Sistemas Operacionais

Rafael Sachetto Oliveira
sachetto@ufsj.edu.br

6 de abril de 2011

O modelo de processo

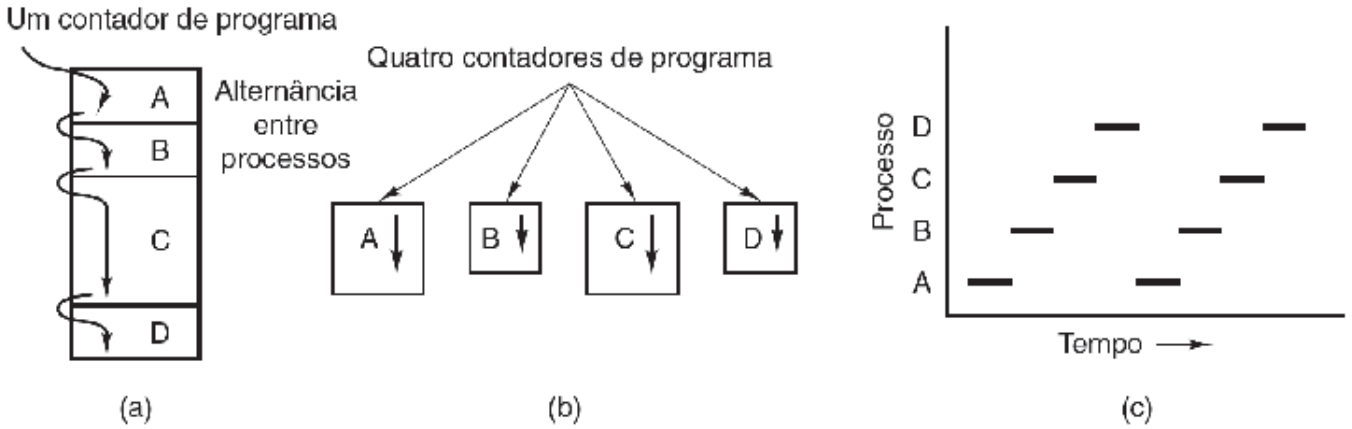


Figura 2.1 (a) Multiprogramação de quatro programas. (b) Modelo conceitual de quatro processos sequenciais independentes. (c) Somente um programa está ativo a cada momento.

Criação de processos

Eventos que causam a criação de processos:

- Inicialização de sistema.
- Execução de uma chamada de sistema de criação de processo por um processo em execução.
- Requisição do usuário para criar um novo processo.
- Inicialização de uma tarefa em lotes.

Término de processos

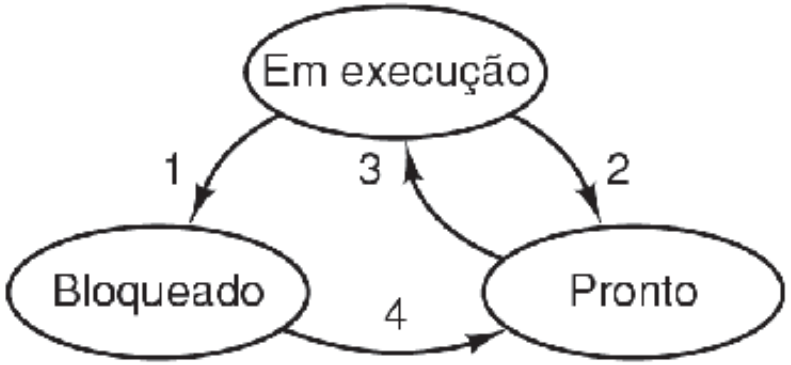
Eventos que causam o término de um processo:

- Saída normal (voluntária).
- Saída por erro (voluntária).
- Erro fatal (involuntário).
- Cancelamento por outro processo (involuntário)

Hierarquia de processos

- No Unix, processos pais criam processos filhos
- No Windows não existe tal hierarquia.

Estados de processos



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

Figura 2.2 Um processo pode estar nos estados em execução, bloqueado ou pronto. As transições entre esses estados são mostradas.

Implementação de processos

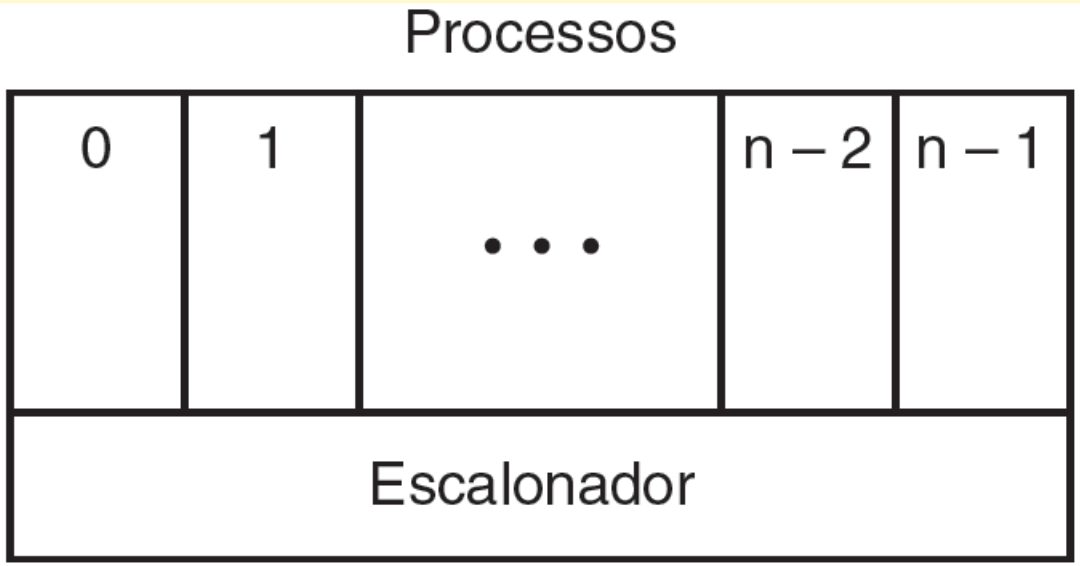


Figura 2.3 O nível mais baixo de um sistema operacional estruturado em processos controla interrupções e escalonamento. Acima desse nível estão processos sequenciais.

Tabela de Processos

Gerenciamento de processo	Gerenciamento de memória	Gerenciamento de arquivo
Registros	Ponteiro para informações sobre o segmento de texto	Diretório-raiz
Contador de programa		Diretório de trabalho
Palavra de estado do programa	Ponteiro para informações sobre o segmento de texto	Descritores de arquivo
Ponteiro da pilha		ID do usuário
Estado do processo	Ponteiro para informações sobre o segmento de texto	ID do grupo
Prioridade		
Parâmetros de escalonamento		
ID do processo		
Processo pai		
Grupo do processo		
Sinais		
Momento em que um processo foi iniciado		
Tempo de CPU usado		
Tempo de CPU do processo filho		
Tempo do alarme seguinte		

Tabela 2.1 Alguns dos campos de um processo típico de entrada na tabela.

1. O hardware empilha o contador de programa etc.
2. O hardware carrega o novo contador de programa a partir do arranjo de interrupções.
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.

Tabela 2.2 O esqueleto do que o nível mais baixo do sistema operacional faz quando ocorre uma interrupção.

Modelando a multiprogramação

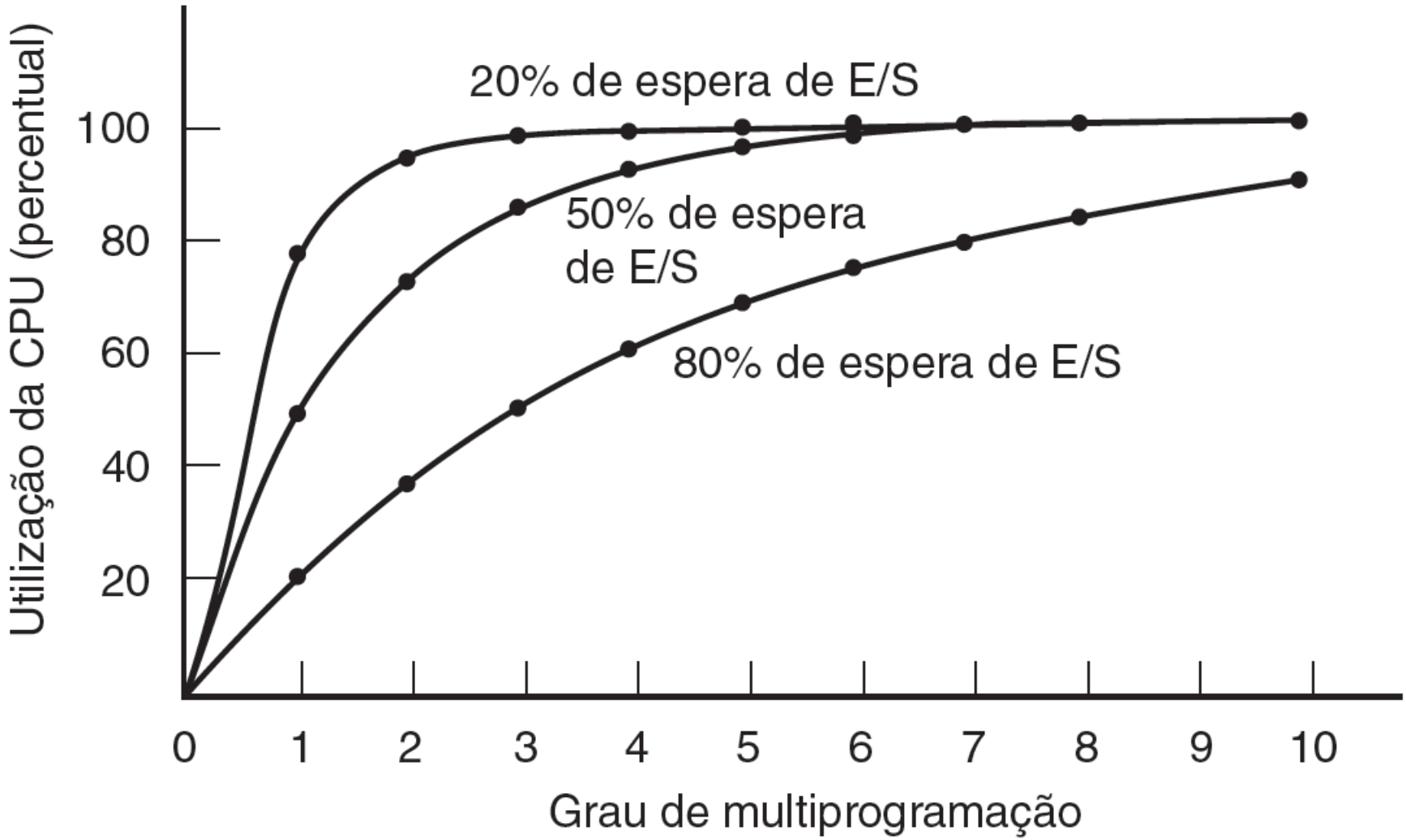


Figura 2.4 Utilização da CPU como função do número de processos na memória.

