BC1518 - Sistemas Operacionais

Aula 4: Processos

Tópicos desta aula

- Conceito de Processo
- Escalonamento de Processos
- Operações sobre Processos
- Processos Cooperativos
- Comunicação entre Processos

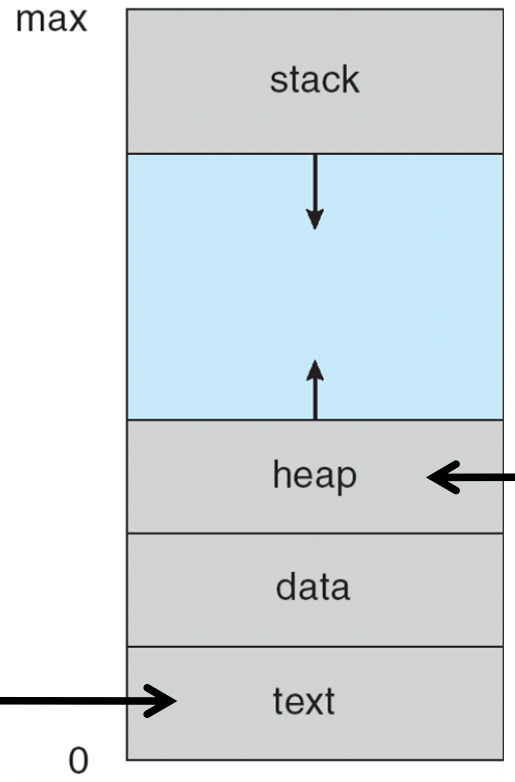
Conceito de Processo

- Um **sistema operacional** executa uma variedade de programas
  - Sistemas *Batch* – executam tarefas (*jobs*)
  - Sistemas de Tempo Compartilhado – executam programas de usuário ou tarefas
- O livro texto usa os termos **tarefa** e **processo** de forma intercambiável

- **Processo – um programa em execução**
  - A execução do processo precisa ocorrer de maneira sequencial

- Um **processo inclui**:
  - Contador de Programa – PC: indica a próxima instrução a ser executada
  - Pilha (*stack*): contém dados temporários, como as variáveis locais, os parâmetros de métodos e endereços de retorno
  - Seção de dados: contém as variáveis globais

Processo na memória



Um programa se torna um processo quando um arquivo executável é carregado na memória

A pilha de ***heap*** é a memória alocada dinamicamente durante o tempo de execução do processo

Código-fonte do programa

Estados do processo

➢Durante a execução, um processo pode estar em um dos estados:

- Novo (*new*): o processo está sendo criado
- Pronto (*ready*): o processo está pronto para ser atribuído ao processador
- Executando (*running*): as instruções estão sendo executadas
- Esperando (*waiting*): o processo está esperando algum evento (por exemplo, o término de uma operação de E/S)
- Terminado (*terminated*): o processo terminou a sua execução

Diagrama de estado do processo

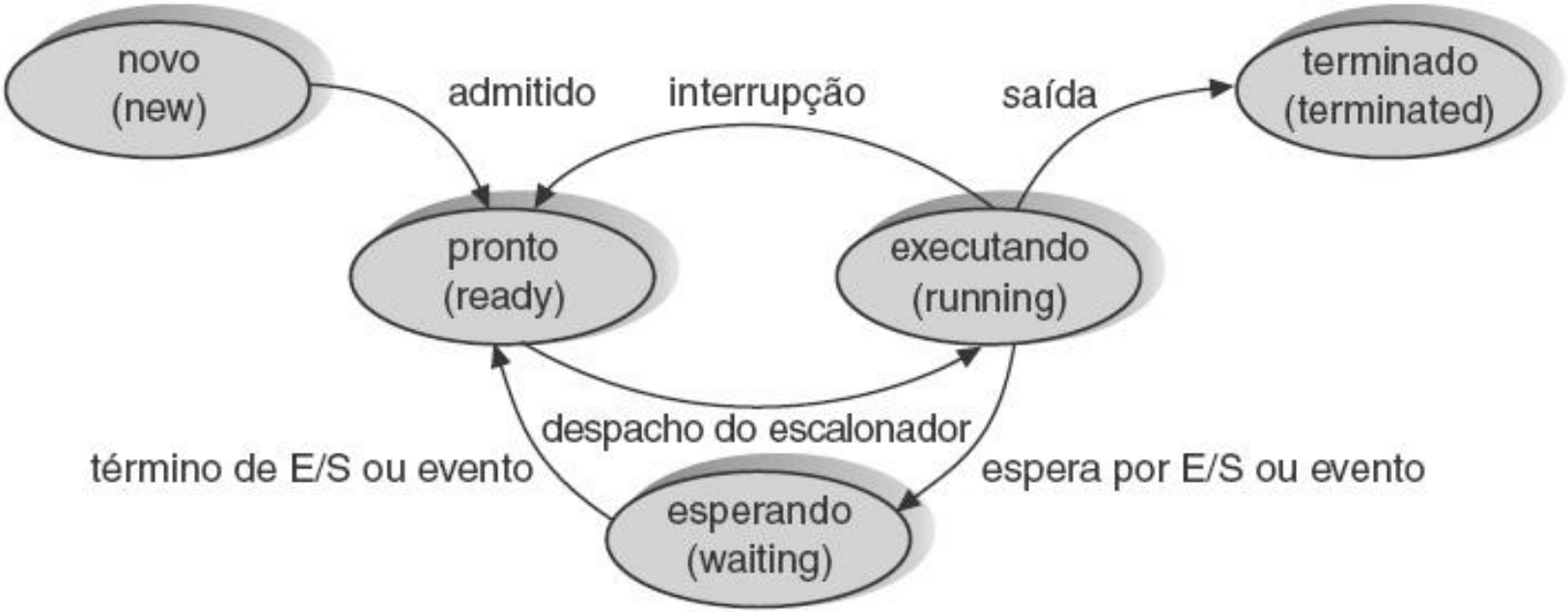


Diagrama de estado do processo [Silberschatz]

Somente um processo pode estar **executando** em um processador, embora muitos processos possam estar **prontos** ou **esperando**

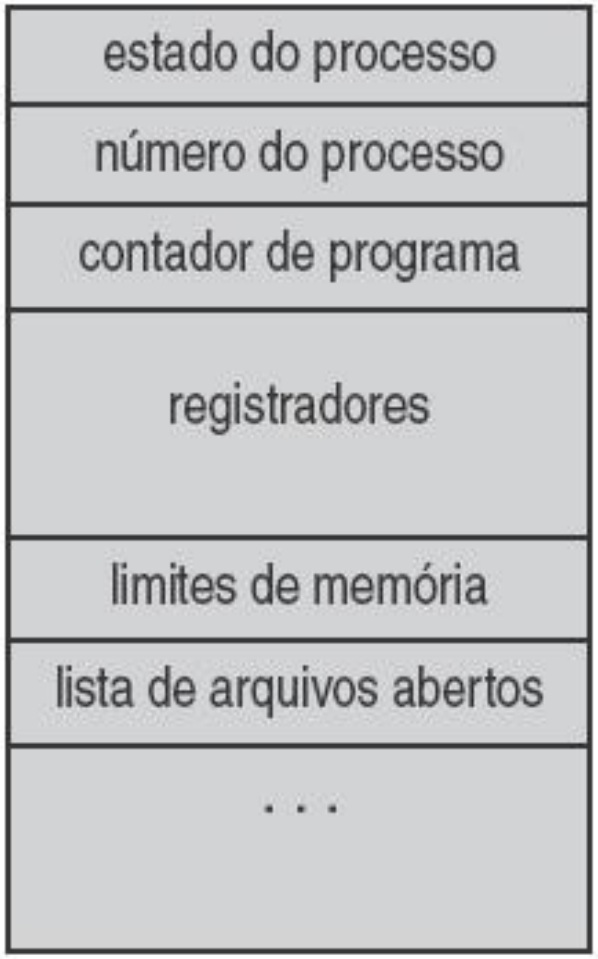
Bloco de controle de processo (*Process Control Block* - PCB)

Cada processo é representado no Sistema Operacional por um PCB (*Process Control Block*), ou bloco de controle de tarefa

O PCB contém Informações associadas a cada processo:

- Estado do processo (novo, pronto, executando, etc.)
- Contador de programa (indica o endereço da próxima instrução)
- Registradores de CPU (acumuladores, ponteiro de pilha, registradores de uso geral, informação de *status*)
- Informações de escalonamento de CPU (prioridade do processo, informações para escalonamento)
- Informações de gerenciamento de memória (dados como registradores de base e limite)
- Informações de contabilização (tempo de CPU, nº do processo, etc.)
- Informações de *status* de E/S (lista de dispositivos de E/S associados, lista de arquivos abertos, etc.)

