UFFS
Ciência da Computação

Sistemas Operacionais

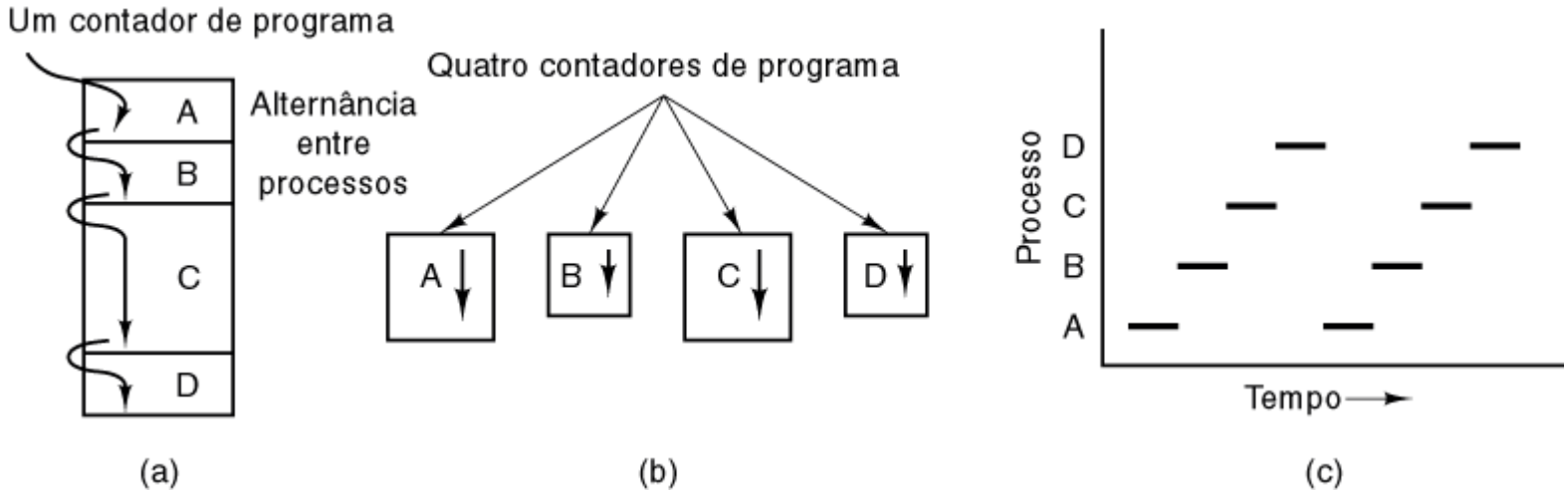
Prof. Marco Aurélio Spohn

Capítulo 2

Processos e *Threads*

2.1 Processos
2.2 *Threads*
2.3 Comunicação interprocesso
2.4 Problemas clássicos de IPC
2.5 Escalonamento

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

Hierarquias de Processos

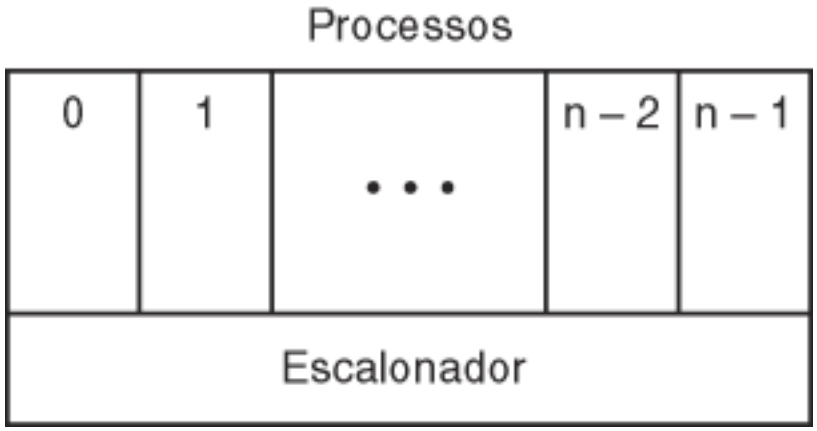
- Pai cria um processo filho, processo filho pode criar seu próprio processo
- Formam uma hierarquia
  - UNIX chama isso de “grupo de processos”
- Windows não possui o conceito de hierarquia de processos
  - Todos os processos são criados iguais

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

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): no xv6

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  volatile int pid;            // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

