



SISTEMAS OPERACIONAIS

Módulo 4 – Comunicação entre Threads

Prof. Daniel Sundfeld
daniel.sundfeld@unb.br



A ARTE DE SE MULTIPROGRAMAR



Fonte:

https://www.reddit.com/r/ProgrammerHumor/comments/2smem5/multithreading_expectations_vs_reality/



CONDIÇÃO DE CORRIDA

- Também chamadas de condições de disputa
- Como o sistema operacional determina através do seu escalonador como os processos irão executar, não sabemos a ordem que os processos podem executar
- Trocas de contexto podem acontecer a qualquer momento!!!
- Condições de corrida são extremamente importantes para nosso curso. Vamos rever.



CONDIÇÃO DE CORRIDA 1

- Considere os seguinte Processos/Thread incrementando uma variável em memória compartilhada

Processo / Thread A

$$x = x + 1$$

Processo / Thread B

$$x = x + 1$$

- Considere, $x = 0$ inicialmente. Quais valores possíveis que x pode obter ao final?



CONDIÇÃO DE CORRIDA 1

- Escalonamento: $A \rightarrow B$
- Assumindo que X está na posição de memória

0x2000

Processo / Thread A

LOAD 0x2000, R1 (x=0)

INC R1

STORE 0x2000, R1 (x=1)

Processo / Thread B

LOAD 0x2000, R1 (x=1)

INC R1

STORE 0x2000, R1 (x=2)

