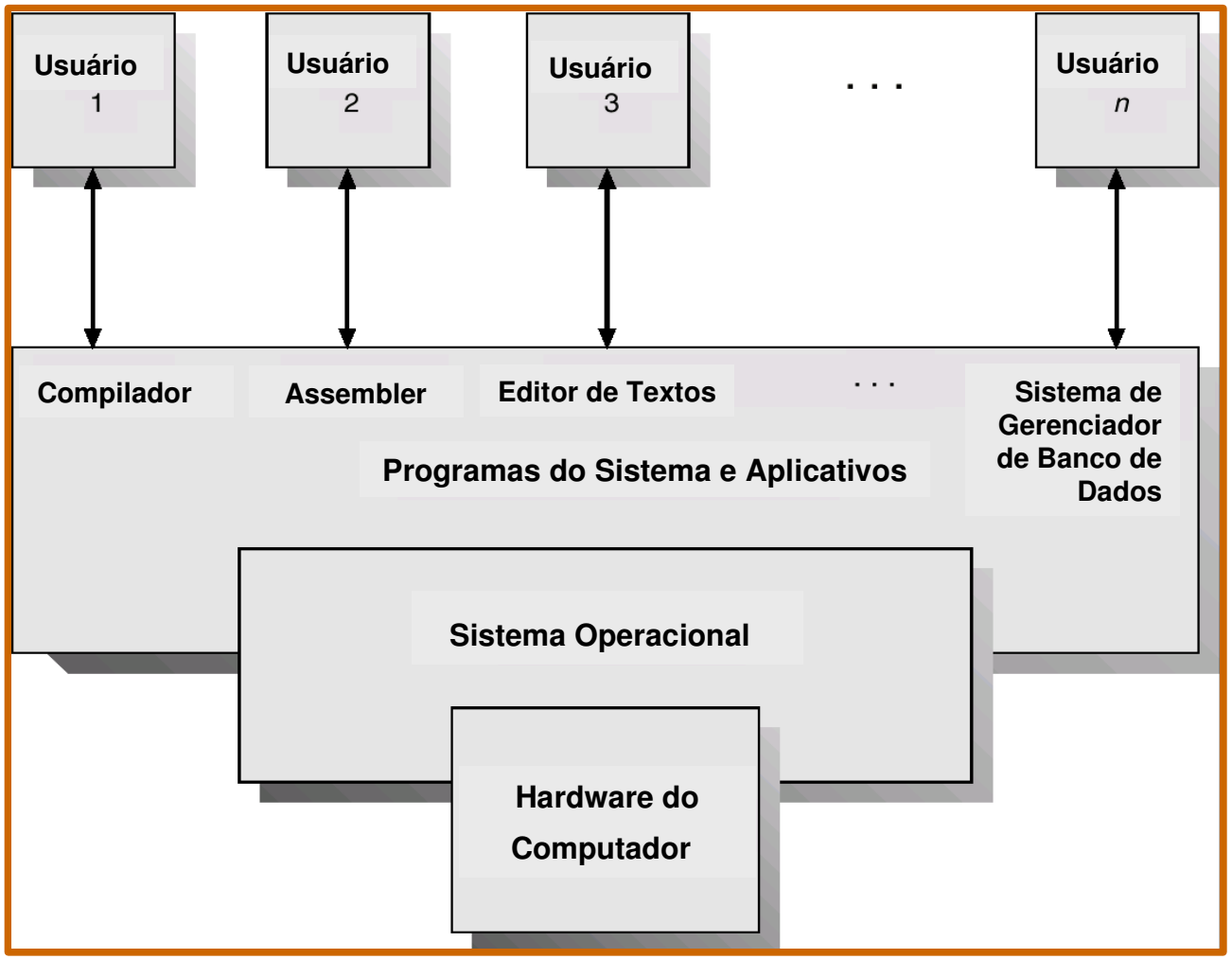
S.O.: Conceitos Básicos

- Camada de software localizada entre o hardware e os programas que executam tarefas para o usuário;
  - Acessa os periféricos
  - Entrada e Saída
  - Esconde os detalhes do hardware para o programador

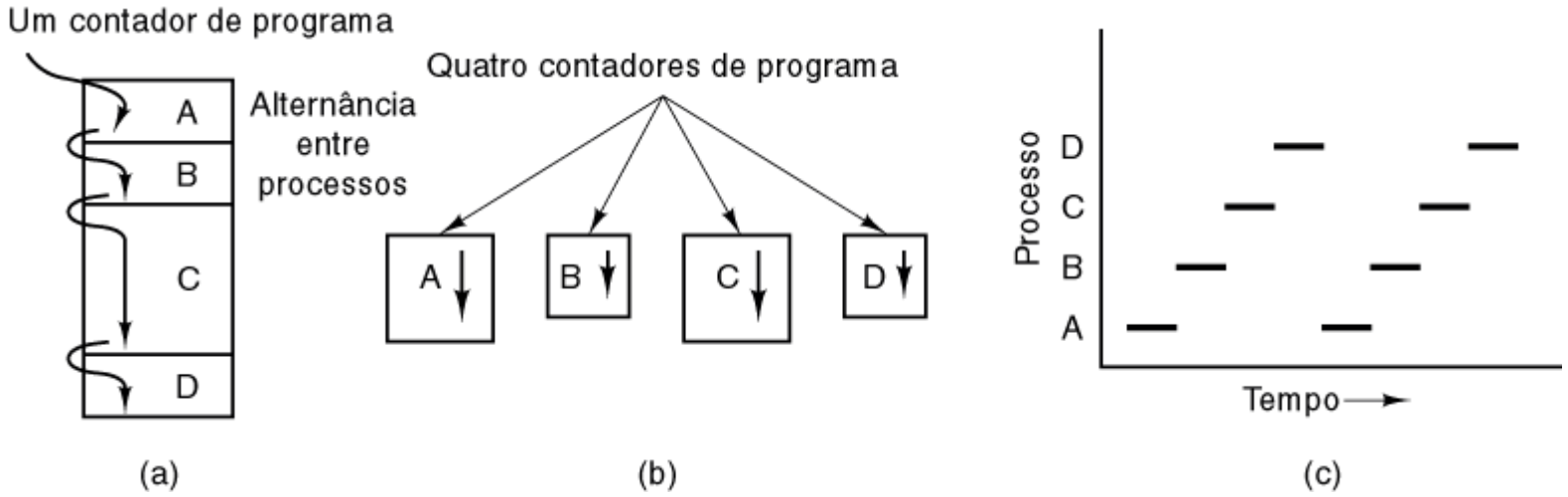
Visão Abstrata dos Componentes do Sistema



Conceito de Processo

- Um sistema operacional executa uma variedade de programas:
  - Sistemas Batch– tarefas
  - Sistemas de tempo compartilhado – programas de usuário ou tarefas
- O livro texto usa os termos tarefa e processo quase que indistintamente
- Processo – um programa em execução
  - A execução do processo precisa ocorrer de maneira seqüencial
- Um processo inclui:
  - contador de programa
  - pilha
  - seção de dados

Processos
O Modelo de Processo



- Multiprogramação de quatro programas
- Modelo conceitual de 4 processos sequenciais, independentes
- Somente um programa está ativo a cada momento

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
3. Solicitação do usuário para criar um novo processo
4. Início de um job em lote

Término de Processos

Condições que levam ao término de processos

2. Saída normal (voluntária)
3. Saída por erro (voluntária)
4. Erro fatal (involuntário)
5. Cancelamento por um outro processo (involuntário)

Hierarquias de Processos

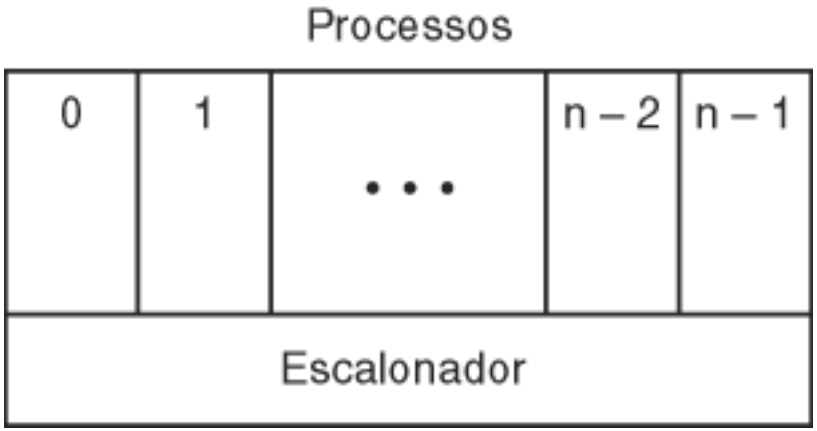
- Pai cria um processo filho, processo filho pode criar seu próprio processo
- Formam uma hierarquia
  - UNIX chama isso de “grupo de processos”
- Windows não possui o conceito de hierarquia de processos
  - Todos os processos são criados iguais

Estados de Processos (1)

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

- Possíveis estados de processos
  - em execução
  - bloqueado
  - pronto
- Mostradas as transições entre os estados

Estados de Processos (2)

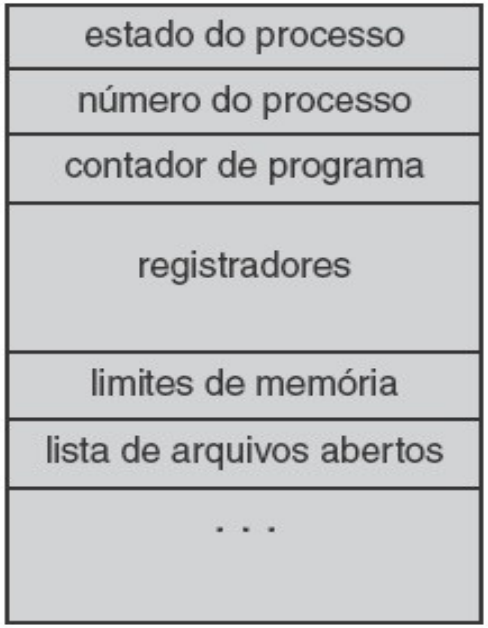


- Camada mais inferior de um SO estruturado por processos
  - trata interrupções, escalonamento
- Acima daquela camada estão os processos sequenciais

Bloco de Controle de Processo (Process Control Block)

- Informações associadas a cada processo
- Estado do processo
- Contador de programa
- Registradores da CPU
- Informação de escalonamento de CPU
- Informação de gerenciamento de memória
- Informação contábil
- Informação de status de E/S

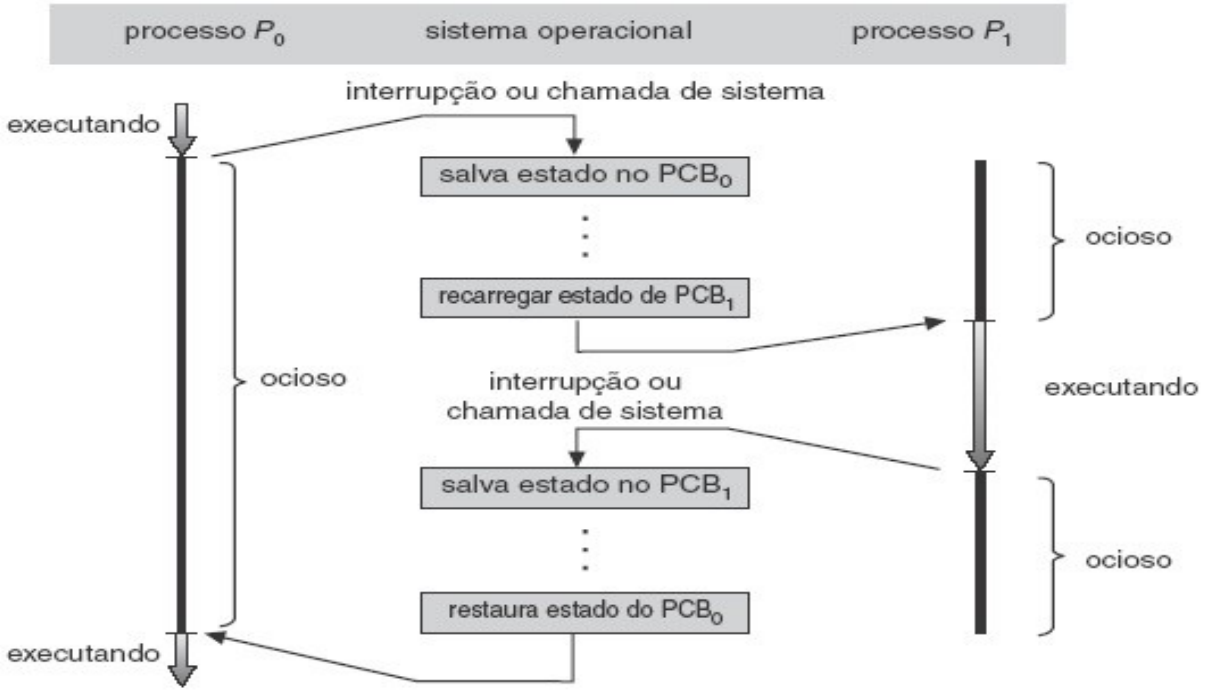
Bloco de Controle de Processo (Process Control Block)



