Ministério da Educação
Centro Federal de Educação Tecnológica Celso Suckow da Fonseca
UNED Nova Friburgo
Bacharelado em Sistemas da Informação

Processos e Threads

Sistemas Operacionais

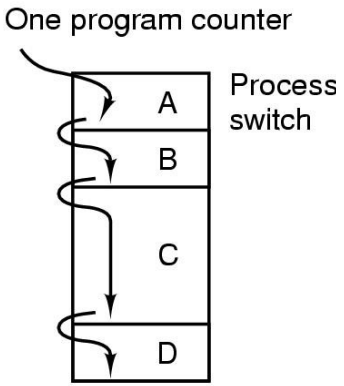
Prof. Bruno Policarpo Toledo Freitas
bruno.freitas@cefet-rj.br

Objetivo

- **Definir e diferenciar processo e thread**
- **Identificar as principais características de um processo dentro do sistema operacional**
- **Compreender os custos e o papel do sistema operacional / processador ao trocar processos**
- **Introduzir o conceito de multithreading**

Processos
O Modelo de Processos

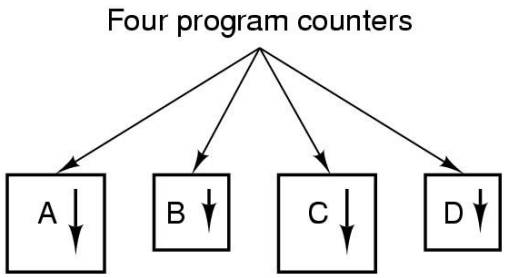
- **Quando um programa passa a ser executado, ele se torna um *processo*, passando a competir pelo processador**
- **Os computadores modernos funcionam baseados na *multiprogramação (multitasking)*, executando múltiplos *processos* residentes na memória**
- **Exemplo: multiprogramação de 4 programas**
  - Modelo conceitual de 4 processos independentes e sequenciais
  - Apenas 1 programa está ativo por vez



Visão do computador / memória

Processos
O Modelo de Processos

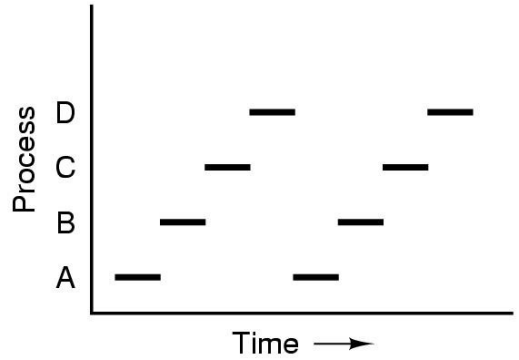
- **Quando um programa passa a ser executado, ele se torna um *processo*, passando a competir pelo processador**
- **Os computadores modernos funcionam baseados na *multiprogramação (multitasking)*, executando múltiplos *processos* residentes na memória**
- **Exemplo: multiprogramação de 4 programas**
  - Modelo conceitual de 4 processos independentes e sequenciais
  - Apenas 1 programa está ativo por vez