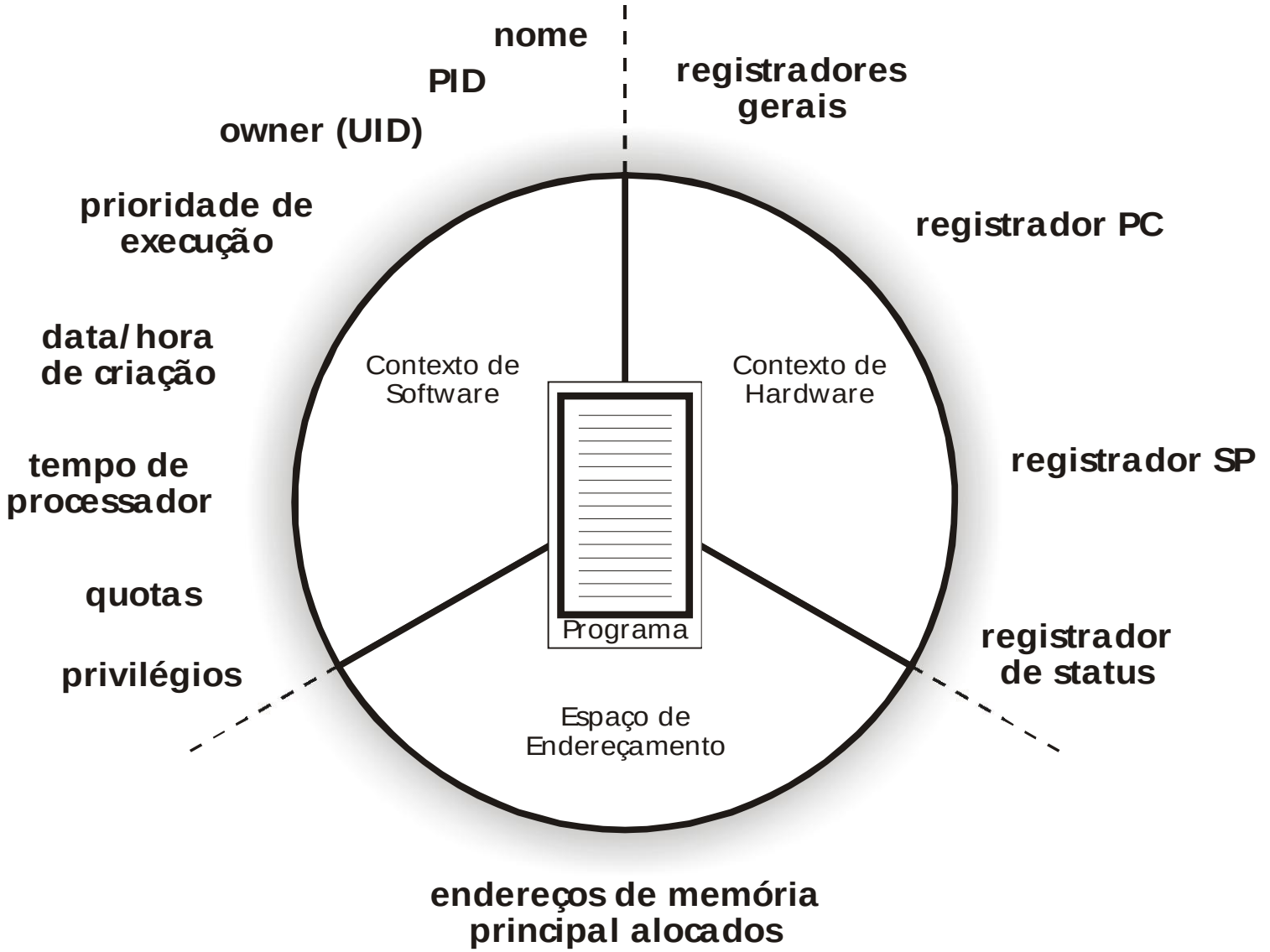
Bloco de controle de processo (PCB)



O PCB serve como um repositório de informações, e essas informações variam de um processo para outro.

Bloco de controle de processo (PCB) [Silberschatz]

Características da estrutura de um processo



Características da estrutura de um processo [Machado]

UFABC

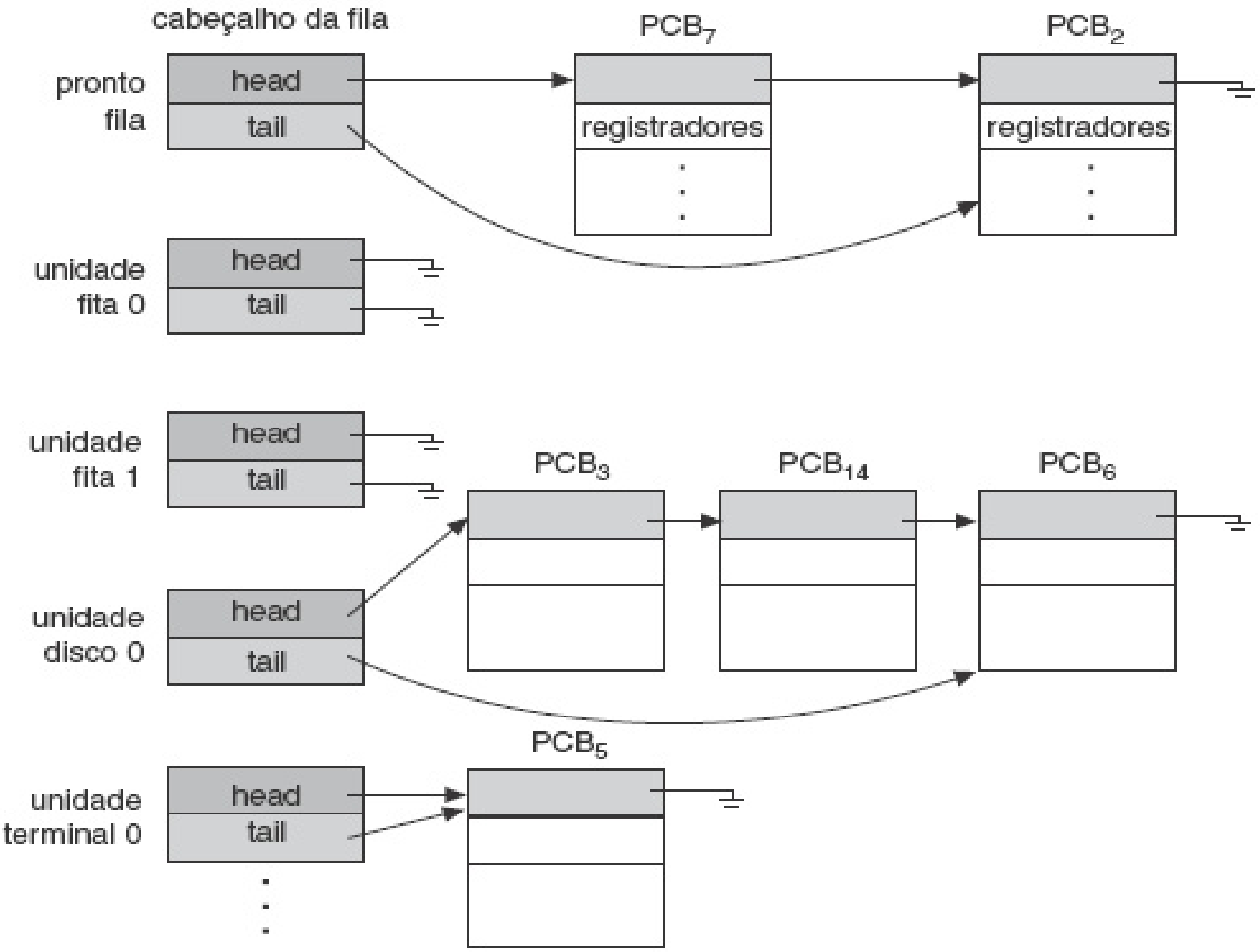
Escalonamento de processos

Fila de Escalonamento de Processos

➢Como um sistema com um único processador pode ter somente um processo executando, se existirem mais processos, eles devem aguardar em filas até que a CPU esteja livre

