

INE5412 Sistemas Operacionais I

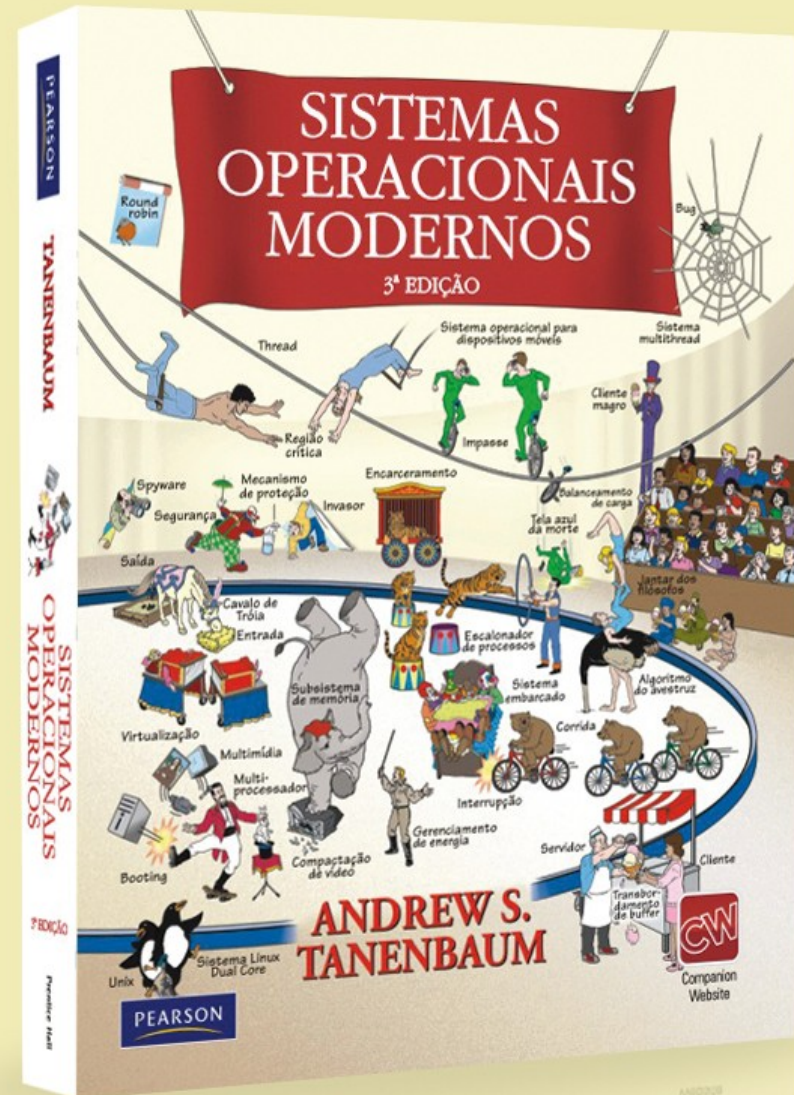
L. F. Friedrich

Capítulo 2
Gerência de Processos – Parte 1

Sistemas operacionais modernos

Terceira edição
ANDREW S. TANENBAUM

Capítulo 2 Processos e Threads



Multiprogramação

- Tornar mais eficiente o aproveitamento dos recursos do computador
- Execução simultânea* de vários programas
 - Diversos programas são mantidos na memória
 - Conceitos necessários a multiprogramação
 - Processo
 - Interrupção
 - Proteção entre processos

(*) Na realidade a execução é feita de forma concorrente (máquinas monoprocessadas)!!

Fração de Espera da CPU = $w = b/e+b$, onde e =tempo de execução e b =tempo de E/S. Ex. $e=10s$ $b=9s \implies w=9/10 = 0.9$

Qual aumento de eficiência com Multiprogramação? $w' \approx w^2$

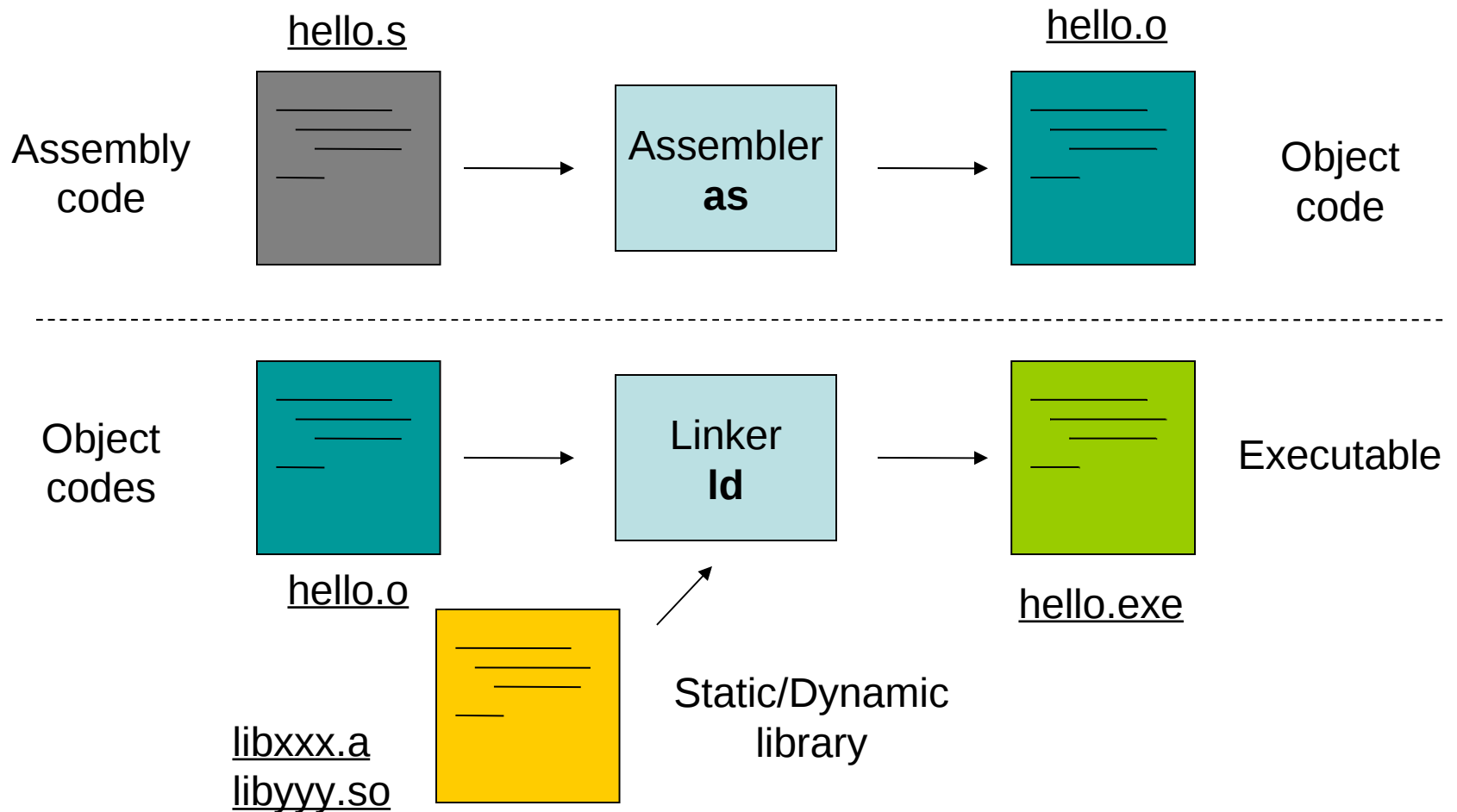
Conceito de Processo

- Diferenciação entre o programa e sua execução
- Programa:
 - Entidade estática e permanente
 - Sequência de instruções
 - Passivo sob o ponto de vista do sistema operacional
- Processo:
 - Entidade dinâmica e efêmera
 - Altera seu estado a medida que avança sua execução
 - Composto por:
 - Programa
 - Dados
 - Contexto (valores)

O que é um programa?

- Um programa é apenas código.
- Que tipo de código?
 - Linguagem alto nível: C, C++, etc.
 - Linguagem baixo nível: código assembly.
 - Um quase executavel: código objeto.
 - Executavel: contém apenas código de máquina.

