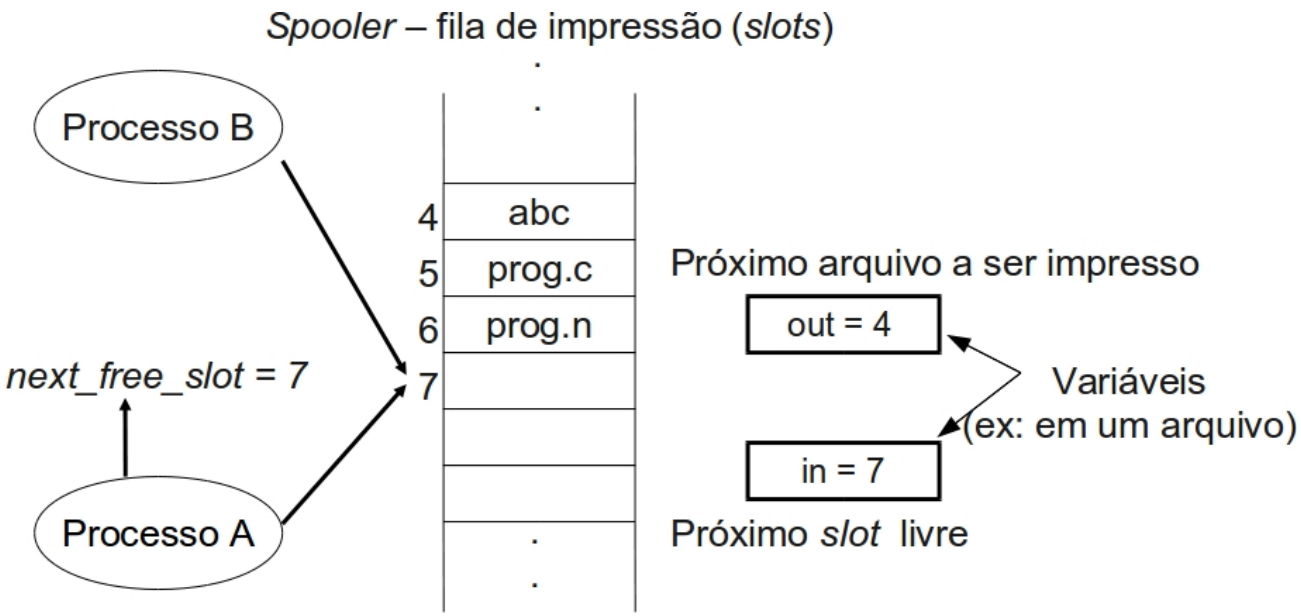
Condição de Corrida (Race Condition)

- Um exemplo: Print Spooler
  - Quando um processo deseja imprimir um arquivo, ele coloca o nome do arquivo em uma lista (diretório) de impressão (spooler directory)
  - Um processo chamado "printer daemon", verifica a lista periodicamente para ver se existe algum arquivo para ser impresso e, se existir, ele os imprime e remove seus nomes da lista.
  - Suponha que dois processos, A e B, irão enviar seus documentos para impressão