Implementação de Processos (3): no xv6

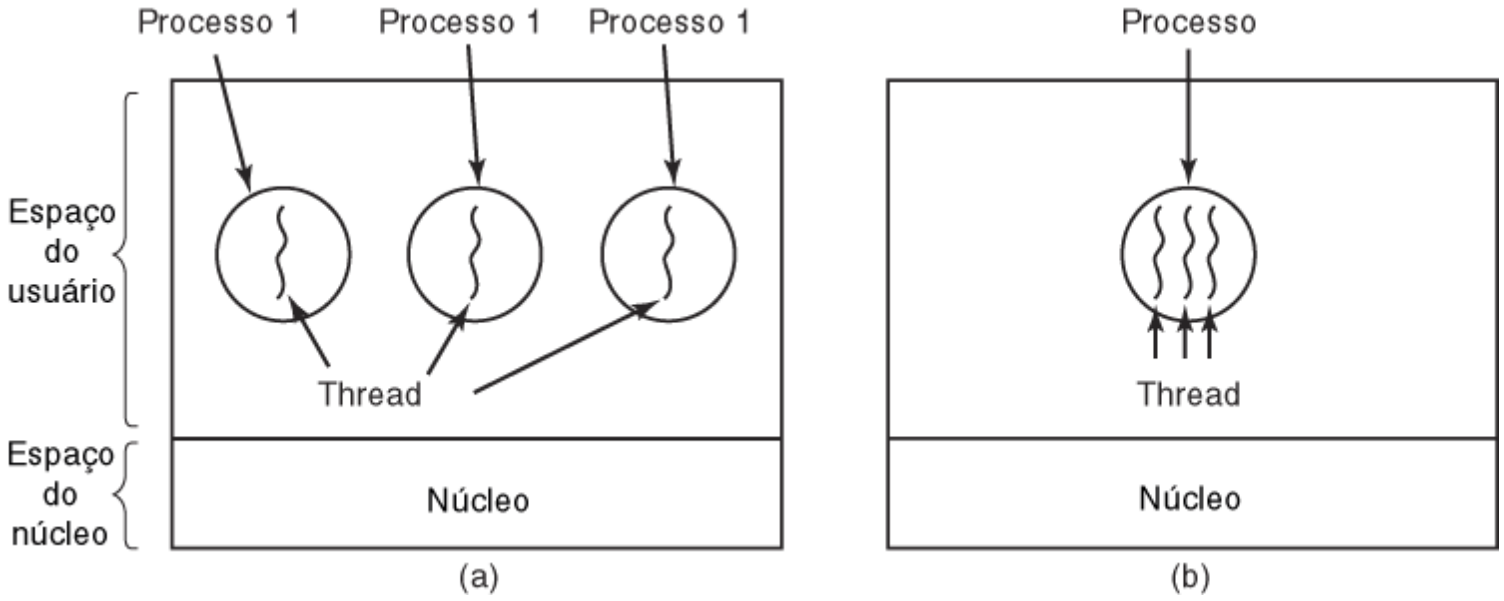
```
//PAGEBREAK: 17
// Saved registers for kernel context switches.
// Don't need to save all the segment registers (%cs, etc),
// because they are constant across kernel contexts.
// Don't need to save %eax, %ecx, %edx, because the
// x86 convention is that the caller has saved them.
// Contexts are stored at the bottom of the stack they
// describe; the stack pointer is the address of the context.
// The layout of the context matches the layout of the stack in swtch.S
// at the "Switch stacks" comment. Switch doesn't save eip explicitly,
// but it is on the stack and allocproc() manipulates it.
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

Implementação de Processos (4)

