

Sistemas Operacionais

Aula 06 - Sincronização de Processos

Prof. Samuel Souza Brito

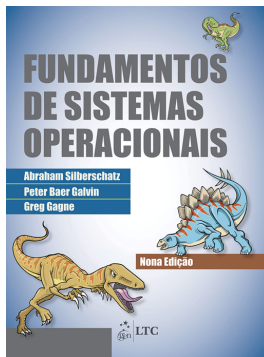
Universidade Federal de Ouro Preto – UFOP
Instituto de Ciências Exatas e Aplicadas – ICEA
Departamento de Computação e Sistemas – DECSI



UFOP

João Monlevade-MG
2024/2

- O material aqui apresentado é baseado no Capítulo 5 do livro:
 - Silberschatz, A.; Galvin, P. B.; Gagne, G., Fundamentos de Sistemas Operacionais, Editora LTC, 9ª edição, 2015.



- Um processo cooperativo é aquele que pode afetar ou ser afetado por outros processos em execução no sistema.
- Processos cooperativos podem:
 - Compartilhar diretamente um espaço de endereços lógico.
 - Memória compartilhada e *threads*.
 - Ter permissão para compartilhar dados apenas por meio de arquivos ou mensagens.
 - Troca de mensagens.

Antecedentes

- Processos podem ser executados concorrentemente ou em paralelo.
 - Os conceitos apresentados aqui valem para processos e *threads*.
- Escalonador da CPU alterna rapidamente entre os processos para fornecer **execução concorrente**.
 - Um processo pode ser interrompido a qualquer momento em seu fluxo de instruções.
 - O núcleo de processamento pode ser designado para executar instruções de outro processo.
- Na **execução paralela**, dois fluxos de instruções são executados simultaneamente em núcleos de processamento separados.

- Execução concorrente ou paralela podem resultar em inconsistências nos dados compartilhados.
 - Manter dados consistentes exige mecanismos para garantir a execução cooperativa de processos.
- Exemplo de compartilhamento de dados:
 - Modelo do Produtor-Consumidor.
 - Relembrando o modelo Produtor-Consumidor com buffer limitado...

Dados compartilhados em uma fila circular:

```
public class BoundedBuffer implements Buffer {
    private static final int BUFFER_SIZE = 5;
    private int count; // número de itens contidos no buffer
    private int in; // aponta para a próxima posição livre
    private int out; // aponta para a primeira posição preenchida
    private Object[] buffer;

    public BoundedBuffer() {
        // criando um buffer vazio
        count = 0;
        in = 0;
        out = 0;
        buffer = new Object[BUFFER_SIZE];
    }

    // produtores executam este método
    public void insert(Object item) {
        ...
    }

    // consumidores executam este método
    public Object remove() {
        ...
    }
}
```

Produtor:

```
// produtores executam este método
public void insert(Object item) {
    while(count == BUFFER_SIZE); // buffer cheio

    // acrescenta um item ao buffer
    count++;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
}
```


Consumidor:

```
// consumidores executam este método
public Object remove() {
    while(count == 0); // buffer vazio

    // remove um item do buffer
    count--;
    Object item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    return item; // retorna o item removido
}
```

Exercício: Simule um esquema Produtor-Consumidor com buffer de tamanho 5 para as seguintes transações. Mostre a configuração do buffer ao final.

- 1 Produtor executa `insert(5);`
- 2 Produtor executa `insert(3);`
- 3 Produtor executa `insert(1);`
- 4 Consumidor executa `remove();`
- 5 Produtor executa `insert(4);`
- 6 Consumidor executa `remove();`
- 7 Consumidor executa `remove();`
- 8 Consumidor executa `remove();`
- 9 Produtor executa `insert(1);`
- 10 Produtor executa `insert(6);`

- As rotinas do produtor e do consumidor estão corretas se executadas separadamente.
 - Entretanto, elas não funcionam de forma correta quando executadas de forma concorrente.
- Suponha que o valor da variável `cont` atualmente seja 4 e que produtor e consumidor executem as instruções `cont++` e `cont--` de maneira concorrente.
 - Qual o problema?

- As rotinas do produtor e do consumidor estão corretas se executadas separadamente.
 - Entretanto, elas não funcionam de forma correta quando executadas de forma concorrente.
- Suponha que o valor da variável `cont` atualmente seja 4 e que produtor e consumidor executem as instruções `cont++` e `cont--` de maneira concorrente.
 - Qual o problema?
 - Após a execução das duas instruções, o valor de `cont` poderá ser 3, 4 ou 5.
 - O único resultado correto é `cont` ter valor igual a 4.

- Para mostrar que o valor de `cont` pode ficar incorreto, vamos relembrar o código em linguagem de máquina MIPS:

- `cont++:`

```
lw $s0, cont
addi $s0, $s0, 1
sw $s0, cont
```




- `cont--:`

```
lw $s1, cont
addi $s1, $s1, -1
sw $s1, cont
```

- Cada instrução em linguagem de máquina é independente.
 - Consequentemente, durante a execução dos trechos acima, trocas de contexto podem ocorrer na CPU.

- A execução concorrente de `cont++` e `cont--` é equivalente a uma execução sequencial na qual as instruções em linguagem de máquina desses comandos são intercaladas em alguma ordem arbitrária.
 - Porém, dentro de cada instrução de alto nível, a ordem é preservada.

- Uma possível intercalação seria:

Produtor: lw \$s0, cont	
Produtor: addi \$s0, \$s0, 1	
Consumidor: lw \$s1, cont	
Consumidor: addi \$s1, \$s1, -1	
Produtor: sw \$s0, cont	
Consumidor: sw \$s1, cont	

- Qual o valor de cont para essa sequência?
 - 3
- O valor de cont poderia ser 3 ou 5, mas o correto seria 4.
 - Essa variável não pode ser manipulada de forma concorrente!

■ **Condição de corrida:**

- Uma situação em que vários processos acessam e manipulam os mesmos dados concorrentemente e o resultado da execução depende da ordem específica em que o acesso ocorre.

■ Como proteger um código contra a condição de corrida?