Uso de Thread

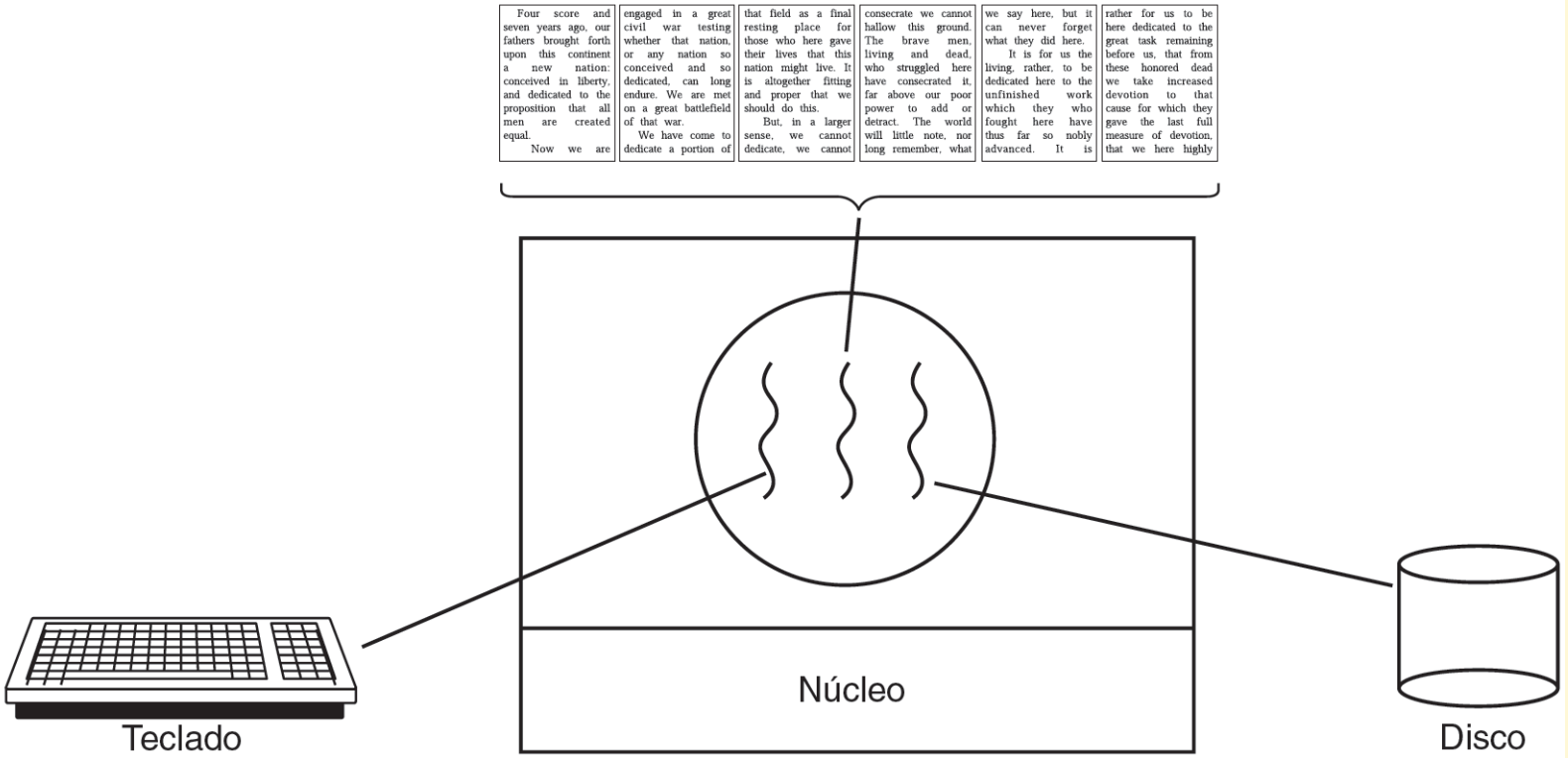


Figura 2.5 Um processador de textos com três threads.

Servidor Web

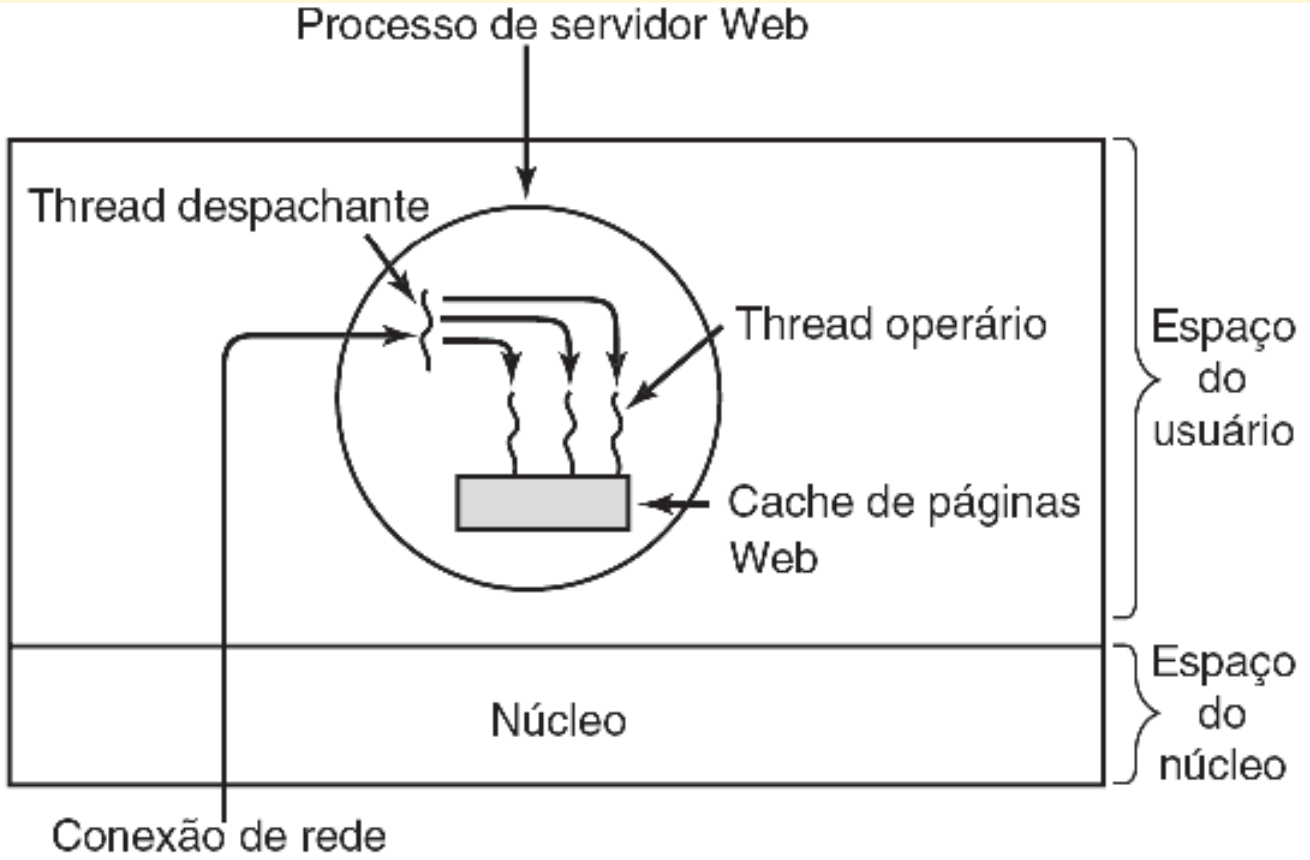


Figura 2.6 Um servidor Web multithread.