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: sequência pode variar, dependendo do SO. OBS.: embora tudo (código *assembly* e C) é convertido para linguagem de máquina, destaca-se que algumas partes do SO são programadas diretamente em linguagem *assembly* para enfatizar que tais instruções (e.g., salvar registradores) não podem ser descritas diretamente em linguagem C.

Threads
O Modelo de Thread (1)



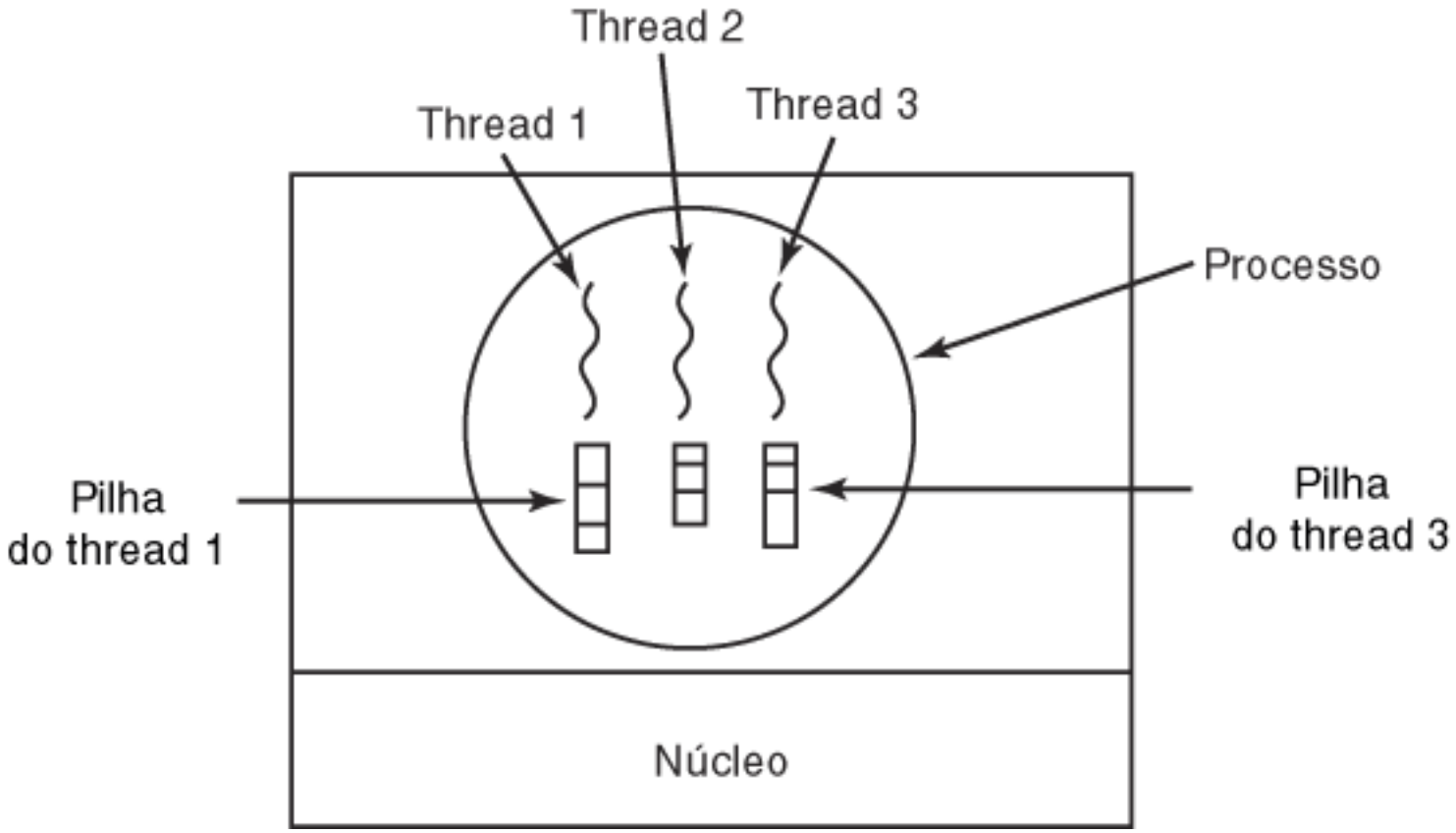
a) Três processos cada um com um *thread*
- (b) Um processo com três *threads*

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	

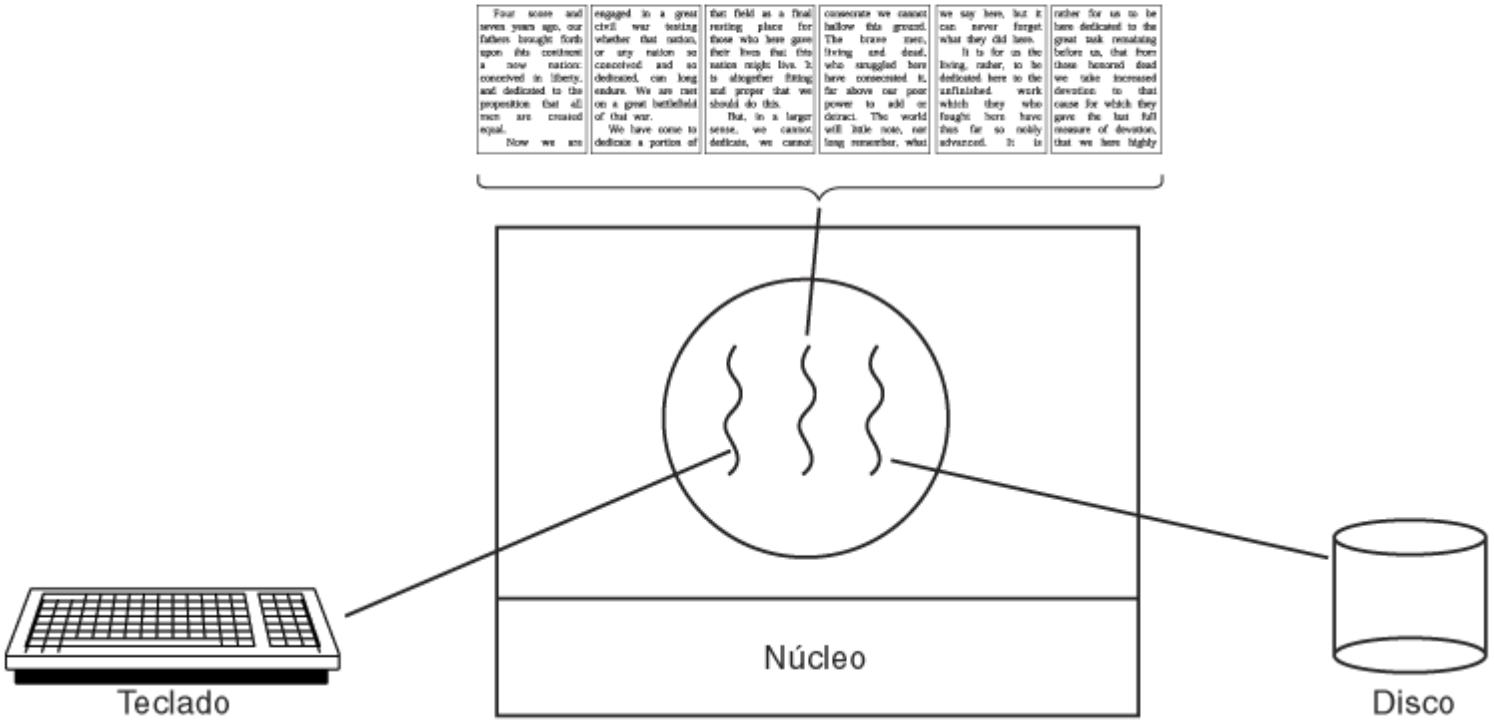
- Items compartilhados por todos os threads em um processo
- Itens privativos de cada thread

O Modelo de Thread (3)



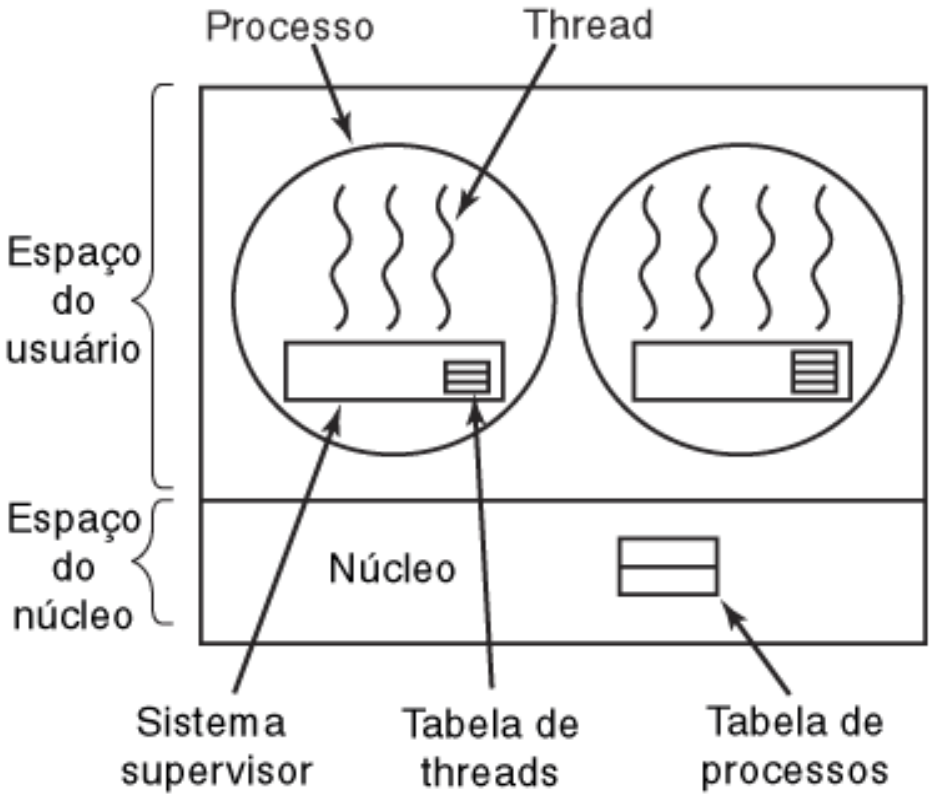
- Cada thread tem sua própria pilha

Uso de Threads (1)



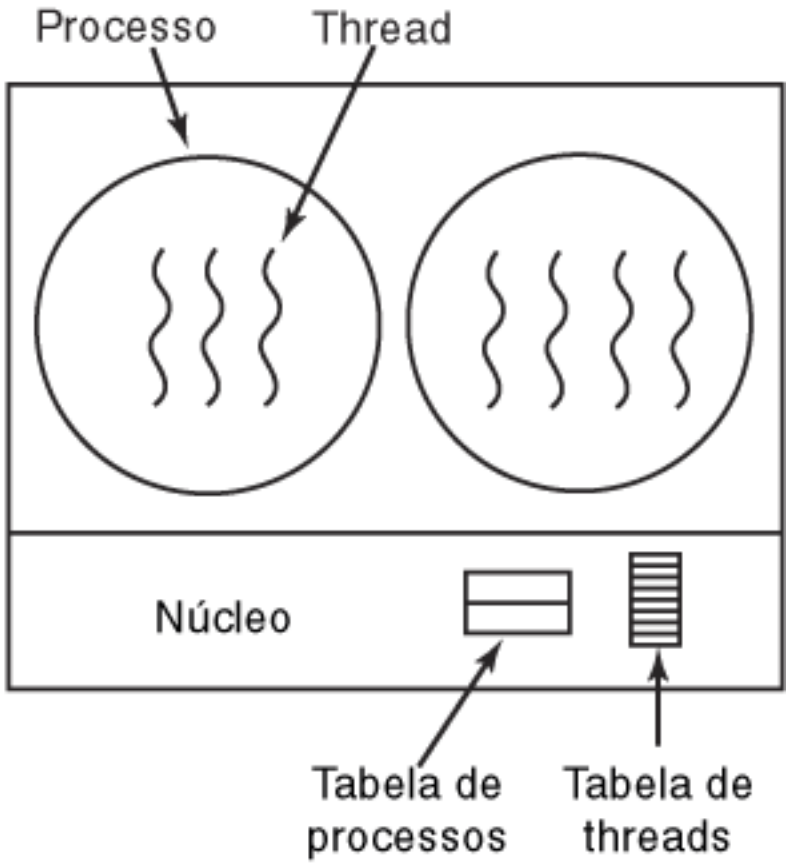
- Um processador de texto com três threads

Implementação de Threads de Usuário