Troca de Contexto

- Quando a CPU passa para outro processo, o sistema precisa salvar o estado do processo antigo e carregar o estado salvo do novo processo
- O tempo da troca de contexto é custo adicional; o sistema não realiza qualquer trabalho útil durante a troca
- O tempo depende do suporte do hardware

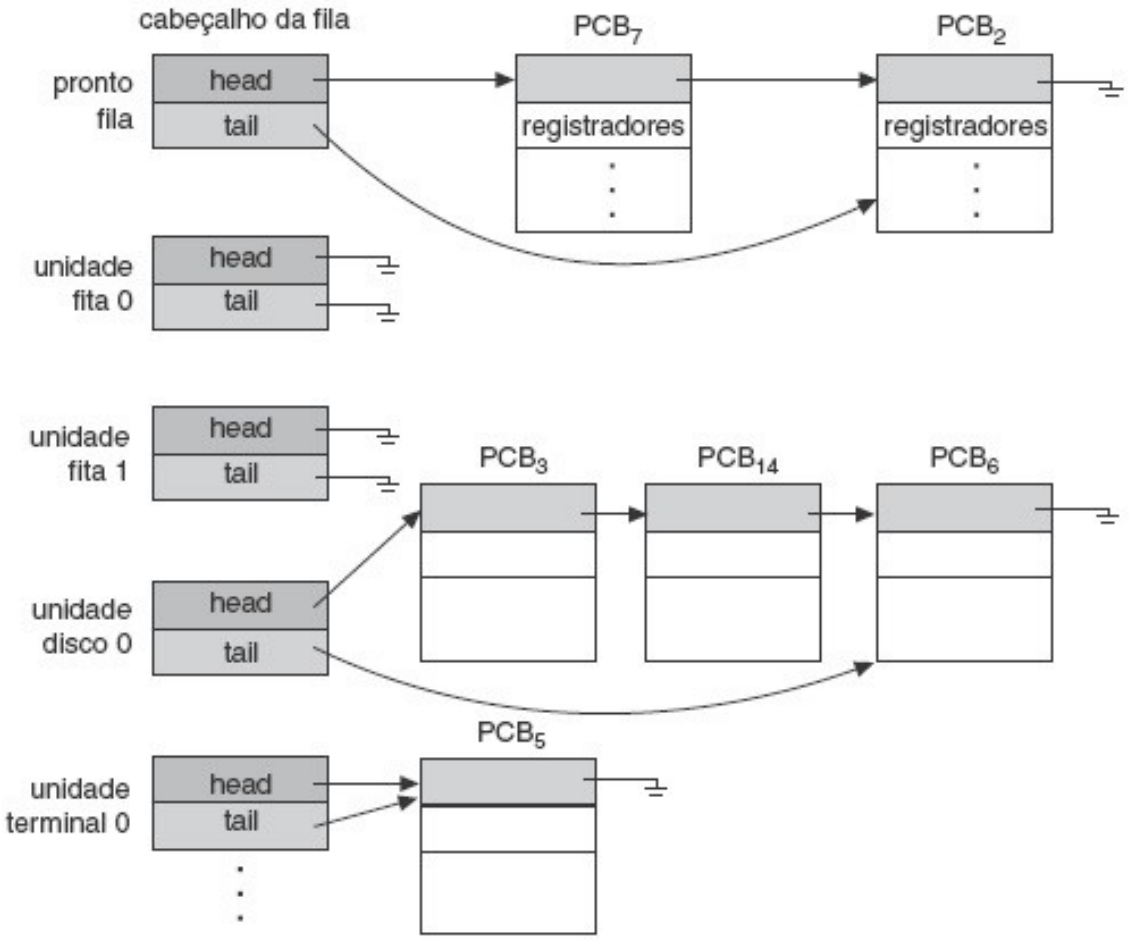
Troca da CPU de um Processo para Outro



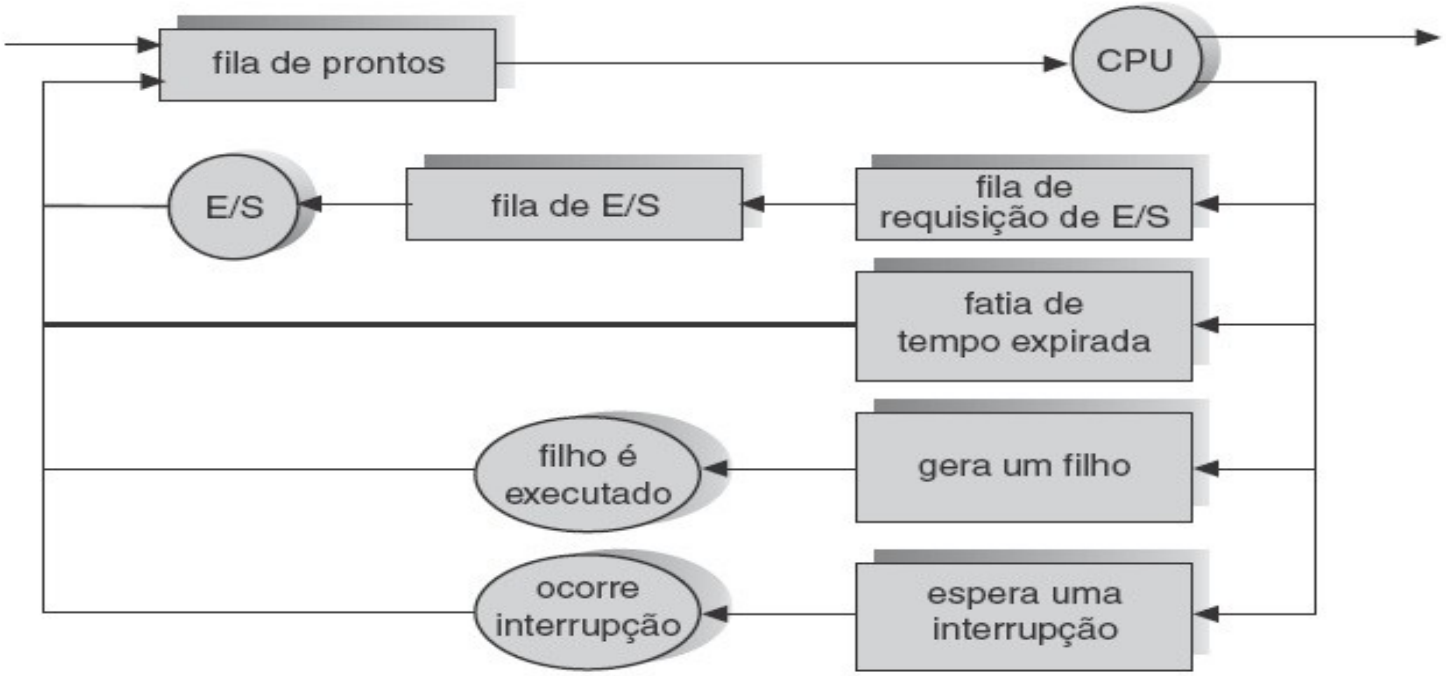
Filas de Escalonamento de Processo

- **Fila de tarefas (Job Queue)** – conjunto de todos os processos no sistema
- **Fila de prontos (Ready Queue)** – conjunto de todos os processos residindo na memória principal que estão prontos e esperando para serem executados
- **Fila de dispositivo (Device Queue)** – conjunto dos processos esperando um dispositivo de E/S
- Migração de processo entre as diversas filas

Fila de Prontos e Várias Filas de Dispositivo de E/S



Representação do Escalonamento de Processos



Implementação de Processos (1)

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		

Campos da entrada de uma tabela de processos

Implementação de Processos (2)

1. O hardware empilha o contador de programa etc.
2. O hardware carrega o novo contador de programa a partir do vetor de interrupção.
3. O procedimento em linguagem de montagem salva os registradores.
4. O procedimento em linguagem de montagem configura uma nova pilha.
5. O serviço de interrupção em C executa (em geral lê e armazena temporariamente a entrada).
6. O escalonador decide qual processo é o próximo a executar.
7. O procedimento em C retorna para o código em linguagem de montagem.
8. O procedimento em linguagem de montagem inicia o novo processo atual.

Esqueleto do que o nível mais baixo do SO faz quando ocorre uma interrupção

Criação de Processo

- Os processos pai criam processos filhos, que, por sua vez, criam outros processos, formando uma árvore de processos
- Compartilhamento de recursos
  - Pai e filhos compartilham todos os recursos
  - Filhos compartilham um subconjunto dos recursos do pai
  - Pai e filho não compartilham recurso algum
- Execução
  - Pai e filhos são executados concorrentemente
  - Pai espera até que os filhos terminem

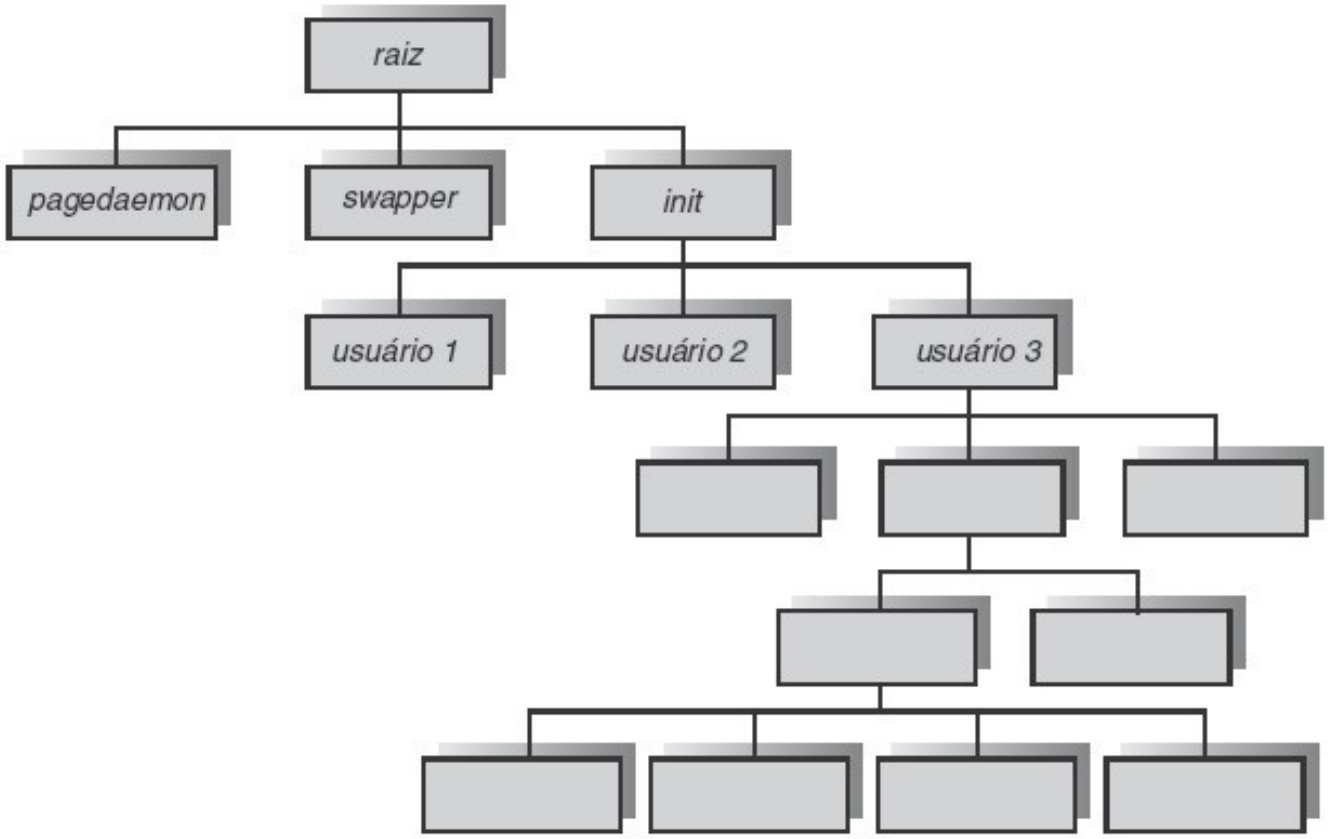
Criação de Processo

- Espaço de endereços
  - Cópia filha do processo pai
  - Filho contém um programa carregado
- Exemplos do UNIX
  - Chamada de sistema **fork** cria novo processo
  - Chamada de sistema **exec** usada após um **fork** para substituir o espaço de memória do processo por um novo programa

