Infra-Estrutura de Software

Processos, Threads, Concorrência

Criação de Processos

Principais eventos que levam à criação de processos

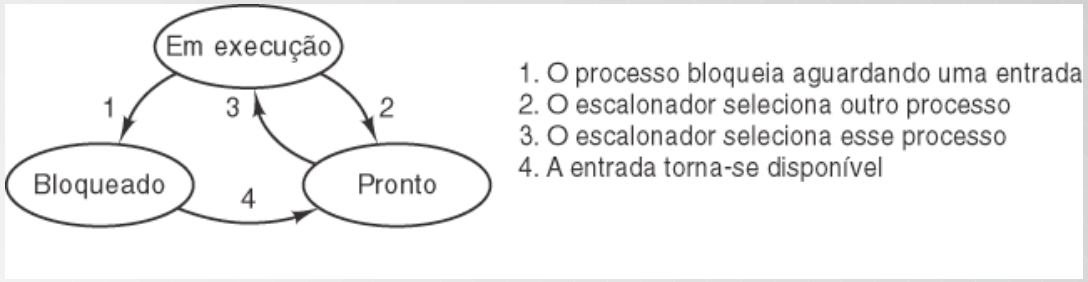
- Início do sistema
- Execução de chamada ao sistema de criação de processos
- Solicitação do usuário para criar um novo processo
- Início de um job em lote

Término de Processos

Condições que levam ao término de processos

- Saída normal (voluntária) – programado
- Saída por erro (voluntária) – programado
- Erro fatal (involuntário)
- Cancelamento por um outro processo (involuntário)

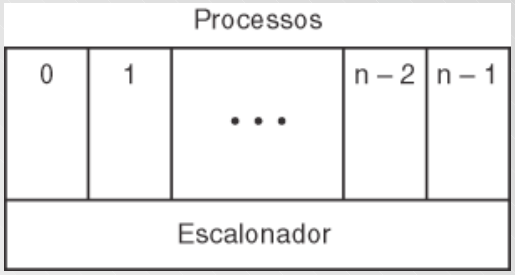
Estados de Processos



- Possíveis estados de processos
  - em execução
  - bloqueado
  - pronto

Centro de Informática U·F·P·E

Camadas de Processos



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

Centro de Informática U·F·P·E

Escalonamento de processos

- **Qual processo** executa **quando**
  - Processos **prontos**
- A parte do sistema operacional responsável por essa decisão é chamada escalonador
  - O algoritmo usado para tal é chamado de algoritmo de escalonamento
- Para que um processo não execute tempo demais, são usadas interrupções de relógio
  - Praticamente todos os computadores dão suporte a isso

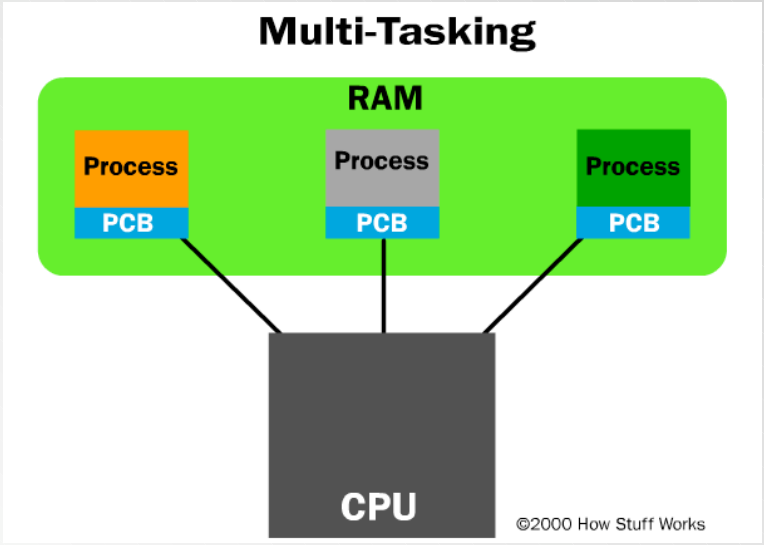
Implementação de Processos (1)

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

Centro de Informática U·F·P·E

Múltiplos 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.