Tempo

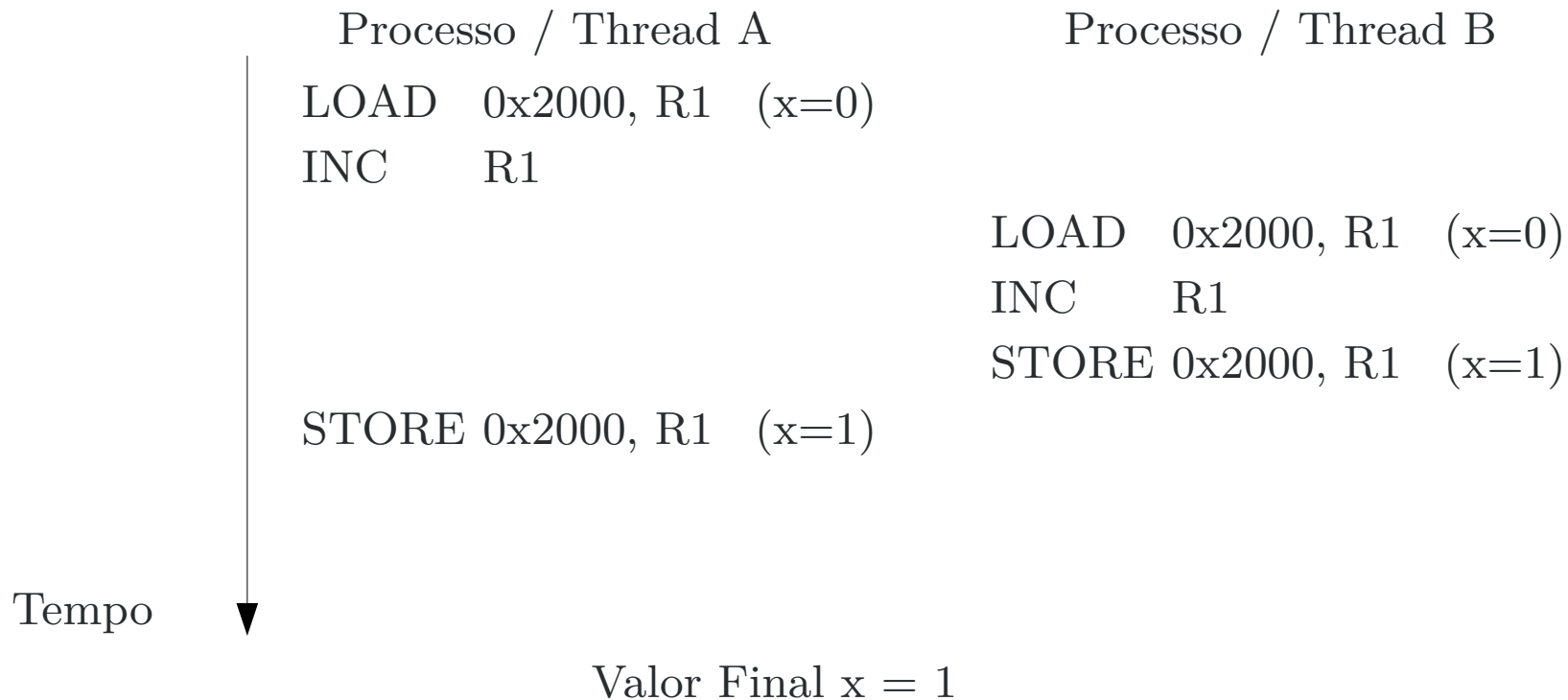


Valor Final $x = 2$



CONDIÇÃO DE CORRIDA 1

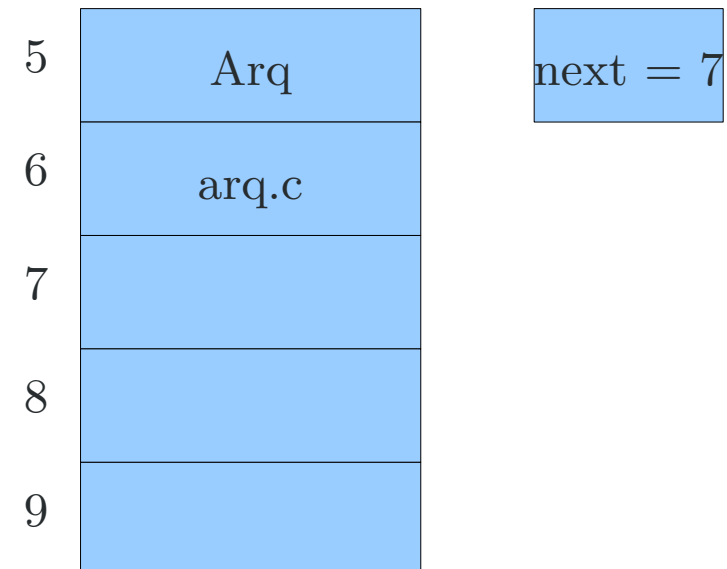
- Escalonamento: $A \rightarrow B \rightarrow A$





CONDIÇÃO DE CORRIDA 2

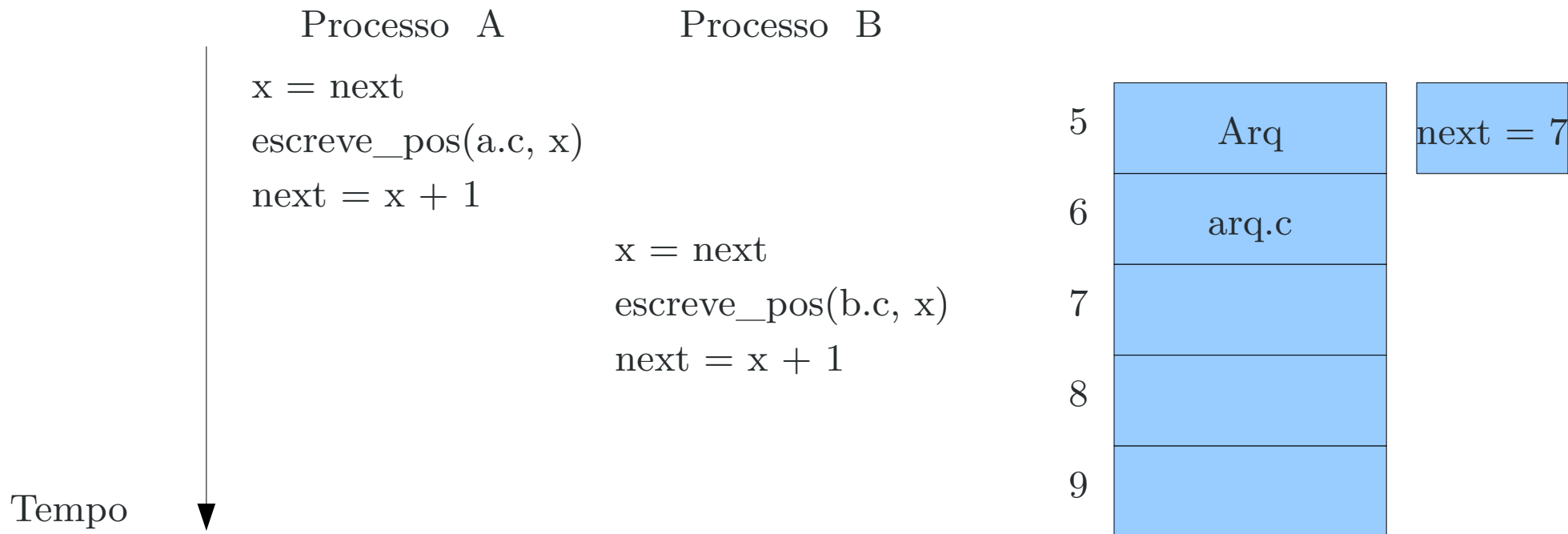
- Imagina que um servidor de impressão enumere as vagas dos arquivos impressos 0, 1, 2, ...
- Uma variável `next` aponta para a próxima posição livre a ser impressa
- Imagina agora, que o processo A e B desejaram imprimir um arquivo e o servidor está com a seguinte configuração





