



# Sistemas Operacionais

## Processos

# Processos

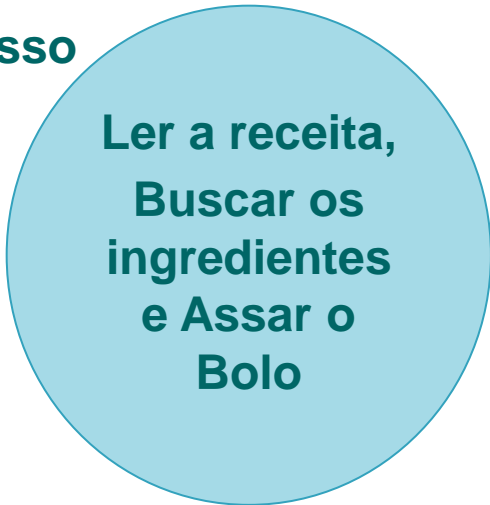
- Conceito fundamental para todos os sistemas operacionais
- Os processos mantêm a capacidade de operações concorrentes mesmo quando há apenas uma CPU disponível
- Um processo é um programa em execução

- Exemplo:

- Fazer um bolo

- Receita → Programa
    - Ingredientes → Dados de entrada
    - O cozinheiro → CPU

## Processo



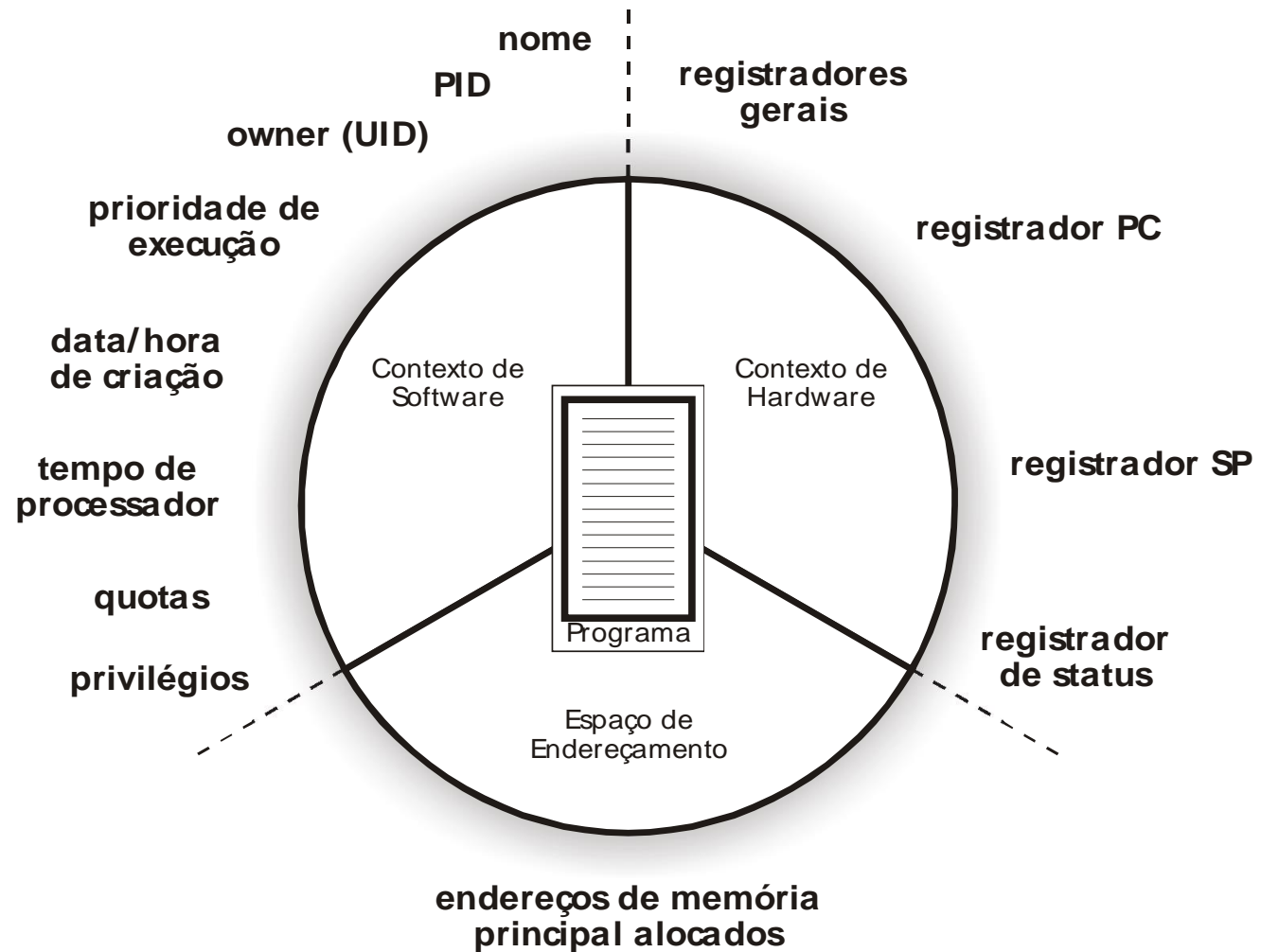
**Ler a receita,  
Buscar os  
ingredientes  
e Assar o  
Bolo**



