# Sistemas Operacionais

**Professor: Cristiano Bonato Both**

# Sumário

- Programação concorrente
  - Propriedade para exclusão mútua
  - Mutex
  - Semáforos
  - Condições para ocorrência de deadlocks
- Referência

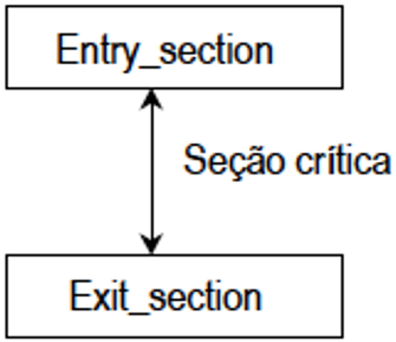
# Relembrando aulas passadas

- Corrida (*race condition*)
  - Situação que ocorre quando vários processos manipulam o mesmo conjunto de dados concorrentemente e o resultado depende da ordem em que os acessos são feitos
- Seção crítica
  - Segmento de código no qual um processo realiza a alteração de um recurso compartilhado

# Demo de Race Condition

# Necessidade de programação concorrente

- Eliminar corridas
- Criação de um protocolo para permitir que processos possam cooperar sem afetar a consistência dos dados
- Controle de acesso a seção crítica
  - Garantir a exclusão mútua



# Propriedades do problema de seção crítica

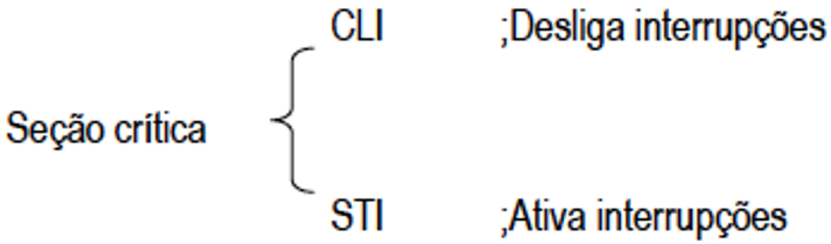
- Regra 1 – Exclusão mútua
  - Dois ou mais processos não podem estar simultaneamente em uma seção crítica
- Regra 2 – Progressão
  - Nenhum processo fora da seção crítica pode bloquear a execução de um outro processo
- Regra 3 – Espera limitada
  - Nenhum processo deve esperar infinitamente para entrar em uma seção crítica
- Regra 4 - Processamento
  - Não fazer considerações sobre o número de processadores, nem de suas velocidades relativas

# Obtenção da exclusão mútua

- Desabilitar as interrupções
- Variáveis especiais do tipo *lock*
- Alternância de execução

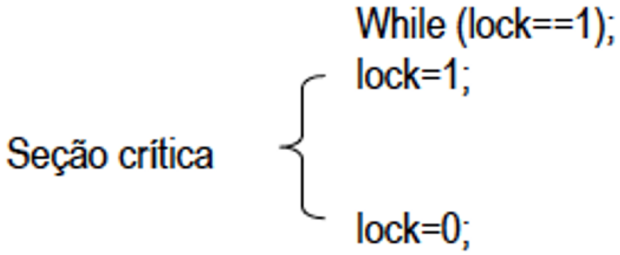
# Desabilitar as interrupções

- Não há troca de processos com a ocorrência de interrupções de tempo ou de eventos externos
- Desvantagens
  - Poder demais para um usuário
  - Não funciona em máquinas multiprocessadas (SMP), pois apenas a CPU que realiza a instrução é afetada



# Variáveis do tipo *lock*

- Criação de uma variável especial compartilhada que armazena dois estados:
  - Zero: livre
  - Um: ocupado
- Desvantagem:
  - Apresenta *race conditions*



# Alternância

- Desvantagem
  - Teste contínuo do valor da variável compartilhada provoca o desperdício do tempo do processador *(busy waiting)*
  - Viola a regra 2, se a parte não crítica de um processo for maior que a do outro

```
while (TRUE) {
   while (turn!=0);
       critical_section();
   turn=1;
   non_critical_section();
}
```

```
while (TRUE) {
   while (turn!=1);
       critical_section();
   turn=0;
   non_critical_section();
}
```