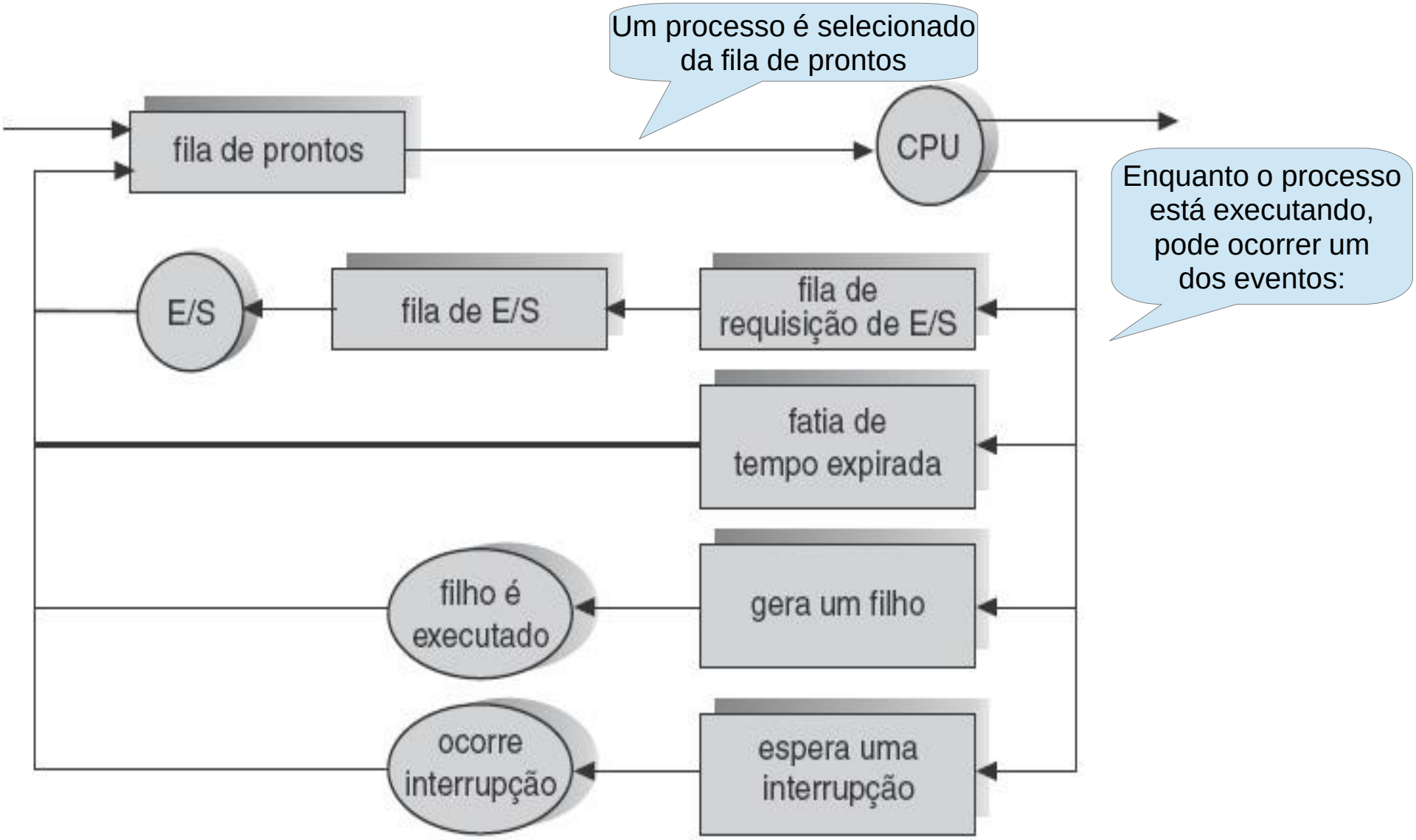
- Fila de tarefas (*Job Queue*): conjunto de todos os processos no sistema
- Fila de processos prontos (*Ready Queue*): conjunto de todos os processos residindo na memória principal que estão prontos e esperando para serem executados
- Fila de dispositivo (*Device Queue*): conjunto dos processos esperando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

Fila de processos prontos e várias filas de dispositivos de E/S



Fila de processos prontos e diversas filas de dispositivos de E/S [Silberschatz]

Representação de escalonamento de processos (Diagrama de Filas)



Representação do diagrama de fila do escalonamento de processos [Silberschatz]

Escalonadores

➢Com a multiprogramação, diversos processos disputam pelos recursos disponíveis no sistema, dentre eles, o tempo de CPU

➢Uma vez que pode existir mais de um processo pronto para execução, é necessário escolher um deles

➢O escalonamento de processos permite selecionar um processo dentre um conjunto de processos disponíveis para execução na CPU

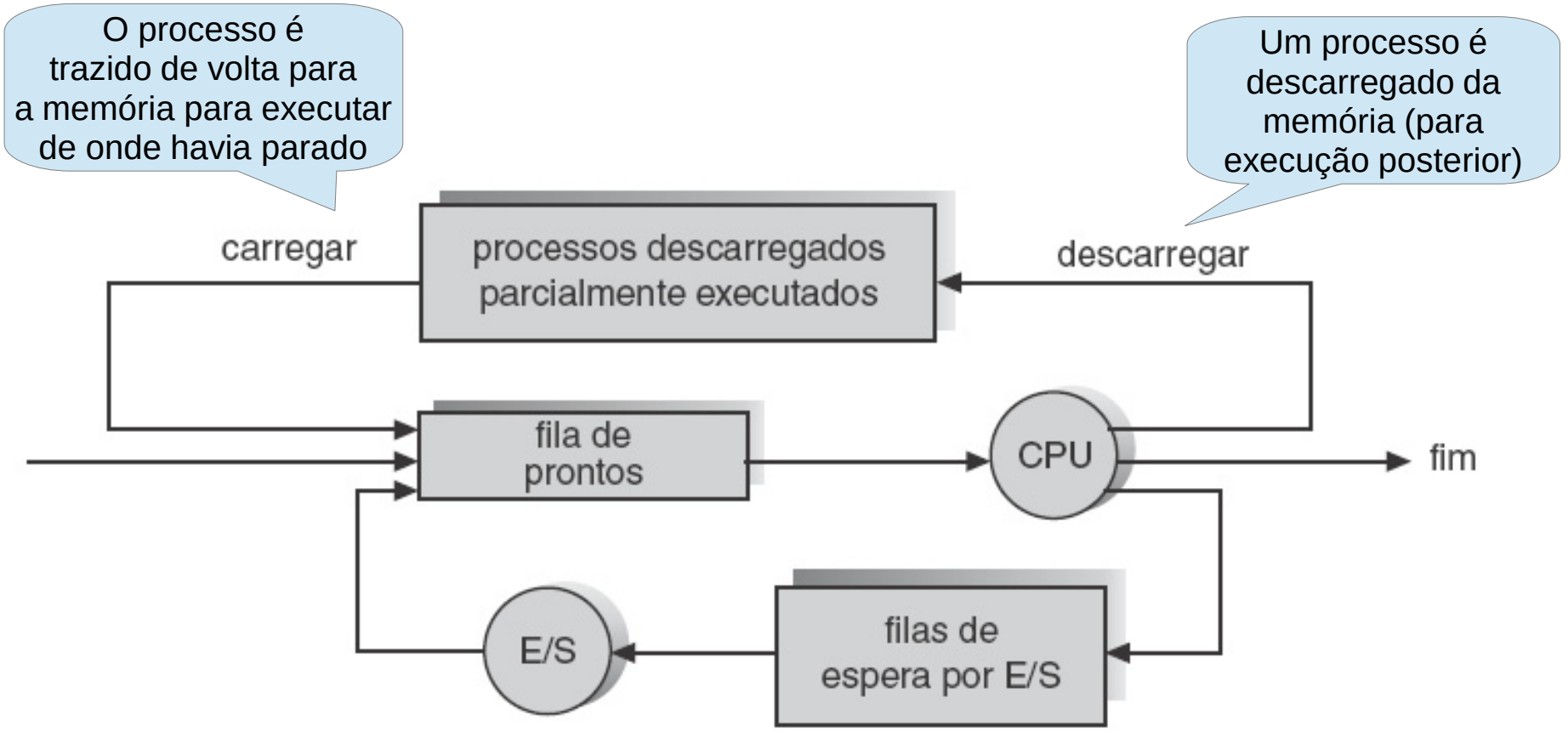
➢Para isso, há três tipos de escalonadores:

- Escalonador de longo prazo (ou escalonador de tarefas):
  - Decide quando um processo deve ser efetivamente criado (a criação de um processo pode ser adiada, caso a carga no computador estiver muito grande
  - Carrega um novo processo na memória
- Escalonador de curto prazo (ou escalonador de CPU): seleciona um dos processos da **fila de prontos** para execução e aloca a CPU para o mesmo

Escalonadores

- Escalonador de médio prazo: utilizado para liberar memória
  - Muitas vezes, pode ocorrer de o sistema ficar sobrecarregado e a quantidade de memória demandada pode exceder a memória disponível
  - Neste caso, este escalonador seleciona um ou mais processos na memória que estão disputando pela CPU e os "decarrega" da memória (*swap-out*). Com isso, o processo é temporariamente suspenso e seu contexto é salvo em disco
  - Após um tempo, os processos são trazidos de volta do disco para a memória (*swap-in*) e ficam novamente prontos para execução
  - Esse esquema é chamado troca (*swapping*) e enquanto um processo está suspenso, ele não ocupa memória e nem disputa pela CPU

Escalonamento de médio prazo ao diagrama de filas



Acréscimo do escalonador de médio prazo [Silberschatz]

Escalonadores (cont.)

➢O **escalonador de curto prazo** é o mais importante e é chamado **com alta frequência** (milissegundos)

- Precisa ser veloz para escolher um dos processos na fila de prontos

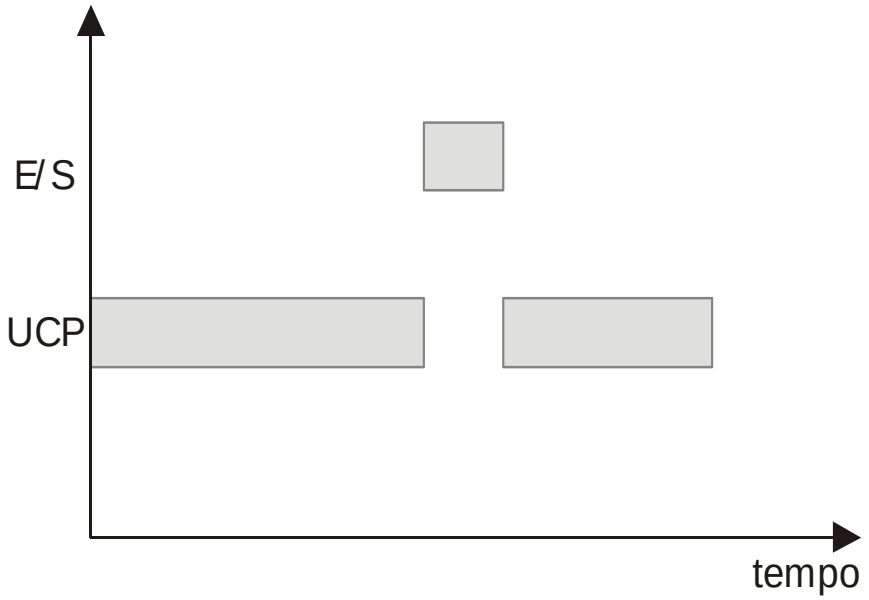
➢O **escalonador de médio prazo** é usado para retirada temporária de um processo da memória, possibilitando a liberação de espaço