# Processos

- Possui 3 elementos básicos: contexto de SW, contexto de HW e espaço de endereçamento
  - Contexto de SW – Características do processo como: identificação, número máximo de arquivos abertos, privilégios, etc
  - Contexto de HW – Constitui basicamente o conteúdo dos registradores
  - Espaço de endereçamento – É a área de memória pertencente ao processo, onde estarão armazenados as instruções e os dados para a execução.

# Processos



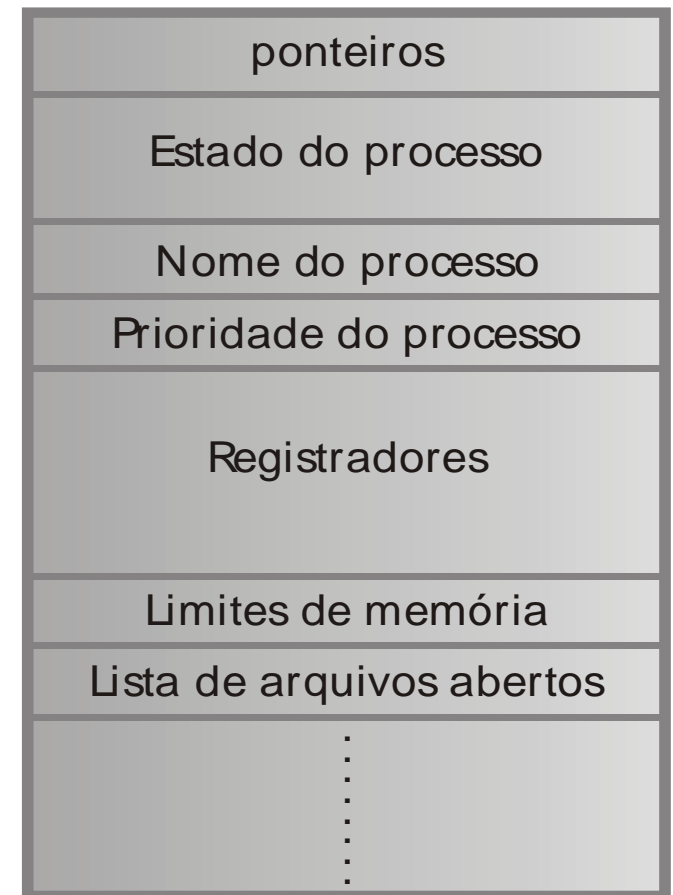


# Processos

- O que esperar do SO:
  - Alternar a execução de processos de forma a maximizar a utilização da CPU e fornecer tempo de resposta razoável
  - Alocar recursos a processos
  - Suportar criação de processos pelo usuário
  - Suportar comunicação entre processos
- Para gerenciar processos o SO precisa conhecer onde o processo está localizado e os atributos do processo