CONDIÇÃO DE CORRIDA 2

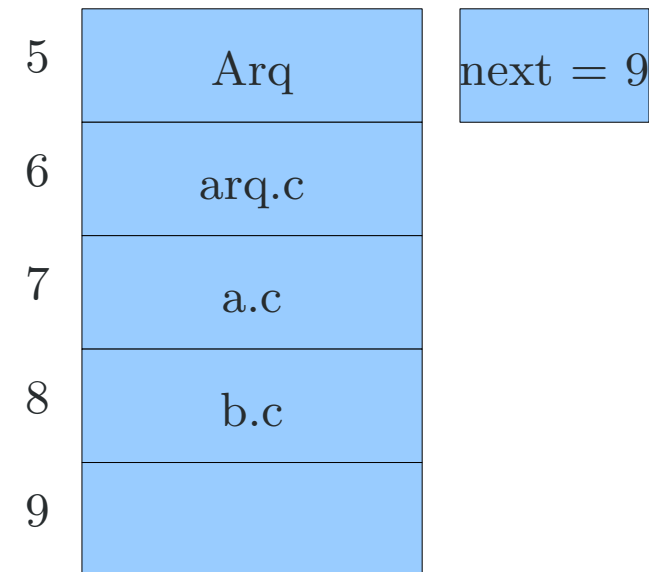
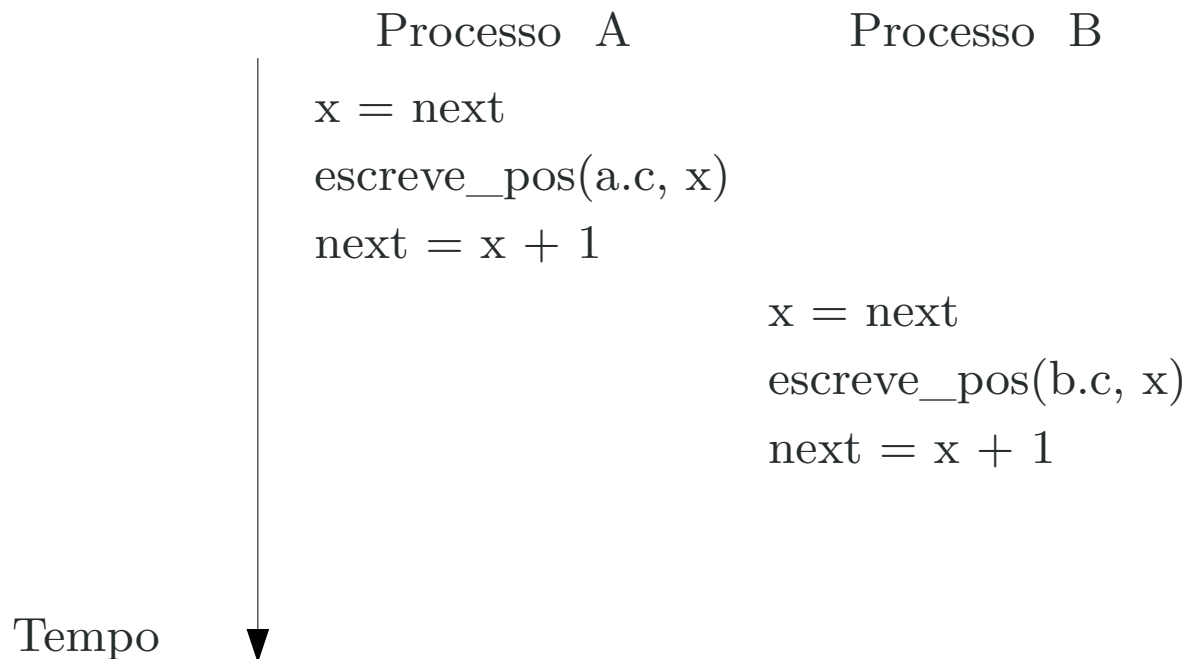
- Escalonamento: $A \rightarrow B$
- x é uma variável local





CONDIÇÃO DE CORRIDA 2

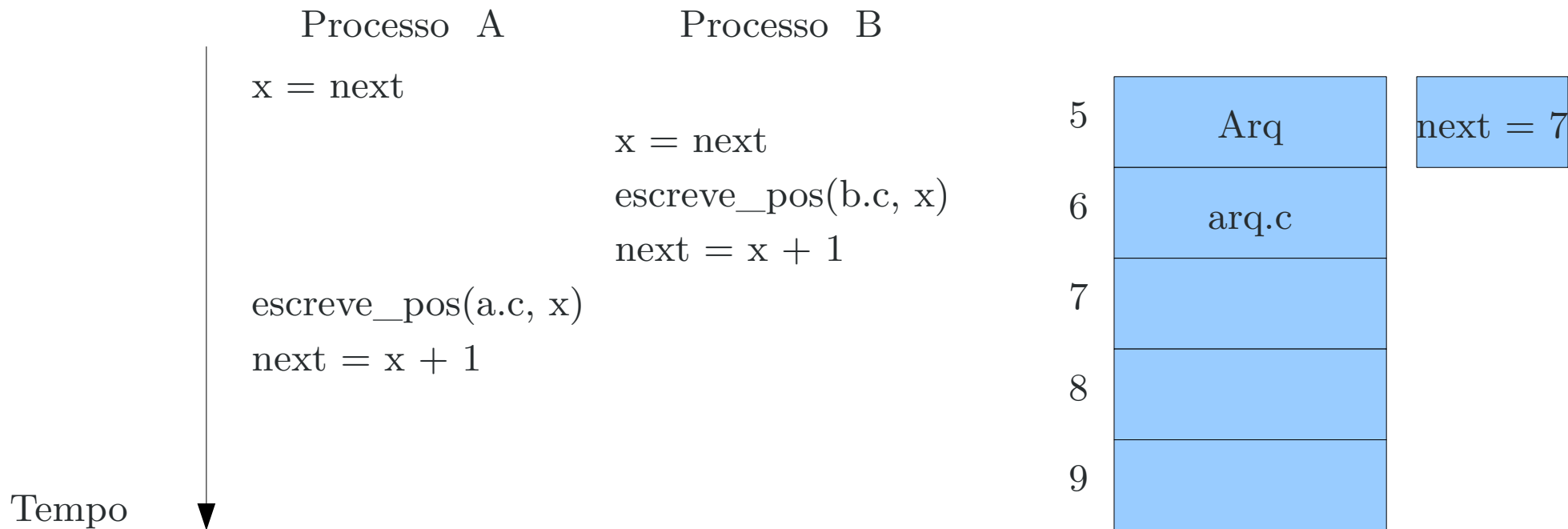
- Escalonamento: $A \rightarrow B$
- x é uma variável local





