Universidade Federal de Uberlândia
Faculdade de Computação
Sistemas Operacionais

Concorrência
Parte 1

Prof. Dr. Marcelo Zanchetta do Nascimento

Roteiro

- Processos concorrentes
- Comunicação de processos
- Condição de corrida
- Exclusão mútua
- Sincronização condicional
- Leituras Sugeridas

Motivação

No gerenciamento de processos/threads tem-se:

- **Multiprogramação**: o gerenciamento de múltiplos processos em um único processador;
- **Multiprocessamento:** o gerenciamento de múltiplos processos dentro de multiprocessadores;
- **Processamento distribuído:** o gerenciamento de múltiplos processos executando em múltiplos sistemas computacionais distribuídos.

Motivação

- Os sistemas permitem estruturar as aplicações (software) para que **diferentes partes do código-fonte** possam executar de forma **concorrente**;

- Os processos/threads disputam recursos comuns:

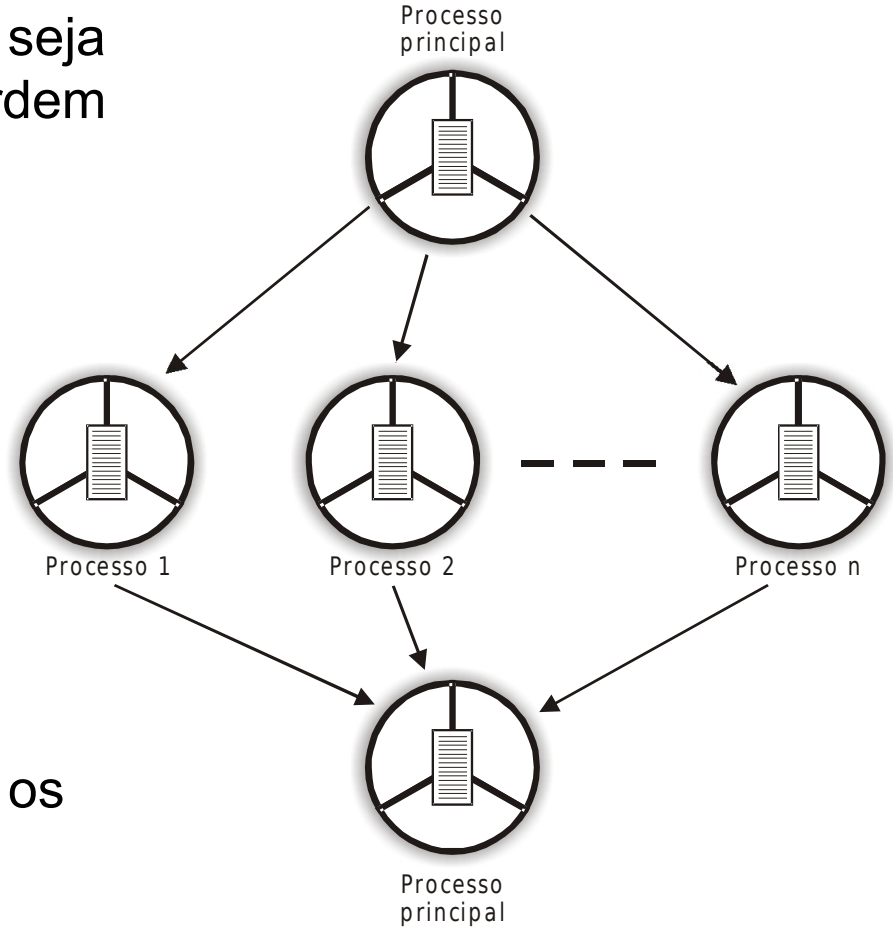
  – Variáveis, periféricos, registros, áreas de memória, etc.

- As execuções devem ser sincronizadas por um mecanismo oferecido pelo sistema operacional.

Especificação de Concorrência em Programas

Início: especifica que a sequência seja executada **concorrentemente** em ordem **imprevisível**;

