DCT2101 – Sistemas Operacionais
Processos e Threads

João Borges

Universidade Federal do Rio Grande do Norte – UFRN
Centro de Ensino Superior do Seridó – CERES
Departamento de Computação e Tecnologia – DCT

02 de março de 2020

Agenda

Multiprogramação

- Tornar mais eficiente o aproveitamento dos recursos do computador
- Execução "simultânea" de vários programas
  - Diversos programas são mantidos na memória
  - Conceitos necessários à multiprogramação
    - Processo
    - Interrupção
    - Proteção entre processos
- Próprio sistema operacional é um programa

O 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 (código), dados e contexto (valores)

O conceito de processo

- Abstração que representa um programa em execução
- Diferentes instâncias
  - Um programa pode ter várias instâncias em execução, i.e., diferentes processos
  - Mesmo código (programa) porém dados e momentos de execução (contexto) diferentes
- Forma pela qual o sistema operacional "enxerga" um programa e possibilita sua execução
- Processos executam:
  - Programas de usuários
  - Programas do próprio sistema operacional (daemons)

Ciclos de vida de um processo

- Criação
- Execução
- Término

Ciclos de vida de um processo

Criação

1. Momento da execução
2. Chamadas de sistemas
   - e.g.: fork, etc.
3. Podem ser associados a uma sessão de trabalho
   - e.g.: login de usuários: login + senha $\rightarrow$ shell (processo)
   - Identificado por um número único (PID)

Ciclos de vida de um processo

Execução

- Processos apresentam dois ciclos básicos de operação
  - Ciclo de processador
    - Tempo que ocupa a CPU
  - Ciclo de entrada e saída
    - Tempo em espera pela conclusão de um evento (e.g. E/S)
- Primeiro ciclo é sempre de processador
  - Trocas de ciclos por:
    - CPU $\rightarrow$ E/S: chamada de sistema
    - E/S $\rightarrow$ CPU: ocorrência de evento (interrupção)
- Tipos de Processos
  - CPU bound - Ciclo de processador $>$ ciclo de E/S
  - I/O bound - Ciclo de E/S $>$ ciclo de processador
  - Sem quantificação exata
  - Situação ideal:
    - Misturar processos CPU bound com I/O bound
    - Benefícios a nível de escalonamento

Ciclos de vida de um processo

Término

- Final de execução (normal)
- Por erros
  - e.g.: proteção, aritméticos, E/S, tentativa de execução de instruções inválidas, falta de memória, exceder tempo de limite
  - Intervenção de outros processos (kill)
  - Log off de usuários

Relacionamento entre processos

- Processos independentes
  - Não apresentam relacionamentos com outros processos
- Grupo de processos
  - Apresentam algum tipo de relacionamento
    - e.g. filiação
  - Podem compartilhar recursos
  - Definição de hierarquia

Relacionamento entre processos

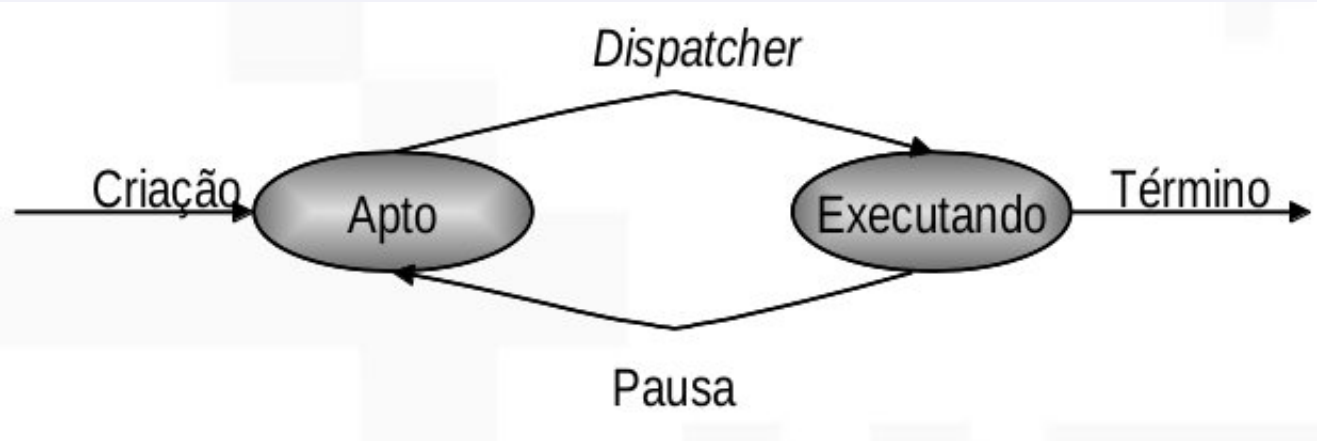
- Hierarquia de processos:
  - Processo criador é processo pai
  - Processo criado é processo filho
- Representação através de uma árvore (*pstree*)
  - Evolução dinâmica
- Semântica associada: O que fazer na destruição de um processo?
  - Toda a descendência "morre"
  - A descendência é herdada pelo processo "avô"
  - Postergar a destruição efetiva do processo pai até o final de todos processos filhos

Estados de um processo

- Após criado o processo necessita entrar em ciclo de processador
- Hipotéses:
  - Processador não está disponível
  - Vários processos sendo criados
- Que fazer?
  - Criação de uma fila de aptos (p/ espera pelo processador)

Modelo simplificado a dois estados

- Manter uma fila de processos aptos a executar
  - Esperando pelo processador ficar livre
- Escalonador (dispatcher):
  - Atribui o processador a um processo da fila de aptos
  - Pode prevenir um único processo de monopolizar o processador



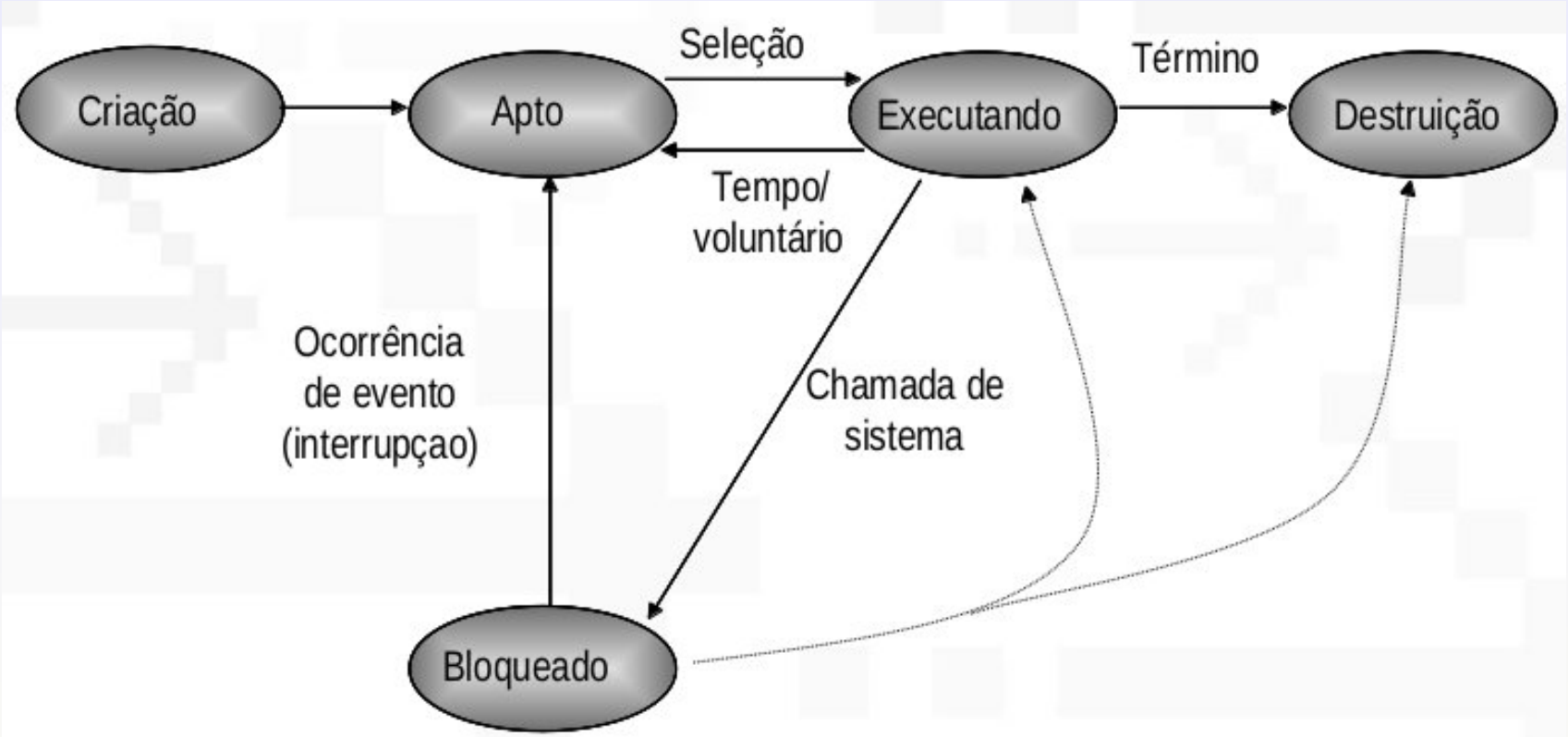
Limitação do modelo simplificado

- Causas para um processo não executar
  - Esperando pelo processador
    - Aptos para executar
  - Esperando pela ocorrência de eventos externos
    - Bloqueado
- Escalonador não pode selecionar um processo bloqueado, logo modelo a dois estados não é suficiente
  - Criação de novos estados