Comunicação Interprocessos

Condição de Corrida: Exemplo

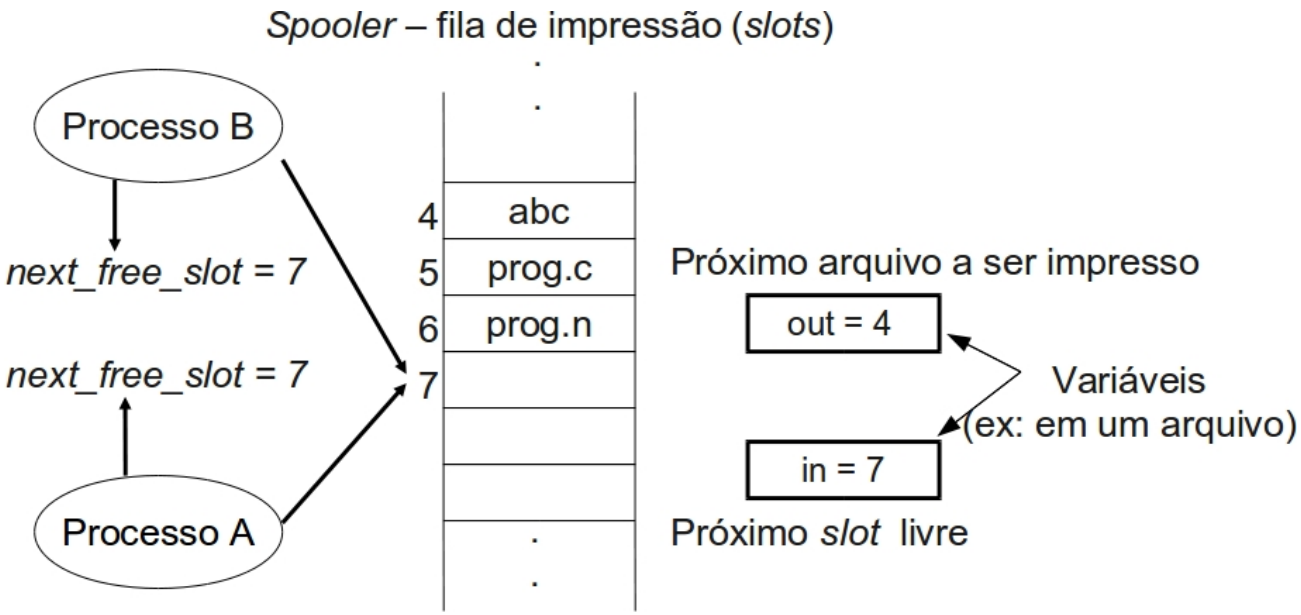
- A vê que 7 é a próxima posição livre, guardando-o em uma variável interna



Comunicação Interprocessos

Condição de Corrida (Race Condition)

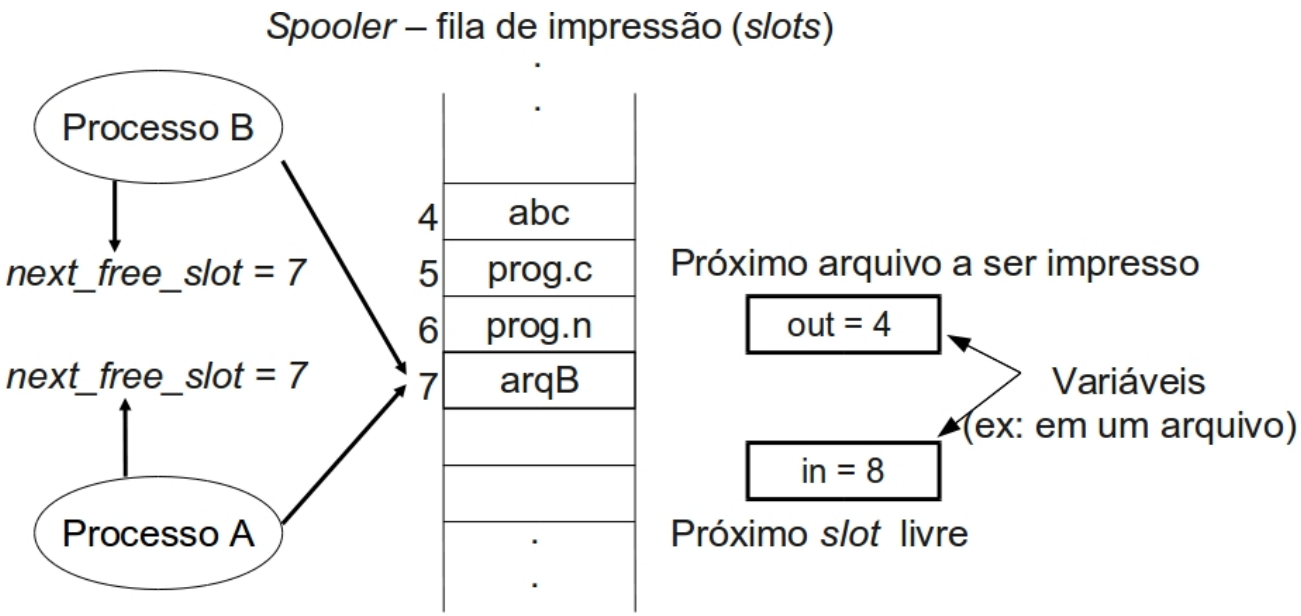
- O clock interrompe, e B é posto para rodar, que lê o mesmo valor para sua variável interna



Comunicação Interprocessos

Condição de Corrida (Race Condition)

