# Sistemas Operacionais: Processos e Threads

**Prof. Maurício Aronne Pillon**
**Prof. Rafael R. Obelheiro**

UDESC/CCT – Departamento de Ciência da Computação
{mauricio.pillon,rafael.obelheiro}@udesc.br

Joinville, setembro de 2017



## Sumário







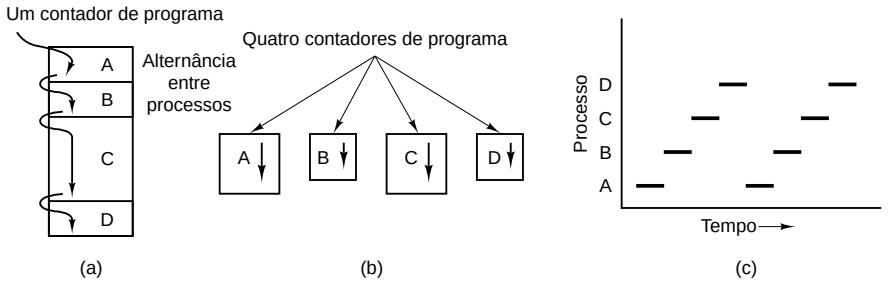
## Conceito

- Um **processo** é um **programa em execução**
  - registradores, contador de programa, pilha
  - cada processo enxerga uma CPU virtual
- **Multiprogramação**: vários processos carregados na memória ao mesmo tempo
  - máquinas monoprocessadas: apenas um processo executa de cada vez
    - **pseudoparalelismo**
  - máquinas multiprocessadas: paralelismo real





## Multiprogramação de quatro processos




- Processos não devem fazer hipóteses temporais ou sobre a ordem de execução
  - primitivas de sincronização

## Criação de processos

Principais eventos que levam à criação de processos:

1. Início do sistema
2. Execução de chamada ao sistema de criação de processos
   - `fork` (UNIX), `CreateProcess` (Windows), `SYS$CREPRC` (VAX/VMS)
3. Solicitação do usuário para criar um novo processo
4. Início de um job em lote





## Tipos de processos

- Processos interativos: interagem com usuários
  - primeiro plano (*foreground*)
- Processos de segundo plano (*background*): serviços do sistema
  - *daemons*





## Exemplo: chamada `fork`

- No UNIX, processos são criados através da chamada `fork`
- O processo filho é idêntico ao processo pai:
  - código e dados são copiados
  - diferença está no valor de retorno da função `fork()`
    - no processo pai, a função retorna o identificador (PID) do filho
    - no processo filho, a função retorna 0
  - a chamada `exec` pode ser usada para substituir o processo corrente

```
f = fork();
if (f == 0)  {                    /* processo filho */
   printf("processo filho\n");
   exit(4);                            /* retorna 4 */
} else {                           /* processo pai */
   printf("processo pai\n")
   w = waitpid(f, &rc, 0);      /* espera retorno   */
}                               /* do filho (rc==4) */
```





## Término de processos

- Condições para o término de um processo:
  - saída normal (voluntária)
  - saída por erro (voluntária)
    - programa detecta um erro
  - erro fatal (involuntário)
    - programa faz algo ilegal
  - cancelamento por outro processo (involuntário)
- O término de um processo pode causar o término dos processos que ele criou
  - não ocorre nem em UNIX nem em Windows

# Hierarquias de processos

- Processos "procriam" por várias gerações
  - um processo pai cria processos filhos, que por sua vez também criam seus filhos, *ad nauseam*
- Leva à formação de **hierarquias** de processos
- Chamadas "grupos de processos" no UNIX
  - sinalizações de eventos se propagam através do grupo, e cada processo decide o que fazer com o sinal (ignorar, tratar ou "ser morto")
  - todos os processos UNIX descendem de *init*
    - *systemd* em várias distribuições Linux
- Windows não possui hierarquias de processos
  - todos os processos são criados iguais





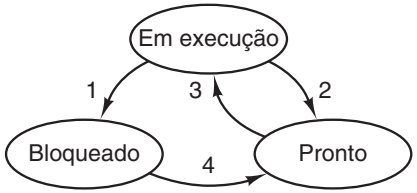
# Estados de um processo

- Um processo pode assumir diversos estados no sistema
  - **em execução**: processo que está usando a CPU
  - **pronto**: processo temporariamente parado enquanto outro processo executa
    - fila de prontos (aptos)
  - **bloqueado**: esperando por um evento externo





# Transições de estado de um processo




1. O processo bloqueia aguardando uma entrada
2. O escalonador seleciona outro processo
3. O escalonador seleciona esse processo
4. A entrada torna-se disponível

- Processos entram no sistema na fila de prontos
- Transições dependem de interrupções para sinalizar condições
  - término de operações de E/S, passagem do tempo, ...





# Implementação de processos

- As informações sobre os processos do sistema são armazenadas na **tabela de processos**
  - cada entrada é chamada de **descritor de processo** ou **bloco de controle de processo**

| Gerenciamento de processos | Gerenciamento de memória | Gerenciamento de arquivos |
|---|---|---|
| Registradores | Ponteiro para o segmento de código | Diretório-raiz |
| Contador de programa | Ponteiro para o segmento de dados | Diretório de trabalho |
| Palavra de estado do programa | Ponteiro para o segmento de pilha | Descritores de arquivos |
| Ponteiro de pilha | | Identificador (ID) do usuário |
| Estado do processo | | Identificador (ID) do grupo |
| Prioridade | | |
| Parâmetros de escalonamento | | |
| Identificador (ID) do processo | | |
| Processo pai | | |
| Grupo do processo | | |
| Sinais | | |
| Momento em que o processo iniciou | | |
| Tempo usado da CPU | | |
| Tempo de CPU do filho | | |
| Momento do próximo alarme | | |

## O papel das interrupções

- Interrupções são fundamentais para multiprogramação
  - sinalizam eventos no sistema
  - dão oportunidade para que o SO assuma o controle e decida o que fazer
- **Processos não executam sob o controle direto do SO**
  - o SO só assume quando ocorrem interrupções ou chamadas de sistema (implementadas com *traps*)





## Tratamento de interrupções

1. HW empilha contador de programa, PSW, etc.
2. HW carrega o novo PC a partir do vetor de interrupções
3. Rotina ASM salva os registradores
4. Rotina ASM configura uma nova pilha
5. Tratador de interrupções em C executa
6. O escalonador decide qual processo é o próximo a executar
7. Tratador de interrupções retorna para rotina ASM
8. Rotina ASM inicia o novo processo corrente





## Sumário





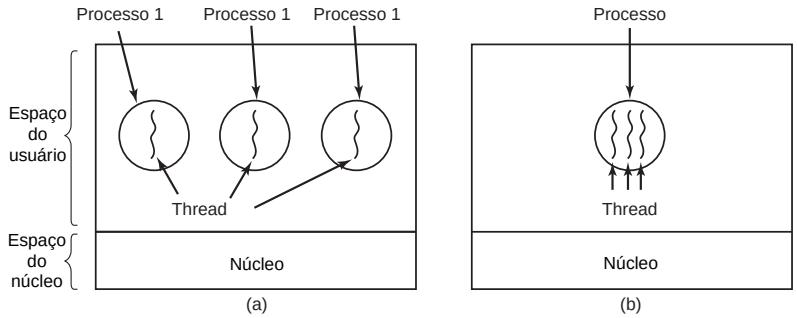


## O modelo de thread (1/2)

- Processos possuem
  - um espaço de endereçamento
  - uma thread de execução ou fluxo de controle
- Processos agrupam recursos
  - espaço de endereçamento (código+dados), arquivos, processos filhos, alarmes pendentes, ...
  - esse agrupamento facilita o gerenciamento
- A thread representa o estado atual de execução
  - contador de programa, registradores, pilha
- A unificação é uma conveniência, não um requisito

# O modelo de thread (2/2)

- Múltiplas threads em um processo permitem execuções paralelas sobre os mesmos recursos
  - análogo a vários processos em paralelo
- Processos leves ou multithread




(a) 3 processos com uma thread (b) Um processo com 3 threads





# Compartilhamento de recursos (1/2)

- As várias threads de um processo compartilham muitos dos recursos do processo
  - **não existe proteção entre threads**