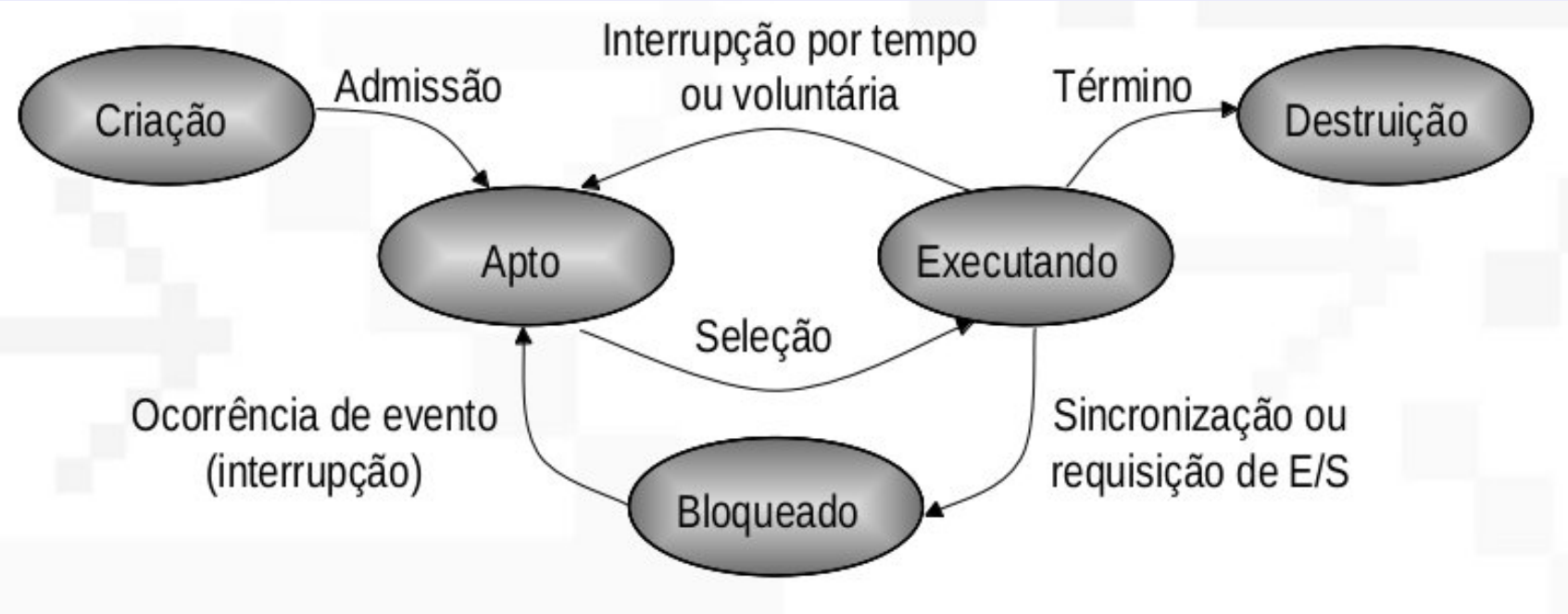
Modelo de 5 estados

- Executando (Running)
- Apto (Ready)
- Bloqueado (Blocked)
- Criação (New)
- Destruição (Exit)

Modelo a 5 estados



Eventos de transição de estados



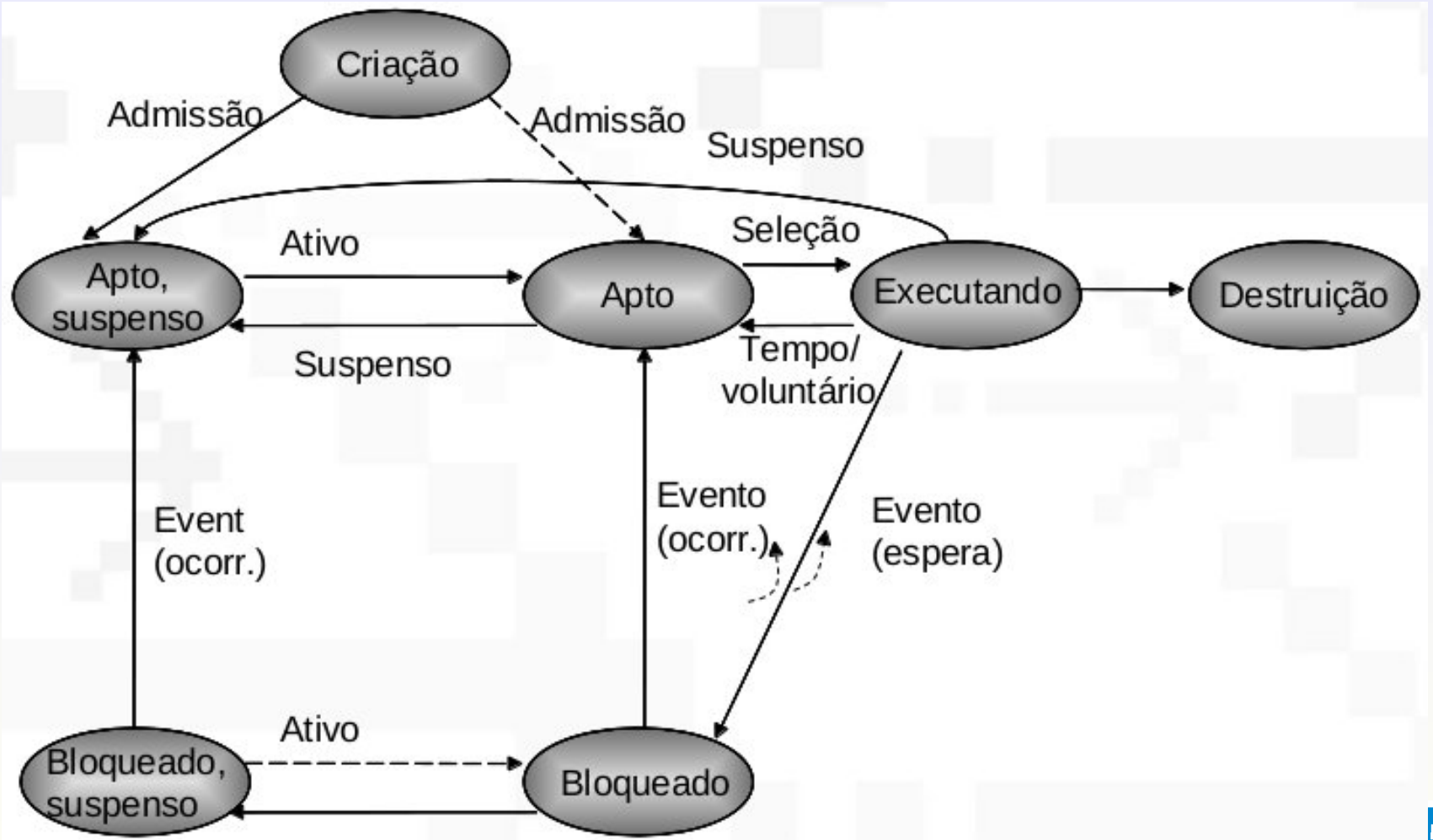
Processos suspensos

- Processador é mais rápido que operações de E/S
  - Possibilidade de todos processos estarem bloqueados esperando por E/S
- Liberar memória ocupada por estes processos
  - Transferidos para o disco (swap)
- Estado bloqueado assume duas situações:
  - Bloqueado com processo em memória
  - Bloqueado com processo no disco
- Necessidade de novos estados
  - Bloqueado, suspenso (Blocked, suspend)
  - Apto, suspenso (Ready, suspend)

Razões para suspender um processo

- Swapping:
  - SO necessita liberar memória para executar um novo processo
- Solicitação do usuário
  - Comportamento típico de depuradores
- Temporização:
  - Processo deve ter sua execução interrompida por um certo período de tempo
- Processo suspender outro processo
  - e.g. sincronização

Diagrama de estados de processos



Implementação

- Multiprogramação pressupõe a existência simultânea de vários processos disputando o processador
- Necessidade de "intermediar" esta disputa de forma justa
  - Gerência do processador
    - Algoritmos de escalonamento
- Necessidade de "representar" um processo
  - Implementação de processos
  - Estruturas de dados

Representação de processo

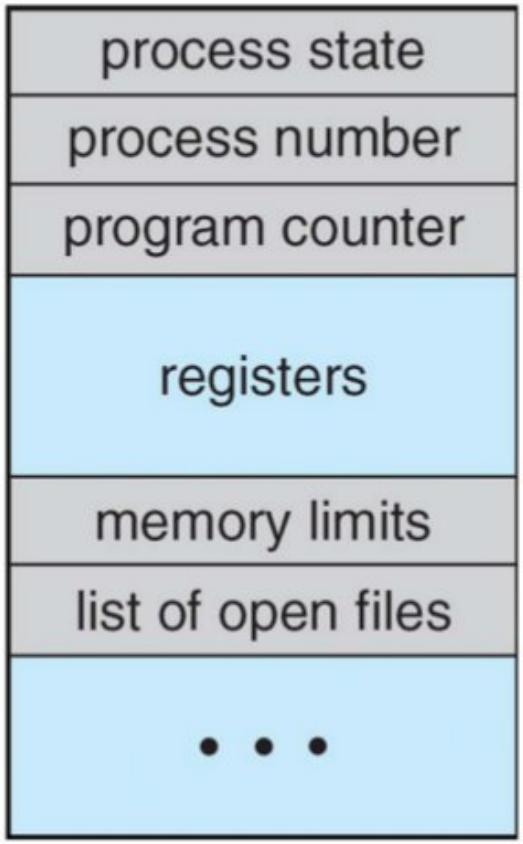
- Processo é um programa em execução
  - Áreas na memória para código, dados e pilha
- Possui uma série de estados (apto, executando, bloqueado, etc.) para representar sua evolução no tempo, implica em:
  - Organizar os processos nos diferentes estados
  - Determinar eventos que realizam a transição entre os estados
  - Determinar quando um processo tem direito a "utilizar" o processador
- Necessário manter informações a respeito do processo
  - e.g.: prioridades, localização em memória, estado atual, direitos de acesso, recursos que emprega, etc.