Programa em C
criando um processo separado

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char **argv)
{
int pid;
      /* cria outro processo */
      pid = fork();
      if (pid < 0) {                    /* ocorreu erro na execução do
      Fork */
            fprintf(stderr, “Falha no Fork ");
            exit(-1);
      }
      else if (pid == 0) {                        /* processo filho */
            execlp("/bin/ls","ls",NULL);
            exit(1);
      }
      else { /* processo pai */
            wait(NULL);             /* pai espera o término do filho */
            printf(“Filho terminou ");
            exit(0);
      }
```

Árvore de Processos no UNIX



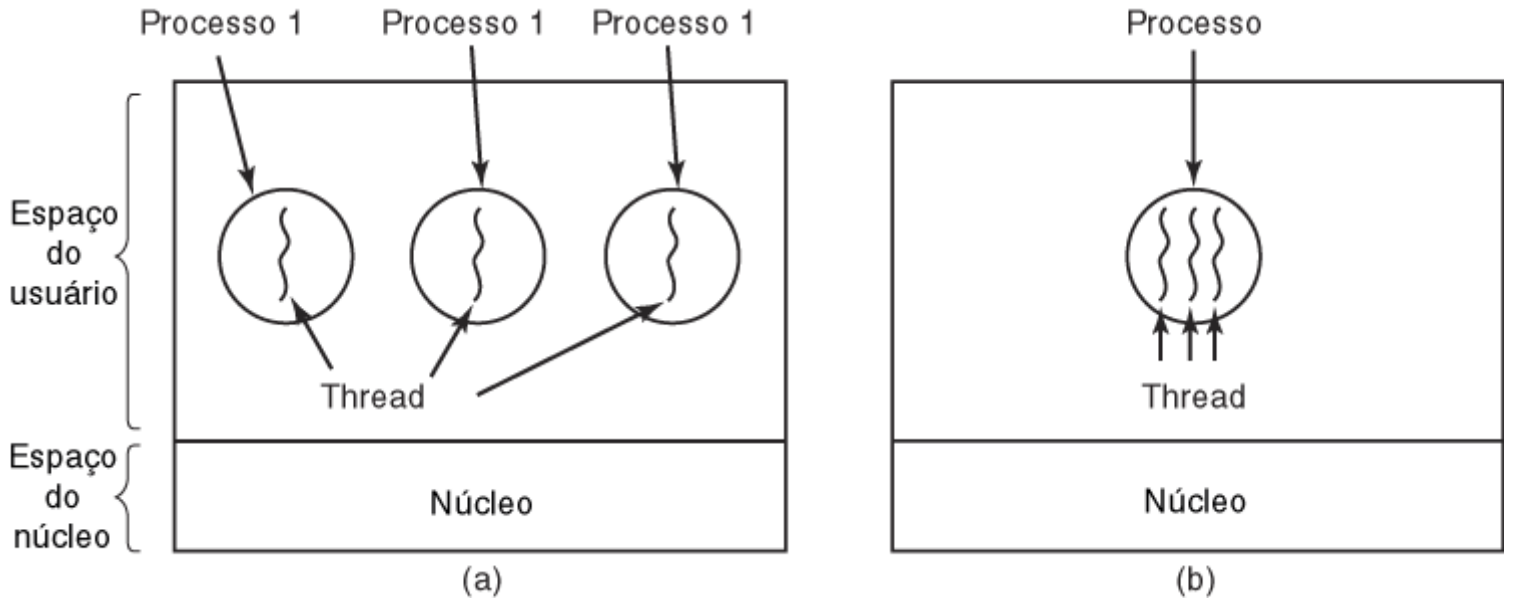
Término de processo

- O processo executa sua instrução final e pede ao sistema operacional que o termine (**exit**())
  - Envia os dados de saída do filho para o pai (através do **wait**())
  - Os recursos do processo são desalocados pelo sistema operacional

Término de processo

- O pai pode terminar a execução dos processos filhos (**abort()**)
  - O filho excedeu os recursos alocados
  - A tarefa atribuída ao filho não é mais necessária
  - Se o pai está terminando
    - Alguns sistemas operacionais não permitem que o filho continue se seu pai tiver terminado
      - Todos os filhos terminaram – término em cascata

Threads
O Modelo de Thread (1)



(a) Três processos cada um com um thread

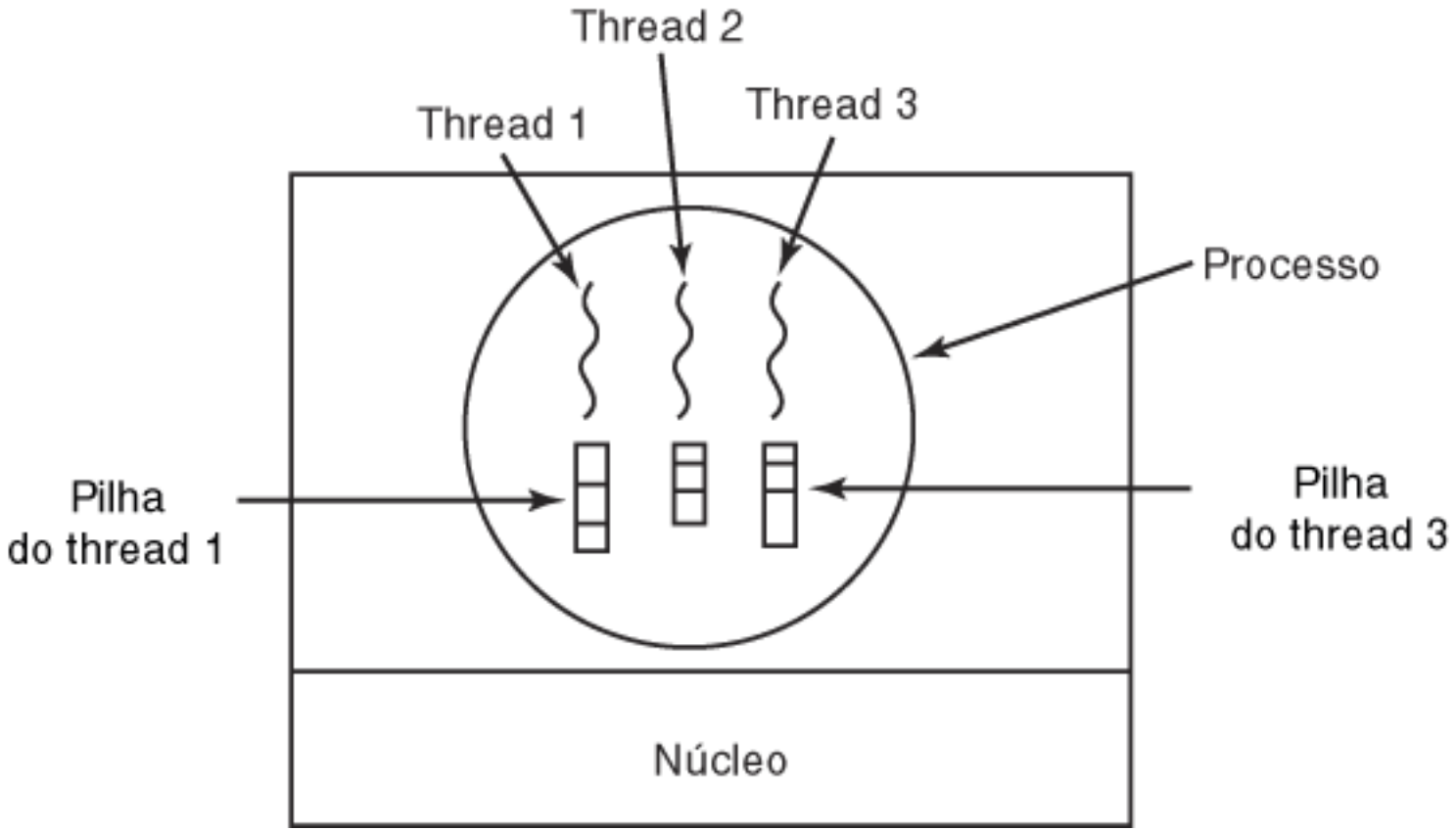
(b) Um processo com três threads

O Modelo de Thread (2)

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	

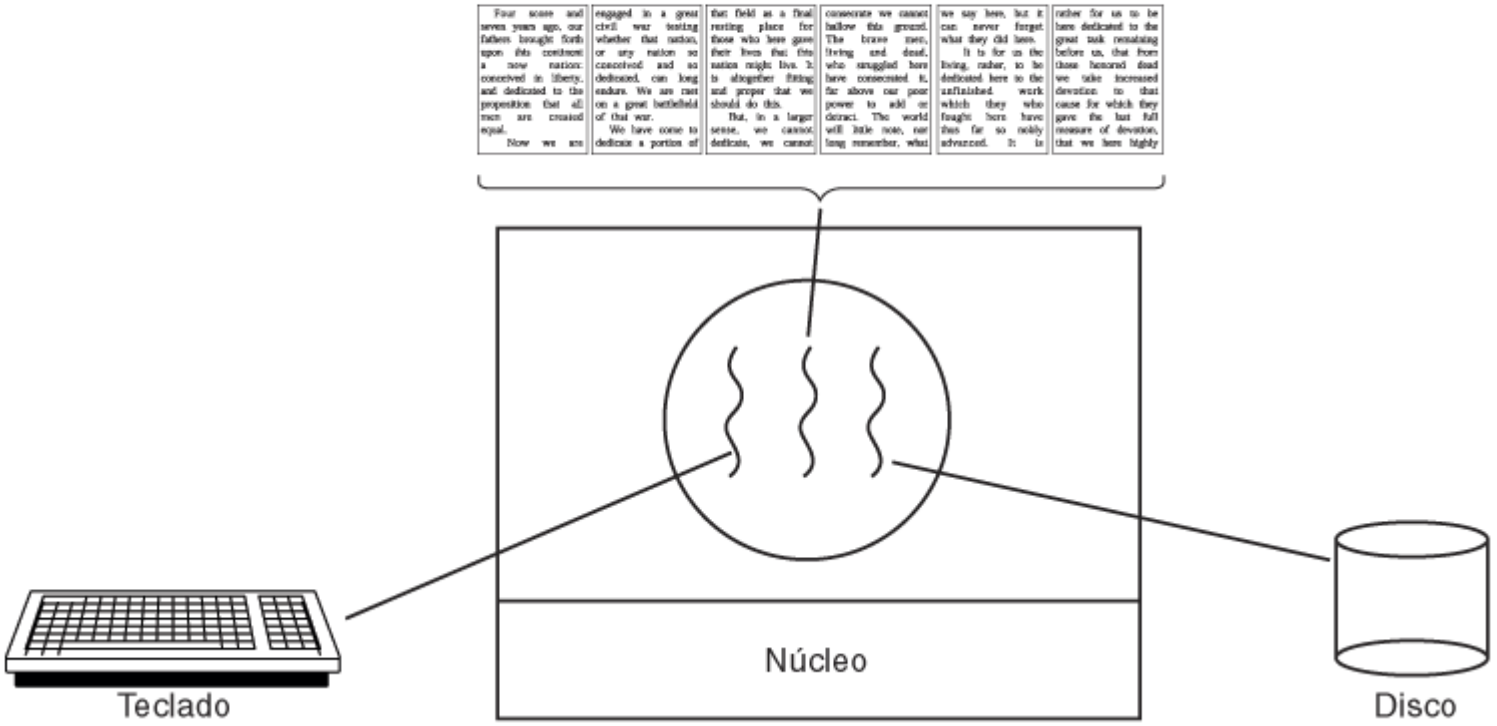
- Items compartilhados por todos os threads em um processo
- Itens privativos de cada thread

O Modelo de Thread (3)



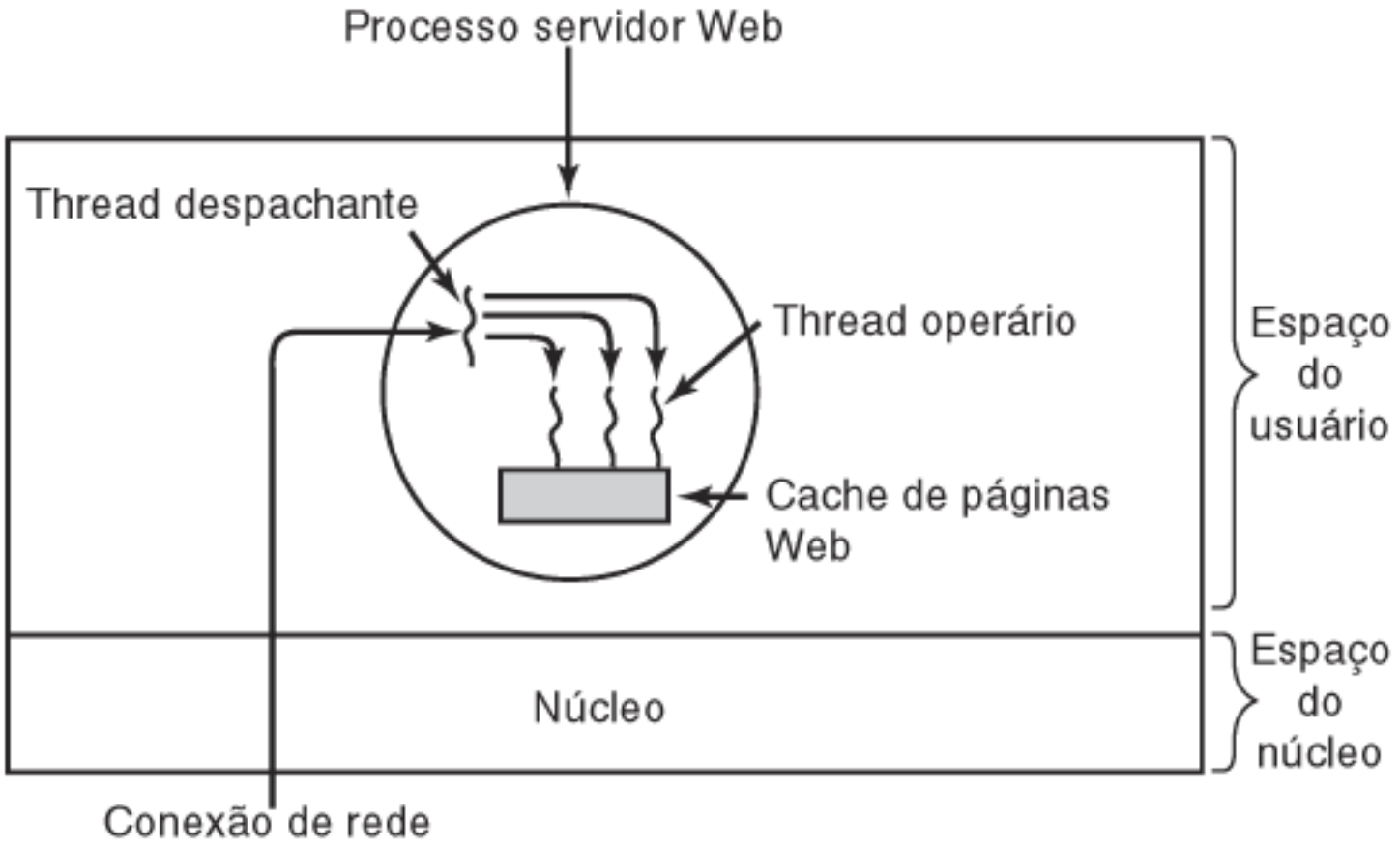
Cada thread tem sua própria pilha

Uso de Thread (1)



Um processador de texto com três threads

Uso de Thread (2)



Um servidor web com múltiplos threads

Uso de Thread (3)

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

(a)

