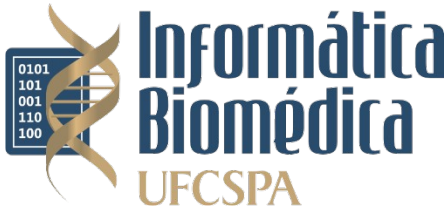
UFCSPA Informática Biomédica

Sistemas Operacionais

Processos

Prof. João Gluz



Porto Alegre, RS, Brasil
2019

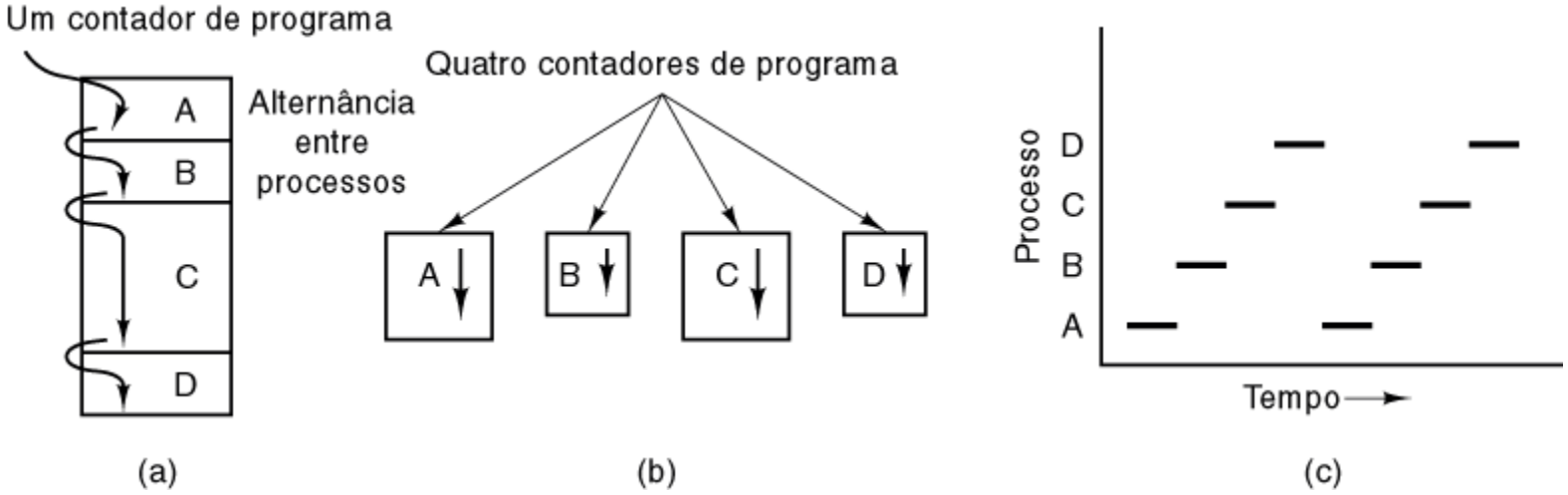
Processos

- Um sistema operacional executa uma variedade de programas:
  - Sistema Batch – *jobs*
  - Sistema Tempo Compartilhado (*Time-shared)* – programas do usuário ou tarefas
- Livros usam os termos *job* e *processo* quase que indeterminadamente.
- Processo – um programa em execução; execução do processo deve progredir de maneira seqüencial.

Conceito de Processo (Cont.)

- Programa é uma entidade ***passiva*** armazenada em disco (**arquivo executável**), processo é uma entidade ***ativa***
  - O program se torna um processo quando o arquivo executável é carregado na memória
- A execução de um programa começa via click do mouse nas interfaces GUI, digitação do nome na linha de comando, etc
- Um programa pode gerar múltiplos processos
  - Por exemplo, vários usuários podem estar rodando o mesmo programa

Processos em Execução



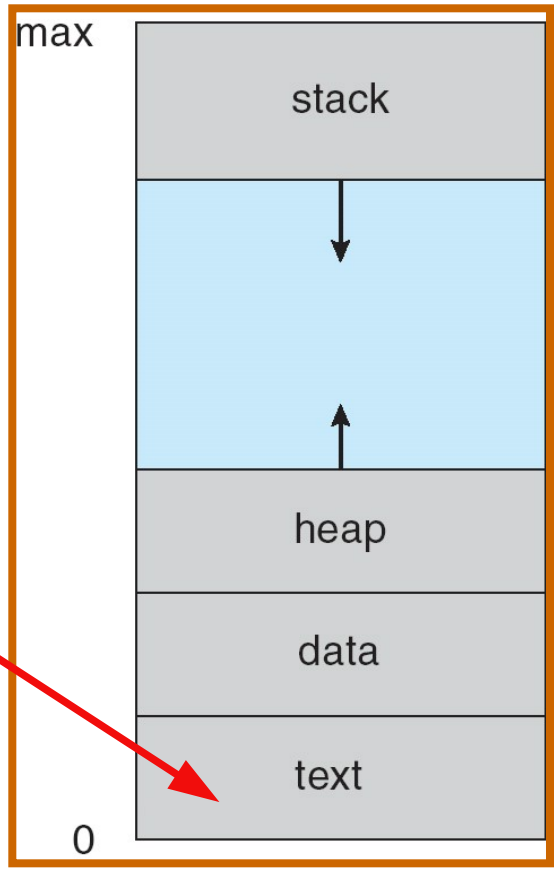
- Multiprogramação de quatro programas – troca explícita (programada) do fluxo de execução da CPU
- Modelo conceitual de 4 processos sequenciais, independentes
- Somente um programa está ativo a cada momento

Partes de um Processo

- Um processo inclui:
  - O código do programa, também chamado de segmento de “texto” (***text***)
  - Estado atual, incluindo contador de programa e registradores do processador
  - Segmento de pilha (***stack***) contendo dados temporários, parâmetros de chamadas de função, endereços de retorno, variáveis locais
  - Segmento de dados (***data***) contendo variáveis globais
  - Segmento com um “amontoado” (***heap***) de blocos de memória alocada dinamicamente durante a execução

Processo na Memória

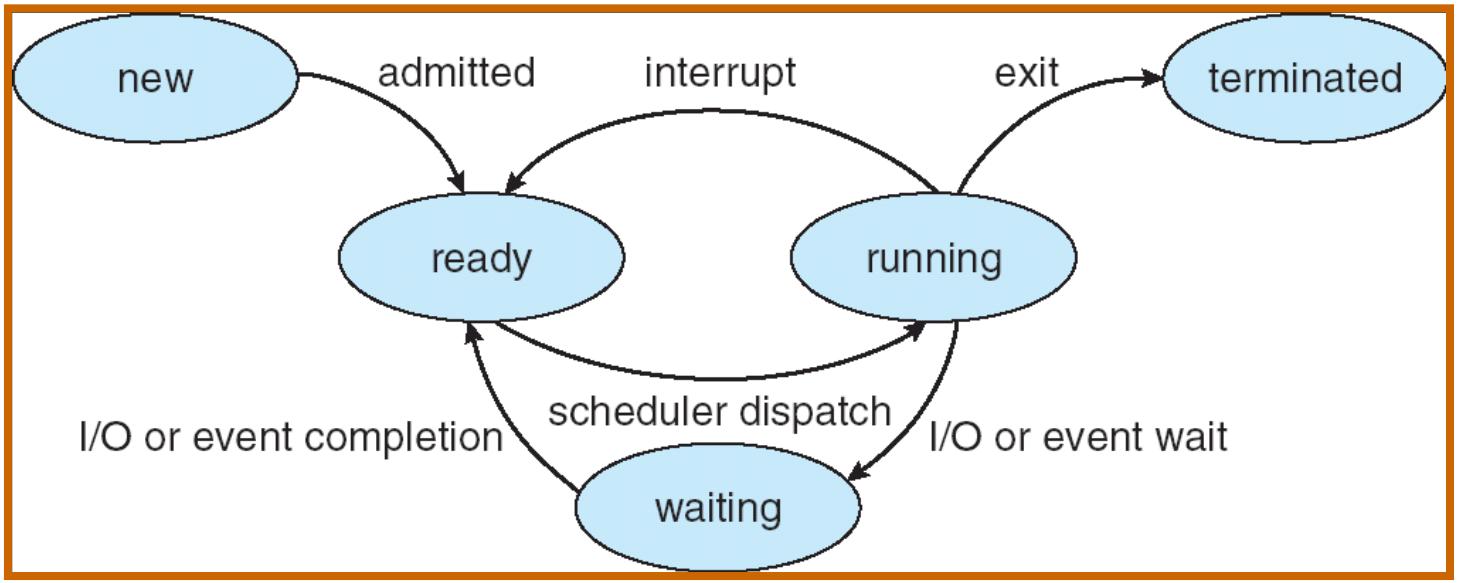
Não é um segmento de “texto”, é o segmento com o próprio programa



Estados de um Processo

- Durante a execução de um processo, ele altera seu *estado*
  - **Novo** (***new***): O processo está sendo criado.
  - **Executando** (***running***): instruções estão sendo executadas.
  - **Esperando** (***waiting***): O processo está esperando algum evento acontecer.
  - **Pronto** (***ready***): O processo está esperando ser associado a um procesador.
  - **Terminado** (***terminated***): O processo terminou sua execução.