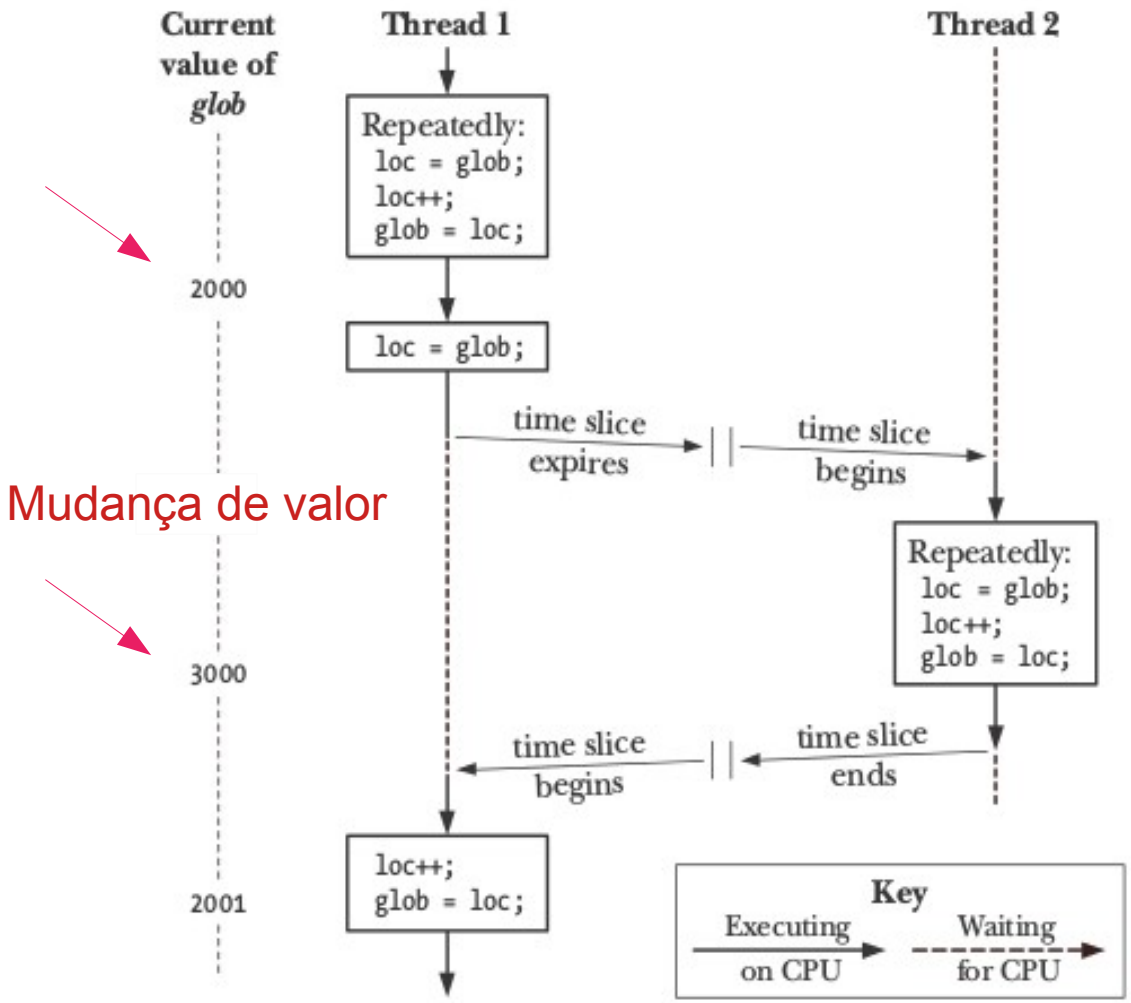
```
Chamada 1 <instrução>
Chamada 2
Chamada 3
...
```



Fim: ponto de sincronização, quando os processos ou threads terminam;

Figura 1: Concorrência entre processos em um software

Especificação de Concorrência em Programa



Vamos analisar a variável ***glob?***

Figura 2: Threads empregadas para incrementar a variável sem sincronização

Outro Programa

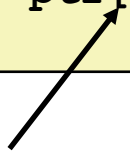
```
char **ptr;  /* global */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[2] = {
        "Olá de m1",
        "Olá de m2"
    };
    ptr = msgs;

    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```

```
/* função com thread */
void *thread(void *vargp)
{
    int myid = (int) vargp;
    static int cnt = 0;