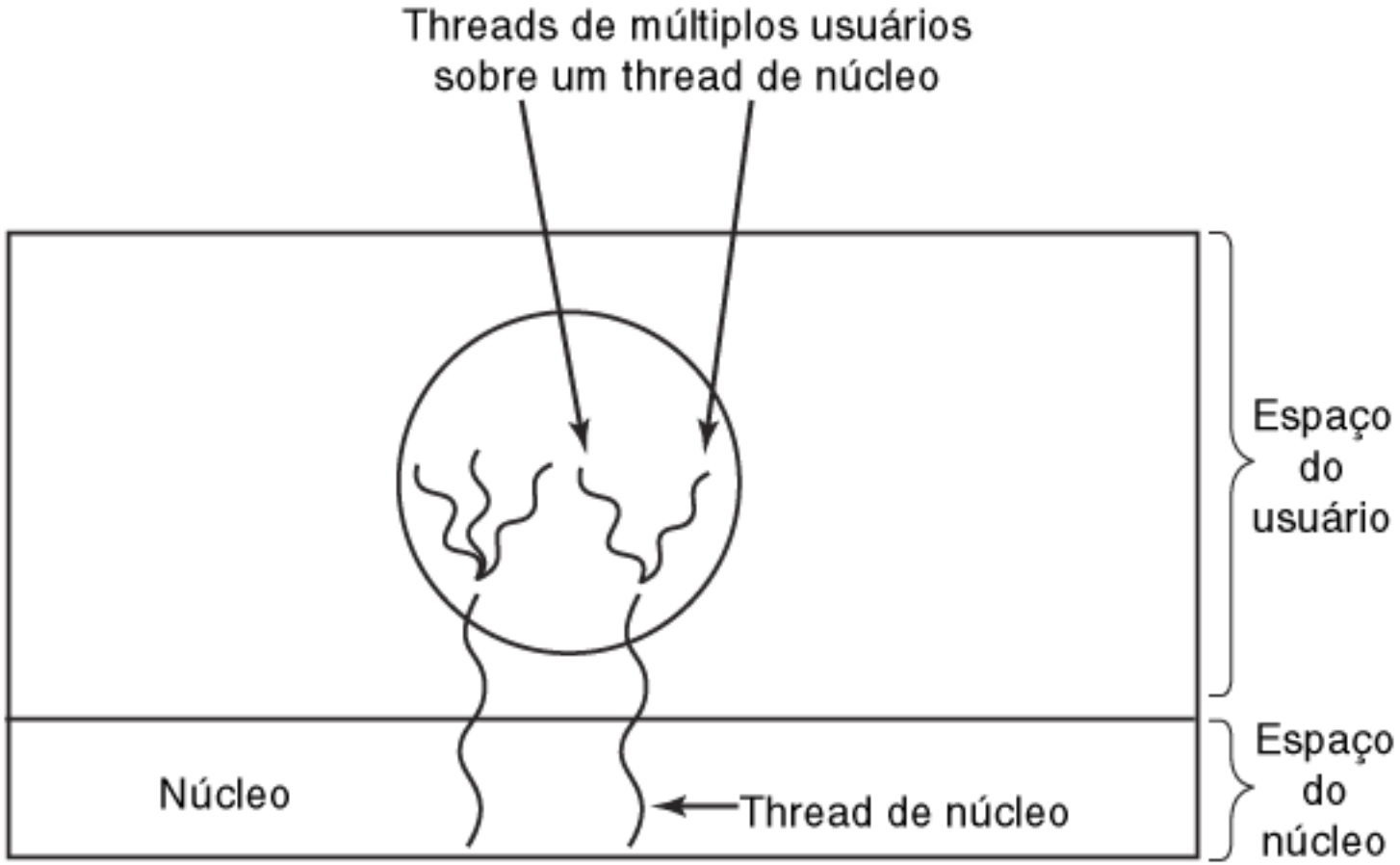
- Um pacote de threads de usuário

Implementação de Threads de Núcleo



- Um pacote de threads gerenciado pelo núcleo

Implementações **Híbridas**

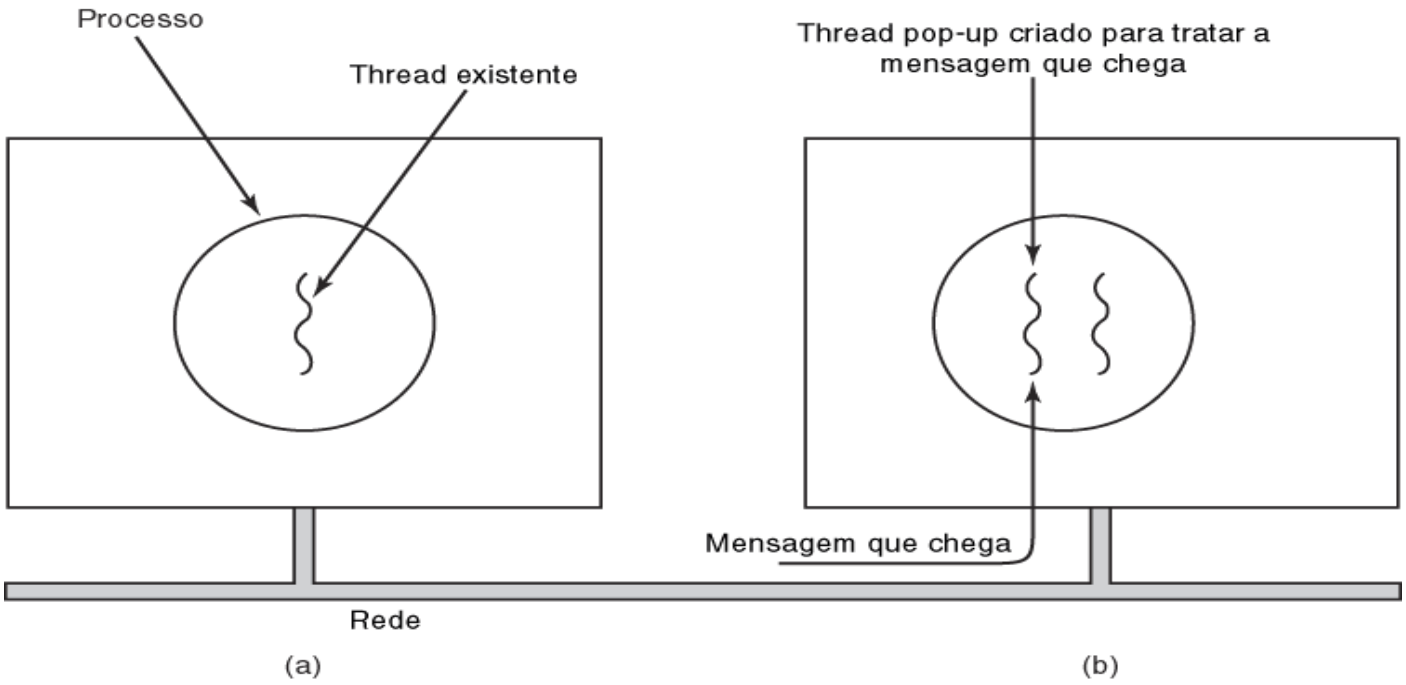


Multiplexação de threads de usuário sobre threads de núcleo

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)

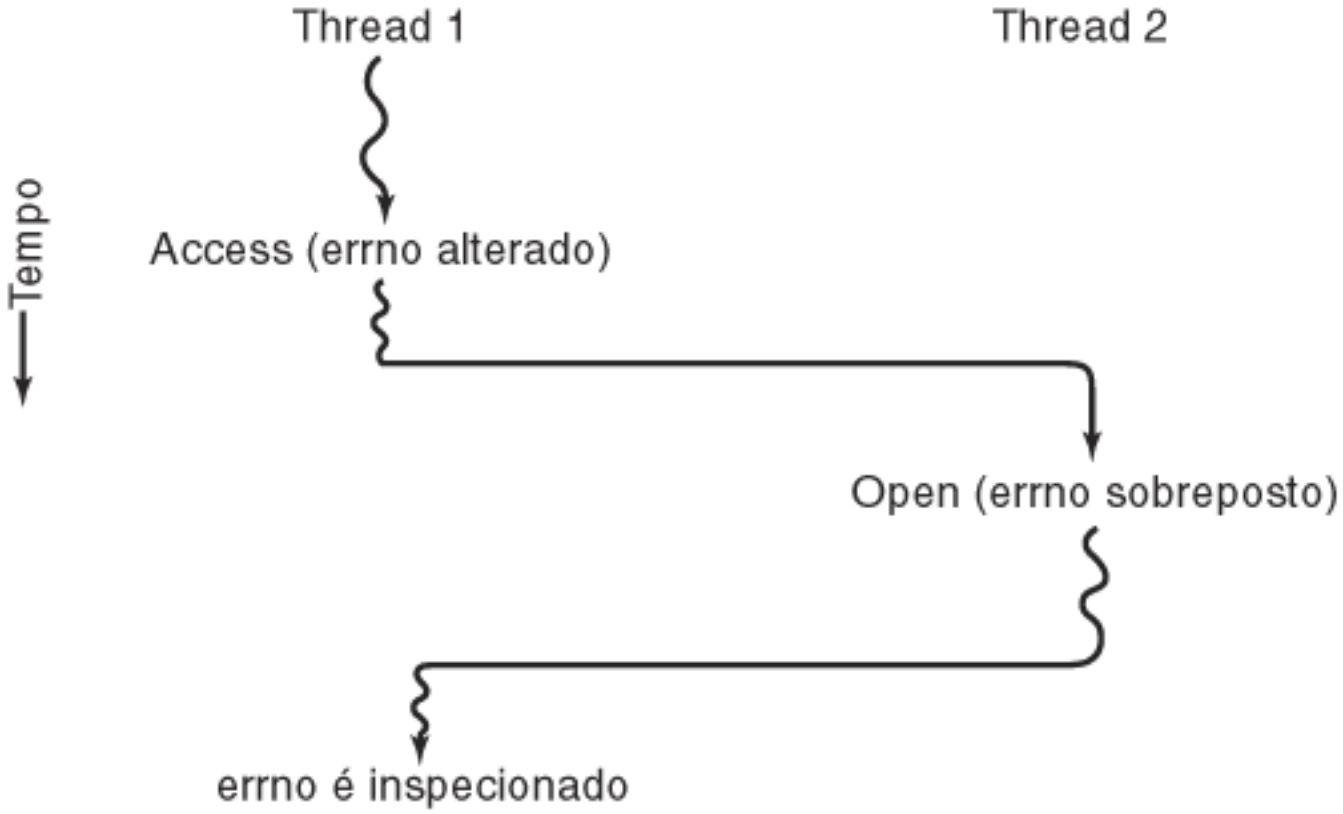
Threads Pop-Up



- Criação de um novo *thread* quando chega uma mensagem

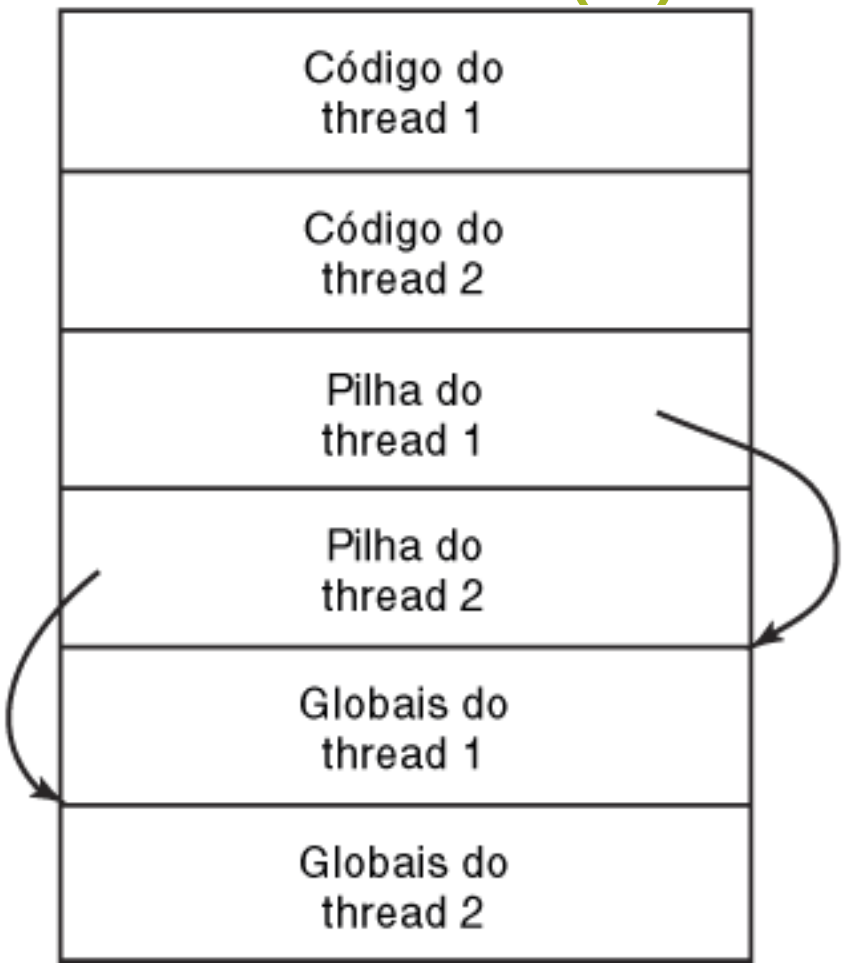
(a) antes da mensagem chegar
(b) depois da mensagem chegar

Convertendo Código *Monothread* em Código *Multithread* (1)



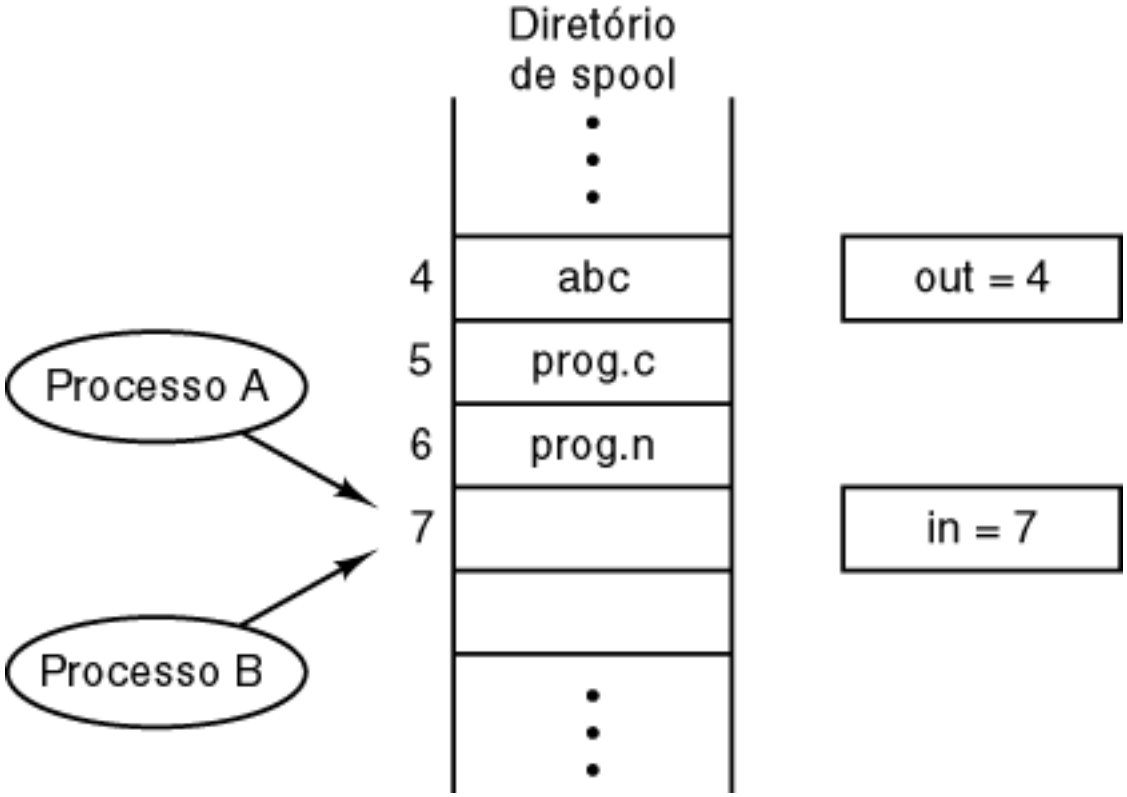
- Conflitos entre threads sobre o uso de uma variável global (após *thread* 2 alterar código em *errno*, *thread* 1 perde o valor que havia sido retornado a ela)

Convertendo Código *Monothread* em Código *Multithread* (2)