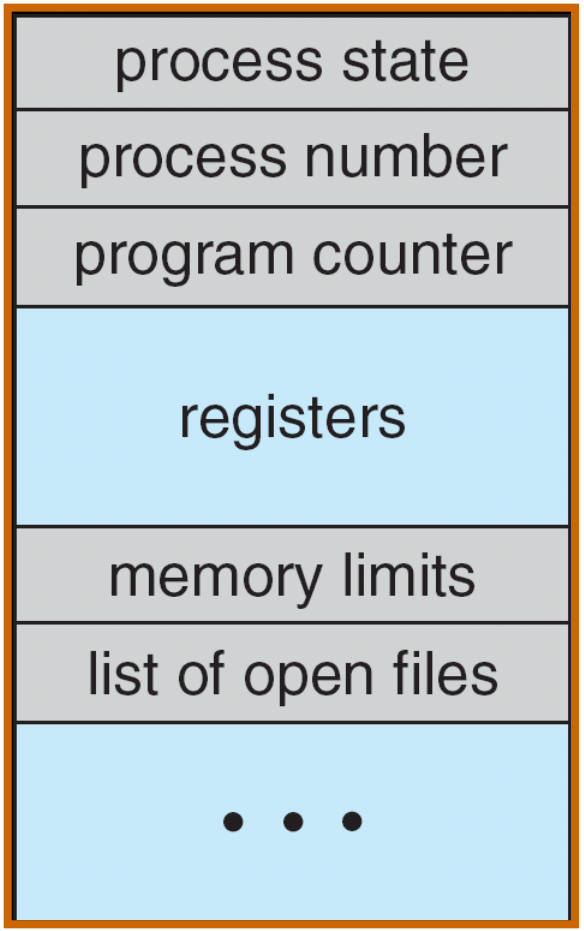
Diagrama de Estados de Processos



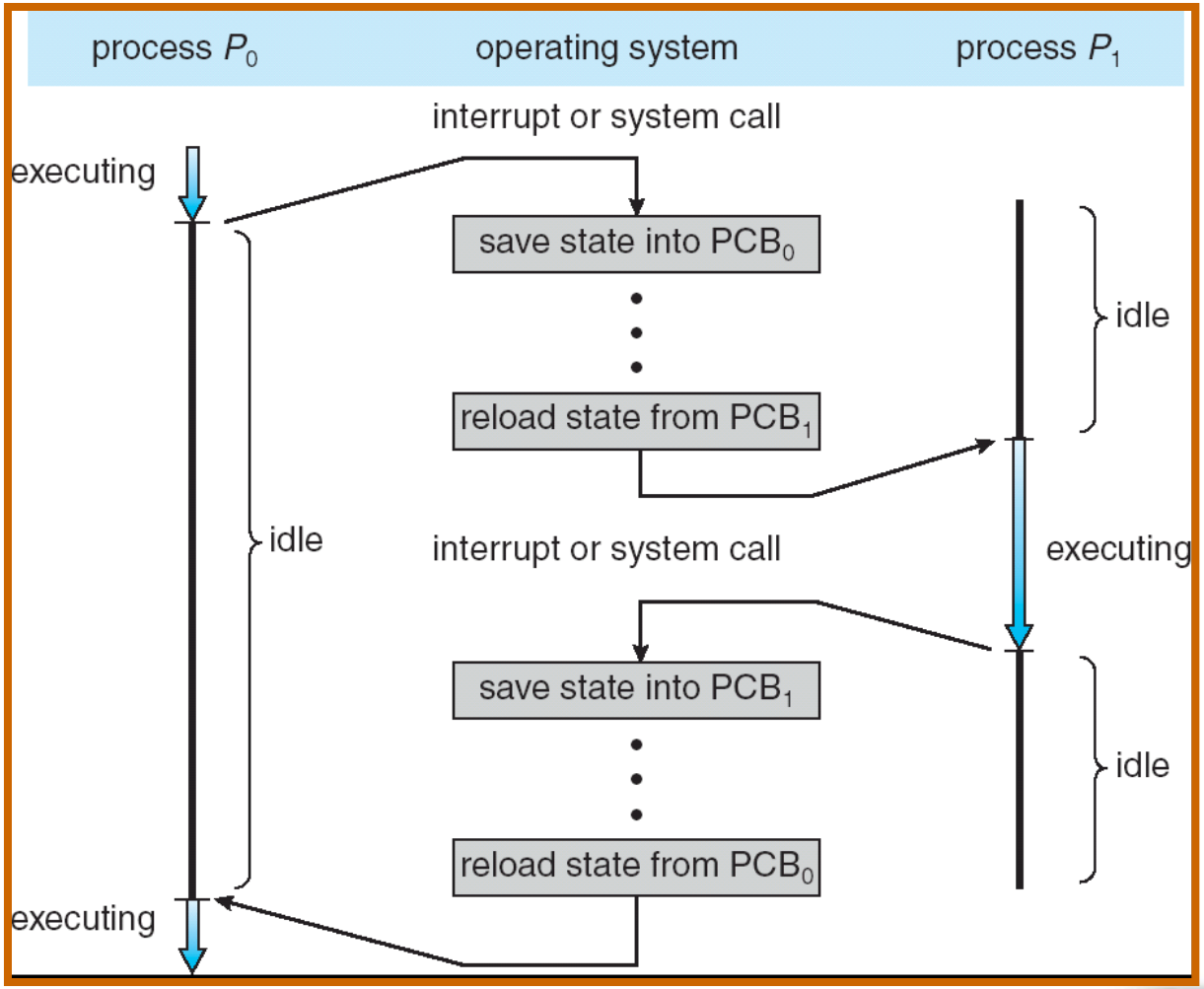
Process Control Block (PCB)

- O **PCB** ou Bloco de Controle de Processos armazena informações associada com cada processo.
  - Estado do Processo
  - Contador de Programas
  - Registradores da CPU
  - Informações de escalonamento da CPU
  - Informação de Gerenciamento de memória
  - Informação para Contabilidade
  - Informações do status de E/S

Process Control Block (PCB)