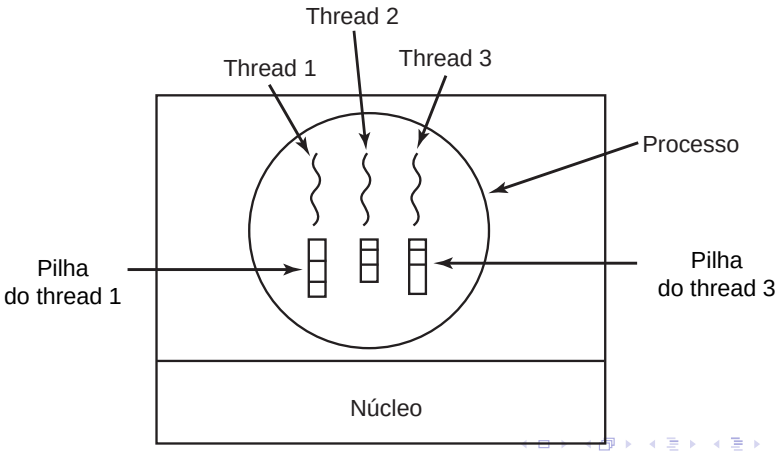
| Itens por processo | Itens por thread |
|---|---|
| Espaço de endereçamento | Contador de programa |
| Variáveis globais | Registradores |
| Arquivos abertos | Pilha |
| Processos filhos | Estado |
| Alarmes pendentes | |
| Sinais e tratadores de sinais | |
| Informação de contabilidade | |





# Compartilhamento de recursos (2/2)

- Cada thread precisa da sua própria pilha
  - mantém suas variáveis locais e histórico de execução

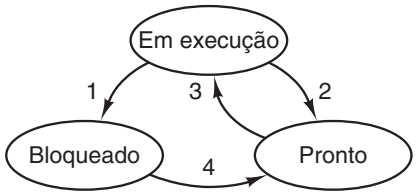







# Estados de uma thread

- Uma thread pode ter os mesmos estados de um processo
  - em execução, pronto, bloqueado




1. O processo bloqueia aguardando uma entrada
2. O escalonador seleciona outro processo
3. O escalonador seleciona esse processo
4. A entrada torna-se disponível

- Dependendo da implementação, o bloqueio de uma das threads de um processo pode bloquear todas as demais

# Vantagens de threads

- Possibilitar soluções paralelas para problemas
  - cada thread sequencial se preocupa com uma parte do problema
  - interessante em aplicações dirigidas a eventos
- Desempenho
  - criar e destruir threads é mais rápido
  - o chaveamento de contexto é muito mais rápido
  - permite combinar threads I/O-bound e CPU-bound





# Problemas com threads

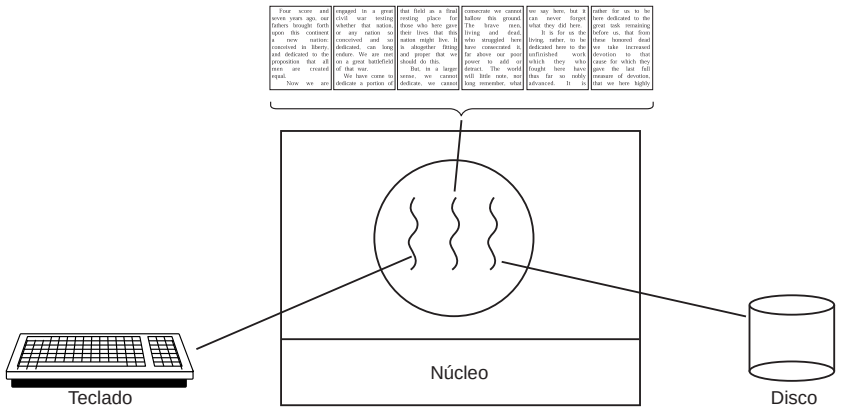
- Complicações no modelo de programação
  - um processo filho herda todas as threads do processo pai?
  - se herdar, o que acontece quando a thread do pai bloqueia por uma entrada de teclado?
- Complicações pelos recursos compartilhados
  - e se uma thread fecha um arquivo que está sendo usado por outra?
  - e se uma thread começa uma alocação de memória e é substituída por outra?





# Exemplos de uso de threads (1/3)

- Processador de texto com 3 threads
  - considere a implementação monothread

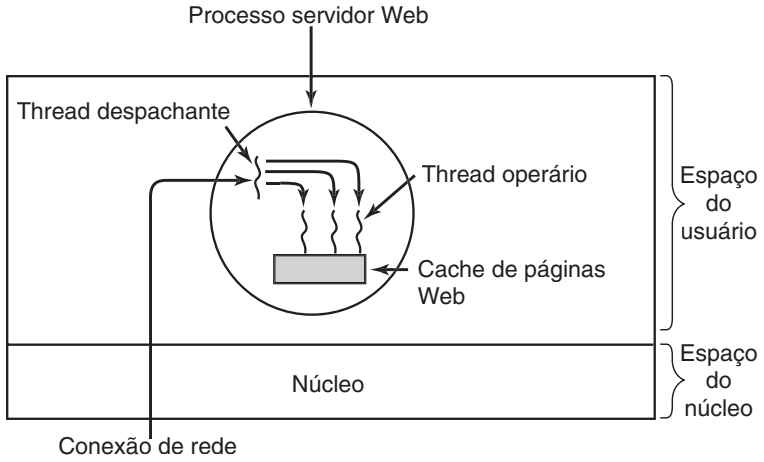







# Exemplos de uso de threads (2/3)

- Servidor web multithreaded



# Exemplos de uso de threads (3/3)

- Código simplificado do servidor web

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

(a) despachante

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
     if(page_not_in_cache(&page))
         read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

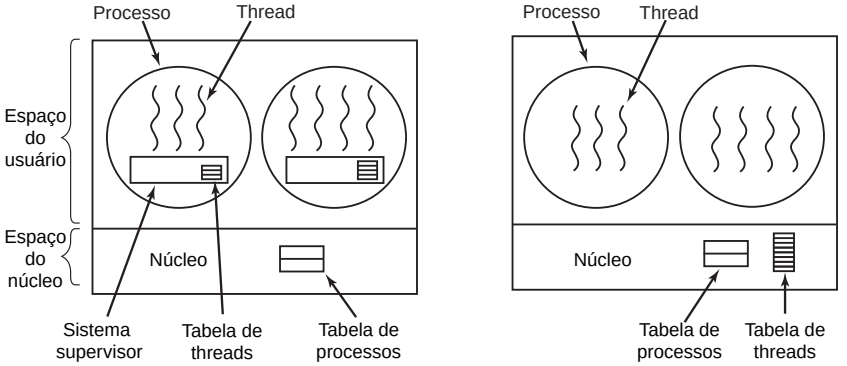
(b) operário





# Implementação de threads

- Existem dois modos principais de se implementar threads
  (a) threads no espaço do usuário (N:1)
  (b) threads no espaço do núcleo (1:1)




- Implementações híbridas também são possíveis





# Threads de usuário

- As threads são implementadas por uma biblioteca, e o núcleo não sabe nada sobre elas
  - N threads são mapeadas em um processo (N:1)
  - núcleo escalona processos, não threads
  - o escalonamento de threads é feito pela biblioteca
- Vantagens
  - permite usar threads em SOs que não têm suporte
  - chaveamento de contexto entre threads não requer chamada de sistema → desempenho
- Desvantagens
  - tratamento de chamadas bloqueantes
  - preempção por tempo é complicada



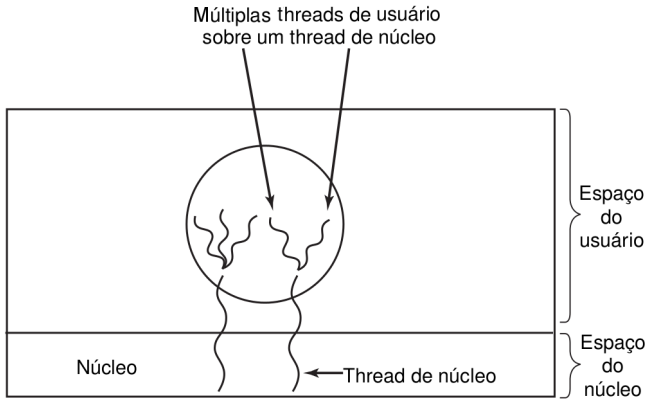


# Threads de núcleo

- O núcleo conhece e escalona as threads
  - não há necessidade de biblioteca
  - modelo 1:1
- Vantagens
  - facilidade para lidar com chamadas bloqueantes
  - preempção entre threads
- Desvantagens
  - operações envolvendo threads têm custo maior
    - exigem chamadas ao núcleo

## Threads híbridas

- Combina os dois modelos anteriores








## Convertendo código para multithreading

- Problemas em potencial
  - variáveis globais modificadas por várias threads
    - proibir o uso de variáveis globais
    - permitir variáveis globais privativas de cada thread
  - bibliotecas não reentrantes: funções que não podem ser executadas por mais de uma thread
    - permitir apenas uma execução por vez
  - sinais
    - quem captura? como tratar?
  - gerenciamento da pilha
    - o sistema precisa tratar o overflow de várias pilhas





## Introdução a Pthreads

- O padrão IEEE POSIX 1003.1c define uma API para programação usando threads
  - POSIX threads ⇒ Pthreads
- Implementações disponíveis para diversas variantes de UNIX e Windows
  - nível de usuário ou nível de núcleo