Visão de cada programa

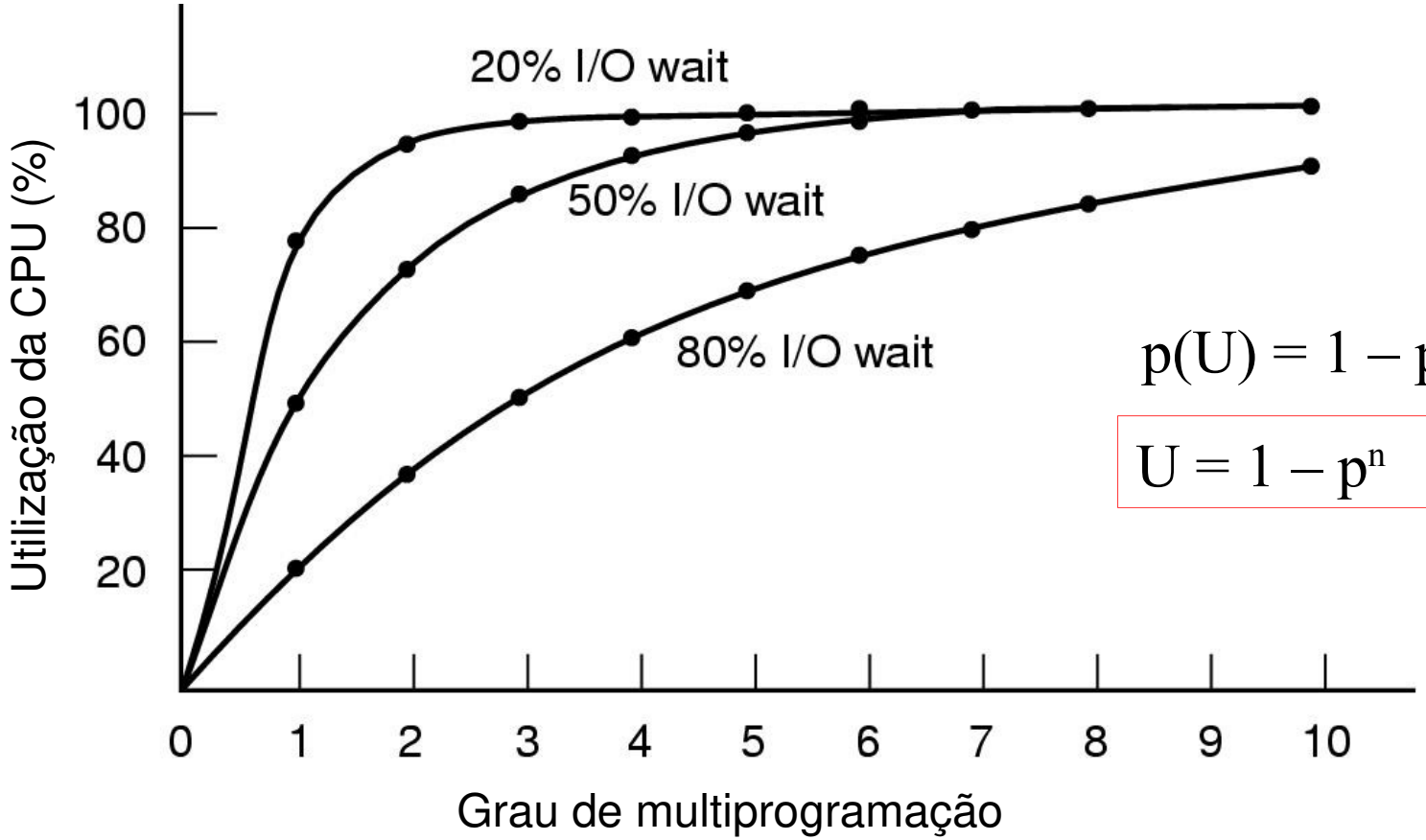
Processos
O Modelo de Processos

- **Quando um programa passa a ser executado, ele se torna um *processo*, passando a competir pelo processador**
- **Os computadores modernos funcionam baseados na *multiprogramação (multitasking)*, executando múltiplos *processos* residentes na memória**
- **Exemplo: multiprogramação de 4 programas**
  - Modelo conceitual de 4 processos independentes e sequenciais
  - Apenas 1 programa está ativo por vez