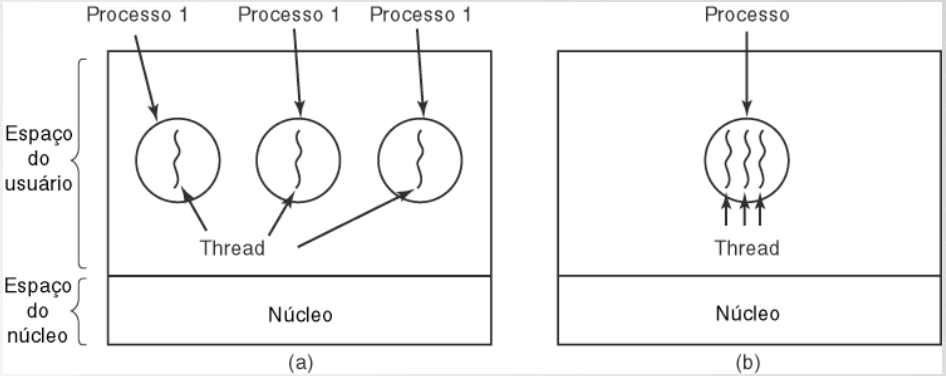
Esqueleto do que o nível mais baixo do SO faz quando ocorre uma interrupção

Utilização de CPU

- Modelada em termos do tempo bloqueado de um processo
  - Um processo passa uma fração *p* de seu tempo esperando por E/S
  - Considerando-se que há *n* processos na memória
  - A utilização da CPU seria dada por:

$$\text{Utilização CPU} = 1 - p^n$$

Threads
O Modelo de Thread (1)



- Três processos, cada um com um thread
- Um processo com três threads

Threads: Motivação
Concorrência

- Problemas:
  - Programas que precisam de mais poder computacional
  - Dificuldade de implementação de CPUs mais rápidas

- Solução:
  - Construção de computadores capazes de executar várias tarefas simultaneamente

Problemas com Concorrência

- Não-determinismo
  - x = 1 || x = 2
    - Qual o valor de "x" após a sua execução?
- Dependência de Velocidade
  - [[f(); x = 1]] || [[g(); x = 2]]
  - O valor final de x depende de qual das funções, f() e g(), terminar primeiro
- *Starvation*
  - Processo de baixa prioridade precisa de um recurso que nunca é fornecido a ele...

Problemas com Concorrência (cont.)

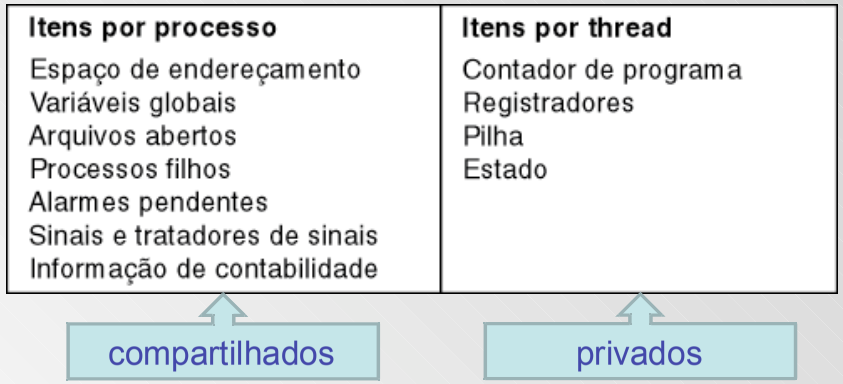
- *Deadlock*
  - Um sistema de bibliotecas só fornece o "nada consta" para alunos matriculados e o sistema de matricula só matricula os alunos perante a apresentação do "nada consta"

  – **Definição**: dois processos bloqueiam a sua execução pois um precisa de um recurso bloqueado pelo outro processo