## Programando com Pthreads no Linux

- Para usar as funções da biblioteca Pthreads, deve-se incluir o cabeçalho `pthread.h`

```
#include <pthread.h>
```

- Para compilar um programa com Pthreads, deve-se passar a opção `-pthread` para o `gcc`

```
$ gcc -Wall -pthread -o prog prog.c
```

## Criando threads

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

- `pthread_t *thread`: identificador (ID) da thread, passado por referência
- `pthread_attr_t *attr`: atributos da thread
  - `NULL` para atributos default
  - manipulados via funções `pthread_attr_`*`nnnn`*
- `void *(*start_routine)`: ponteiro para a função onde inicia a thread
  - função possui um único parâmetro, `void *`
  - valor de retorno da função também é `void *`
- `void *arg`: argumento para `start_routine`
  - `NULL` se não há argumentos
- Retorna 0 para sucesso, outro valor em caso de erro





## Encerrando threads

- A execução da thread encerra quando:
  - ela retorna de `start_routine()`
  - ela invoca `pthread_exit()`
    - permite retornar um código de status
  - ela é cancelada por outra thread com `pthread_cancel()`
  - o processo inteiro encerra com `exit()` ou `exec()`





## Um exemplo simples (`simples.c`)

```
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS    5

void *PrintHello(void *arg) {
   long tid = (long)arg;
   printf("Alo da thread %ld\n",
          tid);
   pthread_exit(NULL);
}
```

```
int main (int argc, char *argv[]) {
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for (t=0; t<NUM_THREADS; t++){
      printf("main: criando thread %ld\n", t);
      rc = pthread_create(&threads[t],
                          NULL,
                          PrintHello,
                          (void *)t);
      if (rc) {
         printf("ERRO - rc=%d\n", rc);
         exit(-1);
      }
   }
   /* Ultima coisa que main() deve fazer */
   pthread_exit(NULL);
}
```





## Passando parâmetros para a thread

- A função onde a thread inicia só aceita um parâmetro `void *`
- Parâmetros de outros tipos requerem casting
  - vide exemplo anterior
- Para passar múltiplos parâmetros, pode ser usada uma `struct`

## Esperando a conclusão de uma thread

- Por padrão, uma thread é criada como *joinable*
  - a thread que a criou pode esperar que ela termine e recuperar o status retornado
- `int pthread_join(pthread_t thread, void **value_ptr)`
  - `pthread_t thread`: ID da thread a esperar
  - `void **value_ptr`: endereço da variável onde é armazenado o valor de retorno
    - especificado em `pthread_exit()`
  - retorna 0 para sucesso, outro valor em caso de erro
  - **thread chamadora fica bloqueada**





## Obtendo o valor de retorno de uma thread

```
void *funcThread(void *arg) {
   long ret;
   ...
   pthread_exit((void *) ret);
}
int main(int argv, char *argv[]) {
   void *status;
   long ret;

   pthread_create(&thr, NULL, funcThread, NULL);
   ...
   pthread_join(thr, &status);
   ret = (long) status;
   ...
}
```

- A thread coloca o valor de retorno em `pthread_exit()`, e o valor é recuperado com `pthread_join()`





## Threads detached

- Em algumas situações, não é necessário esperar a conclusão de uma thread → *detached*
- Threads detached permitem liberar recursos do SO
- Para criar uma thread como detached é preciso:
  1. Declarar uma variável de atributos do tipo `pthread_attr_t`
  2. Inicializar a variável com `pthread_attr_init()`
  3. Setar o atributo detached com `pthread_attr_setdetachstate()`
  4. Criar a thread
  5. Liberar recursos com `pthread_attr_destroy()`
- Pode-se usar `pthread_detach()` com uma thread já criada
- Não é possível usar `pthread_join()` com uma thread detached





## Criando threads detached

```
void *funcThread(void *arg) { ... }

int main(int argv, char *argv[]) {
   pthread_t thr;
   pthread_attr_t attr;

   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
   pthread_create(&thr, &attr, funcThread, NULL);
   pthread_attr_destroy(&attr);
   ...
}
```

- A mesma variável de atributos (`attr`) pode ser usada para criar diversas threads

## Obtendo e comparando IDs de thread

- Cada thread tem um ID único
  - IDs devem ser considerados objetos opacos → o programa não deve assumir nada sobre sua implementação
- `pthread_t pthread_self(void)`
  - retorna o ID da thread corrente
- `int pthread_equal(pthread_t t1, pthread_t t2)`
  - compara os IDs `t1` e `t2`
  - retorna 0 se `t1` $\neq$ `t2`, outro valor se `t1` $=$ `t2`





## Sumário







## Conceitos

- Frequentemente, processos precisam se comunicar para trocar dados
  - exemplo: pipes
- Tópicos envolvidos
  - como processos trocam informações
  - como garantir que um processo não invada o outro quando envolvidos em atividades críticas
  - como determinar a sequência de execução de processos
- Valem igualmente para threads

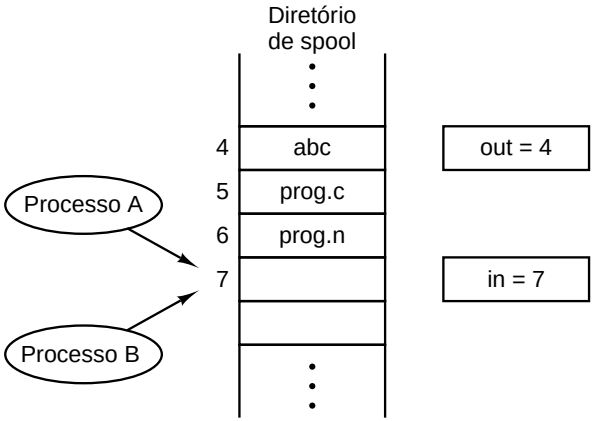




## Condições de disputa

- Quando dois ou mais processos manipulam dados compartilhados simultaneamente e o resultado depende da ordem precisa em que os processos são executados
- Exemplo: spool de impressão
  - processos colocam trabalhos em uma fila
  - servidor de impressão retira trabalhos da fila e os envia para a impressora
  - fila de impressão é uma área de armazenamento compartilhada
    - diretório de spool

## Condição de disputa: fila de impressão




- A e B inserem os seus trabalhos na posição 7





## Condição de disputa: um algoritmo americano

Administrando o estoque de leite

vê se tem leite na geladeira
se acabou:
sai para o mercado
chega no mercado
compra leite
volta para casa
põe leite na geladeira





## Uma execução do algoritmo

| **Alice** | **Bob** |
|---|---|
| vê que acabou o leite | |
| sai para o mercado | |
| chega no mercado | vê que acabou o leite |
| compra leite | sai para o mercado |
| volta para casa | chega no mercado |
| põe leite na geladeira | compra leite |
| | volta para casa |
| | põe leite na geladeira |





## Uma execução do algoritmo

| **Alice** | **Bob** |
|---|---|
| vê que acabou o leite | |
| sai para o mercado | |
| chega no mercado | vê que acabou o leite |
| compra leite | sai para o mercado |
| volta para casa | chega no mercado |
| põe leite na geladeira | compra leite |
| | volta para casa |
| | põe leite na geladeira |
| | danou-se (leite vai azedar) |

**o estoque de leite é um estado compartilhado**

## Região crítica

- Partes do código em que há acesso a memória compartilhada e que pode levar a condições de disputa
- É necessário haver **exclusão mútua** entre os processos durante suas regiões críticas
- Também chamadas de seções críticas





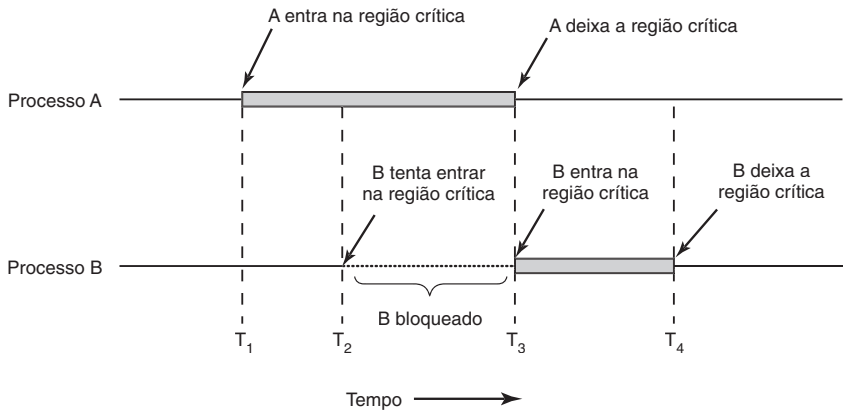
## Condições para exclusão mútua

- Quatro condições necessárias para prover exclusão mútua:
  1. Nunca dois processos podem estar simultaneamente em uma região crítica
  2. Nenhuma afirmação sobre velocidades ou número de CPUs
  3. Nenhum processo executando fora de sua região crítica pode bloquear outros processos
  4. Nenhum processo deve esperar eternamente para entrar em sua região crítica