Construindo um programa (1)



O que é um processo?

- Uma resposta simples: um programa executando.
- Uma resposta melhor:
 - Um processo é a **execução de uma instância** de um programa.
 - Mais de um processo podem executar o código do mesmo programa.
 - Um processo é uma entidade ativa.
 - Um processo tem seu **estado local** de execução.
 - O estado local muda durante a execução.

O modelo de processo

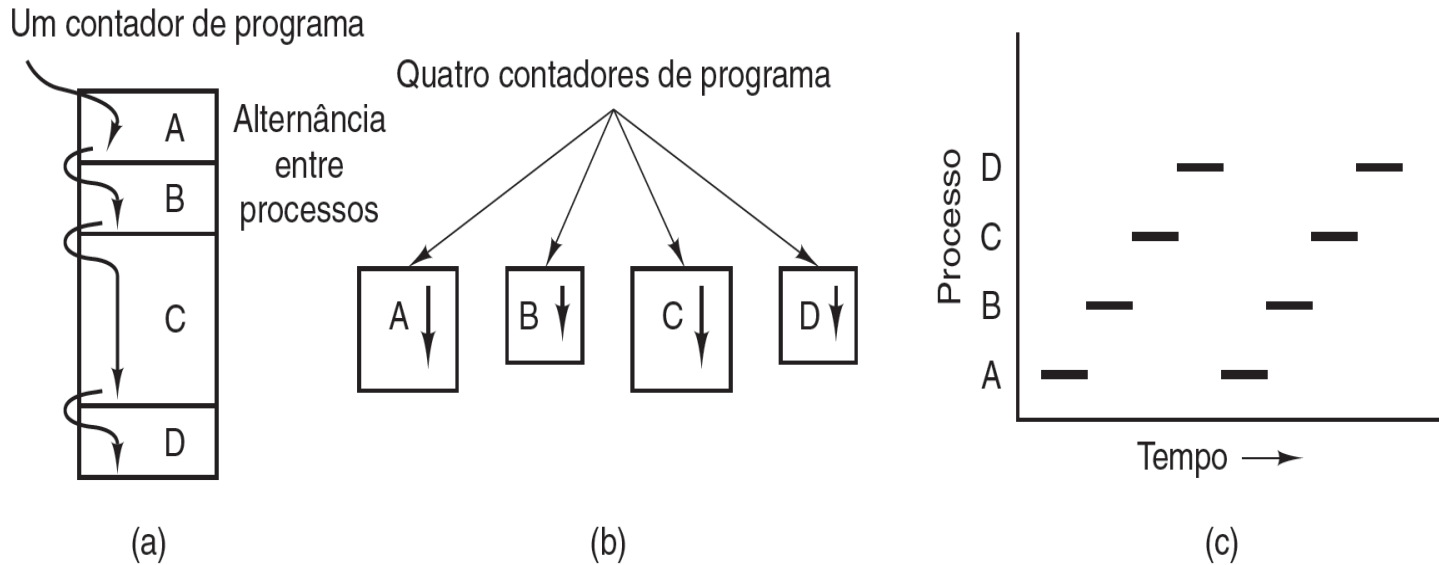
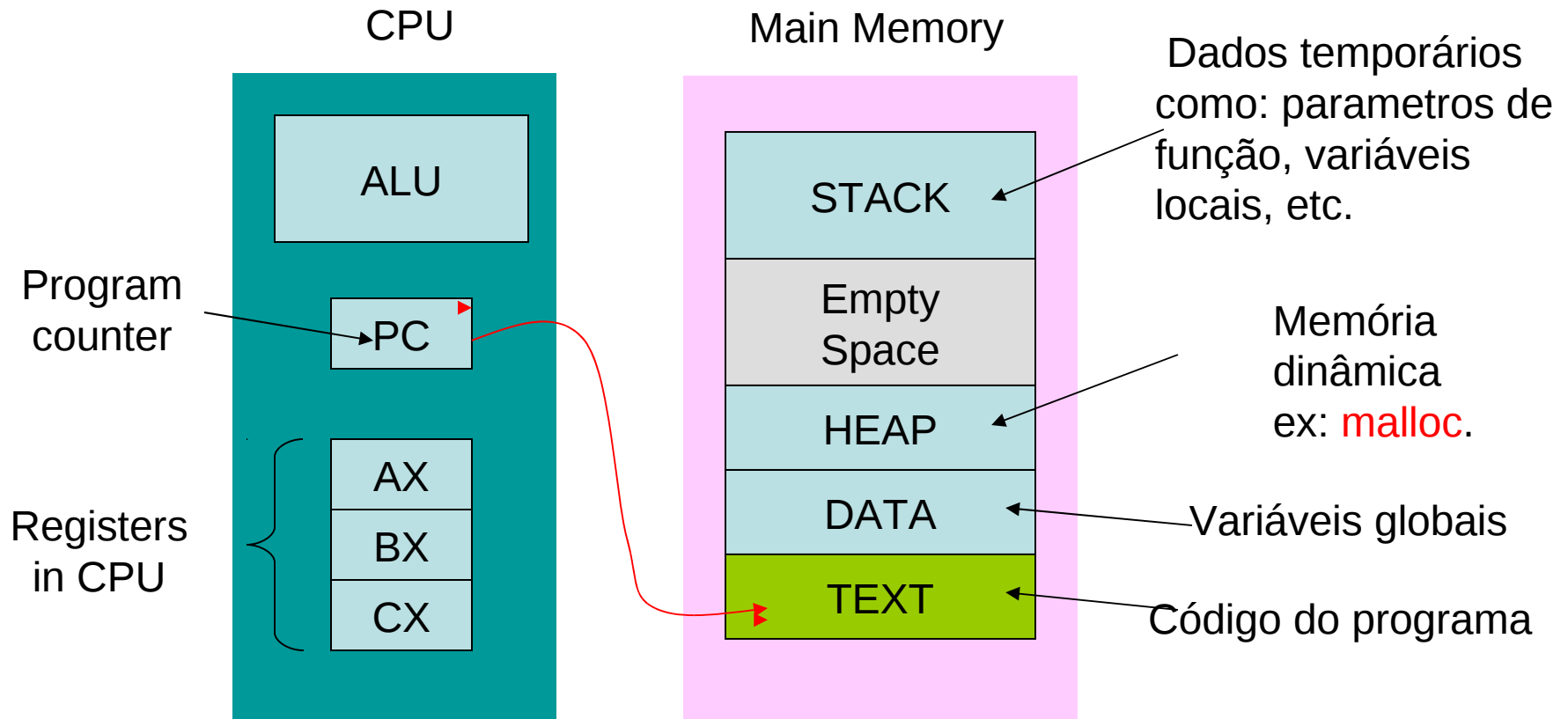
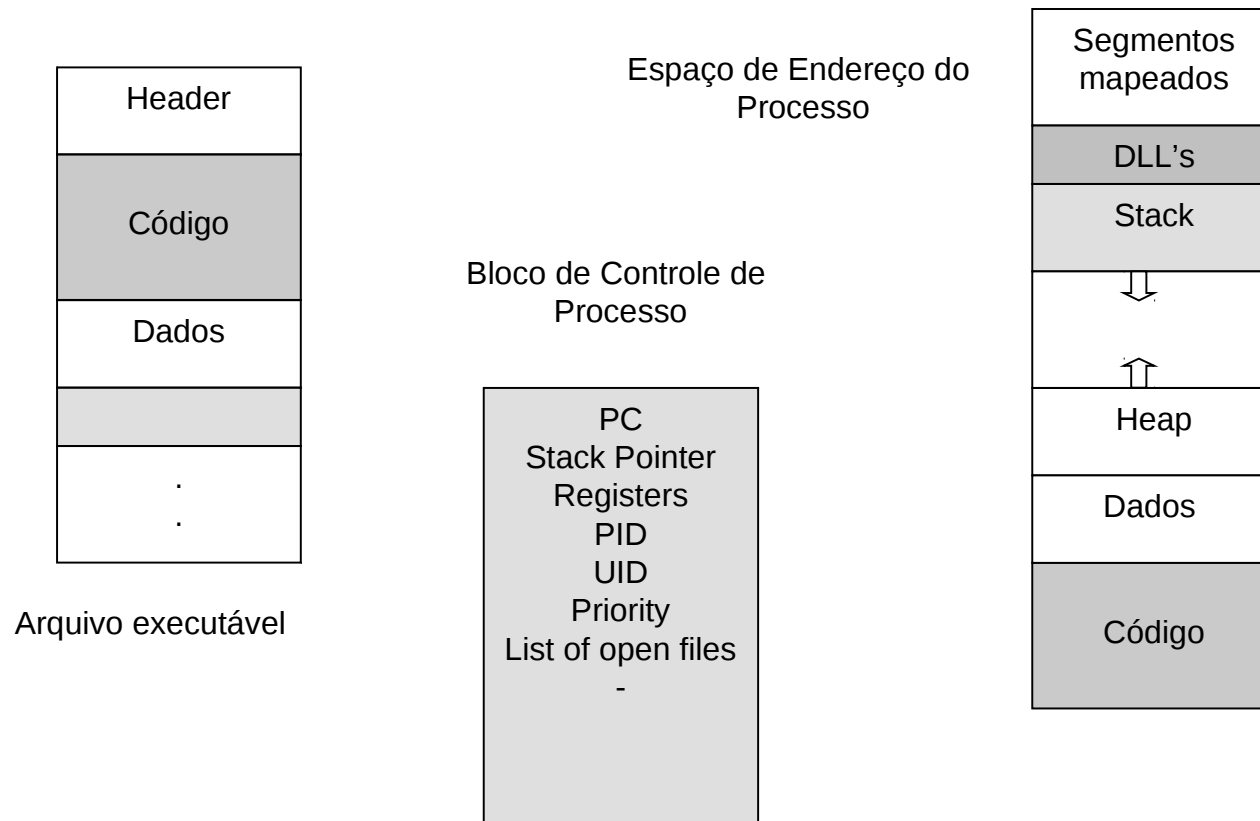


