Universidade Federal de Uberlândia
Faculdade de Computação
Sistemas Operacionais

Concorrência
Parte 2

Prof. Dr. Marcelo Zanchetta do Nascimento

Roteiro

- Exclusão mútua com software de bloqueio:
  - Sleep e wakeup
  - Semáfaro
  - Monitores
- Exemplo em Sistemas Operacionais
- Leituras Sugeridas

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

Primitivas Dormir (sleep) e Acordar (wakeup)

- Mecanismo de sincronização que permite que um processo fique bloqueado até que o recurso fique disponível;
  - Não há consumo de processamento;

- **Primitivas Dormir e Acordar:** Bloqueio e desbloqueio de processos.
  - Isto evita o desperdício de tempo de CPU, como nas soluções com ***busy wait***.

- A primitiva **Dormir** é uma chamada de sistema que suspende a execução de um processo até que outro processo **Acorde**.

Exclusão Mútua

- A primitiva **sleep()** - bloqueia o processo e espera por uma sinalização, isto é, suspende a execução do processo que fez a chamada até que um outro o acorde.

- A primitiva **wakeup()** - sinaliza (acorda) o processo anteriormente bloqueado por **sleep()**;

Exemplo: Produtor/Consumidor (*buffer limitado*):

  - Quando há dois processos que compartilham um *buffer* de tamanho fixo;

  - O processo produtor coloca dados no buffer e o processo consumidor retira dados do buffer.

Exclusão Mútua

Primitivas Dormir e Acordar

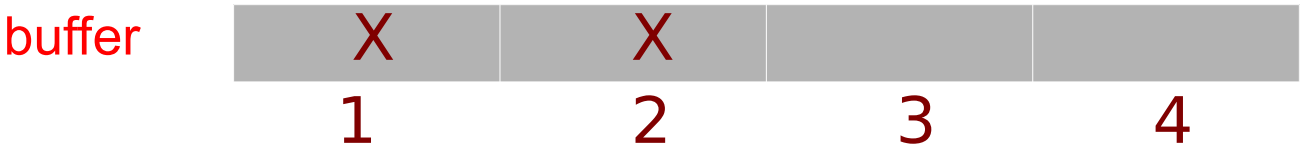
- Solução: colocar os processos para "dormir" até que eles possam ser executados;
  - Buffer: com uma variável count que controla a quantidade de dados presente no buffer;
  - Produtor: Antes de colocar dados no buffer: processo produtor verifica o valor da variável;
    - Se a variável está com valor máximo: o processo **produtor** é colocado para **dormir (bloqueado);**
    - **Caso contrário:** o produtor coloca os dados no buffer e incrementa variável de controle.

Exclusão Mútua

Primitivas Dormir e Acordar

Consumidor: Antes de retirar dados do *buffer,* o processo consumidor verifica o valor da variável count para saber se o estado atual é igual a 0 (zero).

- Se **count = = 0**, o processo vai "dormir".
  - O consumidor é quem deve ser bloqueado;
- Senão retira os dados do *buffer* e decrementa a variável.



Exclusão Mútua

```
#define N 100                                   /* número de lugares no buffer */
int count = 0;                                  /* número de itens no buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* número de itens no buffer */
          item = produce_item( );               /* gera o próximo item */
          if (count == N) sleep( );             /* se o buffer estiver cheio, vá dormir */
          insert_item(item);                    /* ponha um item no buffer */
          count = count + 1;                    /* incremente o contador de itens no buffer */
          if (count == 1) wakeup(consumer);     /* o buffer estava vazio? */
     }
}
```

```
void consumer(void)
{
     int item;

     while (TRUE) {                             /* repita para sempre */
          if (count == 0) sleep( );             /* se o buffer estiver vazio, vá dormir */
          item = remove_item( );                /* retire o item do buffer */
          count = count – 1;                    /* decresça de um o contador de itens no buffer */
          if (count == N – 1) wakeup(producer); /* o buffer estava cheio? */
          consume_item(item);                   /* imprima o item */
     }
}
```

Exclusão Mútua

Problema: a variável **count** ter acesso irrestrito;