## Exclusão mútua em regiões críticas








## Exclusão mútua com espera ocupada

- Existem diversas soluções para o problema de exclusão mútua
- Algumas delas se baseiam em espera ocupada (ociosa)
  - o processo fica em loop até conseguir entrar na seção crítica
- Exemplos
  - desabilitação de interrupções
  - variáveis de impedimento (lock)
  - alternância obrigatória
  - solução de Peterson
  - instrução TSL

# Desabilitação de interrupções

- Se as interrupções forem desabilitadas o processo não perde a CPU
  - transições de estado ocorrem por interrupções de tempo ou E/S
- Poder demais para processos de usuário
  - podem deixar de habilitar as interrupções (de propósito ou não)
- Não funciona em multiprocessadores
- Muito usada no núcleo do SO para seções críticas curtas
  - exemplo: atualização de listas encadeadas





# Variáveis de impedimento (lock)

- Uma variável lógica que indica se a seção crítica está ocupada

```
1: while (lock == 1)
2:     ;     /* loop vazio */
3: lock = 1;
4: /* seção crítica */
5: lock = 0;
6: /* seção não crítica */
```

- Solução sujeita a condições de disputa





# Alternância obrigatória (1/2)

- Cada processo tem a sua vez de entrar na seção crítica
  - variável **turn**
- Ainda é espera ocupada
  - desperdício de CPU
- Não funciona bem se um dos processos é muito mais lento do que o outro
  - viola a condição 3





# Alternância obrigatória (2/2)

```
while (TRUE) {
    while (turn !=0)        /* laço */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

(a) código para o processo 0

```
while (TRUE) {
    while (turn !=1)        /* laço */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

(b) código para o processo 1

# Solução de Peterson (1/2)

- Combina variáveis de lock e alternância obrigatória
- Funcionamento
  - antes de usar as variáveis compartilhadas, cada processo chama **enter_region** com o seu número como parâmetro
  - depois que terminou de usar as variáveis compartilhadas, o processo chama **leave_region**





# Solução de Peterson (2/2)

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

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

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

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

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





# Instrução TSL (*test and set lock*)

- Instrução de máquina que lê o conteúdo de uma variável e armazena o valor 1 nela
  - operação atômica (indivisível)
- Exige suporte de hardware
- Funciona para vários processadores
  - barramento de memória é travado para evitar acessos simultâneos





# Exclusão mútua com TSL

```
enter_region:
    TSL REGISTER,LOCK       | copia lock para o registrador e põe lock em 1
    CMP REGISTER,#0         | lock valia zero?
    JNE enter_region        | se fosse diferente de zero, lock estaria ligado,
                              portanto continue no laço de repetição
    RET | retorna a quem chamou; entrou na região crítica

leave_region:
    MOVE LOCK,#0            | coloque 0 em lock
    RET | retorna a quem chamou
```

# Instrução XCHG (*eXCHanGe*)

- Instrução atômica que troca o conteúdo de dois registradores ou um registrador e uma posição de memória
  - disponível na arquitetura x86
- Exclusão mútua com XCHG

```
1: enter_region:
2:   MOVE REG, #1       ! REG <- 1
3:   XCHG REG, lock     ! troca REG e lock
4:   CMP  REG, #0       ! se REG==0, lock era 0, e RC estava livre
5:   JNZ  enter_region  ! RC estava ocupada, fica no loop
6:   RET

7: leave_region:
8:   MOVE lock, #0
9:   RET
```





# Exclusão mútua sem espera ocupada

- Soluções de exclusão mútua baseadas em espera ocupada são indesejáveis
  - um loop vazio ocupa o processador
- Isso evita que outros processos executem
  - incluindo um processo na seção crítica
- Pode causar **inversão de prioridade**
  - processo mais prioritário fica no loop e um menos prioritário não consegue liberar a seção crítica
- Melhor seria se o processo que encontra a seção crítica ocupada pudesse ficar bloqueado até que a seção crítica fosse liberada





# Sleep e wakeup

- Primitivas simples que causam bloqueio e desbloqueio de processos
- `sleep()` bloqueia o chamador
- `wakeup(proc)` acorda `proc`
- Variante: `sleep(var)` && `wakeup(var)`
  - `var` casa um `sleep()` com o `wakeup()` correspondente





# Problema do produtor-consumidor

- Dois processos compartilham um buffer de tamanho limitado
  - produtor insere itens no buffer
    - não pode produzir se o buffer estiver cheio
  - consumidor retira itens do buffer
    - não pode consumir se o buffer estiver vazio
- Generalização de diversas situações de IPC que ocorrem na prática
  - servidor web multithread
    - thread despachante produz requisições
    - threads operárias consomem requisições

## Solução com sleep e wakeup

```
#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 */
     }
```





## Condição de disputa na solução

- Buffer vazio
- Consumidor verifica `count == 0` e perde o processador antes do `sleep()`
- Produtor insere um item, incrementa `count` e chama `wakeup(consumer)`
- Consumidor não recebe o `wakeup()`, executa o `sleep()` e nunca mais acorda
- Solução simples: usar um bit para armazenar um `wakeup()` perdido
  - não resolve se mais de um processo puder chamar `wakeup()`





## Semáforos

- Solução proposta por E. W. Dijkstra nos anos 60
- Um semáforo S conta o número de `wakeup()`s pendentes
- Duas primitivas atômicas
  - `down(S)`: decrementa S; se $S<0$, bloqueia
  - `up(S)`: incrementa S; se $S\leq 0$, acorda um processo que está esperando por S





## Semáforos

- Solução proposta por E. W. Dijkstra nos anos 60
- Um semáforo S conta o número de `wakeup()`s pendentes
- Duas primitivas atômicas
  - `down(S)`: decrementa S; se $S<0$, bloqueia
  - `up(S)`: incrementa S; se $S\leq 0$, acorda um processo que está esperando por S

**ATENÇÃO**: essa definição das primitivas é ligeiramente diferente da definição do Tanenbaum

# Implementação de semáforos

- Semáforos são implementados no núcleo do SO
- O semáforo é uma variável inteira
- Dificuldade é garantir atomicidade das operações `down()` e `up()`
- Uso de soluções com espera ocupada
  - desabilitação de interrupções
  - instrução TSL





# Produtor-consumidor com semáforos

```
#define N 100                       /* número de lugares no buffer */
typedef int semaphore;              /* semáforos são um tipo especial de int */
semaphore mutex = 1;                /* controla o acesso à região crítica */
semaphore empty = N;                /* conta os lugares vazios no buffer */
semaphore full = 0 ;                /* conta os lugares preenchidos no buffer */

void producer(void)
{
    int item;

    while (TRUE) {                  /* TRUE é a constante 1 */
        item = produce_item();      /* gera algo para pôr no buffer */
        down(&empty);               /* decresce o contador empty */
        down(&mutex);               /* entra na região crítica */
        insert_item(item);          /* põe novo item no buffer */
        up(&mutex);                 /* sai da região crítica */
        up(&full);                  /* incrementa o contador de lugares preenchidos */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                  /* laço infinito */
        down(&full);                /* decresce o contador full */
        down(&mutex);               /* entra na região crítica */
        item = remove_item();       /* pega o item do buffer */
        up(&mutex);                 /* deixa a região crítica */
        up(&empty);                 /* incrementa o contador de lugares vazios */
        consume_item(item);         /* faz algo com o item */
    }
}
```





# Tipos de semáforos

- Semáforos binários
  - inicializados em 1
  - controlam acesso à seção crítica
  - variável `mutex` na solução
- Semáforos contadores
  - sincronizam processos
  - determinam a ordem de execução
  - variáveis `full` e `empty` na solução





# Mutex como primitiva

- Em alguns casos é implementada uma versão simplificada de semáforos binários, chamada de mutex
  - apenas exclusão mútua, sem contagem
  - pode ser implementado em espaço de usuário
    - desde que o processador suporte TSL/XCHG

```
 1: mutex_lock:
 2:   TSL  REG, mutex     ! REG=mutex, mutex=1
 3:   CMP  REG, #0        ! se REG==0, mutex estava livre...
 4:   JZ   done           ! ... mutex adquirido, pode encerrar
 5:   CALL thread_yield   ! escalona outra thread
 6:   JMP  mutex_lock     ! quando voltar, tenta novamente
 7: done:                 ! encerra quando thread obteve mutex
 8:   RET                 !

 9: mutex_unlock:
10:   MOVE mutex, #0      ! libera mutex
11:   RET
```

## Futexes (*Fast userspace mutexes*)

- Mutexes em espaço de usuário são rápidos quando há pouca contenção
  - espera ocupada quase não ocorre na prática
- Chamadas para o kernel evitam espera ocupada, mas são lentas
  - ineficientes com pouca contenção → overhead
- Futexes tentam combinar os benefícios das duas abordagens
  1. A tentativa de travar um futex ocorre em espaço de usuário
  2. Se o futex já estava travado, ocorre uma chamada para o kernel para colocar o processo em uma fila de bloqueados
  3. Ao liberar o futex, o processo que o detinha verifica se há processos bloqueados; se houver, avisa ao kernel para desbloquear um deles
- Kernel só é envolvido quando ocorrer contenção