```
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)

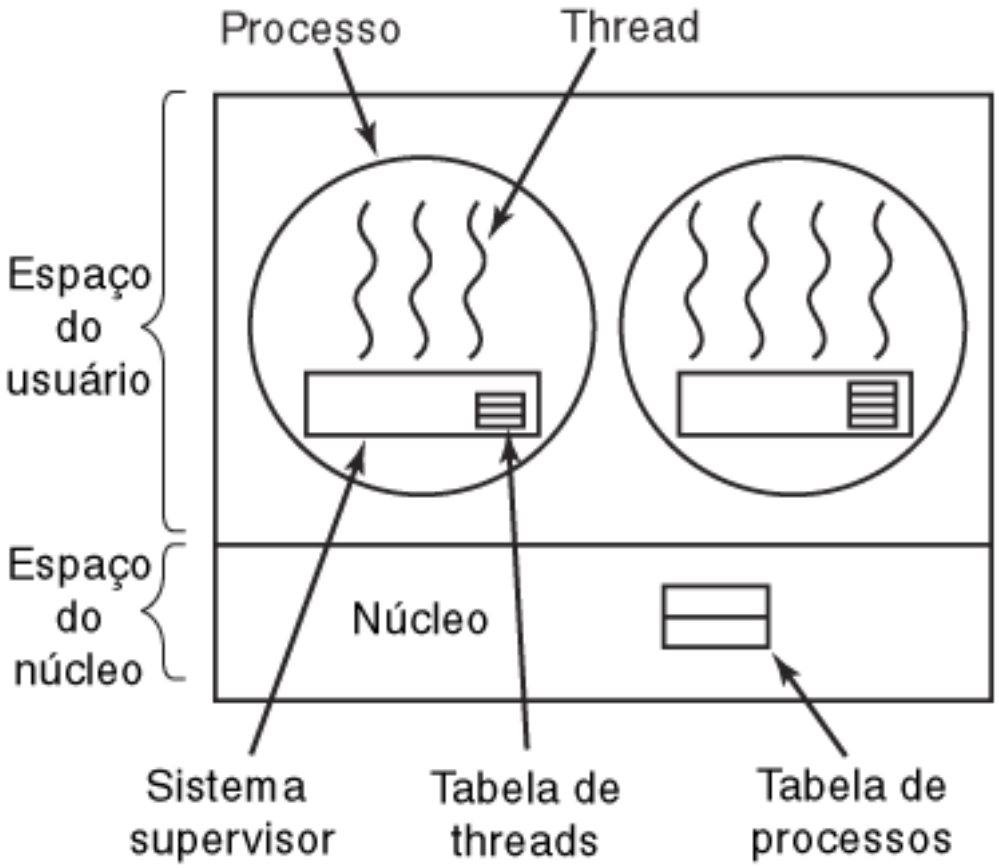
- Código simplificado do slide anterior
  - (a) Thread despachante
  - (b) Thread operário

Uso de Thread (4)

Modelo	Características
Threads	Paralelismo, chamadas ao sistema com bloqueio
Processo monothread	Sem paralelismo, chamadas ao sistema com bloqueio
Máquina de estados finitos	Paralelismo, chamadas ao sistema sem bloqueio, interrupções