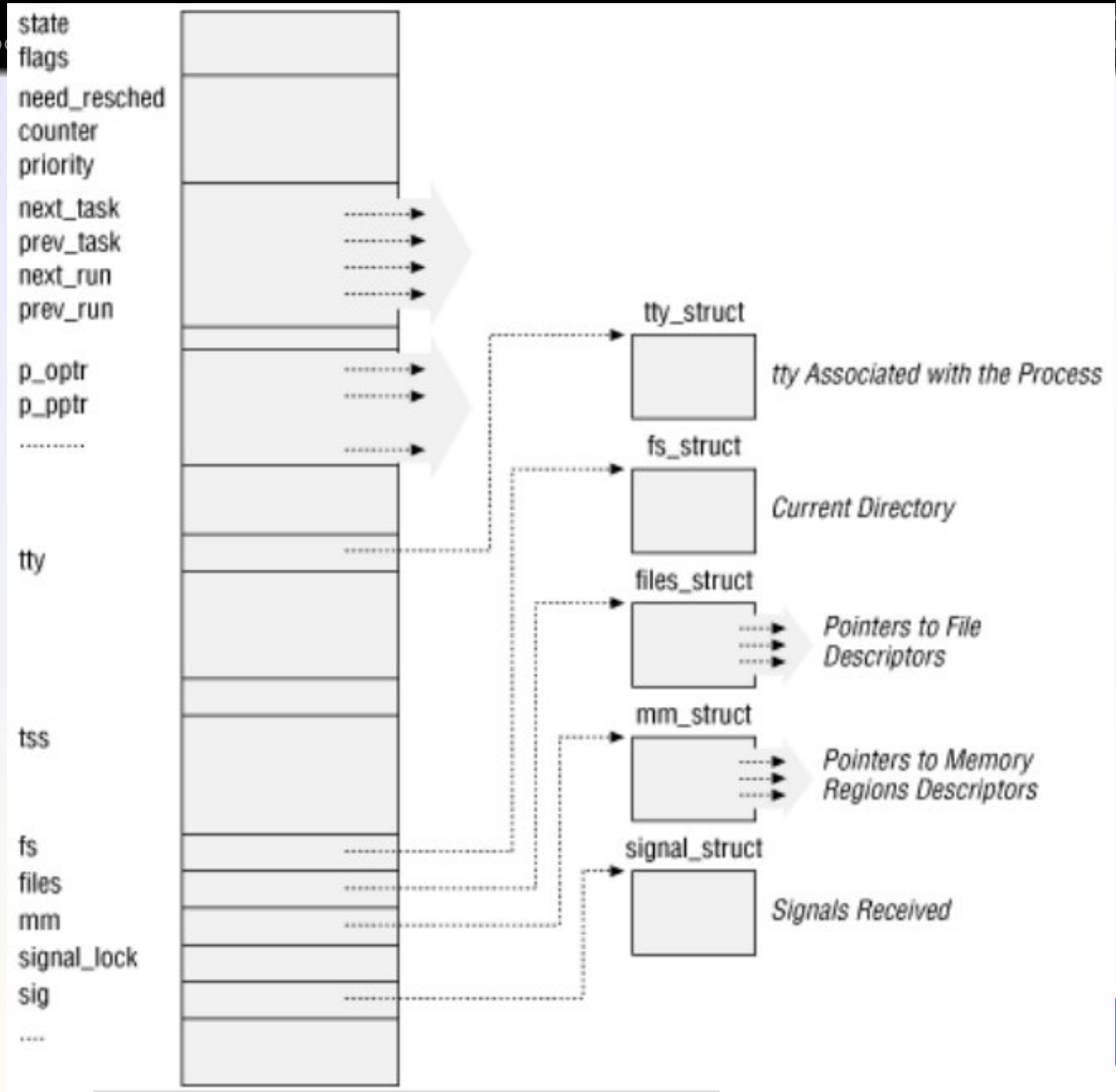
Bloco descritor de processo

- Abstração de processo é implementado através de uma estrutura de dados
  - Bloco descritor de processos (*Process Control Block* - PCB)
- Informações normalmente presentes em um descritor de processo:
  - Prioridade
  - Localização e tamanho na memória principal
  - Identificação de arquivos abertos
  - Informações de contabilidade (tempo CPU, espaço de memória, etc.)
  - Estado do processador (apto, executando, bloqueado, etc.)
  - Contexto de execução
  - Apontadores para encadeamento dos próprios descritores de processo etc.
- Definição através de `struct task_struct` no Linux:

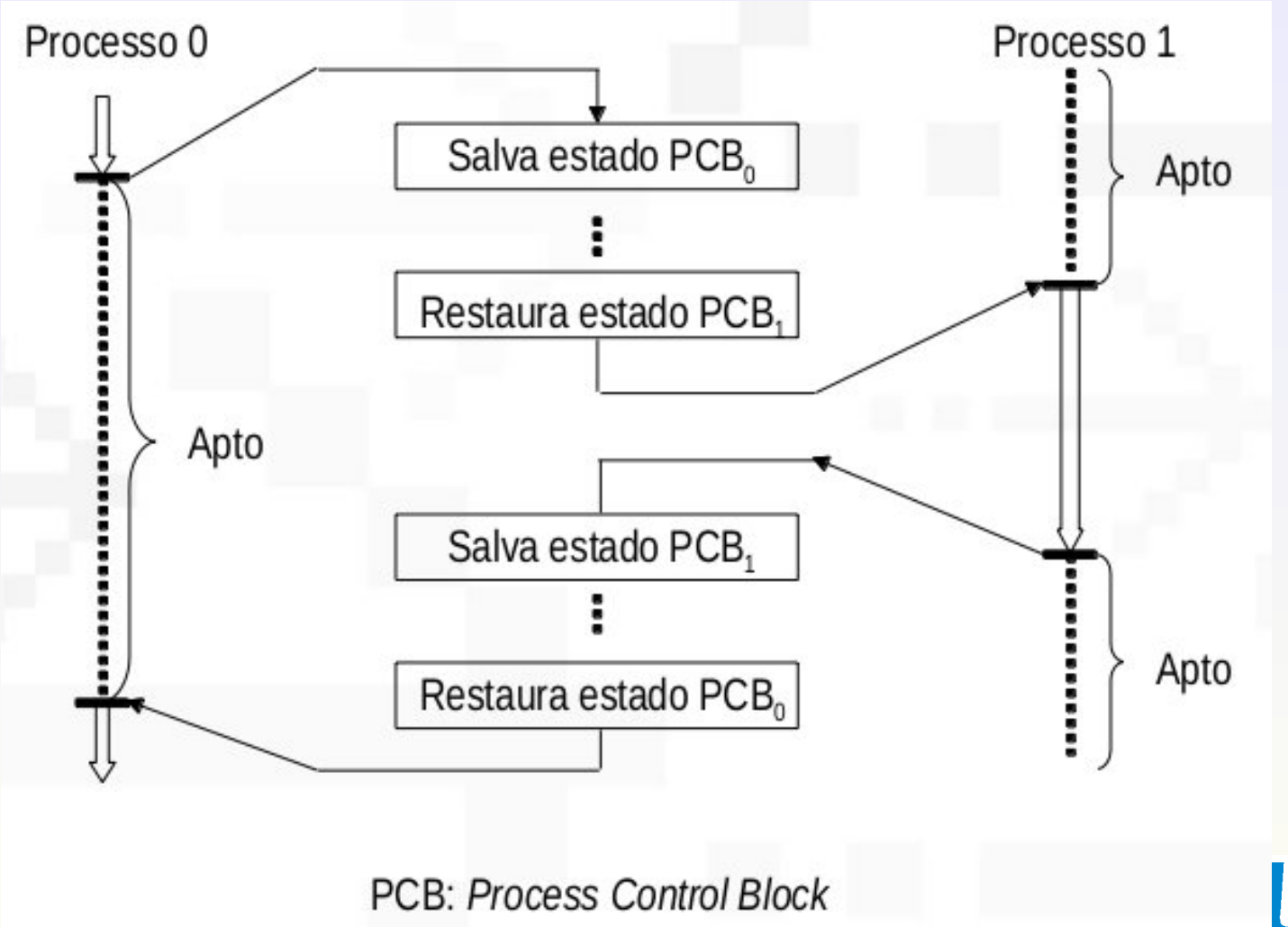
`https://github.com/torvalds/linux/blob/master/include/linux/sched.h`

Bloco descritor de processo



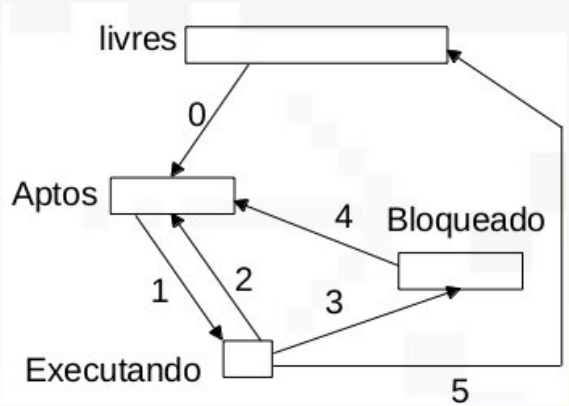

state
flags
need_resched
counter
priority
next_task
prev_task
next_run
prev_run
p_optr
p_pptr
.....
tty
tss
fs
files
mm
signal_lock
sig
....
tty_struct
tty Associated with the Process
fs_struct
Current Directory
files_struct
Pointers to File Descriptors
mm_struct
Pointers to Memory Regions Descriptors
signal_struct
Signals Received