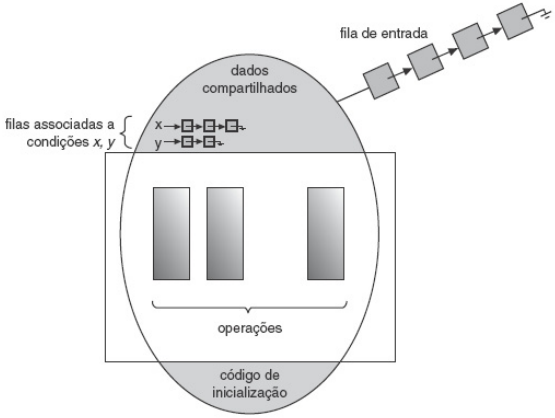
## Monitores

- Sincronização baseada em uma construção de linguagem de programação
- Criados por Per Brinch Hansen e Tony Hoare na década de 70
- Um monitor encapsula dados privados e procedimentos que os acessam
  - semelhante a uma classe
- Apenas um processo pode estar ativo no monitor em um dado instante
  - exclusão mútua entre processos
- Sincronização é feita usando variáveis de condição
  - duas operações: `wait()` e `signal()`





## Estrutura de um monitor








## Semântica de `signal()`

- O que acontece quando um processo executa `signal()`?
  - *signal-and-exit*
    - o processo atual deve sair do monitor após o `signal()`
    - o processo sinalizado entra no monitor
  - *signal-and-continue*
    - o processo atual continua executando
    - o processo sinalizado compete pelo monitor no próximo escalonamento

## Produtor-consumidor usando monitor

```
monitor ProducerConsumer
  condition full, empty;
  integer count := 0;

  procedure enter;
  begin
    if count = N then wait(full);
    enter_item;
    count := count + 1;
    if count = 1 then signal(empty);
  end;

  procedure remove;
  begin
    if count = 0 then wait(empty);
    remove_item;
    count := count - 1;
    if count = N-1 then signal(full);
  end;
end monitor;
```

```
procedure producer;
begin
  while true do
  begin
    produce_item;
    ProducerConsumer.enter;
  end
end;

procedure consumer;
begin
  while true do
  begin
    ProducerConsumer.remove;
    consume_item;
  end
end;
```





## Monitores em Java

- Java tem suporte parcial a monitores através de métodos **synchronized**
- Exclusão mútua no acesso ao objeto
  - apenas para métodos `synchronized`
  - o que acontece se há métodos não-`synchronized`?
- Uma única variável de condição anônima e implícita
- Operações: `wait()`, `notify()`, `notifyAll()`
- Semântica *signal-and-continue*
- O que disse o inventor do conceito de monitor:

  *"It is astounding to me that Java's insecure parallelism is taken seriously by the programming language community a quarter of a century after the invention of monitors and Concurrent Pascal. It has no merit." (Brinch Hansen, 1999)*





## Semáforos vs. monitores

- Monitores são mais fáceis de programar, e reduzem a probabilidade de erros
  - ordem de `down()` e `up()`
- Monitores dependem de linguagem de programação, enquanto semáforos são implementados pelo SO
  - podem ser usados com C, Java, BASIC, ASM, ...
- Ambos podem ser usados apenas em sistemas centralizados (com memória compartilhada)
  - sistemas distribuídos usam troca de mensagens





## Troca de mensagens

- Baseado em primitivas `send()` e `receive()`
- Usado em sistemas distribuídos
  - esquema cliente-servidor
- Questões de projeto
  - confiabilidade (perda ou duplicação de mensagens)
  - desempenho em sistemas centralizados
    - overhead para cópia de mensagens é maior do que para operações com semáforos e monitores

# Produtor-consumidor com mensagens

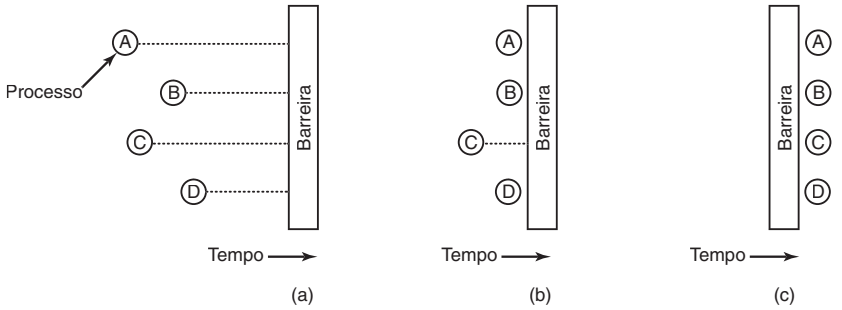
```
#define N 100                           /* número de lugares no buffer */

void producer(void)
{
     int item;
     message m;                         /* buffer de mensagens */

     while (TRUE) {
          item = produce_item( );       /* gera alguma coisa para colocar no buffer */
          receive(consumer, &m);        /* espera que uma mensagem vazia chegue */
          build_message(&m, item);      /* monta uma mensagem para enviar */
          send(consumer, &m);           /* envia item para consumidor */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* envia N mensagens vazias */
     while (TRUE) {
          receive(producer, &m);        /* pega mensagem contendo item */
          item = extract_item(&m);      /* extrai o item da mensagem */
          send(producer, &m);           /* envia a mensagem vazia como resposta */
          consume_item(item);           /* faz alguma coisa com o item */
     }
}
```





# Barreiras

- Mecanismo usado para definir um ponto de sincronização para múltiplos processos/threads








# Mecanismos de IPC no Linux

- Sistemas UNIX dão suporte a diversos mecanismos de IPC
  - pipes, filas de mensagens, memória compartilhada, semáforos
- Consideraremos aqui cinco mecanismos
  - Pthreads
    - mutexes, variáveis de condição e barreiras
  - processos
    - memória compartilhada e semáforos





# Mutexes

- Usados para garantir **exclusão mútua** em regiões críticas
  - semelhantes a semáforos binários
- Principais chamadas envolvendo mutexes
  - criação: `pthread_mutex_init()`
  - uso: `pthread_mutex_lock()`, `pthread_mutex_unlock()`, `pthread_mutex_trylock()`
  - destruição: `pthread_mutex_destroy()`

## Criando um mutex

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *attr);
```

- `pthread_mutex_t *mutex`: endereço do mutex
- `pthread_mutexattr_t *attr`: atributos do mutex
  - `NULL` para atributos default
- Mutex é criado destravado
- Retorna 0 para sucesso, outro valor em caso de erro
- Os dois trechos abaixo criam um mutex `mtx` inicializado com atributos default

```
1. pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

2. pthread_mutex_t mtx;
   pthread_mutex_init(&mtx, NULL);
```





## Destruindo um mutex

Quando não é mais necessário, um mutex deve ser destruído

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- `pthread_mutex_t *mutex`: endereço do mutex
- Retorna 0 para sucesso, outro valor em caso de erro





## Travando e destravando um mutex

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- Retornam 0 para sucesso, outro valor em caso de erro
- Quando um mutex é destravado com `pthread_mutex_unlock()`, não é possível determinar qual das threads bloqueadas será escalonada
- `pthread_mutex_trylock()` trava o mutex caso esteja livre, ou retorna imediatamente (sem bloquear), devolvendo `EBUSY`
  - evita bloqueio
  - é preciso ter cuidado com condições de disputa





## Variáveis de condição

- Mecanismo de **sincronização** entre threads
- Usadas em conjunto com mutexes
- Principais chamadas envolvendo variáveis de condição
  - criação: `pthread_cond_init()`
  - destruição: `pthread_cond_destroy()`
  - uso: `pthread_cond_wait()`, `pthread_cond_signal()`, `pthread_cond_broadcast()`

## Criando uma variável de condição

```
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *attr);
```

- `pthread_cond_t *cond`: endereço da variável de condição
- `pthread_condattr_t *attr`: atributos da variável de condição
  - `NULL` para atributos default
- Retorna 0 para sucesso, outro valor em caso de erro
- Os dois trechos abaixo criam uma variável de condição cond inicializada com atributos default