    printf("[%d]: %s (shareVar=%d)\n",
        myid, ptr[myid], ++cnt);
}
```



Pares de referências dos threads
Indiretamente pela variável global

As variáveis compartilhadas

- Quais variáveis são compartilhadas

Variáveis	*Referência main?*	*Referência thread 0?*	*Referência thread 1?*
`ptr`	yes	yes	yes
`cnt`	no	yes	yes
`i`	yes	no	no
`msgs`	yes	yes	yes
`myid.p0`	no	yes	no
`myid.p1`	no	no	yes

- Uma variável é compartilhada se múltiplos threads referenciam pelo menos uma instância:
  - **`ptr`, `cnt` e `msgs` são compartilhadas**
  - **`i` e `myid` não são compartilhadas**

Processos/Threads concorrentes

- Aumentar o desempenho:
  - Permite explorar o paralelismo real disponível em máquinas multiprocessadas;
  - Sobreposição de operações de E/S com o processamento do recurso CPU;
- A programação concorrente implica em uma forma de compartilhamento de recursos:
  - Variáveis compartilhadas são recursos essenciais para a programação concorrente.
- Acesso aos recursos compartilhados devem ser feitos de forma a manter um estado coerente e correto do sistema.

Comunicação de Processos

Exemplo Clássico:

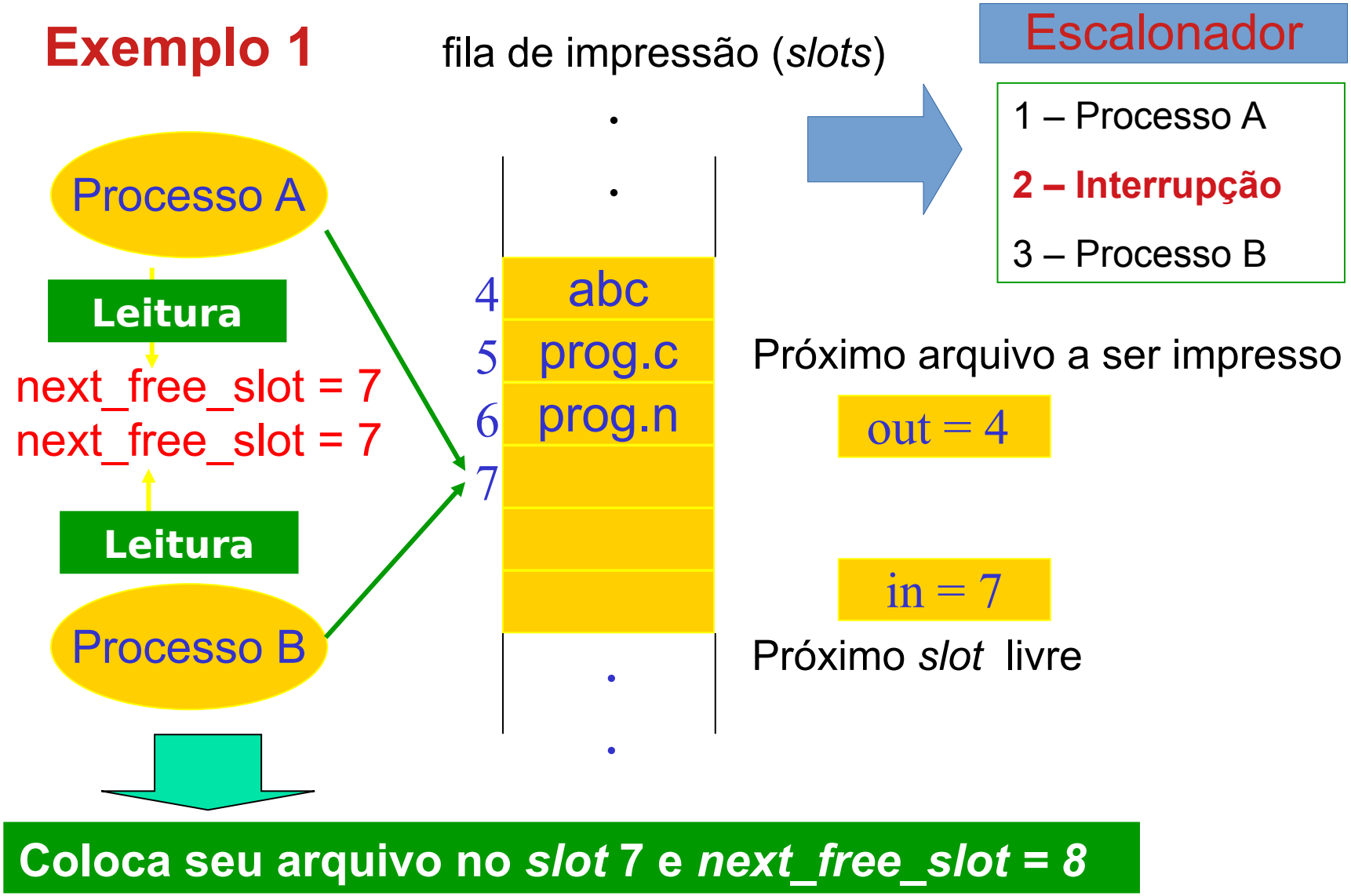
Problema do Produtor/Consumidor: Fila de impressão.

- Qualquer processo que queira imprimir precisa colocar o seu documento na **fila de impressão**;
- Um processo denominado "produtor" produz informações (dados), que são consumidas por um processo "consumidor";
- Para permitir a execução é preciso ter um buffer de itens, que seja preenchido pelo produtor e esvaziados pelo consumidor;
- Ocorre a concorrência nessa tarefa.

Comunicação de Processos

- Quando um processo deseja enviar um arquivo: em um local especial (denominado produtor);
- Um outro processo (denominado consumidor), checa se existe algum arquivo a ser impresso. Se existe, esse arquivo é retirado e impresso;
- Imagine se os dois processos desejarem, ao mesmo tempo, manipular um arquivo;
- **Condição de corrida:**
  - dois ou mais processos estão acessando dados compartilhados;