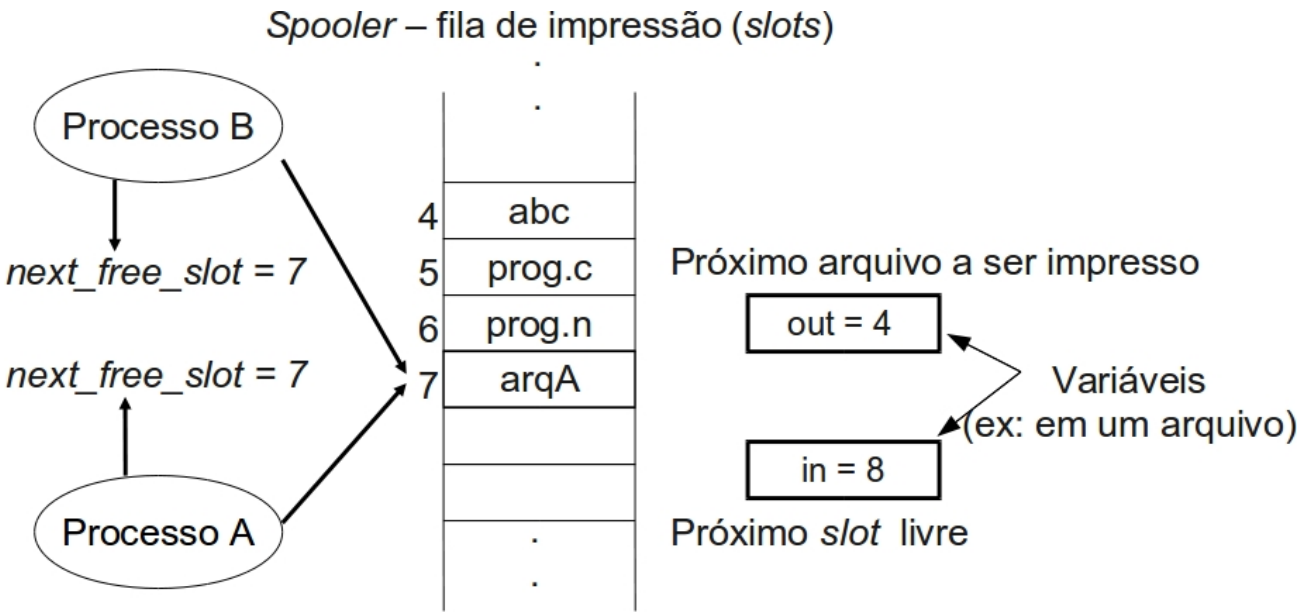
- B então armazena seu arquivo em 7, atualizando *in* (variável compartilhada) com 8



Comunicação Interprocessos

Condição de Corrida (Race Condition)

- A volta a executar, armazenando seu arquivo em 7, e atualiza *in* com 8



Comunicação Interprocessos

Condição de Corrida

- Ex: Vaga em avião
  - Operador OP1 (no Brasil) lê Poltrona1 vaga
  - Operador OP2 (no Japão) lê Poltrona1 vaga
  - Operador OP1 compra Poltrona1
  - Operador OP2 compra Poltrona1

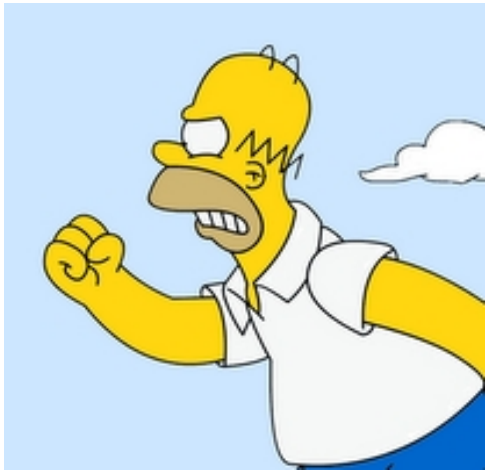
E o cliente?