```
1. pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

2. pthread_cond_t cond;
   pthread_cond_init(&cond, NULL);
```





## Destruindo uma variável de condição

Quando não é mais necessária, uma variável de condição deve ser destruída

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

- `pthread_cond_t *cond`: endereço da variável de condição
- Retorna 0 para sucesso, outro valor em caso de erro





## Esperando por uma variável de condição

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

- Bloqueia a thread até que cond seja sinalizada
- **O acesso à variável de condição** `cond` **deve estar protegido por** `mutex`
  - `mutex` é automaticamente destravado quando a thread bloqueia
  - quando a condição for sinalizada, a thread retoma a execução com `mutex` travado para seu uso
  - é preciso destravar `mutex` ao final da região crítica
- Deve-se verificar se a condição foi efetivamente satisfeita, pois o desbloqueio pode não ter sido causado pela sinalização da variável
- Retorna 0 para sucesso, outro valor em caso de erro





## Sinalizando uma variável de condição

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- `pthread_cond_signal()` acorda uma thread bloqueada por uma variável de condição
  - deve ser invocada com `mutex` travado
  - thread sinalizada só retoma execução depois que `mutex` for destravado por quem executou `pthread_cond_signal()`
- `pthread_cond_broadcast()` acorda todas as threads bloqueadas por uma variável de condição
  - valem as mesmas restrições de `pthread_cond_signal()`
- **Se nenhuma thread estiver bloqueada esperando pela condição, a sinalização é perdida**
  - semelhante a `sleep()` e `wakeup()`
- Retornam 0 para sucesso, outro valor em caso de erro

## Barreiras

- Definem um ponto único de **sincronização** para múltiplas threads
- Principais chamadas envolvendo barreiras:
  - criação: `pthread_barrier_init()`
  - destruição: `pthread_barrier_destroy()`
  - uso: `pthread_barrier_wait()`





## Criação e destruição de barreira

```
int pthread_barrier_init(pthread_barrier_t *barrier,
    pthread_barrierattr_t *attr, unsigned count);

int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

- `pthread_barrier_t *barrier`: endereço da barreira
- `pthread_barrierattr_t *attr`: atributos da barreira
  - `NULL` para atributos default
- `unsigned count`: número de threads que esperam na barreira
  - deve ser maior que 0
- Retornam 0 para sucesso, outro valor em caso de erro





## Sincronizando em uma barreira

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

- `pthread_barrier_t *barrier`: endereço da barreira
- Espera na barreira `barrier` até que `count` threads o façam
- Quando a última thread chega na barreira, esta é reiniciada com `count`
  - se `count` for menor que o número de threads que invocam `pthread_barrier_wait()`, algumas threads podem ficar bloqueadas indefinidamente





## Criação de memória compartilhada

- Processos no UNIX possuem seu próprio espaço de endereçamento
  - mesmo um processo criado com `fork()` tem **apenas uma cópia** do espaço de endereçamento do processo pai
- É possível definir regiões de memória compartilhada entre processos

**Exemplo: compartilhando um inteiro**

```
int *ptr, rc, fd;

fd = shm_open("/shm", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
if (fd == -1) exit(1);
rc = ftruncate(fd, sizeof(int));
if (rc == -1) exit(2);
ptr = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED,
           fd, 0);
if (ptr == MAP_FAILED) exit(3);
```

- `ptr` aponta para a área de memória compartilhada

## shm_open()

`int shm_open(const char *name, int oflag, mode_t mode);`

- `char *name`: nome a ser dado para a região de memória
  - deve começar por / e conter caracteres válidos para nomear arquivos
- `int oflag`: flags de abertura (combinadas com |)
  - `O_RDONLY` (leitura) **ou** `O_RDWR` (leitura e escrita)
  - `O_CREAT`: cria o objeto caso não exista
  - `O_EXCL`: se usada com `O_CREAT` retorna erro caso o objeto exista
  - `O_TRUNC`: trunca o objeto caso já exista
- `mode_t mode`: define as permissões de acesso ao recurso
  - usado apenas quando o recurso é criado (| `O_CREAT`)
  - flags são as mesmas usadas pela chamada `open(2)`
    - `S_IRWXU, S_IRUSR, S_IWUSR, S_IXUSR`
    - `S_IRWXG, S_IRGRP, S_IWGRP, S_IXGRP`
    - `S_IRWXO, S_IROTH, S_IWOTH, S_IXOTH`
- Retorna um descritor de arquivo ou -1 em caso de erro
- Tamanho inicial do objeto é zero





## ftruncate()

`int ftruncate(int fd, off_t length);`

- Usado para definir o tamanho da região de memória compartilhada
  - `shm_open()` cria região com tamanho zero
- `int fd`: descritor de arquivo retornado por `shm_open()`
- `off_t length`: tamanho desejado
  - bytes do conteúdo são zerados
- retorna 0 para sucesso ou -1 em caso de erro





## mmap()

```
void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
```

- Mapeia `length` bytes do arquivo `fd` na memória, iniciando em `offset`
- `void *start`: endereço inicial preferencial para mapear o arquivo
  - `NULL` indica que não há preferência
  - endereço efetivamente usado é retornado pela função
- `size_t length`: tamanho da área de memória a mapear
  - tipicamente é o tamanho do arquivo
- `int prot`: flags de proteção para a área de memória
  - `PROT_NONE` → memória não pode ser acessada
  - uma combinação de `PROT_READ`, `PROT_WRITE`, `PROT_EXEC`





## mmap()

```
void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
```

- `int flags`: opções de mapeamento
  - para memória compartilhada, deve incluir `MAP_SHARED`
- `int fd`: descritor do arquivo a ser mapeado
  - retornado por `shm_open()`
- `off_t offset`: posição inicial do arquivo, em bytes
  - 0 para início
- `mmap()` retorna o endereço inicial do mapeamento ou `MAP_FAILED` em caso de erro
- O descritor de arquivo pode ser fechado sem afetar o acesso à memória compartilhada

## Liberação de memória compartilhada

`int munmap(void *start, size_t length);`

- Desfaz o mapeamento de memória de `length` bytes começando em `start`
- Parâmetros devem ser os mesmos usados no `mmap()`

`int shm_unlink(const char *name);`

- Remove a área de memória compartilhada chamada `name`
- Objeto só é destruído quando todos os mapeamentos tiverem sido desfeitos (com `munmap()`)

**Exemplo: liberando o inteiro compartilhado**

```
rc = munmap(ptr, sizeof(int));
if (rc == -1) exit(7);
rc = shm_unlink("/shm");
if (rc == -1) exit(8);
```





## Semáforos POSIX

- Existem duas APIs para uso de semáforos em UNIX
  - POSIX
  - System V
  - a API POSIX é mais simples, porém menos portável
- Principais chamadas da API POSIX
  - criação: `sem_open()`
  - uso: `sem_wait()`, `sem_post()`
  - liberação: `sem_close()`, `sem_unlink()`





## Criação de semáforos

```
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode,
                unsigned int value);
```

- `char *name`: nome do semáforo
  - valem as mesmas regras de `shm_open()`
- `int oflag`: flags de criação
  - pode conter 0 ou `O_CREAT` e `O_EXCL`
  - se `O_CREAT` for usada, `mode` e `value` têm que estar presentes
- `mode_t mode`: permissões de acesso
  - mesmos valores de `shm_open()`
- `unsigned int value`: valor inicial do semáforo
- A função retorna um ponteiro para o semáforo criado ou `SEM_FAILED` em caso de erro





## Operações sobre semáforos

- `int sem_wait(sem_t *sem);`
  - equivale a `down(&sem)`
- `int sem_post(sem_t *sem);`
  - equivale a `up(&sem)`
- Ambas as funções retornam 0 para sucesso e -1 em caso de erro
  - em caso de erro, o valor do semáforo não é alterado

## Liberação de semáforos

- `int sem_close(sem_t *sem);`
  - desvincula o semáforo do processo
- `int sem_unlink(const char *name);`
  - libera os recursos do SO associados ao semáforo
  - semáforo só é efetivamente destruído quando não estiver sendo usado por nenhum processo
- Ambas as funções retornam 0 para sucesso e -1 em caso de erro





## Sumário

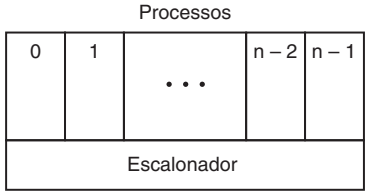






## Conceito de escalonamento

- Um requisito básico de sistemas multiprogramados é decidir qual processo deve executar a seguir, e por quanto tempo
  - o componente do SO que faz isso é o **escalonador** (*scheduler*)
  - o escalonador implementa um **algoritmo de escalonamento**
- Algoritmos de escalonamento podem diferir nos seus objetivos → o que se deseja priorizar?
  - todos visam a usar a CPU de modo eficiente
    - chaveamentos de contexto são caros
- Visão dos processos