Código

```
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)

Figura 2.7 Uma simplificação do código para a Figura 2.6. (a) Thread despachante. (b) Thread operário.

Modos de se implementar

Modelo	Características
Threads	Paralelismo, chamadas de sistema bloqueante
Processo monothread	Não paralelismo, chamadas de sistema bloqueantes
Máquina de estados finitos	Paralelismo, chamadas não-bloqueantes, interrupções

Tabela 2.3 Três modos de construir um servidor.

O modelo de thread clássico

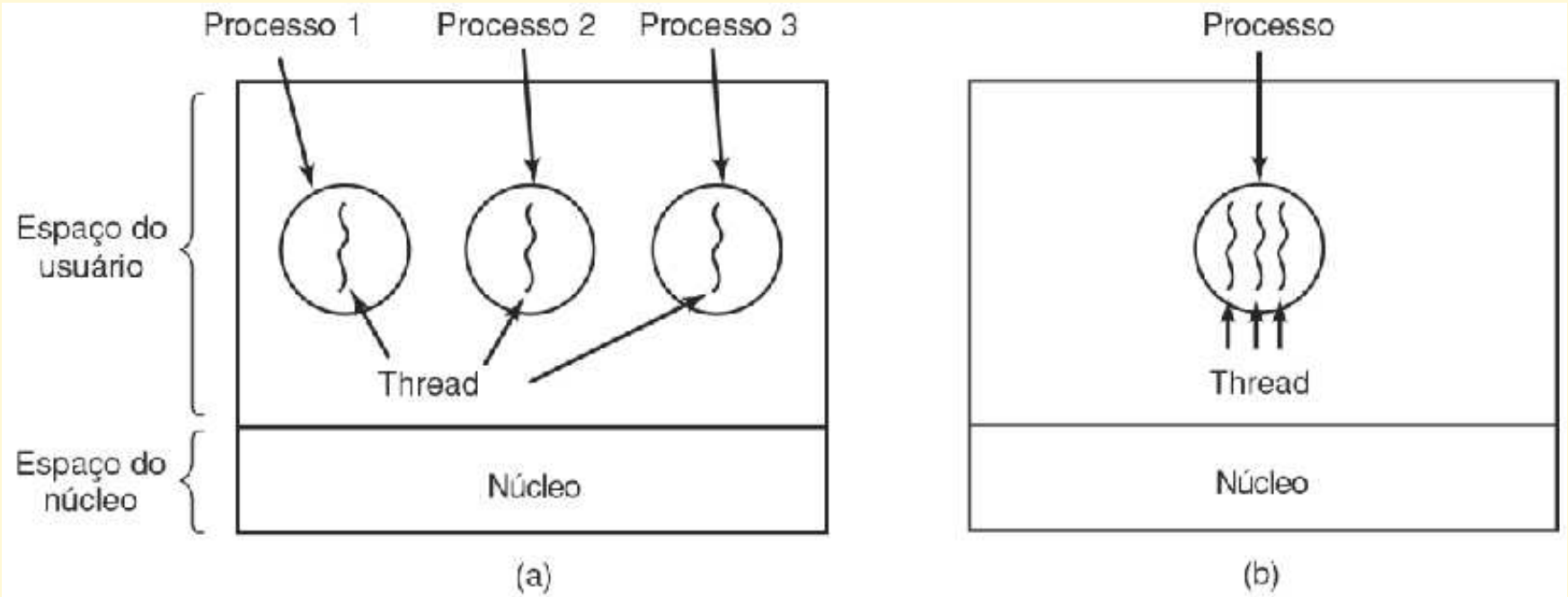


Figura 2.8 (a) Três processos, cada um com um thread. (b) Um processo com três threads.

O modelo de thread clássico

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 manipuladores de sinais	
Informação de contabilidade	

Tabela 2.4 A primeira coluna lista alguns itens compartilhados por todos os threads em um processo. A segunda lista alguns itens específicos a cada thread.

O modelo de thread clássico

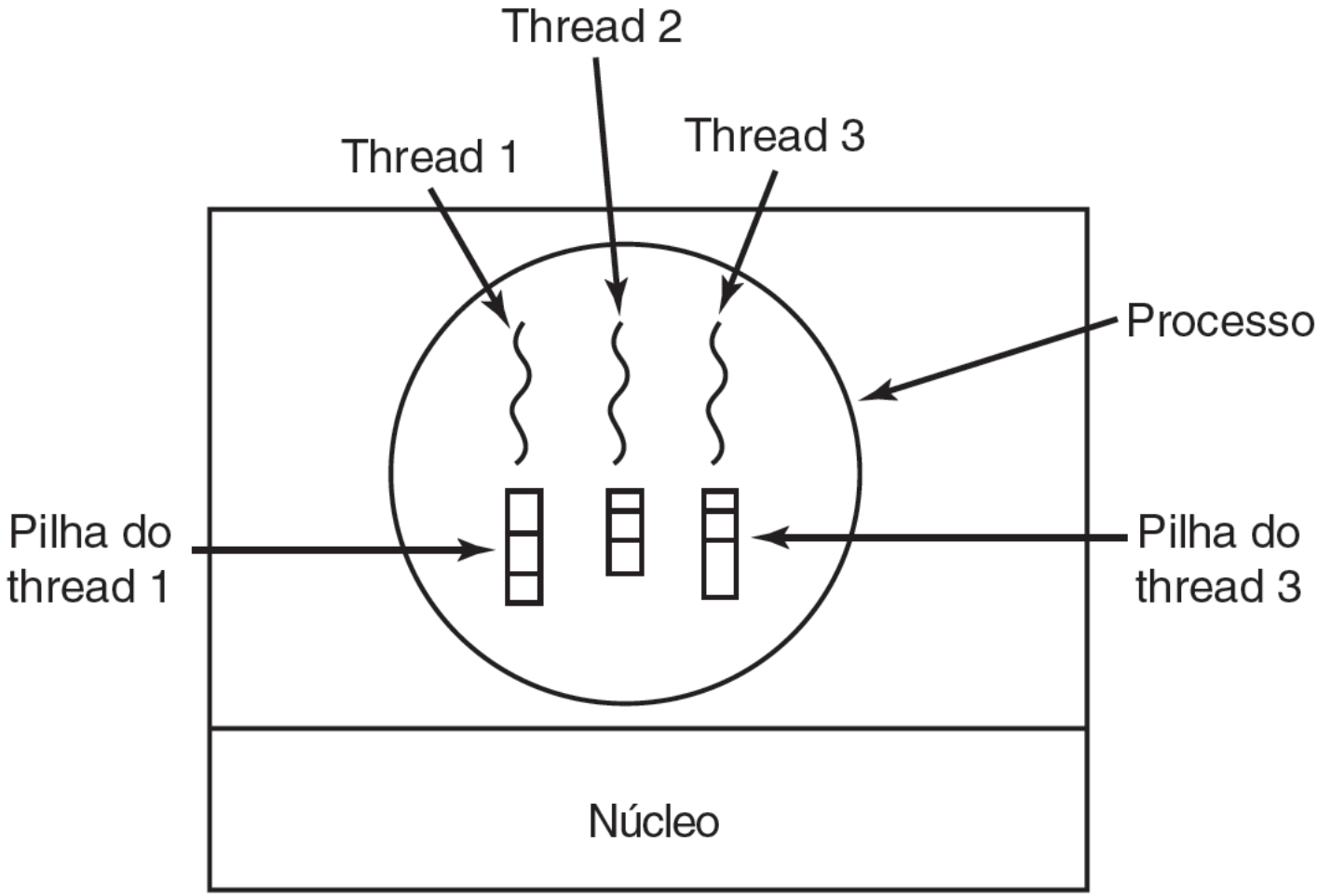


Figura 2.9 Cada thread tem sua própria pilha.

Threads POSIX

Chamada de thread	Descrição
pthread_create	Cria um novo thread
pthread_exit	Conclui a chamada de thread
pthread_join	Espera que um thread específico seja abandonado
pthread_yield	Libera a CPU para que outro thread seja executado
pthread_attr_init	Cria e inicializa uma estrutura de atributos do thread
pthread_attr_destroy	Remove uma estrutura de atributos do thread

Tabela 2.5 Algumas das chamadas de função de Pthreads.

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

#define NUMBER_OF_THREADS     10

void *print_hello_world(void *tid)
{
        /* Esta função imprime o identificador do thread e sai. */
        printf("Hello World. Greetings from thread %d\n", tid);
        pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
        /* O programa principal cria 10 threads e sai. */
        pthread_t threads[NUMBER_OF_THREADS];
        int status, i;

        for(i=0; i < NUMBER_OF_THREADS; i++) {
                printf("Main here. Creating thread %d\n", i);
                status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

                if (status != 0) {
                        printf("Oops. pthread_create returned error code %d\n", status);
                        exit(-1);
                }
        }
        exit(NULL);
}
```

Implementando threads no espaço do usuário

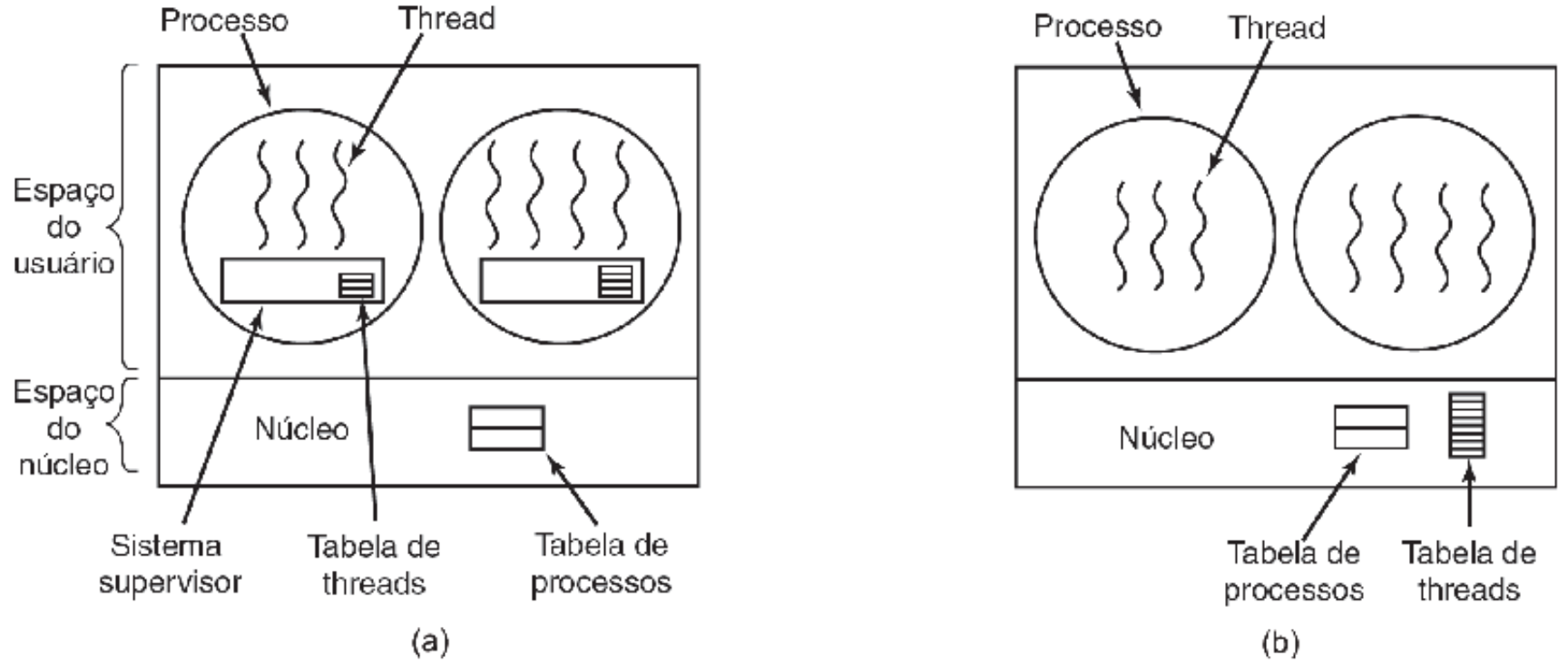


Figura 2.11 (a) Um pacote de threads de usuário. (b) Um pacote de threads administrado pelo núcleo.