Troca de CPU entre Processos



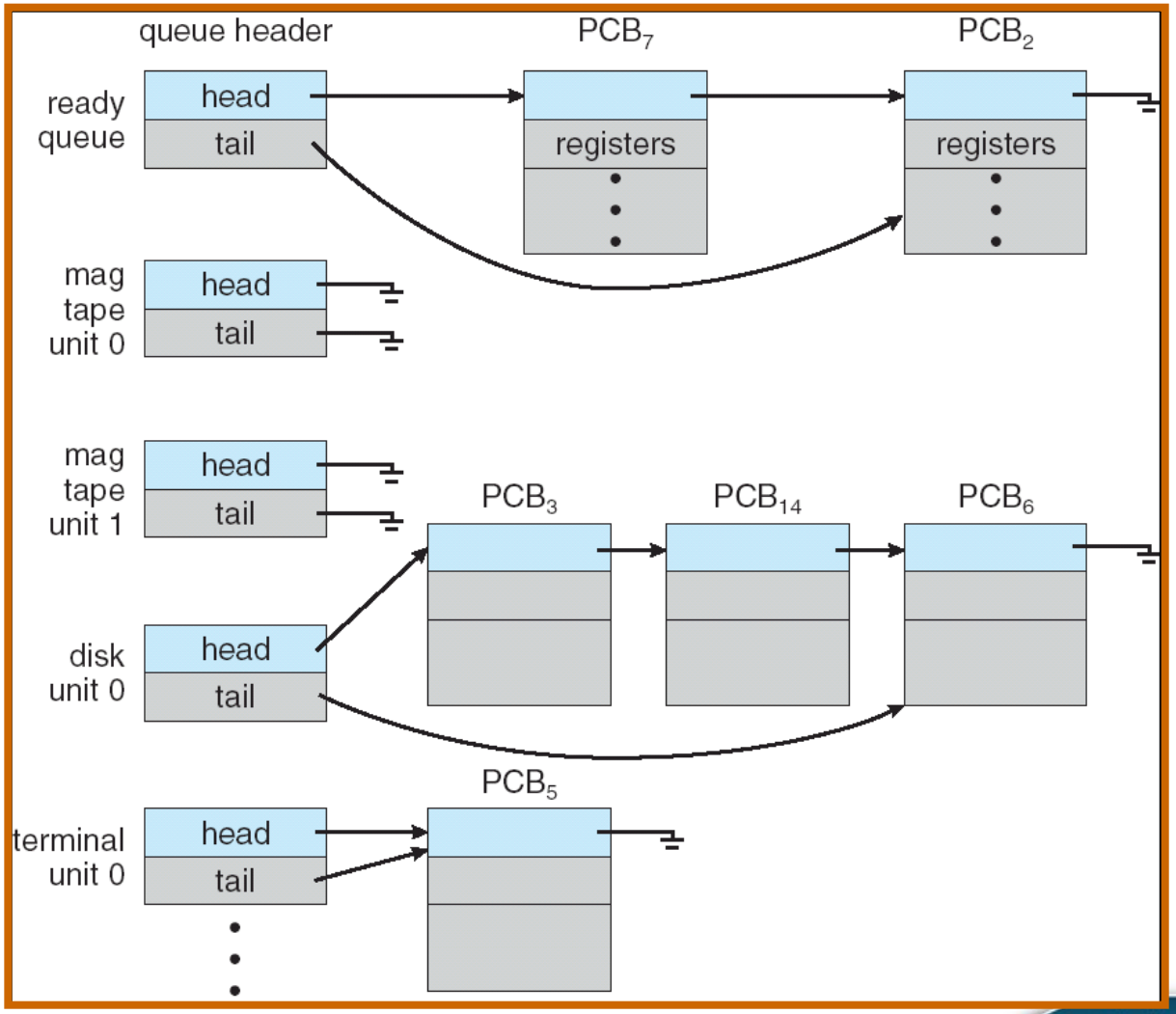
Escalonamento de Processos

- O escalonamento de processos maximiza o uso da CPU, permitindo o compartilhamento de tempo da CPU (time sharing) através do chaveamento dos processos
- Escalonador (*scheduler*) de processos é o componente do kernel que seleciona entre os processo prontos pra execução qual será o próximo a executar na CPU
- Para tanto o escalonador usa diversas **filas de processos** (que na verdade são filas de PCBs)

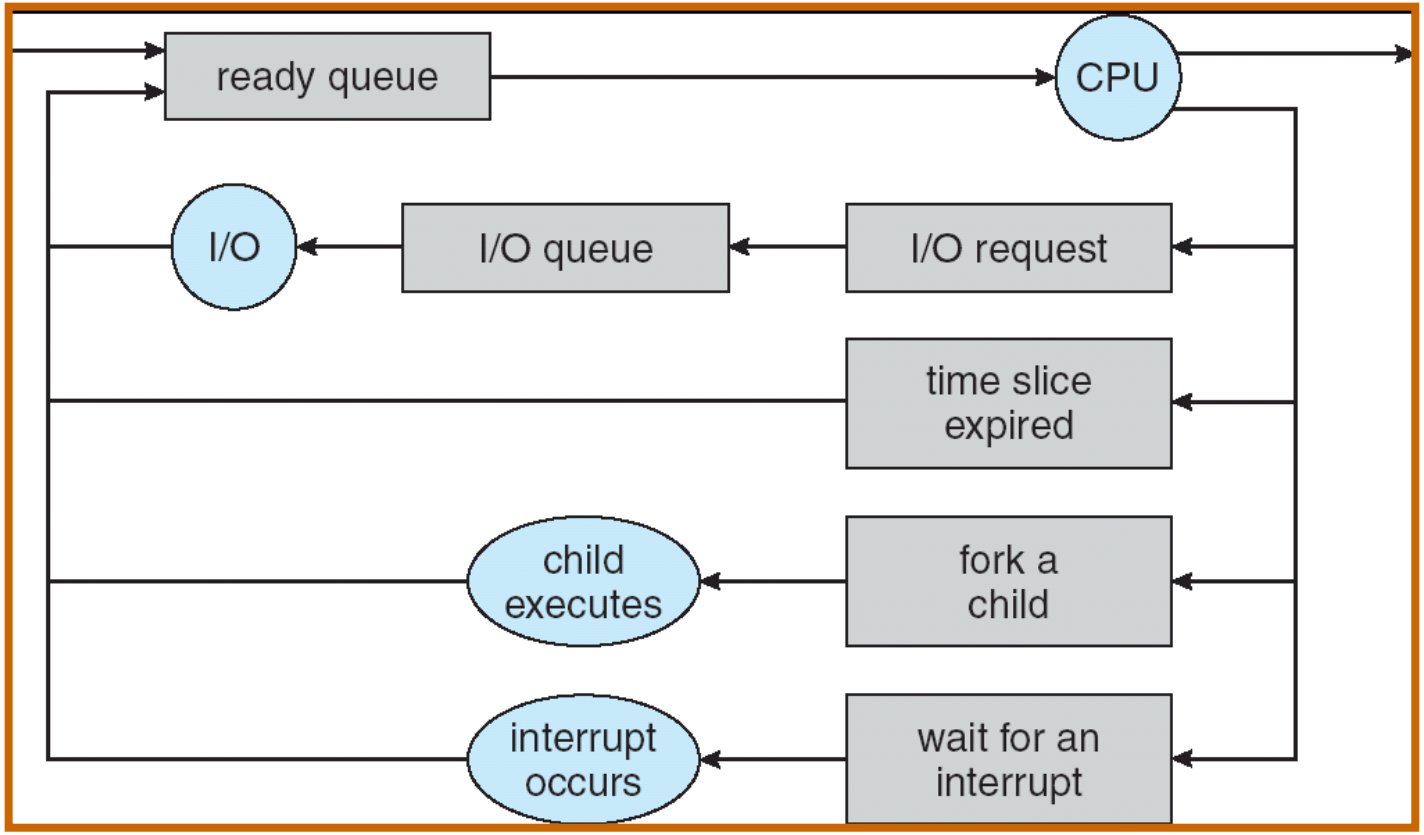
Filas de Escalonamento de Processos

- **Fila de *Jobs*** – conjunto de todos os processos no sistema.
- **Fila de Processos prontos** (*Ready queue*) – conjunto de todos os processos residentes na memória principal, prontos e esperando para executar.
- **Fila de dispositivos** – conjunto dos processos esperando por um dispositivo de E/S.
- Migração de processos entre as várias filas.