Comunicação Interprocessos

Condição de Corrida

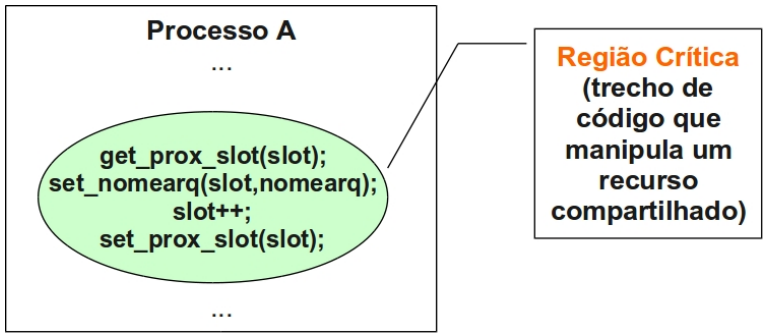
- Ex: Vaga em avião
  - Operador OP1 (no Brasil) lê Poltrona1 vaga
  - Operador OP2 (no Japão) lê Poltrona1 vaga
  - Operador OP1 compra Poltrona1
  - Operador OP2 compra Poltrona1

E o cliente?



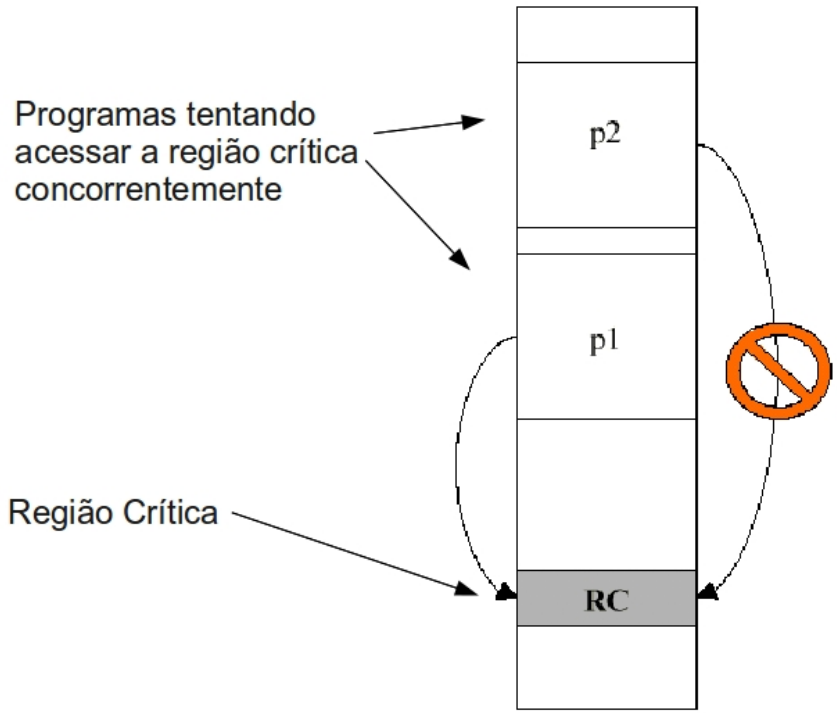
Exclusão Mútua e Região Crítica

- Uma solução para as condições de corrida é impedir que mais de um processo leia e escreva em uma variável compartilhada ao mesmo tempo
  - Esta restrição é conhecida como exclusão mútua
  - Seções do programa onde são efetuados acessos a recursos partilhados por dois ou mais processos são denominados regiões críticas (R.C.).