Implementações híbridas

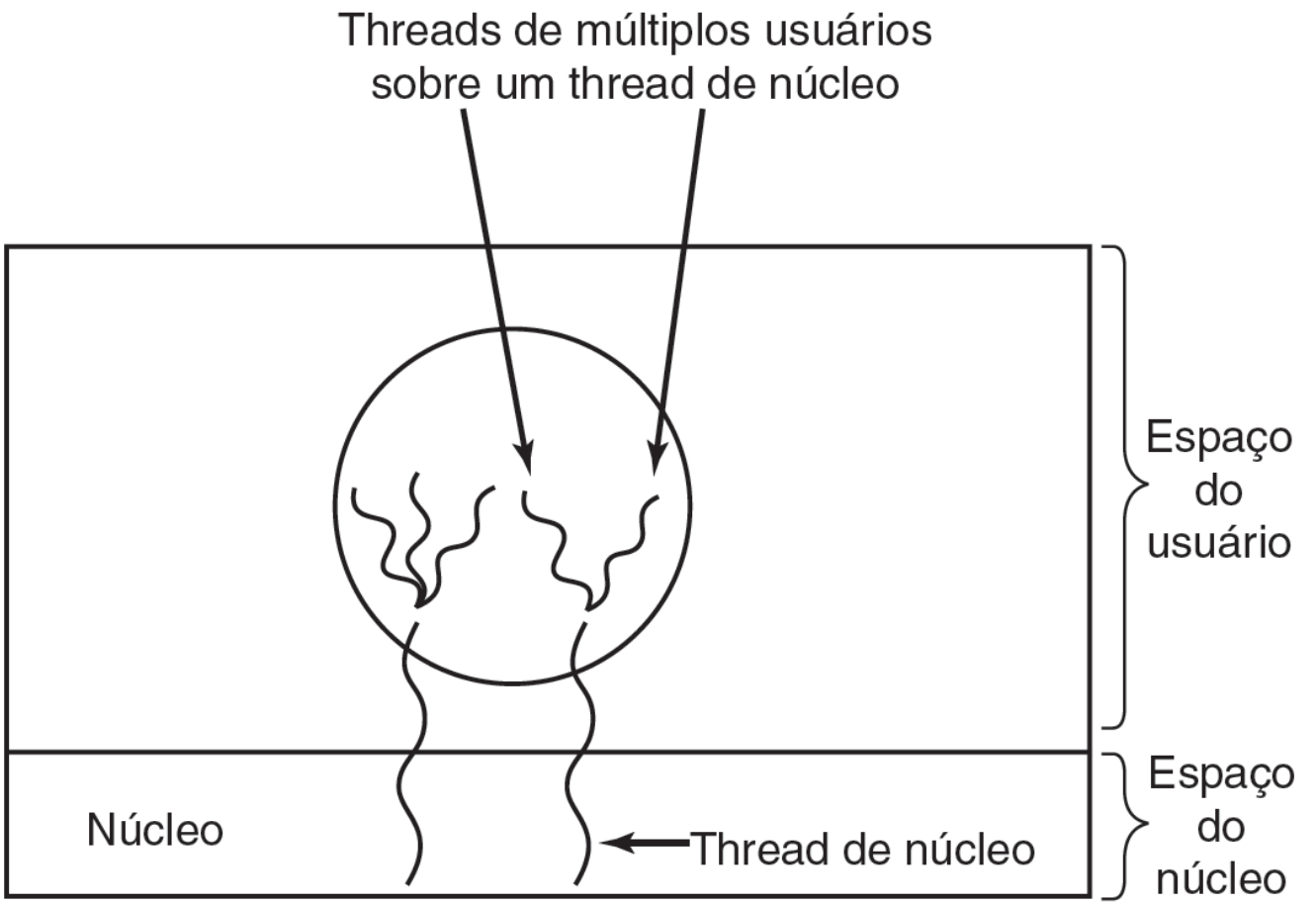


Figura 2.12 Multiplexando threads de usuários sobre threads de núcleo.

Threads pop-up

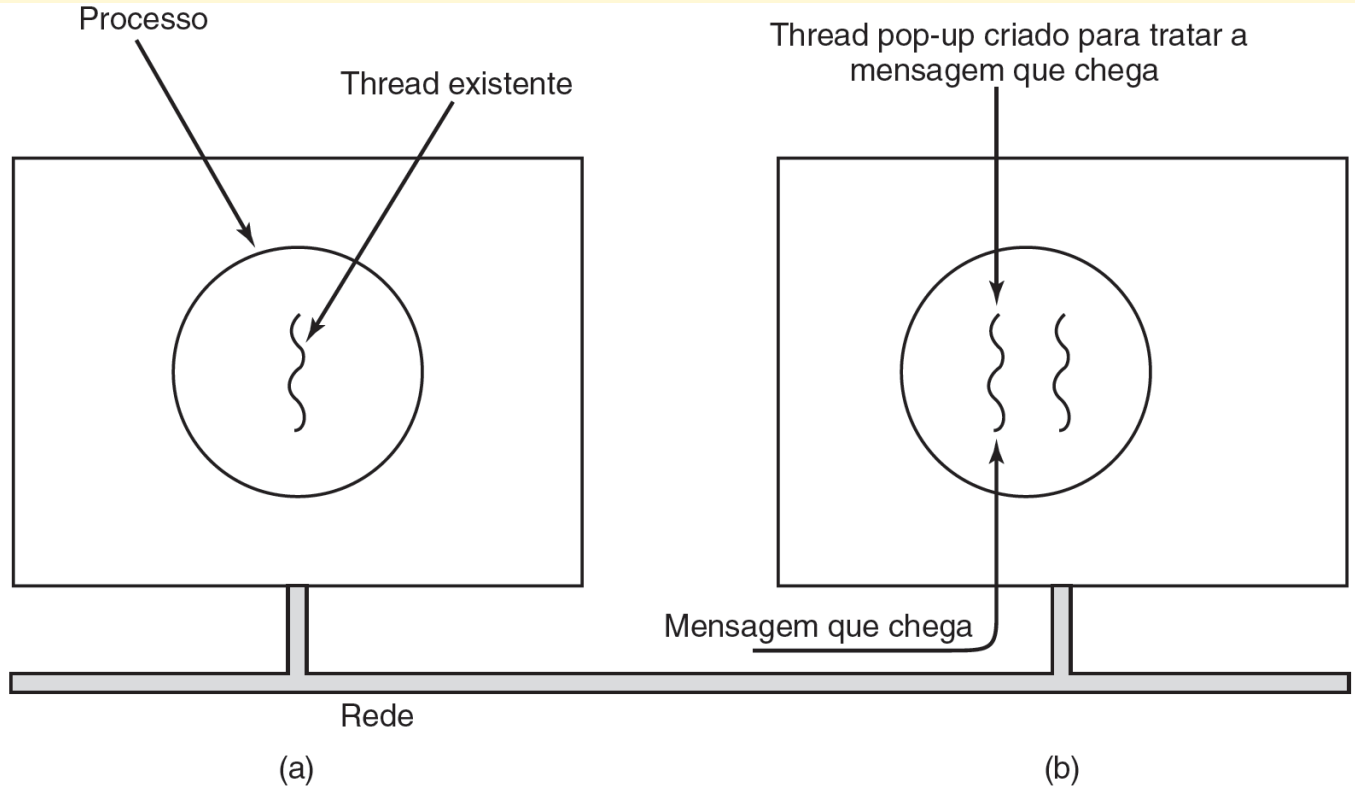


Figura 2.13 Criação de um novo thread quando chega uma mensagem. (a) Antes da chegada da mensagem. (b) Após a chegada da mensagem.

Convertendo o código monothread em código multithread

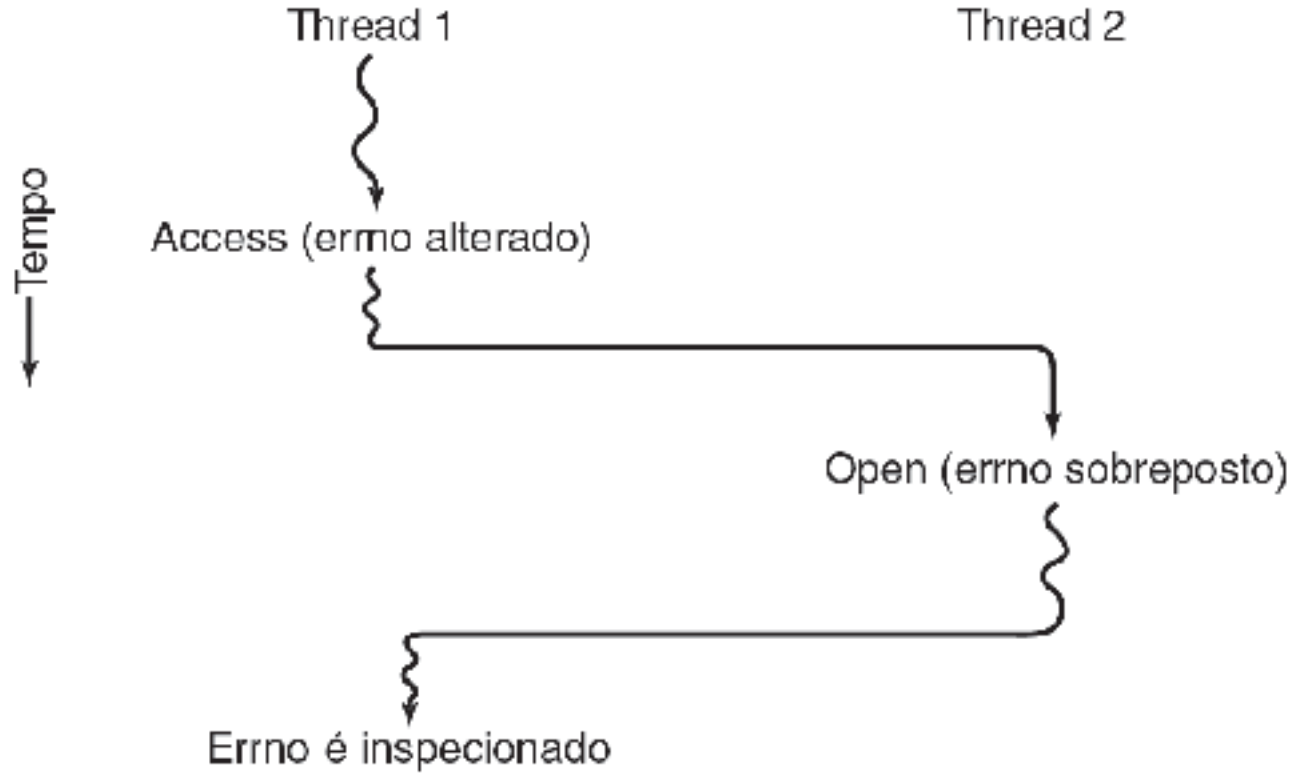


Figura 2.14 Conflitos entre threads sobre o uso de uma variável global.