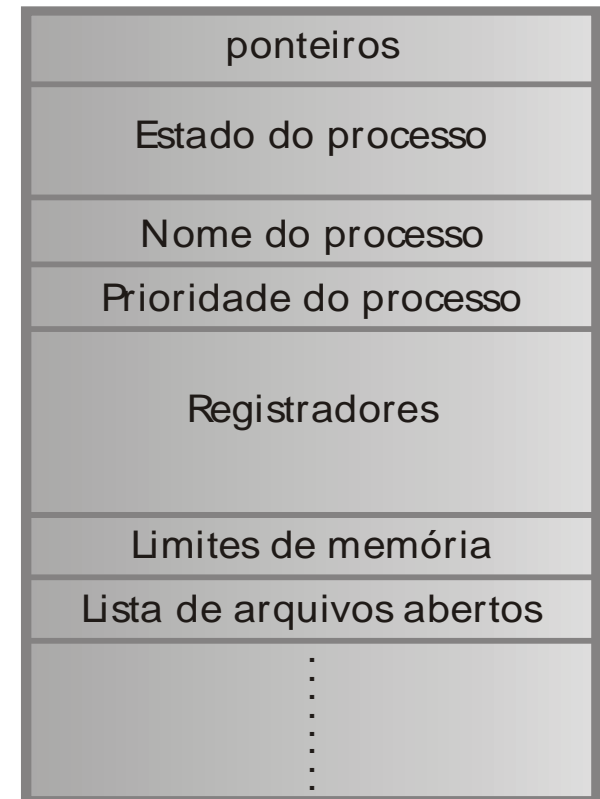
# Processos

- O SO materializa o processo através do bloco de controle de processo – PCB
- O PCB de todos os processos ativos residem na memória principal em uma área exclusiva do SO
- Usado para armazenar informações do processo



# Bloco de Controle do Processo

- Estado do Processo
- Número do Processo (PID)
- Contador de Programa (PC)
- Registradores da CPU
- Informações de gerenciamento da memória (registradores base e limite, tabelas de páginas ou de segmentos, etc)
- Informações de status de I/O (lista de arquivos abertos, lista de dispositivos alocados a um processo, etc)
- Informações de Contabilização (tempo de execução real e de CPU, etc)
- Informações de escalonamento (prioridade, ponteiros para filas de escalonamento, parâmetros)



