Sistemas Operacionais

Aula 02 – Processos e Threads

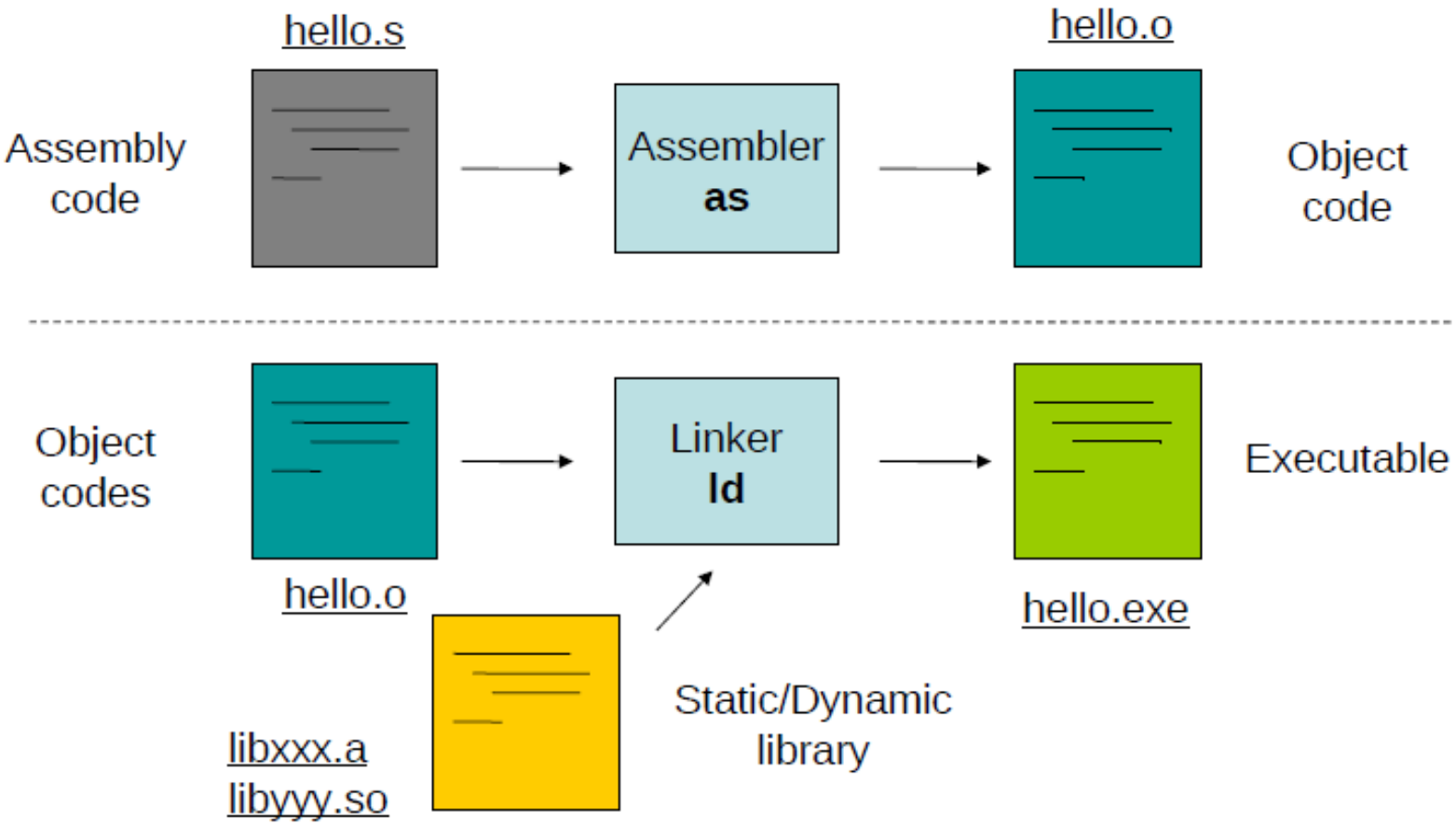
Edirlei Soares de Lima
<edirlei@iprj.uerj.br>

Programas e Processos

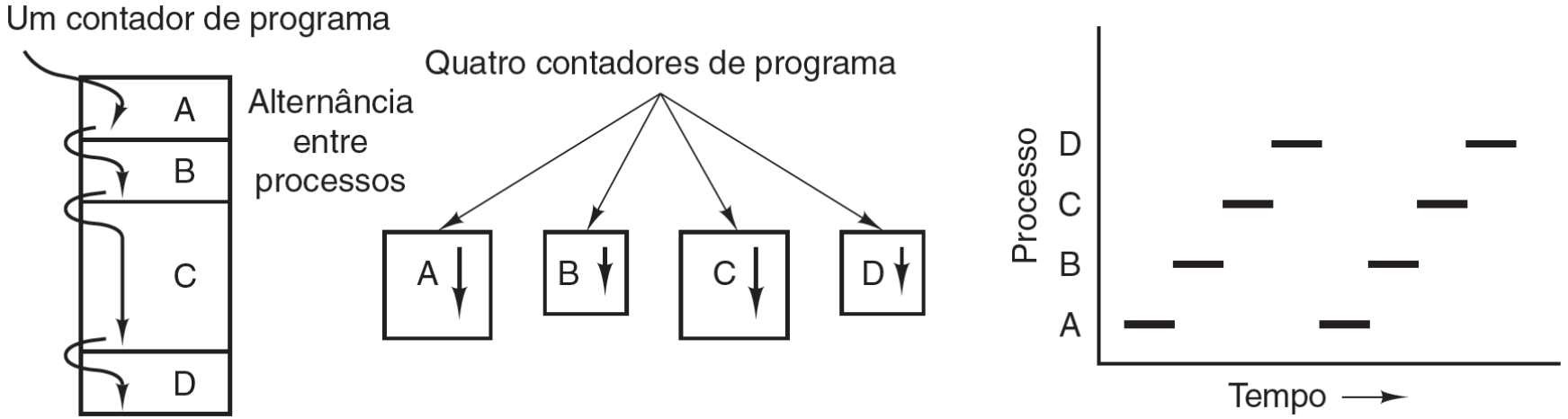
- **Programa:**
  - Entidade estática e permanente;
  - Composto por sequencias de instruções;

- **Processo:**
  - É a execução de uma instância de um programa (mais de um processo pode executar o código do mesmo programa).
  - Entidade dinâmica que alterna o seu estado a medida que avança na sua execução;
  - Composto por: programa, dados e contexto;

Construção de um Programa



Modelo de Processo



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

Criação de Processos

- Sistemas operacionais precisam de mecanismos para criar processos.