Fila de Processos Pronto e Várias Filas de E/S



Representação de Escalonamento de Processos

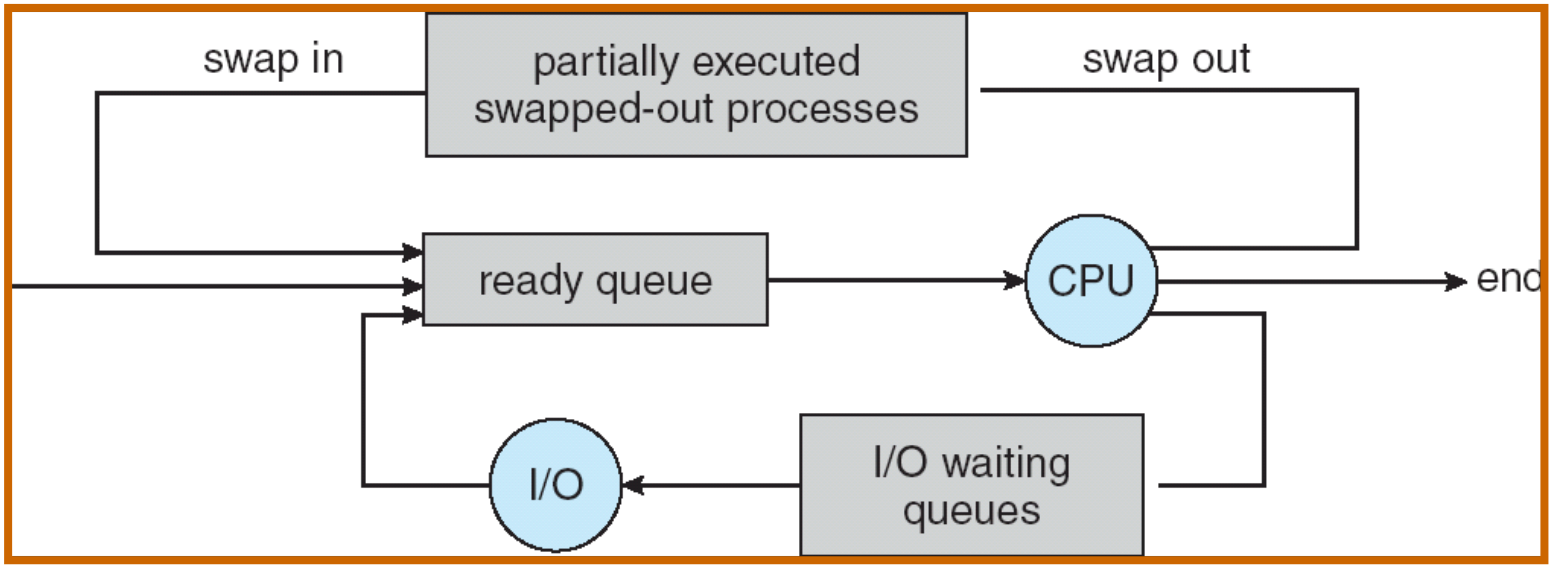


Escalonadores

- **Escalonador de Curto Prazo** (ou escalonador da CPU) – seleciona qual processo deve ser executados a seguir e aloca CPU para ele – é o escalonamento executado pelo kernel do SO
- **Escalonador de Longo Prazo** (ou escalonador de *Jobs*) – seleciona quais processos devem ser trazidos para a fila de processos prontos.
  - Processos podem ser divididos em limitados pela CPU (CPU bound) ou limitados pela E/S (I/O bound)
  - O escalonador de longo prazo busca obter um bom mix de ambos tipos de processos

Inclusão do Escalonador Intermediário

- **Escalonador Intermediário** pode ser usado para diminuir o grau de multiprogramação: remove processos da memória, armazena em disco e recupera do disco para continuar a execução (swapping) – relacionado à memória virtual



Escalonadores (Cont.)

- Escalonador de curto prazo é invocado muito freqüentemente (milisegundos) $\Rightarrow$ (deve ser rápido).
- Escalonador de longo prazo é invocada muito infreqüentemente (segundos, minutos) $\Rightarrow$ (pode ser lento).
- O escalonador de longo prazo controla o *grau de multiprogramação.*
- Processos podem ser descritos como:
  - **Processos com E/S predominante** (I/O-*bound process)* – gasta mais tempo realizando E/S do que computando, muitos ciclos curtos de CPU.
  - **Processos com uso de CPU predominante** (CPU-*bound process)* – gasta mais tempo realizando computações; poucos ciclos longos de CPU.

Troca de Contexto

- Quando CPU alterna para outro processo, o sistema deve salvar o estado do processo deixando o processador e carregar o estado anteriormente salvo do processo novo via **troca de contexto**.
- Contexto de um processo é representado na PCB
- Tempo de troca de contexto é sobrecarga no sistema; o sistema não realiza trabalho útil durante a troca de contexto
  - Quanto mais complexo o SO e o PCB, mais longa é a troca de contexto
- Tempo de Troca de Contexto é dependente de suporte em hardware.

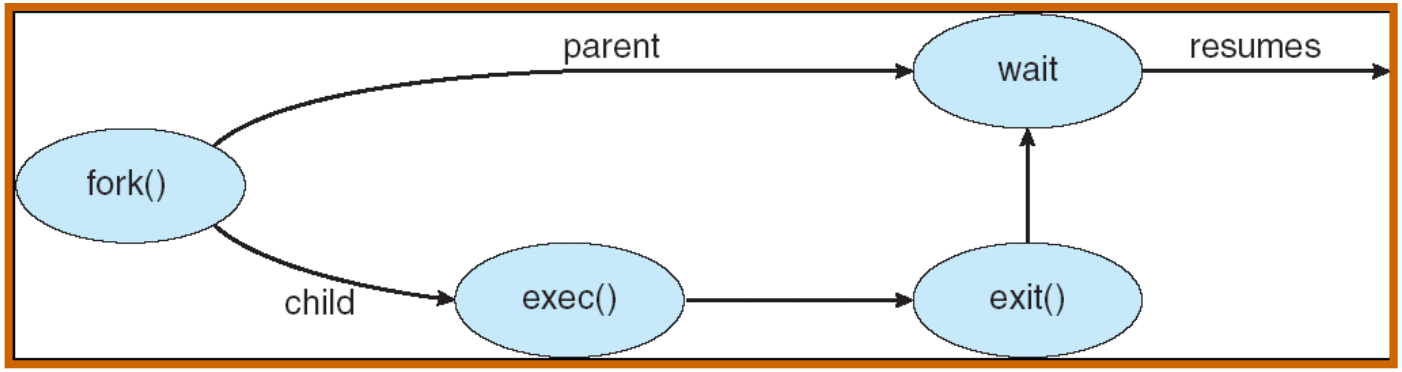
Criação de Processos

- Processo pai cria processo filho, o qual, por sua vez, pode criar outros processos, formando uma árvore de processos.
- Geralmente, processos são identificados e gerenciados via um **Identificador de Processos** (*Process IDentifier* - ***PID***)
- Compartilhamento de Recursos
  - Pai e filho compartilham todos os recursos.
  - Filho compartilha um subconjunto dos recursos do pai.
  - Pai e filho não compartilham recursos.
- Execução
  - Pai e filho executam concorrentemente.
  - Pai espera até filho terminar.

Criação de Processos (Cont.)

- Espaço de endereçamento
  - Filho duplica espaço do pai.
  - Filho tem um programa carregado no seu espaço.
- Exemplos no UNIX
  - Chamada de sistemas ***fork()*** cria um novo processo.
  - Chamada de sistemas ***exec()*** é usada após o ***fork()*** para sobrescrever o espaço de memória do processo com um novo programa.

Criação de Processos (Cont.)



Criando Processos em C no UNIX/Linux

```
int main()
{
   Pid_t  pid;
   /* Criacao do outro processo */
   pid = fork();
   if (pid < 0) {
        /* houve um erro*/
        fprintf(stderr, "Erro no fork()");
        exit(-1);
   } else if (pid == 0) {
        /* este codigo e' do processo filho*/
        execlp("/bin/ls", "ls", NULL);
   } else {
        /* este codigo e´ do processo pai */
        /* pai irá esperar o filho completar execução */
        wait (NULL);
        printf ("Processo filho terminou de executar");
        exit(0);
   }
}
```