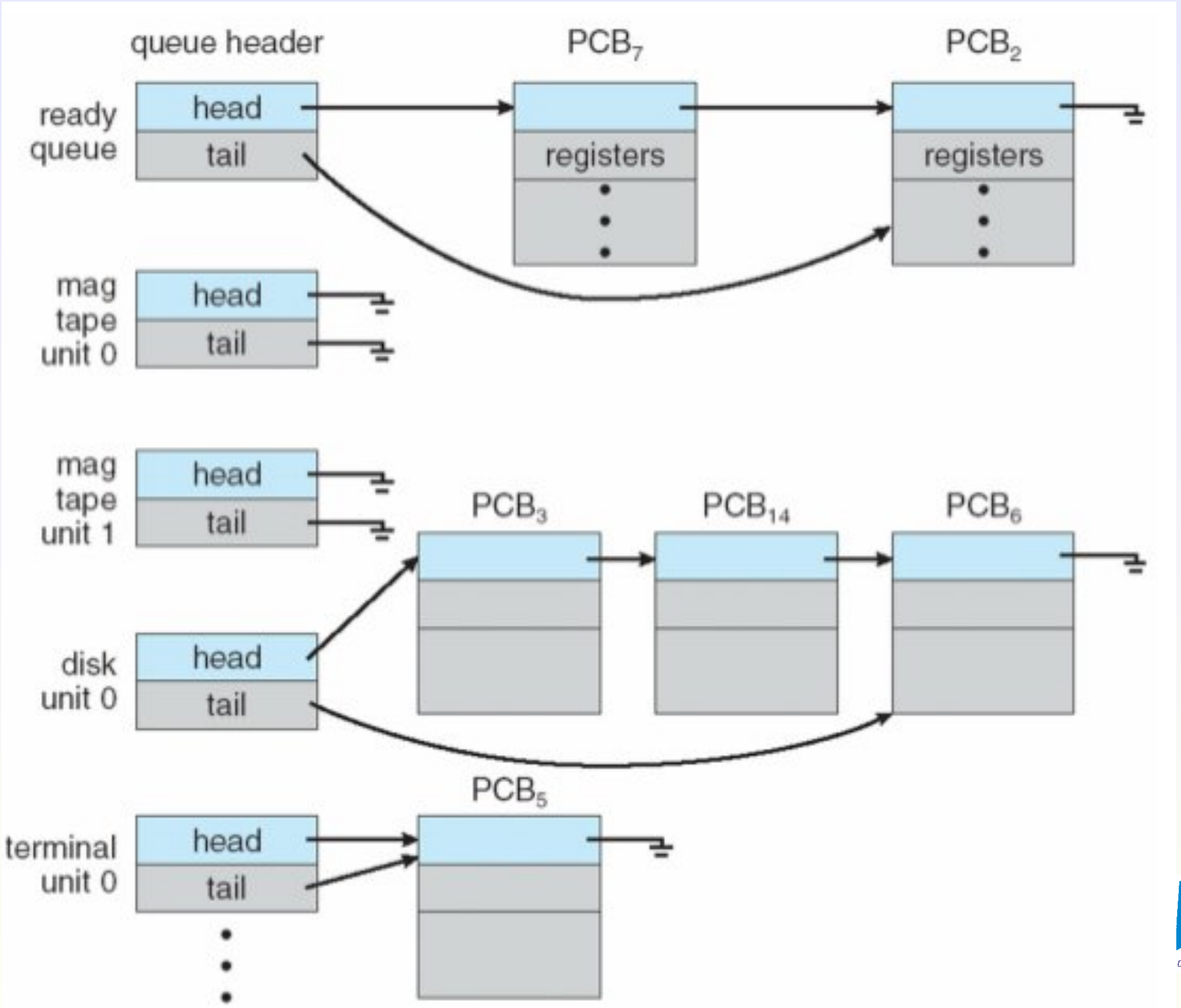
Chaveamento de contexto (dispatcher)



Os processos e as filas

- Um processo sempre faz parte de alguma fila
- Geralmente a própria estrutura de descritores de processos são empregadas como elementos dessas filas:
  - Fila de livres
    - Número fixo (máximo) de processos
    - Alocação dinâmica
  - Fila de aptos
  - Fila de bloqueados
- Eventos realizam transição de uma fila à outra




queue header
PCB7
PCB2
ready queue
head
tail
registers
registers
mag tape unit 0
head
tail
mag tape unit 1
head
tail
PCB3
PCB14
PCB6
disk unit 0
head
tail
PCB5
terminal unit 0
head
tail

Exemplo de bloco descritor de processos

- Estrutura de dados representado bloco descritor de processo

```
struct desc_proc{
    char      estado_atual;
    int       prioridade;
    unsigned inicio_memoria;
    unsigned tamanho_mem;
    struct    arquivos arquivos_abertos[20];
    unsigned tempo_cpu;
    unsigned proc_pc;
    unsigned proc_sp;
    unsigned proc_acc;
    unsigned proc_rx;
    struct    desc_proc *proximo;
}

struct desc_proc tab_desc[MAX_PROCESS];
```

Exemplo de bloco descritor de processos

- Estruturas de filas e inicialização

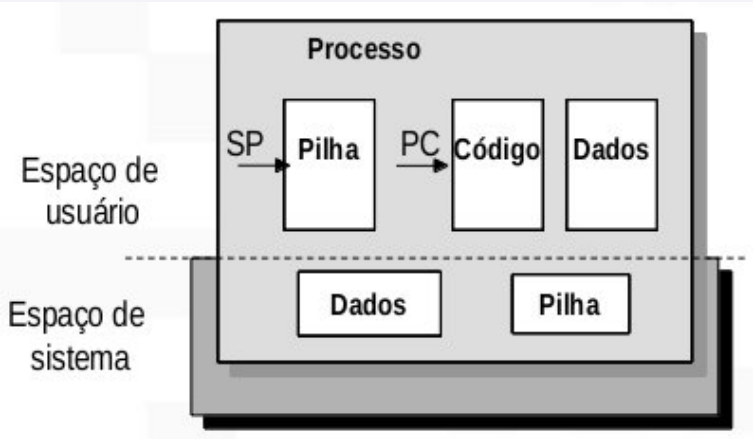
```
struct desc_proc *desc_livre;
struct desc_proc *espera_cpu;
struct desc_proc *usando_cpu;
struct desc_proc *bloqueados;

/* Inicialização das estruturas de controle */

for (i=0; i < MAX_PROCESS - 1; i++)
    tab_desc[i].prox = &tab_desc[i+1];

tab_desc[i].prox = NULL;
desc_livre = &tab_desc[0];

espera_cpu = NULL;
usando_cpu = NULL;
bloqueados = NULL;
```

Tarefas típicas no PCB durante o ciclo de vida

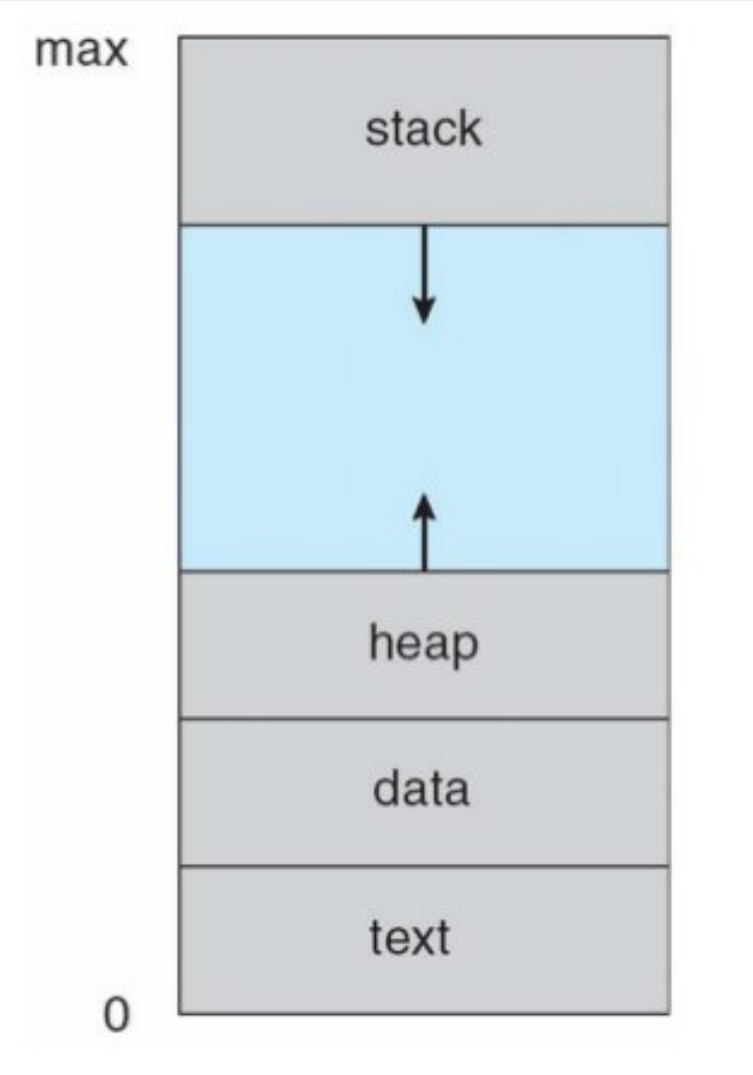
- Criação
  - Alocação de áreas de memória para código, dados e pilha e de estruturas de dados do sistema operacional
  - Inicialização do descritor de processo e inserção em filas do sistema
- Execução
  - Realizam as instruções da área de código
  - Interação com sistema operacional via chamadas de sistema
  - Atualização do bloco descritor de processo
  - Retratar estados e recursos que evoluem dinamicamente com a execução
  - Suscetível ao acionamento do escalonador/dispatcher em resposta a eventos
- Término
  - Liberação de recursos e estruturas de dados utilizadas

O modelo de processo

- Processo é representado por:
  - Espaço de endereçamento: área p/ armazenamento da imagem do processo
  - Estruturas internas do sistema (tabelas internas, áreas de memória, etc.)
    - Mantidos no descritor de processos
  - Contexto de execução (pilha, programa, dados, etc...)