CONDIÇÃO DE CORRIDA 2

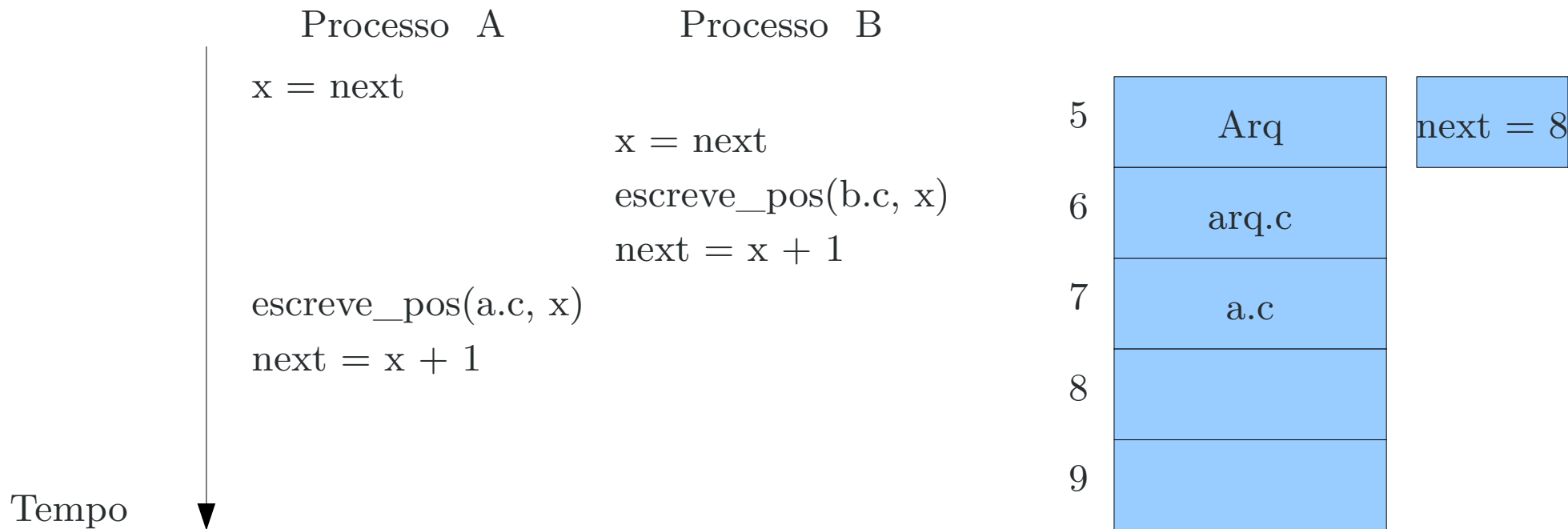
- Escalonamento: $A \rightarrow B \rightarrow A$
- x é uma variável local





CONDIÇÃO DE CORRIDA 2

- Escalonamento: $A \rightarrow B \rightarrow A$
- x é uma variável local





CONDIÇÕES DE CORRIDA

- As condições de corrida levam a resultados inesperados e devem ser evitadas
- Garantir a **exclusão mútua**
- Trecho de código que há acesso de leitura ou escrita à dados compartilhados é chamado de seção crítica



CONDIÇÕES DE CORRIDA

- Para evitar as condições de corrida, são colocadas funções antes de entrar e depois de sair da seção crítica
- Essas funções utilizam diversas técnicas para impedir que dois processos estejam simultaneamente na seção crítica e garantir a exclusão mútua



REGIÕES CRÍTICAS

- Um processo precisa ter acesso à dados compartilhados para poder cooperar entre si
- O trecho de código que há acesso de leitura ou escrita à dados compartilhados é chamado de **seção crítica** (ou região crítica)
- No primeiro exemplo a seção crítica é a operação de incremento. No segundo exemplo é toda a operação de escrever. A seção crítica normalmente é MAIS de uma instrução



VARIÁVEIS DE IMPEDIMENTO

- Variável de impedimento (tipo trava) busca marcar se existe alguém na seção crítica. Se for 1, não procede
- Analise o trecho de código:

```
int thread()  
{  
    while (true)  
    {  
        while (lock == 1) {}  
        lock = 1;  
        regioao_critica();  
        lock = 0;  
    }  
}
```


```
int thread()  
{  
    while (true)  
    {  
        while (lock == 1) {}  
        lock = 1;  
        regioao_critica();  
        lock = 0;  
    }  
}
```



VARIÁVEIS DE IMPEDIMENTO

- Se ocorrer uma troca de contexto depois de sair do loop e antes do processo/thread trocar o valor para 1, há uma condição de corrida

```
int thread()
{
    while (true)
    {
        while (lock == 1) {}
        lock = 1;
        regioao_critica();
        lock = 0;
    }
}
```



```
int thread()
{
    while (true)
    {
        while (lock == 1) {}
        lock = 1;
        regioao_critica();
        lock = 0;
    }
}
```




VARIÁVEIS DE IMPEDIMENTO

- A solução para exclusão mútua não é trivial!
- Entendeu?