# Implementação de mecanismos para exclusão mútua

- Algorítmica
  - Combinação de variáveis do tipo *lock* e alternância (Dekker 1965)
- Primitivas
  - Mutex
  - Semáforo
  - Monitor

# Filósofos jantando

- O problema de sincronização mais conhecido é o do "Jantar dos Filósofos”
- Foi proposto por Dijkstra (1965) como um problema clássico de sincronização

## Demo dos filósofos jantando

# Mutex

- Variável compartilhada para controle de acesso a seção crítica
- CPUs são projetadas levando-se em conta a possibilidade do uso de múltiplos processos
- Inclusão de duas instruções *assembly* para leitura e escrita de posições de memória de forma atômica
  - CAS: *Compare and Store*
    - Copia o valor de uma posição de memória para um registrador interno e escreve nela o valor 1
  - TSL: *Test and Set Lock*
    - Lê o valor de uma posição de memória e coloca nela um valor não zero

# Primitivas lock e unlock

- O emprego de *Mutex* necessita duas primitivas

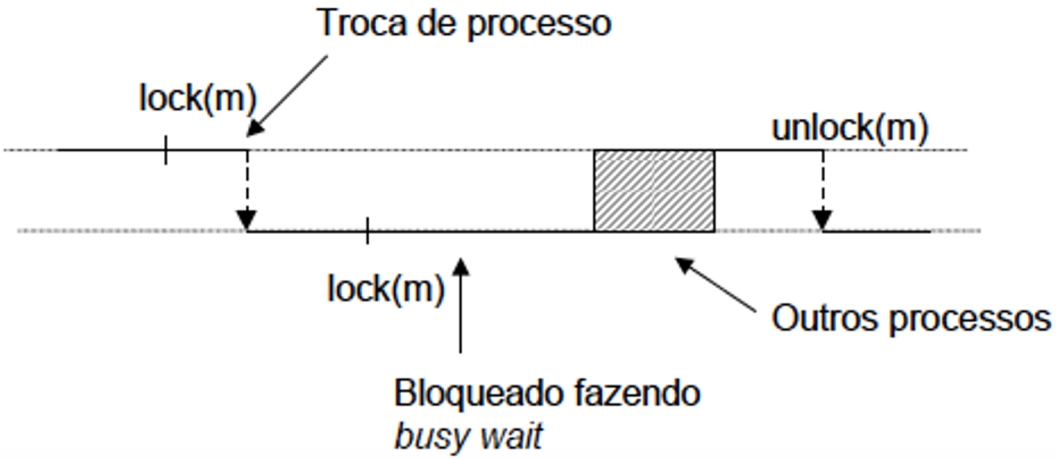
```
enter_region:   tst register,flag
                cmp register,0
                jnz   enter_region
                ret

leave_region:   mov flag,0
                ret
```

```
                lock(flag);
Seção crítica
                unlock(flag);
```

# Primitivas lock e unlock: problemas

- *Busy waiting* (*spin lock*)
- Confiar no processo (programador)!
  - Fazer o *lock* e *unlock* corretamente
- Inversão de prioridades



# Primitivas lock e unlock: problemas

- Solução:
  - Bloquear o processo ao invés de executar *busy waiting*
  - Baseado em duas novas primitivas
    - *sleep*: bloqueia um processo a espera de uma sinalização
    - *wakeup*: sinaliza um processo

# Demo com Mutex

# Semáforo

- Semáforo é uma variável especial protegida (ou tipo abstrato de dados) que tem como função o controle de acesso a recursos compartilhados em um ambiente multitarefa
- Esse tipo de variável foi proposto por Dijkstra (1965) e foi utilizado inicialmente no sistema operacional **THEOS, *i.e.*, Deus ou "The OS"**