  - o resultado depende de quem executa primeiro.

Comunicação de Processos



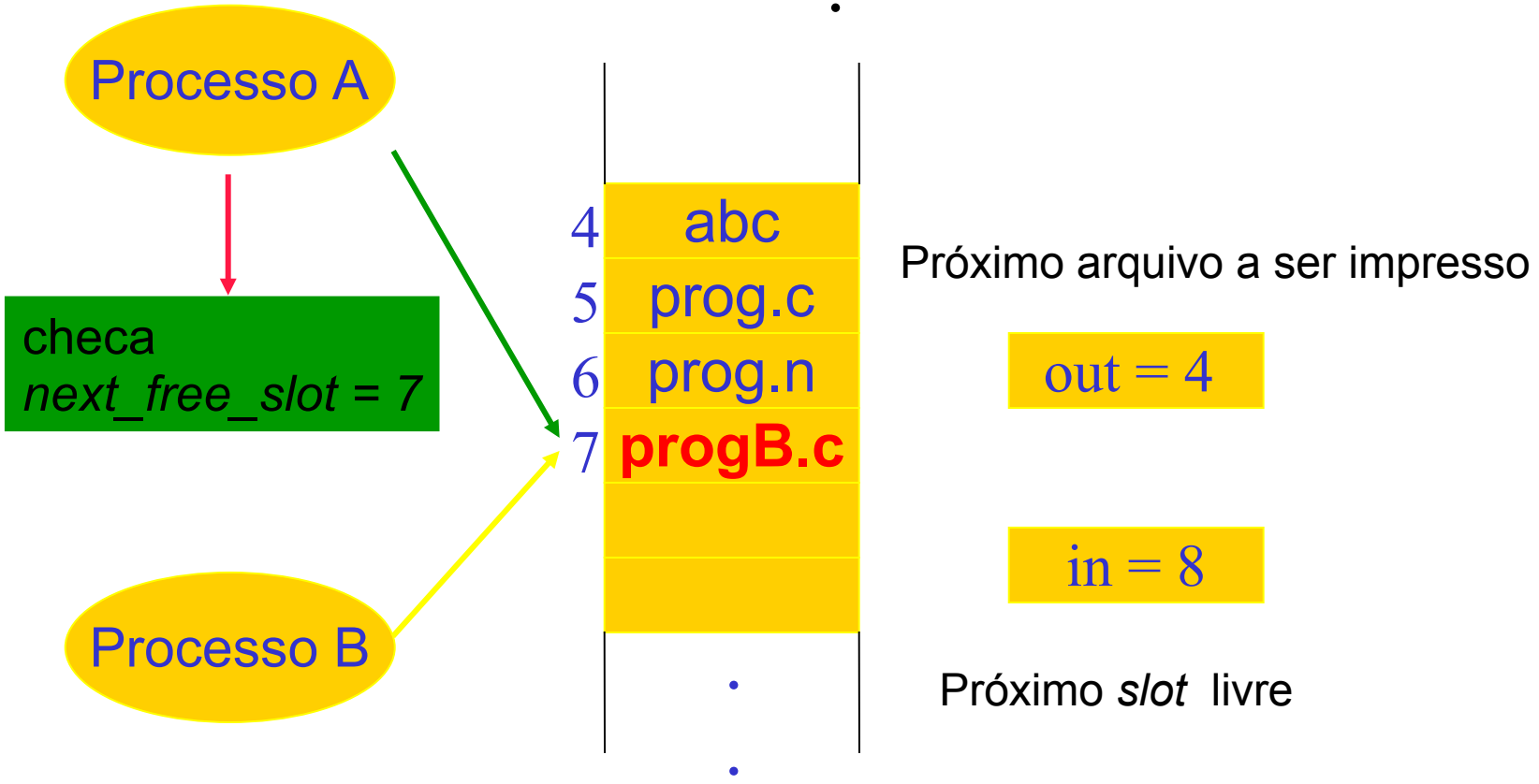
Comunicação de Processos

Exemplo 1

1 – Verifica next_free

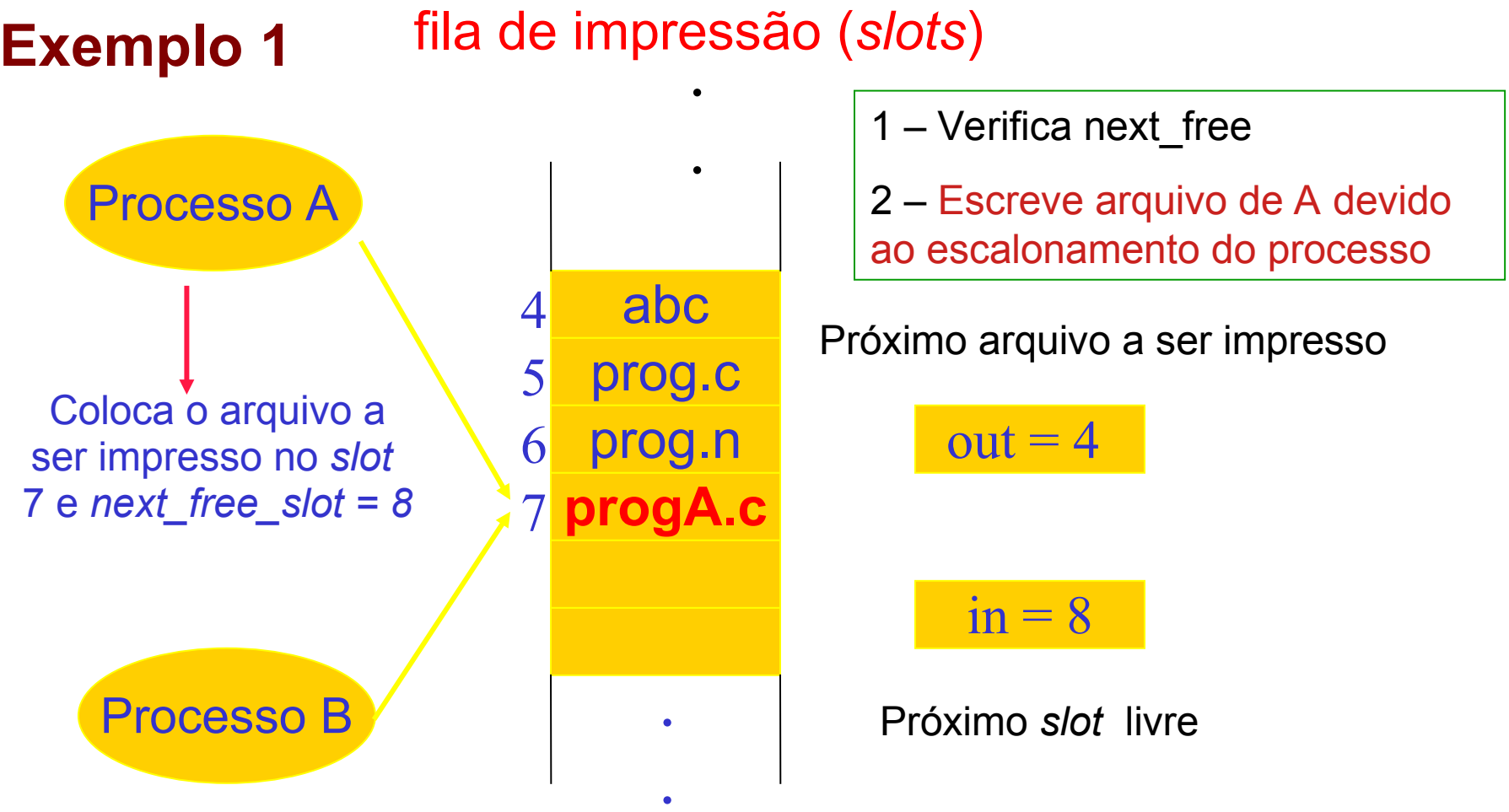
2 – **Escreve arquivo de B devido ao escalonamento do processo**

Spooler – fila de impressão (*slots*)



Comunicação de Processos

Exemplo 1



Processo B nunca receberá sua impressão

Comunicação de Processos

Quando há concorrência entre processos tem-se:

- **Condição de corrida;**
- **Atenção:** Ordem de execução é realizada pelo mecanismo de escalonamento do sistema operacional:
  - Torna a depuração difícil (tarefa dinâmica).
- **Solução:** A **exclusão mútua** garante que somente um processo estará usando os dados compartilhados num dado momento, o que evita a condição de corrida.

Região Crítica

- Exclusão mútua: garantir que um processo não terá acesso à uma região crítica enquanto outro processo está utilizando essa região;