Figura 2.1 (a) Multiprogramação de quatro programas. (b) Modelo conceitual de quatro processos sequenciais independentes. (c) Somente um programa está ativo a cada momento.

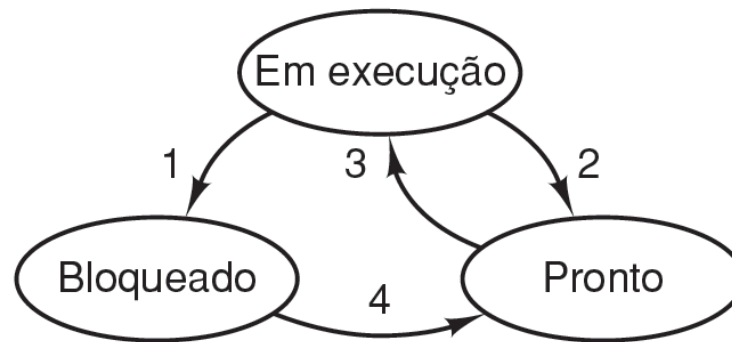
Programa executando (fisicamente)



Anatomia de um Processo



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

Figura 2.2 Um processo pode estar nos estados em execução, bloqueado ou pronto. As transições entre esses estados são mostradas.

Processos Suspenso

- Processador é mais rápido que E/S assim todos processos poderiam estar esperando E/S
- Move (Swap) estes processos para disco para liberar mais memória
- o estado bloqueado torna-se estado suspenso quando processo movido para disco
- 2 novos estados
 - Bloqueado, suspenso
 - Pronto, suspenso



UNIX: Estados do Processo

User Running	Executing in user mode.
Kernel Running	Executing in kernel mode.
Ready to Run, in Memory	Ready to run as soon as the kernel schedules it.
Asleep in Memory	Unable to execute until an event occurs; process is in main memory (a blocked state).
Ready to Run, Swapped	Process is ready to run, but the swapper must swap the process into main memory before the kernel can schedule it to execute.
Sleeping, Swapped	The process is awaiting an event and has been swapped to secondary storage (a blocked state).
Preempted	Process is returning from kernel to user mode, but the kernel preempts it and does a process switch to schedule another process.
Created	Process is newly created and not yet ready to run.
Zombie	Process no longer exists, but it leaves a record for its parent process to collect.

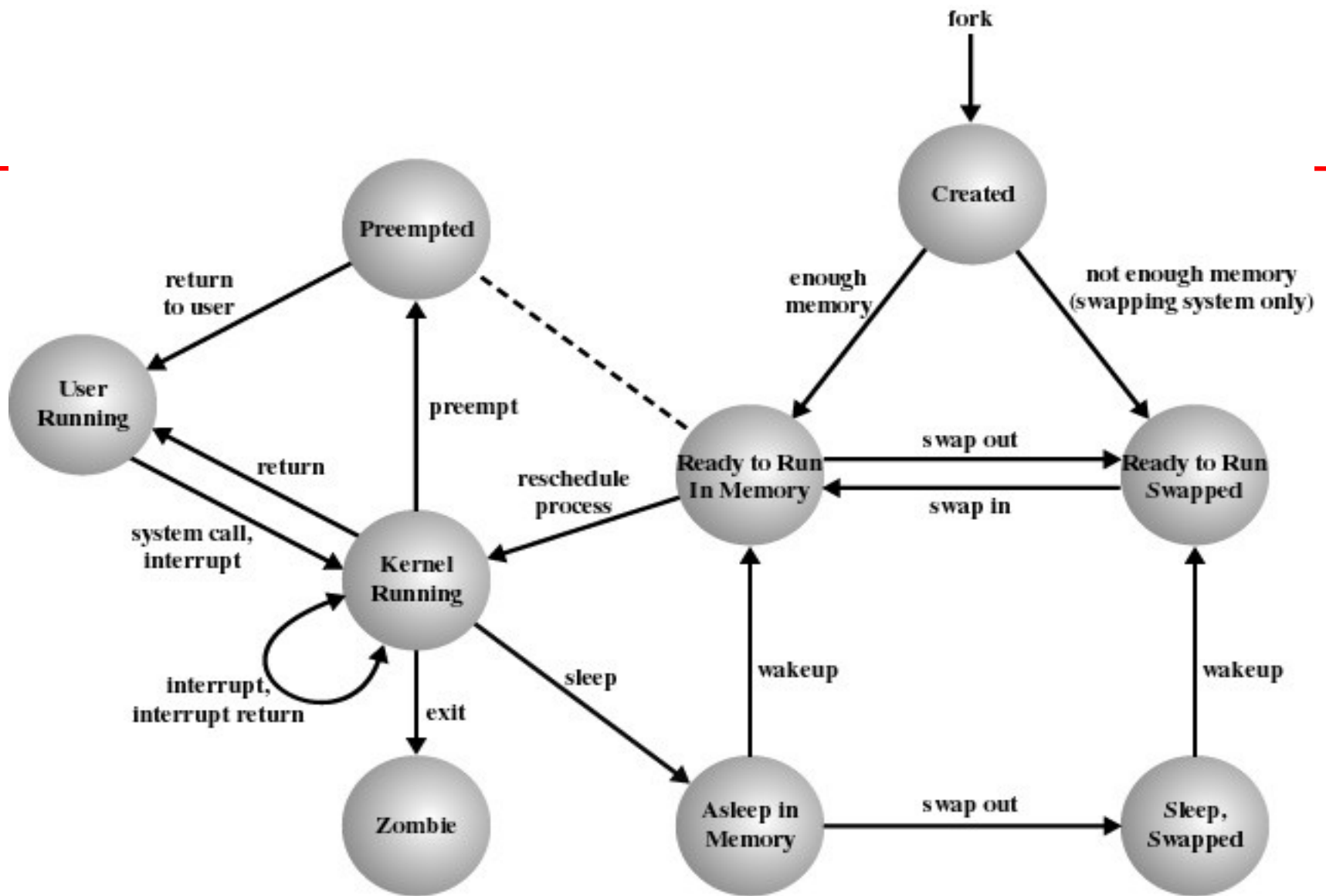
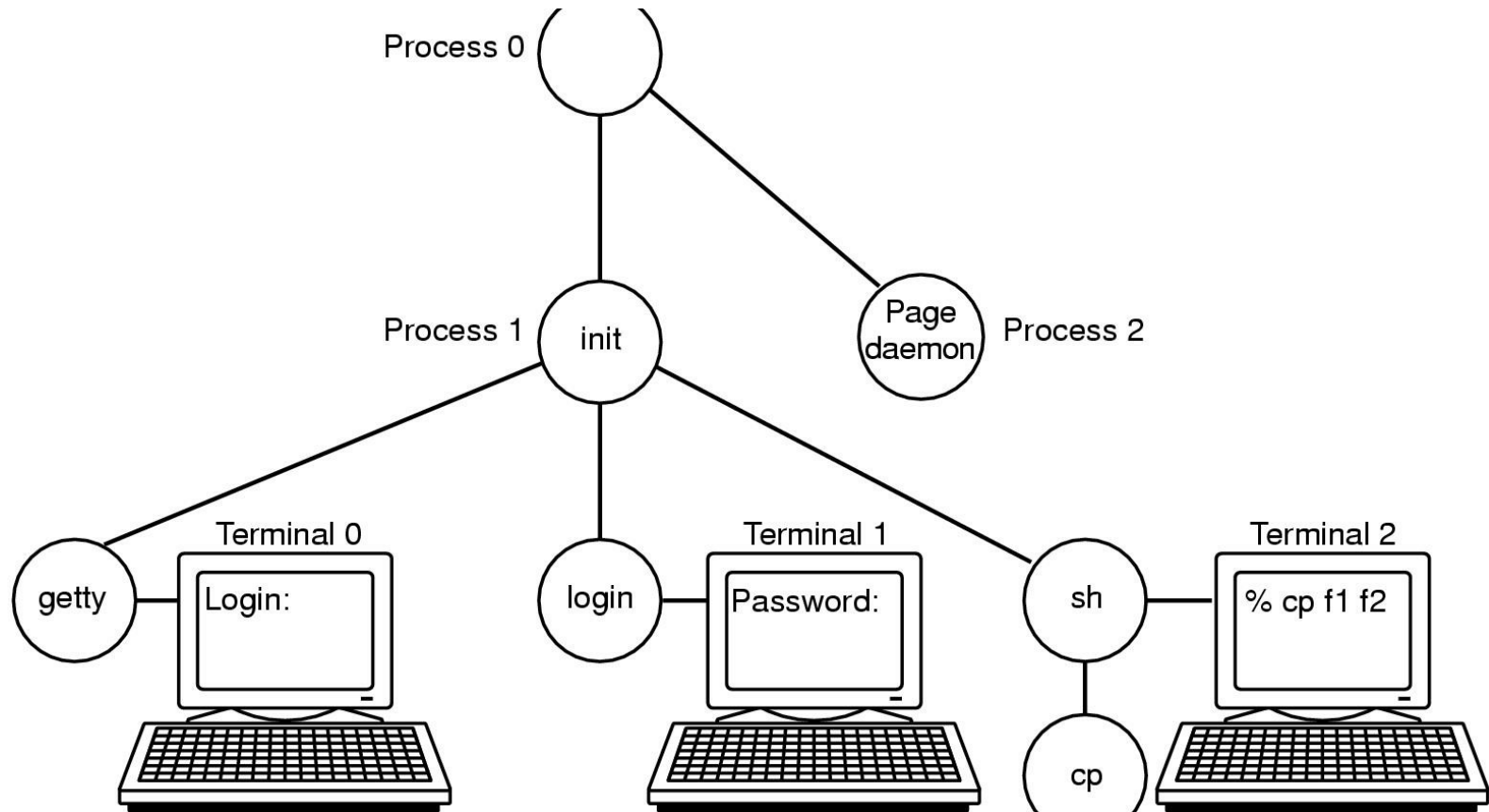