- O *buffer* vazio e o **consumidor acabou de ler** a variável count com valor igual a 0;
  - O algoritmo de escalonamento **decide parar de executar** o consumidor (preempção) e começa a executar o produtor;
- O **produtor** insere um item no *buffer* e incrementa a variável count para igual a 1;
  - Com a regra do código: o valor de **count == 0,** e que o consumidor está dormindo, **produtor** envia um sinal de *wakeup* para o consumidor;
  - `if (count == 1) weakup(consumer);`

Exclusão Mútua

Primitivas Dormir e Acordar

- O **consumidor** não esta dormindo;
  - Retirado de execução pela preempção;
  - O sinal de *wakeup é* ***perdido***;
- Quando o **consumidor** é executado, baseado em uma nova preempção pelo despachante:
  - a variável **count** lida (com estado igual a 0) avalia a condição `if(count == 0);`
  - Então, o **consumidor** vai dormir, pois entende que não há mais informações no *buffer*;

Exclusão Mútua

Primitivas Dormir e Acordar

- Dessa forma, se o **produtor** continuar inserindo dados no buffer;
- Após execuções o **produtor** também irá dormir:
  - `if (count == N)`

Ambos os processos dormem para sempre.

Exclusão Mútua

Primitivas Dormir e Acordar

Solução: Adicionar ao contexto

O ***bit* de controle** armazena o valor true (sinal) enviado para um processo que não está dormindo.

No entanto, no caso de vários pares de processos, vários *bits* devem ser criados e uma sobrecarregar é gerado no sistema.

Cofre para guardar os sinais de acordar

Se tiver 2, 3 ou mais processos?

Improviso: Adicionar mais bits – problema permanecerá

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

Semáforo

- O matemático holandês E. W. Dijkstra (1965) sugeriu usar um novo tipo de variável para contar o número de sinais de acordar salvos para uso futuro;
- Um novo **tipo de variável** denominado **Semáforo;**
- O semáforo é uma variável que pode ser mudada por apenas duas operações primitivas (atômicas): P (proberen, "to test") e V (verhogen, "to increment");
- Um semáforo pode conter:
  - O valor "0" – indicando que nenhum sinal foi salvo;
  - Um valor "positivo" se um ou mais sinais estiverem pendentes.

Semáforo

Problema da exclusão mútua

- A operação **P** também é referenciada como:
  - **down** ou **wait**
- A operação **V** é referenciada por:
  - **up** ou **signal**
- Os semáforos que assumem somente os valores 0 e 1 são denominados **semáforos binários** ou **mutex**.
- Neste caso, as operações P e V são também chamadas de LOCK e UNLOCK, respectivamente.

Semáforo

É feito usando operações atômicas:

```
down(S){
    S = S - 1;
    suspende
}
```

- Iniciar o semáforo com o contador em 1;
- Garante que, uma vez iniciada uma operação de semáforo, nenhum outro processo pode ter acesso ao semáforo até que a operação tenha terminado ou sido bloqueada;
- **Atomicidade absoluta** essencial para resolver o problema de sincronização.

Semáforo

```
Up(S){
    S = S + 1;
    //fila de processos prontos
}
```

- Se um ou mais processos estiverem dormindo naquele semáforo, incapacitados de terminar uma operação, um deles será escolhido pelo sistema e será dado a permissão de terminar o seu down.

Semáforo

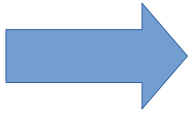
Exclusão mútua (**semáforos binários**):

...

Semaphore mutex = 1; /* var. semáforo, iniciado com 1*/

```
Processo P1
...
P(mutex)
// R.C.
V(mutex)
...
```

```
Processo P2
...
P(mutex)
// R.C.
V(mutex)
...
```



```
Processo Pn
...
P(mutex)
// R.C.
V(mutex)
...
```

Semáforo

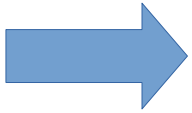
(**Alocação de Recursos - semáforos contadores):**

...