EXCLUSÃO MÚTUA

- Uma boa solução de exclusão mútua:
 - Nunca dois processos podem estar simultaneamente em suas regiões críticas
 - Nada pode ser afirmado sobre a velocidade ou sobre o número de CPUs
 - Nenhum processo executando fora de sua região crítica pode bloquear outros processos
 - Nenhum processo deve esperar eternamente para entrar em sua região crítica



TÉCNICAS DE IMPLEMENTAÇÃO DE EXCLUSÃO MÚTUA

- Inibir Interrupções
- Com espera ocupada:
 - Estrita Alternância
 - Algoritmo de Peterson
 - Utilizar hardware adicional
- Com bloqueio de processos:
 - Semáforos
 - Mutexes
 - Locks
 - Monitores
 - Variáveis de condição



TÉCNICAS PARA EXCLUSÃO MÚTUA

- **Inibir interrupções**
- Com espera ocupada
- Com bloqueio de processos



INIBIR INTERRUPÇÕES

- Se for possível desabilitar interrupções, um processo pode desabilitar e ligar antes e depois de acessar sua região crítica, respectivamente
- Um processo que desabilitou as interrupções não pode ser retirado à força da CPU, não é interrompido pelo escalonador
- Sendo assim, não há problemas de acesso concorrentes



INIBIR INTERRUPÇÕES

- Ao inibir interrupções, um algoritmo garante todas as características
- Porém, os programas podem ter defeitos.
Grande impacto se um programador “esquecer” de desabilitar
- Função amplamente utilizada apenas em modo kernel, o programador no espaço de usuário não tem acesso à essa chamada



TÉCNICAS PARA EXCLUSÃO MÚTUA

- Inibir interrupções
- **Com espera ocupada**
- Com bloqueio de processos



ESPERA OCUPADA

- A espera ocupada também é chamada de busy waiting
- `while (vez != minha) { };`
- A espera ocupada desperdiça o tempo que possui CPU fazendo um teste trivial
- Deve ser utilizada quando há uma expectativa de esperar pouco/muito pouco
- Algumas vezes obrigatória em modo kernel



BLOQUEIO DE PROCESSOS

- O processo espera a permissão de entrada na seção crítica e executa uma primitiva, chamada de sistema, que causa o seu bloqueio até que a seção crítica seja liberada
- `if (vez != minha)`
 `wait_my_turn(vez);`
- O bloqueio ocasiona uma troca de contexto entre processos/threads e pode causar uma espera longa
- Chamadas blocantes podem ser não estar disponíveis em modo protegido



TÉCNICAS DE IMPLEMENTAÇÃO DE EXCLUSÃO MÚTUA

- Inibir Interrupções
- Com espera ocupada:
 - Estrita Alternância
 - Algoritmo de Peterson
 - Utilizar hardware adicional
- Com bloqueio de processos:
 - Semáforos
 - Mutexes
 - Locks
 - Monitores
 - Variáveis de condição



ESTRITA ALTERNÂNCIA

- A estrita alternância resolve o problema da exclusão mútua para dois processos
- A ideia é que um processo marque que está na seção crítica
- O outro processo aguarda a saída



ESTRITA ALTERNÂNCIA

- Solução: `int turn = 0;`

```
int thread_A()  
{  
    while (true)  
    {  
        while (turn != 1)  
            ;  
        regiao_critica();  
        turn = 0;  
        regiao_nao_critica();  
    }  
}
```

```
int thread_B()  
{  
    while (true)  
    {  
        while (turn != 0)  
            ;  
        regiao_critica();  
        turn = 1;  
        regiao_nao_critica();  
    }  
}
```



ESTRITA ALTERNÂNCIA

- Desvantagem: a estrita alternância não deve ser utilizada quando um processo é muito mais lento do que outro
- Porém, é bem implementada em processos iguais
- Ela viola a regra de um processo fora da seção crítica bloquear outro processo
- Não é uma solução genérica



ALGORITMO DE PETERSON

- Em 1981, Peterson melhorou um algoritmo baseado em outro (chamado algoritmo de Dekker)
- Esse algoritmo minimizou o número de loops e comparações necessárias
- Garante a exclusão mútua



ALGORITMO DE PETERSON

- Os processos possuem um id único (0 ou 1)
- O processo deve chamar uma função `enter_region`, que retorna só quando for seguro entrar na seção
- Ao terminar o processamento, a função `leave_region` deve ser chamada para indicar que outros processos podem prosseguir
- Isso garante que um processo fora da seção crítica não bloqueie outros



ALGORITMO DE PETERSON

```
/* argumento e' id do processo */      turn = 0;
void enter_region(int id)               interested[0] = false;
{                                       interested[1] = false;

    int other;

    other = 1 - id;                    /* O outro processo */
    interested[id] = true;              /* Mostra que esta interessado */
    turn = id;                          /* mostra que a vez e' minha */
    while (turn == id && interested[other] == true)
        ;
}

void leave_region(int id)
{
    interested[id] = false;
}
```




ALGORITMO DE PETERSON

- Caso simples: um processo sempre chama a função `enter_region` quando o outro está na seção crítica
- Nesse caso, o processo sempre trava na condição `interested[other] == true;`



ALGORITMO DE PETERSON

- Caso complicado, os dois processos chamam juntos a função de entrar na seção crítica
- Nesse caso, algum deles irá escrever por último na variável turn
- Desta forma, o loop do OUTRO processo será falso e apenas um processo entra na seção crítica



A INSTRUÇÃO TSL

- Existe uma solução que requer uma função de hardware adicional
- Neste caso, a instrução testa e altera um valor de maneira **atômica** (indivisível)
- Operações atômicas são importantes em diversas áreas da computação



A INSTRUÇÃO TSL

- Desta forma, podemos fazer um código parecido com as variáveis de impedimento
- `while (test_and_set(v) != 0){ }`
- A instrução TSL equivale logicamente a:
 - `while (lock == 0) {};`
 - `lock = 1;`