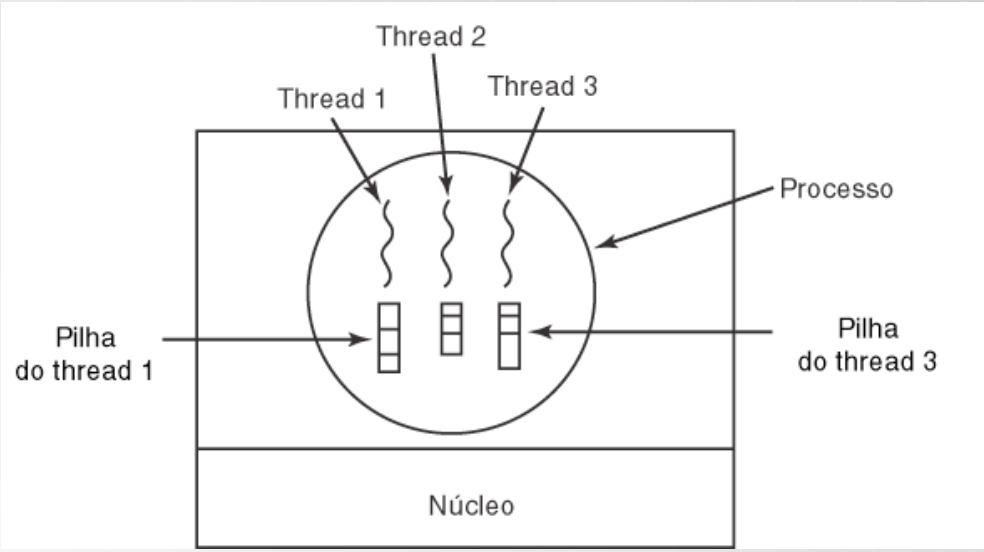
  – Técnicas para lidar com não determinismo podem levar a *deadlock*

  Conceitos: **starvation** e **deadlock**

O Modelo de Thread (2)



O Modelo de Thread (3)