- Principais eventos que levam à criação de processos:
  - Início do sistema (processos de background e foreground);
  - Execução de chamada ao sistema de criação de processos;
  - Solicitação do usuário para criar um novo processo;
  - Início de um job em lote;

Criação de Processos

- Todo novo processo (filho) é criado por um processo existente (pai).
  - UNIX: fork e execve;
  - Windows: CreateProcess;

Gerenciamento de processos

Chamada	Descrição
pid = fork( )	Cria um processo filho idêntico ao pai
pid = waitpid(pid, &statloc, options)	Espera que um processo filho seja concluído
s = execve(name, argv, environp)	Substitui a imagem do núcleo de um processo
exit(status)	Conclui a execução do processo e devolve status

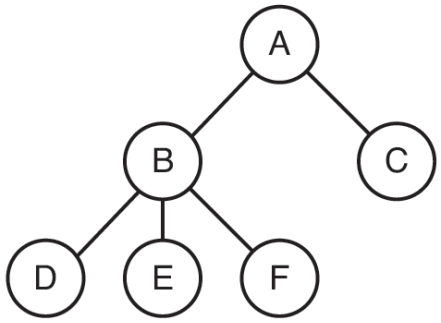
Termino de Processos

- Condições que levam ao término de processos:
  - Saída normal (voluntária);
  - Saída por erro (voluntária);
  - Erro fatal (involuntário);
  - Cancelamento por um outro processo (involuntário);

- Em alguns sistemas, quando um processo termina, todos os processos criados por ele são imediatamente cancelados.
  - UNIX e Windows não funcionam dessa maneira.

Hierarquias de Processos

- Um processo pai cria um processo filho e o processo filho pode criar seu próprio processo, formando uma hierarquia.
  - UNIX chama isso de "grupo de processos“.
  - Windows não possui o conceito de hierarquia de processos. Todos os processos são criados iguais.
    - Processo pai tem apenas acesso a um handle;

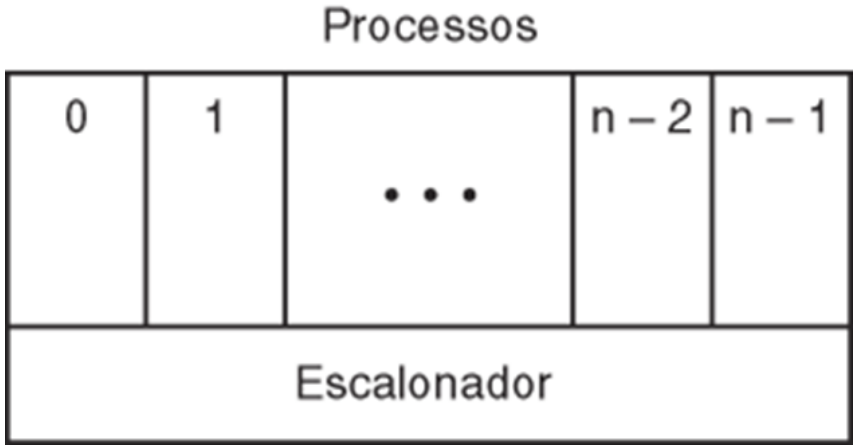


Estados de Processos

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

- **Em execução:** realmente usando a CPU naquele instante;
- **Pronto:** executável, porém temporariamente parado para dar lugar a outro processo na CPU;
- **Bloqueado:** incapaz de executar enquanto não ocorrer um evento externo;

Estados de Processos

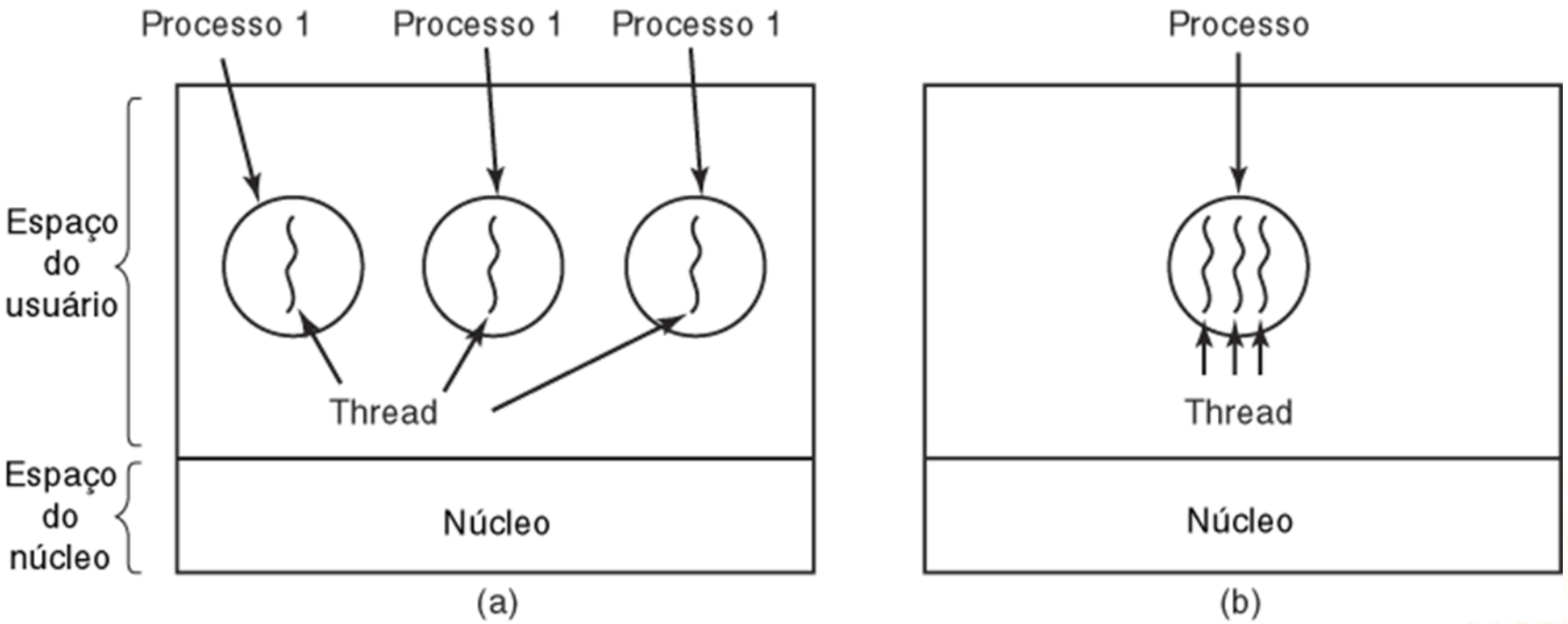


- O Escalonador encontra-se no nivel mais baixo do sistema operacional:
  - Oculta todos os detalhes sobre o processo de inicialização, escalonamento e bloqueio de processos;