Figure 3.16 UNIX Process State Transition Diagram

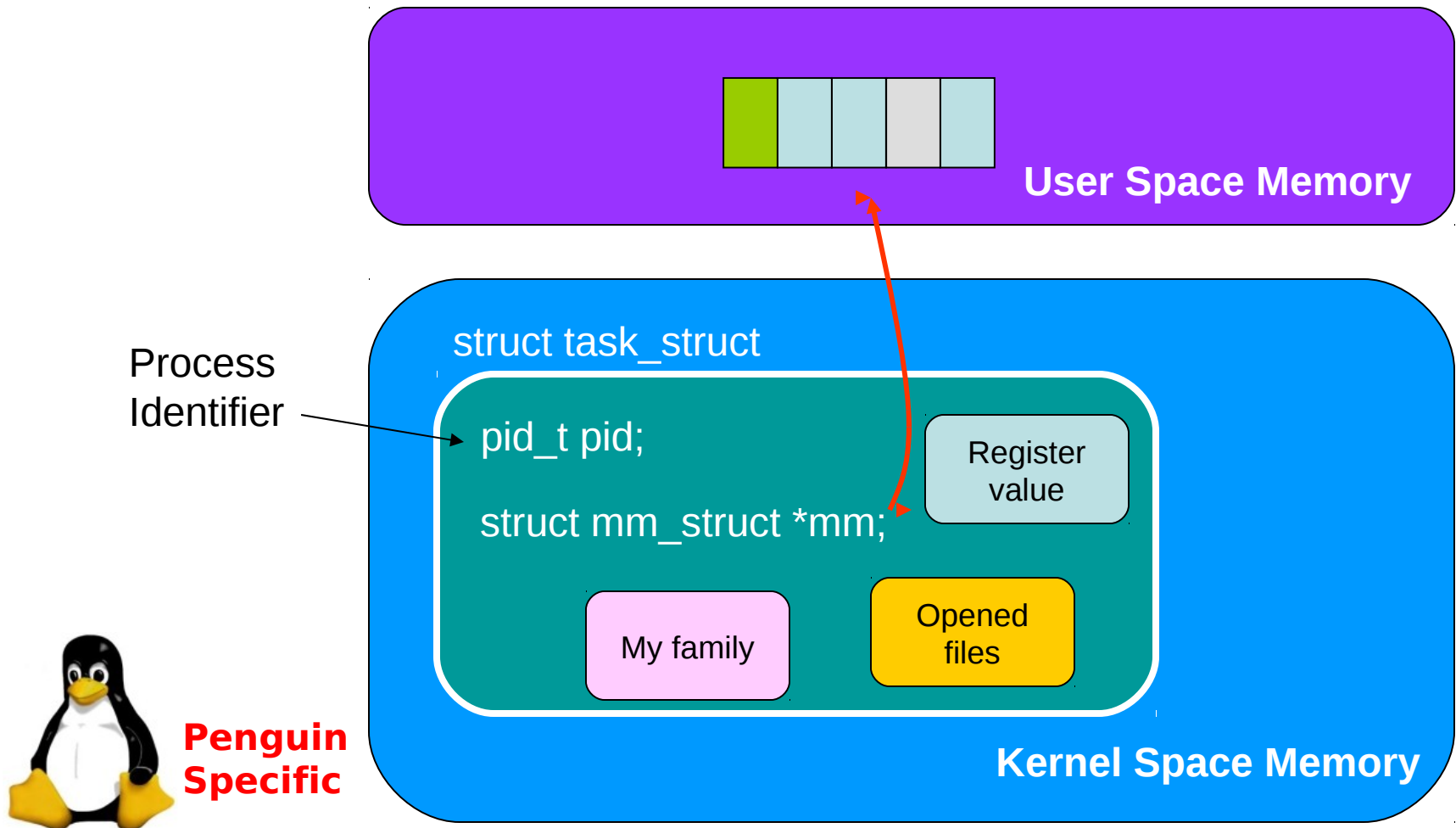
Hierarquia de Processos

- 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

Hierarquia de Processos



Programa executando (no kernel)