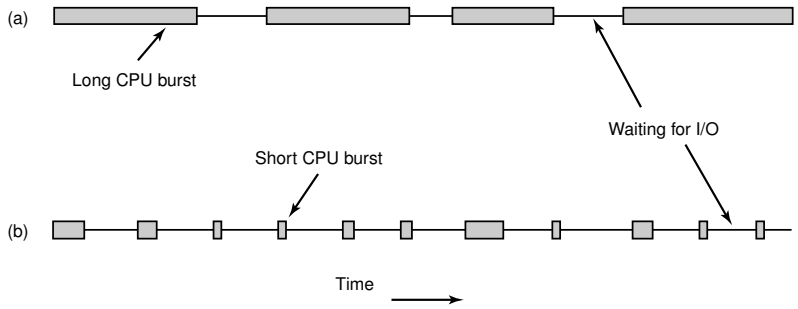





## Comportamento dos processos

- Em geral, processos alternam ciclos de uso de CPU com ciclos de requisição de E/S
  - o processo executa várias instruções de máquina e faz uma chamada de sistema solicitando um serviço do SO
- Existem duas grandes classes de processos
  - orientados a CPU (*CPU-bound*)
  - orientados a E/S (*I/O-bound*)
  - há processos que alternam essas características

## Representação do comportamento




(a) um processo orientado a CPU

(b) um processo orientado a E/S





## Quando escalonar

Existem diversas situações em que o escalonador é invocado

- na criação de um processo
- no encerramento de um processo
- quando um processo bloqueia
- quando ocorre uma interrupção de E/S
- quando ocorre uma interrupção de relógio
  - escalonamento preemptivo





## Escalonamento preemptivo e não preemptivo

- No escalonamento não preemptivo, um processo só pára de executar na CPU se quiser
  - invocação de uma chamada de sistema
  - liberação voluntária da CPU
- No escalonamento preemptivo um processo pode perder a CPU mesmo contra sua vontade
  - preempção por tempo (mais comum)
  - preempção por prioridade
    - chegada de um processo mais prioritário
  - além das possibilidades do não preemptivo





## Categorias de algoritmos

- Existem três categorias básicas de algoritmos de escalonamento
- **Lote (batch)**
  - sem usuários interativos
  - ciclos longos são aceitáveis – menos preempções
- **Interativo**
  - com usuários interativos
  - ciclos curtos para que todos os processos progridam
- **Tempo real**
  - processos com requisitos temporais específicos

## Objetivos do algoritmo de escalonamento

- Quais os critérios podem ser usados para avaliar um algoritmo de escalonamento?
- **Vazão (throughput)**: número de jobs processados por hora
- **Tempo de retorno**: tempo médio do momento que um job é submetido até o momento em que foi terminado
  - sistemas em lote
- **Tempo de reação** (*response time*): tempo entre a emissão de um comando e a obtenção do resultado
  - sistemas interativos





## Algoritmos de escalonamento

- Escalonamento para sistemas em lote
  1. Primeiro a chegar, primeiro a ser servido (FCFS)
  2. Job mais curto primeiro (SJF)
  3. Próximo de menor tempo restante (SRTN)
- Escalonamento para sistemas interativos
  1. Alternância circular (*round-robin*)
  2. Por prioridades
  3. Filas múltiplas
  4. Fração justa





## FCFS

- Processos são atendidos por ordem de chegada
  - primeiro a chegar, primeiro a ser servido
  - *first come, first served (FCFS)*
- O processo escalonado usa a CPU por quanto tempo quiser – não preemptivo
  - até encerrar, bloquear ou entregar o controle
- Simples de implementar
- Não diferencia processos orientados a CPU e orientados a E/S
  - pode prejudicar os orientados a E/S





## SJF (*Shortest Job First*)

- Os processos mais curtos são atendidos primeiro
  - mais curto = menor tempo de CPU
- Não preemptivo
- Algoritmo com menor tempo médio de retorno
- Premissas
  - todos os jobs estão disponíveis simultaneamente
  - a duração dos ciclos de CPU é conhecida a priori

## Próximo de menor tempo restante

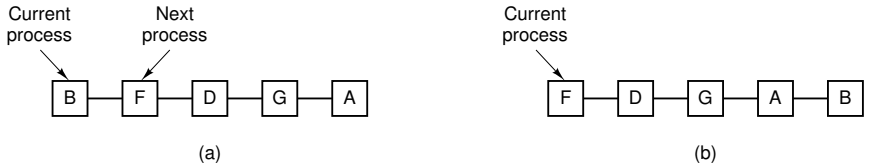
- *Shortest remaining time next (SRTN)*
- Variante preemptiva do SJF
- Quando chega um novo processo, seu tempo de CPU é comparado com o tempo restante do processo que está executando
  - se for menor, o processo atual é preemptado e o novo processo escalonado em seu lugar
- Garante bom desempenho para jobs curtos
- Requer tempos conhecidos de CPU





## Alternância circular (*round-robin*)

- Cada processo que ganha a CPU executa durante um determinado tempo (o **quantum**)
- Se o processo não liberar a CPU, ao final do quantum ele perde o processador e volta para a fila de prontos
  - algoritmo preemptivo
- Exemplo: B usa todo o seu quantum








## Determinando o quantum

- A decisão de projeto mais importante no *round-robin* é o tamanho do quantum
- Quanto menor o quantum, maior o overhead
  - tempo para chaveamento de contexto se aproxima do tempo de execução
- Quanto maior o quantum, pior o tempo de reação
  - ocorrem menos preempções
  - processo demora mais a ser escalonado
  - prejudica processos orientados a E/S
- Na prática, o quantum fica entre 20 e 100 ms





## Escalonamento por prioridades (1/3)

- Nem todos os processos têm a mesma prioridade
  - o antivírus não deve prejudicar a exibição de um vídeo, por exemplo
- Escalonamento por prioridades
  - cada processo recebe uma prioridade
  - o processo de maior prioridade executa
  - para evitar que processos mais prioritários executem indefinidamente, a prioridade pode ser periodicamente reduzida
  - prioridade preemptiva vs não preemptiva

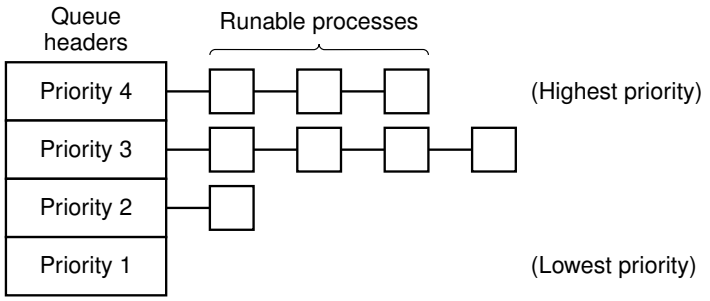
# Escalonamento por prioridades (2/3)

- Prioridades podem ser estáticas ou dinâmicas
  - igual à fração do último quantum usada, p.ex.
- É comum agrupar os processos em classes de prioridades
  - prioridade entre as classes
  - *round-robin* dentro de cada classe





# Escalonamento por prioridades (3/3)








# Inanição e envelhecimento

- No escalonamento por prioridades, os processos de baixa prioridade podem sofrer **inanição** (*starvation*)
  - nunca serem escalonados devido aos processos de alta prioridade monopolizarem o processador
    - processos de longa duração, que não bloqueiam
    - chegada constante de novos processos de alta prioridade
- A solução é usar um mecanismo de **envelhecimento** (*aging*)
  - aumenta a prioridade dos processos que estão há muito tempo na fila de prontos sem executar
    - prioridades dinâmicas, e não estáticas





# Filas múltiplas

- Variante do escalonamento por prioridades
- Cada classe de prioridade tem um quantum
  - classes mais prioritárias têm quantum menor
  - se o quantum acaba antes que o processo consiga concluir o ciclo de CPU, ele muda de prioridade
- Reduz a quantidade de chaveamentos de contexto para processos orientados a CPU
- Processos interativos têm alta prioridade
  - usuários de processos em lote descobriram que podiam acelerar seus processos usando o terminal

## Escalonamento por fração justa

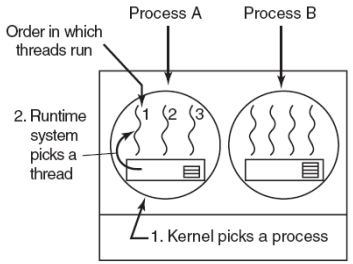
- *Fair share scheduling*
- Os algoritmos anteriores tratam todos os processos de forma igual
  - usuários com muitos processos têm mais tempo de CPU do que usuários com poucos processos
- A ideia do *fair share* é atribuir uma fração da CPU para cada usuário
  - o escalonador escolhe o processo a executar de modo a respeitar essas frações
- Outras possibilidades existem, dependendo da noção de "justiça"





## Escalonamento de threads de usuário

- O núcleo escalona um processo, e o escalonador do runtime pode chavear entre as threads desse processo
  - preempção em geral voluntária (`pthread_yield()`), não por tempo
  - qualquer algoritmo de escalonamento pode ser usado, inclusive um específico para a aplicação



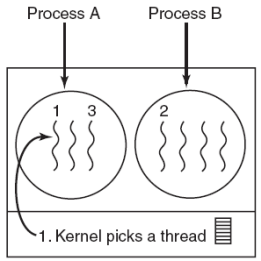
(a)