```
while (true) {

    entry section

        critical section

    exit section

        remainder section

}
```

- Região de entrada - código que requer permissão para entrar na região crítica;
- Região de Saída – código que será executado após saída da região crítica.

Regiões Críticas

- Exclusão mútua: os códigos ilustram o mecanismo de exclusão mútua:
  - Existem **n** processos para serem executados concorrentemente;
  - Cada processo incluí uma região crítica que opera no recurso "**Ra**";

```
      /* PROCESS 1 */

void P1
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

```
      /* PROCESS 2 */

void P2
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

. . .

```
      /* PROCESS n */

void Pn
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

Comunicação de Processos

- **Definição:**
  - A **região Crítica** é a parte do programa (código) em que os dados compartilhados são acessados por todos.

- Como solucionar os problemas de condição de corrida (***Race Condition)***?
  - Propor soluções (**forma de sincronização**) que garantam que somente um processo de cada vez possa manipular o recurso disponível compartilhado entre os processos.

Regiões Críticas

As **situações indesejáveis** que devem ser evitadas:

- Espera indefinida: situação em que um processo nunca consegue executar sua região crítica e, consequentemente, acessar o recurso compartilhado;
  - Exemplo: Regras do algoritmo de escalonamento dos processos;
- Impedimento do progresso: Um processo fora de sua região crítica impede que outros processos entrem nas próprias regiões críticas.

Regiões Críticas

As condições **para uma boa solução**:

1) Dois processos não podem estar simultaneamente em regiões críticas;

2) Processos que não estão em regiões críticas não podem bloquear outros processos;

3) Nada pode ser afirmado sobre a velocidade ou sobre o número de CPUs;

4) Nenhum processo deve esperar eternamente para entrar em sua região crítica.

Exclusão Mútua

- **Soluções de Hardware**
  - Desabilitar interrupções (hardware)
  - Instrução TSL (apresenta busy wait)
- **Soluções de software com busy wait**
  - Variável de impedimento;
  - Alternância obrigatória (*Strict Alternation)*
  - Algoritmo de Peterson
  - **Problema:** constante checagem por um valor (looping)
- **Soluções de software com bloqueio**
  - Sleep / Wakeup, Semáforos e Monitores

Exclusão Mútua

Desabilitar Interrupções

- Processo desabilita todas as interrupções ao entrar na região crítica e reabilita as interrupções ao sair da região crítica;
- Usa instruções DI / EI (*DI = disable interrupt e EI = enable interrupt*) via hardware;
- Com as interrupções desabilitadas, a CPU não realiza chaveamento entre os processos.

Problema:

- Essa solução não é segura, pois um processo de usuário pode não reabilitar as interrupções ao sair da região;
  - Não será finalizado (“Fim do SO” - comprometido);
- Desabilita apenas uma CPU e **não funciona** para multiprocessadores.

Exclusão Mútua

- Solução de hardware - Desabilitar interrupções

```
while (true) {
        acquire lock
            critical section
        release lock
            remainder section
}
```

Desabilita interrupção

Habilita interrupção

Exclusão Mútua

- **Soluções de Hardware**
  - Desabilitar interrupções (hardware)
  - Instrução TSL (apresenta busy wait)
- **Soluções de software com busy wait**
  - Variável de impedimento;
  - Alternância obrigatória (*Strict Alternation)*
  - Algoritmo de Peterson
  - **Problema:** constante checagem por algum valor (looping)
- **Soluções de software com bloqueio**
  - Sleep / Wakeup, Semáforos e Monitores

Exclusão Mútua

Instrução TSL = “Test and Set Lock”

- Solução via **hardware** para a exclusão mútua em ambiente com multiprocessadores;
- A CPU que executa a instrução TSL bloqueia o barramento de memória, impedindo que outras CPUs acessem a memória principal até que a instrução seja finalizada;
- A instrução TSL opera:
  - Lê o conteúdo de um endereço de memória (variável “lock”, usada para proteger a região crítica) para um registrador e armazena um valor diferente de zero, normalmente, igual a 1 para esse endereço.