Semaphore S = 3; /* var. semáforo, iniciado com 1*/

```
Processo P1
...
P(S)
// R.C.
V(S)
...
```

```
Processo P2
...
P(S)
// R.C.
V(S)
...
```



```
Processo Pn
...
P(S)
// R.C.
V(S)
...
```

Semáforo

Problema: Produtor/consumidor:

- Resolve o problema de perda de sinais enviados;
- Úteis quando aplicados em problemas de sincronização condicional, onde existem processos concorrentes alocando recursos do mesmo tipo;
- No entanto, necessidade de três semáforos:
  - Controle do número de posições em um buffer;
  - Mutex para controle de acesso.

Semáforo

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
        int item;

        while (TRUE) {                          /* TRUE is the constant 1 */
                item = produce_item( );         /* generate something to put in buffer */
                down(&empty);                   /* decrement empty count */
                down(&mutex);                   /* enter critical region */
                insert_item(item);              /* put new item in buffer */
                up(&mutex);                     /* leave critical region */
                up(&full);                      /* increment count of full slots */
        }
}
```

Semáforo

```
void consumer(void)
{
        int item;

        while (TRUE) {                          /* infinite loop */
                down(&full);                    /* decrement full count */
                down(&mutex);                   /* enter critical region */
                item = remove_item( );          /* take item from buffer */
                up(&mutex);                     /* leave critical region */
                up(&empty);                     /* increment count of empty slots */
                consume_item(item);             /* do something with the item */
        }
}
```

Semáforo - POSIX

Algumas chamadas para semáforos:

#include <semaphore.h>

```
// inicializa um semáforo, com valor "value"
int sem_init (sem_t *sem, int pshared, unsigned int value);
```

sem : especifica o semáforo a ser inicializado.

pshared : Este argumento especifica se o semáforo é ou não compartilhado entre processos ou threads.

Um valor diferente de zero significa que o semáforo é compartilhado entre processos e um valor zero significa que ele é compartilhado entre threads.

valor : especifica o valor a ser atribuído ao semáforo recém-inicializado.

Semáforo - POSIX

Algumas chamadas para semáforos:

#include <semaphore.h>