Visão do usuário do computador

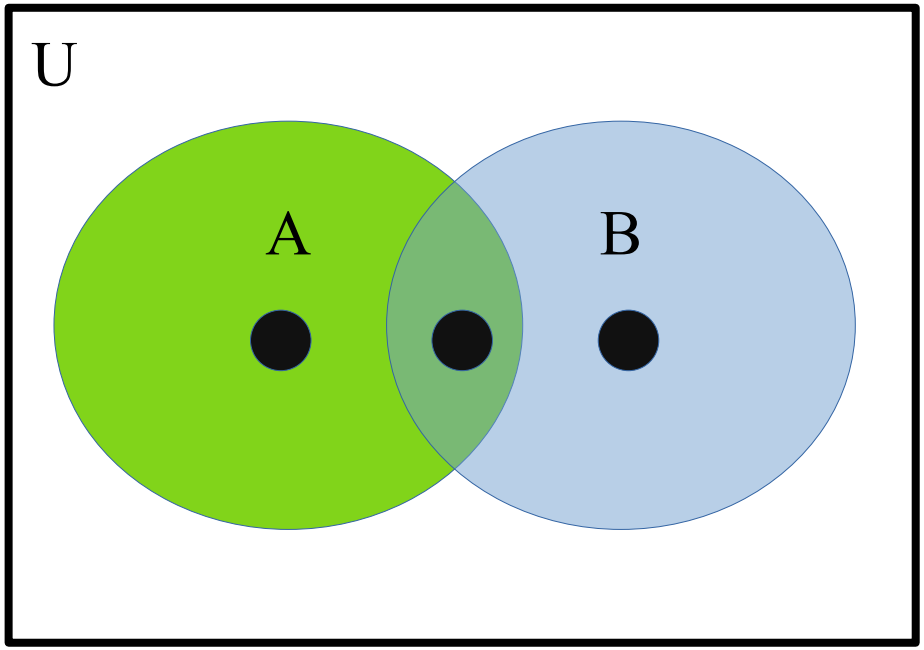
Modelagem da multiprogramação



$$p(U) = 1 - p(!U)$$

$$U = 1 - p^n$$

Utilização da CPU em função do número de processos na memória



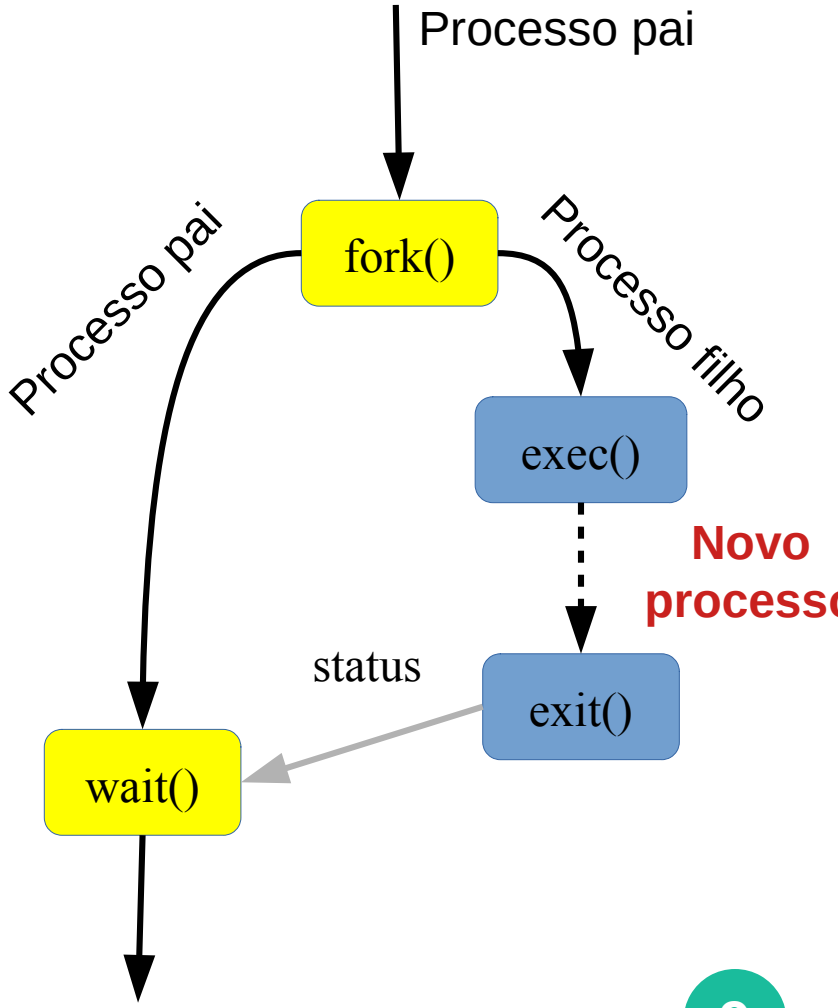
$$p(A) = p(A) - p(A \cap B)$$

Criação de Processos

- **Principais eventos que causam criação de processos:**
  - Inicialização do sistema (processos *daemons)*
    - Windows: Serviços, acessado pelas “Ferramentas Administrativas”
    - Linux moderno: controlados pela aplicação *systemctl* do *systemd*
  - Execução de um criador de processos por um outro processo
  - Usuário solicita criação de um processo
  - Início de um processo de *batch*

Criação de Processos

- **Win32: procedural**
  - *CreateProcess*
- **UNIX: modelo *fork-exec***
  - Cria hierarquia de processos
  - Base de funcionamento do *shell* UNIX



Criação de processos
Shell

```
while (TRUE) {                          /* repeat forever */
    type_prompt( );                                  /* display prompt */
    read_command (command, parameters)          /* input from
    terminal */

    if (fork() != 0) {                          /* fork off child process */
      /* Parent code */
      waitpid( -1, &status, 0);                  /* wait for child to exit */
    } else {
      /* Child code */
      execve (command, parameters, 0); /* execute command */
    }
}
```

Processos em Execução

- **Windows: Gerenciador de Tarefas**
- **UNIX:**
  - Programa ***ps***
    - ***ps -u [usuario]***
    - ***ps -e***
    - ***ps aux***
  - Programa ***top***
  - Programas **fg, bg, disown / nohup, &**

Término de Processos

- **Condições que terminam processos:**
  - Saída normal (voluntário)
    - Retorno SEMPRE é 0
  - Saída com erro (voluntário)
    - Número diferente de 0
    - Número deve identificar o tipo de erro
  - Erro fatal (involuntário)
  - Morto por outro processo (involuntário)
- **UNIX: código de retorno pela variável $?**

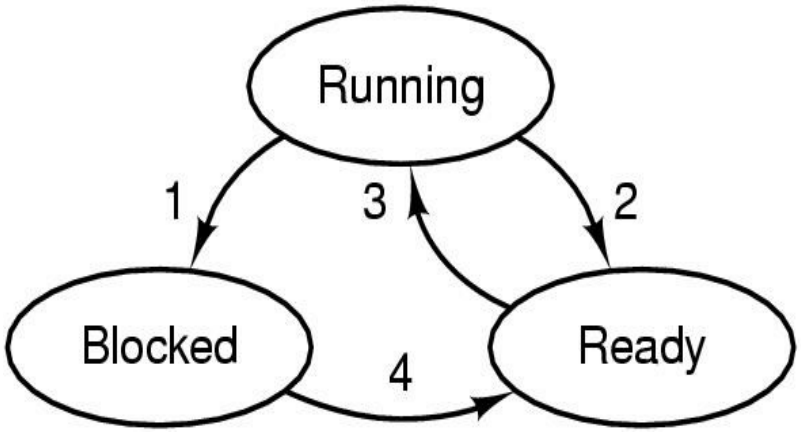
Morte de Processos

- **Windows:**
  - Gerenciador de tarefas
- **UNIX:**
  - comando ***kill*** (envia sinais)
  - **[sudo] kill [-9] [pid]**
  - **[sudo] killall [-9] [nome do processo]**

Hierarquia de Processos

- **Pais podem criar novos processos filhos, e filhos podem criar novos “netos”**
- **Formam uma hieraquia**
  - Em UNIX: "Grupo de Processos"
- **Windows não possui conceito de hierarquia**
  - Todos os processos são iguais