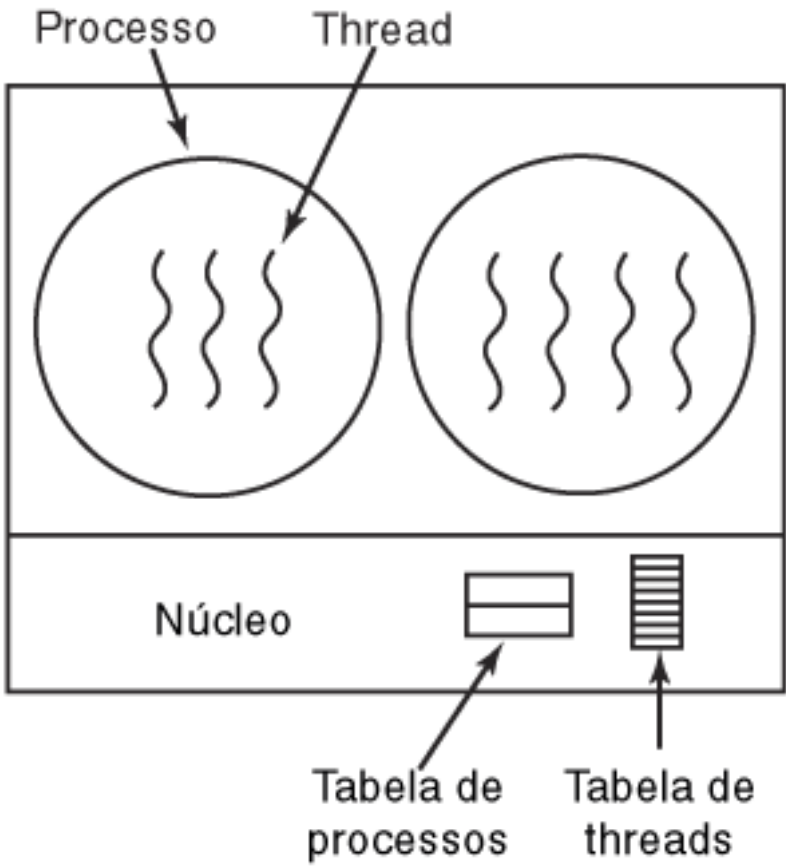
Três maneiras de construir um servidor

Implementação de Threads de Usuário



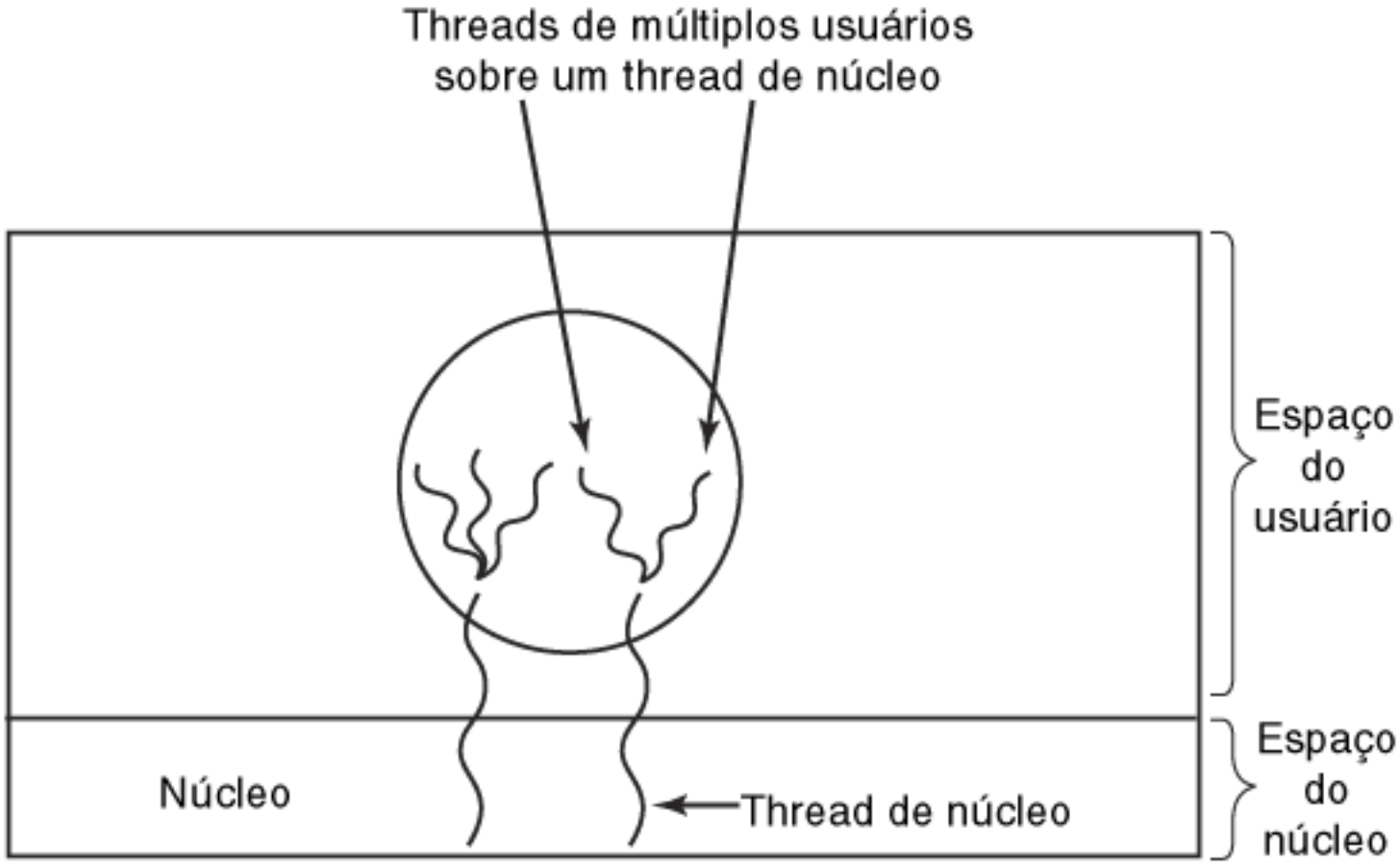
Um pacote de threads de usuário

Implementação de Threads de Núcleo



Um pacote de threads gerenciado pelo núcleo

Implementações Híbridas

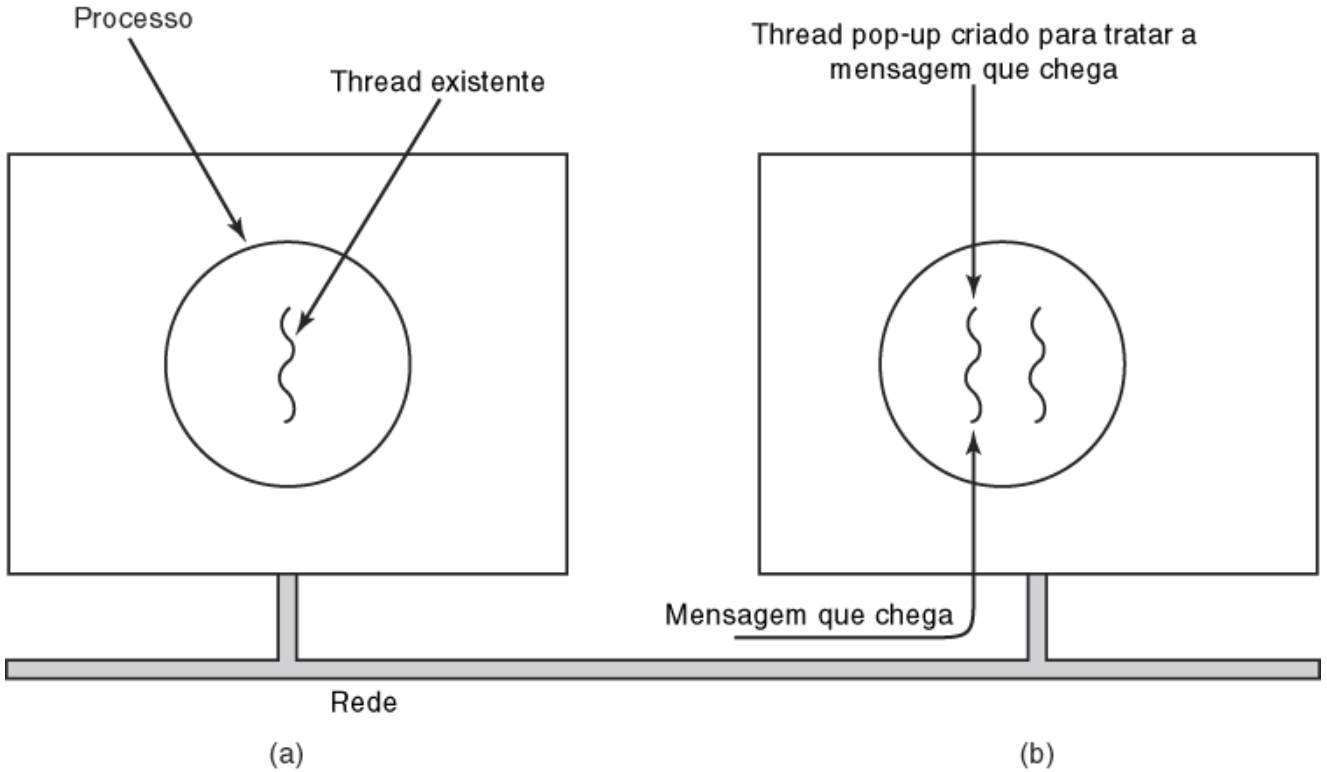


Multiplexação de threads de usuário sobre threads de núcleo

Ativações do Escalonador

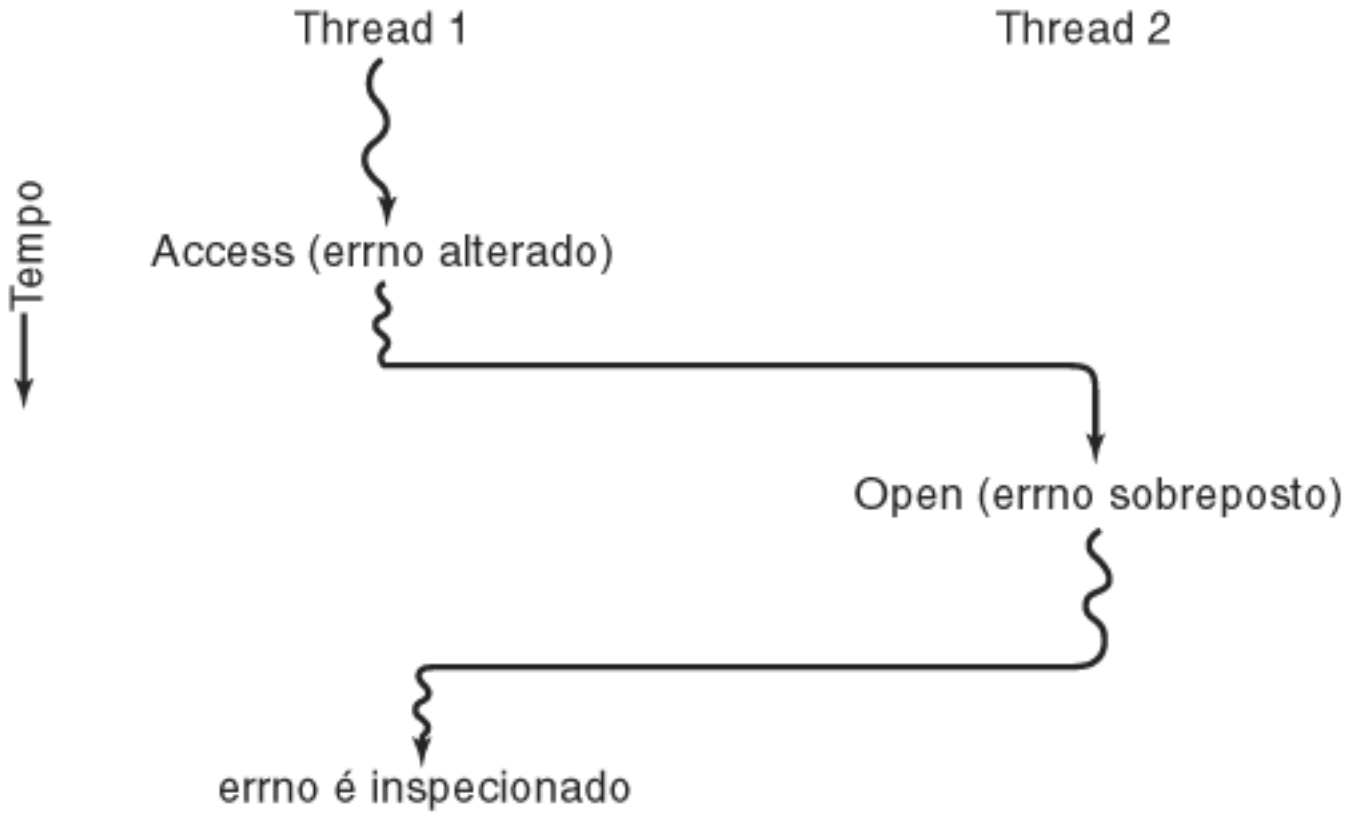
- Objetivo – imitar a funcionalidade dos threads de núcleo
  - ganha desempenho de threads de usuário
- Evita transições usuário/núcleo desnecessárias
- Núcleo atribui processadores virtuais para cada processo
  - deixa o sistema supervisor alocar threads para processadores
- Problema:
  Baseia-se fundamentalmente nos *upcalls* - o núcleo (camada inferior) chamando procedimentos no espaço do usuário (camada superior)

Threads Pop-Up



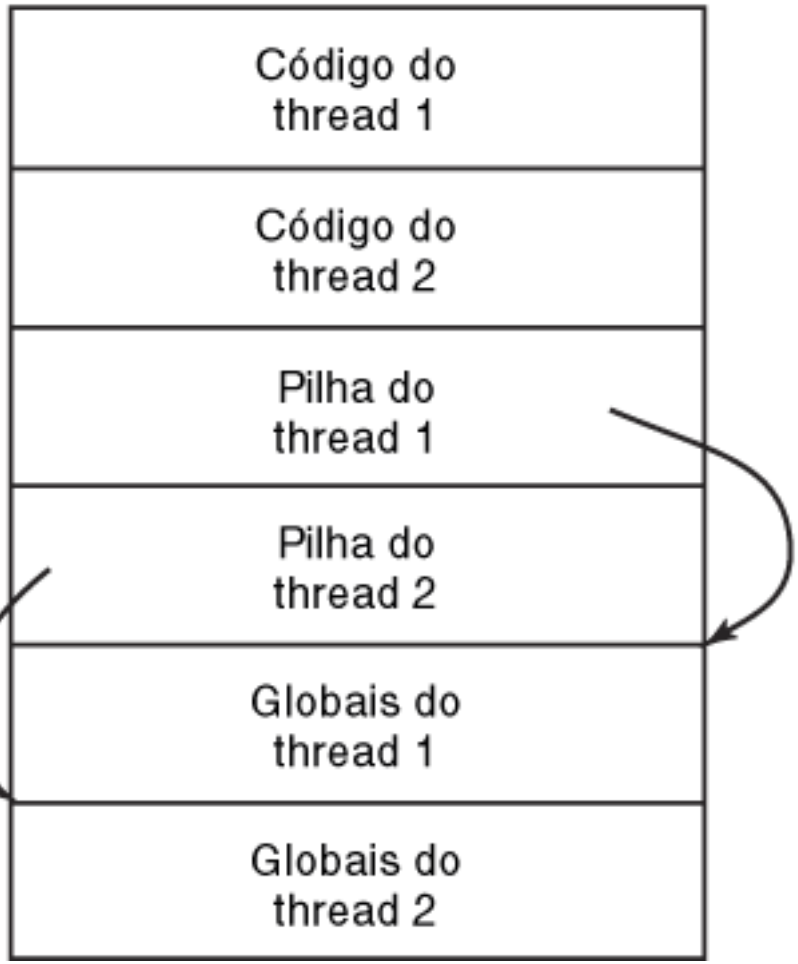
- Criação de um novo thread quando chega uma mensagem
  (a) antes da mensagem chegar
  (b) depois da mensagem chegar

Convertendo Código Monothread em Código Multithread (1)



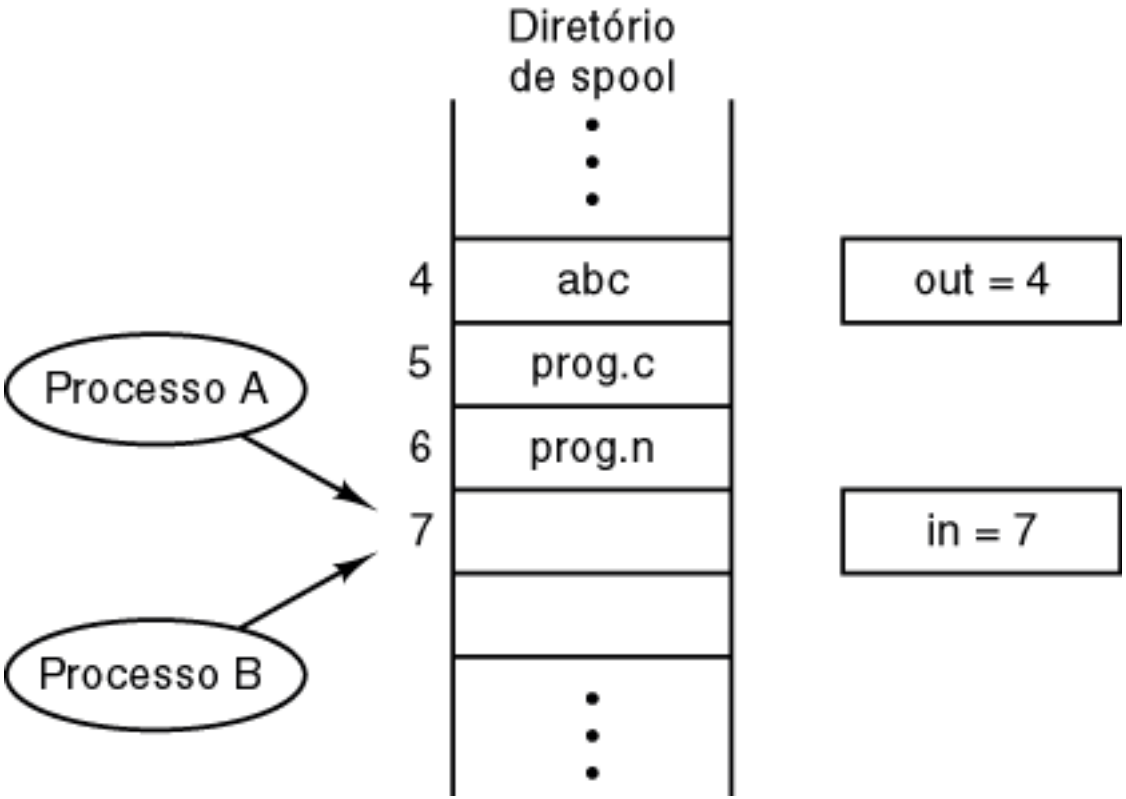
Conflitos entre threads sobre o uso de uma variável global

Convertendo Código Monothread em Código Multithread (2)