- *Threads* podem ter variáveis globais privadas

Comunicação Interprocesso
Condições de Disputa (*race conditions*)



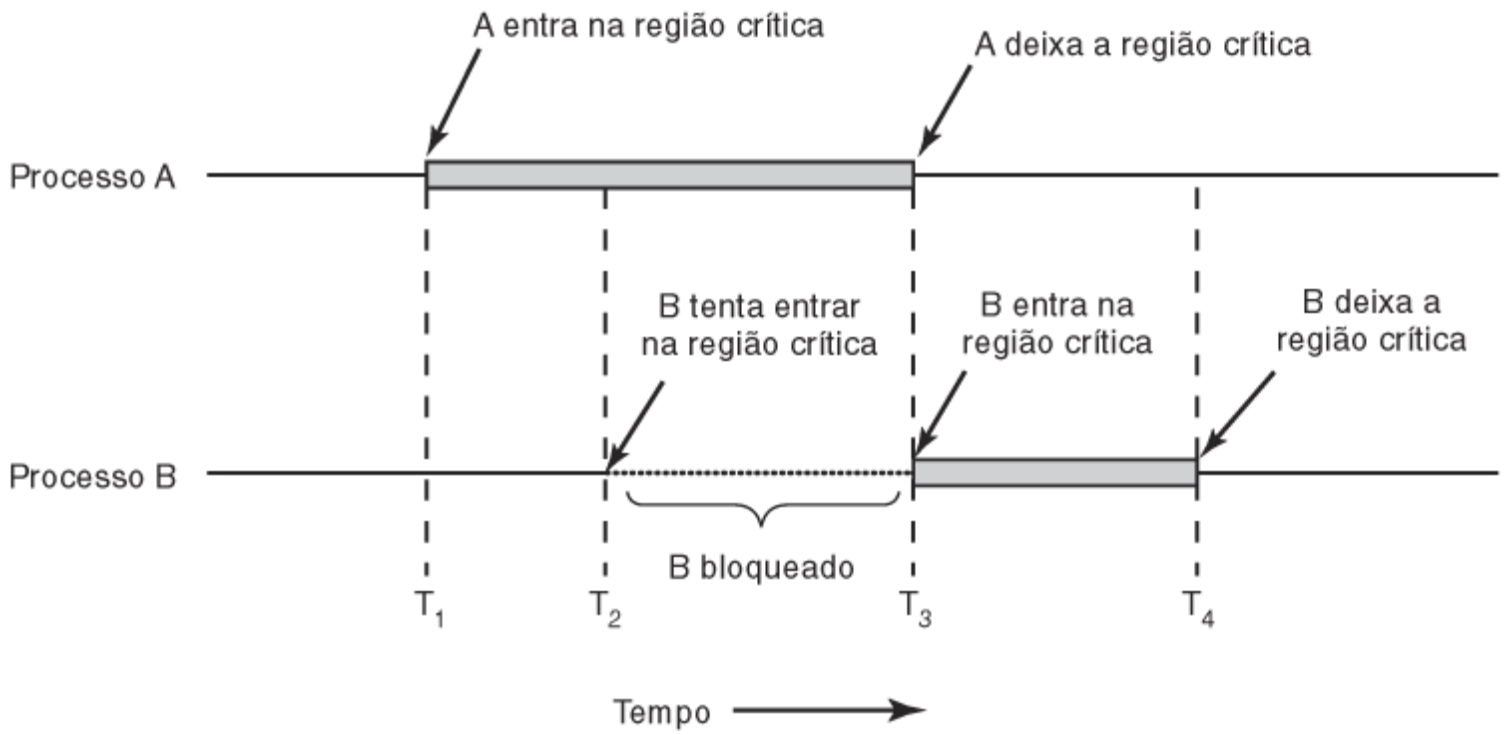
- Dois processos querem ter acesso simultaneamente à memória compartilhada (no exemplo da fila de impressão, apenas o último processo a atualizar a fila do *spooler* conseguirá incluir seu arquivo na fila).

Regiões Críticas (1)

Quatro condições necessárias para prover uma boa solução à exclusão mútua:

1. Nunca dois processos simultaneamente em uma região crítica
2. Nenhuma afirmação/exigência 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 (***deadlock***??)

Regiões Críticas (2)



- Exclusão mútua usando regiões críticas

Exclusão Mútua com Espera Ociosa (1): toda fatia de CPU é desperdiçada no segundo laço verificando o valor da variável *turn* (laço "vazio"); Problema: **viola a condição 3**.