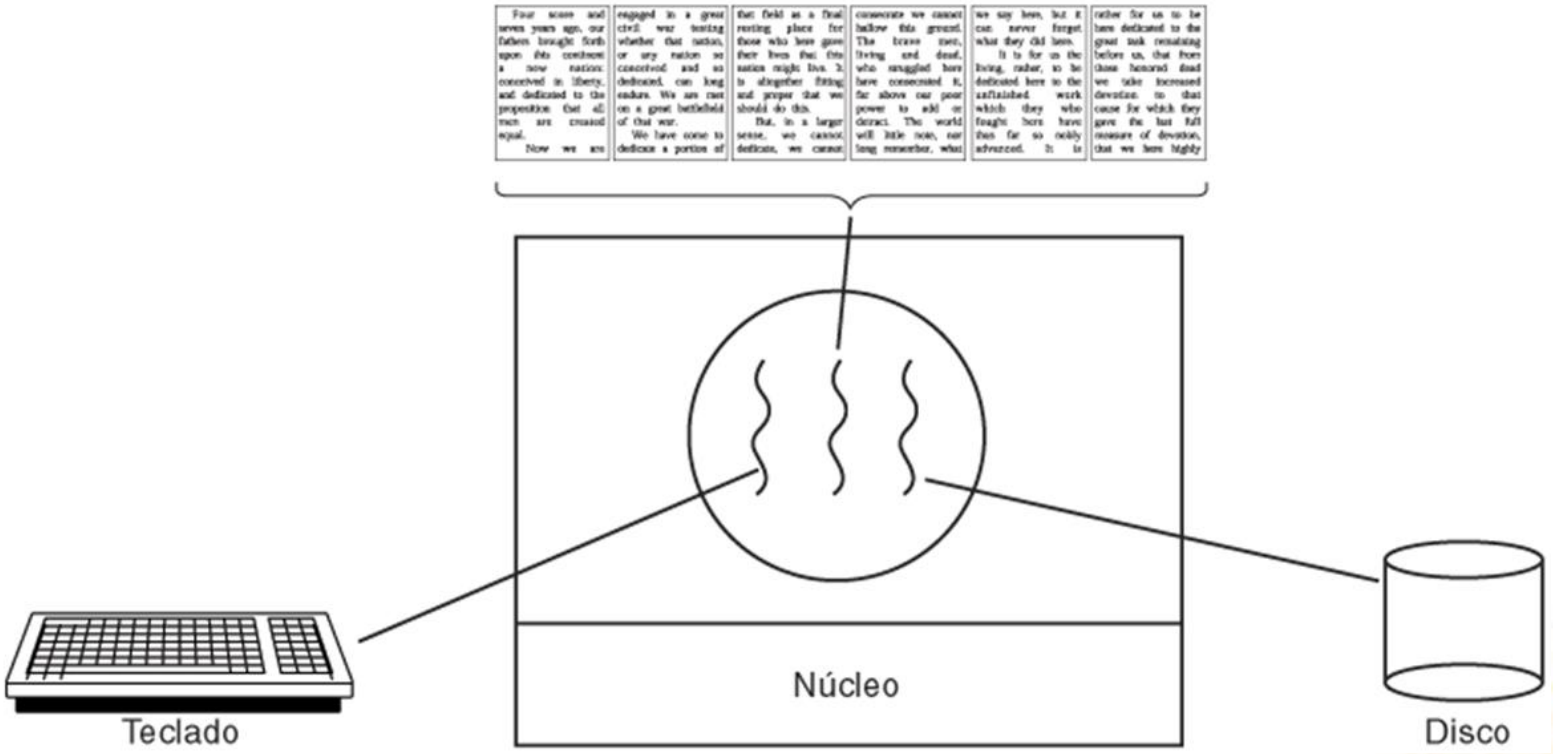
Threads

- Uma thread é uma entidade de execução dentro de um processo.
- O processo que conhecemos é um processo com uma única thread (single-threaded).
  - Existe apenas uma linha de execução no processo.
- Porém, é possível ter mais de uma linha de execução em um processo.
  - Este tipo de processo é chamado processo com múltiplas threads (multi-threaded process).

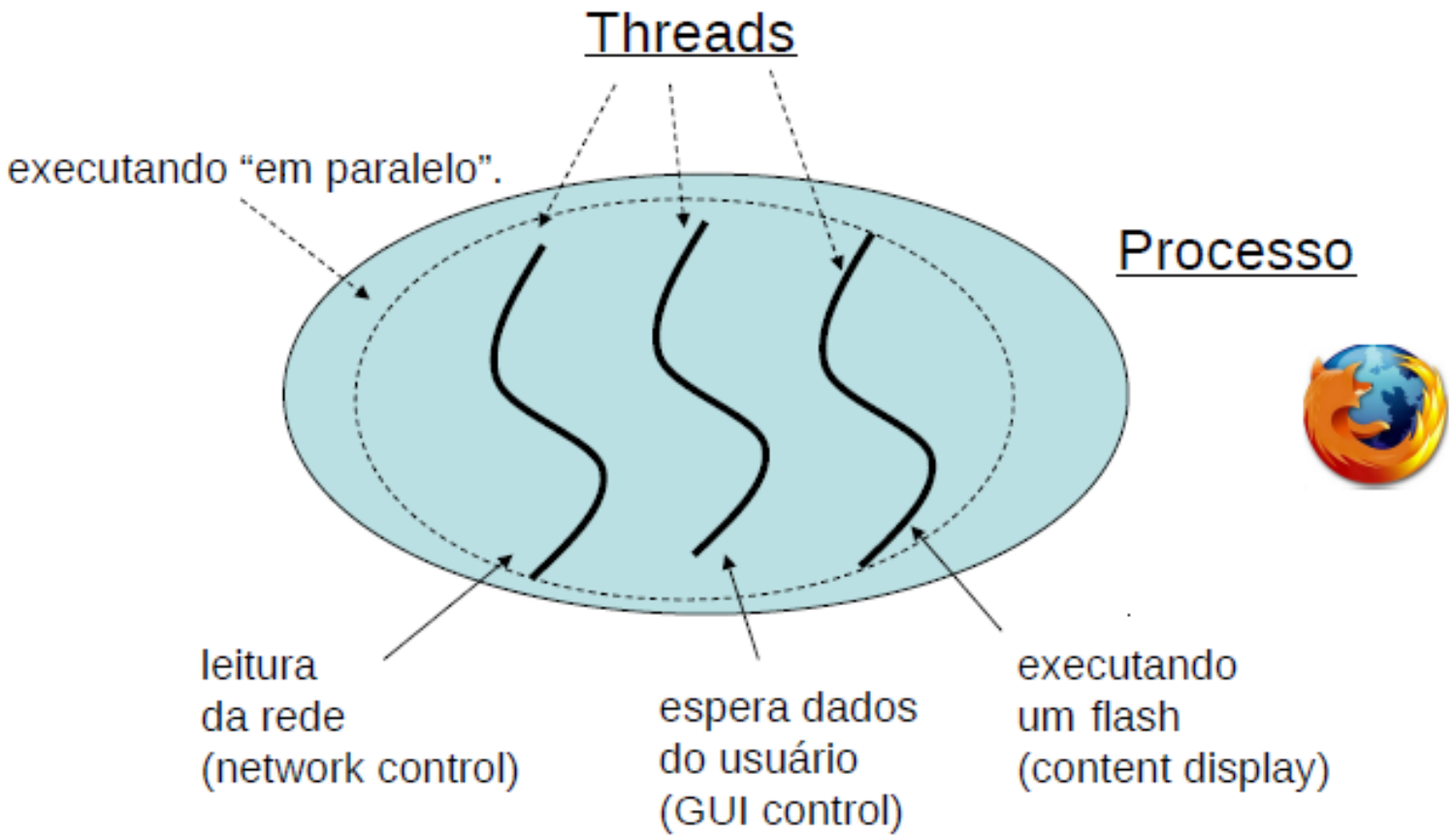
Threads


Processo 1
Processo 1
Processo 1
Espaço do usuário
Thread
Espaço do núcleo
Núcleo
(a)
Processo
Thread
Núcleo
(b)