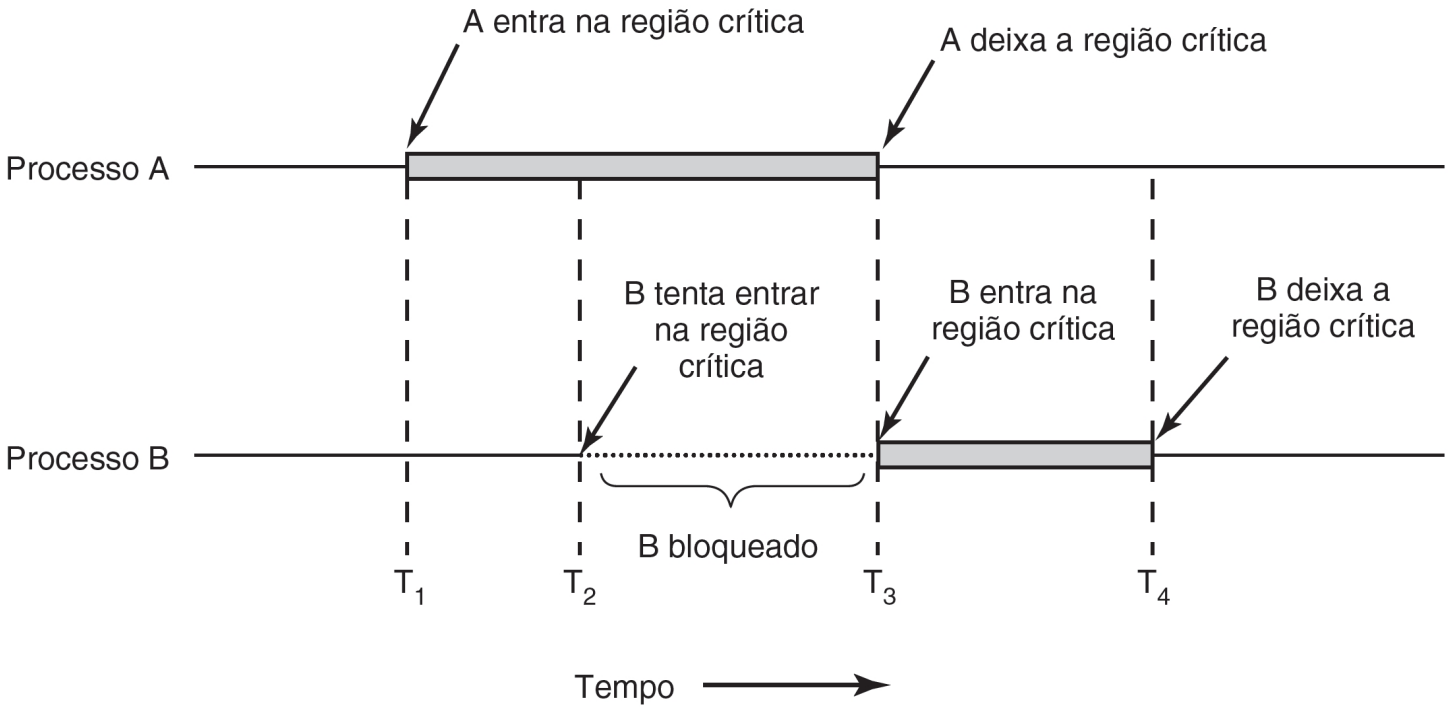


Exclusão Mútua e Região Crítica

- Regiões Críticas:
  - Exclusão mútua é então garantir que um processo não terá acesso a uma região crítica quando outro processo estiver utilizando essa região
  - Pode gerar uma fila de clientes nos computadores $\rightarrow$ ineficiência!



Exclusão Mútua e Região Crítica



Exclusão Mútua e Região Crítica

- Assegura-se a exclusão mútua recorrendo aos mecanismos de sincronização fornecidos pelo SO
  - Presentes também na JVM
- Estas afirmações são válidas também para as threads
  - Caso que é ainda mais crítico, pois todas as threads dentro do mesmo processo partilham os mesmos recursos.

Exclusão Mútua e Região Crítica

- Regras para programação concorrente (condições para uma boa solução)
  1. Dois processos nunca podem estar simultaneamente dentro de suas regiões críticas
  2. Não se pode fazer suposições em relação à velocidade e ao número de CPUs
  3. Um processo fora da região crítica não deve causar bloqueio a outro processo
  4. Um processo não pode esperar eternamente para entrar em sua região crítica

Soluções de Exclusão Mútua

- Espera Ocupada (Busy Waiting)
- Primitivas Sleep/Wakeup
- Semáforos
- Monitores
- Troca de Mensagem

Espera Ocupada

- Consiste geralmente na constante checagem de uma variável até que algum valor apareça
  - Gasta tempo de CPU
- Algumas soluções para Exclusão Mútua com Espera Ocupada:
  - Desabilitar interrupções
  - Variáveis de Travamento (Lock)
  - Estrita Alternância (Strict Alternation)
  - Solução de Peterson
  - Instrução TSL

Espera Ocupada – Desabilitar interrupções

- É a solução mais simples (com um único processador)
- Cada processo desabilita todas as interrupções (inclusive a do relógio) ao entrar na região crítica e as habilita novamente antes de deixá-la
  - Com as interrupções desabilitadas, a CPU não realiza chaveamento entre os processos
    - Pois a CPU é chaveada somente como resultado da interrupção do relógio ou de outra interrupção
    - Não haverá outro processo rodando na mesma hora
    - Viola condição 2 (Não se pode fazer suposições em relação à CPU – pressupõe processador de núcleo único)