```
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) com *test and set lock (TSL):* instrução atômica, impede o acesso ao barramento da memória para impedir que outras CPUs tenham acesso (**desabilitar todas as interrupções não produziria o mesmo efeito, pois não afetaria outras CPUs**)

```
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 (variável *count*): **caso sinal de acordar seja enviado para um processo que ainda não está dormindo, o sinal é perdido!!!**

Semáforos: contador atualizado via ações atômicas, **indicando o número de "sinais de acordar" salvos para uso futuro**.

```
#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: semáforo simplificado, sem a opção de contagem

```
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): uma construção da linguagem de programação para implementar exclusão mútua; apenas um processo por vez acessa o monitor; variáveis condicionais: *wait* bloqueia processo caso monitor esteja ocupado e *signal* desbloqueia algum dos processos esperando (escalonador escolhe).
OBS.: *signal* não opera como um contador; caso nenhum processo esteja esperando, o sinal é perdido.

```
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;
```

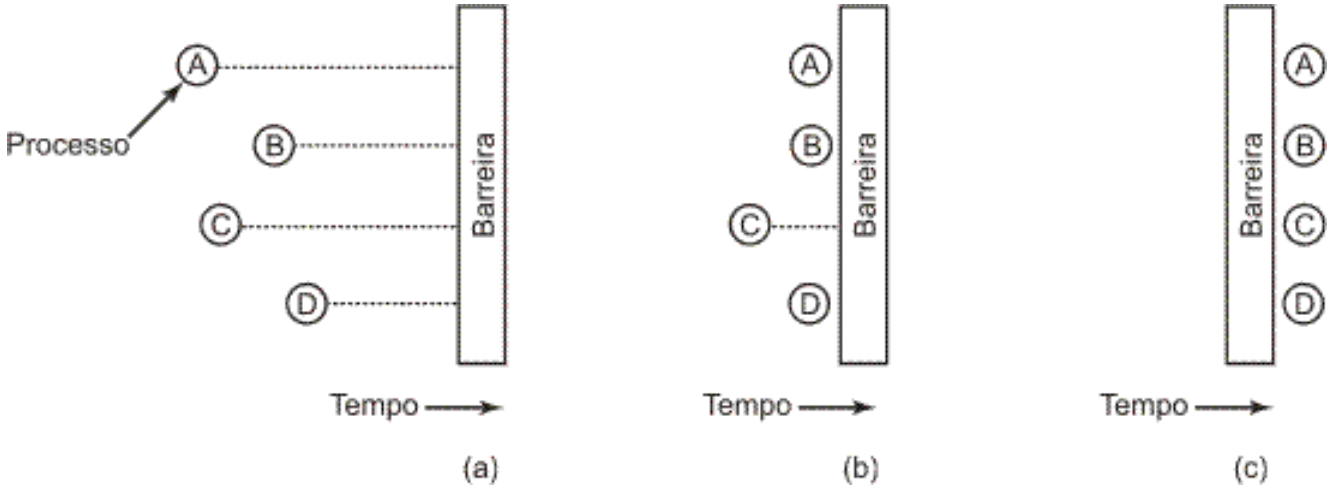
- Delineamento do problema do produtor-consumidor com monitores
  - somente um procedimento está ativo por vez no monitor
  - o buffer tem N lugares

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

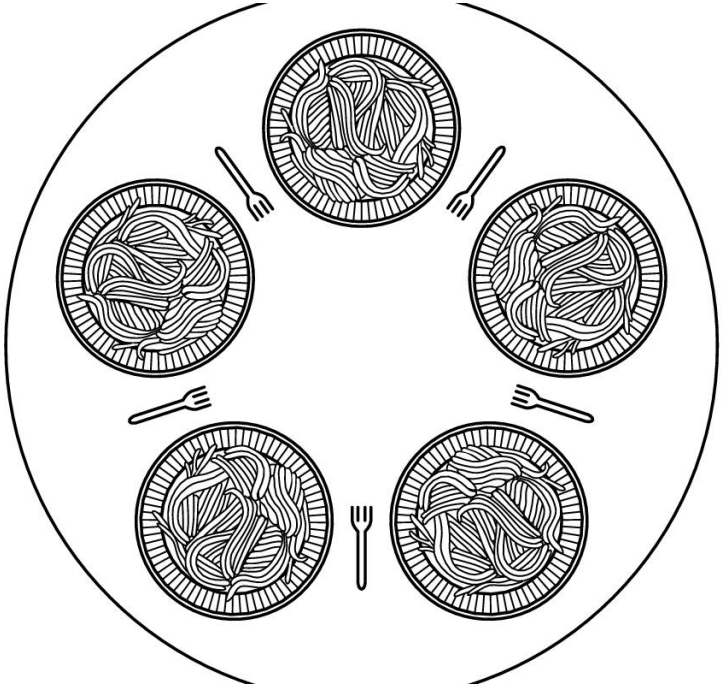
Barreiras



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

Jantar dos Filósofos (1)

- Filósofos comem/pensam
- Cada um precisa de 2 garfos para comer (não questione, apenas coma com os dois garfos :)
- Pega um garfo por vez
- Como prevenir *deadlock* (impasse)??



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 (correta**?**) 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]);
      }
}
```