- Garantir que somente um processo de cada vez possa manipular as variáveis compartilhadas.
- No exemplo anterior, um processo por vez pode manipular a variável `cont`.
- Processos precisam ser sincronizados de alguma maneira.

O Problema da Seção Crítica

O Problema da Seção Crítica

- Considere um sistema composto por n processos (P_0, P_1, \dots, P_{n-1}).
- Cada processo tem um segmento de código, chamado **seção crítica**.
 - Onde o processo pode estar alterando variáveis comuns, atualizando uma tabela, gravando um arquivo, e assim por diante.
- A característica importante do sistema é que, quando um processo está executando sua seção crítica, nenhum outro processo deve ter autorização para fazer o mesmo.
 - Dois processos não podem executar suas seções críticas ao mesmo tempo.

■ Problema da seção crítica:

- Projetar um protocolo que os processos possam usar para cooperar.
- Cada processo deve solicitar permissão para entrar em sua seção crítica.
 - **Seção de entrada.**
- A seção crítica pode ser seguida por uma **seção de saída**.
- O código restante é a **seção remanescente**.

O Problema da Seção Crítica

- Uma solução para o problema da seção crítica deve satisfazer aos três requisitos:

O Problema da Seção Crítica

- Uma solução para o problema da seção crítica deve satisfazer aos três requisitos:

- 1 Exclusão mútua:**

- Se o processo P_i está executando sua seção crítica, então nenhum outro processo pode executar sua seção crítica.

O Problema da Seção Crítica

- Uma solução para o problema da seção crítica deve satisfazer aos três requisitos:

- 1 Exclusão mútua:**

- Se o processo P_i está executando sua seção crítica, então nenhum outro processo pode executar sua seção crítica.

- 2 Progresso:**

- Se nenhum processo está executando sua seção crítica e algum processo quer entrar em sua seção crítica, então somente aqueles processos que não estão executando suas seções remanescentes podem participar da decisão de qual entrará em sua seção crítica a seguir, e essa seleção não pode ser adiada indefinidamente.

O Problema da Seção Crítica

- Uma solução para o problema da seção crítica deve satisfazer aos três requisitos:

1 Exclusão mútua:

- Se o processo P_i está executando sua seção crítica, então nenhum outro processo pode executar sua seção crítica.

2 Progresso:

- Se nenhum processo está executando sua seção crítica e algum processo quer entrar em sua seção crítica, então somente aqueles processos que não estão executando suas seções remanescentes podem participar da decisão de qual entrará em sua seção crítica a seguir, e essa seleção não pode ser adiada indefinidamente.

3 Espera limitada:

- Um processo não pode ficar indefinidamente esperando para entrar na sua seção crítica, enquanto outros processos, repetidamente, entram e saem de suas respectivas seções críticas.

Solução de Peterson

- Solução clássica baseada em software.
- Se restringe a dois processos que se alternam na execução de suas seções críticas e seções remanescentes.
- Os processos são numerados como P_0 e P_1 .
 - Por conveniência, quando apresentamos P_i , usamos P_j para representar o outro processo.
 - $j = 1 - i$

- Requer que os dois processos compartilhem os seguintes itens:

```
int turn;  
boolean flag[2];
```

- `turn` indica de quem é a vez de entrar em sua seção crítica.
 - Se `turn == i`, então o processo P_i é autorizado a executar sua seção crítica.
- `flag` é usado para indicar se um processo está pronto para entrar em sua seção crítica.
 - Se `flag[i] == true`, então P_i está pronto para entrar em sua seção crítica.

Solução de Peterson

Estrutura do processo P_i :

```
while(true) {  
    flag[i] = true;  
    turn = j;  
    while(flag[j] && turn == j);  
    seciaoCritica();  
    flag[i] = false;  
    seciaoRestante();  
}
```

- Para entrar na seção crítica, primeiro o processo P_i posiciona `flag[i]` como `true`.
- Em seguida, atribui a `turn` o valor j , assegurando que, se o outro processo quiser entrar na seção crítica, ele possa fazê-lo.
- Se os dois processos tentarem entrar ao mesmo tempo, `turn` será posicionada tanto com i quanto com j quase ao mesmo tempo.
 - Apenas uma dessas atribuições permanecerá.
 - A outra ocorrerá, mas será sobreposta imediatamente.
- O valor final de `turn` determina qual dos dois processos é autorizado a entrar primeiro em sua seção crítica.

- A solução está correta? Precisamos provar que:
 - 1 A exclusão mútua é preservada.
 - 2 O requisito de progresso é atendido.
 - 3 O requisito de espera limitada é atendido.
- Provando a preservação da **exclusão mútua**:
 - Para que P_i acesse a região crítica, $\text{flag}[j] == \text{false}$ ou $\text{turn} == i$.
 - Se os dois processos puderem executar suas seções críticas ao mesmo tempo, então $\text{flag}[i] == \text{flag}[j] == \text{true}$.
 - Desempate é feito pela variável turn , que só pode ser 0 ou 1, mas não ambos.

- Provando os requisitos de **progresso** e **espera limitada**:
 - P_i pode ser impedido de entrar na seção crítica somente se ficar preso no loop while ($\text{flag}[j] == \text{true}$ e $\text{turn} == j$).
 - Se P_j não estiver pronto para entrar na seção crítica, então $\text{flag}[j] == \text{false}$, e P_i pode entrar em sua seção crítica.
 - Se P_j posicionou $\text{flag}[j]$ como true e também está executando seu comando while, então $\text{turn} == i$ ou $\text{turn} == j$.
 - Se $\text{turn} == i$, então P_i entrará na seção crítica.
 - Se $\text{turn} == j$, então P_j entrará na seção crítica.
 - No entanto, quando P_j sair de sua seção crítica, ele posicionará $\text{flag}[j]$ como false, permitindo que P_i entre em sua seção crítica.
 - Se P_j posicionar $\text{flag}[j]$ como true, também deve atribuir a turn o valor i .
 - Assim, já que P_i não altera o valor da variável turn enquanto executa o comando while, P_i entrará na seção crítica (**progresso**) após no máximo uma entrada de P_j (**espera limitada**).