Variáveis Globais

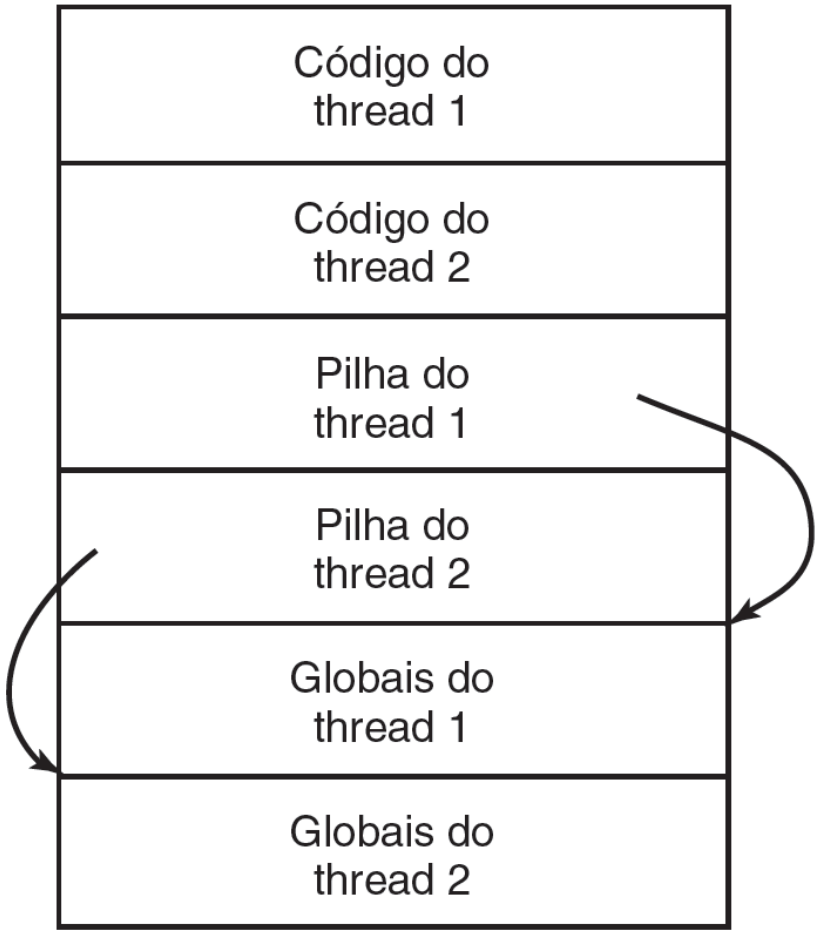


Figura 2.15 Threads podem ter variáveis globais individuais.

Condições de Corrida

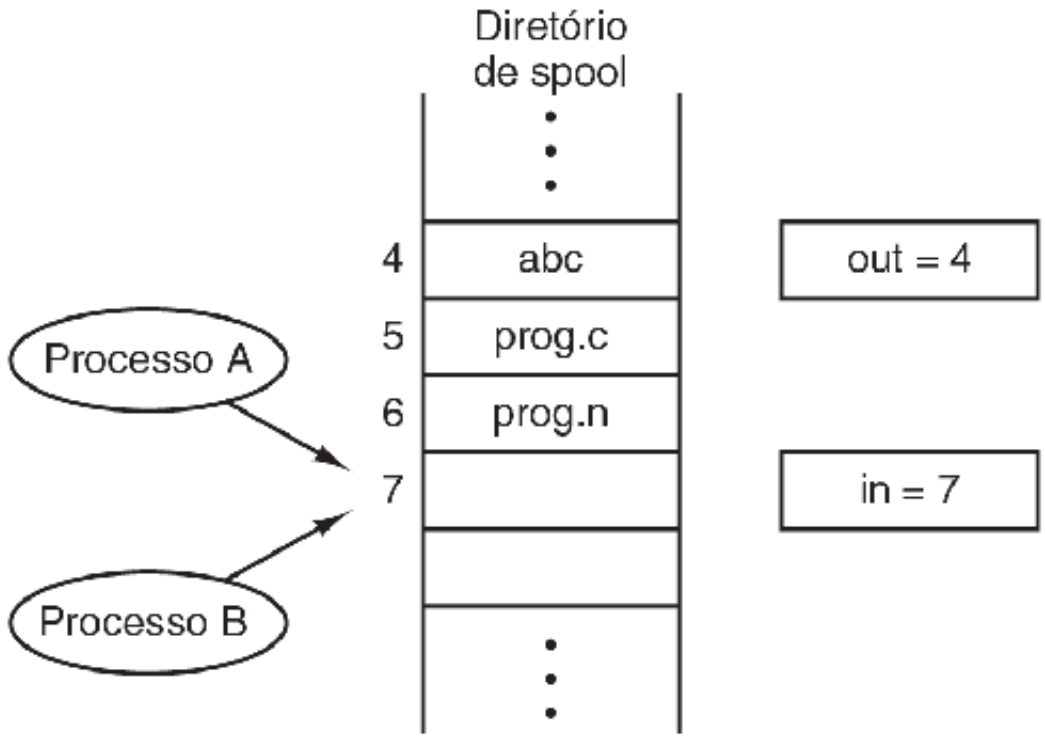


Figura 2.16 Dois processos querem acessar a memória compartilhada ao mesmo tempo.

Regiões críticas

- Dois processos não podem estar simultaneamente dentro de suas regiões críticas.
- Nada pode ser afirmado sobre a velocidade ou sobre o número de CPUs.
- Nenhum processo sendo executado fora de sua região crítica pode bloquear outros processos.
- Nenhum processo deve esperar eternamente para entrar em sua região crítica.

Exclusão Mútua

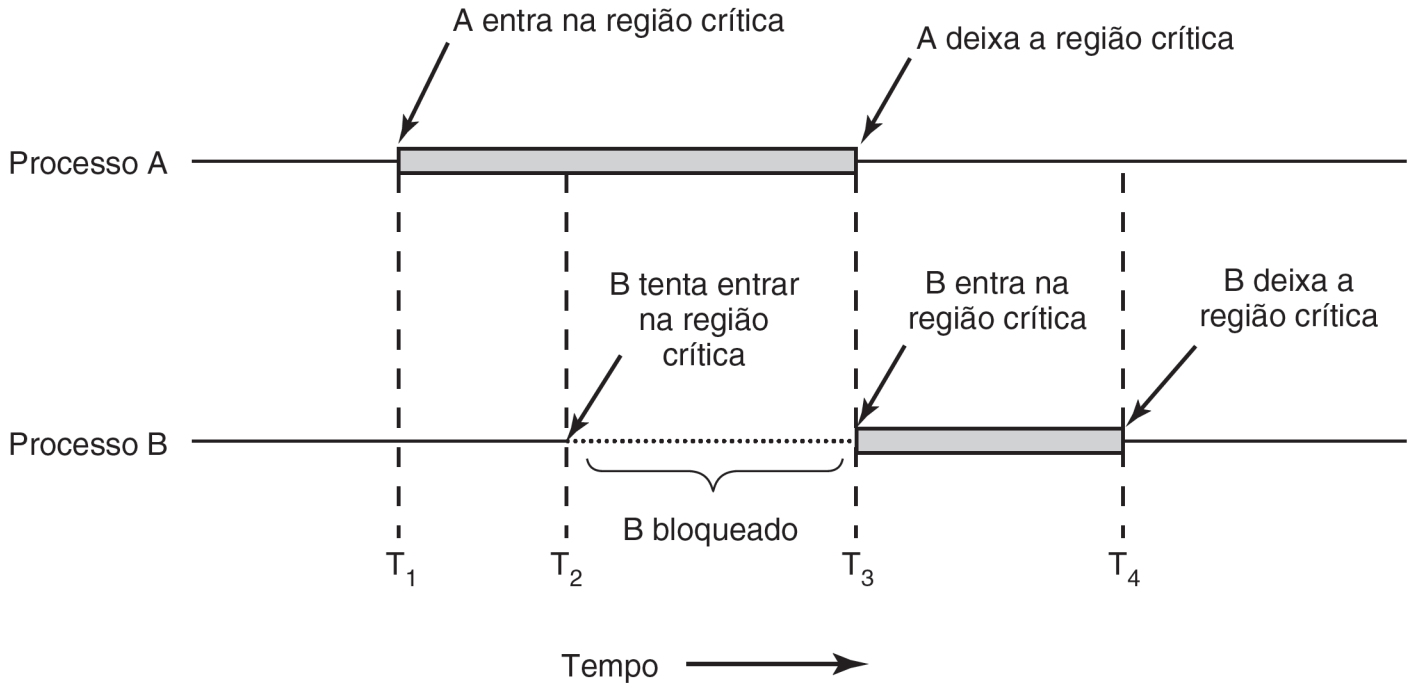


Figura 2.17 Exclusão mútua usando regiões críticas.