Threads podem ter variáveis globais privadas

Comunicação Interprocesso
Condições de Disputa



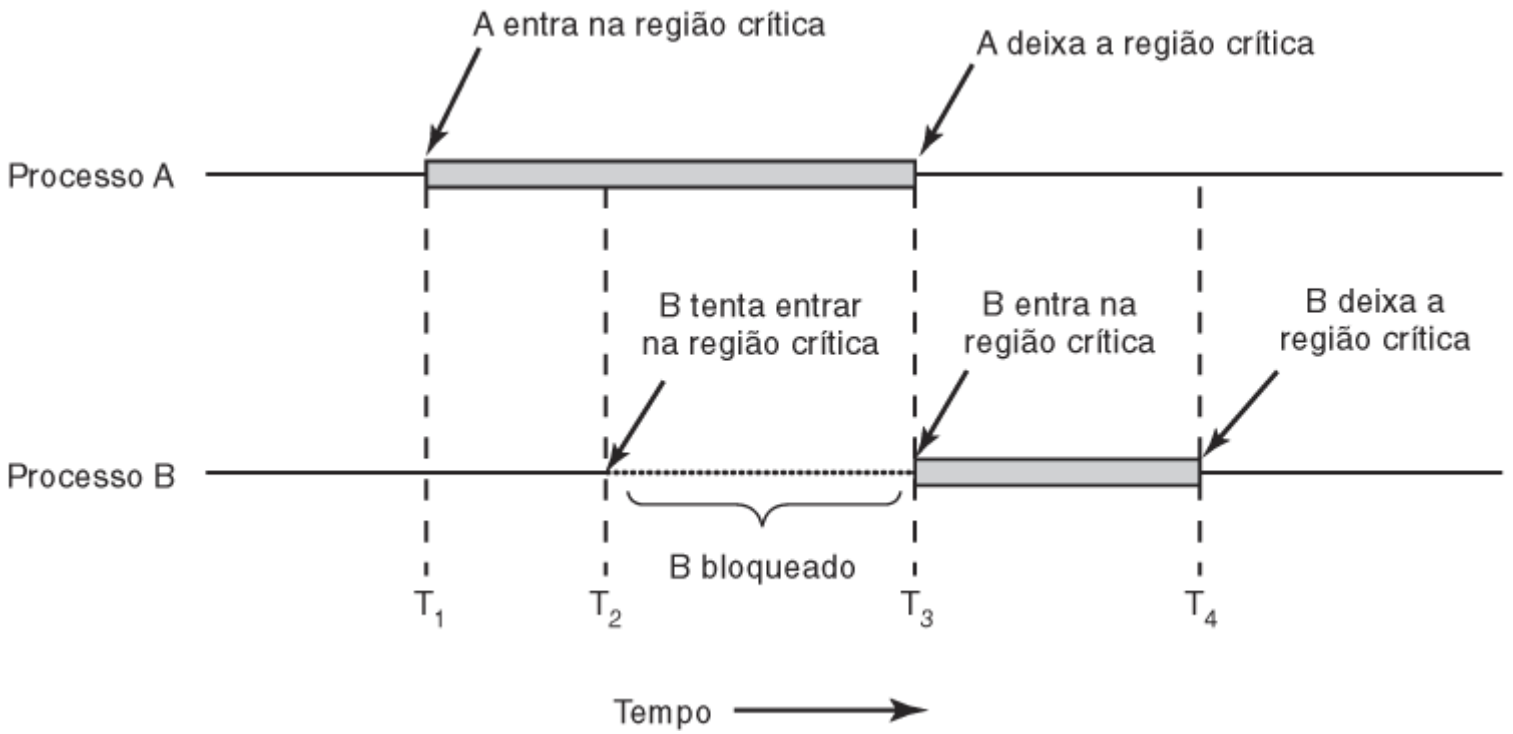
Dois processos querem ter acesso simultaneamente à memória compartilhada

Regiões Críticas (1)

Quatro condições necessárias para prover exclusão mútua:

1. Nunca dois processos simultaneamente em uma região crítica
2. Nenhuma afirmação sobre velocidades ou números 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

Regiões Críticas (2)



Exclusão mútua usando regiões críticas

Exclusão Mútua com Espera Ociosa (1)

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

(a)

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

(b)

Solução proposta para o problema da região crítica

(a) Processo 0. (b) Processo 1.

Exclusão Mútua com Espera Ociosa (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 */
}
```

Solução de Peterson para implementar exclusão mútua

Exclusão Mútua com Espera Ociosa (3)

```
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
```

Entrando e saindo de uma região crítica usando a instrução TSL

Dormir e Acordar

```
#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 */
     }
}
```

Problema do produtor-consumidor com uma condição de disputa fatal

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 */
      }
}
```

O problema do produtor-consumidor usando semáforos

Mutexes

```
mutex_lock:
        TSL REGISTER,MUTEX          | copia mutex para o registrador e o põe em 1
        CMP REGISTER,#0             | o mutex era zero?
        JZE ok                      | se era zero, o mutex estava desimpedido, portanto retorne
        CALL thread_yield           | o mutex está ocupado; escalone um outro thread
        JMP mutex_lock              | tente novamente mais tarde
ok:     RET | retorna a quem chamou; entrou na região crítica

mutex_unlock:
        MOVE MUTEX,#0               | põe 0 em mutex
        RET | retorna a quem chamou
```

Implementação de *mutex_lock* e *mutex_unlock*

Monitores (1)

```
monitor example
        integer i;
        condition c;

        procedure producer( );

        .
        .
        .
        end;

        procedure consumer( );

        .
        .
        .
        end;
end monitor;
```

Exemplo de um monitor

Monitores (2)

```
monitor ProducerConsumer
        condition full, empty;
        integer count;
        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;
        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;
        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;
procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

- Delineamento do problema do produtor-consumidor com monitores
  - somente um procedimento está ativo por vez no monitor
  - o buffer tem N lugares

Monitores (3)

```
public class ProducerConsumer {
    static final int N = 100;                   // constante com o tamanho do buffer
    static producer p = new producer( );        // instância de um novo thread produtor
    static consumer c = new consumer( );// instância de um novo thread consumidor
    static our_monitor mon = new our_monitor( );  // instância de um novo monitor

    public static void main(String args[ ]) {
        p.start( );                             // inicia o thread produtor
        c.start( );                             // inicia o thread consumidor
    }

    static class producer extends Thread {
        public void run( )  {                   // o método run contém o código do thread
            int item;
            while (true) {                      // laço do produtor
                item = produce_item( );
                mon.insert(item);
            }
        }
        private int produce_item( ) { ... }     // realmente produz
    }

    static class consumer extends Thread {
        public void run( )  {                   // método run contém o código do thread
            int item;
            while (true) {                      // laço do consumidor
                item = mon.remove( );
                consume_item (item);
            }
        }
        private void consume_item(int item) { ... }     // realmente consome
    }
    static class our_monitor {                  // este é o monitor
        private int buffer[ ] = new int[N];
        private int count = 0, lo = 0, hi = 0; // contadores e índices

        public synchronized void insert(int val) {
            if (count == N) go_to_sleep( );     // se o buffer estiver cheio, vá dormir
            buffer [hi] = val;                  // insere um item no buffer
            hi = (hi + 1) % N;                  // lugar para colocar o próximo item
            count = count + 1;                  // mais um item no buffer agora
            if (count == 1) notify( );          // se o consumidor estava dormindo, acorde-o
        }
        public synchronized int remove( ) {
            int val;
            if (count == 0) go_to_sleep( );     // se o buffer estiver vazio, vá dormir
            val = buffer [lo];                  // busca um item no buffer
            lo = (lo + 1) % N;                  // lugar de onde buscar o próximo item
            count = count - 1;                  // um item a menos no buffer
            if (count == N - 1) notify( );      // se o produtor estava dormindo, acorde-o
            return val;
        }
        private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
    }
}
```

Solução para o problema do produtor-consumidor em Java

Monitores (4)

```
#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 */
     }
}
```

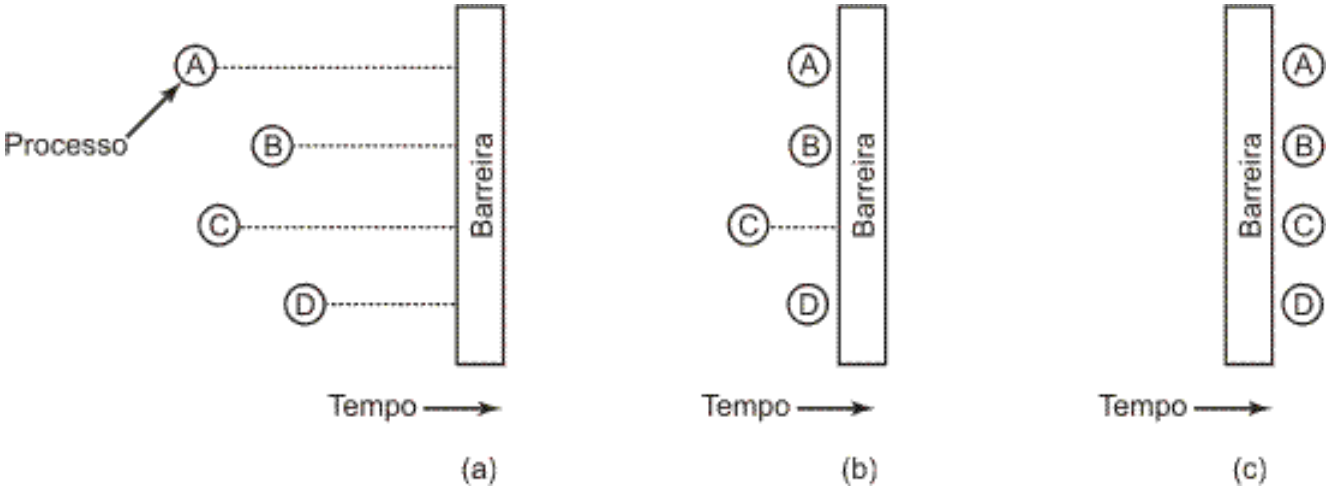
Solução para o problema do produtor-consumidor em Java (parte 2)

Troca de 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 */
     }
}
```

O problema do produtor-consumidor com N mensagens

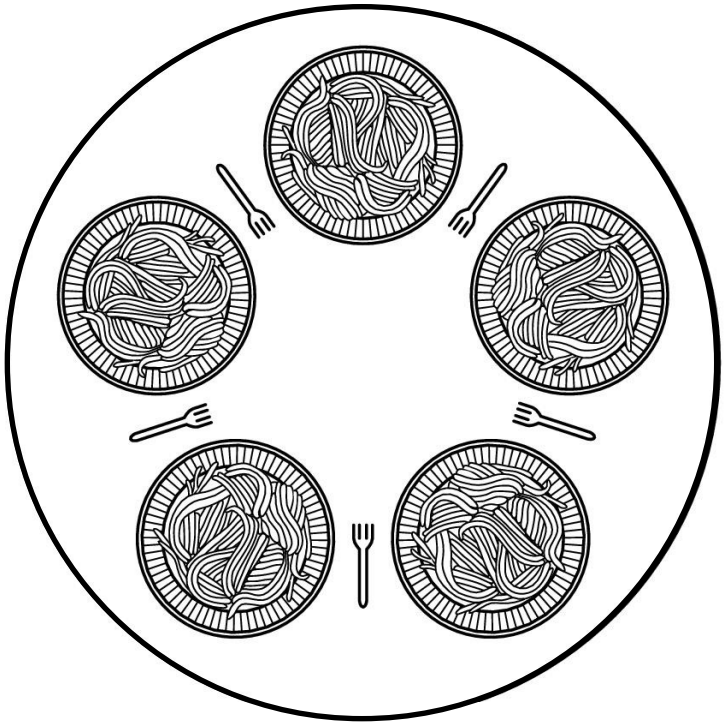
Barreiras



- Uso de uma barreira
  a) processos se aproximando de uma barreira
  b) todos os processos, exceto um, bloqueados pela barreira
  c) último processo chega, todos passam

Jantar dos Filósofos (1)

- Filósofos comem/pensam
- Cada um precisa de 2 garfos para comer
- Pega um garfo por vez
- Como prevenir deadlock



Jantar dos Filósofos (2)

```
#define N 5                                     /* número de filósofos */

void philosopher(int i)                         /* i: número do filósofo, de 0 a 4 */
{
      while (TRUE) {
            think( );                           /* o filósofo está pensando */
            take_fork(i);                       /* pega o garfo esquerdo */
            take_fork((i+1) % N);               /* pega o garfo direito; % é o operador modulo */
            eat( );                             /* hummm! Espaguete! */
            put_fork(i);                        /* devolve o garfo esquerdo à mesa */
            put_fork((i+1) % N);                /* devolve o garfo direito à mesa */
      }
}
```