- Por causa da maneira como as arquiteturas de computador modernas executam instruções básicas de linguagem de máquina, como *load* e *store*, não há garantias de que a solução de Peterson funcione corretamente nessas arquiteturas.

Hardware de Sincronização

- Ambiente com um único processador:
 - Impedir que interrupções ocorram enquanto uma variável compartilhada estivesse sendo modificada.
 - A sequência de instruções corrente seria autorizada a executar em ordem e sem preempção.
 - Nenhuma outra instrução seria executada e, portanto, nenhuma modificação inesperada poderia ser feita na variável compartilhada.
 - Abordagem usada por kernels preemptivos.

- Essa solução não é tão viável em um ambiente multiprocessador.
 - A desabilitação de interrupções em um multiprocessador pode ser demorada.
 - A mensagem deve ser passada a todos os processadores.
 - Atrasa a entrada em cada seção crítica e a eficiência do sistema diminui.
 - Problemas com o relógio de um sistema se ele for mantido atualizado por interrupções.

- Muitos sistemas de computação modernos fornecem instruções de hardware especiais.
 - Permitem testar e modificar o conteúdo de uma palavra ou permutar os conteúdos de duas palavras **atomicamente**.
 - Como uma unidade impossível de interromper.
 - Se duas instruções forem executadas simultaneamente, elas serão executadas sequencialmente em alguma ordem arbitrária.
- Podemos usar essas instruções especiais para resolver o problema da seção crítica.
- Em vez de discutir uma instrução específica de determinada máquina, abstraímos os principais conceitos existentes por trás desses tipos de instruções descrevendo os métodos `get-and-set` e `swap`.

Hardware de Sincronização

```
public class HardwareData {
    private boolean value;

    public HardwareData(boolean value) {
        this.value = value;
    }

    public boolean get() {
        return this.value;
    }

    public void set(boolean newValue) {
        this.value = newValue;
    }

    public boolean getAndSet(boolean newValue) {
        boolean oldValue = this.value;
        this.value = newValue;

        return oldValue;
    }

    public void swap(HardwareData other) {
        boolean temp = this.value;

        this.value = other.value;
        other.value = temp;
    }
}
```

Instrução get-and-set

- Também conhecida como test-and-set.
- Instrução atômica.
- Se o hardware admitir a instrução get-and-set, podemos implementar a exclusão mútua:
 - Criando um objeto da classe HardwareData e inicializando-o como *false*.
 - Todas as threads devem compartilhar esse objeto.

Instrução get-and-set

- Objeto compartilhado:

HardwareData lock = new HardwareData(false);

- Código de cada thread:

```
while (true) {  
    while (lock.getAndSet(true)) {  
        Thread.yield();  
    }  
  
    criticalSection();  
    lock.set(false);  
    remainderSection();  
}
```

- Outra solução baseada em hardware é utilização da instrução `swap`:
 - Definida no método `swap()`, ela opera sobre o conteúdo de duas palavras.
 - Assim como `get-and-set`, ela é executada atomicamente.
- Se o hardware admitir a instrução `swap`, então a exclusão mútua pode ser fornecida da seguinte maneira:
 - Todas as threads compartilham o objeto `lock`, da classe `HardwareData`, que é inicializado como *false*.
 - Cada thread possui um objeto `HardwareData` local, chamado `key`, inicializado como *true*.

Instrução swap

- Objeto compartilhado:

HardwareData lock = new HardwareData(false);

- Código de cada thread:

```
HardwareData key = new HardwareData(true);

while (true) {
    key.set(true);

    do {
        lock.swap(key);
    } while (key.get());

    criticalSection();
    lock.set(false);
    remainderSection();
}
```


- As soluções anteriores satisfazem a exclusão mútua, mas não satisfazem ao requisito da espera limitada.
 - O processo que acabou de executar sua seção crítica pode novamente executá-la.
 - Dependendo da forma de escalonamento, um processo pode sempre executar sua seção crítica enquanto os outros esperam.
- Ao final da seção 5.4 do livro “Fundamentos de Sistemas Operacionais” é apresentada uma solução utilizando get-and-set que satisfaz a todos requisitos da seção crítica.

Locks Mutex

- Soluções baseadas em hardware para o problema da seção crítica são complicadas e geralmente inacessíveis aos programadores de aplicações.
- Como alternativa, os projetistas de SOs constroem ferramentas de software para resolver o problema da seção crítica.
- A mais simples dessas ferramentas é o **lock mutex**.
 - *Mutual exclusion*

- Usamos o *lock mutex* para proteger regiões críticas e, assim, evitar condições de corrida.
- Um processo deve adquirir o *lock* antes de entrar em uma seção crítica.
 - Ele libera o *lock* quando sai da seção crítica.

```
while (true) {  


adquire lock

  
    seção crítica  


libera lock

  
    seção remanescente  
}
```

- Um *lock mutex* tem uma variável *booleana*.
 - Indica se o *lock* está ou não disponível.
- A função `acquire()` adquire o *lock* e a função `release()` o libera.
- Se o *lock* está disponível, uma chamada a `acquire()` é bem-sucedida.
 - *lock* é então considerado indisponível.
- Um processo que tente adquirir um *lock* indisponível é bloqueado até que o *lock* seja liberado.

Locks Mutex

Definição de um *lock mutex*:

```
public class Mutex {  
    private boolean available;  
  
    public Mutex(boolean available) {  
        this.available = available;  
    }  
  
    public void acquire() {  
        while (!available); //aguardando  
  
        available = false;  
    }  
  
    public void release() {  
        available = true;  
    }  
}
```

Importante: os métodos `acquire()` e `release()` devem ser atômicos!

- Chamadas a `acquire()` ou `release()` devem ser executadas atomicamente.
 - *Locks mutex* são, com frequência, implementados com o uso de um dos mecanismos de hardware descritos anteriormente.

- Qual a principal desvantagem da implementação apresentada?

- Qual a principal desvantagem da implementação apresentada?
 - **Espera ocupada** (ou espera em ação).
 - Enquanto um processo está em sua seção crítica, qualquer outro processo que tente entrar em sua seção crítica deve entrar em um loop contínuo na chamada a `acquire()`.
 - Também é conhecido como ***spinlock***.
 - Processo “gira” (*spin*) enquanto espera que o *lock* se torne disponível.
- Mesmo problema visto na utilização das instruções `get-and-set` e `swap`.
 - E também nos métodos `insert` e `remove` no problema do Produtor-Consumidor.