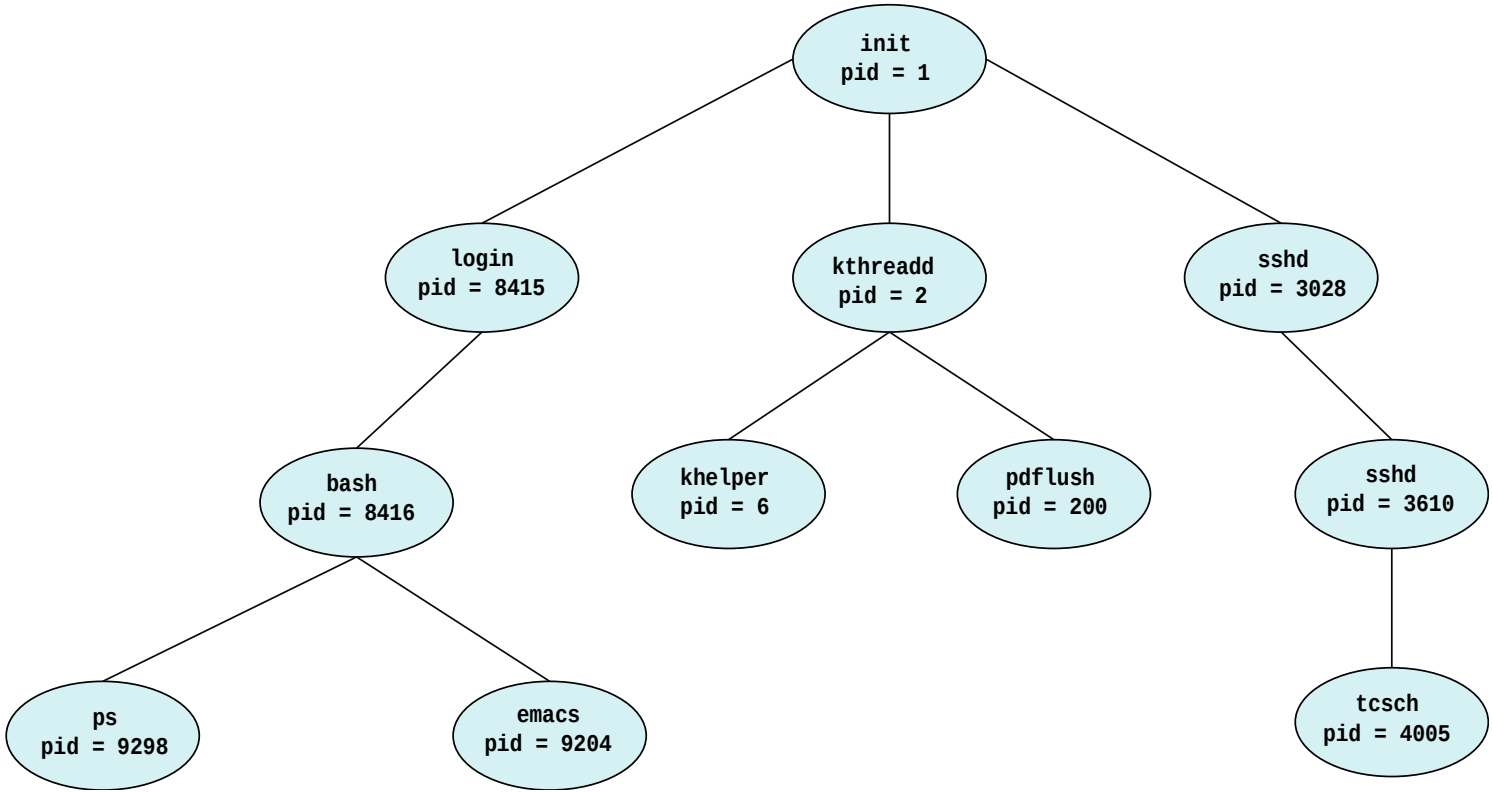
Criando Processos em C no Windows

```
#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 */
     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);
}
```

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

Exemplo de Hierarquia de Processos do Linux



Terminação de Processos UNIX/Linux

- Processo executa última declaração e pede ao sistema operacional para decidir (***exit()***).
  - Dados de saída passam do filho para o pai (via ***wait()***).
  - Recursos do processo são desalocados pelo sistema operacional.
- Pai pode terminar a execução do processo filho (***abort()***).
  - Filho se excedeu alocando recursos.
  - Tarefa delegada ao filho não é mais necessária.
  - Pai está terminando.
    - Sistema operacional não permite que um filho continue sua execução se seu pai terminou.
    - Todos os filhos terminam - Terminação em cascata.

Terminação de Processos

- Alguns sistemas operacionais não permitem a existência de filhos se seu pai tiver terminado. Se um processo terminar, todos os seus filhos também deverão ser encerrados.
  - **terminação em cascata** - todos os filhos, netos, etc. são demitidos.
  - A finalização é iniciada pelo sistema operacional.
- No UNIX/Linux o processo pai pode aguardar o término de um processo filho usando a chamada de sistema ***wait()***. A chamada retorna informações de status e o pid do processo finalizado

```
pid = wait(&status);
```

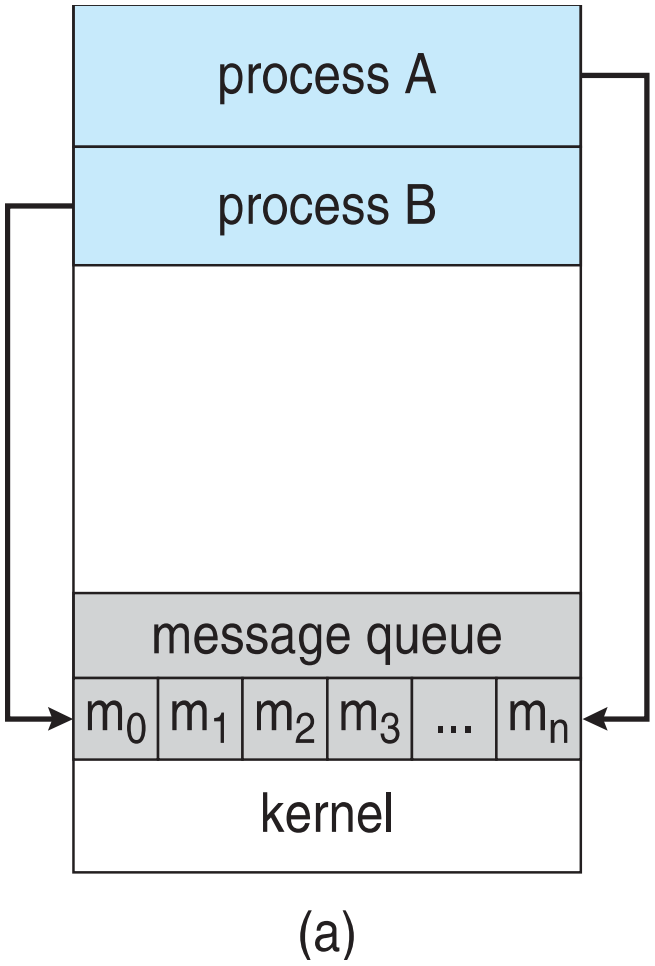
- Se nenhum processo pai em espera (não invocou ***wait()***) o processo se torna um **zumbi**
- Se o pai foi finalizado sem chamar a espera, o processo é **órfão**

Comunicação entre Processos (IPC)

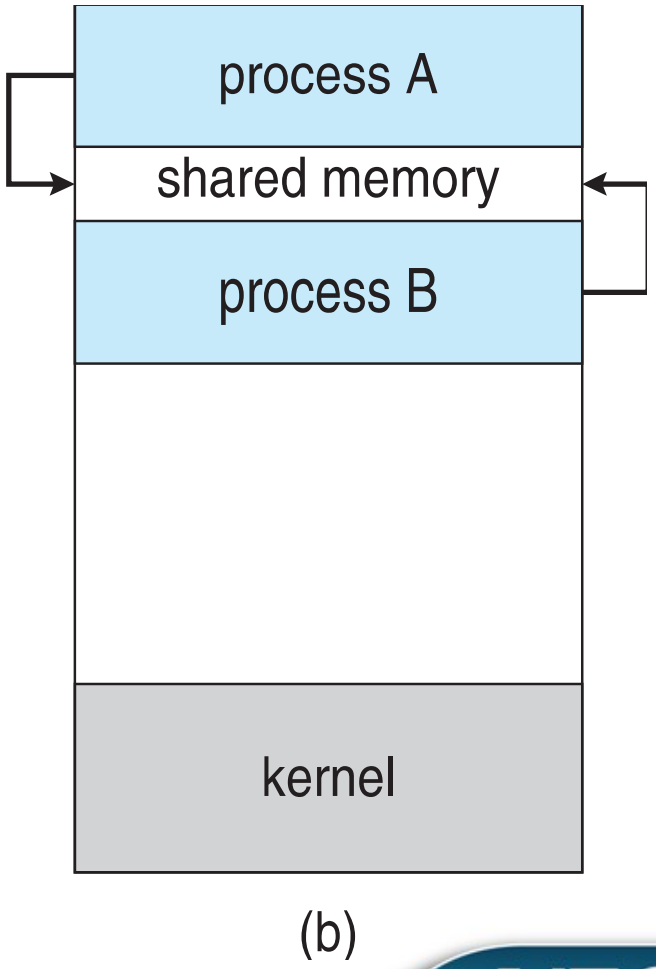
- Processos em um sistema podem ser **Independentes** ou **Cooperantes**
- Processos **Independentes** não podem afetar ou ser afetados pela execução de outro processo.
- Processos **Cooperantes** podem afetar ou ser afetados pela execução de outro processo
- Razões para cooperação entre processos:
  - Compartilhamento de Informações
  - Aumento na velocidade da computação
  - Modularidade
  - Conveniência
- Processos cooperantes precisam de **Comunicação entre Processos** (**IPC** – *interprocess communication*)
- Dois modelos de IPC: **memória compartilhada** e **troca de mensagens**