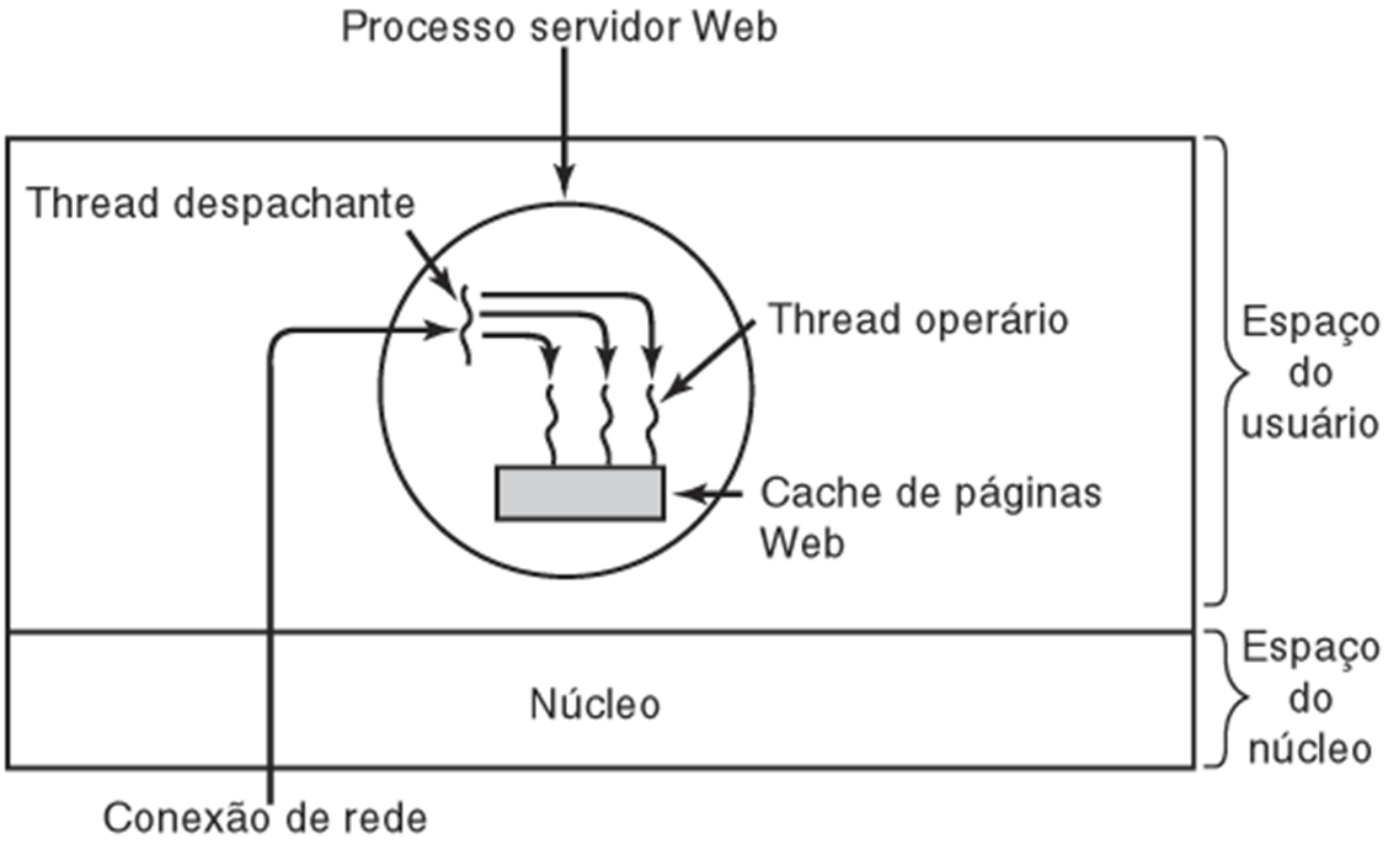
Uso de Threads – Editor de Texto



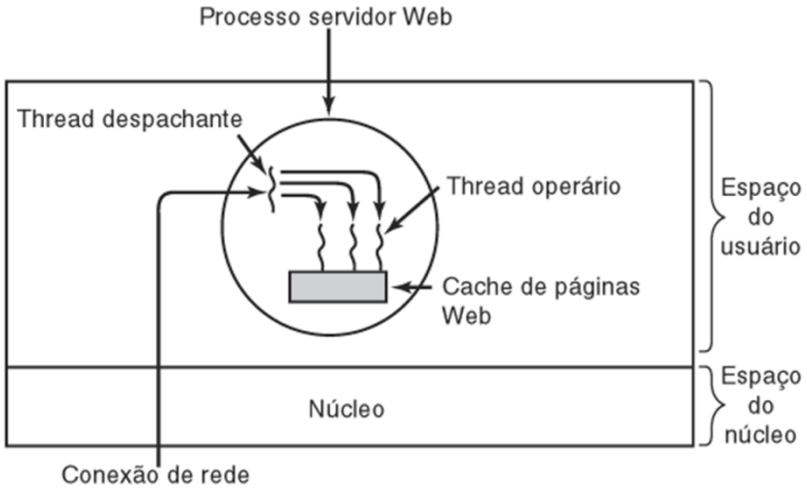
Uso de Threads – Navegador Web



Uso de Threads – Servidor Web



Uso de Threads – Servidor Web



Thread Despachante:

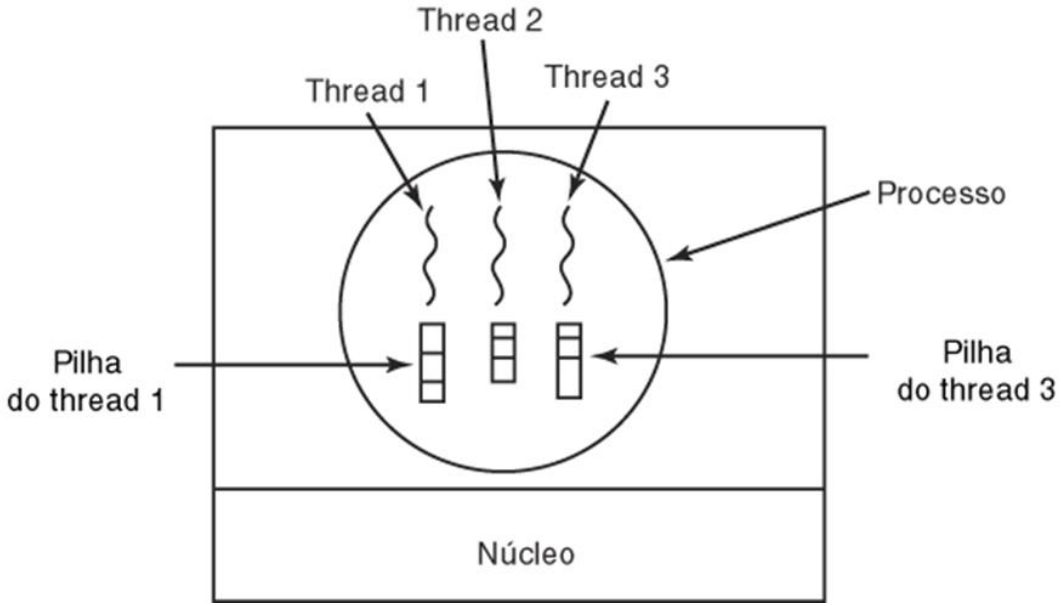
```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

Thread Operário:

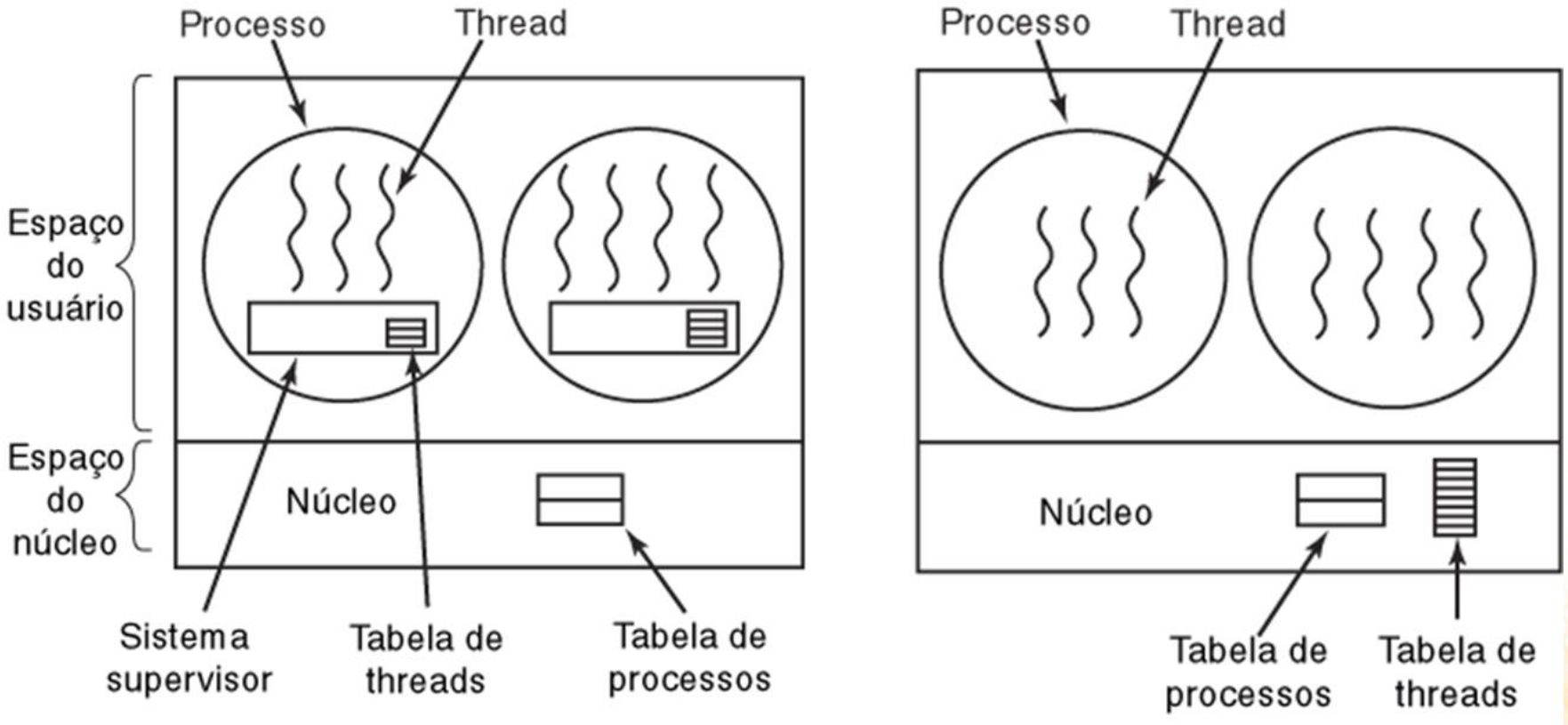
```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if(page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

Threads

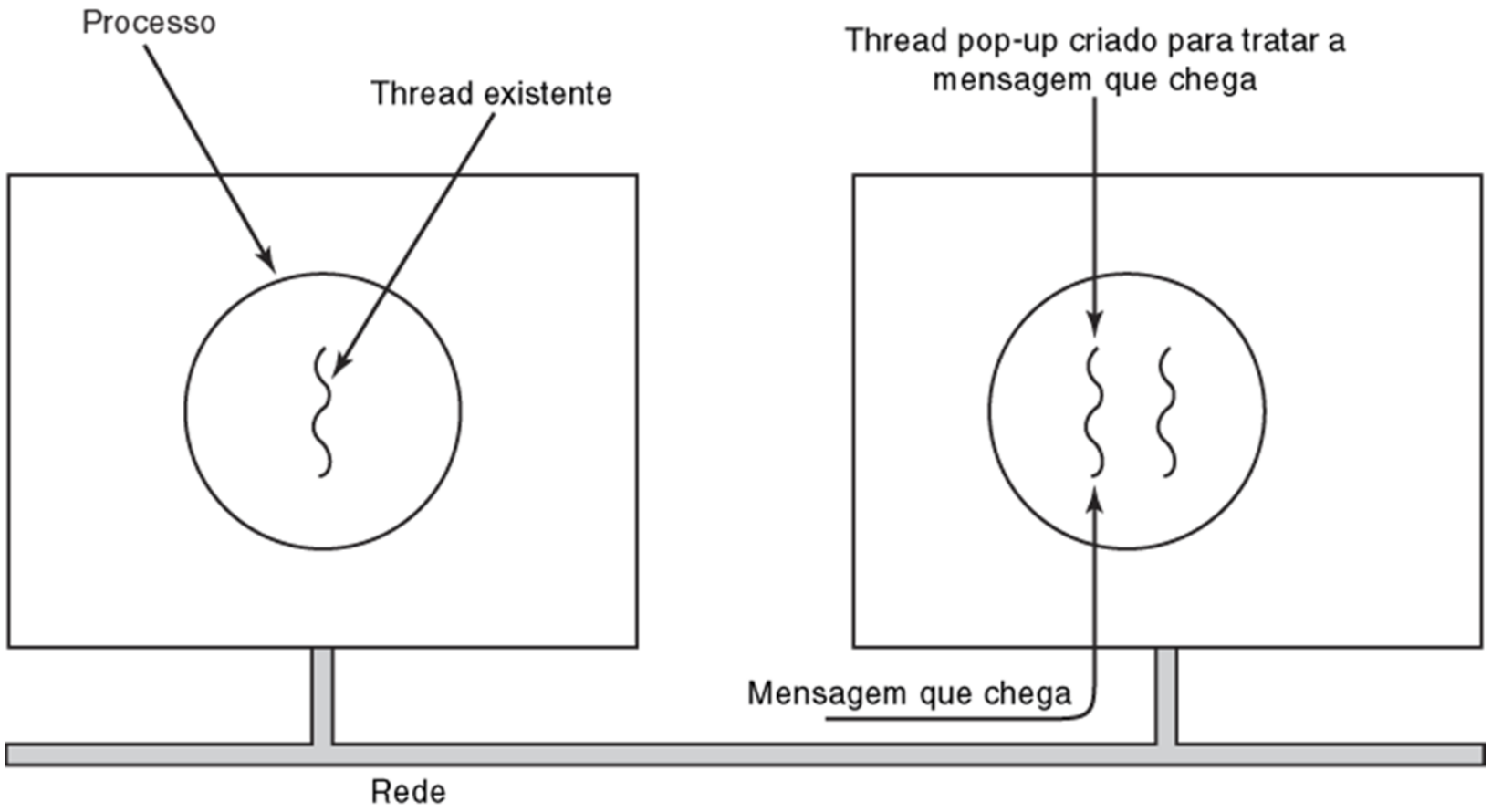
- Todas as threads de um processo compartilham o mesmo código e o mesmo conjunto de variáveis globais.
  - Cada thread tem seu próprio estado, pilha de memória, etc.