A INSTRUÇÃO TSL

- A instrução TSL resolve o problema das variáveis de impedimento

TSL:

Faz o teste do loop e altera a variável para 1 atomicamente, ou seja, na mesma instrução ou de forma que não possa ser

```
int thread()  
{  
    while (true)  
    {  
        while (lock == 1) {}  
        lock = 1;  
        regioao_critica();  
        lock = 0;  
    }  
}
```

Algumas linguagens de programação permitem criar variáveis atômicas

Por exemplo, C++ podemos declarar:

`std::atomic<int> x;`



ESPERADA OCUPADA: DESVANTAGENS

- A espera ocupada deve ser usada em modo protegido ou quando a espera do lock for baixa
- Porém, se essas situação não forem atingidas, é desejável colocar o processo para dormir, de forma que ele não consuma CPU
- Além disso, existe o problema de prioridades invertidas, quando utiliza-se um escalonador de prioridade estática



PRIORIDADE INVERTIDA

- O processo de PB (prioridade baixa) entra na seção crítica
- O processo PA (prioridade alta) entra no loop de acesso à seção crítica
- Como o processo de alta prioridade fica utilizando a CPU, o escalonador sempre irá escolher esse processo para ser executado
- O processo de baixa prioridade não é executado nunca



TÉCNICAS PARA EXCLUSÃO MÚTUA

- Inibir interrupções
- Com espera ocupada
- **Com bloqueio de processos**



DORMIR E ACORDAR

- Algumas primitivas podem ser utilizadas para que o processo seja bloqueado
- A primitiva sleep/wakeup podem ser usadas para colocar um processo para dormir ou acordá-lo
- Desta forma, o processo não consome CPU enquanto espera para entrar na seção crítica



PRODUTOR CONSUMIDOR

- No problema do Produtor-Consumidor, uma thread irá produzir itens e uma outra thread irá consumir itens
- Produção e consumo em paralelo
- Porém existe um buffer limitado
- Desta forma, o produtor deve dormir, caso esteja cheio e o consumidor deve dormir se vazio
- Um deve acordar o outro



PRODUTOR CONSUMIDOR

```
#define N 100
int count = 0;

void producer()
{
    int item;

    while (true)
    {
        item = produce_item();
        if (count == N)
            sleep(producer);
        insert_item(item);
        count = count + 1;
        if (count == 1)
            wakeup(consumer);
    }
}
```

```
void consumer()
{
    int item;

    while (true)
    {
        if (count == 0)
            sleep(consumer);
        item = remove_item();
        count = count - 1;
        if (count == N-1)
            wakeup(producer);
        consume_item(item);
    }
}
```



PRODUTOR CONSUMIDOR

- Nota: a chamada `wakeup(id)` não existe na linguagem C e a chamada `sleep` é utilizada em outro sentido
- Pseudo-código em C
- Pergunta: esse código funciona?



PRODUTOR CONSUMIDOR

- Pode existir uma troca de contexto que cause problema no código
- Problema se a troca for imediatamente antes do valor novo ser corrigido



SEMÁFORO

- Pensando nesse problema, Dijkstra sugeriu um novo tipo de variável chamada **semáforo**
- Essa variável serve como contador de quantos sinais foram recebidos
- Neste caso, uma variável de semáforo possui valor 0 se não recebeu nenhum sinal
- Ou possui um valor positivo para o número de sinais a serem tratados



SEMÁFORO

- Os semáforos se baseiam, assim como TSL, em operações **atômicas**
- Operação $\text{down}(\text{sem})$ ou $P(\text{sem})$
 - Decrementa o valor do semáforo se for maior do que 0 e continua; ou Bloqueia se o valor for 0
- Operação $\text{up}(\text{sem})$ ou $V(\text{sem})$
 - Incrementa o valor de um semáforo. Se algum processo estiver dormindo nele, algum deles é escolhido para tratar
 - Nenhum processo é bloqueado ao dar um up ou wakeup



SEMÁFORO



```
semaphore s = 0;  
int thread1()  
{  
    printf("T1 - Inicio\n");  
    up(s);  
    printf("T1 - Fim\n");  
}  
  
int thread2()  
{  
    down(s);  
    printf("T2\n");  
}
```

- A primeira mensagem sempre é “T1 – Inicio”
- “T1- Fim” ou “T2” podem ser impressos depois dela, dependendo do escalonamento



SEMÁFORO – PRODUTOR CONSUMIDOR

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

```
void producer()
{
    int item;

    while (true)
    {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

```
void consumer()
{
    int item;

    while (true)
    {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```



SEMÁFORO

- Neste exemplo, combinamos diferentes usos de semáforos
- O semáforo mutex foi usado para garantir que as threads não acessem simultaneamente a seção crítica
 - Semáforo binário
- O semáforo full e empty foram usados para sincronizar o trabalho
 - Semáforo de sincronização



MUTEX

- Quando a capacidade de contagem do semáforo não é necessária, pode-se usar uma versão simplificada chamada mutex
- Mutex pode estar em dois estados: travados e destravados
- Se um mutex está travado, ele bloqueia a chamada de lock
- Se um mutex está destravado, ele permite que o processo continue



MUTEX

- Mutexes são uma solução simples, facilmente implementados e que resolvem o problema da exclusão mútua

```
void thread()  
{  
    mutex_lock(&my_mutex);  
    regioao_critica();  
    mutex_unlock(&my_mutex);  
}
```