Espera Ocupada – Desabilitar interrupções

- Em sistemas com várias CPUs, desabilitar interrupções em uma CPU não evita que as outras acessem a memória compartilhada
  - Viola condição 1 (Dois processos nunca podem estar simultaneamente dentro da região crítica)
- Não é uma solução segura, pois um processo pode esquecer de reabilitar suas interrupções e não ser finalizado
  - Viola condição 4 (Processo não podem esperar eternamente para entrar em sua região crítica)

Espera Ocupada – Desabilitar interrupções

- É útil, contudo, que o kernel tenha o poder de desabilitar interrupções, para sua própria manutenção:
  - Como quando atualiza a lista de processos prontos, caso em que uma interrupção poderia deixá-la em estado inconsistente
  - Ainda assim, com vários processadores não funciona bem
- Mas não é prudente que os processos de usuário usem este método de exclusão mútua

Espera Ocupada – Lock variables

O Problema de Espaço na Geladeira

Hora	Pessoa A	Pessoa B
6:00	Olha a geladeira: sem leite	-
6:05	Sai para a padaria	-
6:10	Chega na padaria	Olha a geladeira: sem leite
6:15	Sai da padaria	Sai para a padaria
6:20	Chega em casa: guarda o leite	Chega na padaria
6:25	-	Sai da padaria
6:30	-	Chega em casa: Ops!

Espera Ocupada – Lock variables

O Problema de Espaço na Geladeira

Hora	Pessoa A	Pessoa B
6:00	Olha a geladeira: sem leite	-
6:05	Sai para a padaria	-
6:10	Chega na padaria	Olha a geladeira: sem leite
6:15	Sai da padaria	Sai para a padaria
6:20	Chega em casa: guarda o leite	Chega na padaria
6:25	-	Sai da padaria
6:30	-	Chega em casa: Ops!

Regra	Exemplo da geladeira
1. Trancar antes de utilizar	Deixar aviso
2. Destrancar quando terminar	Retirar o aviso
3. Esperar se estiver trancado	Não sai para comprar se houver aviso

Espera Ocupada – Lock variables

- O processo que deseja utilizar uma região crítica atribui um valor a uma variável compartilhada – lock
  - Se a variável está com valor 0 (zero) significa que nenhum processo está na região crítica
  - Se a variável está com valor 1 (um) significa que existe um processo na região crítica
  - Para entrar em sua região crítica, o processo verifica o conteúdo da variável. Se for 0, muda para 1 e entra. Se for 1, espera até que seja 0.

Espera Ocupada – Lock variables

- O processo que deseja utilizar uma região crítica atribui um valor a uma variável compartilhada – lock
  - Se a variável está com valor 0 (zero) significa que nenhum processo está na região crítica
  - Se a variável está com valor 1 (um) significa que existe um processo na região crítica
  - Para entrar em sua região crítica, o processo verifica o conteúdo da variável. Se for 0, muda para 1 e entra. Se for 1, espera até que seja 0.

Espera Ocupada – Lock variables

- Problema:
  - Suponha que um processo A leia a variável lock com valor 0
  - Antes que o processo A possa alterar a variável para o valor 1, um processo B é escalonado e altera o valor de lock para 1
  - Quando o processo A for escalonado novamente, ele altera o valor de lock para 1, e ambos os processos estão na região crítica
    - Viola condição 1 (Dois processos não podem estar simultaneamente na região crítica)

Espera Ocupada – Lock variables

- Exemplo:

```
while(true){
     while(lock!=0); //loop
     lock=1;
     critical_region();
     lock=0;
      non-critical_region();
}
```

Processo A

```
while(true){
     while(lock!=0); //loop
     lock=1;
     critical_region();
     lock=0;
     non-critical_region();
}
```

Processo B

Espera Ocupada – Strict Alternation

- Possível solução: Strict Alternation
  - Fragmentos de programa controlam o acesso às regiões críticas
    - Variável turn, inicialmente em 0, estabelece qual processo pode entrar na região crítica

```
while(true){
     while(turn!=0); //loop
     critical_region();
     turn=1;
      non-critical_region();
}
```

Processo A

```
while(true){
     while(turn!=1); //loop
     critical_region();
     turn=0;
     non-critical_region();
}
```