➢**O escalonador de longo prazo** controla o **grau de multiprogramação** e é chamado **raramente** (segundos ou minutos)

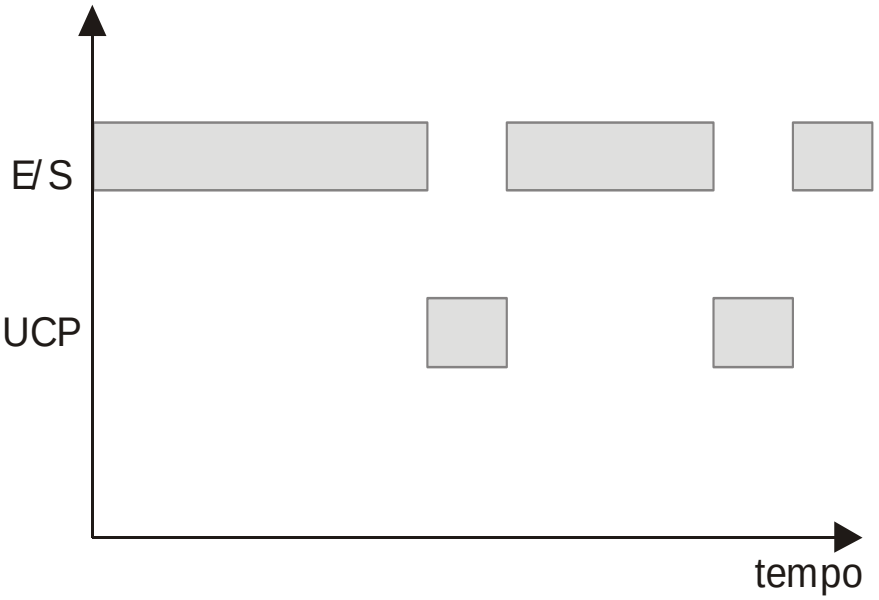
- quando a CPU está com pouco trabalho, ele traz outro processo para a fila

Processos IO-bound e CPU-bound

- Os processos podem ser descritos como:
  - Processo limitado por E/S (*IO-bound*): gasta mais tempo realizando operações de E/S do que cálculos => poucos processos na fila de prontos, menor utilização da CPU
  - Processo limitado por CPU (*CPU-bound*): gasta mais tempo realizando cálculos; gera pedidos de E/S com pouca frequência => utiliza pouco os dispositivos de E/S, mas muita CPU



(a) CPU-bound



(b) I/O-bound

[Silberschatz]

Processos IO-bound e CPU-bound (cont.)

➢Se houver muito mais processos *IO-bound* na memória do que processos *CPU-bound* => significa que há poucos processos na fila de prontos (a maioria está nas filas de espera de E/S); há uma menor utilização da CPU

➢Se, ao contrário, houver muito mais processos *CPU-bound* do que *IO-bound* na memória => a fila de prontos é grande, há pouca utilização dos dispositivos de E/S e muita utilização da CPU

➢Para um bom desempenho do sistema, é necessário uma boa combinação de processos *CPU-bound* e *IO-bound*

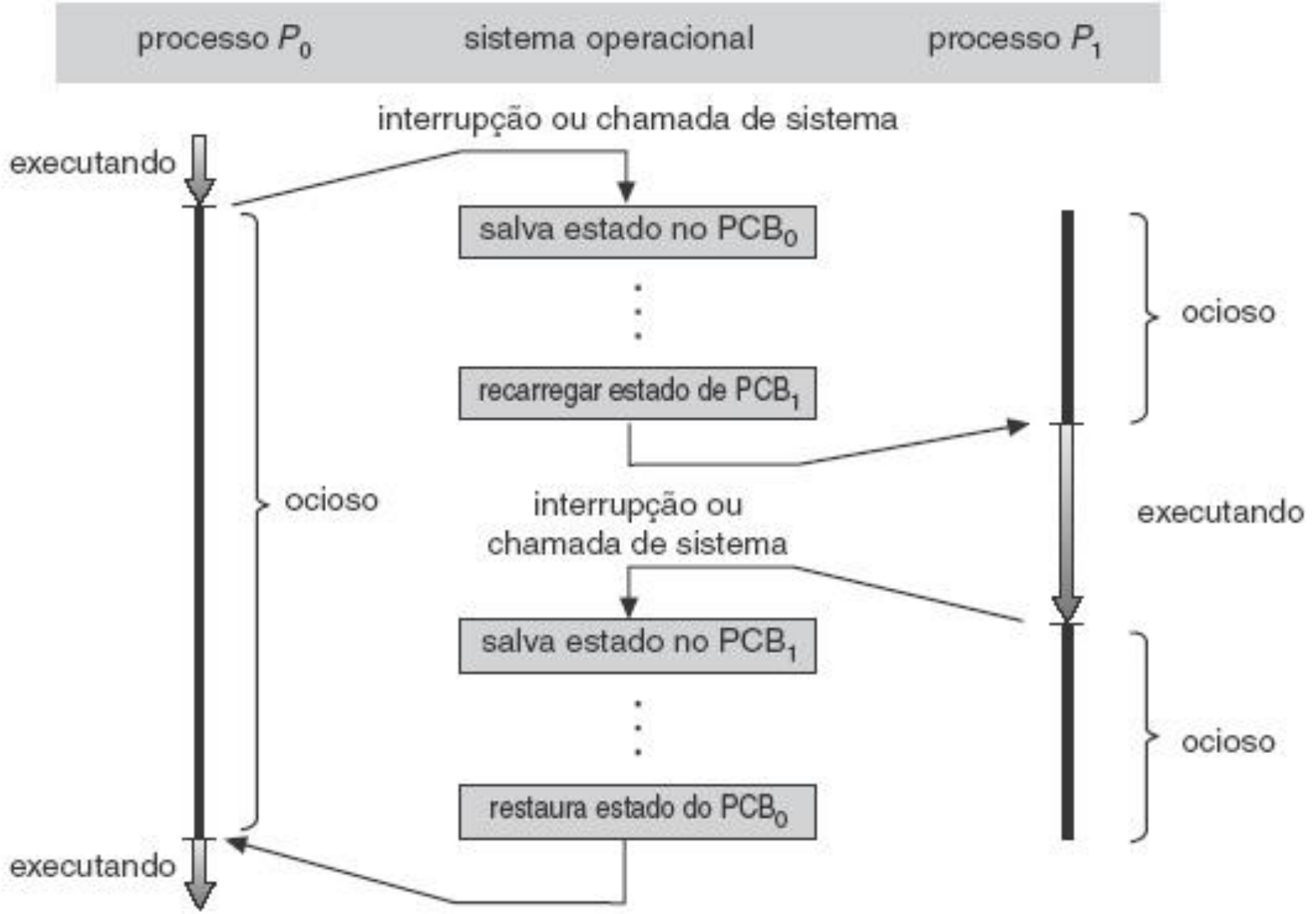
Troca de contexto

➢Quando a **CPU passa para outro processo**, o sistema precisa **salvar o estado do processo antigo** (informações do PCB) e **carregar o estado salvo do novo processo**

➢O tempo da **troca de contexto** é **custo adicional**; o sistema não realiza qualquer trabalho útil durante a troca

➢O **tempo de troca depende do suporte de *hardware*** (velocidade da memória, número de registradores a ser copiado e existência ou não de instruções especiais para carregar ou armazenar todos os registradores)

Troca da CPU de um processo para outro



Troca da CPU de um processo para outro [Silberschatz]