MUTEX

- Quando um processo pede um `mutex_lock` e ele está destravado, o processo continua a execução
- Se o mutex está travado, ele bloqueia até que alguém libere
- Ao liberar o mutex, apenas um processo/thread que está aguardando é liberado
 - Necessário para garantir a seção crítica



MUTEX

- A maioria das implementações dos computadores modernos utilizam mutex como chamada blocantes
- Porém é possível utilizar busy waiting para implementar um mutex: ele utiliza CPU até que a chamada termine



MUTEX

- O Linux inovou na implementação de mutex, chamando-o de futex (Fast userspace mutex)
- Em resumo, mutex são uma das mais eficientes e rápidas estruturas de comunicação entre threads
- Quando não há contenção, não é necessário uma troca de contexto entre processos e SO



LOCKS

- Mutex podem ser chamados de locks exclusivos, isto é, sempre tem apenas um processo na seção crítica
- Porém, muitas vezes as estruturas vão compartilhar os dados, mas não vão escrever
- Desta forma, se a operação for de apenas leitura, mais de uma thread pode acessar a seção crítica sem condições de corrida



LOCKS

- Desta forma, locks definem 2 funções:
 - read_lock: indicando que a thread irá apenas ler os dados compartilhados
 - write_lock: indicando que a thread irá escrever e só pode entrar na seção crítica quando apenas um processo/thread estiver nela



MONITORES

- A comunicação com semáforos e mutexes pode parecer simples, no entanto vários erros podem surgir em um projeto multithread
- Por exemplo, no código do Produtor/Consumidor o caos irá acontecer se inverter as linhas dos mutex
- Linhas de código diferentes devem manter uma ordem de aquisição dos mutex, caso contrário não funciona!
 - Manutenção cara



MONITORES

- Com semáforos e mutex, o programador recebe explicitamente as ferramentas que irão proteger a área de dados
- Em 1973/1974, Hansen/Hoare propuseram um mecanismo de alto nível de sincronização de processos, chamado monitor
- Busca facilitar a escrita de programas corretos



MONITORES

- Coleção de rotinas, variáveis e estruturas de dados reunidas em um tipo especial de módulo ou pacote
- Processos podem chamar rotinas de um monitor, mas nunca acessar os dados dentro deles
- Apenas um processo pode estar ativo em um determinado monitor a cada instante



MONITORES

- Monitores são normalmente primitivas de uma linguagem de programação
- Desta forma, o paralelismo e a exclusão mútua é projetado pelo compilador e não pelo programador
- Fácil programação
- Menos provável que dê errado



MONITORES

- Normalmente, o compilador usa as estruturas de baixo nível, porém linguagens modernas não possuem o conceito de monitor nelas
 - C não possui
- Java possui uma palavra-chave “synchronized” que permite para adicionar a um objeto, que garante a exclusão mútua entre todos os métodos synchronized de um mesmo objeto



MONITORES

- Outro problema é a necessidade de bloquear. E no caso do produtor/consumidor, o que fazer para o consumidor esperar ter uma tarefa?
- Neste caso precisamos de um mecanismo de sincronização: variáveis de condição



VARIÁVEIS DE CONDIÇÃO

- Nesse caso, quando o Produtor vai produzir um item e o buffer está cheio, ele vai esperar em um mecanismo de sincronização chamado variável de condição
- Funções: wait e signal
- Quando um consumidor consome um item, ele desbloqueia o produtor usando a chamada
- Chamada pode parecer igual a sleep e wakeup que falhava (slides anteriores)



VARIÁVEIS DE CONDIÇÃO

- Essa chamada falha apenas porque algum outro processo altera o sleep antes
- Variáveis de condição assumem que a condição existente antes do teste de dormir não é alterada

```
void produtor()  
{  
    int item;  
  
    item = produz_item();  
    if (n == N)  
        wait(&cond);  
    insert_item(item);  
}
```

Atômico!
Isso está
dentro de um
monitor



VARIÁVEIS DE CONDIÇÃO

- Essa condição pode ser atingida, adicionando o mecanismo de sincronização dentro de um monitor
- Em C, que não existe monitores, o pacote Pthreads definiu que a variável de condição exige o uso de um mutex em conjunto para garantir que essa condição é satisfeita



VARIÁVEIS DE CONDIÇÃO

- Variáveis de condição não são contadores
- Se alguém emitir um signal antes de um wait, esse sinal será perdido para sempre
- Deste modo é necessário verificar a condição antes do wait

Certo:

```
void produtor()  
{  
    int item;  
  
    item = produz_item();  
    if (n == N)  
        wait(&cond);  
    insert_item(item);  
}
```

Errado:

```
void produtor()  
{  
    int item;  
  
    item = produz_item();  
    wait(&cond);  
    insert_item(item);  
}
```



PRODUTOR CONSUMIDOR C/ MONITORES

```
BEGIN_MONITOR monitor
condicao cheio, vazio;
int cont;
count = 0;

void insere(item)
{
    if (cont == N) wait(cheio);
    insere_item(item);
    cont++;
    if (cont == 1) signal(vazio);
}

int remove()
{
    if (cont == 0) wait(vazio);
    item = remove_item();
    cont--;
    if (cont == N-1) signal(cheio);
    return item;
}

END_MONITOR
```

```
void produtor()
{
    while (true)
    {
        item = produz_item();
        monitor.insere(item);
    }
}

void consumidor()
{
    while (true)
    {
        item = monitor.remove(item);
        consome_item(item);
    }
}
```