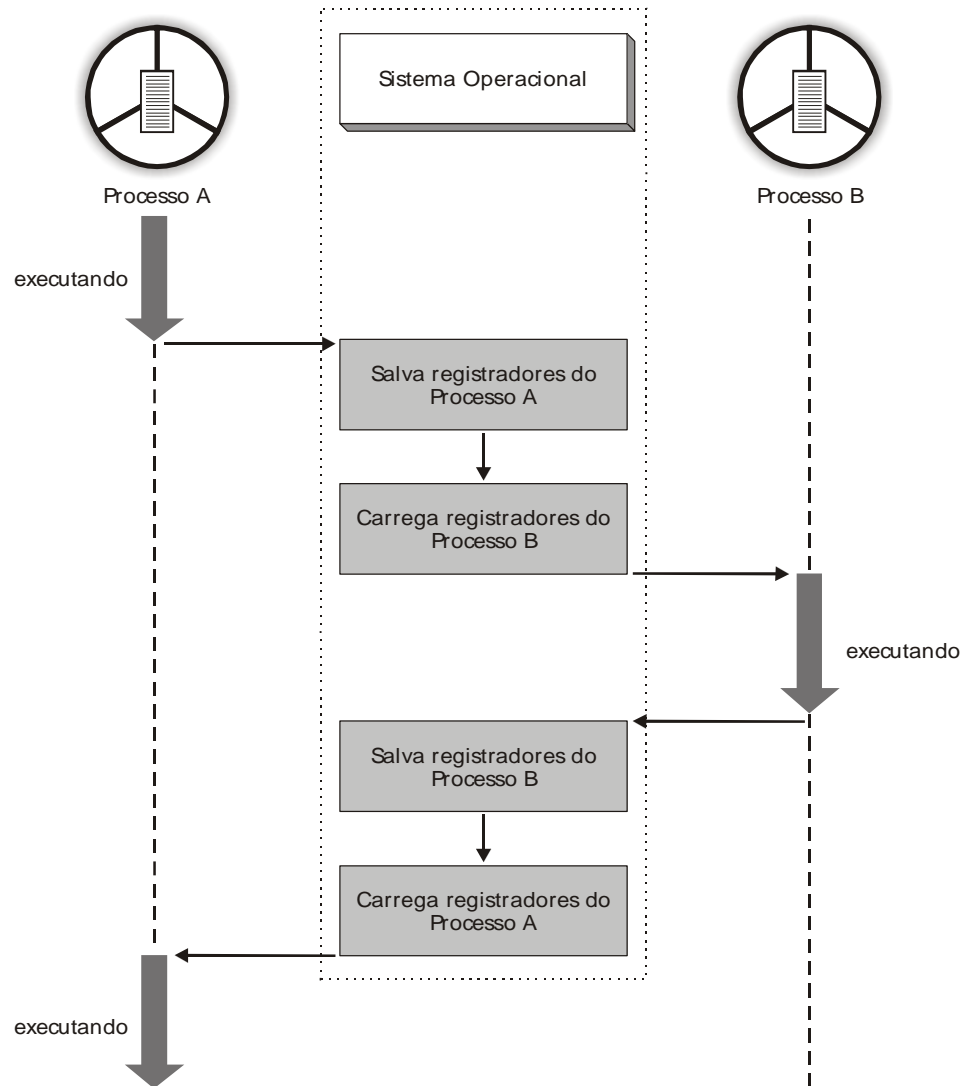


# Processos

- A gerência de processos é realizada por intermédio de chamadas às rotinas do SO, que realizam funções como criação, eliminação, sincronização, etc
- O SO toma grande cuidado para que processos independentes não afetem, de modo intencional ou por acidente, a correção de comportamento um do outro
  - Vários processos compartilham concorrentemente a CPU e outros recursos de HW de forma transparente
- A troca de um processo por outra é comandada pelo SO → Troca de Contexto



# Processos – Troca de Contexto





# Processos – Troca de Contexto

- Sobrecarga associada troca de contexto:
  - salva contexto do processo
  - atualiza bloco de controle do processo (PCB)
    - gravação do novo estado (pronto/bloqueado...)
  - move o processo (PCB) para a fila apropriada
  - escolhe novo processo para execução
  - atualiza PCB do novo processo e dados relativos a MP
  - restaura contexto do novo processo



# Processos – Criação

- O que faz o SO para criar processos?
  - constrói estruturas de dados
  - aloca espaço de endereçamento
- Quando o processo é criado?
  - Início do sistema
  - Requisição do usuário
  - Submissão de um job (batch)
  - Processo cria outros processos

# Processos – Término

- Um processo termina devido alguma das seguintes situações:
  - Saída normal (voluntária)
  - Saída por erro (voluntária)
    - Ex. Um compilador termina a execução quando vai compilar um arquivo que não existe
  - Erro fatal (involuntário)
    - Ex. Execução de instrução ilegal, divisão por zero...
  - Cancelamento por outro processo (involuntário)
    - Ex. Comando *kill* no linux

# Estado dos Processos

- Um processo muda de estado durante o seu processamento
  - Em função de eventos gerados pelo SO ou por ele próprio
- Exemplo:

```
cat arq1 arq2 | grep tree
```

- O processo gerado pelo comando grep irá buscar a palavra tree na concatenação dos arquivos 1 e 2
- O processo pode estar PRONTO para executar
- Entretanto, o processo pode ficar BLOQUEADO até que a entrada esteja disponível



# Estados dos Processos

- Novo – O processo está sendo criado
- Em execução – Instruções estão sendo executadas
- Em espera (ou bloqueado) – aguardando por algum evento (conclusão de I/O ou recebimento de um sinal)
- Pronto – esperando para ser atribuído a um processador
- Terminado – O processo terminou a sua execução