Exclusão mútua com espera ociosa

Propostas para obtenção de exclusão mútua:

- Desabilitando interrupções.
- Variáveis do tipo trava.
- Chaveamento obrigatório.
- Solução de Peterson.
- A instrução TSL.

Exclusão mútua com espera ociosa

Desabilitar interrupções:

- Problemas: processo do usuário ganha muito poder, poder acabar com o sistema.
- Não adianta desabilitar interrupções em Sistemas com mais de uma CPU;
- É uma técnica adequada para processos do S.O., mas não de usuários comum.

Exclusão mútua com espera ociosa

Variáveis de do tipo trava (Lock):

- Geram uma condições de corrida!

Exclusão mútua com espera ociosa

```
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)

Figura 2.18 Solução proposta para o problema da região crítica. (a) Processo 0. (b) Processo 1. Em ambos os casos, não deixe de observar os ponto-e-vírgulas concluindo os comandos while.

Chaveamento Obrigatório

- Utiliza espera ociosa, pois a CPU fica ocupada (laço while);
- Conhecida por spin lock;
- Chaveamento obrigatório não funciona quando um processo é muito mais lento que o outro;
- Viola a condição 3: um processo fica bloqueado por outro processo que não está em região crítica;

Exclusão mútua com espera ociosa

Solução de Peterson

```
#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 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 */
}
```

Figura 2.19 A solução de Peterson para implementar a exclusão mútua.

Exclusão mútua com espera ociosa

A instrução 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 re
        RET                     | retorna a quem chamou; entrou na região crítica


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

Figura 2.20 Entrando e saindo de uma região crítica usando a instrução TSL.

Exclusão mútua com espera ociosa

TSL - test and set lock

- A cpu impede o acesso ao barramento. Resolve o problema de multicore
- Espera ociosa
- Alternativa é a instrução XCHG
- Podem apresentar o problema de inversão de prioridade

O Problema do produtor consumidor

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

void producer(void)
{
     int item;

     while (TRUE) {                             /* repita para sempre */
          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 cheio, 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 */
     }
}
```

Figura 2.22 O problema produtor–consumidor com uma condição de disputa fatal.

Dormir e Acordar

- Um processo pode dormir e ser acordado por outro
- Pode se perder o envio de sleep
- Podemos usar o bit de espera pelo sinal acordar

Semáforos

- Idealizados por Dijkstra em 1965
- Utilizam down e up (wait e signal)
- up, down e implementados de maneira indivisível
- São implementados desabilitando interrupções ou usando TSL e travas no caso de multicores.
- Semáforos podem ser utilizados para o gerenciamento de E/S;

Produto 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 item do buffer */
        up(&mutex);                             /* sai da região crítica */
        up(&empty);                             /* incrementa o contador de lugares vazios */
        consume_item(item);                     /* faz algo com o item */
    }
}
```

Figura 2.23 O problema produtor–consumidor usando semáforos.

Mutexes

- Servem para implementar exclusão mútua;
- mutex_lock e mutex_unlock
- fáceis de implementar em pacotes de usuário;
- podem haver chamadas do tipo try_lock

Mutexes

```
mutex_lock:
    TSL REGISTER,MUTEX      | copia mutex para o registrador e atribui  a ele o valor 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           | coloca 0 em mutex
    RET                     | retorna a quem chamou
```

Figura 2.24 Implementação do *mutex_lock* e do *mutex_unlock*.

Mutexes com pthreads

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

Tabela 2.6 Algumas chamadas de Pthreads relacionadas a mutexes.

Variáveis de condição com pthreads

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

Tabela 2.7 Algumas chamadas de Pthreads relacionadas a variáveis de condição.

Produto Consumidor com variáveis de condição

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000                          /* quantos números produzir */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;                    /* usado para sinalização */
int buffer = 0;                                 /* buffer usado entre produtor e consumidor */

void *producer(void *ptr)                       /* dados do produtor */
{       int i;

        for (i= 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);   /* obtém acesso exclusivo ao buffer */
                while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
                buffer = i;                       /* põe item no buffer */
                pthread_cond_signal(&condc);      /* acorda consumidor */
                pthread_mutex_unlock(&the_mutex);/* libera acesso ao buffer */
        }
        pthread_exit(0);
}

void *consumer(void *ptr)                       /* dados do consumidor */
{       int i;

        for (i = 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);   /* obtém acesso exclusivo ao buffer */
                while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
                buffer = 0;                       /* retire o item do buffer */
                pthread_cond_signal(&condp);      /* acorda o produtor */
                pthread_mutex_unlock(&the_mutex);/* libera acesso ao buffer */
        }
        pthread_exit(0);
}
```

Produto Consumidor com variáveis de condição

```
int main(int argc, char **argv)
{
        pthread_t pro, con;
        pthread_mutex_init(&the_mutex, 0);
        pthread_cond_init(&condc, 0);
        pthread_cond_init(&condp, 0);
        pthread_create(&con, 0, consumer, 0);
        pthread_create(&pro, 0, producer, 0);
        pthread_join(pro, 0);
        pthread_join(con, 0);
        pthread_cond_destroy(&condc);
        pthread_cond_destroy(&condp);
        pthread_mutex_destroy(&the_mutex);
}
```

Monitores

- Uso de semáforos pode levar a erros complexos;
- Unidade básica de sincronização de alto nível;
- São conceitos de linguagem e C não possui;
- Somente um processo pode estar ativo em um monitor em um dado momento
- Cabe ao compilador implementar a exclusão mútua nas entradas do monitor;
- Converter as regiões críticas em rotinas do monitor garante exclusão mútua;

Monitores

- possuem wait e signal para sincronização: bloqueam e permitem que outro processo entre no monitor;
- Para evitar conflito, signal só pode ser emitida como último comando da rotina de um monitor;
- wait deve vir antes do signal, pois variáveis condicionais não guardam estado;
- não existe o problema de sleep, wakeup por causa da exclusão mútua;
- Semáforos e monitores são para memória compartilhada;

Monitores

```
monitor example
    integer i;
    condition c;

    procedure producer ( );

    .
    .
    .
    end;

    procedure consumer ( );

    .       .       .

    end;
end monitor;
```

Figura 2.26 Um monitor.

Produto Consumidor com monitores - Pseudo Linguagem

```
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;
```

Produto Consumidor com monitores - Pseudo Linguagem

```
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;
```

Produto Consumidor com monitores - Java

```
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
        }
```

Produto Consumidor com monitores - Java

```
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
    }
```

Produto Consumidor com monitores - Java

```
        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) {};}
    }
}
```

Troca de Mensagens

- Usam send e receive;
- o receive bloqueia o receptor até que uma mensagem chegue, pode retornar imediatamente com um código de erro;
- Problemas inerentes:
  - A mensagem pode ser perdida pela rede;
  - Nomes de processos;
  - Autenticação;
  - Desempenho;
- Podemos usar caixa postal (mail boxes);
- MPI

Produto Consumidor com passagem de mensagem

```
#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 */
     }
}
```