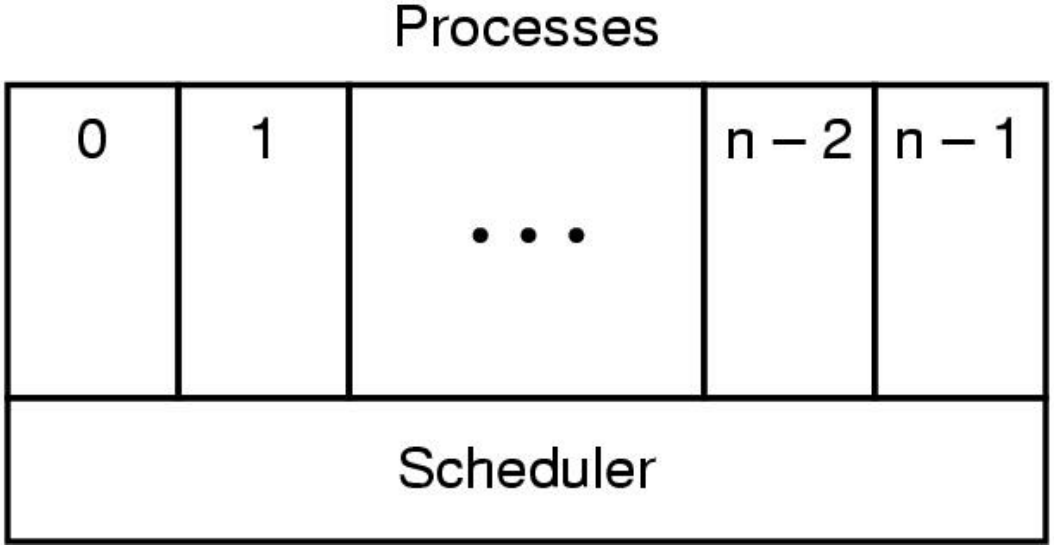
Estados de um Processo



1.Processo bloqueia por Entrada/Saída
2.Escalonador escolhe um processo
3.Escalonador coloca o processo escolhido para executar
4.Entrada/Saída se torna disponível

- **Estados possíveis de um processo:**
  - Executando
  - Bloqueado
  - Pronto

O Escalonador



- **Camada mais baixa de um SO estruturado em torno de processos**
  - Trata as interrupções e escalonamento de processos
- **Acima do escalonador, tudo é baseado em processos**

Implementação de Processos

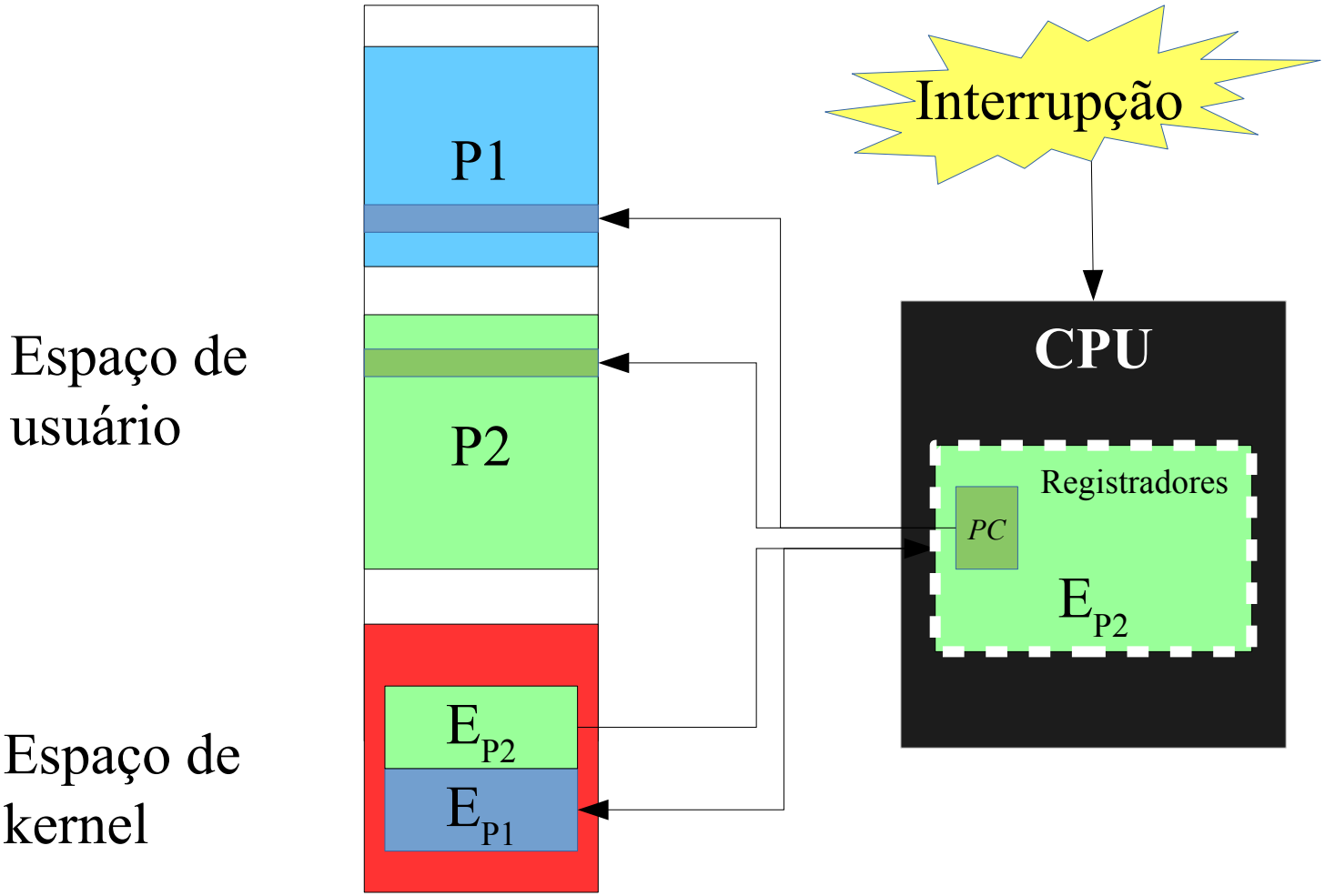
Campos de uma entrada na tabela de processos