UFABC

Operações sobre processos

Criação de Processos

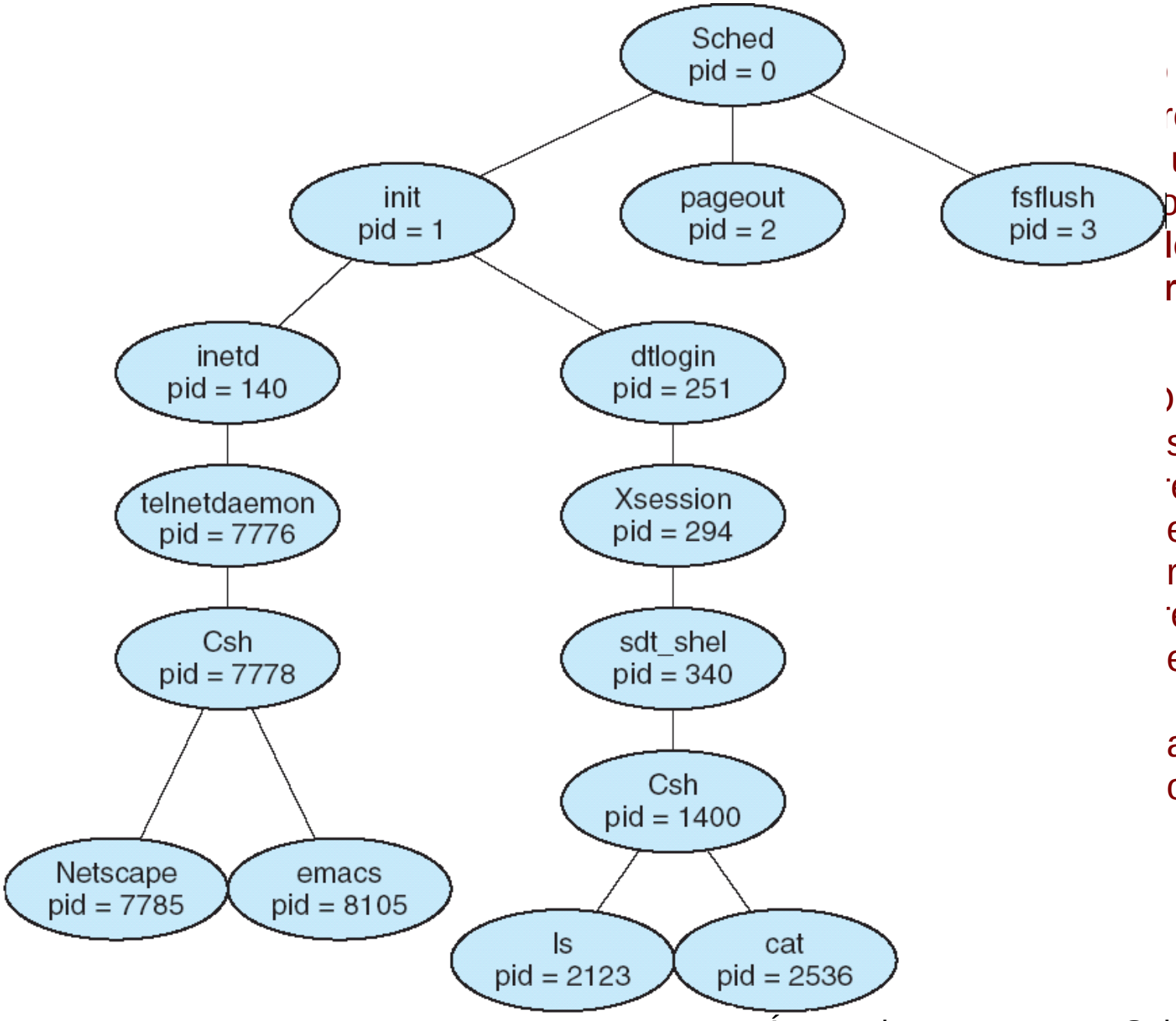
➢Processos podem ser criados e removidos dinamicamente do sistema

➢Um processo, chamado de processo **pai**, pode criar um ou mais processos (chamados processos **filhos**) e, estes por sua vez, podem criar outros processos, formando uma **árvore de processos**

➢O sistema possui inúmeros processos:

- Inicialmente, ao carregar o Sistema Operacional, são criados vários processos
- O usuário ao executar programas, cria um processo para cada um deles
- É possível também criar processos através de chamadas de sistema (*system call*) a partir de outro processo

Árvore de processos no Solaris

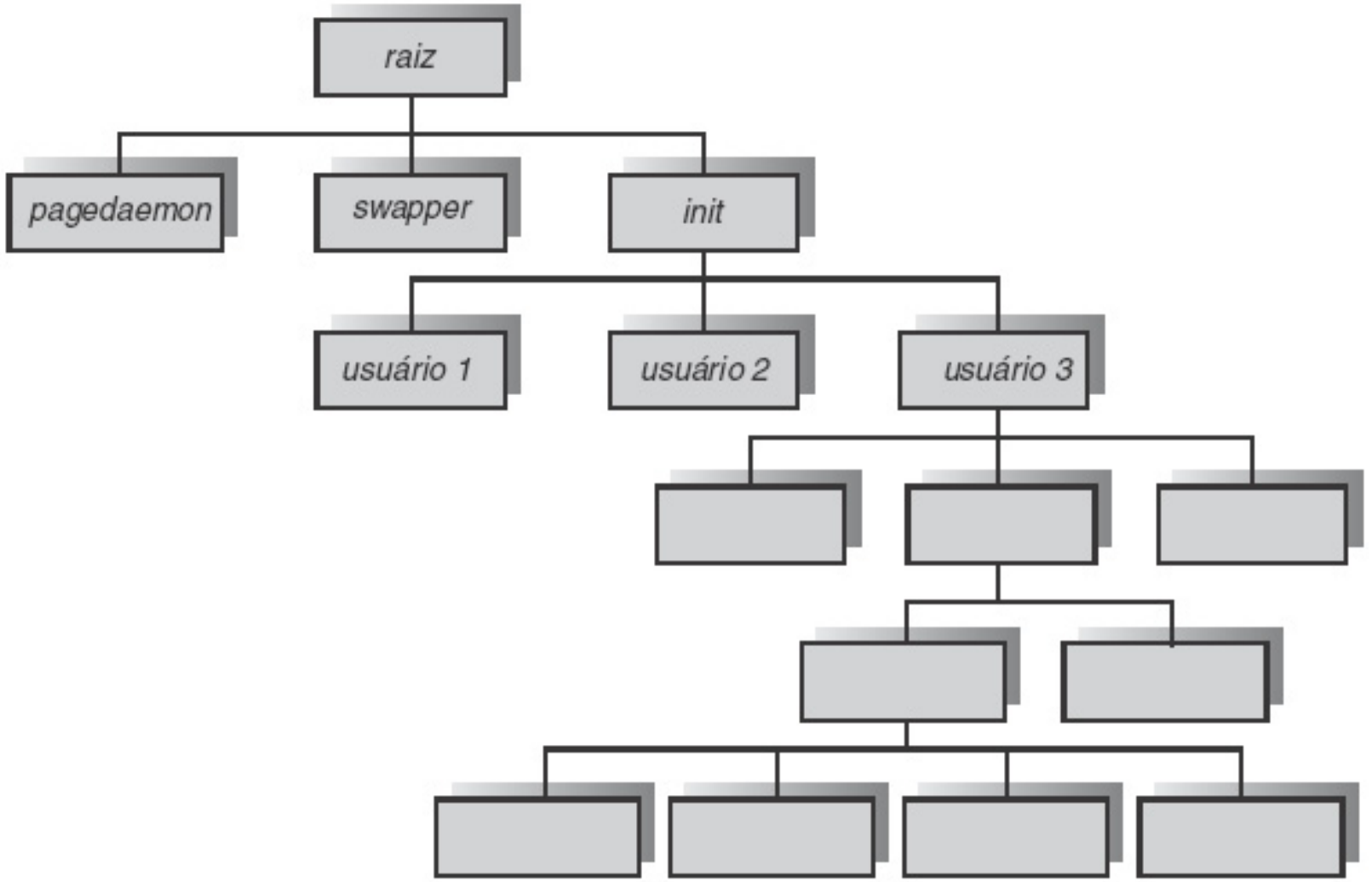


SO identifica cada
rocesso com um nome
um valor inteiro,
pnhecido como
lentificador de
rocesso (ou ***pid***)

processo **Sched** cria
s processos **fsflush**
esponsável por
erenciar o sistema de
rquivos), o **pageout**
esponsável por
erenciar a memória) e
init (processo raiz pai
ara todos os processos
o usuário)

Árvore de processos no Solaris [Silberschatz]

Árvore de processos no UNIX



Criação de Processos

➢Quando um processo cria um outro processo, há algumas possibilidades:

➢**Quanto ao compartilhamento de recursos:**

- Pai e filhos compartilham todos os recursos, ou
- Filhos compartilham um subconjunto dos recursos do pai, ou
- Pai e filho não compartilham recurso algum

➢**Quanto à execução (sincronização):**

- Pai e filhos são executados concorrentemente, ou
- Pai espera até que os filhos terminem a respectiva execução

➢**Quanto ao espaço de endereçamento:**

- Processo-filho é uma duplicata do processo-pai
- Processo-filho contém um novo programa carregado nele

Exemplo: criação de processo no Unix

➢Exemplo no UNIX

- A chamada de sistema **fork()** cria uma cópia exata do processo original e passa a esta cópia todos os atributos do primeiro (como arquivos abertos); como cada um tem sua própria imagem da memória, se o processo-pai alterar suas variáveis, essas alterações não serão visíveis pelo processo-filho e vice-versa

- A chamada de sistema **exec()** é usada após um **fork()** para dar início à execução de um programa a partir do processo-filho, substituindo sua imagem na memória pelo arquivo indicado no primeiro parâmetro da chamada

- A chamada de sistema **wait()** permite a sincronização dos processos pai e filho, interrompendo o processo pai até que o processo filho tenha terminado