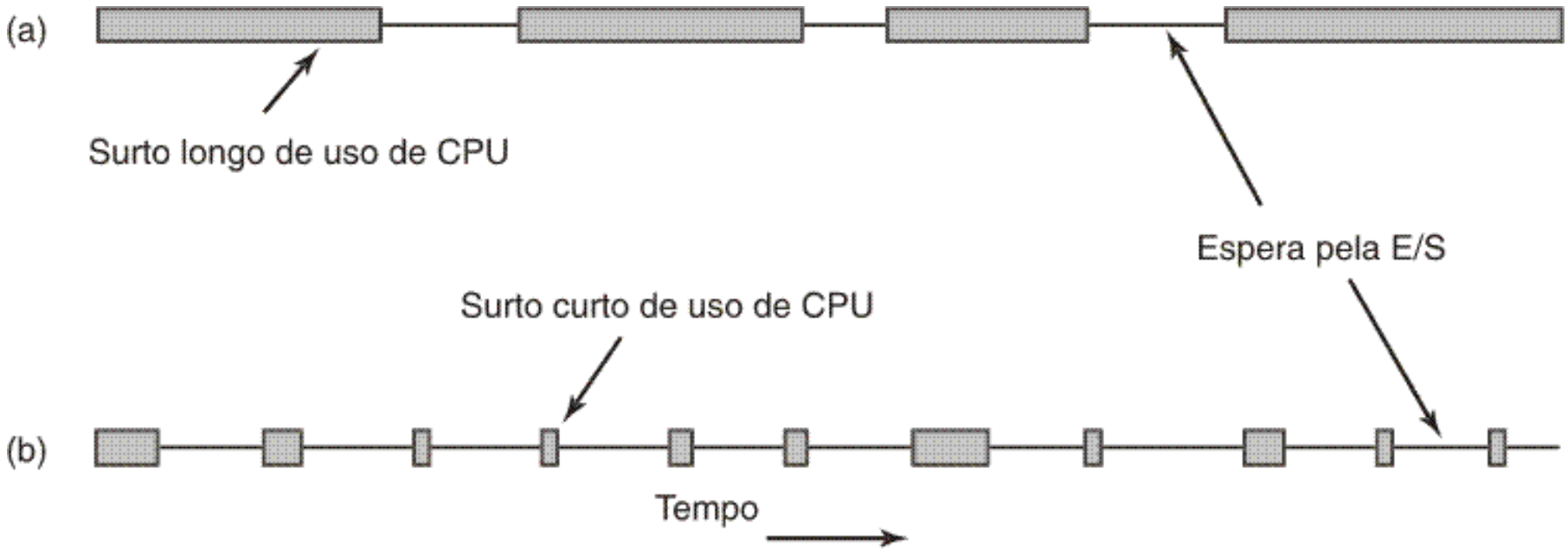
- Uma solução para o problema do jantar dos filósofos (parte 2)

O Problema dos Leitores e Escritores (modela o acesso a banco de dados)

```
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

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)

- Primeiro a chegar, primeiro a ser servido.
- Tarefa mais curta primeiro.
- Próximo de menor tempo restante.

Escalonamento em Sistemas em Lote (1)

- *First-Come First-Served*

- FILA CLÁSSICA, RESPEITANDO-SE A ORDEM DE CHEGADA!!!

Escalonamento em Sistemas em Lote (2)

- Um exemplo (figura (b)) de escalonamento do tipo *job mais curto primeiro (shortest job first)*: *nesse exemplo assume-se que todos o jobs chegaram simultaneamente*

Escalonamento em Sistemas em Lote (2)

- Caso os processos tenham tempos de chegada distintos, como fica o *Shortest Job First*. Vejamos um exemplo:
- Processos A a E, com tempos de execução: 2, 4, 1, 1 e 1
- Com tempo de chegada: 0, 0, 3, 3 e 3, respectivamente
- Execução seria: A, B, C, D e E; qual o tempo médio de espera (para finalizar a execução de cada *job*)?
- E se executássemos na seguinte ordem: B, C, D , E, A... qual seria o tempo médio de espera?

Escalonamento em Sistemas em Lote (2)

- *Shortest Remaining Time Next*: solução preemptiva (caso chegue um mais curto, pode “preemptar”/retirar quem está executando)... novamente o mesmo problema com abordagens semelhantes: como saber quanto tempo ainda resta para o *job* finalizar sua execução?

Escalonamento em Sistemas Interativos: no *xv6*

```
//PAGEBREAK: 42
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run
//  - swtch to start running that process
//  - eventually that process transfers control
//      via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p;
  for(;;){
    // Enable interrupts on this processor.
    sti();
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();
      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
    }
    release(&ptable.lock);
  }
}
```

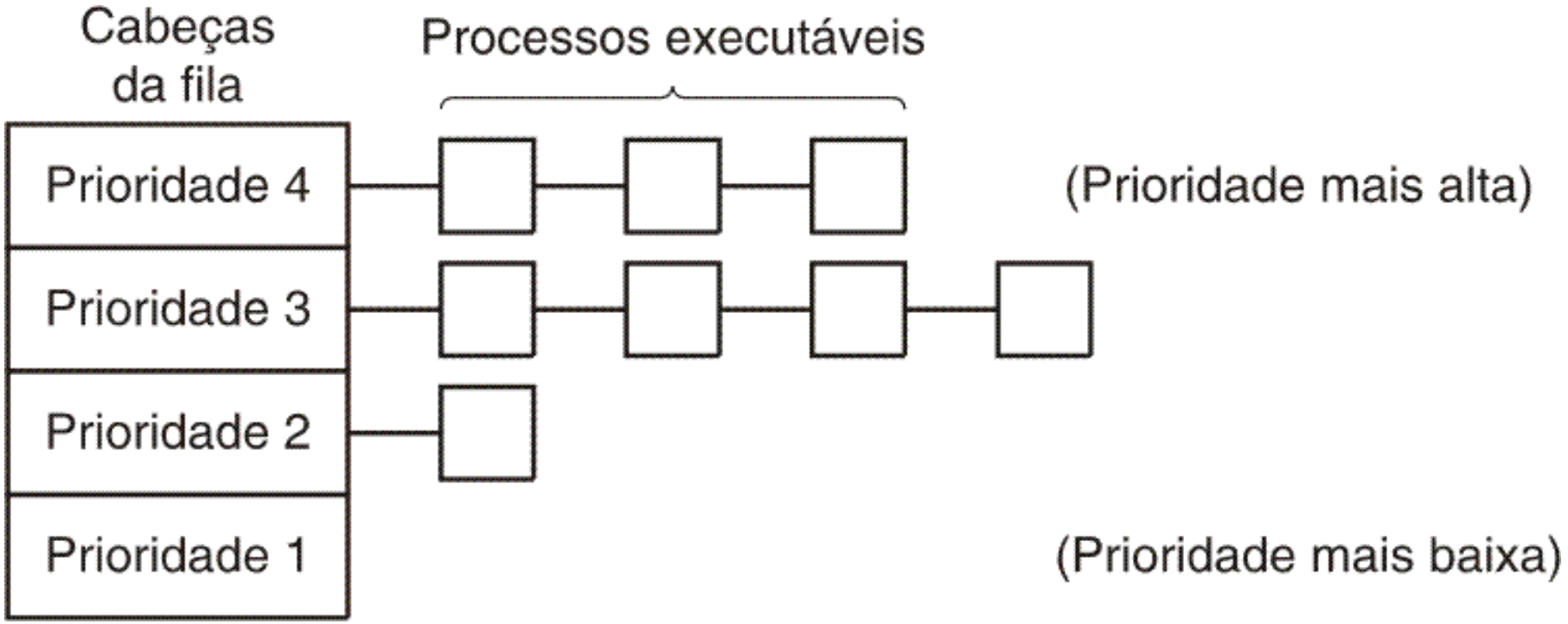
- Cada CPU tem o seu escalonador.
- - *switchuvm(p)*: atualiza a memória virtual (vm:virtual memory) no espaço do usuário para o processo p
- - *switchkvm()*: atualiza a memória virtual para o kernel (em modo kernel)

Escalonamento em Sistemas Interativos: no *xv6*

```
// from source file trap.c
void trap(struct trapframe *tf)
{
  ...
   // Force process to give up CPU on clock tick.
   // If interrupts were on while locks held, would need to check nlock.
   if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
     yield();