- Esse looping contínuo é um problema em um sistema de multiprogramação real com uma única CPU.
 - Desperdiça ciclos da CPU que algum outro processo poderia usar produtivamente.
- *Spinlocks* apresentam uma vantagem:
 - Nenhuma mudança de contexto é requerida quando um processo tem de esperar em um *lock*.
 - Uma mudança de contexto pode levar um tempo considerável.
 - São úteis quando o uso de *locks* é esperado por períodos curtos.
 - Costumam ser empregados em sistemas multiprocessadores em que um thread pode “desenvolver um spin” em um processador enquanto outro thread executa sua seção crítica em outro processador.

Semáforos

- Um **semáforo** contém uma variável inteira que, exceto na inicialização, é acessada apenas por meio de duas operações atômicas padrão:
 - `acquire()` e `release()`.
 - Inicialmente conhecidas como `P()` e `V()`.
 - Também encontradas na literatura como `down()` e `up()` ou `wait()` e `signal()`.

Semáforos

Definição de um semáforo:

```
public class Semaphore {  
    private int value;  
  
    public Semaphore(int value) {  
        this.value = value;  
    }  
  
    public void acquire() {  
        while (value <= 0); //aguardando  
  
        value--;  
    }  
  
    public void release() {  
        value++;  
    }  
}
```

Importante: os métodos `acquire()` e `release()` devem ser atômicos!

- Todas as modificações do valor inteiro do semáforo nas operações `acquire()` e `release()` devem ser executadas indivisivelmente.
 - Quando um processo modifica o valor do semáforo, nenhum outro processo pode modificar o valor desse mesmo semáforo simultaneamente.

- Dois tipos de semáforos:

- contador** : o valor inteiro do semáforo pode variar por um domínio irrestrito.

- binário** : o valor inteiro do semáforo só pode variar entre 0 e 1. Semelhante ao *lock mutex*.

- Pode ser usado para controlar o acesso a determinado recurso composto por um número finito de instâncias.
- Inicializado com o número de recursos disponíveis.
- Cada processo que deseja usar um recurso executa uma operação `acquire()` no semáforo.
- Quando um processo libera um recurso, ele executa uma operação `release()`.
- Quando a contagem do semáforo chega a zero, todos os recursos estão sendo usados.
 - Processos que queiram usar um recurso ficarão bloqueados até a contagem se tornar maior do que zero.

- Também podemos usar semáforos para resolver vários problemas de sincronização.
 - Exemplo: Considere dois processos sendo executados concorrentemente: P_1 com um comando S_1 e P_2 com um comando S_2 . Suponha que S_2 deve ser executado somente após S_1 ser concluído. Como resolver?

- Também podemos usar semáforos para resolver vários problemas de sincronização.
 - Exemplo: Considere dois processos sendo executados concorrentemente: P_1 com um comando S_1 e P_2 com um comando S_2 . Suponha que S_2 deve ser executado somente após S_1 ser concluído. Como resolver?
 - Solução: P_1 e P_2 compartilham um semáforo *sinc*, inicializado com 0.

```
Processo P1:  
S1();  
sinc.release();
```

```
Processo P2:  
sinc.acquire();  
S2();
```

Implementação do Semáforo

- A implementação de semáforo apresentada também possui a **espera ocupada** (*busy wating*).
- Para eliminar a necessidade da espera em ação, devemos modificar a definição das operações `acquire()` e `release()`.
- Quando um processo executa `acquire()` e descobre que o valor do semáforo não é positivo, ele deve esperar.
 - Em vez de entrar na espera em ação, o processo pode bloquear a si próprio.
 - A operação de bloqueio insere o processo em uma fila de espera associada ao semáforo, e o estado do processo é comutado para o estado de espera.
 - Em seguida, o controle é transferido ao scheduler da CPU, que seleciona outro processo para execução.

Implementação do Semáforo

- Um processo que está bloqueado, esperando em um semáforo deve ser “desbloqueado” quando algum outro processo executar uma operação `release()`.
 - O processo é reiniciado por uma operação `wakeup()` que o passa do estado de espera para o estado de pronto.
 - O processo é então inserido na fila de prontos.
- Para essa implementação, um semáforo deve possuir:
 - Um valor inteiro.
 - Uma lista de processos.

Implementação do Semáforo

```
public void acquire() {  
    value--;  
  
    if (value < 0) {  
        adiciona o processo à lista;  
        block();  
    }  
}  
  
public void release() {  
    value++;  
  
    if (value <= 0) {  
        remove um processo P da lista  
        wakeup(P);  
    }  
}
```

block() suspende o processo que a invocou.

wakeup(P) retoma a execução de um processo bloqueado *P*.

Ambas são fornecidas pelo SO como chamadas de sistema.

Implementação do Semáforo

- O valor da variável inteira do semáforo pode ter valor negativo.
 - Embora esse valor nunca seja negativo sob a definição clássica.

Implementação do Semáforo

- O valor da variável inteira do semáforo pode ter valor negativo.
 - Embora esse valor nunca seja negativo sob a definição clássica.
- Se esse valor for negativo, sua magnitude é a quantidade de processos esperando por esse semáforo.
- A lista de processos pode ser implementada por um link em cada bloco de controle de processo (PCB).
 - Lista de PCBs.

Implementação do Semáforo

- É vital que as operações de semáforo sejam executadas atomicamente.
 - Dois processos não podem executar operações `acquire()` e `release()` no mesmo semáforo ao mesmo tempo.
 - Problema de seção crítica.
- Em um ambiente com um único processador, podemos inibir interrupções durante o tempo em que essas operações estejam sendo executadas.
- Em um ambiente multiprocessador, desabilitar as interrupções pode ser uma tarefa difícil e pode piorar seriamente o desempenho.
 - Portanto, sistemas SMP devem fornecer técnicas alternativas de uso do *lock* para assegurar que `acquire()` e `release()` sejam executadas atomicamente.
 - Instruções *get-and-set* e *swap* ou *spinlocks*, por exemplo.