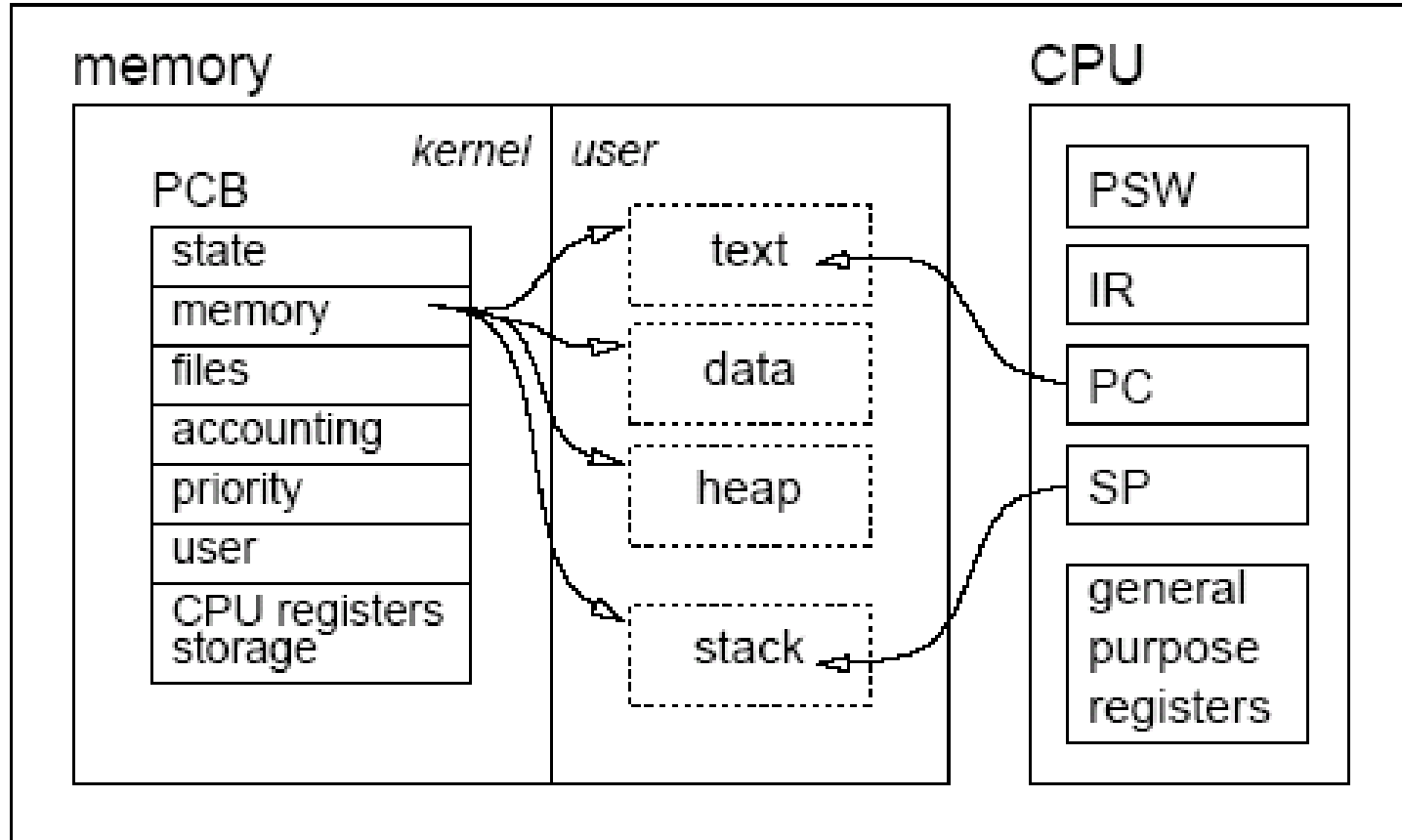


Implementação de Processos (1)

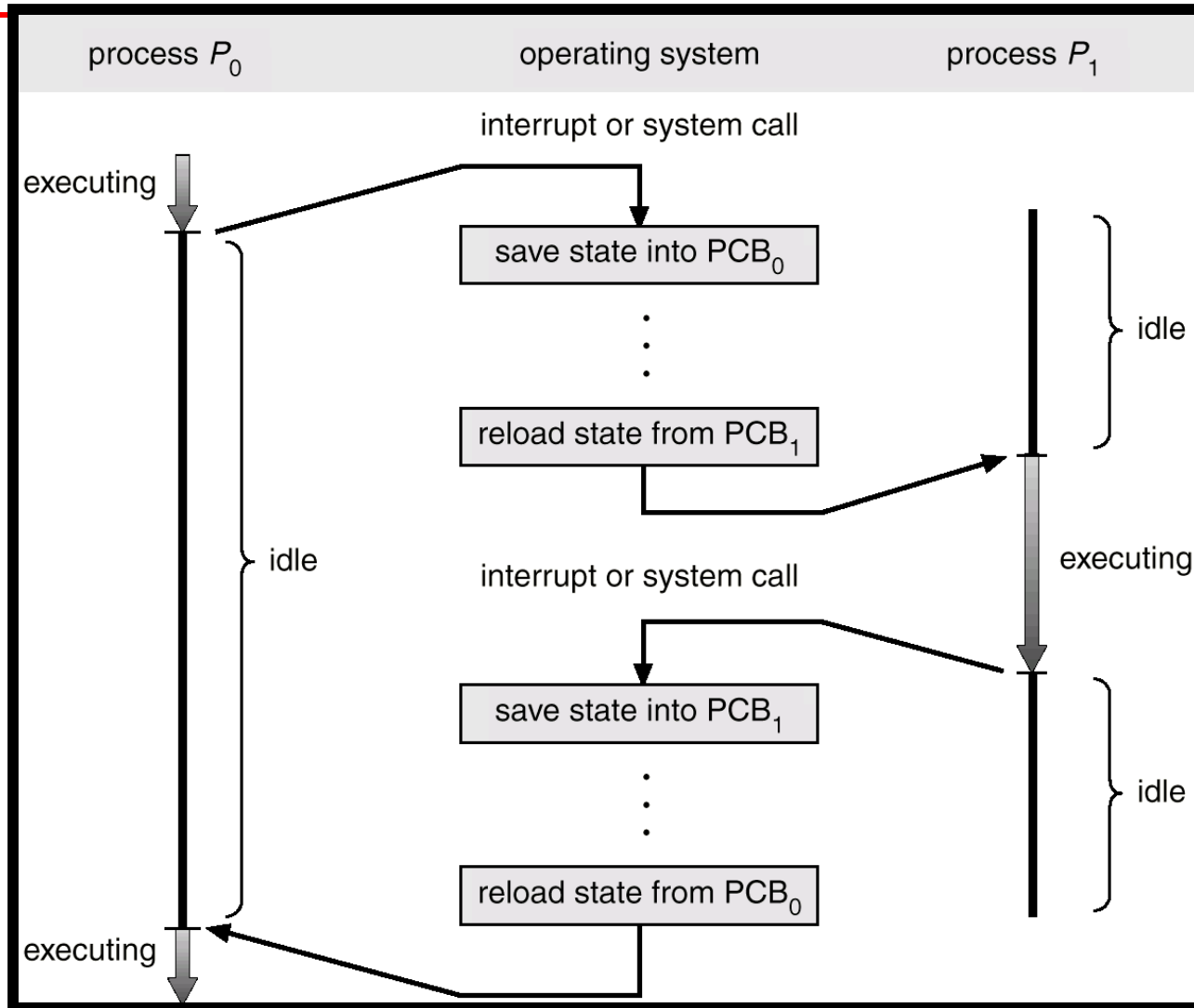
Gerenciamento de processo	Gerenciamento de memória	Gerenciamento de arquivo
Registros Contador de programa Palavra de estado do programa Ponteiro da pilha Estado do processo Prioridade Parâmetros de escalonamento ID do processo Processo pai Grupo de processo Sinais Momento em que um processo foi iniciado Tempo de CPU usado Tempo de CPU do processo filho Tempo do alarme seguinte	Ponteiro para informações sobre o segmento de texto Ponteiro para informações sobre o segmento de texto Ponteiro para informações sobre o segmento de texto	Diretório-raiz Diretório de trabalho Descritores de arquivo ID do usuário ID do grupo

■ **Tabela 2.1** Alguns dos campos de um processo típico de entrada na tabela.