TROCA DE MENSAGENS

- A troca de mensagens pode ser utilizada em ambientes distribuídos:
 - `send(msg)`: envia uma mensagem
 - `recv(msg)`: recebe uma mensagem
- A troca é explícita e pode incluir um envio por rede (processos em máquinas diferentes) ou cópia de regiões de memória de processos diferentes (processos na mesma máquina)



TROCA DE MENSAGENS

- Quando projetamos a comunicação por troca de mensagens, podemos ter diversas características:
 - Primitivas bloqueadas ou não bloqueadas: o processo deve bloquear até o outro receber a mensagem ou pode aguardar?
 - Tipo de comunicação: um processo send envia para apenas um processo, ou pode enviar para vários (broadcast)



TROCA DE MENSAGENS

- A troca de mensagens é amplamente utilizada em programação paralela
- MPI – Message Passing Interface
- Interface de troca de mensagens



PRODUTOR CONSUMIDOR COM TROCA DE MENSAGENS

- Muito simples:

```
void produtor()
{
    while (true)
    {
        produz_item(&item);
        send(item);
    }
}
```

```
void consumidor()
{
    while (true)
    {
        recv(item);
        consome_item(item);
    }
}
```



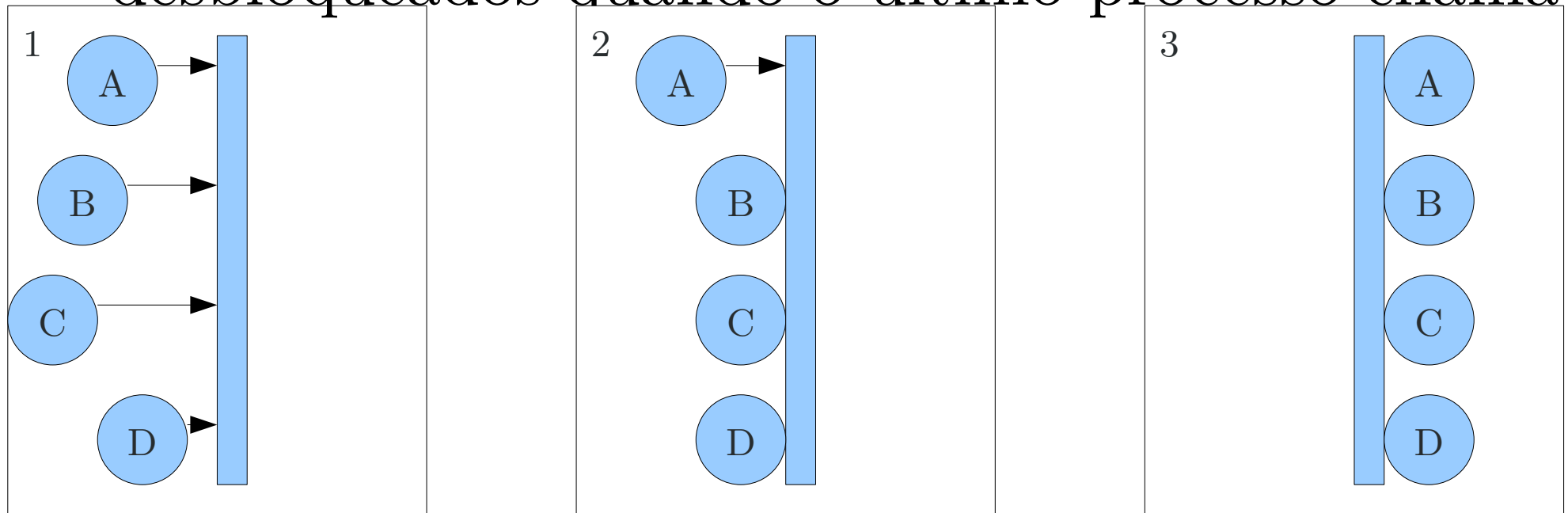

BARREIRAS

- Um mecanismo de sincronização amplamente utilizado e muito desejável são as barreiras
- Dependendo do problema, uma thread deve prosseguir para uma nova fase apenas quando todas as outras terminarem a primeira
- Isso é possível ao se colocar uma barreira ao final de cada fase



BARREIRAS

- Uma barreira é uma chamada blocante, onde todos os processos aguardam e são desbloqueados quando o último processo chama



Tempo



BARREIRAS

- As barreiras não estão explicitamente criadas no pacote Pthread
- É possível usar um mutex, uma variável de condição e um contador para implementar uma barreira



BARREIRAS

- Exemplo de barreira:

```
condition cond;  
mutex lock;  
void sync_threads()  
{  
    mutex_lock(&lock);  
    if (++sync_count < N_THREADS)  
        cond.wait(lock);  
    else  
    {  
        sync_count = 0;  
        cond.notify_all();  
    }  
    mutex_unlock(&lock);  
}
```



BARREIRA EM OUTRAS LINGUAGENS

- Algumas linguagens de programação possuem explicitamente uma barreira já declaradas
- CUDA C:
 - `__syncthreads();`
- OpenMP:
 - `#pragma omp barrier`



Referências

- Capítulo 2 – TANENBAUM, A. S. *Sistemas Operacionais Modernos*. 4^a ed. Prentice Hall, 2016.
- Capítulo 7 – MACHADO, F. B; MAIA, L. P. *Arquitetura de Sistemas Operacionais*. 5^a ed. Rio de Janeiro: LTC, 2013.