Exclusão Mútua

- Instrução TSL
  - **Se lock = 0 ⇒ R.C. livre;**
  - **Se lock = 1 ⇒ R.C. ocupada.**
- Lock é iniciada com o valor igual a 0;

```
enter_region:
  TSL REGISTER, LOCK | Copia lock para reg. e lock = 1
  CMP REGISTER, #0   | lock vale zero?
  JNE enter_region | Se sim, entra na região crítica,
                   | Senão, continua no laço
  RET              | Retorna para o processo chamador


leave_region:
 MOVE LOCK, #0 | lock = 0
 RET           | Retorna para o processo chamador
```

Exclusão Mútua

- **Soluções de Hardware**
  - Desabilitar interrupções (hardware)
  - Instrução TSL (apresenta busy wait)
- **Soluções de software com busy wait**
  - Variável de impedimento
  - Alternância obrigatória (*Strict Alternation)*
  - Algoritmo de Peterson
  - **Problema:** constante checagem por algum valor (looping)
- **Soluções de software com bloqueio**
  - Sleep / Wakeup, Semáforos, Monitores

Exclusão Mútua

Variável de Impedimento Lock

- Ocorre a **verificação da permissão** para que um processo possa acessar a região crítica.
- Caso não seja permitido deve esperar em um laço até que o acesso a região seja liberado:
  - Ex: `while (cont == critério) {“não faz nada”};`
- O processo que entra na **região crítica** atribuí valor a variável. A consequência: *desperdício de tempo de CPU (teste de condição);*
- Se o processo com baixa prioridade acessar a região critica e ocorrer uma interrupção, se um processo de alta prioridade for selecionado **pode ocorrer a espera ativa.**

Exclusão Mútua

Variável de impedimento Lock

- A variável de bloqueio é compartilhada e indica se a RC está ou não em uso;
- Uma parte do programa controla o acesso às regiões críticas;
- A variável "**turn == 0**" (RC livre) ou "**turn == 1**" (RC em uso).

```
var turn = 0 ou 1
turn = 0
Processo Pi (i= 0 ou 1):
 ...
 while (turn == 1);
 turn = 1;
   <região critica>
 turn = 0;
 ...
```

Exclusão Mútua

Variáveis de impedimento

- Se os processos concluírem "simultaneamente" que a RC está livre;

- Ex: os dois processos podem testar o valor de **turn(zero)** antes que a variável seja atribuído **1** por um deles.

- Exemplo: "Fila de impressão"

```
var turn = 0 ou 1
turn = 0
Processo Pi (i= 0 ou 1):
 ...
 while (turn == 1);
 turn = 1;
   <região critica>
 turn = 0;
 ...
```

Exclusão Mútua

Código de Exemplo: tentativa_lock.c

```
int lock=0;
void* funcaoA(void *argumento) {
   while (lock == 1);
   lock = 1;
   valor = 0;
   printf("Thread A, valor = %d.\n", valor);
   lock = 0;
}
```

Exclusão Mútua

- **Soluções de Hardware**
  - Desabilitar interrupções (hardware)
  - Instrução TSL (apresenta busy wait)
- **Soluções de software com busy wait**
  - Variável de impedimento
  - Alternância obrigatória (*Strict Alternation)*
  - Algoritmo de Peterson
  - **Problema:** constante checagem por algum valor (looping);
- **Soluções de software com bloqueio**
  - Sleep / Wakeup, Semáforos, Monitores

Exclusão Mútua

Alternância Obrigatória – Spin Lock (Solução 2)

- Variável **`turn`** indica de quem é a vez de entrar na RC;
- Ex: **`Processo 1 –> turn = 0`**;
- Processo 2: O que acontece se o processo 2 for mais rápido quando sair da região crítica?

```
var turn = 0...1;
```

```
while (TRUE) {
    while (turn!= 0); //loop
    critical_region();
    turn = 1;
    noncritical region();
}
```

(Processo 1)

```
while (TRUE) {
    while (turn!= 1); //loop
    critical_region();
    turn = 0;
    noncritical region();
}
```

(Processo 2)

Exclusão Mútua

Problema:

- Ambos os processos estão fora da região crítica (t**urn = 0**);
- **`Processo 1`** terminar antes de executar sua região **não crítica** e retorna ao início do *loop*;
  - **`turn = 0`** e **`processo 1`** entra na região crítica
  - **`processo 2`** ainda está na **região não crítica**;
- Ao sair da **região crítica**, o **`processo 1`** atribui **`turn = 1`** e entra na sua região **não crítica**;
- Novamente, os processos estão na região **não crítica** e **`turn = 1`**.