```
// Up(s)
int sem_post (sem_t *sem);
```

```
// Down(s)
int sem_wait (sem_t *sem);
```

Semáforo - Threads

Multex em pthreads

- Há várias funções que podem ser usadas para sincronização dos threads;

- O mecanismo básico usa uma variável mutex, que pode ser travada ou destravada, para proteger a região crítica;

- Cabe ao programador assegurar que os threads os utilizem corretamente.

Mutexes em pthreads

Chamada de thread	Descrição
pthread_mutex_init	Cria um mutex
pthread_mutex_destroy	Destrói um mutex existente
pthread_mutex_lock	Conquista uma trava ou bloqueio
pthread_mutex_trylock	Conquista uma trava ou falha
pthread_mutex_unlock	Libera uma trava

O mutex permite bloquear e desbloquear acesso a região crítica

Semáforo - Threads

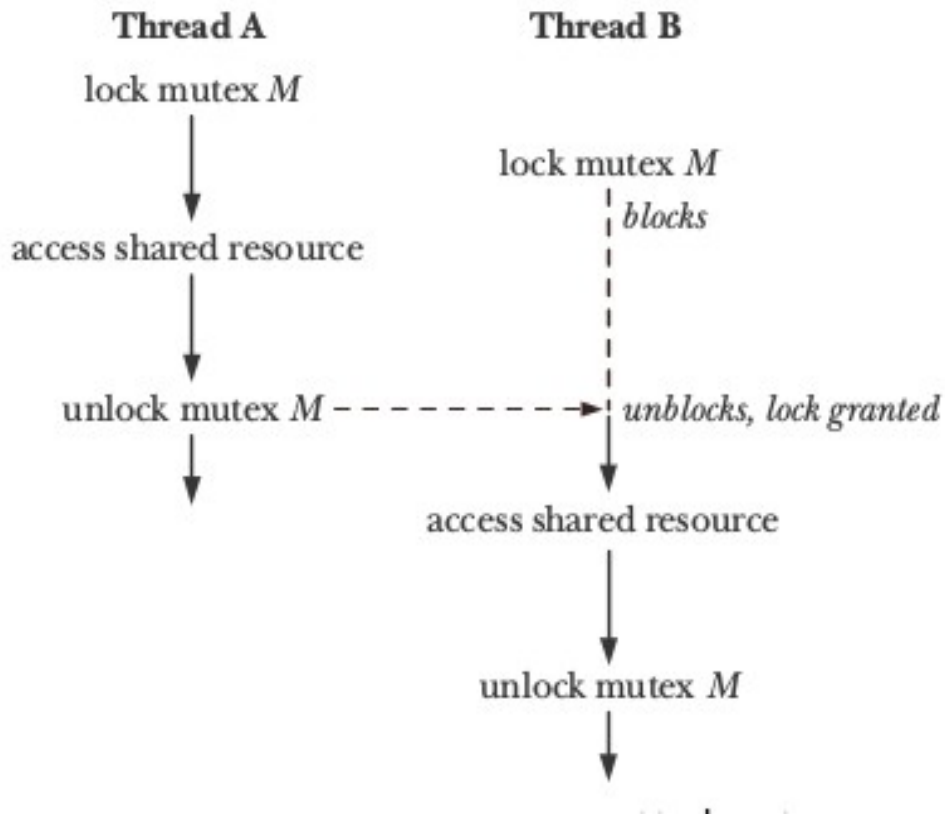


Figura 1: Controle de Threads com mutex

Mutexes em pthreads

Chamada de thread	Descrição
pthread_cond_init	Cria uma variável de condição
pthread_cond_destroy	Destrói uma variável de condição
pthread_cond_wait	Bloqueio esperando por um sinal
pthread_cond_signal	Sinaliza para outro thread e o desperta
pthread_cond_broadcast	Sinaliza para múltiplos threads e desperta todos eles

variáveis de condição

Semáforos

Problema:

- Primitiva deve ser encaixada dentro do código de cada processo.
- Isso esta sujeito a erros por duas razões:
  - As pessoas se esquecem das coisas;
  - As pessoas podem ignorar a regras para ganhar uma vantagem em desempenho ou violar a segurança;
- Erro de programação pode gerar um *deadlock*;
  - Suponha que o código seja trocado no processo produtor.

Semáforos

Problema:

- Suponha que os dois down do código do produtor estivessem invertidos.
- Neste caso, mutex seria diminuído antes de empty.
- Se o buffer estivesse completamente cheio;
- O produtor bloquearia com mutex = 0.

Semáforos

Produtor

```
empty.adquire();
mutex.adquire();
enter_item(item);

mutex.release () ;
      full.release();
```

```
   mutex.adquire();
    empty.adquire();

enter_item(item);
    mutex.release();
    full.release();
```

Consumidor

```
    full.adquire();
    mutex.adquire();