# Semáforo

- O valor de um semáforo indica quantos processos podem acessar um recurso compartilhado
- Operações:
  - **Inicialização:** um valor inteiro indicando a quantidade de processos que podem acessar o recurso
  - **Operação P (testar):** decrementa o valor do semáforo. Se valor zero, o processo é posto para dormir
  - **Operação V (incrementar):** Se o semáforo estiver com o valor zero e existir algum processo adormecido, um processo será acordado. Caso contrário, o valor do semáforo é incrementado

# Implementação de Semáforo

Primitivas P e V

```
P(s): s.valor = s.valor - 1
      Se s.valor < 0 {
        Bloqueia processo (sleep);
        Insere processo em S.fila;
      }
```

```
V(s): s.valor = s.valor + 1
      Se S.valor <=0 {
        Retira processo de S.fila;
        Acorda processo (wakeup);
      }
```

- Necessidade de garantir a atomicidade nas operações de incremento (decremento) e teste da variável compartilhada s.valor
  - Uso de mutex
- Dependendo dos valores assumidos por s.valor
  - Semáforos binários: s.valor = 1
  - Semáforos contadores: s.valor = n

# Demo com Semáforo

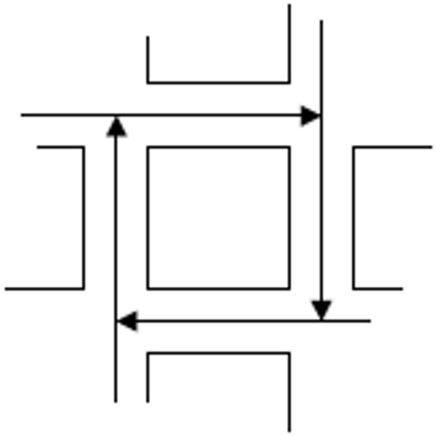
# Semáforo versus Mutex

- Primitivas *lock* e *unlock* são necessariamente feitas por um mesmo processo
  - Acesso a seção crítica
- Primitivas P e V podem ser realizadas por processos diferentes
  - Gerência de recursos

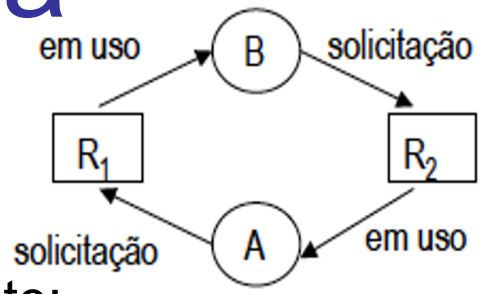
**Isso faz toda a diferença!**

# Deadlock

- Situação na qual um processo (ou mais) fica impedido de prosseguir sua execução, devido ao fato de cada um estar aguardando acesso a recursos já alocados por outro processo



# Condições para ocorrência de deadlocks




- Quatro condições devem ser satisfeitas simultaneamente:
  - Exclusão mútua:
    - Todo recurso ou está disponível ou está atribuído a um único processo
  - Segura/espera:
    - Os processos que detém um recurso podem solicitar novos recursos
  - Recurso não-preemptivo:
    - Um recurso concedido não pode ser retirado de um processo por outro
  - Espera circular:
    - Existência de um ciclo de 2 ou mais processos cada um esperando por um recurso já adquirido (em uso) pelo próximo processo no ciclo

# Estratégias para tratamento de deadlocks

- Ignorar (estatística)
- Detecção e recuperação
  - Monitoração dos recursos liberados e alocados
  - Eliminação de processos
- Impedir ocorrência cuidando a alocação de recursos
- Prevenção (por construção)
  - Evitar a ocorrência de pelo menos uma das quatro condições necessárias

# Referências Bibliográficas


- SILBERSCHATZ, A.; GALVIN, Peter; GAGNE Greg, Operating System Concepts Essentials. John Wiley & Sons, Inc. 2th edition, 2013.
- TANENBAUM, Andrew S. Sistemas operacionais modernos. 3a. ed. São Paulo: Pearson, 2009-2013. p. 653.
- OLIVEIRA, Rômulo; CARÍSSIMI, Alexandre; TOSCANI, Simão. Sistemas Operacionais. Porto Alegre: Bookman, 4a. ed. 2010.