Modelos de Comunicações

(a) Troca de mensagens



(b) Memória compartilhada



Problema do Produtor-Consumidor

- Paradigma para processos cooperantes, processo *produtor* produz informação que é consumida por um processo *consumidor*.
  - Buffer de tamanho ilimitado *(unbounded-buffer)* não coloca limite prático no tamanho do buffer.
  - Buffer de tamanho fixo (*bounded-buffer*) assume que existe um tamanho fixo do buffer.

Solução Buffer Tamanho Fixo - Memória Compartilhada

- Dados compartilhados em uma fila circular de items (um *buffer*)

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solução está correta, mas somente pode usar BUFFER_SIZE-1 elementos

Buffer Tamanho Fixo – Produtor

```
item proximo_item_produzido;
while (true) {
     /* Produz um item */
     while (((in + 1) % BUFFER_SIZE)== out)
            ;   /* não faz nada - sem buffers livres*/
     /* Insere o próximo item no buffer */
     buffer[in] = proximo_item_produzido;
     in = (in + 1) % BUFFER_SIZE;
}
```

Buffer Tamanho Fixo – Consumidor

```
item proximo_item_consumido;
while (true) {
      while (in == out)
            ; /*não faz nada - nada para consumir*/
      /* remove um item do buffer */
      proximo_item_consumido = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      /* consome o item */
 }
```

Comunicação entre Processos – Memória Compartilhada

- A comunicação ocorre por uma área de memória compartilhada entre os processos que desejam se comunicar
- A comunicação está sob o controle dos processos dos usuários e não do sistema operacional.
- O principal problema é fornecer um mecanismo que permita que os processos do usuário sincronizem suas ações quando acessarem a memória compartilhada.
- Porém é o SO que provê primitivas para a criação e associação de áreas de memória compartilhada aos processos (***shget()***, ***shctl()***, ***shmat()*** e ***shmdt()*** no UNIX/Linux)

Comunicação entre Processos – Troca de Mensagens

- Mecanismo para processos se comunicarem e sincronizarem suas ações.
- Sistema de mensagens – processos se comunicam uns com os outros sem utilização de variáveis compartilhadas.
- Suporte a IPC (*InterProcess Communication*) provê duas operações uma para envio outra para recebimento:
  - **send**(*mensagem*) – tamanho da mensagem fixo ou variável
  - **receive**(*mensagem*)
- Se *P* e *Q* querem se comunicar, eles necessitam:
  - Estabelecer um *link de comunicação* entre eles
  - Trocar mensagens via send/receive
- Implementação de links de comunicação
  - Físico (ex. Memória compartilha, barramento de hardware)
  - Lógico (ex. Propriedades lógicas)

Questões de Implementação

- Como são estabelecidas as ligações?
- Pode um link estar associado com mais de dois processos?
- Quantos links podem existir entre cada par de processos comunicantes?
- Qual a capacidade de um link?
- O tamanho da mensagem utilizado pelo link é fixo ou variável?
- O link é unidirecional ou bidirecional?

Possibilidades de Implementação

- Implementação do link de comunicação
- Fisica:
  - Memória compartilhada
  - Barramento de hardware
  - Rede
- Lógica:
  - Direto ou indireto
  - Síncrono ou assíncrono
  - Armazenamento em buffer automático ou explícito

Comunicação Direta

- Processos devem nomear o outro explicitamente:
  - **send** (*P, mensagem*) – envia uma mensagem ao processo P
  - **receive**(*Q, mensagem*) – recebe uma mensagem do processo Q

- Propriedades dos links de comunicação
  - Links são estabelecidos automaticamente.
  - Um link é associado com exatamente um par de processos comunicantes.
  - Entre cada par de processos existe exatamente um link.
  - O link pode ser unidirecional, mas é usualmente bidirecional.

Comunicação Indireta

- Mensagens são dirigidas e recebidas de caixas postais – *mailboxes* (também chamadas de portas).
  - Cada *mailbox* possui uma única identificação.
  - Processos podem se comunicar somente se eles compartilham a *mailbox*.

- Propriedades do link de comunicação:
  - O link é estabelecido somente se os processos compartilham uma *mailbox* comum
  - Um link pode estar associado com muitos processos.
  - Cada par de processos pode compartilhar vários links de comunicação.
  - Link pode ser unidirecional ou bidirecional.

Comunicação Indireta (Cont.)

- Operações
  - Criar uma nova caixa postal
  - Enviar e receber mensagens através da caixa postal
  - Destruir uma caixa postal

- Primitivas são definidas como:

  send(*A, mensagem*) – envia uma mensagem para a caixa postal A

  receive(*A, mensagem*) – recebe uma mensagem da caixa postal A

Comunicação Indireta (Cont.)

- Compartilhamento de Caixa Postal
  - P_1, P_2, e P_3 compartilham caixa postal A.
  - P_1, envia; P_2 e P_3 recebem.
  - Quem recebe a mensagem?
- Soluções:
  - Permitir que um link esteja associado com no máximo dois processos.
  - Permitir somente a um processo de cada vez executar uma operação de recebimento.
  - Permitir ao sistema selecionar arbitrariamente por um receptor. Remetente é notificado de quem foi o receptor.

Sincronização

- Troca de Mensagens pode ser bloqueante ou não-bloqueante
- **Bloqueante** é considerado **síncrono**
  - **Envio (send) bloqueante** inibe o remetente até que a mensagem seja recebida
  - **Recepção (receive) bloqueante** inibe o receptor até uma mensagem estar disponível
- **Não-Bloqueante** é considerado **assíncrono**
  - **Envio não-bloqueante** o remetente envia a mensagem e continua executando
  - **Recepção não-bloqueante** o receptor obtém uma mensagem válida ou nula

Sincronização (cont...)

- O problema produtor-consumidor se torna trivial:

```
message next_produced;
while (true) {
      /* produce an item in next produced */
      send(next_produced);
}

message next_consumed;
while (true) {
    receive(next_consumed);
    /* consume the item in next consumed */
}
```

Bufferização

- Fila de mensagens associada ao link; implementada em uma dentre três formas.
  1. Capacidade Zero – 0 mensagens
     Remetente deve esperar pelo receptor (*rendezvous*).
  2. Capacidade Limitada – tamanho finito de n mensagens
     Remetente deve aguardar se link está cheio.
  3. Capacidade Ilimitada – tamanho infinito
     Remetente nunca espera.