Uma solução errada para o problema do jantar dos filósofos

Jantar dos Filósofos (3)

```
#define N           5                   /* número de filósofos */
#define LEFT        (i+N-1)%N           /* número do vizinho à esquerda de i */
#define RIGHT       (i+1)%N             /* número do vizinho à direita de i */
#define THINKING    0                   /* o filósofo está pensando */
#define HUNGRY      1                   /* o filósofo está tentando pegar garfos */
#define EATING      2                   /* o filósofo está comendo */
typedef int semaphore;                  /* semáforos são um tipo especial de int */
int state[N];                           /* arranjo para controlar o estado de cada um */
semaphore mutex = 1;                    /* exclusão mútua para as regiões críticas */
semaphore s[N];                         /* um semáforo por filósofo */

void philosopher(int i)                 /* i: o número do filósofo, de 0 a N-1 */
{
    while (TRUE) {                      /* repete para sempre */
        think( );                       /* o filósofo está pensando */
        take_forks(i);                  /* pega dois garfos ou bloqueia */
        eat( );                         /* hummm! Espaguete! */
        put_forks(i);                   /* devolve os dois garfos à mesa */
    }
}
```

Uma solução para o problema do jantar dos filósofos (parte 1)

Jantar dos Filósofos (4)

```
void take_forks(int i)                  /* i: o número do filósofo, de 0 a N–1 */
{
     down(&mutex);                      /* entra na região crítica */
     state[i] = HUNGRY;                 /* registra que o filósofo está faminto */
     test(i);                           /* tenta pegar dois garfos */
     up(&mutex);                        /* sai da região crítica */
     down(&s[i]);                       /* bloqueia se os garfos não foram pegos */
}

void put_forks(i)                       /* i: o número do filósofo, de 0 a N–1 */
{
     down(&mutex);                      /* entra na região crítica */
     state[i] = THINKING;               /* o filósofo acabou de comer */
     test(LEFT);                        /* vê se o vizinho da esquerda pode comer agora */
     test(RIGHT);                       /* vê se o vizinho da direita pode comer agora */
     up(&mutex);                        /* sai da região crítica */
}

void test(i)                            /* i: o número do filósofo, de 0 a N–1 */
{
     if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
          state[i] = EATING;
          up(&s[i]);
     }
}
```

Uma solução para o problema do jantar dos filósofos (parte 2)

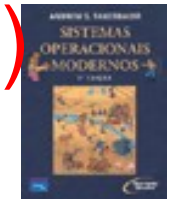
O Problema dos Leitores e Escritores

```
typedef int semaphore;                  /* use sua imaginação */
semaphore mutex = 1;                    /* controla o acesso a 'rc' */
semaphore db = 1;                       /* controla o acesso a base de dados */
int rc = 0;                             /* número de processos lendo ou querendo ler */

void reader(void)
{
     while (TRUE) {                     /* repete para sempre */
          down(&mutex);                 /* obtém acesso exclusivo a 'rc' */
          rc = rc + 1;                  /* um leitor a mais agora */
          if (rc == 1) down(&db);       /* se este for o primeiro leitor ... */
          up(&mutex);                   /* libera o acesso exclusivo a 'rc' */
          read_data_base( );            /* acesso aos dados */
          down(&mutex);                 /* obtém acesso exclusivo a 'rc' */
          rc = rc – 1;                  /* um leitor a menos agora */
          if (rc == 0) up(&db);         /* se este for o último leitor ... */
          up(&mutex);                   /* libera o acesso exclusivo a 'rc' */
          use_data_read( );             /* região não crítica */
     }
}


void writer(void)
{
     while (TRUE) {                     /* repete para sempre */
          think_up_data( );             /* região não crítica */
          down(&db);                    /* obtém acesso exclusivo */
          write_data_base( );           /* atualiza os dados */
          up(&db);                      /* libera o acesso exclusivo */
     }
}
```

Uma solução para o problema dos leitores e escritores

O Problema do Barbeiro Sonolento (1)

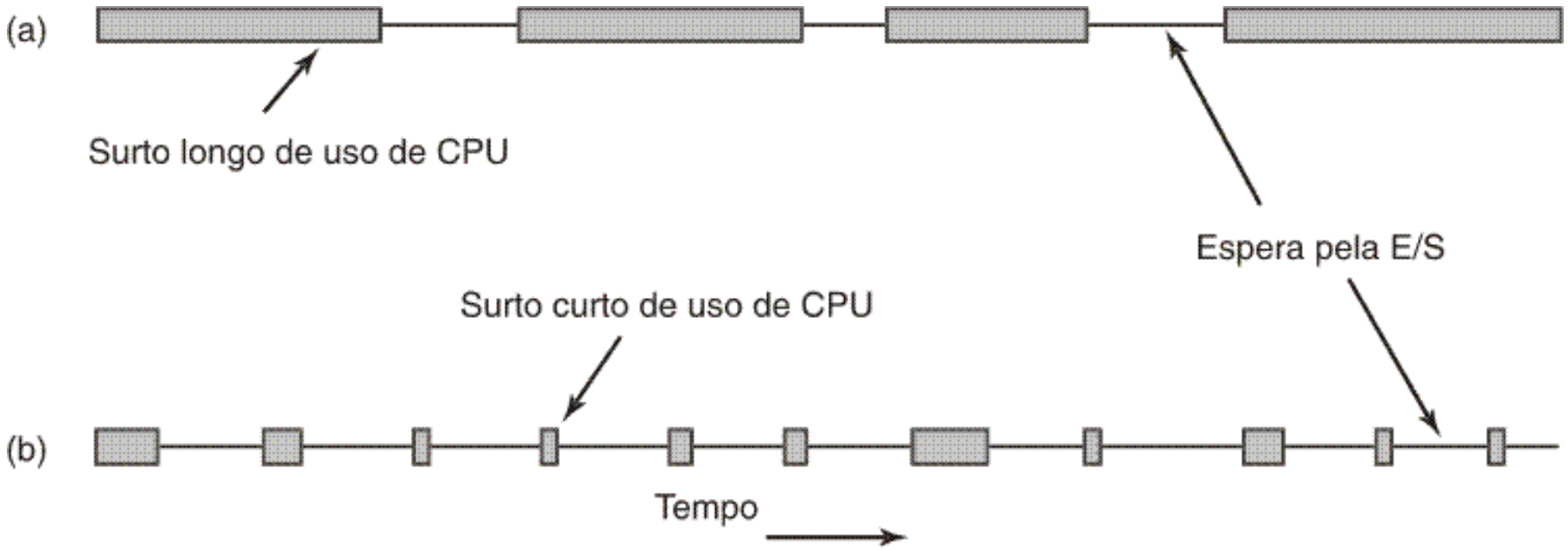
O Problema do Barbeiro Sonolento (2)



```
#define CHAIRS 5                    /* número de cadeiras para os clientes à espera */

typedef int semaphore;              /* use sua imaginação */

semaphore customers = 0;            /* número de clientes à espera de atendimento*/
semaphore barbers = 0;              /* número de barbeiros à espera de clientes */
semaphore mutex = 1;                /* para exclusão mútua */
int waiting = 0;                    /* clientes estão esperando (não estão cortando) */

void barber(void)
{
    while (TRUE) {
        down(&customers);           /* vai dormir se o número de clientes for 0 */
        down(&mutex);               /* obtém acesso a 'waiting' */
        waiting = waiting - 1;      /* decresce de um o contador de clientes à espera */
        up(&barbers);               /* um barbeiro está agora pronto para cortar cabelo */
        up(&mutex);                 /* libera 'waiting' */
        cut_hair();                 /* corta o cabelo (fora da região crítica) */
    }
}

void customer(void)
{
    down(&mutex);                   /* entra na região crítica */
    if (waiting < CHAIRS) {         /* se não houver cadeiras livres, saia */
        waiting = waiting + 1;      /* incrementa o contador de clientes à espera */
        up(&customers);             /* acorda o barbeiro se necessário */
        up(&mutex);                 /* libera o acesso a 'waiting' */
        down(&barbers);             /* vai dormir se o número de barbeiros livres for 0 */
        get_haircut();              /* sentado e sendo servido */
    } else {
        up(&mutex);                 /* a barbearia está cheia; não espere */
    }
}
```

Solução para o problema do barbeiro sonolento

Escalonamento
Introdução ao Escalonamento (1)



- Surtos de uso da CPU alternam-se com períodos de espera por E/S
  a) um processo orientado à CPU
  b) um processo orientado à E/S

Introdução ao Escalonamento (2)

Todos os sistemas

Justiça — dar a cada processo uma porção justa da UCP
Aplicação da política — verificar se a política estabelecida é cumprida
Equilíbrio — manter ocupadas todas as partes do sistema

Sistemas em lote

Vazão (throughput) — maximizar o número de jobs por hora
Tempo de retorno — minimizar o tempo entre a submissão e o término
Utilização de UCP — manter a UCP ocupada o tempo todo

Sistemas interativos

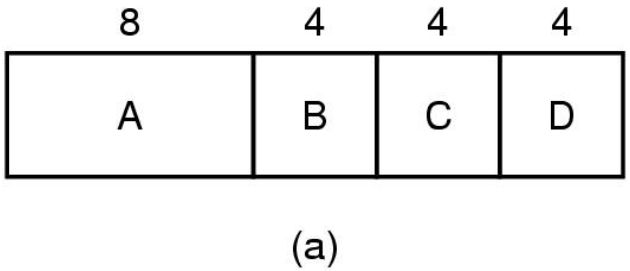
Tempo de resposta — responder rapidamente às requisições
Proporcionalidade — satisfazer as expectativas dos usuários

Sistemas de tempo real

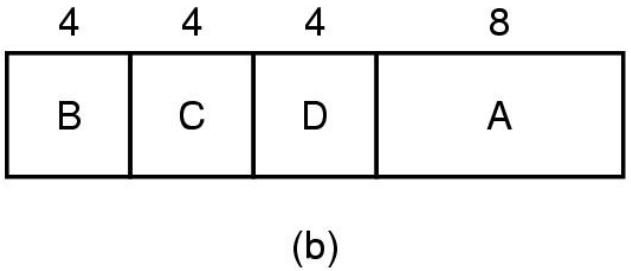
Cumprimento dos prazos — evitar a perda de dados
Previsibilidade — evitar a degradação da qualidade em sistemas multimídia

Objetivos do algoritmo de escalonamento

Escalonamento em Sistemas em Lote (1)



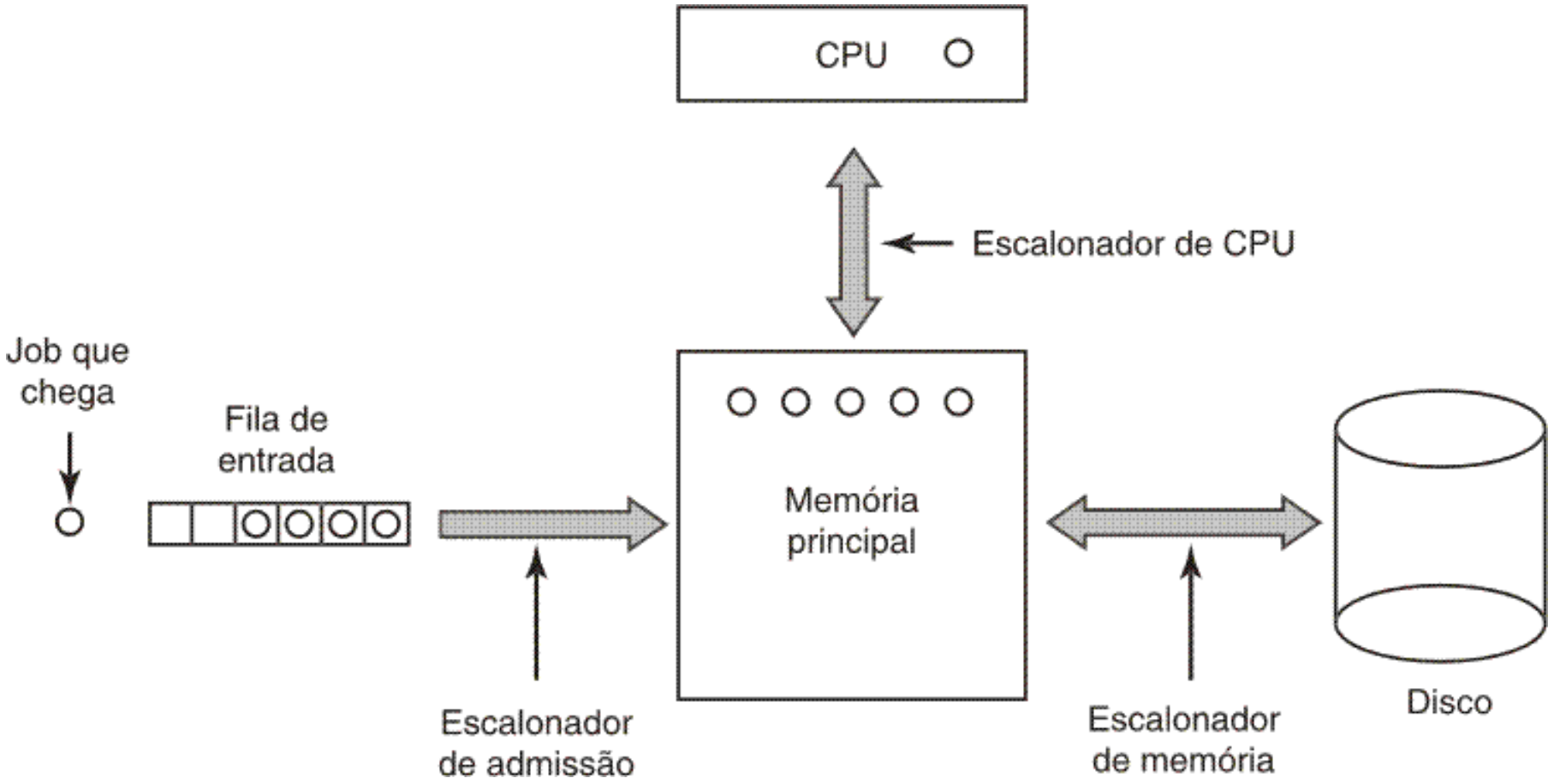
(a)