Exclusão Mútua

Problema:

- Quando o **processo 1** tenta entrar na região crítica, não consegue devido a variável **turn = 1**;

- Se, no algoritmo de escalonamento, o **processo 1** tem prioridade superior ao **processo 2** :

  - Assim, o **processo 1** "bloquea" o **processo 2** que está na sua região não crítica.

  - **Espera indefinida**;

- Além disso, se um **processo falhar ou terminar**, o outro não poderá mais entra na sua RC, ficando bloqueado permanentemente.

Exclusão Mútua

- **Soluções de Hardware**
  - Desabilitar interrupções (hardware)
  - Instrução TSL (apresenta busy wait)
- **Soluções de software com busy wait**
  - Variável de impedimento
  - Alternância obrigatória (*Strict Alternation)*
  - Algoritmo de Peterson
  - **Problema:** constante checagem por algum valor (looping);
- **Soluções de software com bloqueio**
  - Sleep / Wakeup, Semáforos, Monitores

Exclusão Mútua

Solução de Gary L. Peterson

- Em 1981, Peterson descobriu uma forma mais simples para exclusão mútua;
- A proposta do algoritmo consiste no seguinte:
  - ao marcar a sua intenção de entrar, o processo já indica (para o caso de empate) que a vez é do outro.
- Utiliza uma variável de condição **(interested),** que indica o desejo de cada processo entrar na região crítica.

Exclusão Mútua

Cada processo usa:
enter_region e leave_region

P0 ou P1

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

int turn;                               /* de quem é a vez? */
int interested[N];                      /* todos os valores inicialmente em 0 (FALSE) */

void enter_region(int process);         /* processo é 0 ou 1 */
{
    int other;                          /* número de outro processo */

    other = 1 – process;                /* o oposto do processo */
    interested[process] = TRUE;         /* mostra que você está interessado */
    turn = process;                     /* altera o valor de turn */
    while (turn == process && interested[other] == TRUE) /* comando nulo */;
}

void leave_region(int process)          /* processo: quem está saindo */
{
    interested[process] = FALSE;        /* indica a saída da região crítica */
}
```

Veja o código: peterson.c

Exclusão Mútua

Solução de G. L. Peterson

- Exclusão mútua é atingida:
  - Uma vez que processo **P0** tenha feito "**interested[0] = TRUE**": **P1** não pode entrar na sua R.C.;
  - Se **P1** já estiver na sua RC (**interested[1] = TRUE**):
    - P0 está impedido de entrar na R.C.;
- Não requer alternância obrigatória;

Exclusão Mútua

Solução de G. L. Peterson

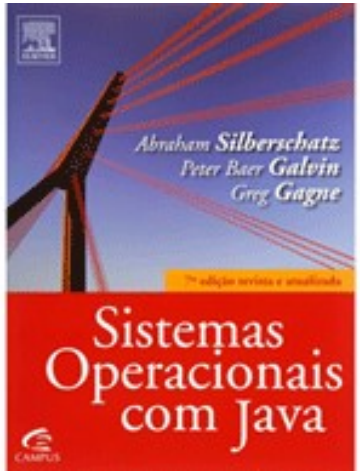
- Bloqueio mútuo (deadlock) é evitado.
  - Supondo P0 bloqueado na condição:
    - **`while (interested[1] == true e turn == 0)`**;
    - Se interested[1] = true e turn = 0.
  - P1 por sua vez pode entrar na sua seção crítica
    - interested[0] = false;
  - P0 só poderá entrar quando:
    - interested[1] = false ou turn = 0.

Exclusão Mútua

- Limitações dessas soluções:
  - Essas soluções utilizam espera ocupada => processos ficam em estado de espera (*looping)* até que possam utilizar a região crítica;
  - Tempo de processamento da CPU – ocasionar problemas de desempenho;
  - Determinadas threads podem ficar ocupadas (ex. prioridade);

Leituras

- Silberschatz, A., Galvin, P. B. Gagne, G. Sistemas Operacionais com Java. 7º, edição. Editora, Campus, 2008.
  - Capítulo 6



- Andrew S. Tanembaum. **Sistemas Operacionais. Modernos**. 2ª Ed. Editora Pearson, 2003.
  - Capítulo 2