Exemplos de Sistemas IPC - POSIX

- Memória Compartilhada no POSIX
  - Processo cria primeiro um segmento de memória compartilhado

```
segment id = shmget(IPC PRIVATE, size, S IRUSR
  | S IWUSR);
```

  - Processo que deseja acesso a essa memória compartilhada deve se anexar a ela

```
shared memory = (char *) shmat(id, NULL, 0);
```

  - Agora o processo pode escrever na memória compartilhada

```
sprintf(shared memory, "Writing to shared
  memory");
```

  - Quando terminar, um processo pode desanexar a memória compartilhada do seu espaço de armazenamento

```
shmdt(shared memory);
```

Exemplos de Sistemas IPC - Mach

- Comunicação no Mach é baseado em mensagens
  - Até mesmo chamada de sistemas são mensagens
  - Cada tarefa obtém duas *mailboxes* na criação - *Kernel* e *Notify*
  - Somente três chamadas de sistemas são necessárias para transferência de mensagens

`msg_send(), msg_receive(), msg_rpc()`

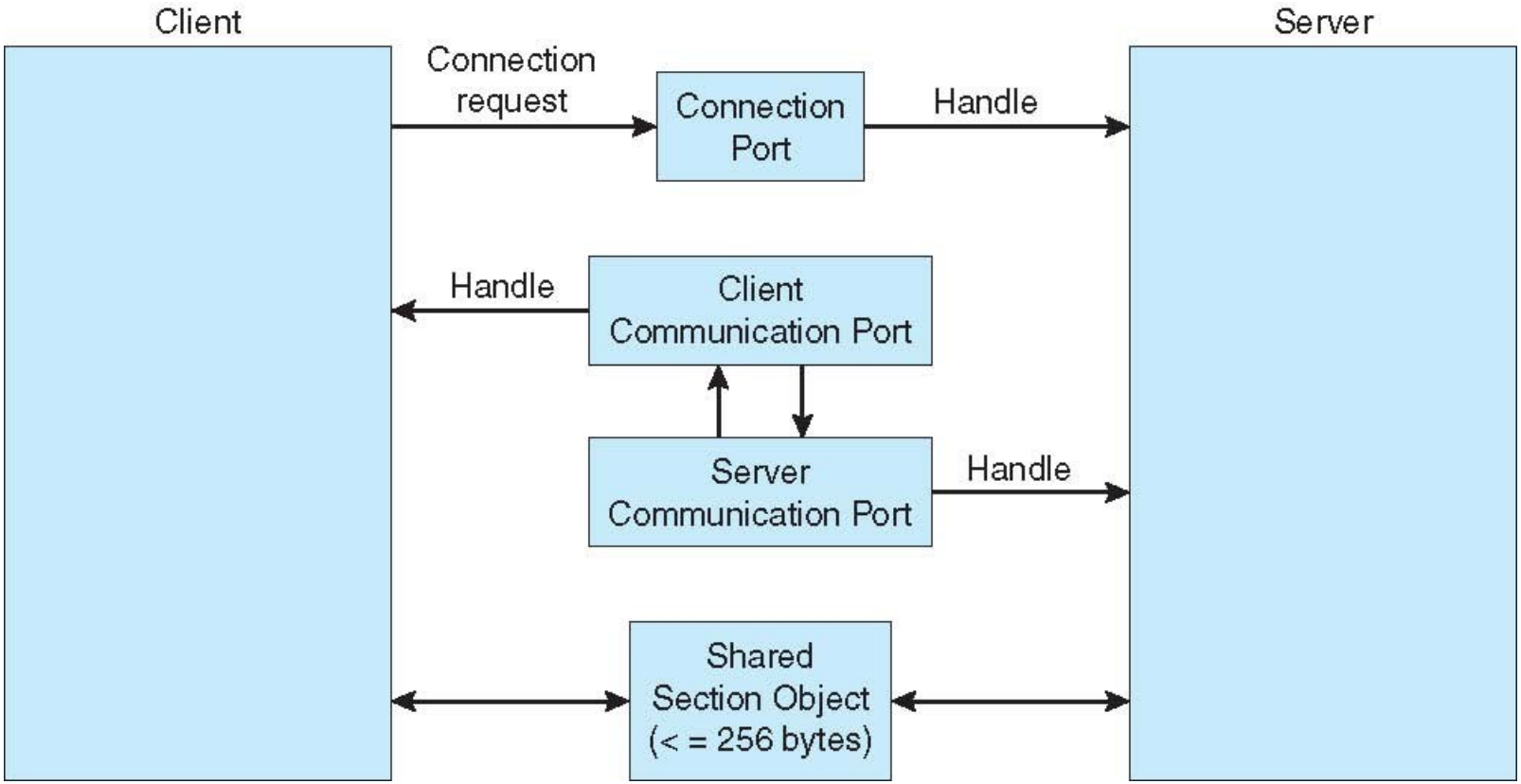
  - *Mailboxes* necessárias para comunicação, criadas via

`port_allocate()`

Exemplos de Sistemas IPC – Windows

- Recurso de troca de mensagens é chamado de ***local procedure call*** **(LPC)**
  - Só funciona entre processos no mesmo sistema
  - Usa portas (como *mailboxes*) para estabelecer e manter canais de comunicação
  - Comunicação funciona da seguinte forma:
    - O cliente abre um manipulador para o objeto porta de conexão do subsistema.
    - O cliente envia uma solicitação de conexão.
    - O servidor cria duas portas de comunicação privadas e retorna o manipulador de uma delas para o cliente.
    - O cliente e o servidor usam o manipulador da porta correspondente para enviar mensagens ou retornos de chamadas e ouvir respostas.

Local Procedure Calls no Windows



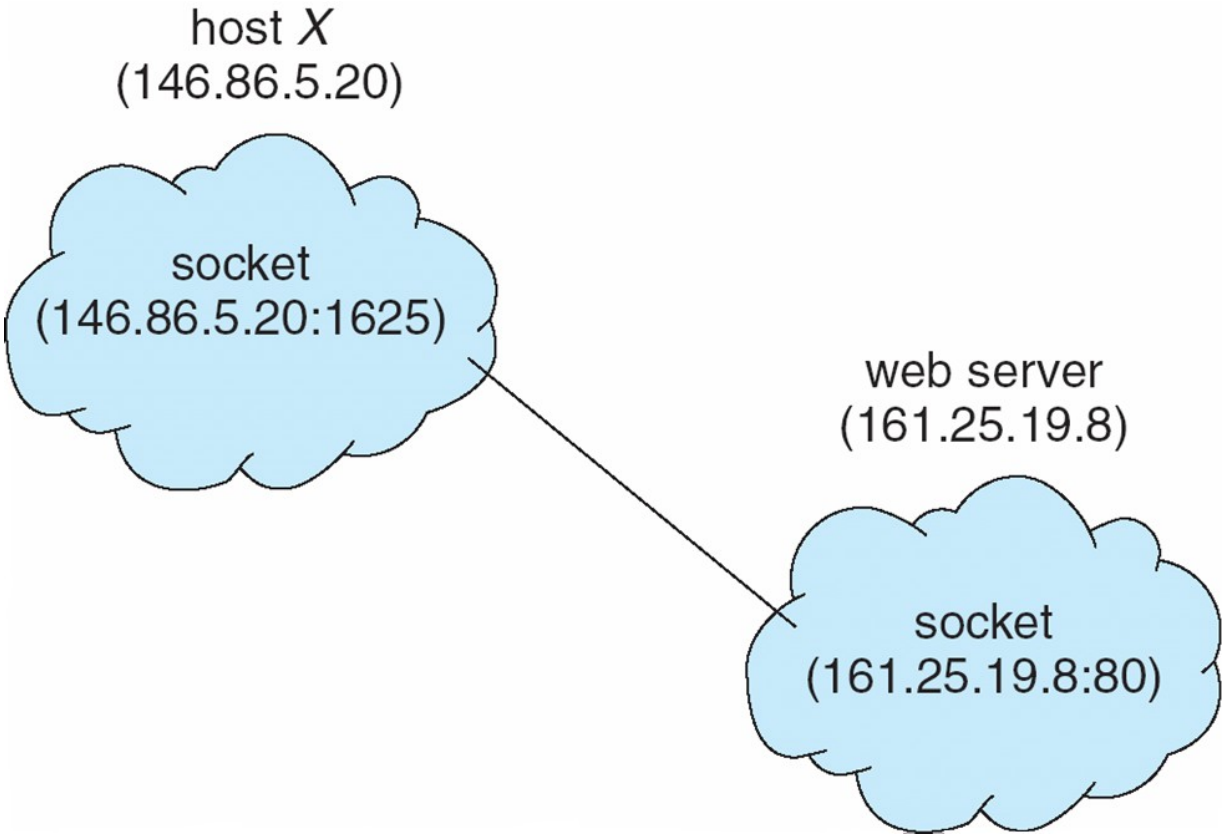
Comunicação Cliente-Servidor

- Sockets
- Pipes
- Chamada a Procedimento Remoto (RPC)
- Invocação Remota de Método (RMI em Java)