(b)

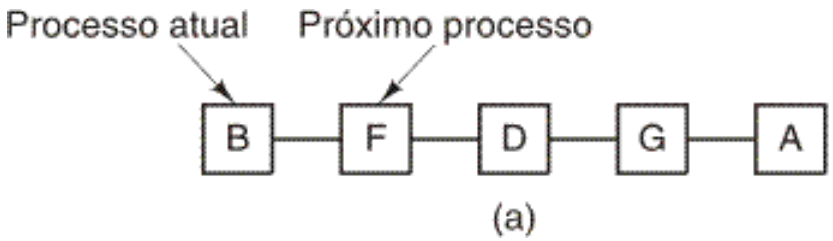
Um exemplo de escalonamento *job mais curto primeiro*

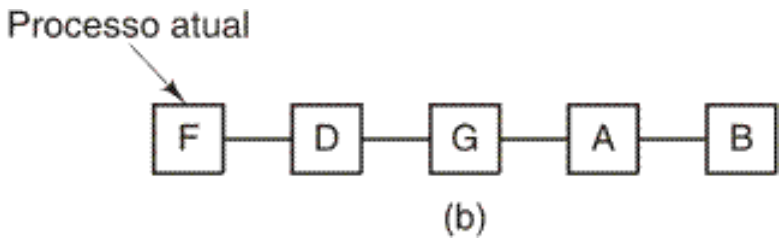
Escalonamento em Sistemas em Lote (2)



Escalonamento em três níveis

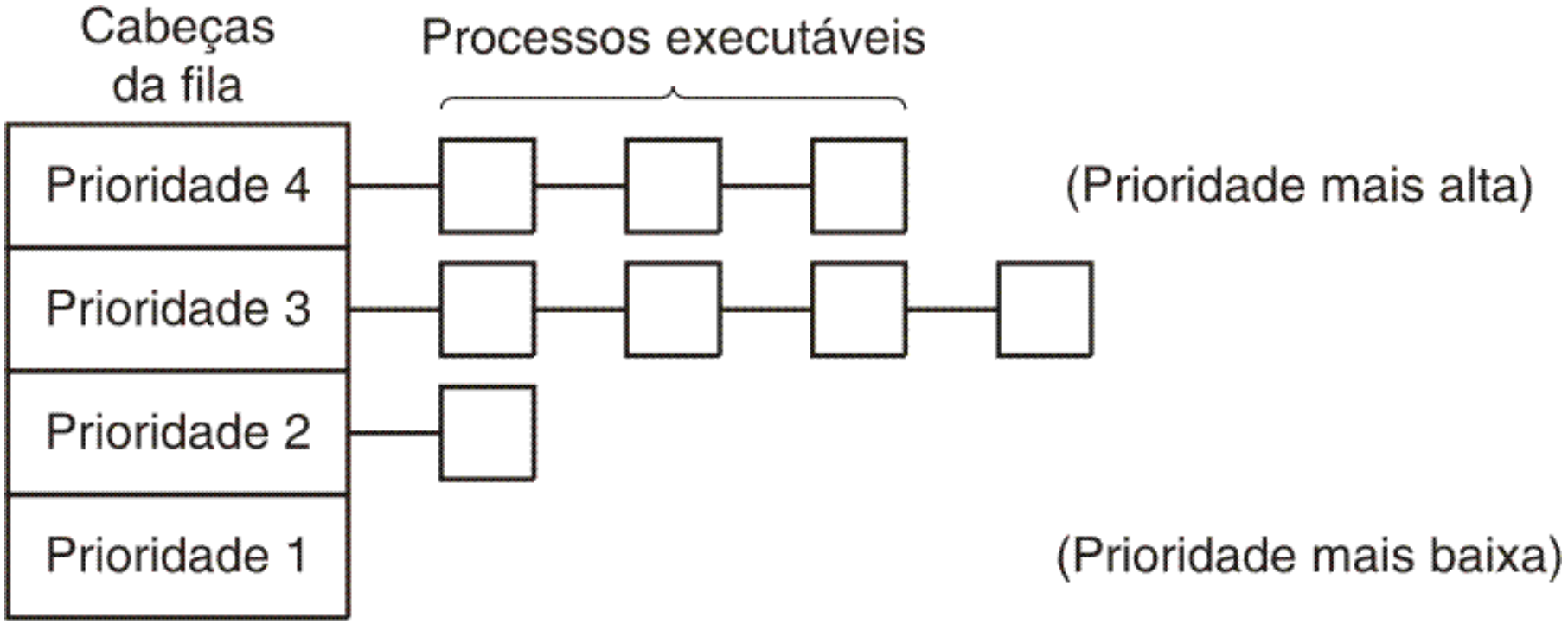
Escalonamento em Sistemas Interativos (1)





- Escalonamento por alternância circular *(round-robin)*
  a) lista de processos executáveis
  b) lista de processos executáveis depois que B usou todo o seu quantum

Escalonamento em Sistemas Interativos (2)



Um algoritmo de escalonamento com quatro classes de prioridade

Escalonamento em Sistemas de Tempo-Real

Sistema de tempo-real escalonável

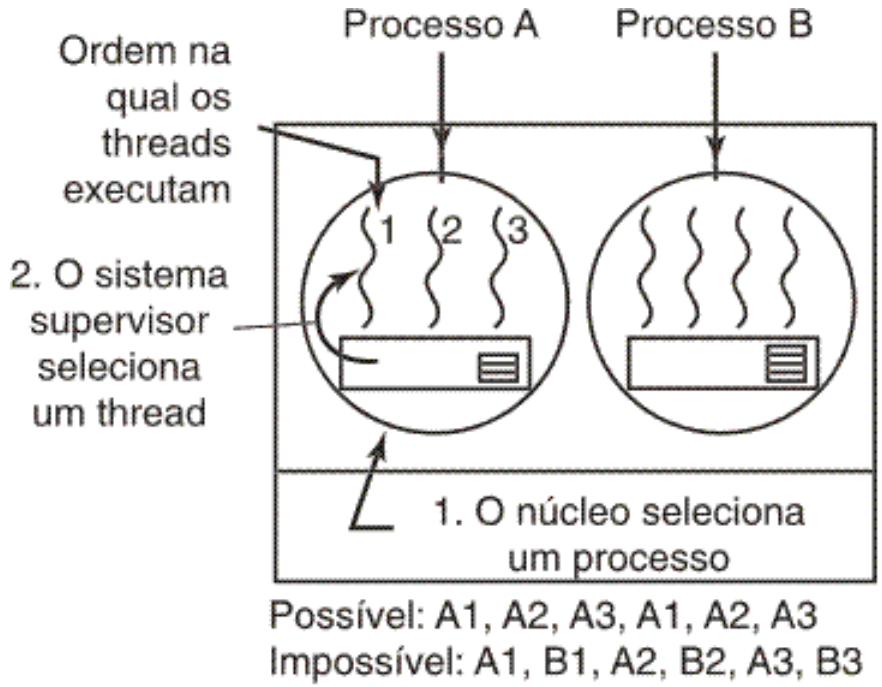
- Dados
  - *m* eventos periódicos
  - evento *i* ocorre dentro do período P_i e requer C_i segundos
- Então a carga poderá ser tratada somente se

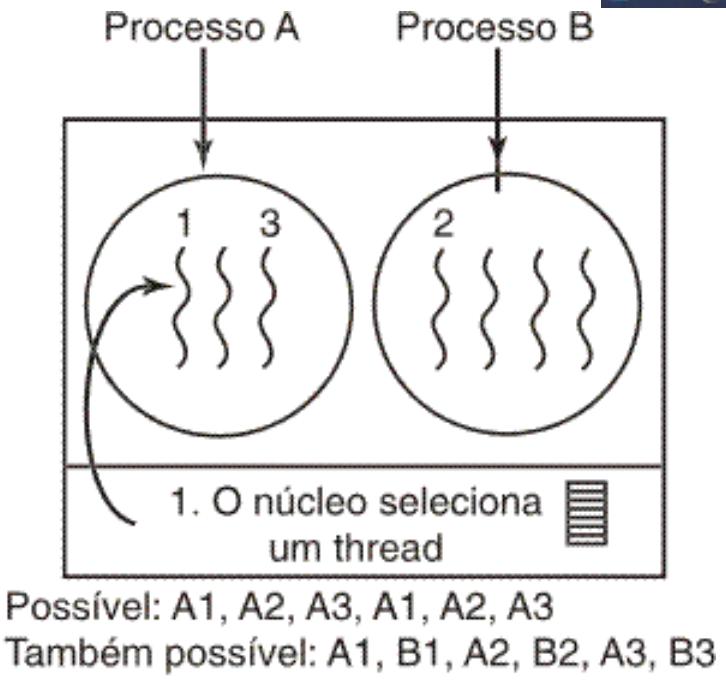
$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Política *versus* Mecanismo

- Separa o que é permitido ser feito do como é feito
  - um processo sabe quais de seus threads filhos são importantes e precisam de prioridade
- Algoritmo de escalonamento parametrizado
  - mecanismo no núcleo
- Parâmetros preenchidos pelos processos do usuário
  - política estabelecida pelo processo do usuário

Escalonamento de Threads (1)

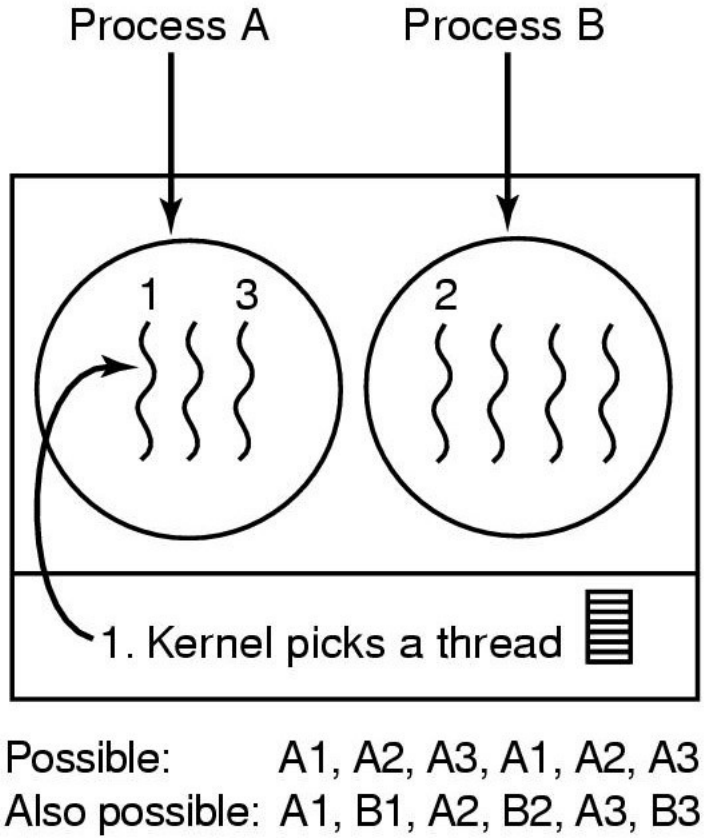




Possível escalonamento de threads de usuário

- processo com quantum de 50-mseg
- threads executam 5 mseg por surto de CPU

Escalonamento de Threads (2)



Possível escalonamento de threads de núcleo

- processo com quantum de 50-mseg
- threads executam 5 mseg por surto de CPU