Threads de Usuário e de Núcleo



Threads Pop-Up



Implementando Threads (Windows)

- Criando Threads:

```
HANDLE WINAPI CreateThread(
  _In_opt_  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  _In_      SIZE_T                 dwStackSize,
  _In_      LPTHREAD_START_ROUTINE lpStartAddress,
  _In_opt_  LPVOID                 lpParameter,
  _In_      DWORD                  dwCreationFlags,
  _Out_opt_ LPDWORD                lpThreadId
);
```

- Exemplo:

```
HANDLE thread = CreateThread(NULL, 0, ThreadFunc,
                             &data_thread, 0, NULL);
```

https://msdn.microsoft.com/en-us/library/windows/desktop/ms682453%28v=vs.85%29.aspx

Implementando Threads (Windows)

```
#include <windows.h>

DWORD WINAPI ThreadFunc(void* data)
{
  //processo da Thread
  return 0;
}

int main(void)
{
  HANDLE thread = CreateThread(NULL, 0, ThreadFunc, NULL,
                               0, NULL);
  WaitForSingleObject(thread, INFINITE);

  CloseHandle(thread);

  return 0;
}
```

Implementando Threads: Corrida de Threads

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

DWORD WINAPI ThreadFunc(void* data)
{
  int i;
  int *info = (int*)data;
  for (i = 1; i < 30; i++)
  {
    printf("Thread %d - Volta %d!\n", *info, i);
    Sleep(10);
  }
  printf("Thread %d Chegou!!!\n", *info);
  return 0;
}
```

Implementando Threads: Corrida de Threads

```
int main()
{
  HANDLE threads[4];
  int i, data_thread[] = {1, 2, 3, 4};
  for (i = 0; i < 4; i++)
  {
    threads[i] = CreateThread(NULL, 0, ThreadFunc,
                              &data_thread[i], 0, NULL);
  }
  WaitForMultipleObjects(4, threads, TRUE, INFINITE);