Exemplo: modelo de processo



Gerenciando processos - Identificação

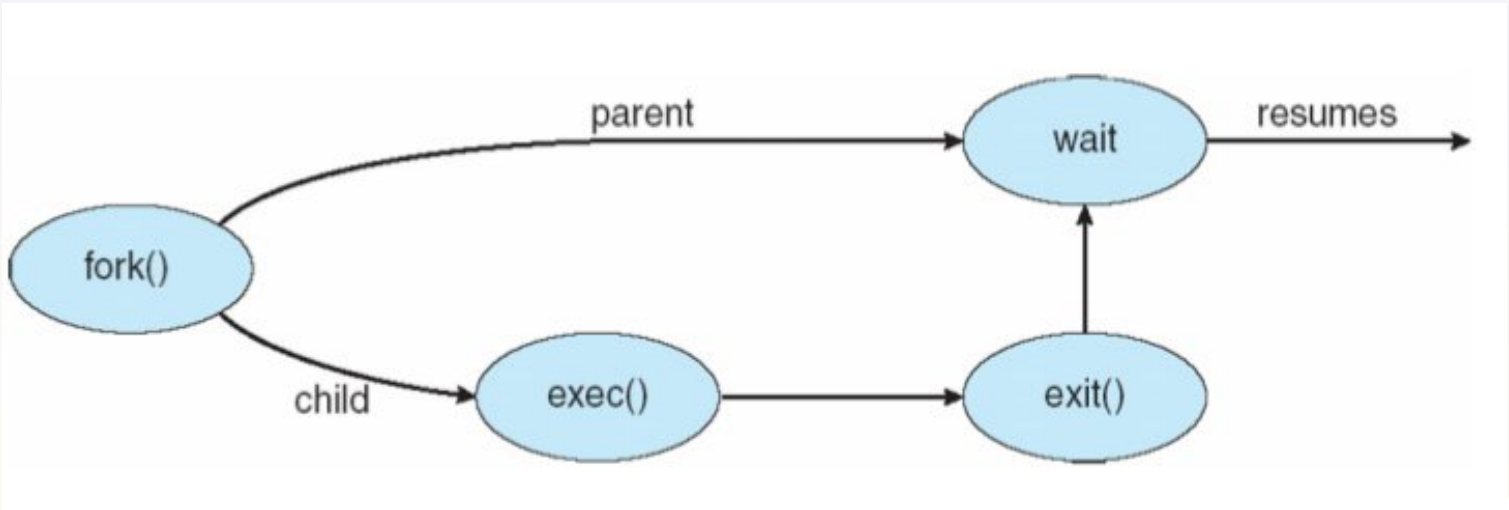
- Processos são identificados e gerenciados por seu PID
  - PID: *Process IDentifier*
- Um processo pode ser identificado via shell:
  - ps aux
  - pstree
- Um processo pode ser identificado via código (C):
  - getpid() - retorna o PID do processo em execução
  - getppid() - retorna o PID do processo pai
  - getpgrp() - retorna o PID do grupo do processo
- Exemplo:
  - Livro Sistemas Distribuídos
  - *meupid.c*, Cáp 3, pág. 55.

Gerenciando processos - Criação

- Processo pai cria processos filhos
- Processos filhos podem criar outros processos (árvore de processos)
- Quanto ao compartilhamento de recursos:
  - Pai e filho compartilham mesmos recursos
  - Filho compartilha um subconjunto dos recursos do pai
  - Pai e filho não compartilham recursos
- Quanto à execução
  - Pai e filho executam concorrentemente
  - Pai espera até que os filhos terminem
- Quanto ao espaço de endereçamento
  - Filho é uma duplicata do pai
  - Filho tem um novo programa carregado nele
  - Processos se comunicam por meio de sinais
- 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.

Gerenciando processos - Criação

- Criação de processos no UNIX
  - *fork()* - chamada de sistema para criação de processos
  - *exec()* - chamada de sistema após o fork para substituir o espaço de memória com um novo programa
  - *system()* - o processo não é substituído, espera o filho terminar
  - *wait()* - espera pelo término do processo indicado

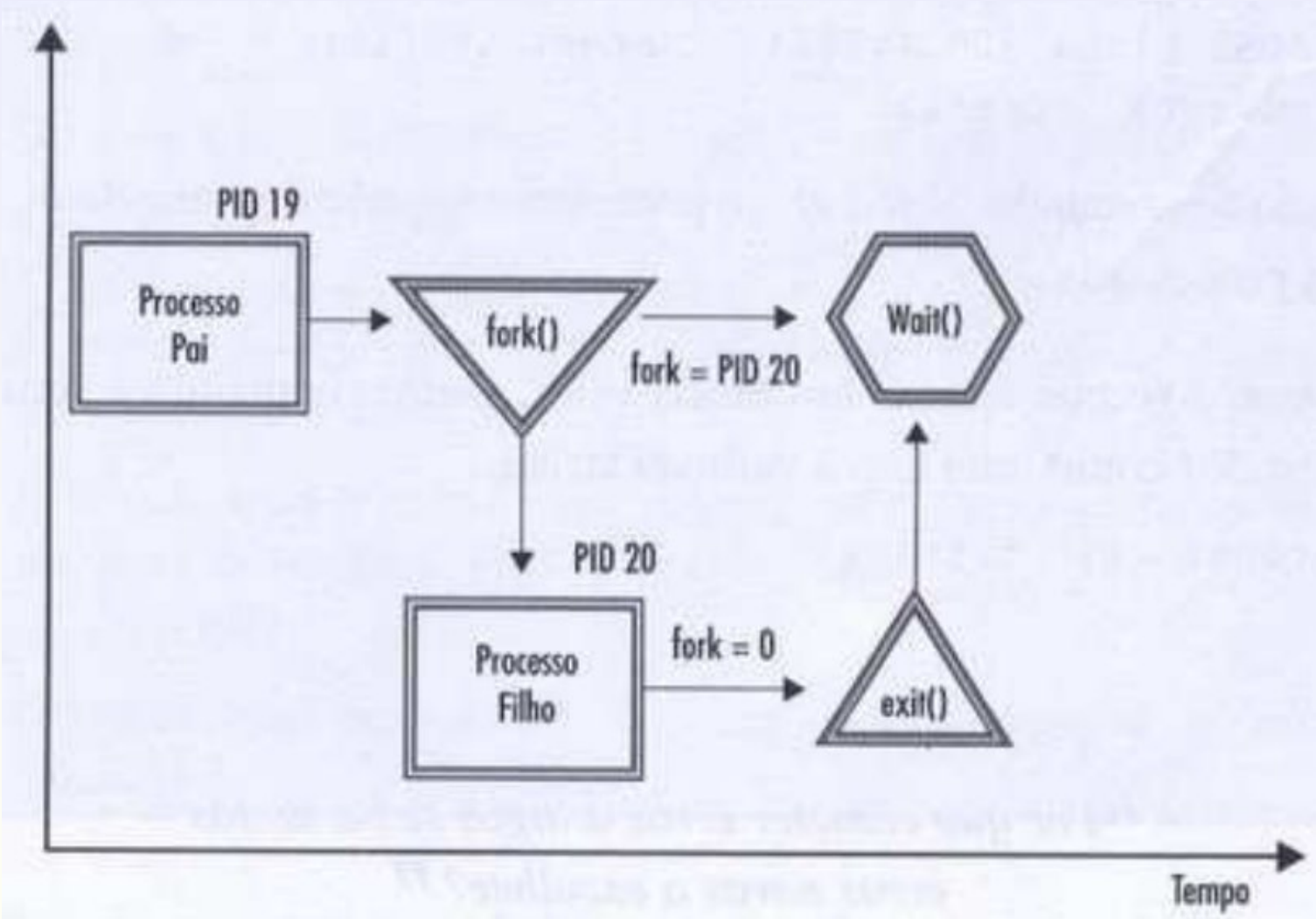


Gerenciando processos - Criação

- Exemplo de utilização da *exec()*
  - *execv.c*, Cáp 3, pág. 56
- Exemplo de utilização da *system()*
  - *system.c*, Cáp 3, pág. 63
- Exemplo de utilização de *fork()* e *wait()*
  - *fork_simples.c*, Cáp 3, págs. 65-66
  - *fork_loop.c*, Cáp 3, págs. 67-68

Gerenciando processos - Criação

Criação de um processo filho através do fork():



Gerenciando processos - Término

- Normal: Processos executam sua última instrução e pedem ao SO para ser terminado (exit)
  - Saída do filho para o pai (via wait)
  - Recursos do processo são desalocados pelo SO
- Pai pode terminar a execução do filho (kill)
  - Filho pode ter excedido os recursos alocados
  - A tarefa definida para o filho não é mais requerida
  - Se o pai está terminando
    - Alguns SOs não permitem o filho continuar se o pai termina
    - Filhos terminados em cascata

Gerenciando processos - Término

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    int pid = 22275;

    printf("Sou o processo %d ",getpid());
    printf("e matarei o processo %d\n",pid);

    // SIGKILL = 9 (termina processo incondicionalmente)
    kill(pid,9);

    return 0;
}
```

Exemplos fork()

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

int main()
{
    /* gera um processo filho */
    fork();

    /* gera outro processo filho */
    fork();

    /* e gera ainda mais um */
    fork();

    return 0;
```

Figura 3.28 Quantos processos são criados?