- A implementação de um semáforo com fila de espera pode resultar em uma situação em que dois ou mais processos fiquem esperando indefinidamente por um evento que pode ser causado somente por um dos processos em espera.
 - O evento em questão é a execução de uma operação `release()`.
- Quando tal estado é alcançado, dizemos que esses processos estão em ***deadlock***.

Deadlocks e Inanição



■ Exemplo:

- Considere dois processos P_0 e P_1 , cada um acessando dois semáforos S e Q , ambos inicializados com valor 1.

P_0	P_1
$S.acquire();$	$Q.acquire();$
$Q.acquire();$	$S.acquire();$
\cdot	\cdot
\cdot	\cdot
\cdot	\cdot
$S.release();$	$Q.release();$
$Q.release();$	$S.release();$

- Suponha que P_0 execute `S.acquire()` e depois P_1 execute `Q.acquire()`.
- Quando P_0 executar `Q.acquire()`, ele terá de esperar até P_1 executar `Q.release()`.
- De modo semelhante, quando P_1 executar `S.acquire()`, ele terá de esperar até P_0 executar `S.release()`.
- Como essas operações `release()` não podem ser executadas, P_0 e P_1 estão em **deadlock**.

- Dizemos que um conjunto de processos está em estado de **deadlock** quando cada processo do conjunto está esperando por um evento que só pode ser causado por outro processo do conjunto.
- Outro problema relacionado com os *deadlocks* é o **bloqueio indefinido** (inanição ou *starvation*).
 - Situação em que os processos esperam indefinidamente dentro do semáforo.
 - Pode ocorrer se removermos processos da lista associada a um semáforo em ordem LIFO (último a entrar, primeiro a sair).

Problemas Clássicos de Sincronização

- Alguns problemas clássicos:
 - Problema do buffer limitado
 - Problema dos filósofos comensais
- São usados para testar quase todo esquema de sincronismo recém-proposto.
- As soluções vistas aqui utilizarão semáforos.

O Problema do Buffer Limitado

- O problema do buffer limitado (Produtor-Consumidor com Buffer Limitado) foi implementado anteriormente.
 - Produtores inserem itens no buffer e consumidores removem.
- Um problema dessa abordagem era que as operações sobre a variável `cont` deveriam ser executadas em uma seção crítica.
 - Caso contrário, o valor de `cont` dependeria da ordem de escalonamento das instruções.

O Problema do Buffer Limitado

Solução:

- Utilização de três semáforos.
 - Semáforo `mutex` provê a exclusão mútua para os acessos ao buffer e é inicializado com 1.
 - Semáforos `empty` e `full` contam o número de posições vazias e preenchidas, respectivamente.
 - `empty` é inicializado com a capacidade do buffer.
 - `full` é inicializado com 0.
- Alteração nas definições dos métodos `insert` e `remove`.

O Problema do Buffer Limitado

```
public class BoundedBuffer implements Buffer
{
    private static final int BUFFER_SIZE = 5;
    private Object[] buffer;
    private int in, out;
    private Semaphore mutex;
    private Semaphore empty;
    private Semaphore full;

    public BoundedBuffer() {
        // buffer is initially empty
        in = 0;
        out = 0;
        buffer = new Object[BUFFER_SIZE];

        mutex = new Semaphore(1);
        empty = new Semaphore(BUFFER_SIZE);
        full = new Semaphore(0);
    }

    public void insert(Object item) {
        ver slide 68
    }

    public Object remove() {
        ver slide 69
    }
}
```

O Problema do Buffer Limitado

```
public void insert(Object item) {  
    empty.acquire();  
    mutex.acquire();  
  
    // add an item to the buffer  
    buffer[in] = item;  
    in = (in + 1) % BUFFER_SIZE;  
  
    mutex.release();  
    full.release();  
}
```

O Problema do Buffer Limitado

```
public Object remove() {  
    full.acquire();  
    mutex.acquire();  
  
    // remove an item from the buffer  
    Object item = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
  
    mutex.release();  
    empty.release();  
  
    return item;  
}
```

O Problema do Buffer Limitado

```
public class Producer implements Runnable
{
    private Buffer buffer;

    public Producer(Buffer buffer) {
        this.buffer = buffer;
    }

    public void run() {
        Date message;

        while (true) {
            // nap for awhile
            SleepUtilities.nap();
            // produce an item & enter it into the buffer
            message = new Date();
            buffer.insert(message);
        }
    }
}
```

O Problema do Buffer Limitado

```
public class Consumer implements Runnable
{
    private Buffer buffer;

    public Consumer(Buffer buffer) {
        this.buffer = buffer;
    }

    public void run() {
        Date message;

        while (true) {
            // nap for awhile
            SleepUtilities.nap();
            // consume an item from the buffer
            message = (Date)buffer.remove();
        }
    }
}
```

O Problema do Buffer Limitado

```
public class Factory
{
    public static void main(String args[]) {
        Buffer buffer = new BoundedBuffer();

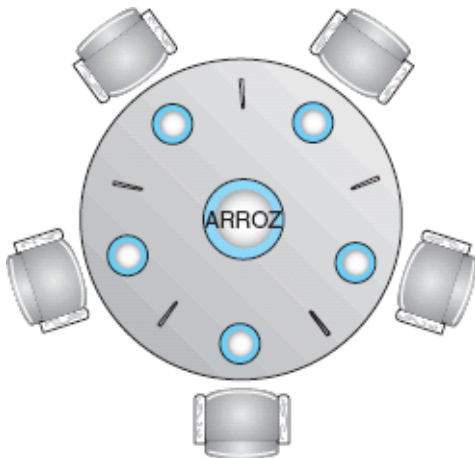
        // now create the producer and consumer threads
        Thread producer = new Thread(new Producer(buffer));
        Thread consumer = new Thread(new Consumer(buffer));

        producer.start();
        consumer.start();
    }
}
```

O Problema dos Filósofos Comensais

- Considere cinco filósofos que passam suas vidas pensando e comendo.
- Eles compartilham uma mesa redonda, cercada por cinco cadeiras, cada uma pertencendo a um filósofo.
- No centro da mesa existe uma tigela de arroz, e a mesa está disposta com apenas cinco hashis.