Gerência de processos	Gerência de memória	Gerência de arquivos
Registradores	Ponteiro para o texto (text)	Diretório-raiz
Contador de programa	Ponteiro para o dados (data)	Diretório de trabalho
Registrador de status	Ponteiro para pilha (stack)	Descritores de arquivo
Ponteiro para pilha		ID de usuário
Estado do processo		ID de grupo
Prioridade		
Parâmetros de escalonamento		
ID do processo (PID)		
Processo-pai		
Sinais		
Tempo de início		
Tempo de CPU usado		
Tempo de CPU usado dos filhos		
Próximo alarme		

Estado de execução

- O estado de execução de um processo é composto pelos valores de todos os registradores do processador em um dado instante
- Quando ocorre uma troca de processos, o estado de execução de um processo precisa ser salvo e o estado do novo processo restaurado

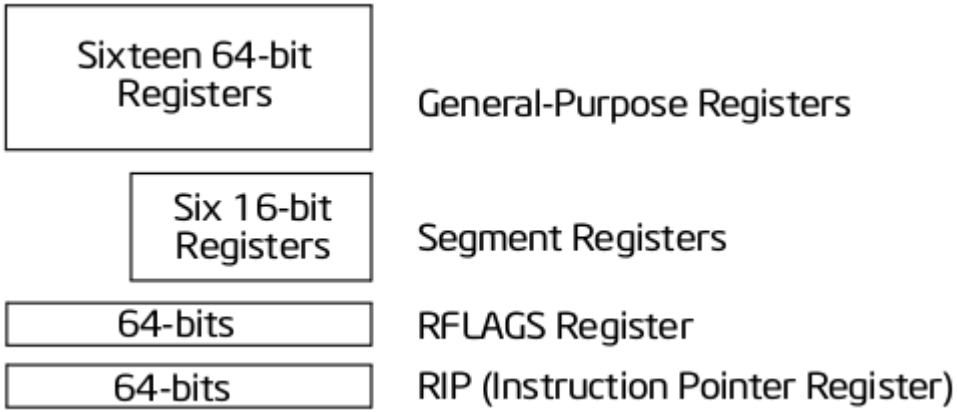
Trocas de contexto entre processos



Exercício

Considerando um processador com o conjunto de registradores da figura abaixo:

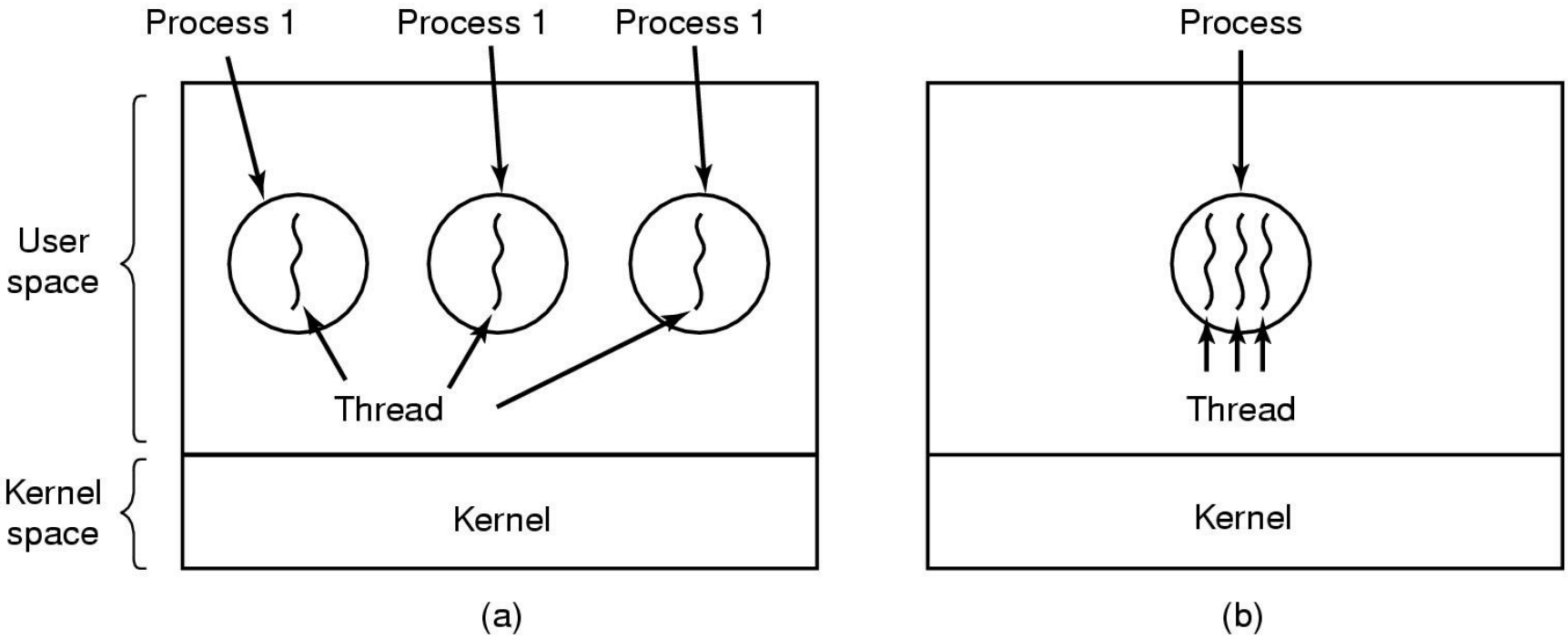
1) Quanto espaço é necessário na memória para que o estado de execução de um processo seja salvo?
2) Suponha que em um dado SO reserve para a tabela de processos 16MiB. Desconsiderando as demais entradas da tabela, quantos processos podem ser executados ao mesmo tempo por esse SO?
3) Suponha que salvar um registrador da memória ou carregá-lo da memória demore 2ns. Quanto tempo é necessário para trocar o contexto de execução de um processo para outro?