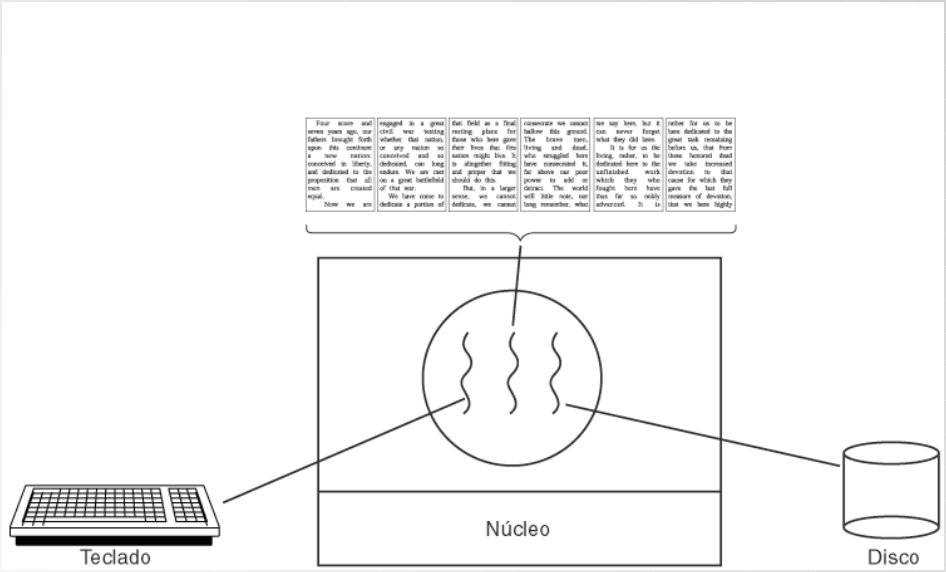


Cada thread tem sua própria pilha

Concorrência cooperativa

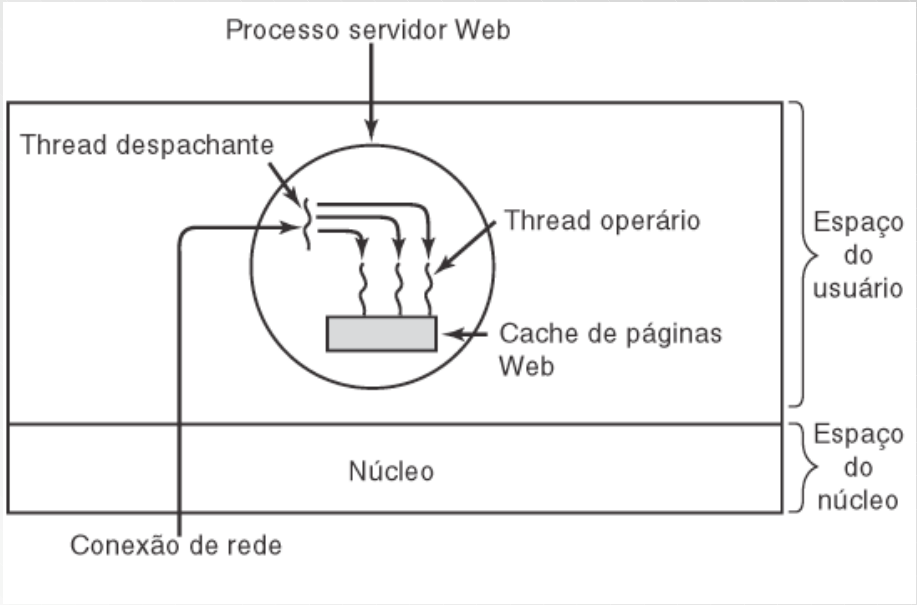
Centro de Informática U·F·P·E

Uso de Threads (1)



Um processador de texto com três threads

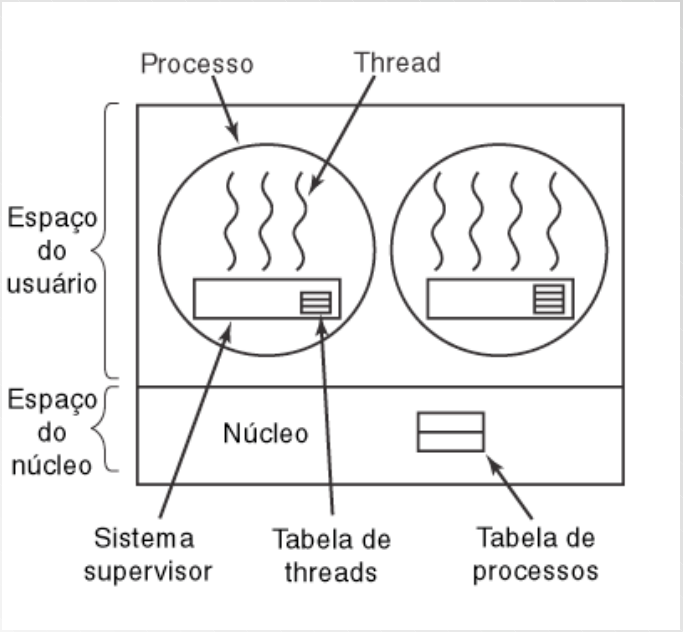
Uso de Threads (2)



Um servidor web com múltiplos threads

vs um serviço Web com múltiplos servidores (mais adiante – módulo II)