Bloco de Controle de Processo



Contexto dos Processos



Quando “Chavear” Contexto de um Processo

- Interrupção de Relógio (Clock)
 - processo executou o máximo tempo (time slice) permitido
- Interrupção de E/S
- Término de um Processo
- Excessão
 - erro ocorrido
 - pode causar o término do processo
- Chamada de sistema (Supervisor)
 - como open de arquivo

Manipulação de Processos

- Manipulação básica de processos envolve : criação, carga de programa, terminação, ...
- Exemplo: API UNIX (POSIX)
 - Criação: `fork()`,
 - Carga de programa: `exec()`,
 - Espera: `wait()`,
 - Término: `exit()`
 - Sinalização de processos: `kill()`
 - Controle de processos: `ptrace()`, `nice()`, `sleep()`

Chamadas de sistema para gerenciamento de processo no Linux

Chamada de sistema	Descrição
<code>pid = fork()</code>	Cria um processo filho idêntico ao pai
<code>pid = waitpid(pid, &statloc, opts)</code>	Espera o processo filho terminar
<code>s = execve(name, argv, envp)</code>	Substitui a imagem da memória de um processo
<code>exit(status)</code>	Termina a execução de um processo e retorna o status
<code>s = sigaction(sig, &act, &oldact)</code>	Define a ação a ser tomada nos sinais
<code>s = sigreturn(&context)</code>	Retorna de um sinal
<code>s = sigprocmask(how, &set, &old)</code>	Examina ou modifica a máscara do sinal
<code>s = sigpending(set)</code>	Obtém o conjunto de sinais bloqueados
<code>s = sigsuspend(sigmask)</code>	Substitui a máscara de sinal e suspende o processo
<code>s = kill(pid, sig)</code>	Envia um sinal para um processo
<code>residual = alarm(seconds)</code>	Ajusta o relógio do alarme
<code>s = pause()</code>	Suspende o chamador até o próximo sinal

Tabela 10.3 Algumas chamadas ao sistema relacionadas com processos. O código de retorno *s* é -1 quando ocorre um erro, *pid* é o ID do processo e *residual* é o tempo restante no alarme anterior. Os parâmetros são aqueles sugeridos pelos próprios nomes.

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

Criação de Processo - ações

- Atribui um identificador único ao processo (PID)
- Aloca espaço para o processo
- Inicializa o BCP
- Prepara ligações apropriadas
 - Ex: adiciona novo processo na lista ligada usada para fila de escalonamento
- Cria ou expande outras estruturas de dados
 - Ex: mantém um arquivo de contabilidade

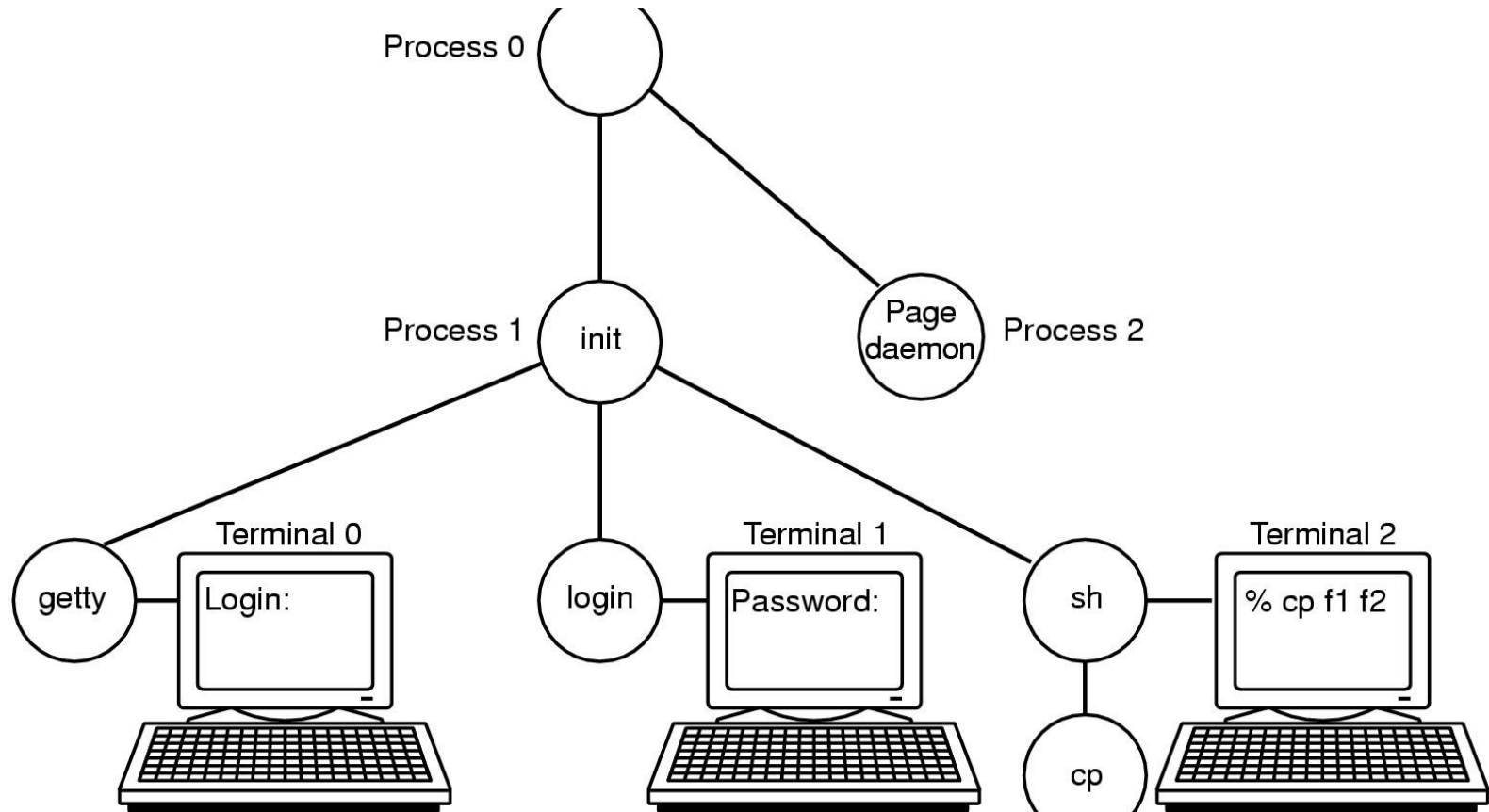
Criação de Processo - implementação

- Processo pai cria processos filhos, que, podem criar outros processos, formando uma árvore de processos.
- Recursos compartilhados
 - Pai e filho compartilham todos recursos.
 - Filho compartilha parte dos recursos do pai.
 - Pai e filho não compartilham recursos.
- Execução
 - Pai e filho executam de forma concorrente.
 - Pai espera filho terminar.
- Espaço de endereços
 - Filho é duplicata do pai.
 - Filho carrega um programa

Criação de Processo - Unix

- Sistema cria o primeiro processo, processo 0 (sysproc), que cria outros
- No Unix, o segundo processo, processo 1, é chamado *init*
 - Cria todos os *gettys* (processo *login*) e daemons
 - Processo nunca morre
 - Controla configuração do sistema (num. de processos, prioridades,)
- Interface POSIX inclui uma chamada para criação de processos
 - **fork()**

Criação de Processo - Unix



Criação de Processo – Unix fork()

- Cria um processo filho que herda:
 - Cópia idêntica de variáveis e memória do pai
 - Cópia idêntica de todos os registradores
- Pai e filho executam no mesmo ponto após o fork():
 - Para o filho, fork() retorna 0
 - Para o pai, fork() retorna o identificador do processo filho
- Implementação do **fork()**:
 - Aloca memória para o processo filho
 - Copia memória e registradores do pai para o filho
 - Custo alto!!

Como fork funciona

- Inicio do programa

```
int main(void)
{
    puts("before fork ...");

    if(fork() == 0)
        puts("I'm the child.");
    else
        puts("I'm the parent.");

    puts("program terminated.");
}
```

before fork ...

—

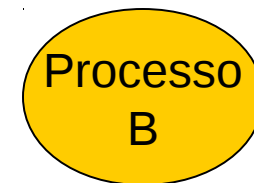
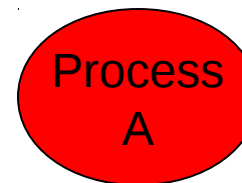
Processo
A

Como fork funciona

- **fork()** cria um novo processo- Processo B.