Threads

- **Threads são fluxos de de execução dentro de um processo**
- **Cada processo possui no mínimo 1 thread**
  - *main()*
- **Por quê usar threads?**
  - Aplicações podem possuir múltiplas tarefas
  - Simplicidade de criação/destruição
  - Desempenho nos casos em que há alto uso de IO
  - Paralelismo real em múltiplas CPU’s

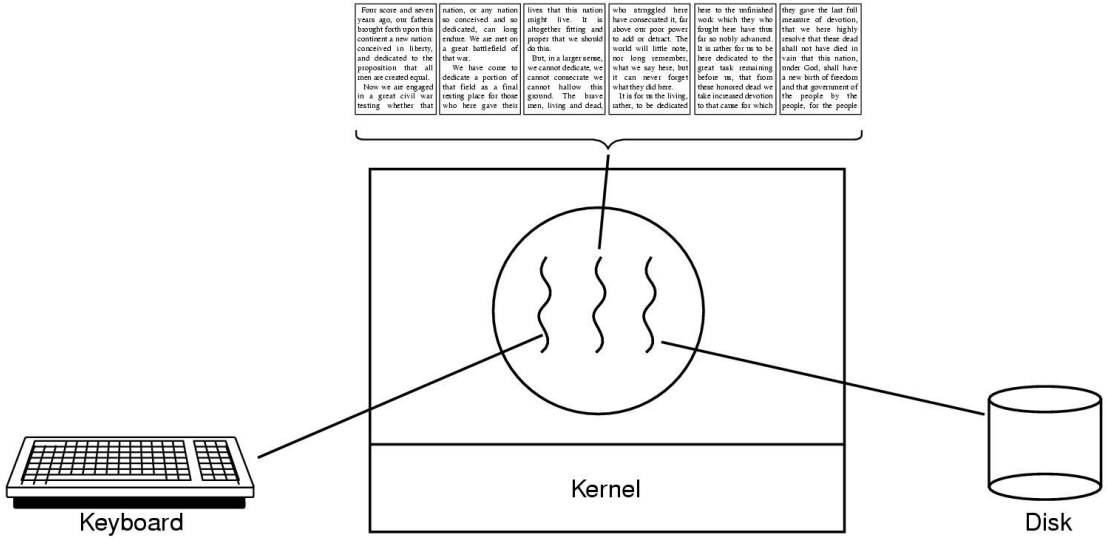
Threads



(a) 3 processos, cada um com 1 thread

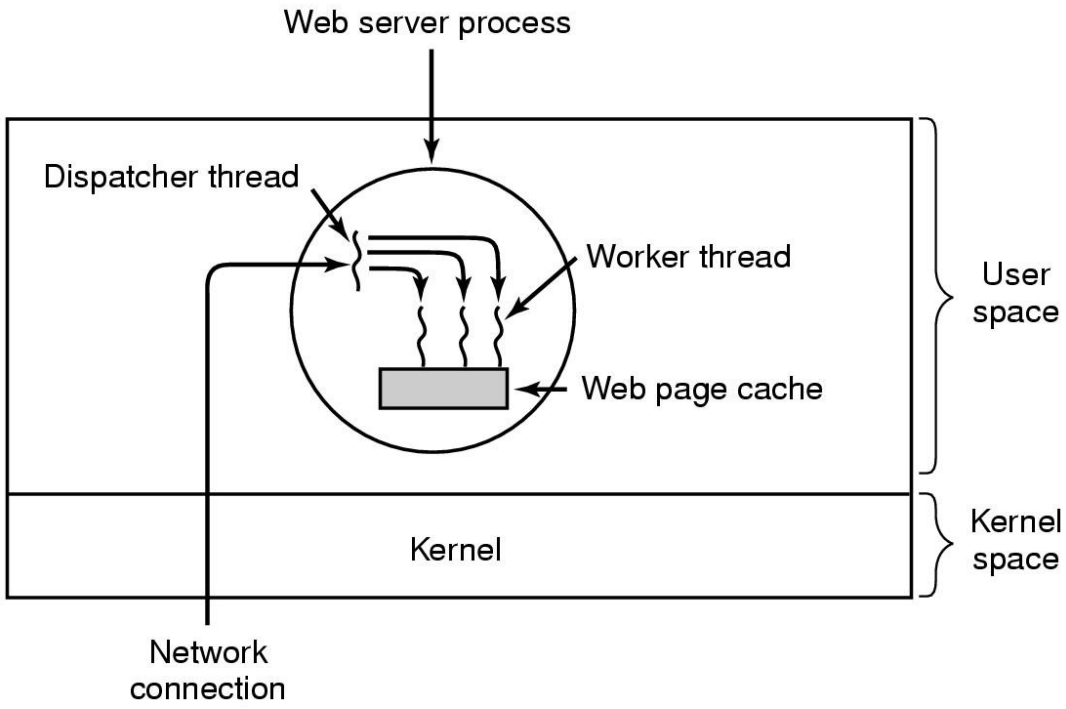
(b) Um processo com 3 threads

Exemplos de utilização de threads
Processador de texto



- **Processador de texto com 3 threads:**
  - GUI
  - Teclado
  - Disco (salvamentos automáticos)