Produto Consumidor com passagem de mensagem

```
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 de sincronização;
- Processos não podem avançar de fazer se todos não estiverem aptos a fazê-lo;
- Quando alcança a barreira, um processo fica bloqueado até que todos cheguem nela;

Barreiras

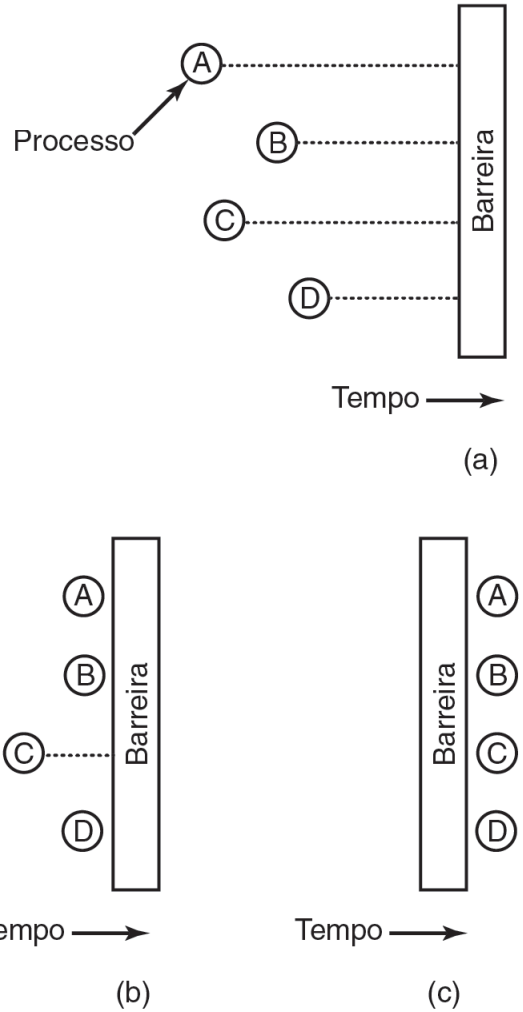


Figura 2.30 Uso de uma barreira. (a) Processos se aproximando de uma barreira. (b) Todos os processos, exceto um, estão bloqueados pela barreira. (c) Quando o último processo chega à barreira, todos passam por ela.

Escalonamento

- Em sistemas multiprogramados processos e threads competem pela CPU;
- O escaladador escolhe quem deve executar a partir da lista de prontos e usa um algoritmo de escalonamento;

Escalonamento - Introdução

- Sistemas de lotes: algoritmos simples, execute o próxima da fita;
- Sistemas multiprogramados: algoritmos complexos, vários usuários esperando pelo serviço;
- Um bom escalador faz diferença no desempenho observado;
- Computadores modernos: escalonamento não é tão importante em PCs de uso geral;
- Servidores e estações de alto desempenho: escalonamento é fundamental;
- A escolha deve ser bem feita, pois chavear processos é caro;

Escalonamento - Comportamento do processo

- Processos alternam surtos de computação com requisições de E/S;
- Processos podem ser limitados pela CPU (cpu bound) ou limitados pela E/S (I/O bound);
- A medida que cpus ficam mais rápidas, os processos passam a ser limitados por E/S. CPU mais rápida que o disco;

Escalonamento - Quando Escalonar?

- Executar o pai ou o filho? (fork);
- Ao término de um processo uma decisão deverá ser tomada;
- Quando um processo fica bloqueado por semáforo ou mutex, outro deve ser escalonado;
- Quando uma requisição de E/S termina;
- Uma decisão de escalonamento pode ser tomada a cada interrupção de relógio;
- Algoritmos podem ser preemptivos ou não;

Categorias dos algorítimos de escalonamento

- Em lote.
- Interativa.
- Tempo real.

Objetivos do algoritmo de escalonamento

Todos os sistemas	
	Justiça — dar a cada processo uma porção justa da CPU
	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 tarefas por hora
	Tempo de retorno — minimizar o tempo entre a submissão e o término
	Utilização de CPU — manter a CPU ocupada o tempo todo
Sistemas interativos	
	Tempo de resposta — responder rapidamente às requisições
	Proporcionalidade — satisfazer às expectativas dos usuários
Sistemas de tempo real	
	Cumprimento dos prazos — evitar a perda de dados
	Previsibilidade — evitar a degradação da qualidade em sistemas multimídia

Tabela 2.8 Alguns objetivos do algoritmo de escalonamento sob diferentes circunstâncias.

Escalonamento em sistemas em lote

Pode haver preempção ou não.

- Primeiro a chegar, primeiro a ser servido (FCFS).
- Tarefa mais curta primeiro (SJF).
- Próximo de menor tempo restante (SRTN).

Primeiro a chegar primeiro a ser servido (FCFS)

- Não preemptivo
- Simples de implementar

Tarefa mais curta primeiro (SJF)

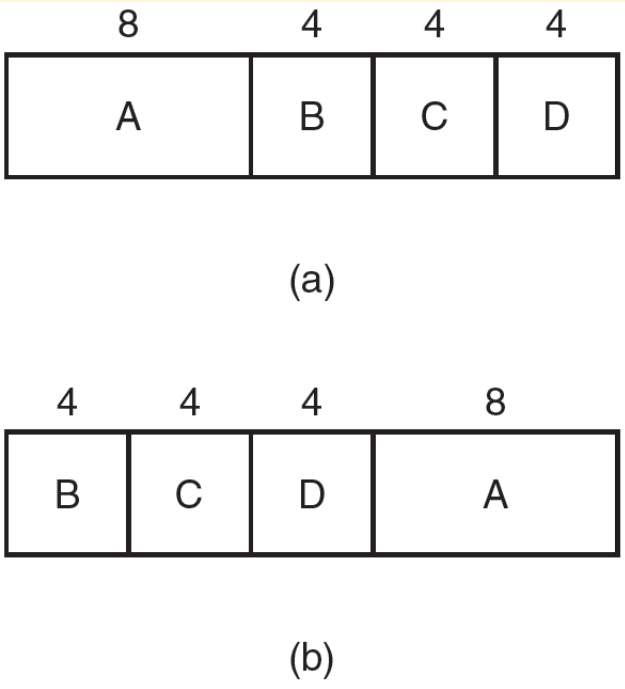


Figura 2.32 Um exemplo do escalonamento *tarefa mais curta primeiro*. (a) Execução de quatro tarefas na ordem original. (b) Execução na ordem *tarefa mais curta primeiro*.

Todas as tarefas tem que estar disponíveis.

Próximo de menor tempo restante (SRTN)

- Escolhe sempre a tarefa cujo tempo de execução seja o menor;
- O tempo de execução precisa ser conhecido previamente;

Escalonamento em sistemas interativos

- Escalonamento por chaveamento circular.
- Escalonamento por prioridades.
- Filas múltiplas.
- Próximo processo mais curto.
- Escalonamento garantido.
- Escalonamento por loteria.
- Escalonamento por fração justa.

Escalonamento por chaveamento circular (Round Robin)

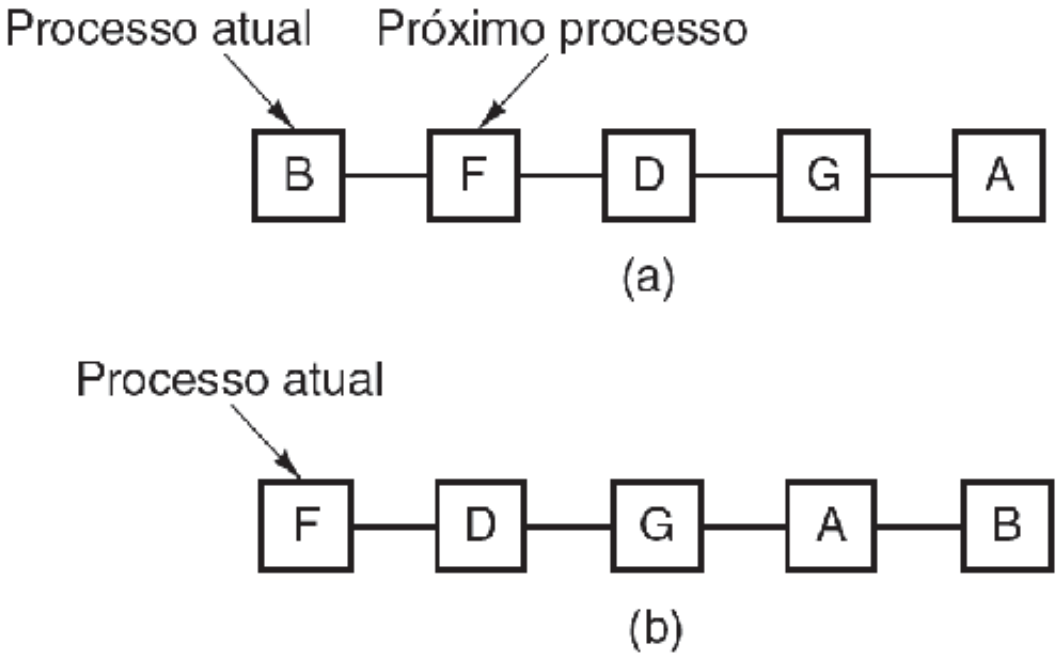


Figura 2.33 Escalonamento circular (*round robin*). (a) Lista de processos executáveis. (b) Lista de processos executáveis depois que *B* usou todo o seu quantum.