Processo B

Espera Ocupada – Strict Alternation

- Ao deixar a região crítica, cada processo muda turn, para permitir que o outro possa entrar
- Problema:
  - Suponha que B termina seu trabalho nessa região rapidamente $\rightarrow$ ambos estão em regiões não críticas com turn=0
  - Agora o processo A entra na região crítica, saindo rapidamente, e fazendo turn=1 $\rightarrow$ ambos estão em regiões não críticas com turn=1

Espera Ocupada – Strict Alternation

- Problema:
  - A tenta entrar novamente na região crítica $\rightarrow$ não consegue, pois turn=1, e B está ocupado em sua região não crítica
  - A tem que esperar até que turn seja 0 $\rightarrow$ violação da condição 3 (processos fora da região crítica não devem bloquear outro processo)
- Moral: turnos não são uma boa idéia quando um dos processos é muito mais lento que o outro
  - Esta solução requer que 2 processos alternem sempre na região crítica
    - Um mesmo não pode entrar nela mais de uma vez seguida

Espera Ocupada – Solução de Peterson

- Solução de Peterson (1981)
  - Combina as idéias das variáveis de travamento com chaveamento obrigatório
  - Uma variável (ou programa) é utilizada para bloquear a entrada de um processo na região crítica quando um outro processo está na região
  - Essa variável é compartilhada pelos processos que concorrem pelo uso da região crítica
  - Ambos processos possuem fragmentos de programas que controlam a entrada e a saída da região crítica

Espera Ocupada – Solução de Peterson

- Antes de entrar na região crítica:
  - Cada processo chama entra_região, demonstrando seu interesse em entrar na região crítica
  - Somente quando entra_região retorna é que ele pode entrar na região crítica (espera até que seja seguro entrar na região)

```
#define FALSE 0
#define TRUE 1
#define N 2 /*Número de processos*/

int turn; /* de quem é a vez de esperar? */
int interessado[N]; /*inicialmente, FALSE para
                                              todos*/

/*processo é o número do processo que deseja entrar
na região: 0 ou 1*/
void entra_região(int processo) {
   int outro; /*número do outro processo*/

   outro = 1-processo; /*o outro processo*/
   interessado[processo] = TRUE;
   turn = processo;
   while (turn == processo && interessado[outro]);
                                          /*espera*/
}

void deixa_região(int processo) { /*processo:
                                    quem tá saindo*/
   interessado[processo] = FALSE;
}
```

Espera Ocupada – Solução de Peterson

- Após terminar na região crítica
  - Chama deixa_região, permitindo que outro processo entre
- entra_região e deixa_região são partes de uma biblioteca
  - turn e interessado são compartilhadas

```
#define FALSE 0
#define TRUE 1
#define N 2 /*Número de processos*/

int turn; /* de quem é a vez de esperar? */
int interessado[N]; /*inicialmente, FALSE para
                                          todos*/

/*processo é o número do processo que deseja entrar
na região: 0 ou 1*/
void entra_região(int processo) {
   int outro; /*número do outro processo*/

   outro = 1-processo; /*o outro processo*/
   interessado[processo] = TRUE;
   turn = processo;
   while (turn == processo && interessado[outro]);
                                        /*espera*/
}

void deixa_região(int processo) { /*processo:
                                    quem tá saindo*/
   interessado[processo] = FALSE;
}
```

Espera Ocupada – Solução de Peterson

- E se dois processos chamarem entra_região ao mesmo tempo?
  - Ambos armazenarão seus ids na variável turn
  - O último a modificá-la é quem a definirá → será quem irá esperar
  - Note que eles alteram regiões diferentes do vetor de interessados

```
/*processo é o número do processo que deseja entrar
na região: 0 ou 1*/
void entra_região(int processo) {
   int outro; /*número do outro processo*/

   outro = 1-processo; /*o outro processo*/
   interessado[processo] = TRUE;
   turn = processo;
   while (turn == processo && interessado[outro]);
                                          /*espera*/
}

void deixa_região(int processo) { /*processo:
                                    quem tá saindo*/
   interessado[processo] = FALSE;
}
```

Bastante complicado quando há vários processos

Referências Adicionais

- G. L. Peterson: "Myths About the Mutual Exclusion Problem", Information Processing Letters 12(3) 1981, 115–116