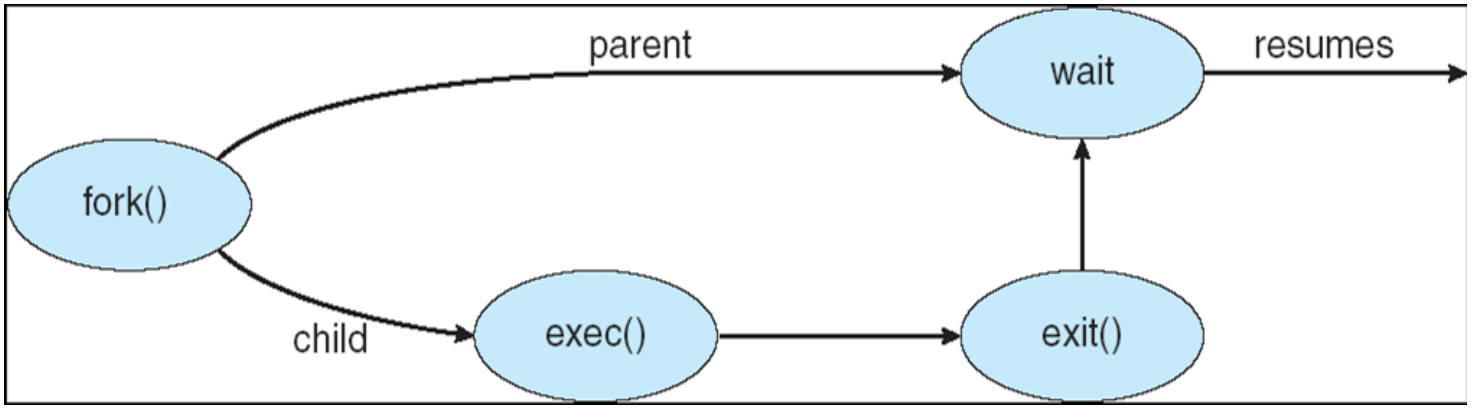
Fork criando um processo separado

```
#include <stdio.h>
#include <unistd.h>
 int main(int argc, char *argv[]) {
   int pid;
/* A chamada de sistema fork() cria um novo processo filho */
pid = fork(); /* O processo filho é uma cópia do espaço de endereços do
                 processo original (pai). Ambos continuam a execução a
                 partir da próxima linha */
if (pid < 0) { /* se pid == -1 houve algum erro, como falta de memória */
      fprintf(stderr, "Fork Falhou");
      exit(-1);
}
else if (pid == 0) { /* Processo Filho */
      execlp("/bin/ls","ls",NULL); /* Filho executa o programa ls */
}
else { /* Processo Pai */
      /* O Processo Pai espera até que o Filho complete a tarefa */
      wait(NULL);
      printf("Processo Filho completou sua tarefa");
      exit(0);
}
}
```

O valor de retorno de **fork** para o processo filho é 0 e para o processo pai é um valor maior que 0 (identificador do filho).

Exemplo de programa C para criar um novo processo [Silberschatz]

Fork criando um processo separado



Criação de processo [Silberschatz]

Criando um processo no Win32

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

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    // allocate memory
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    // create child process
    if (!CreateProcess(NULL, // use command line
      "C:\\WINDOWS\\system32\\mspaint.exe", // command line
      NULL, // don't inherit process handle
      NULL, // don't inherit thread handle
      FALSE, // disable handle inheritance
      0, // no creation flags
      NULL, // use parent's environment block
      NULL, // use parent's existing directory
      &si,
      &pi))
    {
       fprintf(stderr, "Create Process Failed");
       return -1;
    }
    // parent will wait for the child to complete
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    // close handles
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Criação de processos com API Win32 [Silberschatz]

Criando um processo em Java

```
import java.io.*;

public class OSProcess
{
 public static void main(String[] args) throws IOException {
  if (args.length != 1) {
   System.err.println("Usage: java OSProcess <command>");
   System.exit(0);
  }

  // args[0] is the command
  ProcessBuilder pb = new ProcessBuilder(args[0]);
  Process proc = pb.start();

  // obtain the input stream
  InputStream is = proc.getInputStream();
  InputStreamReader isr = new InputStreamReader(is);
  BufferedReader br = new BufferedReader(isr);

  // read what is returned by the command
  String line;
  while ( (line = br.readLine()) != null)
    System.out.println(line);

  br.close();
 }
}
```

Esse método cria um processo externo à JVM e retorna um objeto **Process**.

(Obs.: Ao executar este programa, é preciso passar como parâmetro o nome do programa que será executado como um processo extern

Criando um processo externo com a API Java [Silberschatz]

Término de Processo

➢O processo executa sua última instrução e solicita ao SO que o elimine através da chamada de sistema **exit**

- Envia os dados de saída do filho para o pai (através do **wait**)
- Os recursos do processo são desalocados pelo SO (memória, arquivos abertos e *buffers* de E/S)

➢O pai pode terminar a execução dos processos filhos com a chamada de sistema **abort**

- O filho excedeu os recursos alocados
- A tarefa atribuída ao filho não é mais necessária
- Se o pai está terminando
  - Alguns sistemas operacionais não permitem que o filho continue se seu pai tiver terminado, ocorrendo um **término em cascata**
  - No UNIX, se o pai terminar todos os seus filhos receberão como seu novo pai o processo **init**

UFABC

Comunicação entre processos

Processos Cooperativos

- Os processos executando no sistema podem ser cooperativos ou independentes
- Um processo é ***independente*** se não puder afetar ou ser afetado por outro processo em execução no sistema
- Um processo ***cooperativo*** pode afetar ou ser afetado pela execução de outros processos no sistema
- Qualquer processo que compartilhe dados com outro é um processo cooperativo

- **Vantagens** da cooperação de processos
  - Compartilhamento de informações (por ex., arquivo compartilhado)
  - Velocidade na computação (se o computador possuir mais de uma CPU)
  - Modularidade (é importante para sistemas grandes)
  - Conveniência (um usuário pode editar, imprimir e compilar em paralelo)

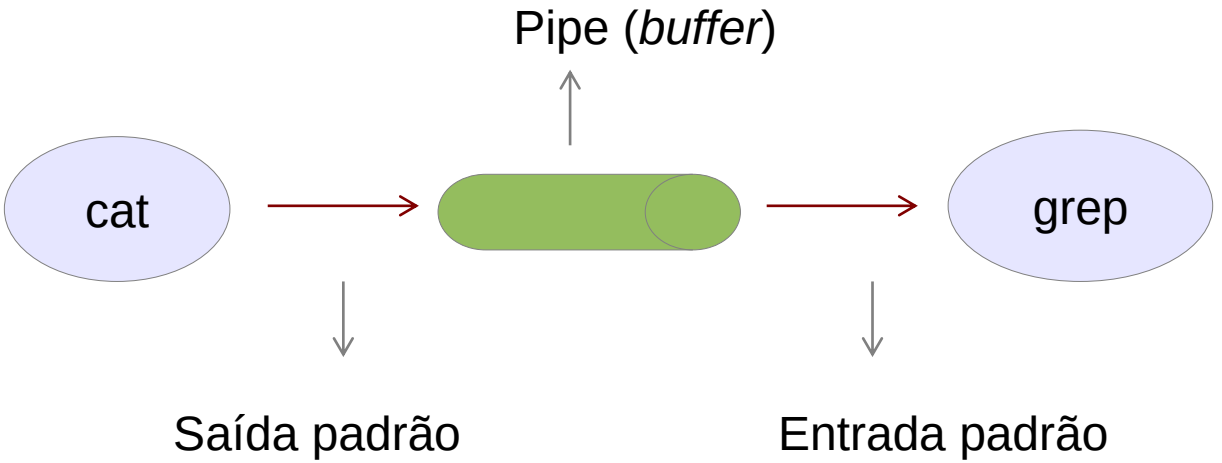
Comunicação entre processos

➢Processos cooperativos precisam de mecanismos de comunicação entre processos (*Interprocess Communication – IPC*) para possibilitar a troca de dados e informações

- Memória compartilhada
  - Uma região de memória é estabelecida para ser compartilhada entre processos
  - Troca de informações pela leitura/escrita de dados na região compartilhada
- Troca de mensagens ou Passagem de mensagens
  - Envio de mensagens entre os processos cooperativos
- Áreas de memória específicas
  - Sistema Operacional (buffers, pipes, área de transferência)

Comunicação entre processos

- Linux / Windows
- Pipe: área de memória (*buffer*) do Sistema Operacional para a transferência de dados E/S entre processos
  - **comando1 | comando2**
  - Redireciona a saída do *comando1* para a entrada do comando2
  - Ex.: cat arq1 arq2 | grep aluno (Linux)
  - O primeiro processo (cat) gera como saída a concatenação de dois arquivos e o segundo processo (grep) seleciona todas as linhas contendo a palavra 'aluno'