Implementação de Threads de Usuário



Um pacote de threads de usuário

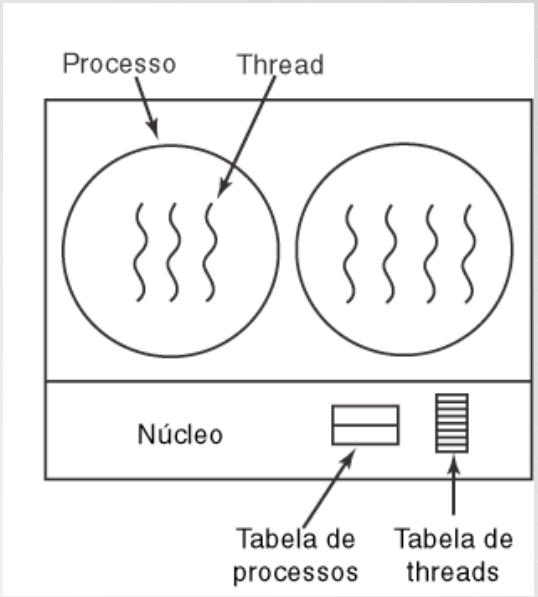
Por que *threads* em modo usuário?

+ são mais escaláveis
(sem espaço de tabela ou de pilha no núcleo)
+ permitem personalização do escalonamento
+ exigem menos transições entre modo usuário e modo núcleo

Por que não?

- É difícil implementar chamadas de sistema com bloqueio.
  - É necessário antever se tal chamada de sistema causaria bloqueio ou não
  - ...ou todas as threads do processo poder ser bloqueadas por causa de apenas uma.
- *Page faults* bloqueiam todas as *threads* de um processo
- Exigem uma infraestrutura adicional de tempo de execução
- Em aplicações que realizam muitas operações de E/S, seria mais barato o núcleo gerenciar as *threads*