  for (i = 0; i < 4; i++)
  {
    CloseHandle(threads[i]);
  }
  return 0;
}
```

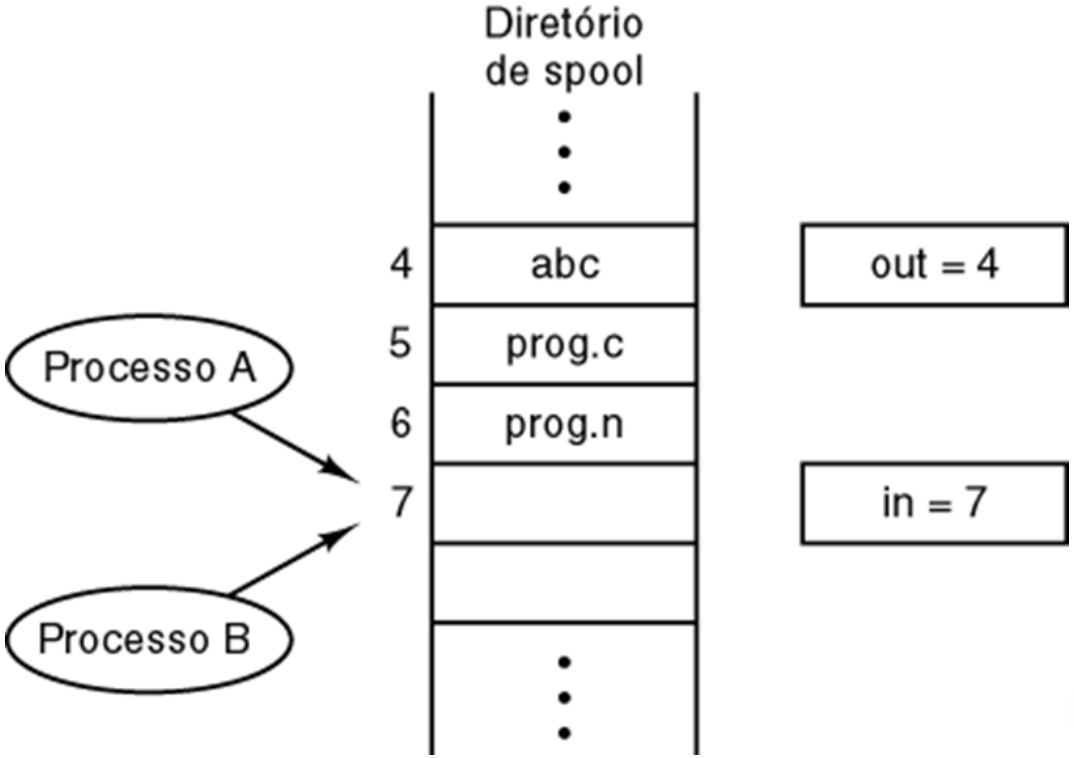
Exercícios

Lista de Exercícios 02

http://www.inf.puc-rio.br/~elima/so/

Comunicação entre Processos

- **Condições de Corrida:** dois processos querem ter acesso simultaneamente ao mesmo recurso compartilhado.

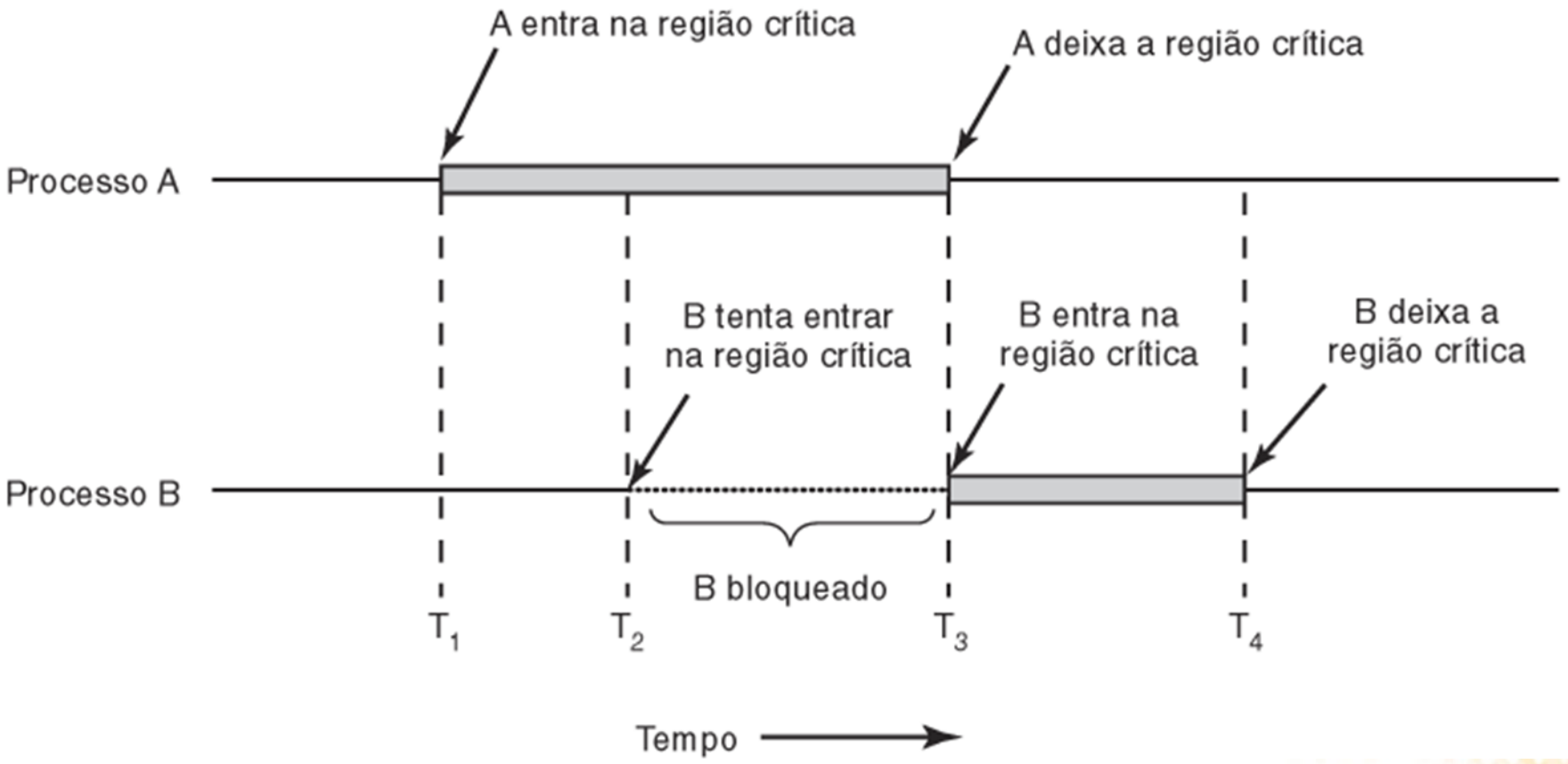


Regiões Críticas

- **Exclusão Mutua:** assegurar que outros processos seja impedidos de usar uma variável, estrutura ou arquivo que já estiver em uso por um processo.

- Condições necessárias:
  - Nunca dois processos podem estar simultaneamente em suas regiões críticas;
  - Nada pode ser afirmado sobre velocidade ou número 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



Exemplo – Condição de Corrida

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

int buffer[100];
int buffer_count = 0;

DWORD WINAPI ThreadFuncProdutor(void* data)
{
  while(TRUE)
  {
    if (buffer_count < 100)
    {
      buffer_count++;
      buffer[buffer_count] = 1;
      printf("Produzi [%d]: %d\n", buffer_count, 1);
    }
  }
  return 0;
}
```

```
DWORD WINAPI ThreadFuncConsumidor(void* data)
{
  int item;
  while(TRUE)
  {
    if (buffer_count > 0)
    {
      item = buffer[buffer_count];
      buffer[buffer_count] = 0;
      printf("Consumi [%d]: %d\n", buffer_count, item);
      buffer_count--;
    }
  }
  return 0;
}

int main()
{
  HANDLE thread_produtor;
  HANDLE thread_consumidor;
  int i;
  for (i = 0; i < 100; i++)
    buffer[i] = 0;
```

```
  thread_produtor = CreateThread(NULL, 0,
                    ThreadFuncProdutor, NULL, 0, NULL);
  thread_consumidor = CreateThread(NULL, 0,
                      ThreadFuncConsumidor, NULL, 0, NULL);

  WaitForSingleObject(thread_produtor, INFINITE);
  WaitForSingleObject(thread_consumidor, INFINITE);

  CloseHandle(thread_produtor);
  CloseHandle(thread_consumidor);

  return 0;
}
```

Exclusão Mútua com Espera Ociosa

- **Solução ingênua 1:** variáveis do tipo trava.
  - Uma única variável compartilhada (trava);
    - Inicialmente, trava = 0;
  - Para entrar em sua região critica, o processe verifica se trava == 0;
    - Se for 0, o processo altera o valor de trava para 1 e entra na região critica;
    - Caso contrario, aguarda até a variável se tornar 0;
  - Ao sair da região critica, o processo altera o valor de trava para 1.

- **Problema:** a condição de corrida continua ocorrendo com a variável trava!

Exclusão Mútua com Espera Ociosa

- **Solução ingênua 2:** chaveamento obrigatório.

```
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)

- **Problema:** se um processo for mais lento que o outro, ambos podem estar executando ao mesmo tempo.

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 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 */
}
```

Exclusão Mútua com Espera Ociosa

- **Solução usando a instrução TSL (hardware):**

```
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
```

Dormir e Acordar

- **Problema com as soluções anteriores:** espera ociosa.

- **Outra alternativa:** sleep e wakeup

- Problema do Produtor-Consumidor:
  - Dois processos compartilham um buffer comum e de tamanho N;
  - O produtor coloca informação dentro do buffer;
  - O consumidor remove informação do buffer;
  - Se o buffer estiver cheio, o produtor deve dormir;
  - Se o buffer estiver vazio, o consumidor deve dormir;

```
#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 */
     }
}
```

Dormir e Acordar: Produtor-Consumidor

- **Problema:**
  - Buffer vazio e o consumidor acabou de ler o valor de *count* para verificar se o seu valor é 0;
  - O escalonador decide parar de executar o consumidor e começa a executar o produtor;
  - O produtor insere um item no buffer, incrementa *count* e percebe que seu valor é 1, então o produtor manda o consumidor acordar;
  - O consumidor não está dormindo e o sinal de acordar é perdido;
  - O consumidor continua a sua execução, usando o valor de *count* lido anteriormente (0), e vai dormir;
  - Após preencher o buffer, o produtor também vai dormir.

Semáforos

- Método de sincronização de processos criado por Dijkstra (1965);

- Características:
  - Variável inteira;
  - Não negativa.