remove_item(item);
    mutex.release();
    empty.release();
```

- Se o *buffer* estiver cheio: o produtor será bloqueado;
- Assim, a próxima vez que o consumidor tentar acessar o *buffer*, ele tenta executar um release sobre o mutex, ficando também bloqueado.

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

Monitores

- Primitiva de alto nível;
- Semáforos exigem bastante cuidado, pois qualquer engano pode levar aos problemas de sincronização imprevisíveis;
- Idealizado por Hoare (1974) e Brinch Hansen (1975):
  - Primitiva de alto nível para sincronizar processos implementadas pelo compilador.
- É um conjunto de procedimentos, variáveis e/ou estruturas de dados agrupados em um único módulo (classe):
  - Exclusão mútua.

Monitores

- Ideia central: Em vez de codificar as seções críticas dentro de cada processo, pode-se codificá-las como procedimentos do monitor.
- Assim, quando um processo precisa referenciar dados compartilhados, chama um procedimento do monitor.
- Resultado: o código da seção crítica não é mais duplicado em cada processo.
- Os dados declarados dentro do monitor são compartilhados por todos os processos, mas só podem ser acessados através dos procedimentos do monitor.

Monitores

- Somente um processo pode estar ativo dentro do monitor em um instante;
  - outros processos ficam bloqueados até que possam estar ativos no monitor.
- O monitor possui uma fila (FIFO) de entrada:
  - processos que desejem executar um procedimento do monitor deve aguardam sua vez.
- Cabe ao **compilador** implementar a exclusão mútua nas entradas do monitor (mutex ou semáforo binário).
  - Compiladores tratam de forma diferente das outras chamadas de procedimento;
  - Compiladores podem implementar a exclusão mútua por meio dos monitores – Ex.: **Java, C++.**

Monitores

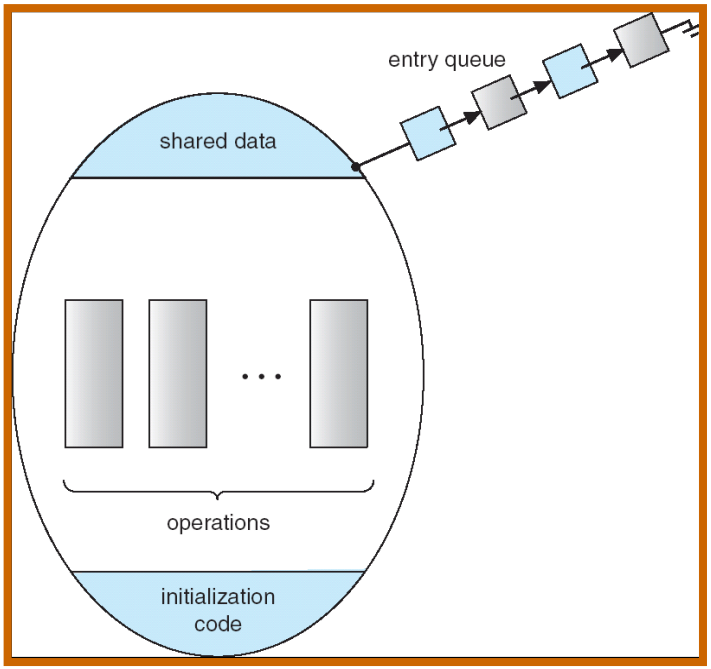
- O programa cria um objeto do tipo monitor durante sua execução;
- Esses processos invocam operações do monitor para executar dados compartilhados e sincronizados.

```
monitor monitor name
{
   // shared variable declarations

   initialization code ( . . . ) {
      . . .
   }

   public P1 ( . . . ) {
      . . .
   }

   public P2 ( . . . ) {
      . . .
   }
         .
         .
         .
   public Pn ( . . . ) {
      . . .
   }
}
```



Monitores

Procedimentos para Execução:

- Chamada a uma rotina do monitor;
- Instruções iniciais:
  - Teste para detectar se um outro processo está ativo dentro do monitor;
- Se positivo, o processo novo ficará bloqueado até que o outro processo deixe o monitor;
- Caso contrário, o processo novo entra no monitor;

Monitores

- Exclusão mútua não é realizada pelo programador: cria procedimentos no monitor;
  - Comunicação ocorre através de chamadas a seus procedimentos e de seus parâmetros passados;
- As regiões críticas devem ser definidas como procedimentos no monitor:
  - O compilador se encarregará de **garantir a exclusão mútua** entre esses procedimentos;

Monitores

```
package monitorjava;

public class Main {

    public static void main(String args[]) {

        ObjetoBuffer umBuffer = new ObjetoBuffer();

        // criacao das threads
        Produtor umProdutor = new Produtor(umBuffer);

        Consumidor umConsumidor = new Consumidor(umBuffer);

        // start threads
        umProdutor.start();

        umConsumidor.start();
    }

}
```

Monitores

- A palavra synchronized significa que o método será executado se puder adquirir o monitor do objeto a quem pertence o método.