Implementação de Threads de Núcleo

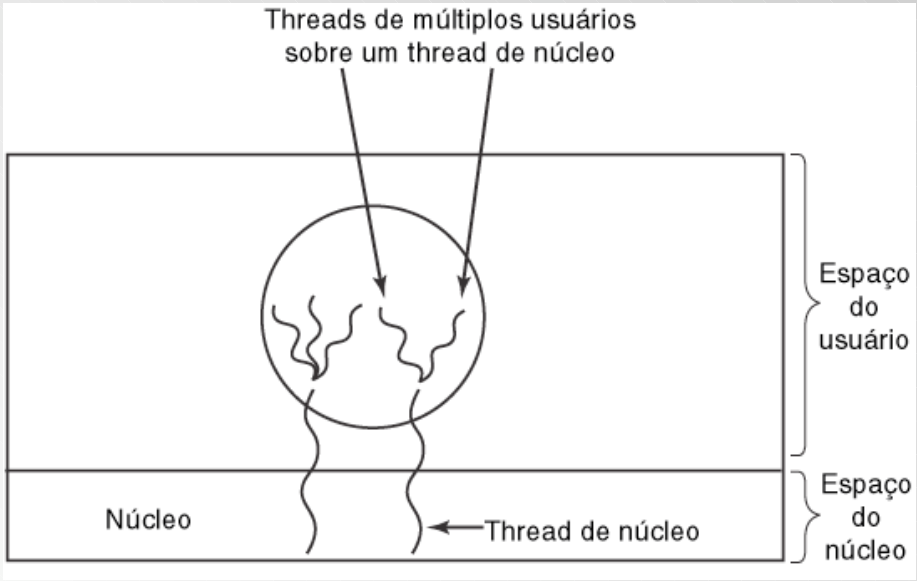


Um pacote de threads gerenciado pelo núcleo

Threads em modo núcleo

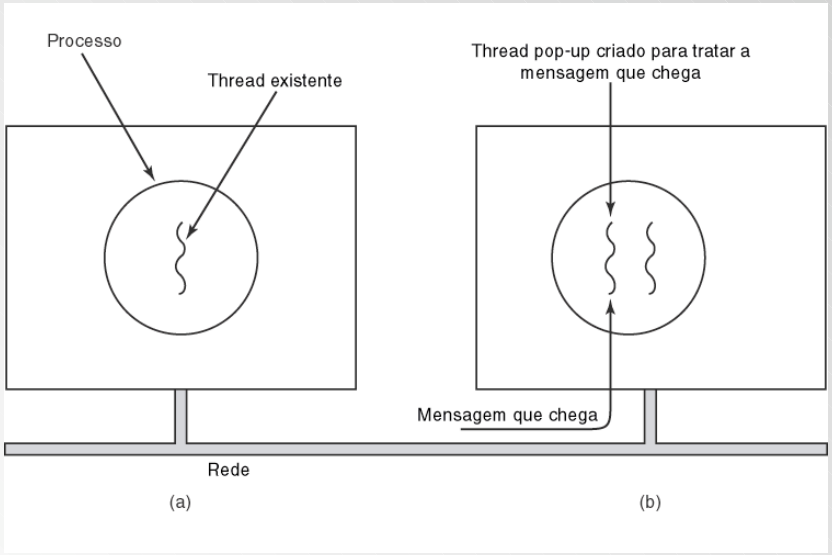
- Não apresentam vários dos problemas das *threads* em modo usuário
- Mas apresentam outros...
  - Quando um processo é bifurcado, o que fazer com as *threads* do processo original?
  - Quando um processo recebe um sinal, qual *thread* deve tratá-lo?
  - Todas as chamadas bloqueantes são chamadas de sistema.
- São comumente recicladas

Implementações Híbridas



Multiplexação de threads de usuário sobre threads de núcleo

Criação de um novo thread quando chega uma mensagem

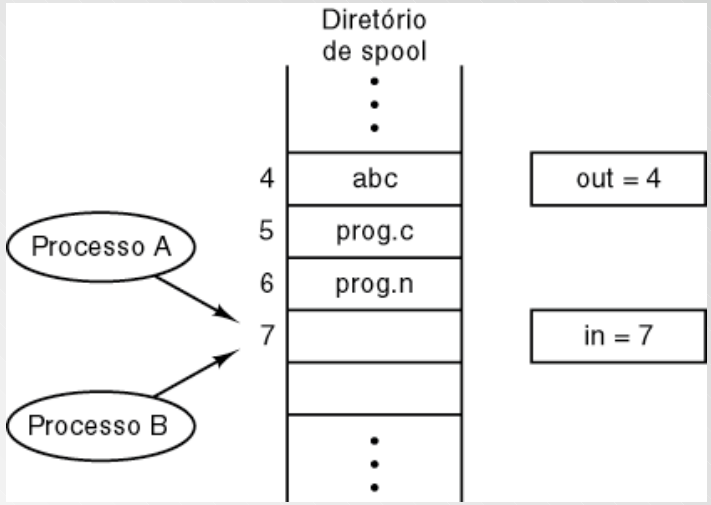


Concorrência

Próxima aula…

Comunicação Interprocesso

Condições de Disputa/Corrida



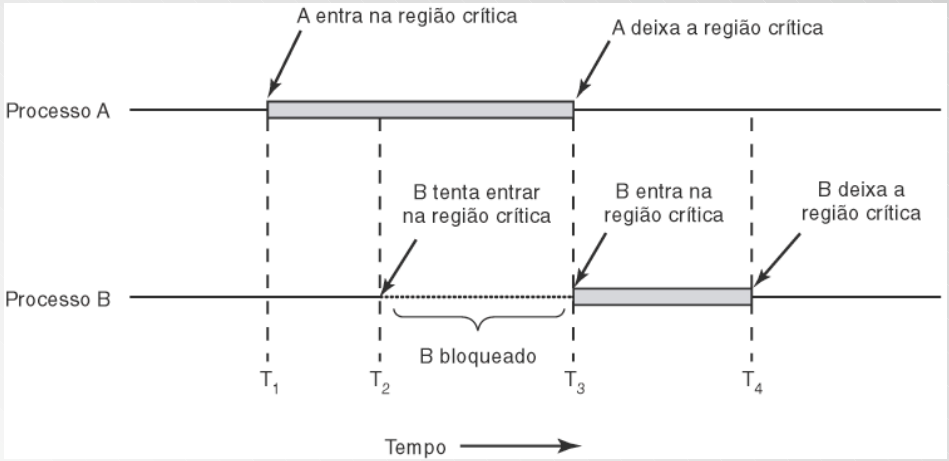
Dois processos querem ter acesso simultaneamente a um recurso compartilhado