Escalonamento por chaveamento circular (Round Robin)

- O que importa é o tamanho do quantum;
- adotar um quantum curto causa muitos chaveamentos;
- um quantum muito grande gera uma resposta pobre;
- um quantum entre 20 ms a 50 ms é bastante razoável;

Escalonamento por prioridades

- Processos são escalonados de acordo com a prioridade;
- Prioridades pode ser atribuídas estática ou dinamicamente;
- Como evitar que processos com alta prioridade nunca deixem a CPU (starvation):
  - Reduz a prioridade a cada interrupção de relógio;
  - Atribui quantum máximo que um processo pode executar;

Escalonamento por prioridades

- Prioridades podem ser atribuídas para atingir certos objetivos;
  - Para processos orientados a E/S atribuir 1/f à prioridade,sendo f a fração do último quantum que o processo usou;
- Classes de prioridade podem ser definidas;
  - Exemplo: CTTS

Escalonamento por prioridades

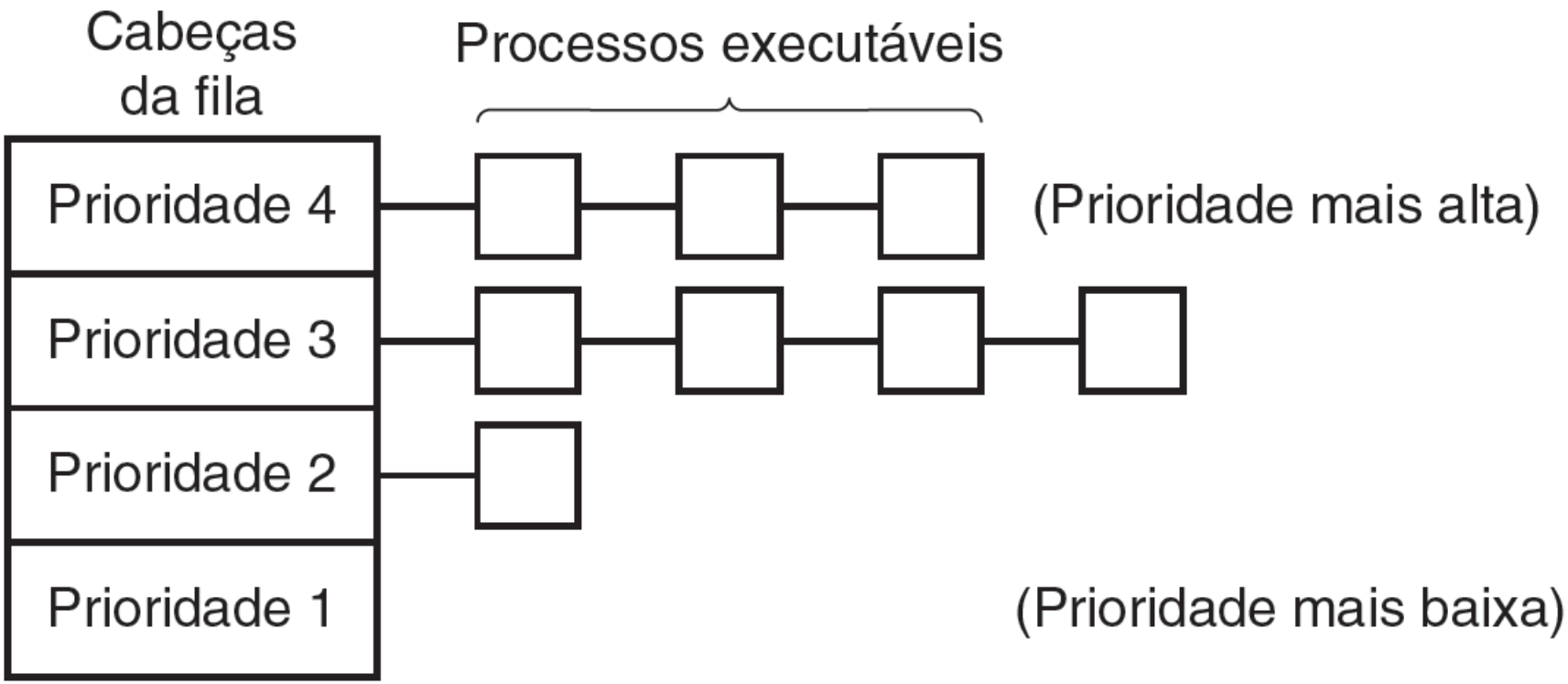


Figura 2.34 Um algoritmo de escalonamento com quatro classes de prioridade.

Próximo processo mais curto (shortest process next)

- Mesma idéia dos sistemas em lote;
- Problema é saber qual dos processos é o mais curto;
- Solução: Estimativas - aging;
- $aT_0 + (1 - a)T_1$

Escalonamento Garantido

- Fazer promessas sobre o desempenho;
- Exemplo: se houver n usuários conectados cada um receberá $1/n$ do tempo de CPU;

Escalonamento por loteria

- Cada processo tem um bilhete onde o prémio são recursos da máquina.
- O escalonador sorteia um bilhete e o processo que tem esse bilhete ganha o direito de executar.
- Processos mais importantes podem obter bilhetes extras.
- Processos bloqueados podem dar seus bilhetes a outros processos.

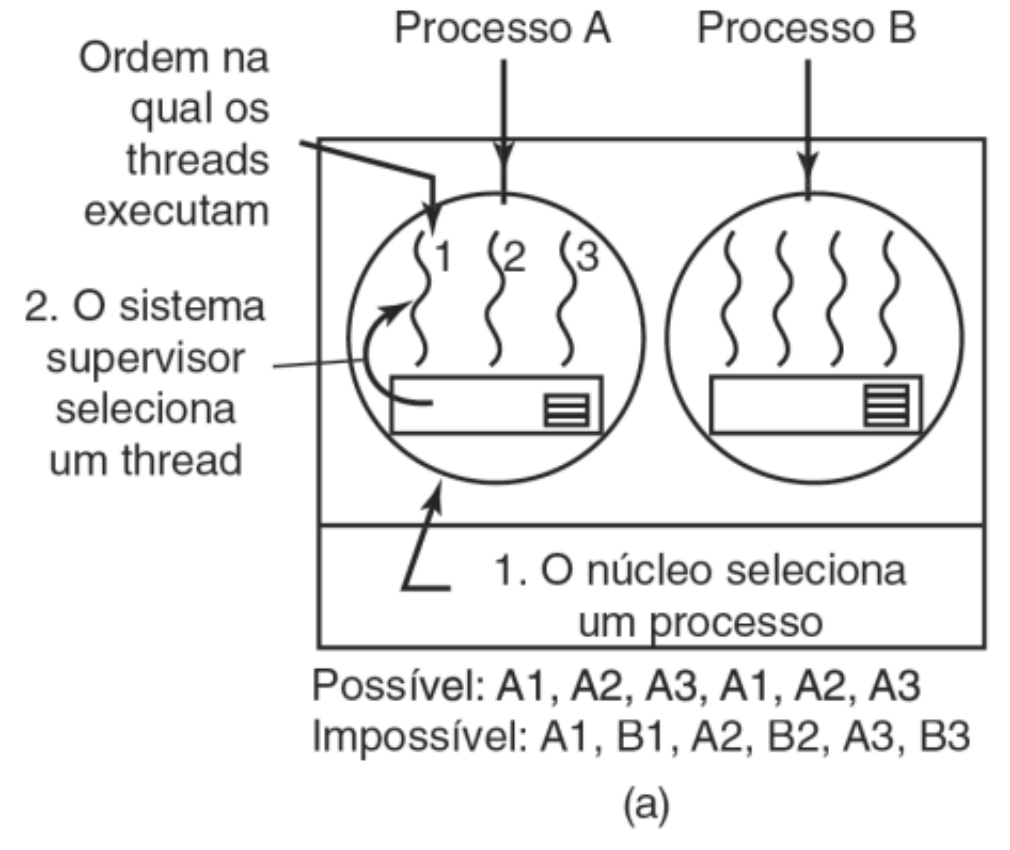
Escalonamento por fração justa (fair-share)

- Os algoritmos anteriores não se preocupam com o número de usuário do sistema;
- Nesse modelo cada usuário recebe uma fração da CPU e o escalonador garante que esta fração será respeitada;

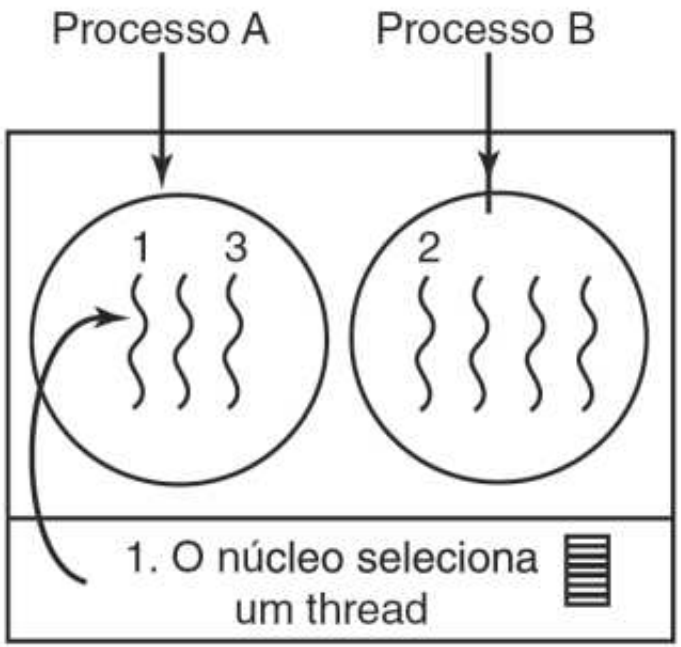
Política versus mecanismo

- Um processo pode ter filhos e saber quais seus filhos tem maior prioridade;
- Mas até agora dissemos que o escalonador que define qual processo excutar e nem sempre ele faz a melhor escolha; Solução:
  - Separar o mecanismo de escalonamento da política de escalonamento;
  - permitir que processos sejam capazes de configurar e alterar as propriedades dos filhos;
  - O mecanismo está no núcleo mas a política está na mão do usuário;

Escalonamento de threads



Escalonamento de threads



Possível: A1, A2, A3, A1, A2, A3
Também possível: A1, B1, A2, B2, A3, B3

(b)

Figura 2.35 (a) Escalonamento possível de threads de usuários com um quantum de processo de 50 ms e threads que executam 5 ms por surto de CPU. (b) Escalonamento possível de threads de núcleo com as mesmas características de (a).

O problema do jantar dos filósofos

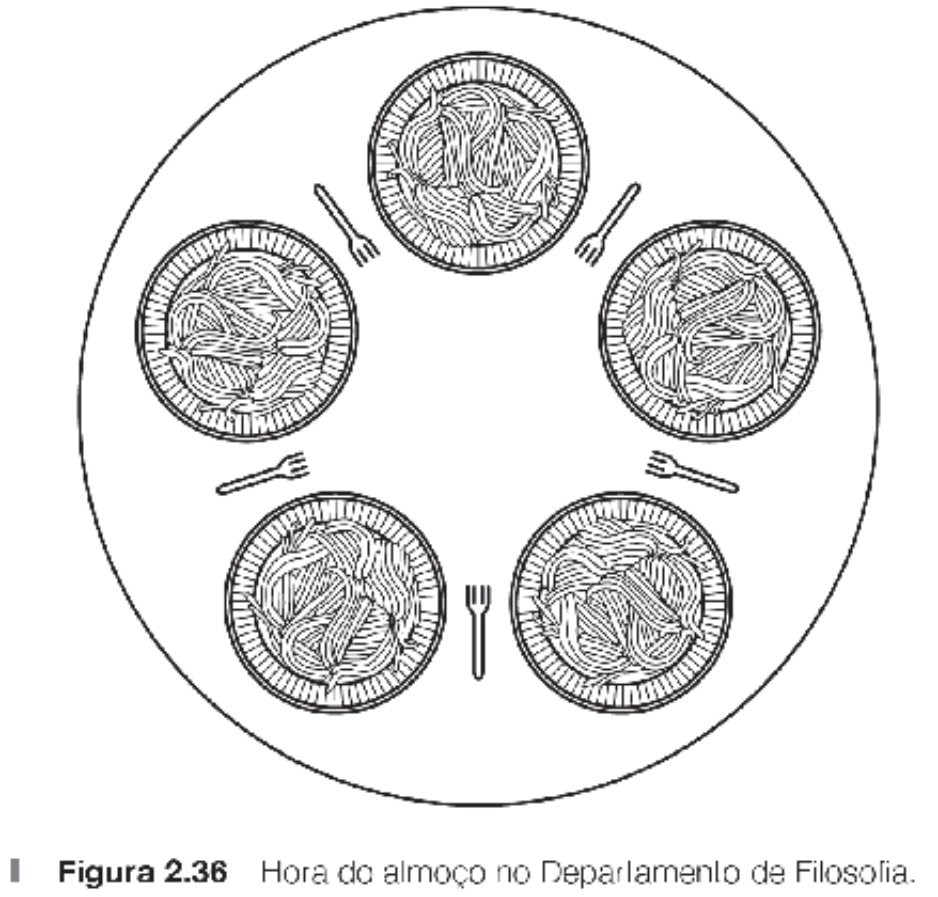


Figura 2.36 Hora do almoço no Departamento de Filosofia.

O problema do jantar dos filósofos

```
#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 módulo */
                eat( );                         /* hummm! Espaguete */
                put  fork(i);                   /* devolve o garfo esquerdo à mesa */
                put_fork((i+1) % N);            /* devolve o garfo direito à mesa */
        }
}
```

Figura 2.37 Uma solução errada para o problema do jantar dos filósofos.

O problema do jantar dos filósofos

```
#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 */
      }
}
```

O problema do jantar dos filósofos

```
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 */
}
```

O problema do jantar dos filósofos

```
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]);
      }
}
```

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 */
    }
}
```

O problema dos leitores e escritores

```
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 */
     }
}
```