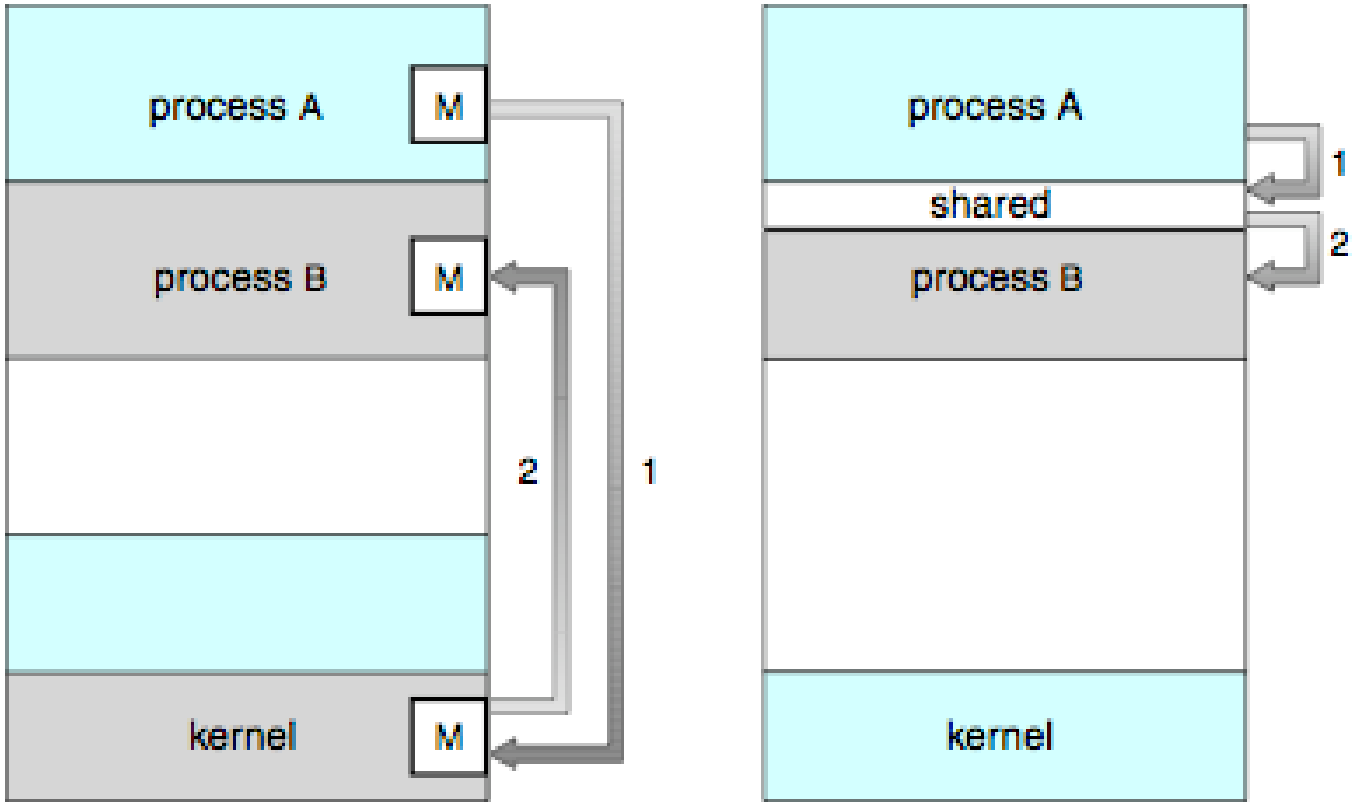


Comunicação entre processos

- *Spooling*
  - Área de impressão de documentos
  - Implementação: FILA de arquivos
  - Todos os processos enviam documentos para o controlador do *spool* que os organiza para então imprimi-los

- Área de Transferência (do Windows, por ex.)
  - *Buffer* para armazenamento de dados (texto, gráficos, tabelas)
  - Imprescindível para as operações CTRL+C e CTRL+V
  - Implementações semelhantes em outros Sistemas Operacionais (Linux, Solaris)

Comunicação entre processos



Troca de mensagens

Memória Compartilhada

Modelos de comunicação [Silberchatz]

A **Memória Compartilhada** é mais rápida do que a **Troca de Mensagem** porque os sistemas de **Troca de Mensagem** normalmente são implementados com o uso de muitas **Chamadas de Sistema**, exigindo a intervenção constante do kernel

Memória compartilhada

- Uma região de memória é estabelecida para ser compartilhada entre processos (no espaço de endereços do processo criador)
- A comunicação entre processos ocorre pela leitura/escrita de dados na região compartilhada
- Oferece maior rapidez e conveniência (não requer assistência do kernel para a comunicação)

- Um exemplo: Problema Produtor-Consumidor
  - Paradigma para processos cooperativos: um processo produtor gera informações que são consumidas por um processo consumidor
  - Uma solução para esse problema utiliza memória compartilhada, um *buffer* de itens de dados
    - Os itens de dados são armazenados no *buffer* pelo processo produtor e
    - O processo consumidor consome (remove) os itens de dados do

Problema Produtor-Consumidor

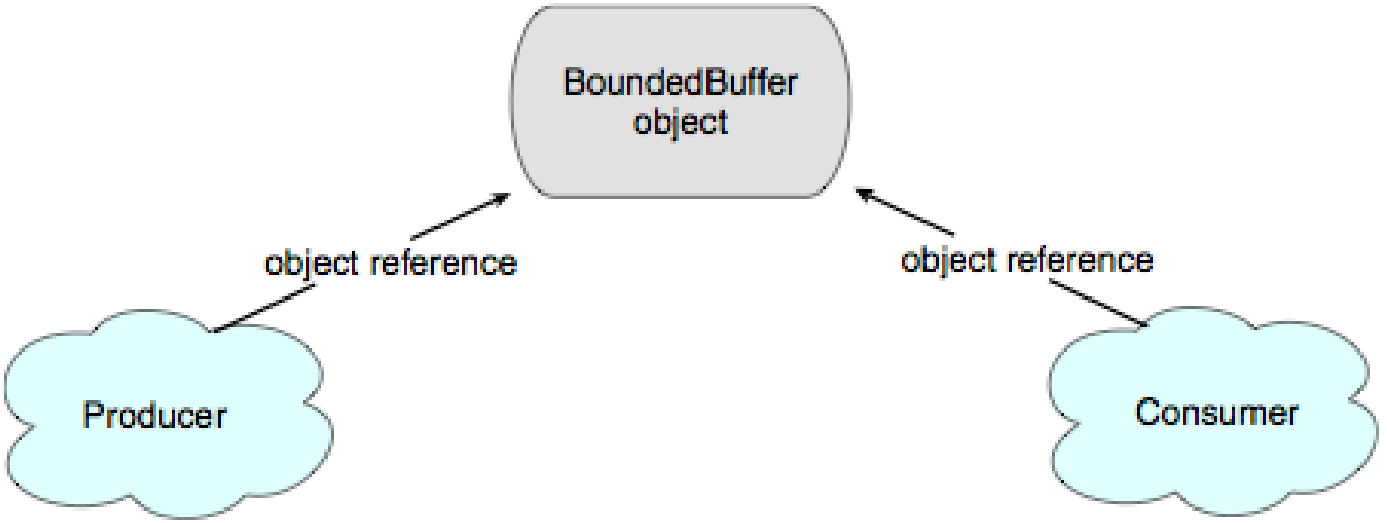
➢O **produtor** e o **consumidor** devem ser **sincronizados** de modo que o consumidor não tente consumir um item que não tenha sido ainda produzido

➢Quanto ao tamanho, o *buffer* pode ser:

- ***buffer ilimitado* (*unbounded-buffer*):** não coloca qualquer limite prático no tamanho do *buffer* (o consumidor precisa checar se há itens no buffer (i.e., se o buffer não está vazio; o produtor pode sempre produzir e inserir novos itens))
- ***buffer limitado* (*bounded-buffer*):** considera que existe um tamanho de *buffer* fixo (consumidor da mesma forma, precisa checar se o buffer não está vazio, enquanto que o produtor precisa verificar se o buffer não está cheio, ou seja, é possível inserir um novo item)

➢O acesso e manipulação da memória compartilhada (***buffer***) deve ser explicitamente codificado pelo **programador da aplicação**

Memória compartilhada em Java



Simulando memória compartilhada em Java [Silberchatz]

Nesta solução é utilizado um *buffer* limitado (objeto BoundedBuffer) utilizando um array circular de objetos para simular a memória compartilhada

Para ser possível que tanto o produtor como o consumidor tenham acesso a esse *buffer*, é necessário que ambos tenham uma referência ao objeto BoundedBuffer, que deve ser passada aos mesmos no momento de sua criação

Bounded-Buffer – Buffer Limitado
Solução para Memória Compartilhada

```
public interface Buffer
{
 // Produtores chamam este método
 public abstract void insert(Object item);

   // Consumidores chamam este método
 public abstract Object remove();
}
```

Interface para implementação de buffer [Silberchatz]

Bounded-buffer – buffer limitado

```
import java.util.*;
public class BoundedBuffer implements Buffer {
    private static final int BUFFER_SIZE = 5;
    private int count; // número de items no buffer
    private int in; // aponta para a próxima posição livre
    private int out; // aponta para a próxima posição cheia
    private Object[] buffer;
    public BoundedBuffer()  {
        // o buffer inicialmente está vazio
        count = 0;
        in = 0;
        out = 0;
        buffer = new Object[BUFFER SIZE];
    }
    // Produtores chamam este método
    public void insert(Object item) {
        // Slide 44
    }
    // Consumidores chamam este método
    public Object remove() {
       // Slide 45
    }
}
```


Solução de memória compartilhada para o problema Produtor-Consumidor [Siberschatz]

Bounded-buffer – método insert()

```
public void insert(Object item) {
    while (count == BUFFER_SIZE)
        ; // não faz nada -- nenhum buffer livre
    // acrescenta um item ao buffer
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
}
```

Método insert() [Silberchatz]

Bounded-buffer – método remove()

```
public Object remove() {
    Object item;
    while (count == 0)
        ; // não faz nada – nada para consumir
    // remove um item do buffer
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    return item;
}
```

Método remove() [Silberchatz]

Troca de mensagens

➢**Troca de Mensagens (ou Passagem de Mensagens)**

- É um método de comunicação de processos que permite que eles se comuniquem entre si sem recorrer às variáveis compartilhadas (útil em ambiente distribuído)

➢O mecanismo de troca de mensagens oferece pelo menos duas operações:

- ***send(mensagem)***
- ***receive(mensagem)***

➢Se **P** e **Q** desejam se comunicar, eles precisam:

- estabelecer um *enlace* ou ***canal de comunicação*** entre eles
- trocar mensagens através de **send/receive**

➢**Implementação** do enlace de comunicação

- físico (memória compartilhada, barramento de *hardware* ou rede)
- lógico (propriedades lógicas)

Troca de mensagens

- Métodos para implementar logicamente um enlace de comunicação e as operações **send/receive**:
  - Comunicação direta ou indireta
  - Comunicação síncrona ou assíncrona
  - Com a utilização de *buffer* ou não

Comunicação direta

➢**Nomeação** – processos que desejam se comunicar precisam ter uma forma de se referenciarem mutuamente

➢Na **comunicação direta** os processos precisam nomear explicitamente um ao outro:

- ***send* (P, mensagem)** – envia uma mensagem ao processo P
- ***receive*(Q, mensagem)** – recebe uma mensagem do processo Q

➢Propriedades do enlace de comunicação

- Os enlaces são estabelecidos automaticamente
- Um enlace é associado a exatamente um par de processos de comunicação
- Entre cada par existe exatamente um enlace (ou canal)
- O enlace pode ser unidirecional, mas normalmente é bidirecional

Comunicação Indireta

- As mensagens são enviadas e recebidas a partir de caixas de correio (***mailbox***)
  - Cada caixa de correio possui uma identificação exclusiva
  - Os processos só podem se comunicar se tiverem uma caixa de correio compartilhada

- As primitivas são definidas como:
  - ***send(A, mensagem)*** – enviar uma mensagem à caixa de correio A
  - ***receive(A, mensagem)*** – receber uma mensagem da caixa de correio A

Comunicação Indireta (cont.)