Exemplos fork()

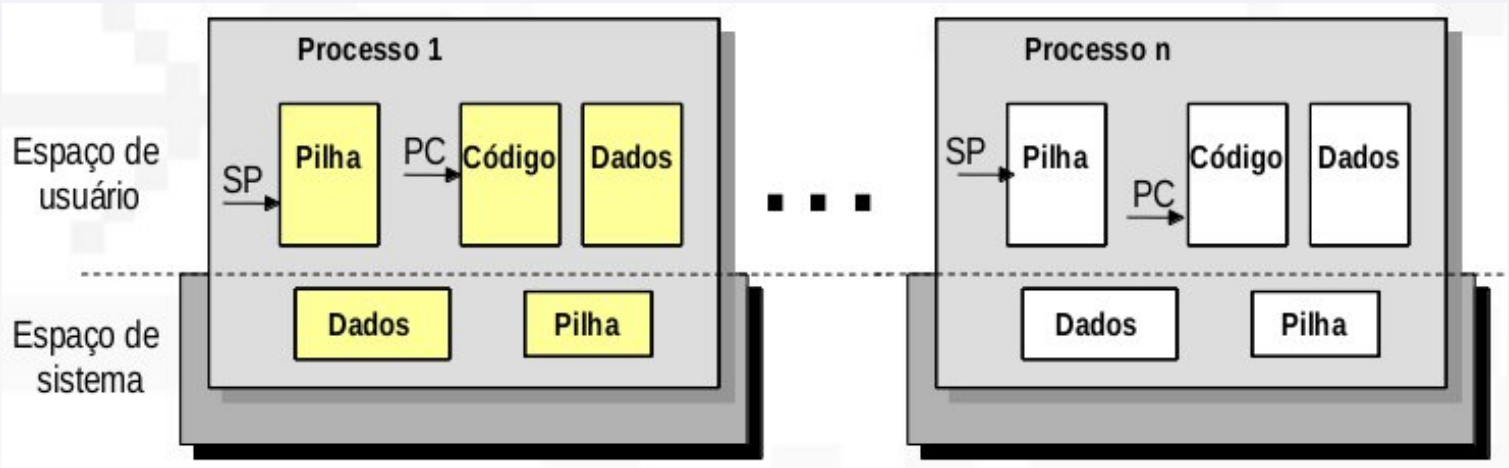
```
#include <stdio.h>
#include <unistd.h>

int value = 5;

int main()
{
pid_t pid;

   pid = fork();

   if (pid == 0) { /* processo filho */
      value += 15;
      return 0;
   }
   else if (pid > 0) { /* processo pai */
      wait(NULL);
      printf("PARENT: value = %d",value); /* LINHA A */
      return 0;
   }
}
```

Figura 3.30 Que saída teremos na linha A?

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid, pid1;

    /* gera um processo filho */
    pid = fork();

    if (pid < 0) { /* um erro ocorreu */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* processo filho */
       pid1 = getpid();
       printf("child: pid = %d",pid); /* A */
       printf("child: pid1 = %d",pid1); /* B */
    }
    else { /* processo pai */
       pid1 = getpid();
       printf("parent: pid = %d",pid); /* C */
       printf("parent: pid1 = %d",pid1); /* D */
       wait(NULL);
    }