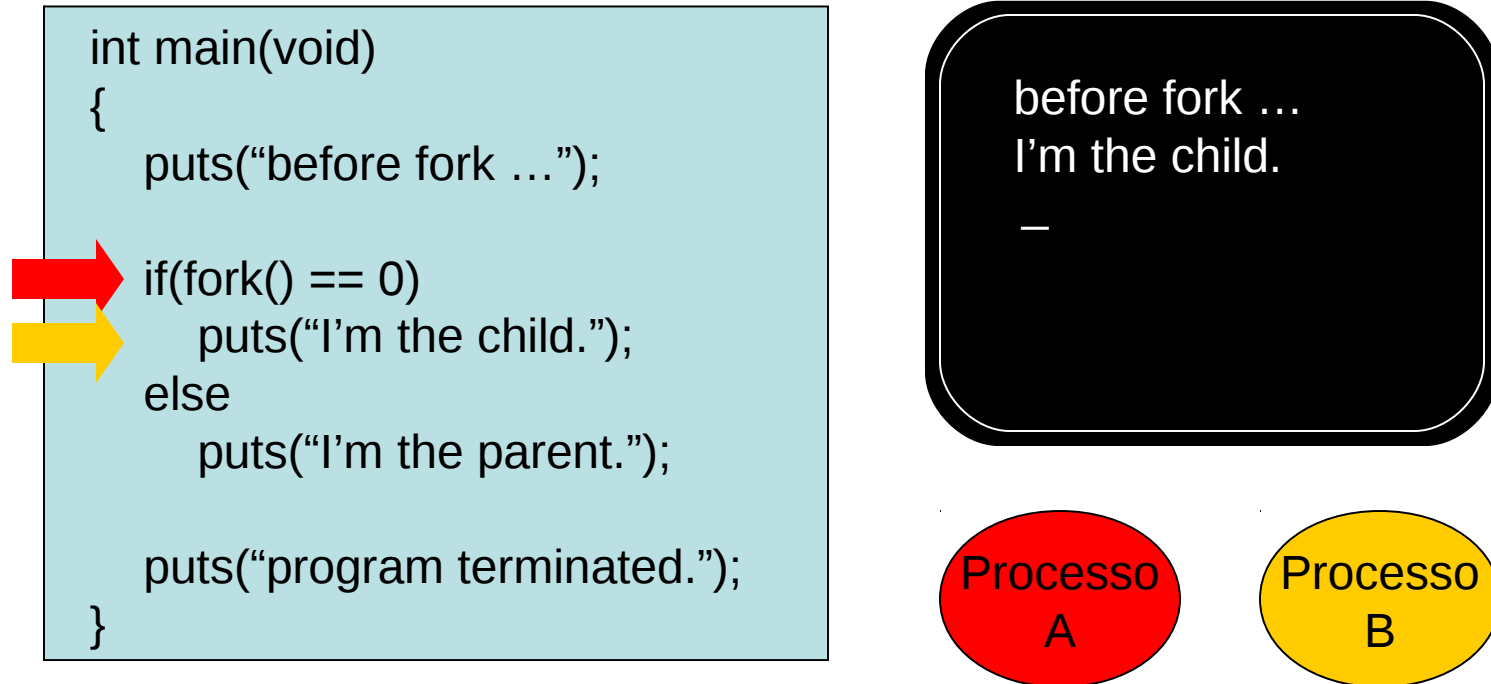
```
int main(void)
{
    puts("before fork ...");
    if(fork() == 0)
        puts("I'm the child.");
    else
        puts("I'm the parent.");

    puts("program terminated.");
}
```



Como fork funciona

- Como só tem uma CPU, apenas um processo irá executar. O **escalonador** pega o Processo B neste caso.



Como fork funciona

- Processo B – termina.

```
int main(void)
{
    puts("before fork ...");
    if(fork() == 0)
        puts("I'm the child.");
    else
        puts("I'm the parent.");

    puts("program terminated.");
}
```

before fork ...
I'm the child.
program terminated.
—

Processo
A

Processo
B


Como fork funciona

- Processo A é escalonado pelo escalonador.

```
int main(void)
{
    puts("before fork ...");

    if(fork() == 0)
        puts("I'm the child.");
    else
        puts("I'm the parent.");

    puts("program terminated.");
}
```



before fork ...
I'm the child.
program terminated.
I'm the parent.
—

Processo
A

Como fork funciona

- Processo A – termina.

```
int main(void)
{
    puts("before fork ...");

    if(fork() == 0)
        puts("I'm the child.");
    else
        puts("I'm the parent.");

    puts("program terminated.");
}
```



before fork ...
I'm the child.
program terminated.
I'm the parent.
program terminated.

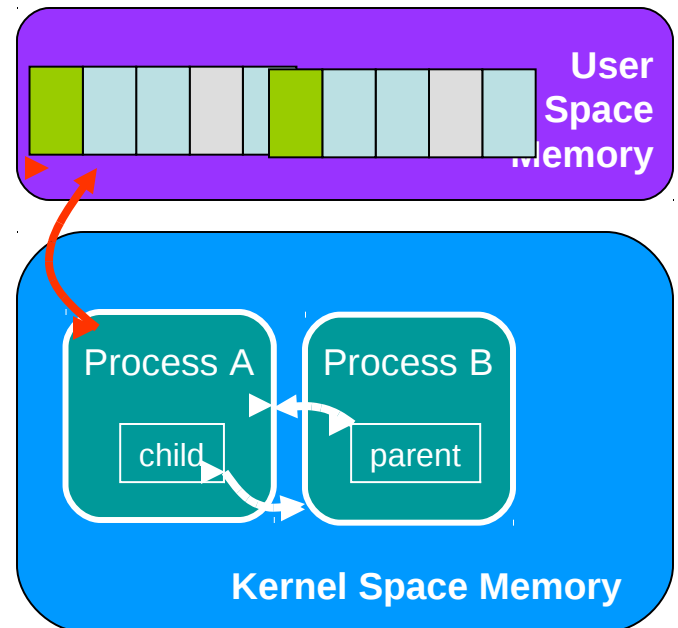
Processo
A

O que acontece no kernel?

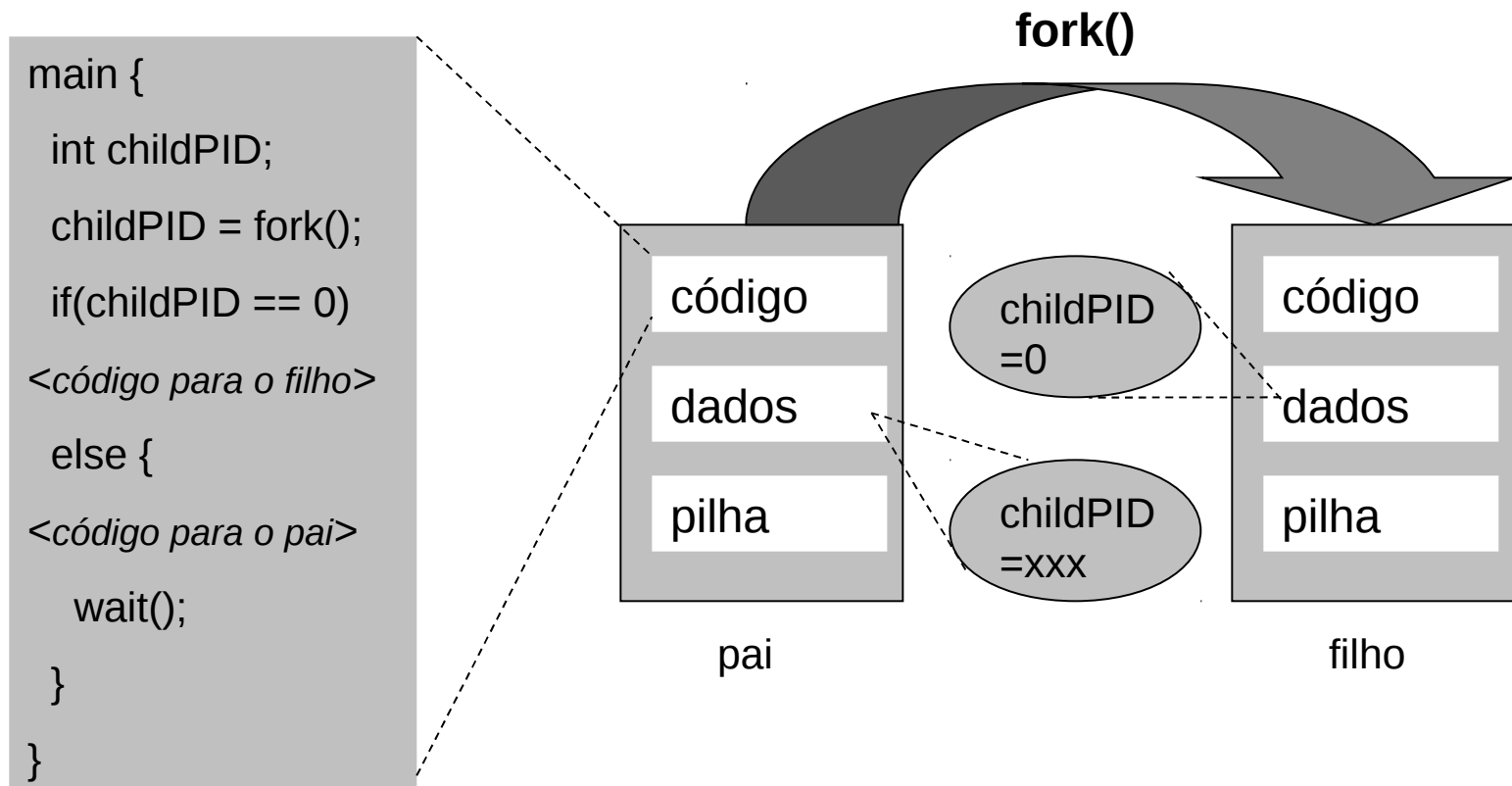
- `fork()` **copia** o código e memória do Processo A para Processo B.

```
int main(void)
{
    puts("before fork ...");
    if(fork() == 0)
        puts("I'm the child.");
    else
        puts("I'm the parent.");

    puts("program terminated.");
}
```