O Problema dos Filósofos Comensais



O Problema dos Filósofos Comensais

- Quando um filósofo pensa, ele não interage com seus colegas.
- Periodicamente, um filósofo fica com fome e tenta pegar os dois hashis que estão mais próximos dele.
 - Os hashis que estão entre ele e seus vizinhos da esquerda e da direita.
- Um filósofo pode pegar somente um hashi de cada vez.
 - É claro que ele não pode pegar um hashi que já esteja na mão de um vizinho.
- Quando um filósofo faminto está com seus dois hashis ao mesmo tempo, ele come sem largá-los.
 - Quando termina de comer, larga seus dois hashis e começa a pensar novamente.

O Problema dos Filósofos Comensais

- O **Problema dos Filósofos Comensais** é considerado um problema clássico de sincronismo.
- É um exemplo de uma grande classe de problemas de controle de concorrência.
 - Representação simples da necessidade de alocar vários recursos entre diversos processos de uma forma livre de deadlocks e de starvation.

Solução?

Solução?

- Cada hashi é representado por um semáforo.
- Um filósofo tenta pegar um hashi executando uma operação `acquire()` sobre esse semáforo.
- Ele solta o hashi executando a operação `release()`.
- Dados compartilhados:
`Semaphore chopStick[] = new Semaphore[5];`
(todos os elementos são inicializados com 1)

Código do filósofo i :

```
while (true) {  
    // get left chopstick  
    chopStick[i].acquire();  
    // get right chopstick  
    chopStick[(i + 1) % 5].acquire();  
  
    eating();  
  
    // return left chopstick  
    chopStick[i].release();  
    // return right chopstick  
    chopStick[(i + 1) % 5].release();  
  
    thinking();  
}
```

O Problema dos Filósofos Comensais

- Suponha que todos os 5 filósofos fiquem com fome ao mesmo tempo e cada um apanha o hashi à sua esquerda.
 - O que acontece?

O Problema dos Filósofos Comensais

- Suponha que todos os 5 filósofos fiquem com fome ao mesmo tempo e cada um apanha o hashi à sua esquerda.
 - O que acontece?
 - Todos os hashis estarão em uso (um com cada filósofo).

O Problema dos Filósofos Comensais

- Suponha que todos os 5 filósofos fiquem com fome ao mesmo tempo e cada um apanha o hashi à sua esquerda.
 - O que acontece?
 - Todos os hashis estarão em uso (um com cada filósofo).
- Quando cada filósofo tentar pegar o hashi da direita, ele esperará indefinidamente.
- A solução anterior garante que dois filósofos vizinhos não comerão simultaneamente.
 - Porém ela deve ser rejeitada, pois pode gerar um **deadlock**!

Soluções para o deadlock?

Soluções para o deadlock?

- Permitir que, no máximo, quatro filósofos sentem simultaneamente à mesa.
- Permitir que um filósofo apanhe os hashis somente se os dois hashis estiverem disponíveis.
 - Seção crítica!
- Usar uma solução assimétrica.
 - Um filósofo ímpar pega primeiro o hashi da esquerda e depois o da direita.
 - Um filósofo par pega primeiro o hashi da direita e depois o da esquerda.

Soluções para o deadlock?

- Permitir que, no máximo, quatro filósofos sentem simultaneamente à mesa.
- Permitir que um filósofo apanhe os hashis somente se os dois hashis estiverem disponíveis.
 - Seção crítica!
- Usar uma solução assimétrica.
 - Um filósofo ímpar pega primeiro o hashi da esquerda e depois o da direita.
 - Um filósofo par pega primeiro o hashi da direita e depois o da esquerda.
- Essas soluções impedem o deadlock colocando restrições sobre os filósofos.
 - Porém, não impedem que um filósofo morra de fome (starvation).
- Solução livre de deadlock não elimina necessariamente a possibilidade de starvation!

Monitores

- A utilização incorreta de semáforos pode resultar em erros de temporização difíceis de detectar.
 - Erros só acontecem se ocorrerem determinadas sequências de execução.
 - Essas sequências nem sempre ocorrem.
- Rever os problemas apresentados nos slides 60 e 61.

- Outros exemplos:

- Suponha que um processo troque a ordem em que são executadas as operações `acquire()` e `release()` sobre o semáforo `mutex`:

```
mutex.release();  
...  
seção crítica  
...  
mutex.acquire();
```

- O que aconteceria nessa situação?

■ Outros exemplos:

- Suponha que um processo troque a ordem em que são executadas as operações `acquire()` e `release()` sobre o semáforo `mutex`:

```
mutex.release();  
...  
seção crítica  
...  
mutex.acquire();
```

- O que aconteceria nessa situação?
 - Vários processos podem estar executando em suas seções críticas concorrentemente, violando o requisito de exclusão mútua.
 - Esse erro só pode ser descoberto se vários processos estiverem ativos de forma simultânea em suas seções críticas.
 - Essa situação pode ser de difícil reprodução.

- Outros exemplos:

- Suponha que um processo substitua `mutex.release()` por `mutex.acquire()`:

```
mutex.acquire();  
...  
seção crítica  
...  
mutex.acquire();
```

- O que aconteceria nessa situação?

- Outros exemplos:

- Suponha que um processo substitua `mutex.release()` por `mutex.acquire()`:

```
mutex.acquire();  
...  
seção crítica  
...  
mutex.acquire();
```

- O que aconteceria nessa situação?
 - Deadlock!

- Outros exemplos:
 - Suponha que um processo omita o `mutex.acquire()` ou o `mutex.release()` ou ambos.
 - O que aconteceria nessas situações?

- Outros exemplos:
 - Suponha que um processo omita o `mutex.acquire()` ou o `mutex.release()` ou ambos.
 - O que aconteceria nessas situações?
 - Ou a exclusão mútua é violada ou haverá deadlock.