Exemplos de utilização de threads
Servidor Web



- **Servidor Web Multithread:**
  - Modelo Despachante e Operário

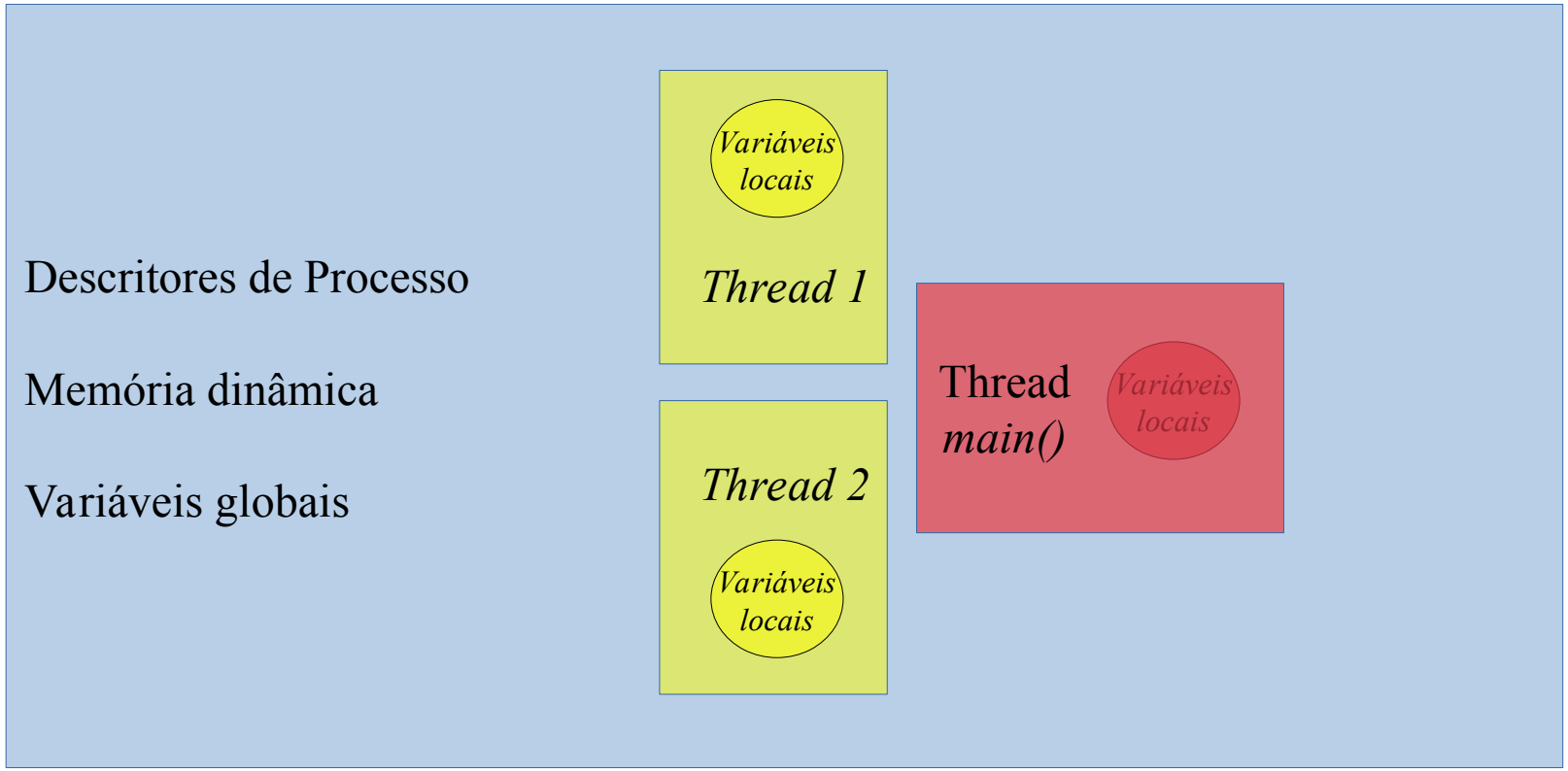
Modelo de Threads
Itens compartilhados e específicos

Propriedades de processos (compartilhados por todas as threads)	Propriedades de threads
Espaço de endereçamento	Contador de programa
Variáveis globais	Estado de execução (registradores)
Arquivos abertos	Pilha
Processos filhos	Estado de execução
Alarmes	
Sinais e tratadoras de sinais	
Informações	