- Propriedades do enlace de comunicação
  - Um enlace é estabelecido apenas se os processos compartilharem uma caixa de correio comum
  - Um enlace pode ser associado a muitos processos
  - Cada par de processos pode compartilhar vários enlaces de comunicação
  - O enlace pode ser unidirecional ou bidirecional

- Operações
  - Criar uma nova caixa de correio
  - Enviar e receber mensagens através da caixa de correio
  - Excluir uma caixa de correio
  - O processo que cria a caixa de correio é por padrão o proprietário da mesma, somente ele poderá receber as mensagens inseridas por outros processos
  - Porém, a posse e privilégio podem ser passados a outros processos (problema: vários receptores para uma mensagem)

Comunicação Indireta (cont.)

- **Compartilhamento** de caixa de correio
  - P_1, P_2 e P_3 compartilham a caixa de correio A
  - P_1, envia; P_2 e P_3 executam a operação ***receive***
  - Quem receberá a mensagem? A resposta depende da opção escolhida

- **Opções**
  - Permitir que apenas um processo por vez execute uma operação ***receive***
  - Permitir que o sistema selecione arbitrariamente o receptor (i.e., P_2 ou P_3, mas não ambos); o emissor é notificado sobre quem foi o receptor

Sincronização

➢A troca de mensagens pode ser **bloqueante** ou **não-bloqueante**

➢**Bloqueante** é considerada **síncrona**

- No ***send* bloqueante**, o emissor é bloqueado até que a mensagem seja recebida pelo processo receptor ou caixa de correio
- No ***receive* bloqueante**, o receptor é bloqueado até que uma mensagem esteja disponível

➢**Não-bloqueante** é considerada **assíncrona**

- No ***send* não-bloqueante**, o emissor envia a mensagem e retoma a operação
- No **receive não-bloqueante**, o receptor recebe uma mensagem válida ou então uma mensagem nula

➢Quando tanto *send* como *receive* forem bloqueantes, ocorre um **ponto de encontro** (*rendezvous*) entre o remetente e o receptor

Buffering

➢Utilizado tanto na **comunicação direta** quanto na **indireta**, as mensagens são armazenadas em uma fila temporária (*buffer*). Um *buffer* podem ser implementado de três maneiras:

1. **Capacidade zero** – nenhuma mensagem armazenada
   O emissor precisa bloquear e esperar o receptor (*rendezvous*)
2. **Capacidade limitada** – tamanho finito para *n* mensagens armazenadas
   O emissor precisa esperar (fica bloqueado) se a fila encher
3. **Capacidade ilimitada** – tamanho infinito
   O emissor nunca espera

O caso de capacidade zero é também chamado de sistema de mensagem sem *buffering*; os outros casos são chamados de *buffering* automático

Interface para troca de mensagem do Produtor-Consumidor

```
public interface Channel
{
   // Send a message to the channel
   public abstract void send(Object item);

   // Receive a message from the channel
   public abstract Object receive();
}
```

Interface para troca de mensagens [Silberchatz]

Caixa de Correio para a troca de mensagem

```
public class MessageQueue implements Channel
{
    private Vector queue;

    public MessageQueue() {
       queue = new Vector();
    }

    // This implements a nonblocking send
    public void send(Object item) {
       queue.addElement(item);
    }

    // This implements a nonblocking receive
    public Object receive() {
       if (queue.size() == 0)
          return null;
       else
          return queue.remove(0);
    }
}
```

Caixa de correio para a troca de mensagem [Silberchatz]

O processo produtor para a solução de troca de mensagem

- O produtor

```
Channel mailBox;

while (true) {
    Date message = new Date();
    mailBox.send(message);
}
```

O processo produtor (parte) [Silberchatz]

O processo consumidor para a solução de troca de mensagem

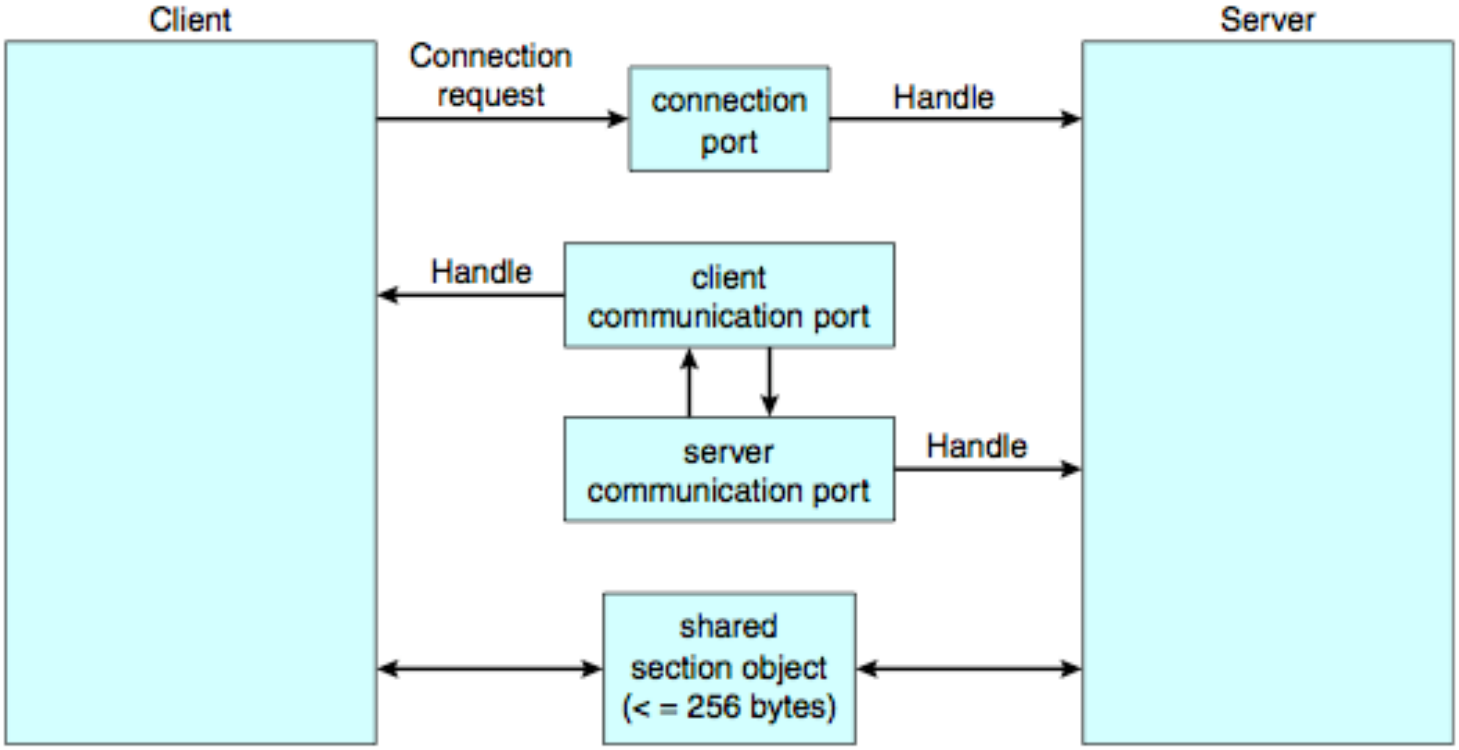
- O consumidor

```
Channel mailBox;

while (true) {
   Date message = (Date) mailBox.receive();
   if (message != null)
      // consume the message
}
```

O processo consumidor [Silberchatz]

Passagem de mensagem no Windows XP



O Windows XP possui uma arquitetura modular (vários subsistemas)
Os programas de aplicação podem ser considerados clientes dos subsistemas (servido
O recurso de troca de mensagens no Windows XP é chamado de **Chamada de Proce**

Referências

- [Silberschatz] SILBERCHATZ, A., GALVIN, P. B. e GAGNE, G. **Sistemas Operacionais com Java**. 7ª ed., Rio de Janeiro: Elsevier, 2008.
- [Tanenbaum] TANENBAUM, A. **Sistemas Operacionais Modernos**. 3ª ed. São Paulo: Prentice Hall, 2009.
- [MACHADO] MACHADO, F. B. e MAIA, L. P. **Arquitetura de Sistemas Operacionais**. 4ª ed., Rio de Janeiro: LTC, 2007.