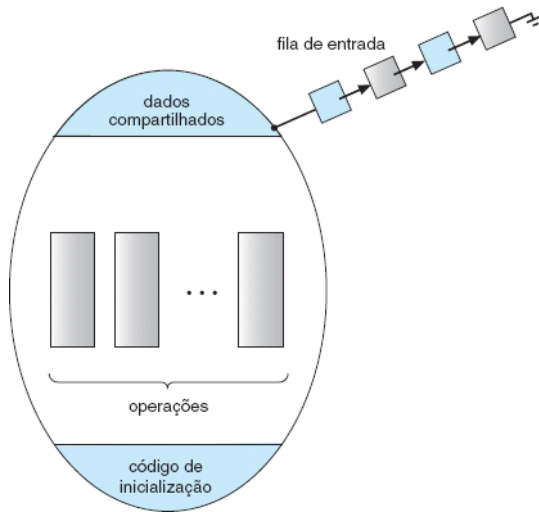
- Os exemplos anteriores ilustram que vários tipos de erros podem ser gerados com facilidade quando os programadores utilizam semáforos incorretamente para resolver o problema da seção crítica.
- Para lidar com tais erros, os pesquisadores desenvolveram construções de linguagem de alto nível.
 - Por exemplo, **monitores**.

- Um **tipo abstrato de dados** (TAD), encapsula dados com um conjunto de métodos para operar sobre esses dados.
- Um **tipo monitor** é um TAD que inclui um conjunto de operações definidas pelo programador e que são dotadas de exclusão mútua dentro do monitor.
- O tipo monitor também declara as variáveis cujos valores definem o estado de uma instância desse tipo, além dos corpos de funções que operam sobre essas variáveis.

Monitores

```
monitor nomeDoMonitor {  
    /* declarações de variáveis compartilhadas */  
  
    código de inicialização (...) {  
        ...  
    }  
  
    função P1 (...) {  
        ...  
    }  
  
    função P2 (...) {  
        ...  
    }  
  
    ...  
  
    função PN (...) {  
        ...  
    }  
}
```

Monitores

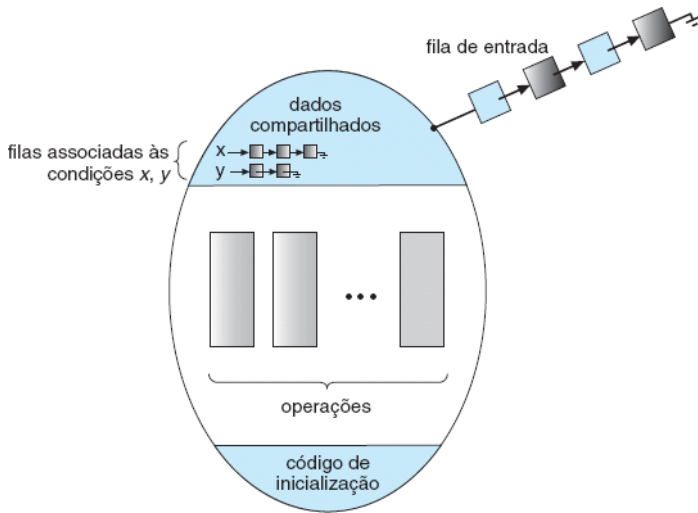


- A construção do monitor assegura que somente um processo pode estar ativo dentro do monitor por vez.
 - Programador não precisa codificar essa restrição de sincronismo explicitamente.
- Entretanto, podemos precisar definir mecanismos de sincronismo adicionais.

- Um programador que precisa escrever seu próprio esquema de sincronismo personalizado pode definir uma ou mais variáveis do tipo `Condition`.
- Exemplo:
`Condition x, y;`
- As únicas operações que podem ser chamadas em uma variável de condição são `wait()` e `signal()`.

- Operação `wait()`
 - O processo que a chama é suspenso.
- Operação `signal()`
 - Retoma um dos processos suspensos (se houver).
 - Se não há nenhum processo suspenso, a chamada de `signal()` não tem efeito.

Monitor com Variáveis Condicionais



Solução para os Filósofos Comensais

- São utilizados 3 estados para descrever um filósofo:
 - Pensando (THINKING), faminto (HUNGRY) e comendo (EATING)

```
enum State {THINKING, HUNGRY, EATING}  
State[] states = new State[5];
```
- O filósofo *i* só pode definir seu estado para EATING se seus dois vizinhos não estiverem comendo:

```
state[(i+4)%5] != State.EATING &&  
state[(i+1)%5] != State.EATING
```

- Também precisamos declarar:

```
Condition[] self = new Condition[5];
```

- onde o filósofo *i* entra em estado de espera quando estiver com fome mas não consegue obter os hashis de que precisa.
- A distribuição de hashis é controlada pelo monitor *dp*, que é uma instância do tipo de monitor *DiningPhilophers* (próximo slide).

Solução para os Filósofos Comensais

```
monitor DiningPhilosophers
{
    enum State {THINKING, HUNGRY, EATING};
    State[] states = new State[5];
    Condition[] self = new Condition[5];

    public DiningPhilosophers {
        for (int i = 0; i < 5; i++)
            state[i] = State.THINKING;
    }

    public void takeForks(int i) {
        state[i] = State.HUNGRY;
        test(i);
        if (state[i] != State.EATING)
            self[i].wait;
    }

    public void returnForks(int i) {
        state[i] = State.THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    private void test(int i) {
        if ( (state[(i + 4) % 5] != State.EATING) &&
            (state[i] == State.HUNGRY) &&
            (state[(i + 1) % 5] != State.EATING) ) {
            state[i] = State.EATING;
            self[i].signal;
        }
    }
}
```

Solução para os Filósofos Comensais

- Cada filósofo, antes de começar a comer, precisa chamar a operação `takeForks()`.
 - Pode resultar na suspensão da thread do filósofo, se ele não consegue apanhar os dois hashis.
- Se a operação for bem-sucedida, o filósofo poderá comer.
- Ao terminar de comer, o filósofo chama a operação `returnForks()` e começa a pensar.
 - Passa a vez para outro filósofo que esteja faminto.

Solução para os Filósofos Comensais

- Sequência de chamadas para a solução do problema:

```
DiningPhilosophers dp = new DiningPhilosophers();  
...  
dp.takeForks(i);  
eat();  
dp.returnForks(i);  
...
```

- Essa solução garante que dois filósofos vizinhos não estarão comendo simultaneamente e que não haverá deadlocks.
 - Problema?