```
public class ObjetoBuffer {

  private int memoria = -1;
  private boolean acessivel = true; //variavel de condicao de escrita

  // metodo de escrita de dados na memoria
  public synchronized void escreveBuffer(int valor) {
      while (!acessivel) { // nao e a vez de escrever
        try {
           wait(); //suspende a thread
        } catch (InterruptedException e) {
           e.printStackTrace();
        }
      }
     System.err.println(Thread.currentThread().getName() +
                 " produzindo o valor: " + valor);
     this.memoria = valor;
     acessivel = false;  // desabilita a memoria para escrita
     notify();  // libera a thread que esta ESPERANDO devido a um wait( )

  }
```

Monitores

```
 // metodo de leitura de dados na memoria
    public synchronized int lerBuffer() {
        while (acessivel) { // nao eh a vez de ler
            try {
                wait(); //suspende a thread que chamou este metodo
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

   System.err.println(Thread.currentThread().getName() +
                " consumindo o valor: " + this.memoria);

   acessivel = true;  // libera buffer para escrita

   notify();  // libera uma thread que esta ESPERANDO devido a um
wait( )

   return this.memoria;
    }
}
```

Monitores

```
public class Produtor extends Thread {
  private  ObjetoBuffer o_Buffer;

  public Produtor( ObjetoBuffer dado ) {
     super( "Produtor" );
     o_Buffer = dado;
  }

  //Thread do Produtor escreve 10 vezes em intervalos de tempo
  public void run(){
     for ( int i = 1; i <= 10; i++ ) {
        try { // dorme por um tempo aleatorio
           Thread.sleep( ( int ) ( Math.random() * 3000 ) );
        }
        catch( InterruptedException exception ) {
           System.err.println( exception.toString() );
        }
        // chama metodo do objeto buffer
       o_Buffer.escreveBuffer( i );
     }
     System.err.println(getName() + " terminou de produzir");
  }
}
```

Monitores

```
public class Consumidor extends Thread {
  private ObjetoBuffer um_Buffer;
  public Consumidor(ObjetoBuffer dado) {
     super("Consumidor");
     um_Buffer = dado;
   }
   //Thread Consumidor lera o buffer 10 vezes em intervalos
   public void run() {
     int valor, soma = 0;
     do { // dorme por um intervalo aleatorio
       try {
         Thread.sleep((int) (Math.random() * 3000));
       } // Tratamento de excecao
       catch (InterruptedException exception) {
           System.err.println(exception.toString());
       }
       valor = um_Buffer.lerBuffer();
       soma += valor;
     } while (valor != 10);
     System.err.println(
                 getName() + " terminou de consumir. Totalizou: "
+ soma);
    }
}
```

Monitores

- **Chamada Notify**
  - Apanha um thread qualquer na lista de threads no conjunto de espera;
  - Move um thread do conjunto de espera para o conjunto de entrada;
  - Define o estado da thread de bloqueado para executável.

Exemplos

Sincronização em Windows:

- Trabalha com **kernel multi-thread** que fornece suporte para aplicações de **tempo real** e **múltiplos processadores**.
- **Recurso em um processador**: irá ocultar temporariamente para todas as interrupções que podem ter acesso ao recurso global.
- **Recurso em multiprocessadores:** usa o **spin-lock** apenas para proteger pequenos segmentos de código.
- **Sincronização de threads:** provê objetos despachantes (mutex, semáforos) para threads fora do kernel.

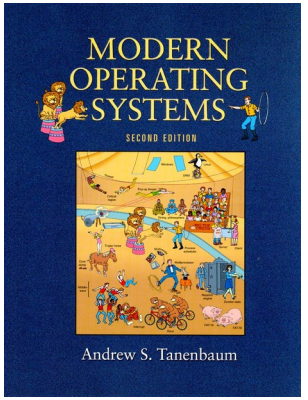
Exemplos

Sincronização em Linux:

- O kernel é **completamente preemptivo** e fornece instruções para **versão atômica** para simples operações matemáticas.
- Em que inteiros atômicos são executados sem interrupção.
- Em um **processador** (sistema embarcado) é utilizado o recurso de **ativação e desativação da preempção** do kernel.
- Em **máquina com multiprocessadores**, o mecanismo fundamental empregado é o **spin-lock.**
- Quando um **bloqueio deve ser mantido por um período** mais longo, os semáforos ou mutex são empregados.

Leituras Sugeridas

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



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