Sockets

- Um socket é definido como um ponto final de comunicação
- Concatenação de um endereço IP e porta
- O socket **161.25.19.8:1625** refere a porta **1625** na máquina **161.25.19.8**
- Comunicação ocorre entre um par de sockets
- Portas abaixo de 1024 são consideradas well-known ports e usadas para serviços padrão
- Endereço IP especial 127.0.0.1 (loopback)

Comunicação com Socket



Pipes

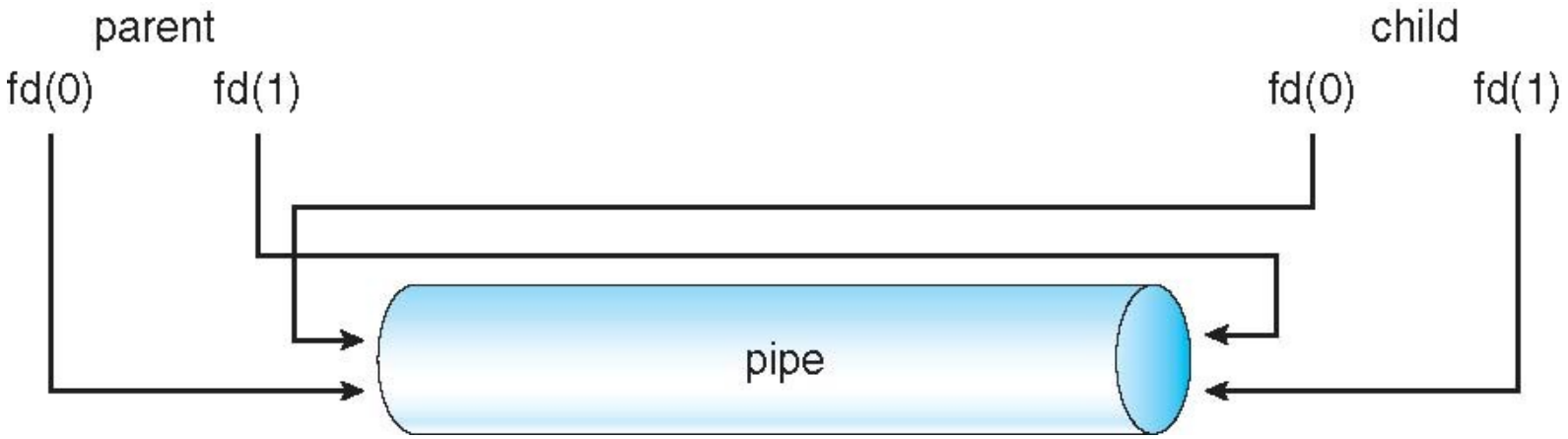
- Agem como canalizações permitindo a comunicação entre dois processos

- Questões
  - A comunicação é unidirecional ou bi-direcional?
  - No caso da comunicação de duas vias, ela é half ou full-duplex?
  - Existe uma relação (ex. Pai-filho) entre os processos comunicantes?
  - É possível usar pipes em uma rede?

Pipes Comuns

- **Pipes comuns** permitem a comunicação no estilo produtor-consumidor
- Produtor escreve em um extremo (o extremo de escrita do pipe)
- Consumidor lê do outro extremo (o extremo de leitura do pipe)
- Pipes comuns são unidrecionais
- Necessitam de relação pai-filho entre os processos comunicantes

Pipes Comuns



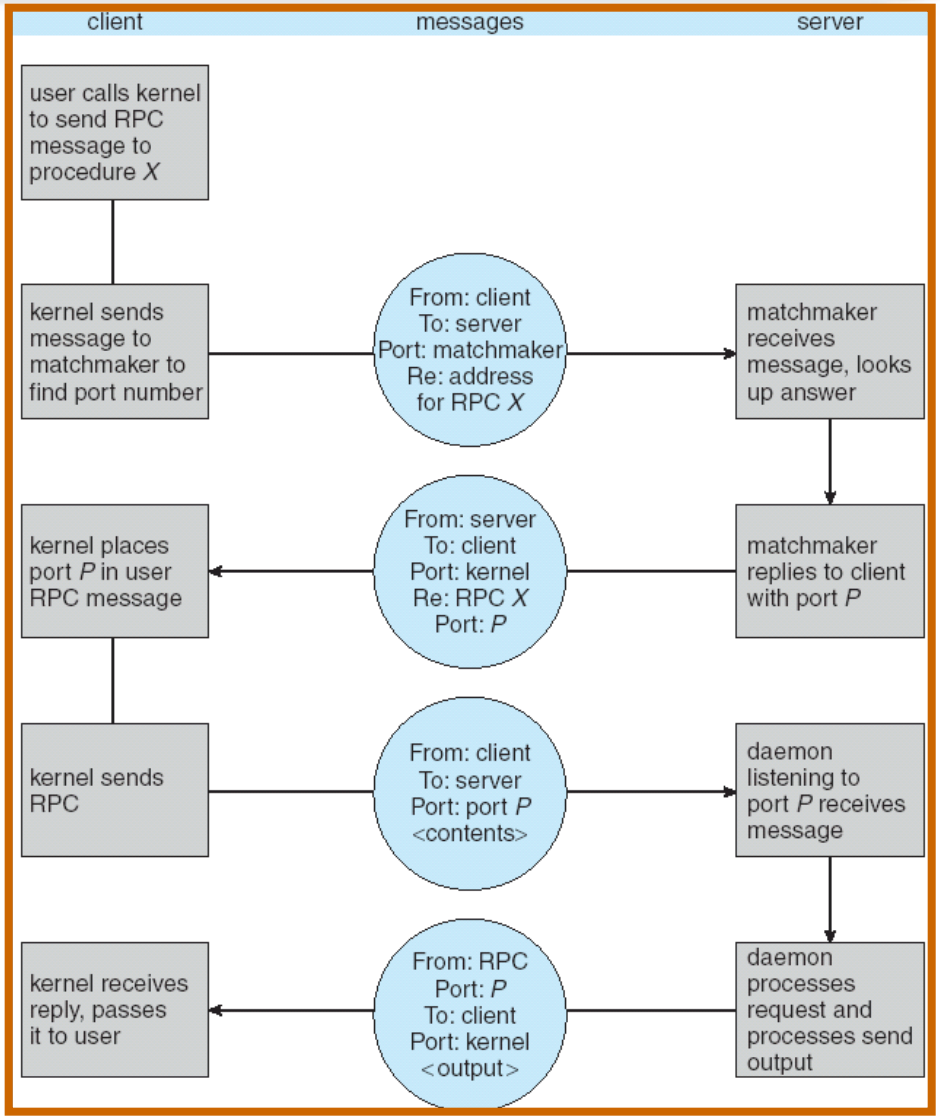
Pipes Nomeados

- Pipes Nomeados são mais poderosos que pipes comuns
- Comunicação é bi-direcional
- Não é necessária relação pai-filho entre processos comunicantes
- Vários processos podem usar os pipes noemados para se comunicarem
- Fornecidos nos sistemas UNIX e Windows

Chamada a Procedimento Remoto (RPC)

- Chamada a Procedimento Remoto ou *Remote procedure call* (RPC) abstrai chamadas de procedimentos entre processos executando nos sistemas em rede.
- ***Stubs*** – procedimento *proxy* no lado do cliente para o procedimento real no servidor.
- O *stub* no lado do cliente localiza o servidor e empacota (*marshall*) os parâmetros.
- O *stub* no lado do servidor recebe esta mensagem, desempacota os parâmetros e dispara a execução do procedimento no servidor
- No Windows, o código *stub* compila a partir de uma especificação em MIDL (*Microsoft Interface Definition Language*)

Execução de RPC



Invocação Remota de Método

- Invocação Remota de Método ou *Remote Method Invocation* (RMI) é um mecanismo Java similar a RPC.
- RMI permite a um programa Java executando em uma máquina invocar um método em um objeto remoto.