Solução para os Filósofos Comensais

- Sequência de chamadas para a solução do problema:

```
DiningPhilosophers dp = new DiningPhilosophers();  
...  
dp.takeForks(i);  
eat();  
dp.returnForks(i);  
...
```

- Essa solução garante que dois filósofos vizinhos não estarão comendo simultaneamente e que não haverá deadlocks.
 - Problema?
 - Um filósofo pode morrer de fome!
 - Soluções?

Conclusões

- Qual a melhor forma de resolver os problemas de sincronismo de processos?

- Qual a melhor forma de resolver os problemas de sincronismo de processos?
 - Todas são funcionalmente equivalentes.
- É possível implementar um semáforo por meio de um monitor (ou solução de Peterson, etc.).
- É possível implementar um monitor por meio de um semáforo (ou solução de Peterson, etc.).

Material Complementar: Sincronismo em Java

- Java oferece sincronismo em nível de linguagem para tratar condições de corrida.
 - Métodos ***synchronized***.
- Cada objeto Java possui um *lock* associado.
 - O *lock* pode estar em posse de uma única thread.
 - Para executar um método sincronizado é preciso obter o *lock* do objeto.
 - Se o *lock* já estiver em posse de outra thread, a thread que chama o método sincronizado é bloqueada e colocada no conjunto de entrada (*entry set*) do *lock* do objeto.
 - Se o *lock* estiver disponível quando um método sincronizado for chamado, a thread que chama se torna proprietária do *lock* do objeto e pode executar o método.
 - O *lock* é liberado quando a thread finaliza a execução do método.
 - Se o conjunto de entrada do lock não estiver vazio quando o lock for liberado, a JVM seleciona arbitrariamente uma thread desse conjunto para ser a proprietária do lock.

Sincronismo em Java

Métodos sincronizados `insert()` e `remove()`, para o problema do Produtor-Consumidor com buffer limitado:

```
public synchronized void insert(Object item) {  
    while (count == BUFFER_SIZE)  
        Thread.yield();  
  
    ++count;  
    buffer[in] = item;  
    in = (in + 1) % BUFFER_SIZE;  
}
```

```
public synchronized Object remove() {  
    Object item;  
  
    while (count == 0)  
        Thread.yield();  
  
    --count;  
    item = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
  
    return item;  
}
```


- Se o produtor chamar o método `insert()` e o *lock* do objeto estiver disponível, o produtor se torna proprietário do *lock*.
 - Ele pode entrar no método e alterar os valores de `count` e outros dados compartilhados.
- Se o consumidor tentar chamar o método `remove()` enquanto o produtor tem a posse do *lock*, o consumidor será bloqueado.
 - Quando o produtor sai do método `insert()`, ele libera o *lock*.
 - O consumidor agora pode obter o *lock* e entrar no método `remove()`.

- O código anterior resolve o problema da condição de corrida, porém leva a outro problema.
 - Suponha que o buffer esteja cheio e o consumidor esteja dormindo.
 - Se o produtor chamar o método `insert()`, ele pode continuar, pois o *lock* estará disponível.
 - Quando o produtor invoca o método `insert()`, ele vê que o buffer está cheio e executa o método `yield()`.
 - Em todo tempo, o produtor ainda tem a posse do *lock* do objeto.
 - Quando o consumidor desperta e tenta chamar o método `remove()`, ele é bloqueado, pois não tem a posse do *lock* do objeto.
 - Produtor e consumidor são incapazes de prosseguir porque o produtor está bloqueado esperando que o consumidor libere espaço no buffer e o consumidor está bloqueado esperando que o produtor libere o *lock*.

- Solução para o deadlock anterior:
 - `wait` e `notify`.
- Quando uma thread invoca `wait()`:
 - A thread libera o *lock* do objeto.
 - O estado da thread é definido como bloqueado.
 - A thread é colocada no conjunto de espera (*wait set*) do objeto.
- Quando um thread invoca `notify()`:
 - Uma thread qualquer T do conjunto de espera é selecionada.
 - T é movida da espera para o conjunto de entrada.
 - O estado de T é definido como executável.

Wait e Notify

```
public synchronized void insert(Object item) {
    while (count == BUFFER_SIZE) {
        try {
            wait();
        }
        catch (InterruptedException e) { }
    }

    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;

    notify();
}

public synchronized Object remove() {
    Object item;

    while (count == 0) {
        try {
            wait();
        }
        catch (InterruptedException e) { }
    }

    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    notify();

    return item;
}
```

- A utilização do método `notify()` funciona bem quando somente uma thread está no conjunto de espera.
 - Quando existem várias threads no conjunto de espera e mais de uma condição para esperar, é possível que a thread cuja condição ainda não tenha sido atendida seja a que receberá a notificação.
 - Como a chamada de `notify()` apanha uma única thread de forma aleatória no conjunto de espera, o desenvolvedor não tem controle sobre qual thread será escolhida.

- Java fornece um mecanismo que permite a notificação a todas as threads no conjunto de espera.
 - Método `notifyAll()`.
 - Semelhante ao `notify()`, exceto que cada thread em espera é removida do conjunto de espera e colocada no conjunto de entrada.
 - Operação mais dispendiosa do que `notify()`.
 - Mais apropriada para situações em que várias threads podem estar no conjunto de espera de um objeto.

Sincronismo em Bloco

Em vez de sincronizar um método inteiro, blocos de código podem ser declarados como sincronizados:

```
Object mutexLock = new Object();  
.  
.  
.  
public void someMethod() {  
    nonCriticalSection();  
  
    synchronized(mutexLock) {  
        criticalSection();  
    }  
  
    remainderSection();  
}
```

Sincronismo em Bloco

Também podemos usar os métodos `wait()` e `notify()` em um bloco sincronizado:

```
Object mutexLock = new Object();  
.  
.  
.  
synchronized(mutexLock) {  
    try {  
        mutexLock.wait();  
    }  
    catch (InterruptedException ie) { }  
}  
  
synchronized(mutexLock) {  
    mutexLock.notify();  
}
```

Dúvidas?