Centro de Informática U·F·P·E

Regiões Críticas (1)

- Quatro condições necessárias para prover exclusão mútua:
  - Dois processos nunca podem estar simultaneamente em uma mesma região crítica
  - Não se pode considerar velocidades ou números de CPUs
  - Nenhum processo executando fora de sua região crítica pode bloquear outros processos
  - Nenhum processo deve esperar eternamente para entrar em sua região crítica

Regiões Críticas (2)



Exclusão mútua usando regiões críticas

Exclusão Mútua com Espera Ocupada (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.

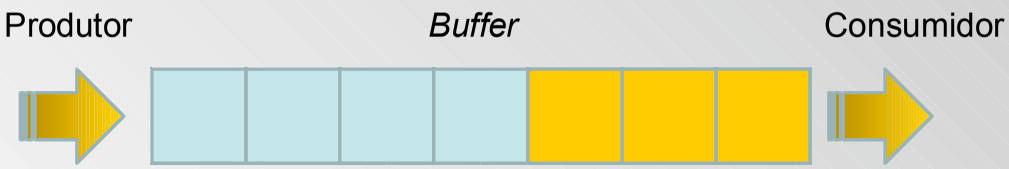
Centro de Informática U·F·P·E

Exclusão Mútua com Espera Ocupada (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 G. L. Peterson para exclusão mútua

Problema do Produtor-Consumidor



- Não é apenas uma questão de exclusão mútua!
- se consumo > produção
  - Buffer esvazia; Consumidor não tem o que consumir
- se consumo < produção
  - Buffer enche; Produtor não consegue produzir mais

Centro de Informática U·F·P·E

```
#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 */
      }
}
```

```
#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 */
      }
}
```

Semáforo

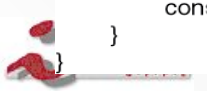
- Variável que tem como função o controle de acesso a recursos compartilhados
- Indica quantos processos (ou *threads*) podem ter acesso a um recurso compartilhado
  - Para se ter **exclusão mútua**, só um processo executa por vez
    - Para isso utiliza-se um semáforo binário, com inicialização em 1
    - Esse semáforo binário atua como um **mutex**

Semáforo (cont.)

- Principais operações sobre semáforos:
  - **Inicialização:** recebe um valor inteiro indicando quantos processos podem acessar um determinado recurso
  - ***wait*** ou ***down*** ou ***P***: decrementa o valor do semáforo. Se o semáforo está com valor 0, o processo é posto para dormir.
  - ***signal*** ou ***up*** ou ***V***: se o semáforo estiver com o valor 0 e existir algum processo adormecido, um processo será acordado.
    - Caso contrário, o valor do semáforo é incrementado.
- As operações de incrementar e decrementar devem ser operações **atômicas** ou **indivisíveis**
  - Apenas um processo por vez executando essas operações em cada semáforo

```
#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 */
      }
}
```

```
#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 */
      }
}
```