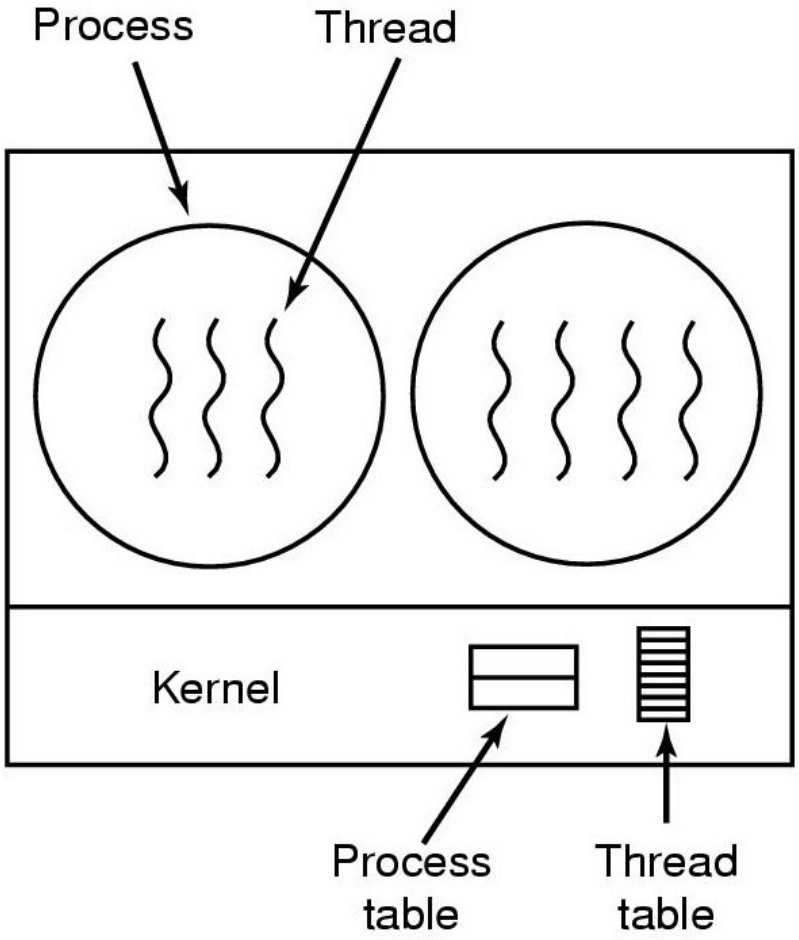
Modelo de Threads
Visão da memória

- **Todo programa possui a thread *main()***
- **Em C, a implementação de threads segue os mesmos padrões de funções e procedimentos**
- **A função só se torna uma nova *thread* quando invocada por funções específicas de criação de thread**
- **Cada thread deve possuir sua própria pilha**
  - Armazena as chamadas de procedimentos de cada thread
- **Exemplo:**
  - Thread *main()*
  - 2 threads criadas usando o procedimento *thread()*

Modelo de Threads
Visão lógica



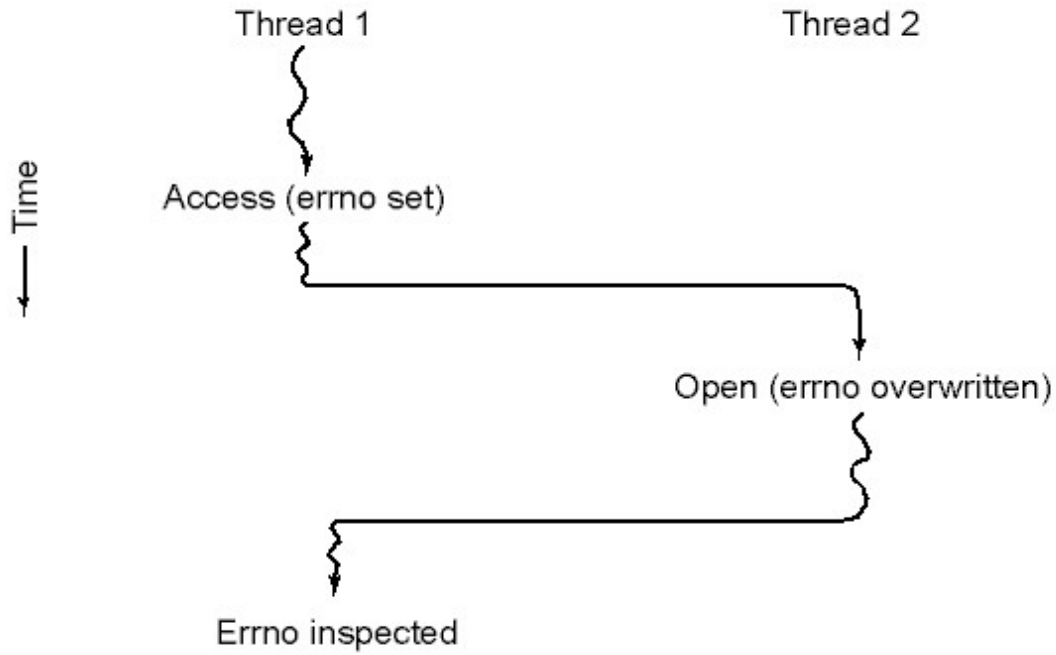
Implementação de Threads
Espaço de Kernel



Implementação de Threads
Espaço de Kernel

- **Vantagens:**
  - Não precisa de chamadas de sistema bloqueantes
- **Desvantagens:**
  - Mais custoso
  - Criação de filhos
- **Implementações modernas de threads em Sistemas Operacionais são baseadas nesse modelo**
  - Linux: criados a partir de filhos do processos
  - Cada thread é uma “entidade de escalonamento” no kernel
  - Mecanismos de sincronização funcionam majoritariamente no espaço de usuário

Programas multithread
Compartilhamento de variáveis



Condição de corrida!

- **Variáveis globais de sistema devem ser tratadas de maneira diferente em ambientes *multithread***
- **Exemplo: o que acontece com a variável *errno*?**

Programas multithread
Rotinas de bibliotecas de programação

- **Rotinas de bibliotecas devem seguir as seguintes regras (Regra dos 3 R's):**
  - Recursivas
  - Reentrantes
  - Relocáveis
- **Para programas *multithread*, é especialmente importante que elas sejam reentrantes.**
  - Exemplo crítico: funções de gerência de memória (por exemplo, *malloc*)
  - Funções devem ser *thread-safe*
    - *https://man7.org/linux/man-pages/man7/pthreads.7.html*

Referências

- **TANENBAUM, A. S. *Sistemas Operacionais Modernos. 3ª. ed.***
  - Capítulo 2: seções 2.1, 2.2.1 até 2.2.7, 2.3.1, 2.3.2
- **Slides originais de Andrew S. Tanenbaum**
  - http://www.cs.vu.nl/~ast/books/book_software.html

Exercícios

- **Capítulo 2:**
  - 1, 3, 4, 7, 8,10, 14, 17