- Manipulados exclusivamente por duas operações atômicas:
  - DOWN (generalização do sleep);
  - UP (generalização do wakeup);
  - Implementados como chamada ao sistema.

```
#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 */
     }
}
```

Implementando Semáforos (Windows)

- Criando Semáforos:

```
HANDLE WINAPI CreateSemaphore(
  _In_opt_ LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
  _In_     LONG                  lInitialCount,
  _In_     LONG                  lMaximumCount,
  _In_opt_ LPCTSTR               lpName
);
```

- Exemplo:

```
HANDLE ghSemaphore = CreateSemaphore(NULL, 1, 1,
                                     L"MeuSemaforo");
```

https://msdn.microsoft.com/en-us/library/windows/desktop/ms682438%28v=vs.85%29.aspx

Implementando Semáforos (Windows)

- Aguardando Liberação de Semáforos:

```
DWORD WINAPI WaitForSingleObject(
  _In_ HANDLE hHandle,
  _In_ DWORD  dwMilliseconds
);
```

- Exemplo:

```
WaitForSingleObject(ghSemaphore , INFINITE);
```

https://msdn.microsoft.com/en-us/library/windows/desktop/ms687032%28v=vs.85%29.aspx

Implementando Semáforos (Windows)

- Liberado Semáforos:

```
BOOL WINAPI ReleaseSemaphore(
  _In_      HANDLE hSemaphore,
  _In_      LONG   lReleaseCount,
  _Out_opt_ LPLONG lpPreviousCount
);
```

- Exemplo:

```
ReleaseSemaphore(ghSemaphore, 1, NULL);
```

https://msdn.microsoft.com/en-us/library/windows/desktop/ms685071%28v=vs.85%29.aspx

Implementando Semáforos (Windows)

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

int buffer[100];
int buffer_count = 0;
HANDLE ghSemaphore;

DWORD WINAPI ThreadFuncProdutor(void* data){
  while(TRUE){
    Sleep(8);
    WaitForSingleObject(ghSemaphore, INFINITE);
    if (buffer_count < 100){
      buffer_count++;
      buffer[buffer_count] = 1;
      printf("Produzi [%d]: %d\n", buffer_count, 1);
    }
    ReleaseSemaphore(ghSemaphore, 1, NULL);
  }
  return 0;
}
```

```
DWORD WINAPI ThreadFuncConsumidor(void* data){
  int item;
  while(TRUE){
    WaitForSingleObject(ghSemaphore, INFINITE);
    if (buffer_count > 0){
      item = buffer[buffer_count];
      buffer[buffer_count] = 0;
      printf("Consumi [%d]: %d\n", buffer_count, item);
      buffer_count--;
    }
    ReleaseSemaphore(ghSemaphore, 1, NULL);
    Sleep(10);
  }
  return 0;
}

int main(){
  HANDLE thread_produtor;
  HANDLE thread_consumidor;
  int i;

  ghSemaphore = CreateSemaphore(NULL, 1, 1, L"MeuSemaforo");
```

```
  for (i = 0; i < 100; i++)
    buffer[i] = 0;

  thread_produtor = CreateThread(NULL, 0,
                    ThreadFuncProdutor, NULL, 0, NULL);
  thread_consumidor = CreateThread(NULL, 0,
                      ThreadFuncConsumidor, NULL, 0, NULL);

  WaitForSingleObject(thread_produtor, INFINITE);
  WaitForSingleObject(thread_consumidor, INFINITE);

  CloseHandle(thread_produtor);
  CloseHandle(thread_consumidor);

  CloseHandle(ghSemaphore);

  return 0;
}
```

Mutex

- Mutex (Mutual Exclusion)

- Implementação de mutex_lock e mutex_unlock:

```
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
```

Implementando Mutex (Windows)

- Criando Mutex:

```
HANDLE WINAPI CreateMutex(
  _In_opt_ LPSECURITY_ATTRIBUTES lpMutexAttributes,
  _In_     BOOL                  bInitialOwner,
  _In_opt_ LPCTSTR               lpName
);
```

- Exemplo:

```
ghMutex = CreateMutex(NULL, FALSE, NULL);
```

https://msdn.microsoft.com/en-us/library/windows/desktop/ms682411%28v=vs.85%29.aspx

Implementando Semáforos (Windows)

- Aguardando Liberação de Mutex:

```
DWORD WINAPI WaitForSingleObject(
  _In_ HANDLE hHandle,
  _In_ DWORD  dwMilliseconds
);
```

- Exemplo:

```
WaitForSingleObject(ghMutex, INFINITE);
```

https://msdn.microsoft.com/en-us/library/windows/desktop/ms687032%28v=vs.85%29.aspx

Implementando Mutex (Windows)

- Liberado Mutex:

```
BOOL WINAPI ReleaseMutex(
  _In_ HANDLE hMutex
);
```

- Exemplo:

```
ReleaseMutex(ghMutex);
```

https://msdn.microsoft.com/en-us/library/windows/desktop/ms685066%28v=vs.85%29.aspx

Implementando Mutex (Windows)

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

int buffer[100];
int buffer_count = 0;
HANDLE ghMutex;

DWORD WINAPI ThreadFuncProdutor(void* data){
  while(TRUE){
    Sleep(8);
    WaitForSingleObject(ghMutex, INFINITE);
    if (buffer_count < 100){
      buffer_count++;
      buffer[buffer_count] = 1;
      printf("Produzi [%d]: %d\n", buffer_count, 1);
    }
    ReleaseMutex(ghMutex);
  }
  return 0;
}
```

```
DWORD WINAPI ThreadFuncConsumidor(void* data){
  int item;
  while(TRUE){
    WaitForSingleObject(ghMutex, INFINITE);
    if (buffer_count > 0){
      item = buffer[buffer_count];
      buffer[buffer_count] = 0;
      printf("Consumi [%d]: %d\n", buffer_count, item);
      buffer_count--;
    }
    ReleaseMutex(ghMutex);
    Sleep(10);
  }
  return 0;
}

int main()
{
  HANDLE thread_produtor;
  HANDLE thread_consumidor;
  int i;
  ghMutex = CreateMutex(NULL, FALSE, NULL);
```

```
  for (i = 0; i < 100; i++)
    buffer[i] = 0;

  thread_produtor = CreateThread(NULL, 0,
                    ThreadFuncProdutor, NULL, 0, NULL);
  thread_consumidor = CreateThread(NULL, 0,
                      ThreadFuncConsumidor, NULL, 0, NULL);

  WaitForSingleObject(thread_produtor, INFINITE);
  WaitForSingleObject(thread_consumidor, INFINITE);

  CloseHandle(thread_produtor);
  CloseHandle(thread_consumidor);

  CloseHandle(ghMutex);

  return 0;
}
```

Exercícios

Lista de Exercícios 03

http://www.inf.puc-rio.br/~elima/so/

Leitura Complementar

- Andrew S. Tanenbaum. **Sistemas Operacionais Modernos**, 3ª Edição, Pearson, 2010.

- **Capitulo 2: Processos e Threads**