O que acontece no kernel?

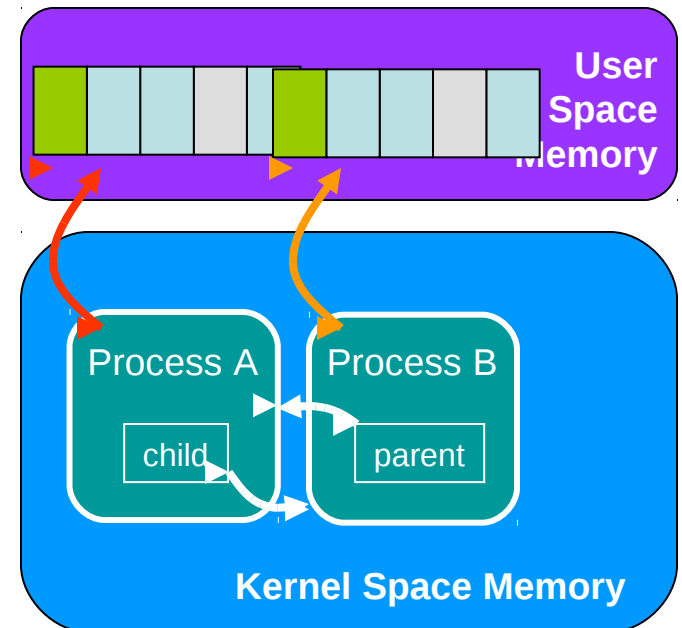



O que acontece no kernel?

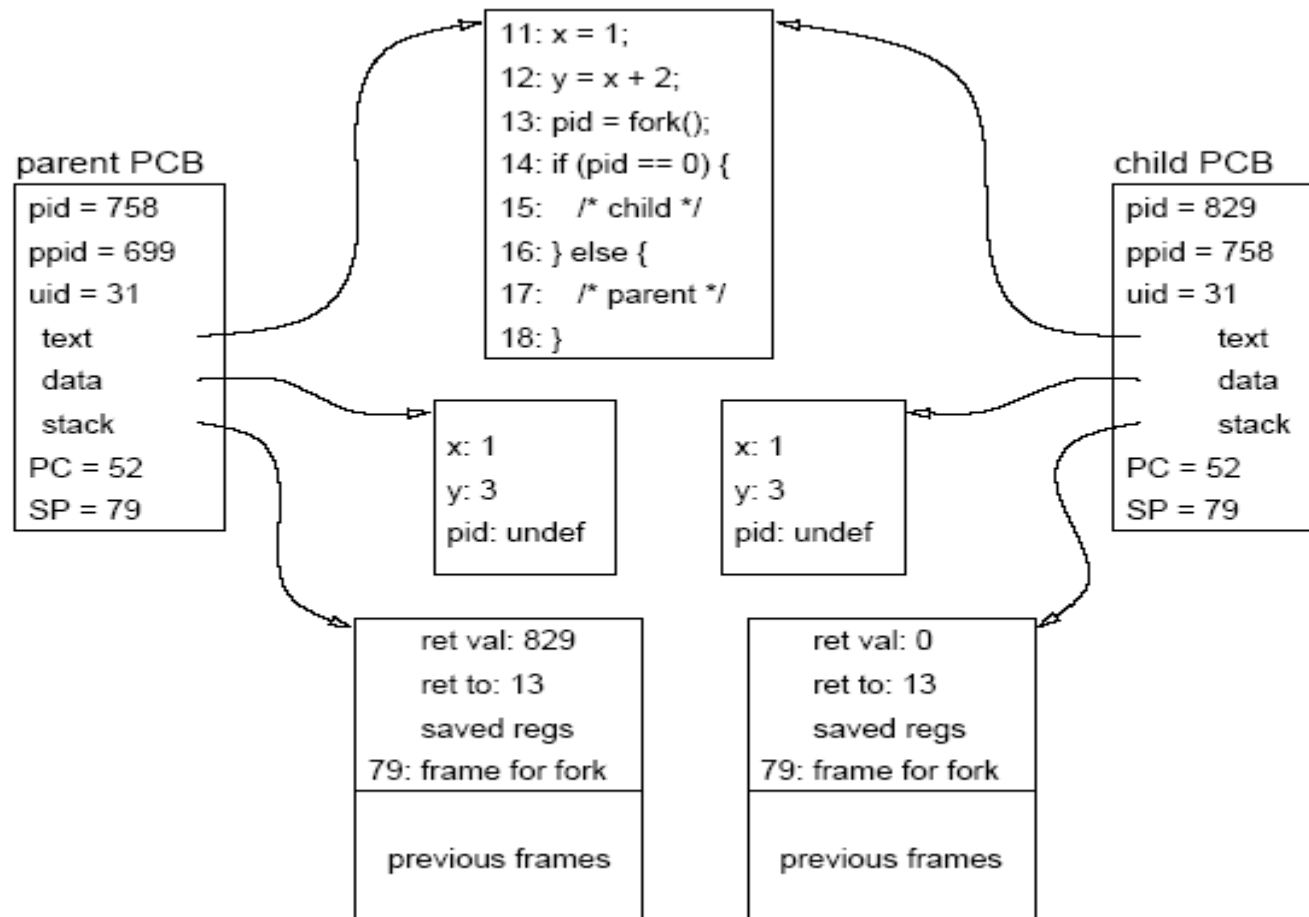
fork() também copia **contador de programa** do Processo A para Processo B.

```
int main(void)
{
    puts("before fork ...");
    if(fork() == 0)
        puts("I'm the child.");
    else
        puts("I'm the parent.");

    puts("program terminated.");
}
```



Fork



Fork - O Duplicador de Processos

- `fork()` duplica **quase tudo** do Processo A para o Processo B
 - Variáveis Globais, locais, memória alocada;
 - Lista de arquivos abertos;
- Exceto:
 - Identificador do processo;
 - relacionamentos do processo;
 - Tempo de execução & estado de execução.

Terminação de Processo

Condições de Terminação de Processos

- Saída normal (voluntária)
- Saída por erro (voluntária)
 - erro de proteção
 - exemplo write em um arquivo sómente de leitura(read-only file)
- Saída por erro fatal (não voluntária)
 - violação de limites(fronteiras)
 - erro aritmético
- Morto por outro processo (não voluntária)
 - requisição do pai