   ...
}
// from source file proc.c

// Give up the CPU for one scheduling round.
void yield(void)
{
   acquire(&ptable.lock);  //DOC: yieldlock
   proc->state = RUNNABLE;
   sched();
   release(&ptable.lock);
}
...
void sched(void)
{
     ...
     swtch(&proc->context, cpu->scheduler);
     ...
}
```

- **No tratamento às interrupções (*trap.c*), caso seja uma interrupção de relógio (IRQ_TIMER) enquanto o processo estiver executando (RUNNING), aciona o escalonador (yield()) para seleção de outro processo pronto para continuar executando.**

Escalonamento em Sistemas Interativos (2)



- Um algoritmo de escalonamento com quatro classes de prioridade: há possibilidade de processos morrerem por inanição (*starvation*)

Escalonamento em Sistemas Interativos (3)

- *Multiple Queues*: - *High priority (one quantum); - second high (two quanta); third high (4 quanta); etc...* Rebaixa para fila inferior caso utilize toda fatia: reduz *swapping* e dá prioridade a processos interativos!

Visão geral: processos *CPU intensive* acabam tendo prioridade reduzida mas, quando selecionados, conseguem uma fatia maior (mais *quanta*) da CPU; processo *I/O intensive* acabam tendo prioridade maior mas, como utilizam pouco a CPU, devolvem ela logo; como processos que precisam frequentemente da CPU permanecem nela mais tempo, acabam sofrendo menos baixas (*swap out*) para o disco!!

Escalonamento em Sistemas Interativos (4)

- *Shortest process next* (agora no contexto de sistemas interativos): como não é possível prever quem é o processo mais curto dentre os remanescentes, utiliza-se uma métrica que reflita o envelhecimento/*aging* (com média ponderada da estimativa de tempo de execução):
- a*T_0 + (1-a)T_1 (T_0 é a medida/previsão anterior e T_1 é a medida atual)

Escalonamento em Sistemas Interativos (5)

- Escalonamento garantido (fração igual: 1/n): necessita-se contabilizar, continuamente, a quantidade (*i.e.*, tempo) de CPU utilizada por cada processo; o escalonador escolhe sempre o processo com a taxa de utilização mais baixa (a fim de atingir a divisão equânime)

Escalonamento em Sistemas Interativos (5)

Escalonamento por loteria: cada processo recebe um determinado número de *tickets* ao iniciar; a cada sorteio (i.e., seleção de processo pelo escalonador), o processo detentor do *ticket* sorteado é executado; assumindo que há um total de 100 *tickets* distribuídos entre os processos, um determinado processo que detenha 20 dos *tickets* terá, após determinado período de tempo, conseguido em torno de 20% do tempo de CPU.

Escalonamento em Sistemas Interativos (5)

- Escalonamento por fração justa: considerar usuário, não somente os processos isoladamente.

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

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

Escalonamento em Sistemas de Tempo-Real

- Então a carga poderá ser tratada somente se

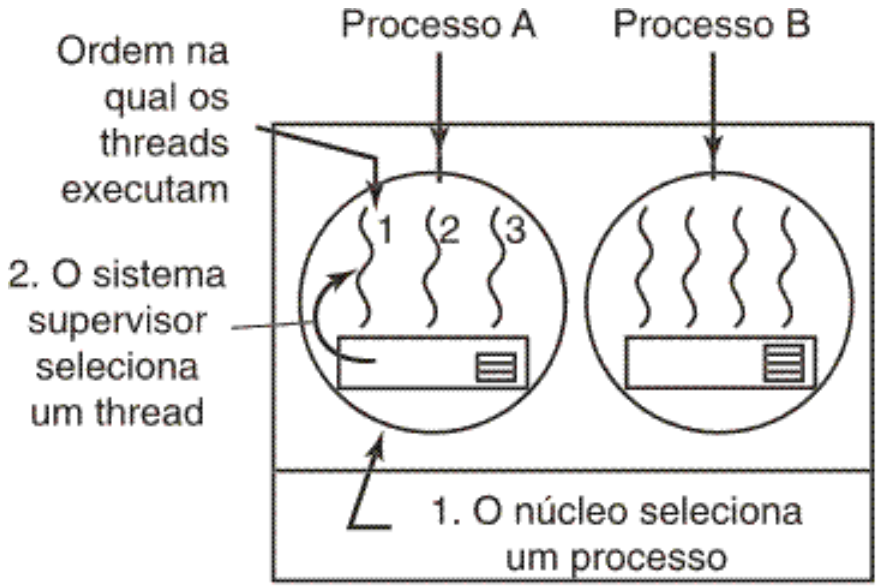
$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

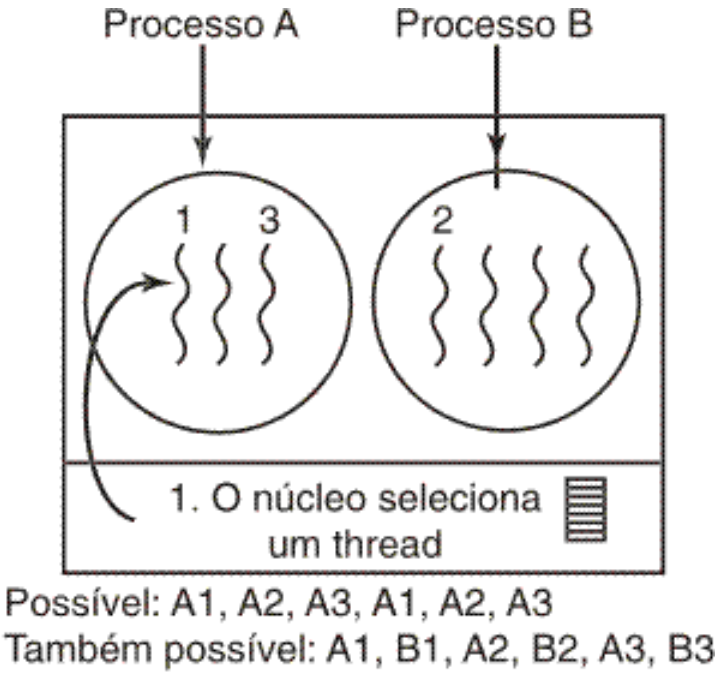
- Ex.: três eventos períodicos com períodos de 100, 200 e 500 ms que requerem, respectivamente, 50, 30 e 100 ms de tempo de CPU. O sistema é escalonável porque 0.5 + 0.15 + 0.2 = 0.85 < 1
- Caso um quarto evento com um período de 1 s seja adicionado, o sistema permanecerá escalonável somente caso esse evento não necessite mais do que 150 ms de CPU por evento!!

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: o usuário, definindo diferentes prioridades para suas *threads*, direciona a sequência de execução das mesmas

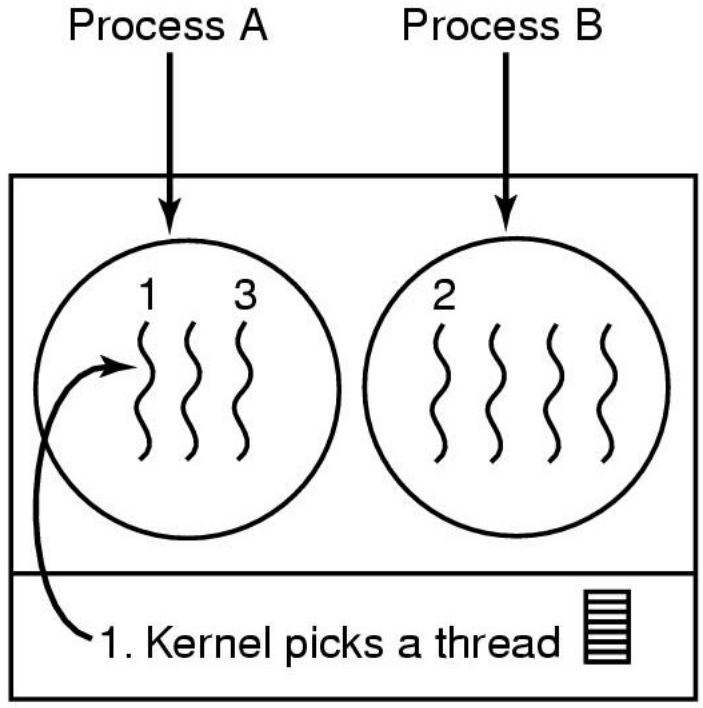
Escalonamento de Threads (1)





- Possível escalonamento de threads de usuário
  (obs.: figura mostra escalonamento parcial para apenas 30 dos 50 ms)
- processo com *quantum* de 50 ms
- *threads* executam 5 ms por surto de CPU

Escalonamento de Threads (2)



Possível escalonamento de *threads* de núcleo

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