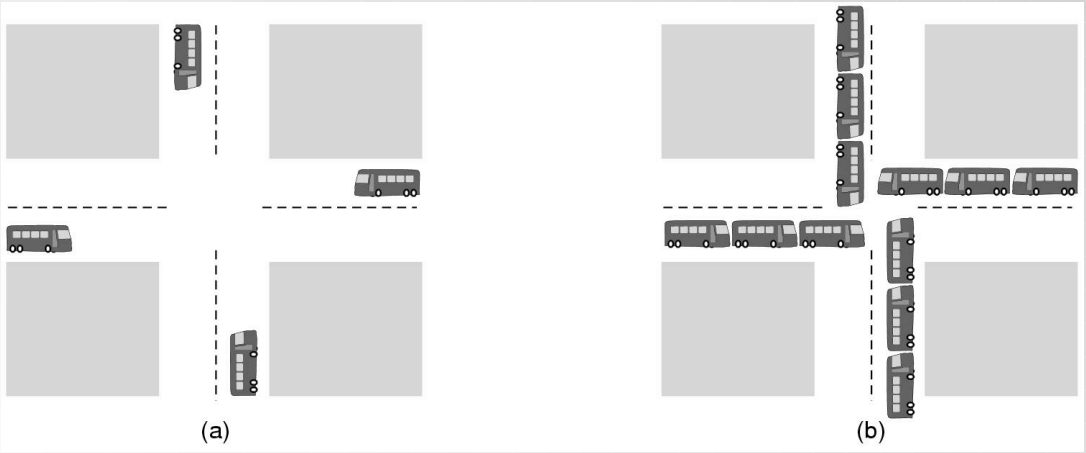
```
#define N 100                                   /* número de lugares no buffer */
typedef int semaphore;                          /* semáforos são um tipo especial de int */
semaphore mutex = 1;                            /* controla o acesso à região crítica */
semaphore empty = N;             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

void producer(void)
{
     int item;

     while (TRUE) {                             /*
          item = produce_item( );               /*
          down(&empty);                         /*
          down(&mutex);                         /*
          insert_item(item);                    /*
          up(&mutex);                           /*
          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 */
     }
}
```

Os dois semáforos, empty e full, são realmente necessários? Ou apenas um já resolveria?

```
#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) {
            item = produce_item( );
            down(&empty);
            down(&mutex);
            insert_item(item);
            up(&mutex);
            up(&full);
      }
}
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 */
      }
}
```

Se invertéssemos a ordem das instruções down(&empty) e down(&mutex), esta solução ainda funcionaria?

Exemplo da necessidade de Semáforos



(a) Um *deadlock* potencial. (b) Um *deadlock* real.

Monitores

- Coleção de **rotinas**, **variáveis** e **estruturas de dados**
- Só um processo ativo no monitor em qualquer momento
- Dependem fortemente de suporte linguístico
- Java e C# implementam
- C e C++ não
- Usam *mutexes* e semáforos internamente

```
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;
```

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

Centro de Informática

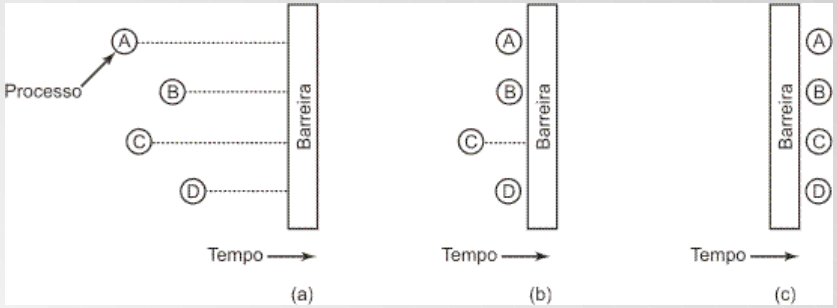
Troca de Mensagens

```
#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

Centro de Informática U·F·P·E

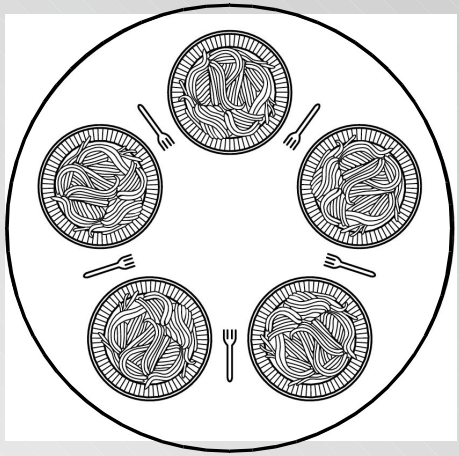
Barreiras: Sincronização



- Uso de barreira
  - processos se aproximando de uma barreira
  - todos os processos, exceto um, bloqueados pela barreira
  - 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 */
#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 (3)

```
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)