## Escalonamento de threads de núcleo

- O escalonador do SO escolhe uma thread de qualquer processo
  - processo pode ou não ser considerado pelo algoritmo
    - como chavear processos é mais caro que chavear threads, escalonador pode dar preferência a outra thread do mesmo processo
  - escalonamento específico para a aplicação é inviável



(b)





## Escalonamento no Linux

- Linux possui threads de kernel
  - escalonamento de threads, não de processos
- Para o escalonamento, três classes de threads são consideradas
  - FIFO (FCFS) de tempo real (`SCHED_FIFO`)
  - round robin de tempo real (`SCHED_RR`)
  - tempo compartilhado (`SCHED_OTHER`)
- Diferentes prioridades são associadas às classes
  - threads de tempo real: 0–99
  - demais threads: 100–139
  - as threads de tempo real são apenas mais prioritárias
    - não há deadlines ou garantias temporais
- O escalonador simplesmente escolhe a primeira thread pronta na fila mais prioritária
  - menor número de prioridade
  - escalonador $O(1)$ → kernel 2.6 (pré 2.6.23)
  - kernel 2.6.23 e posteriores: CFS (*Completely Fair Scheduler*)

# Threads SCHED_FIFO

- Escalonadas em ordem
- Uma thread executa até
  - bloquear
  - liberar voluntariamente a CPU
    - invocando `sched_yield()`
  - uma thread mais prioritária ficar pronta
    - chegar no sistema ou ser desbloqueada
- A thread suspensa volta para a fila de sua mesma prioridade





# Threads SCHED_RR

- Semelhantes às threads FIFO, mas podem ser preemptadas por tempo
- Quando uma thread perde o processador, o tempo de CPU usado é descontado do quantum
  - quando ela retoma o processador, o valor remanescente é usado
- Quando o quantum zera, ele é restaurado ao valor inicial e a thread colocada no final da fila da sua prioridade
- Threads de tempo real usam apenas prioridades estáticas





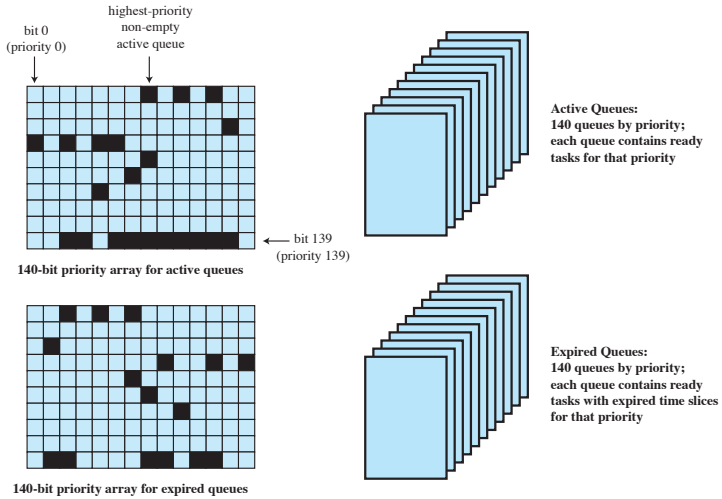
# Threads SCHED_OTHER

- A prioridade de uma thread não-TR fica entre 100 e 139
- Cada CPU no sistema possui uma **runqueue**
  - conjuntos de threads ativas e expiradas
    - até 140 filas em cada um
- O escalonador seleciona a thread mais prioritária entre as filas de threads ativas da CPU
  - thread selecionada executa no máximo pela duração do seu quantum
  - quando o quantum encerra, a thread vai para o final da fila de threads expiradas
  - se a thread bloqueia antes do encerramento do quantum, após ser desbloqueada ela volta para o final da fila de threads ativas com a mesma prioridade, podendo executar pelo tempo remanescente no quantum
  - quando não existem mais threads ativas, as filas de ativas e expiradas são trocadas, e o escalonamento é retomado
    - garante que não ocorra inanição





# Runqueues no escalonador $O(1)$



## Prioridade das threads

- Prioridade default de uma thread não-TR é 120
- Um valor de *nice* $(-20 \ldots +19)$ pode ser definido para a thread
  - *nice* é somado à prioridade estática
  - ideia é reduzir a prioridade de threads CPU-bound não críticas
  - apenas o `root` pode definir $nice < 0$
- O tamanho do quantum varia com a prioridade
  - prioridade = 100, quantum = 800 ms
  - prioridade = 110, quantum = 600 ms
  - prioridade = 120, quantum = 100 ms (default)
  - prioridade = 130, quantum = 50 ms
  - prioridade = 139, quantum = 5 ms





## Prioridades dinâmicas

- A prioridade efetivamente usada pelo escalonador é computada dinamicamente
- Heurísticas definem um valor de **bônus** para cada thread, em função do seu comportamento (CPU-bound ou I/O-bound)
  - ideia é recompensar threads I/O-bound e punir threads CPU-bound
  - função de quanto tempo a thread ficou bloqueada no último segundo (`sleep_avg`)
  - bônus é um valor no intervalo $-5 \ldots +5$
- Prioridade dinâmica leva em consideração a prioridade estática, o valor de *nice* e o bônus





## Escalonamento em multiprocessadores

- Em sistemas com vários processadores, existem benefícios de se manter uma thread sempre na mesma CPU
  - aproveitamento da cache do processador
  - isso é chamado de **afinidade de processador**
- *Runqueues* por CPU favorecem essa afinidade
- Periodicamente o escalonador balanceia a carga entre as CPUs, levando em consideração desempenho e afinidade





## Estruturas adicionais

- Escalonador considera apenas threads escalonáveis
  - conteúdo das *runqueues*
- Threads bloqueadas são colocadas em **waitqueues**
  - cada evento pelo qual uma thread pode estar esperando possui uma *waitqueue*
  - facilita o desbloqueio de threads
- Filas no kernel são gerenciadas com o auxílio de **spinlocks** para garantir exclusão mútua
  - variáveis do tipo `lock`
  - usam instrução tipo TSL em multiprocessadores

# Limitações do escalonador $O(1)$

- Escalonador $O(1)$ usa heurísticas para determinar quais threads são interativas
  - nem sempre a heurística está correta, gerando comportamento indesejado do escalonador
  - código para gerenciar as heurísticas é complexo e difícil de manter
- Para melhorar o tempo de reação do sistema, threads novas e threads "interativas" entram na fila de threads ativas
  - pode levar a inanição em alguns casos, ou atrasar significativamente o escalonamento de threads na fila de expiradas
- Tentando contornar essas limitações, a partir do kernel 2.6.23 foi adotado o *Completely Fair Scheduler (CFS)*





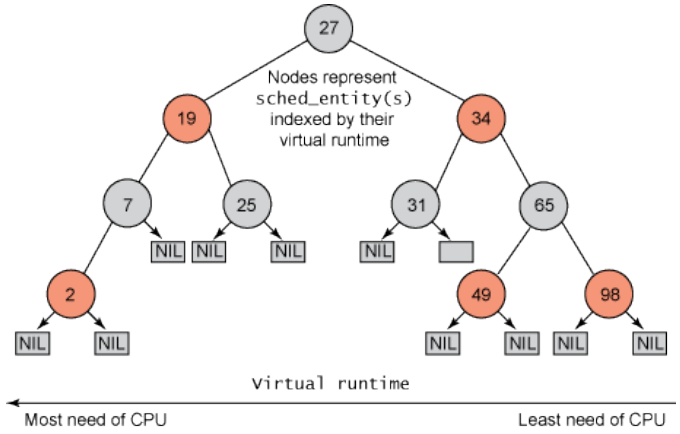
# Completely Fair Scheduler (CFS)

- O CFS tenta melhorar a justiça entre threads da classe `SCHED_OTHER` (não necessariamente entre usuários) em relação ao $O(1)$
  - grupos são usados para justiça entre usuários ou processos
- Threads são ordenadas de acordo com o seu tempo virtual de execução
  - tempo efetivo de execução em ns, ponderado pela prioridade
- Thread com o menor tempo virtual é a próxima a executar
  - a fatia de tempo é calculada dinamicamente, em função da carga no sistema
  - limites superior e inferior são usados para manter a fatia dentro de uma faixa aceitável
- O tempo virtual das threads é mantido em uma árvore rubro-negra
  - uma árvore por CPU
  - menor tempo virtual é sempre o elemento mais à esquerda





# Completely Fair Scheduler (CFS)




**Fonte**: http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/





# Bibliografia Básica

Andrew S. Tanenbaum e Herbert Bos.
*Sistemas Operacionais Modernos, 4ª Edição*. Capítulo 2.
Pearson Prentice Hall, 2016.

Abraham Silberchatz, Greg Gagne e Peter Baer Galvin.
*Fundamentos de Sistemas Operacionais, 6ª Edição*.
LTC - Livros Técnicos e Científicos Editora S.A., 2004.