    return 0;
```

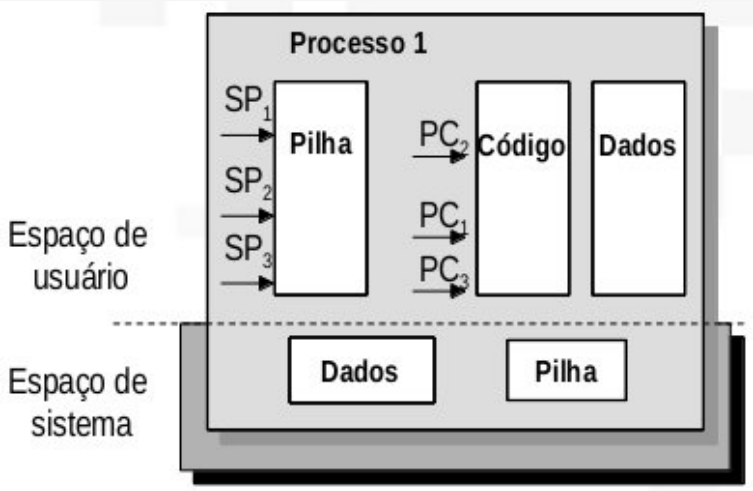
Vários processos

- Um fluxo de controle por processo (thread)
- Troca de processo implica em atualizar estruturas de dados internas do sistema operacional
  - e.g.: contexto, espaço de endereçamento, etc...



Vários fluxos em um único processo

- Um fluxo de instrução é implementado através do contador de programa (PC) e de uma pilha (SP)
- Estruturas comuns compartilhadas
  - Código
  - Dados
  - Descritor de processo
- Conceito de *thread*



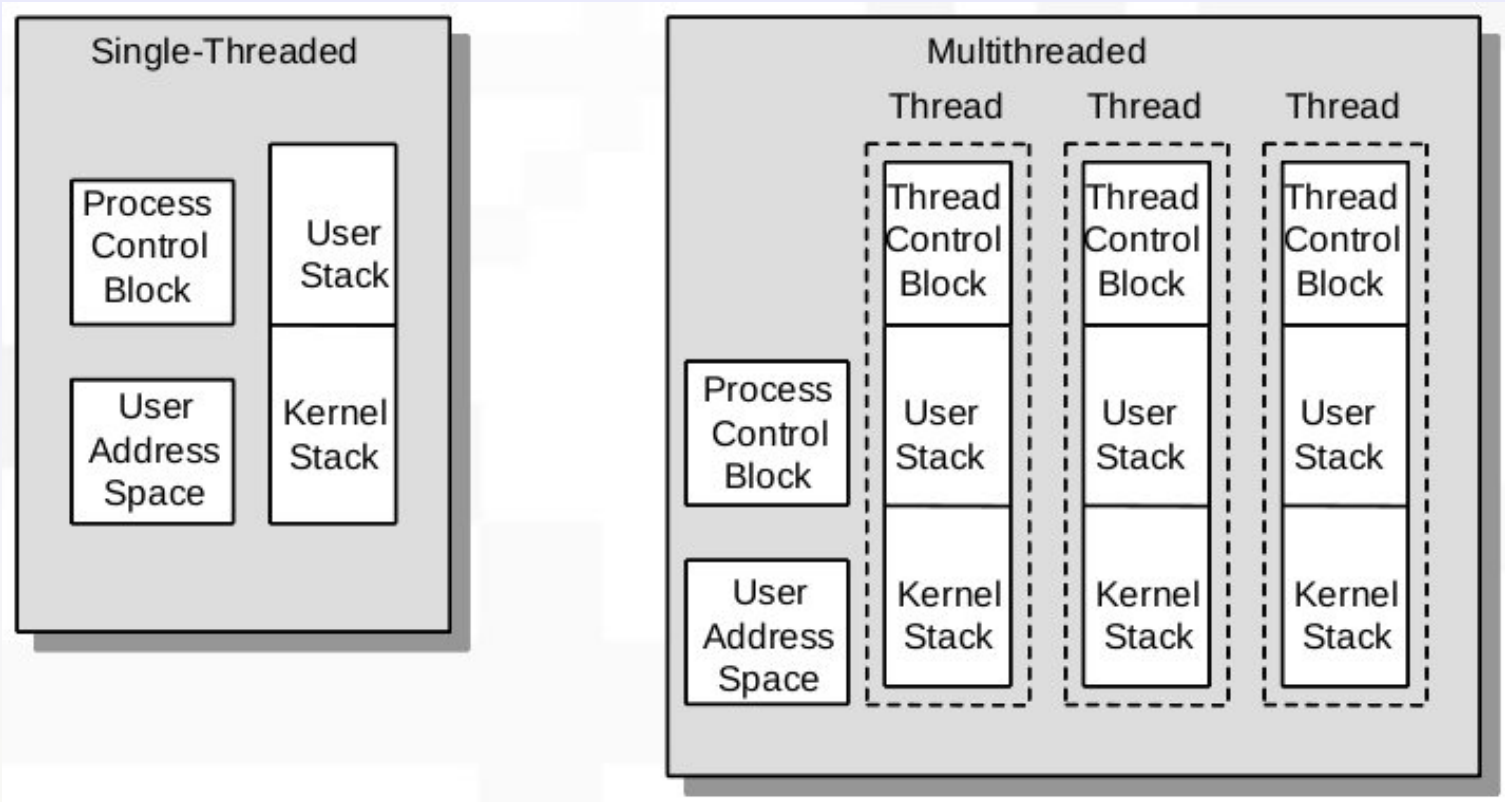
Multiprogramação pesada

- Custos de gerenciamento do modelo de processos
  - Criação do processo
  - Troca de contextos
  - Esquemas de proteção, memória virtual, etc.
- Custos são fator limitante na interação de processos
  - Unidade de manipulação é o processo
  - Mecanismos de IPC (Inter Process Communications) necessitam tratamento de estruturas complexas que representam o processo e sua propriedades
- Solução
  - "Aliviar" os custos, ou seja, reduzir o "peso" das estruturas envolvidas

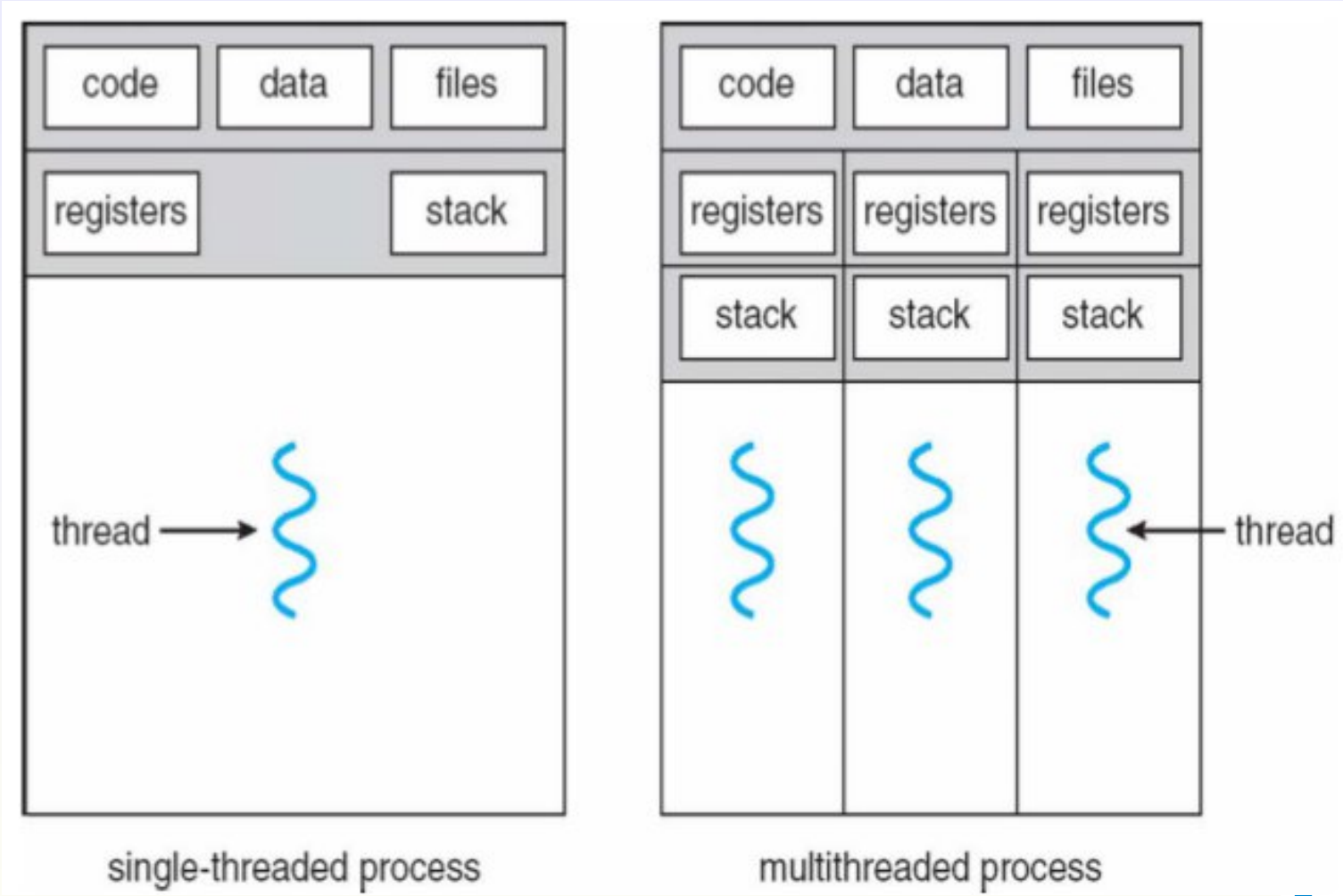
Multiprogramação leve

- Fornecido pela abstração de um fluxo de execução (thread)
  - Basicamente o conceito de processo
- Unidade de interação passa a ser função
- Contexto de uma thread
  - Registradores (pilha, apontador de programa, registradores de uso geral)
- Comunicação através do compartilhamento direto da área de dados

Modelos de processos single threaded e multithreaded

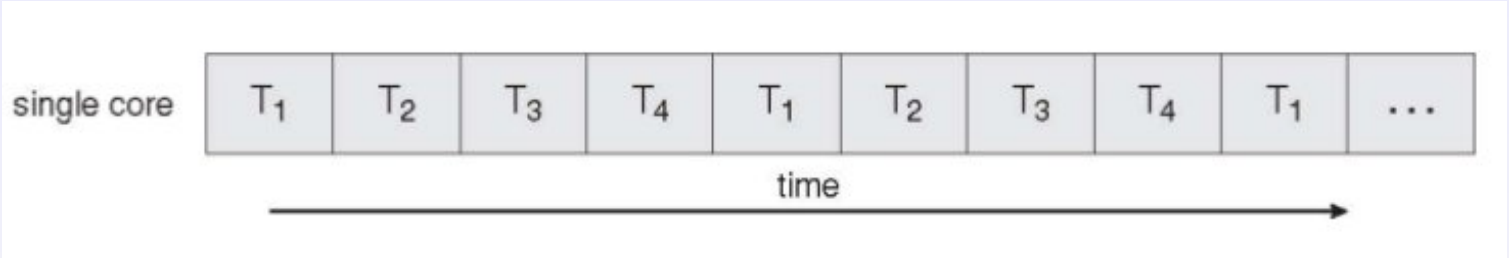


Modelos de processos single threaded e multithreaded

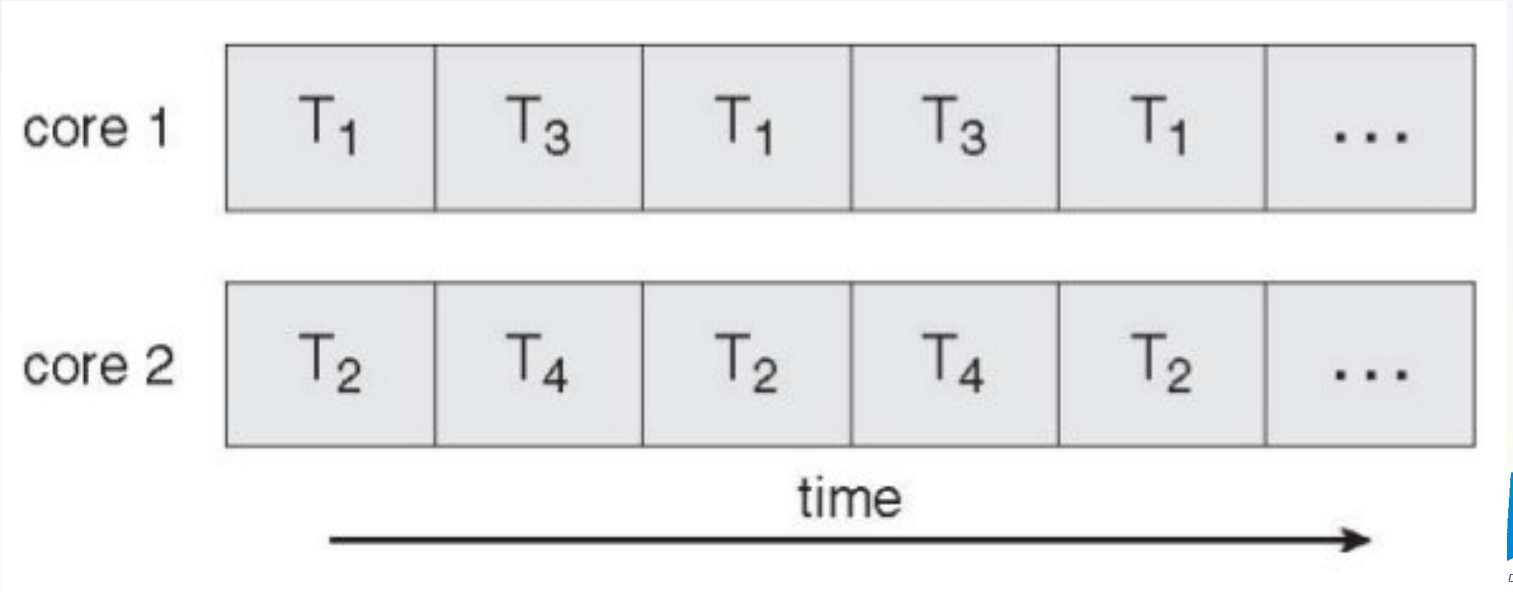


Execução em um sistema Single vs Multi Core

Single core



Multi core



Implementação de threads

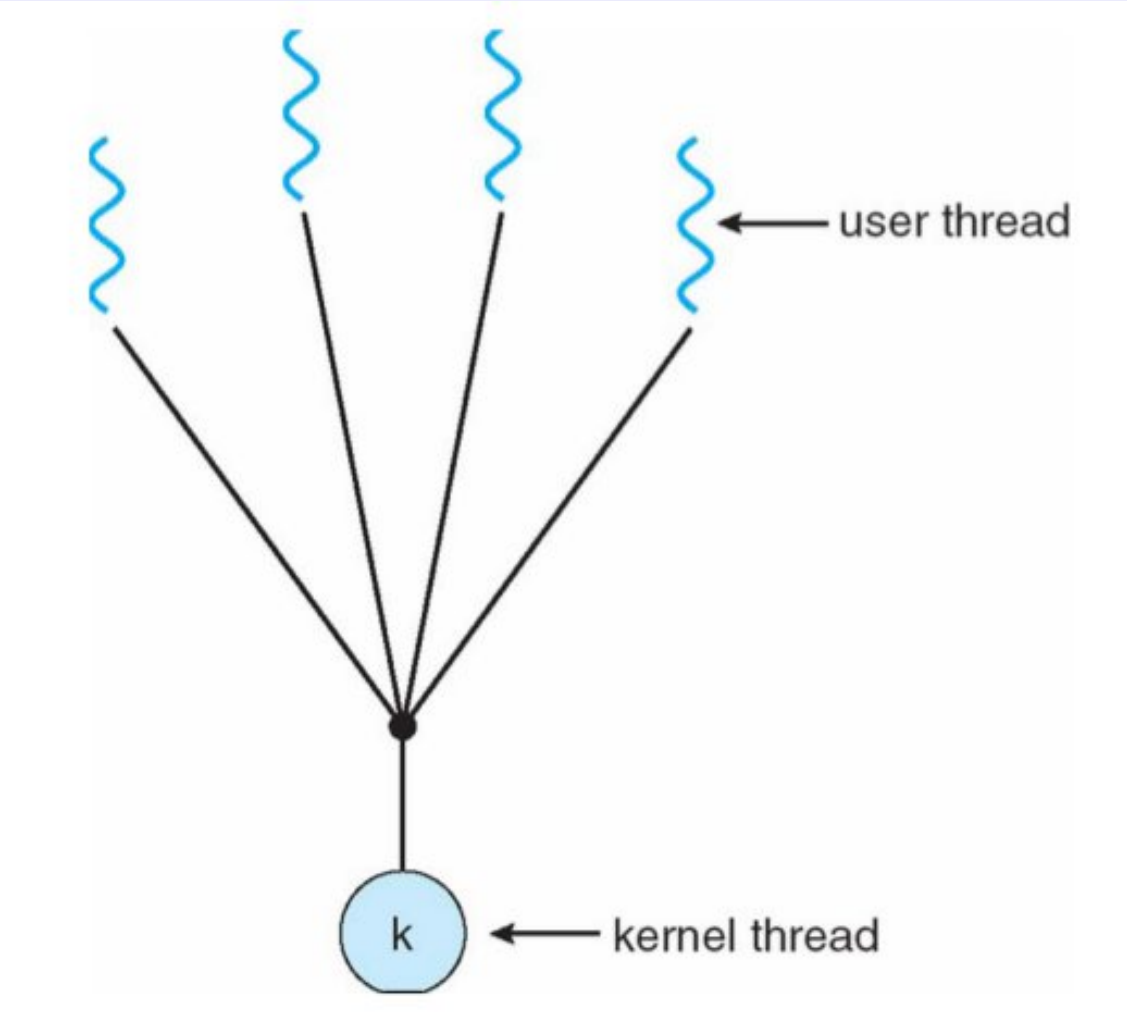
- Threads são implementadas através de estruturas de dados similares ao descritor de processo
  - Descritor de threads
  - Menos complexa (leve)
- Podem ser implementadas em dois níveis diferentes:
  - Espaço de usuário
  - Espaço de sistema
- Modelos de sistemas Multithreaded

N:1 Many-to-One
1:1 One-to-One
M:N Many-to-many

Modelo N:1

- Threads a nível de usuário
  - User level threads ou ainda process scope
- Todas as tarefas de gerenciamento de threads são feitas a nível da aplicação
  - Threads são implementadas por uma biblioteca que é ligada ao programa
  - Interface de programação (API) para funções relacionadas com threads
    - e.g.: criação, sincronismo, término, etc.
- O sistema operacional não "enxerga" a presença das
- A troca de contexto entre threads é feita em modo usuário pelo escalonador embutido na biblioteca threads
  - Não necessita privilégios especiais
  - Escalonamento depende da implementação

Implementação modelo N:1



Modelo N:1

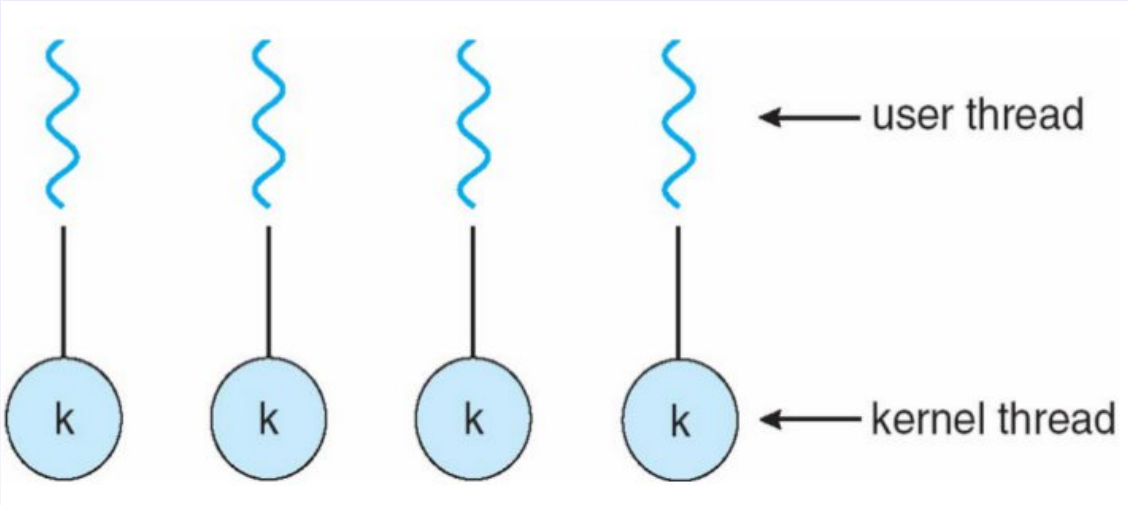
Vantagens e desvantagens

- Vantagens:
  - Sistema operacional divide o tempo do processador entre os processos "pesados" e, a biblioteca de threads divide o tempo do processo entre as threads
  - Leve: sem interação/intervenção do sistema operacional
- Desvantagens:
  - Uma thread que realiza uma chamada de sistema bloqueante leve ao bloqueio de todo o processo
    - e.g.: operaçoes de entrada/saída
  - Não explora paralelismo em máquinas multiprocessadoras

Modelo 1:1

- Threads a nível do sistema
  - kernel level threads ou ainda system scope
- Resolver desvantagens do modelo N:1
- O sistema operacional "enxerga" as threads
  - Sistema operacional mantém informações sobre processos e sobre threads
  - Troca de contexto necessita a intervenção do sistema operacional
- O conceito de threads é considerado na implementação do sistema operacional

Implementação modelo 1:1



Modelo 1:1

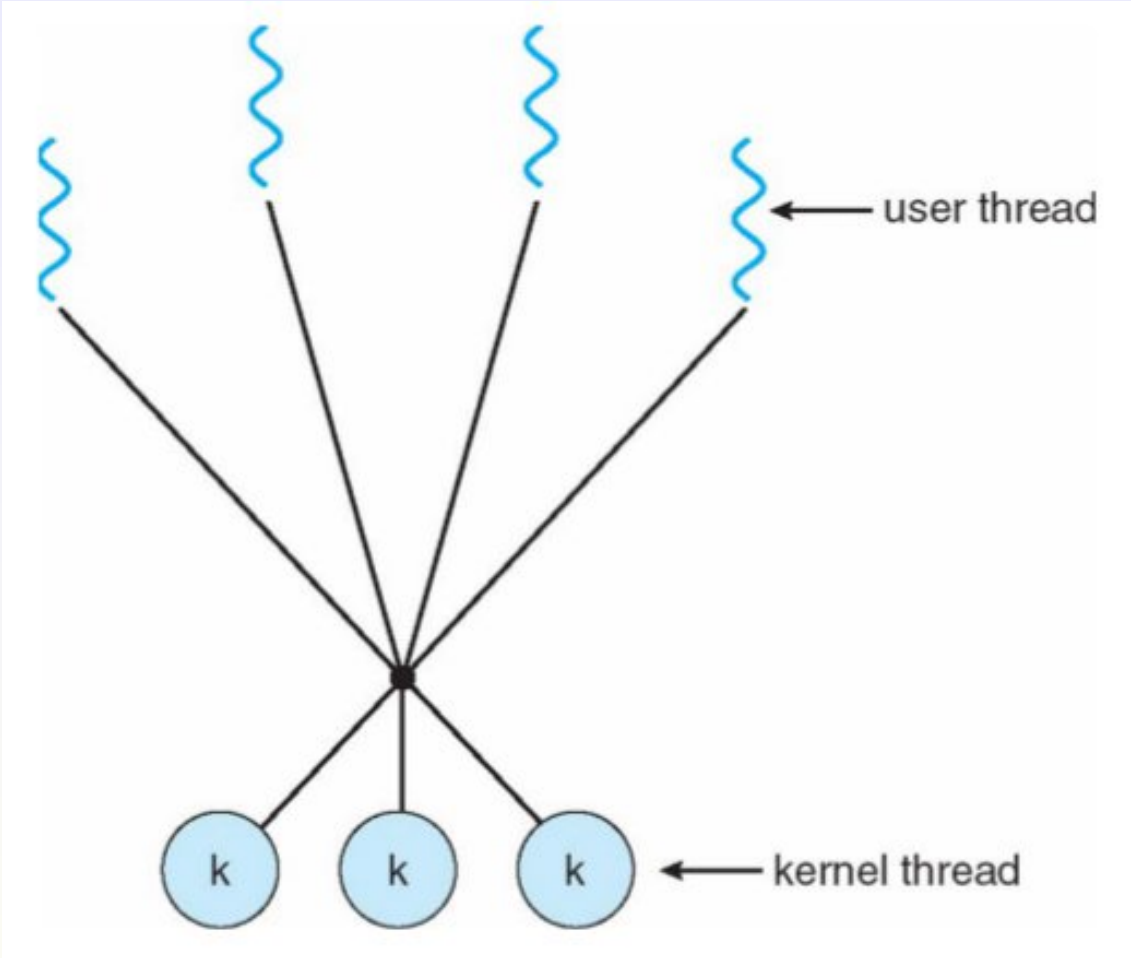
Vantagens e desvantagens

- Vantagens:
  - Explora o paralelismo de máquinas multiprocessadoras (SMP)
  - Facilita o recobrimento de operações de entrada/saída por cálculos
- Desvantagens:
  - Implementação "mais pesada" que o modelo N:1

Modelo M:N

- Abordagem que combina os modelos N:1 e 1:1
- Oferece dois níveis de escalonamento
  - Nível usuário: threads sobre unidade de escalonamento
  - Nível sistema: unidades de escalonamento sobre processador
- Dificuldade é parametrizar M e N

Implementação modelo M:N



Porque utilizar threads ?

- Permitir a exploração do paralelismo real oferecido por máquinas multiprocessadores (modelo M:N ou 1:1)
- Aumentar número de atividades executadas por unidade de tempo (throughput)
- Diminuir tempo de resposta
  - Possibilidade de associar threads a dispositivos de entrada/saída
- Sobrepor operações de cálculo com operações de entrada e saída

Vantagens de multithreading

- Tempo de criação/destruição de threads é inferior que tempo de criação/destruição de um processo
- Chaveamento de contexto entre threads é mais rápido que tempo de chaveamento entre processos
- Como threads compartilham o descritor do processo que as porta, elas dividem o mesmo espaço de endereçamento, o que permite a comunicação por memória compartilhada sem interação com o núcleo

Implementações de Threads no UNIX

- POSIX Threads
  - *Portable Operating System Interface (X = UNIX)*
- *pthread_create()* - permite que uma função do programa seja executada em um processo leve concorrentemente
  - Exemplo : *exemplo_pthread.c* (Cap 4, p. 99)
- *pthread_join()* - suspende a thread que a chamou até que thread identificada termine
- Exemplos:
  - *pthread_join.c* (Cap 4, p. 102-103)
  - *numero_primo.c* (Cap 4, p. 103-104)
- *pthread_detach()* - desconecta uma thread do processo pesado
  - Exemplo: *concorrentes.c* (Cap 4, p. 105-106)
- *pthread_exit()* - término de um processo leve

Implementações de Threads no UNIX