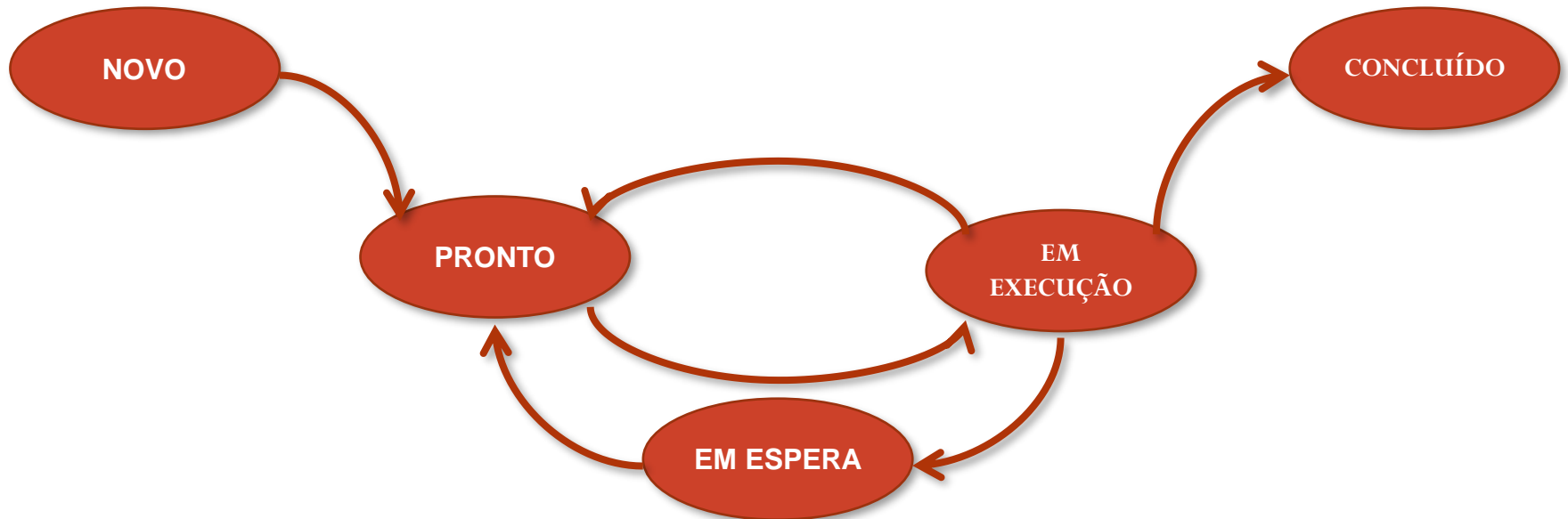
# Estados dos Processos

- Diagrama com 5 estados



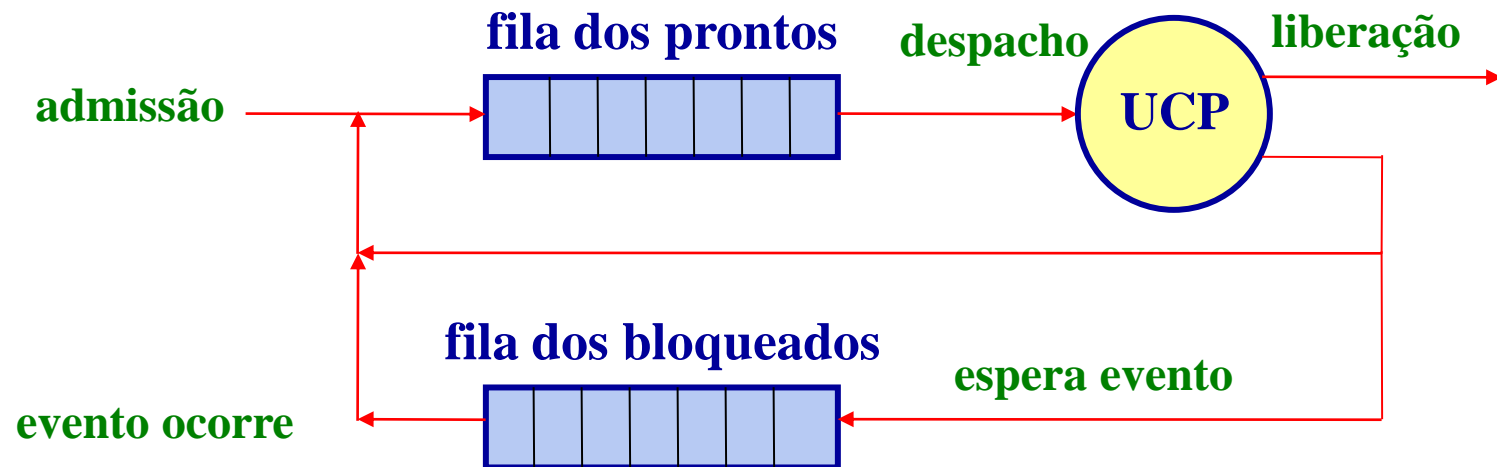
# Estados dos Processos

- Diagrama de Transições

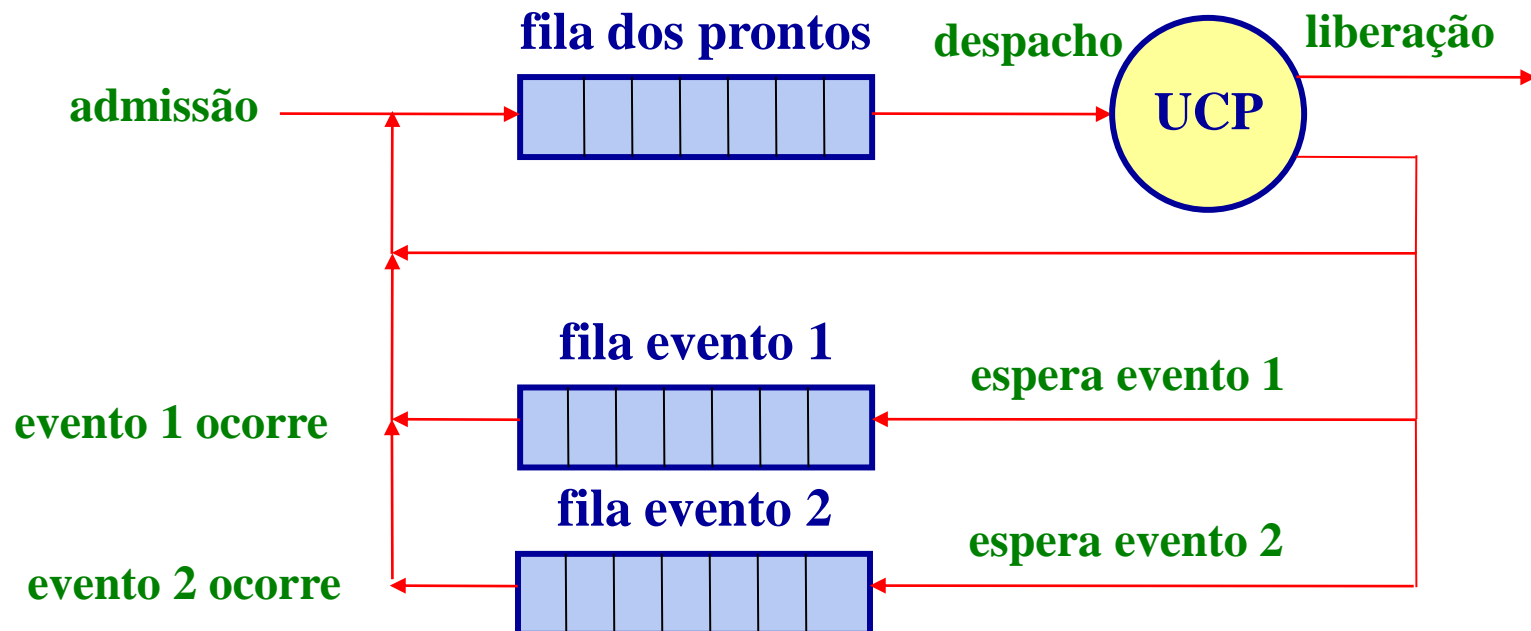




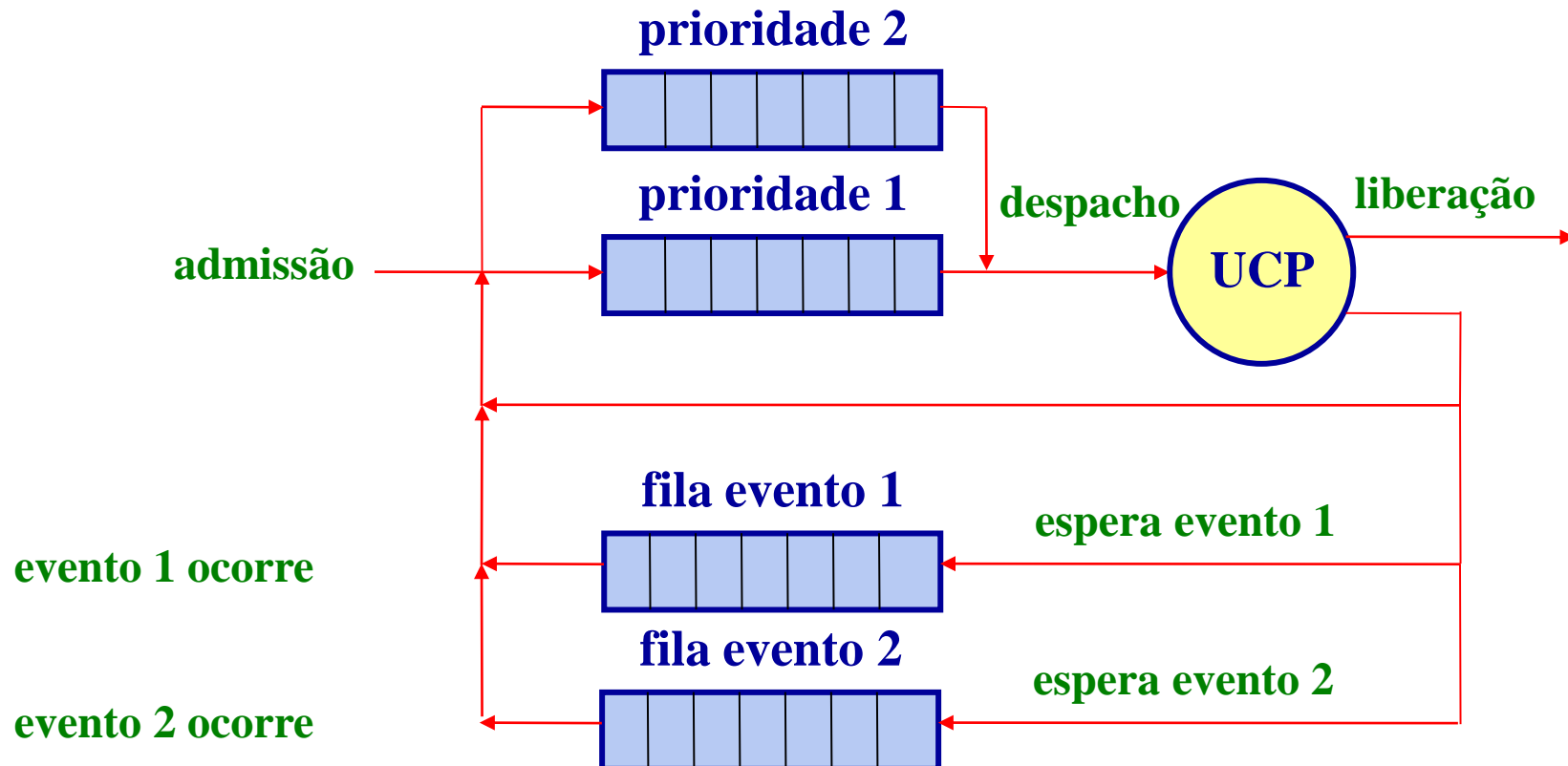
# Estados dos Processos – Filas (1)



# Estados dos Processos – Filas (2)



# Estados dos Processos – Filas (3)





# Estados dos Processos – Suspenso

- Vários processos em execução – necessidade de espaço em MP disponível
- Importante para implementação de memória virtual
- O processador é muito mais rápido que E/S: todos os processos podem estar bloqueados
- Necessidade de novo estado → Suspenso
  - Imagem do processo sai temporariamente da MP
  - SO seleciona um dos bloqueados para sair de MP
  - É um operação de E/S

# Estados dos Processos – Suspenso





# Mudança de Estado – 5 estados

- Quando o SO seleciona um processo para ganhar a CPU
  - Pronto → Execução
- Quando um processo termina a sua execução
  - Execução → Terminado
- Um processo perde a CPU pois expirou a sua fatia de tempo de execução
  - Execução → Pronto

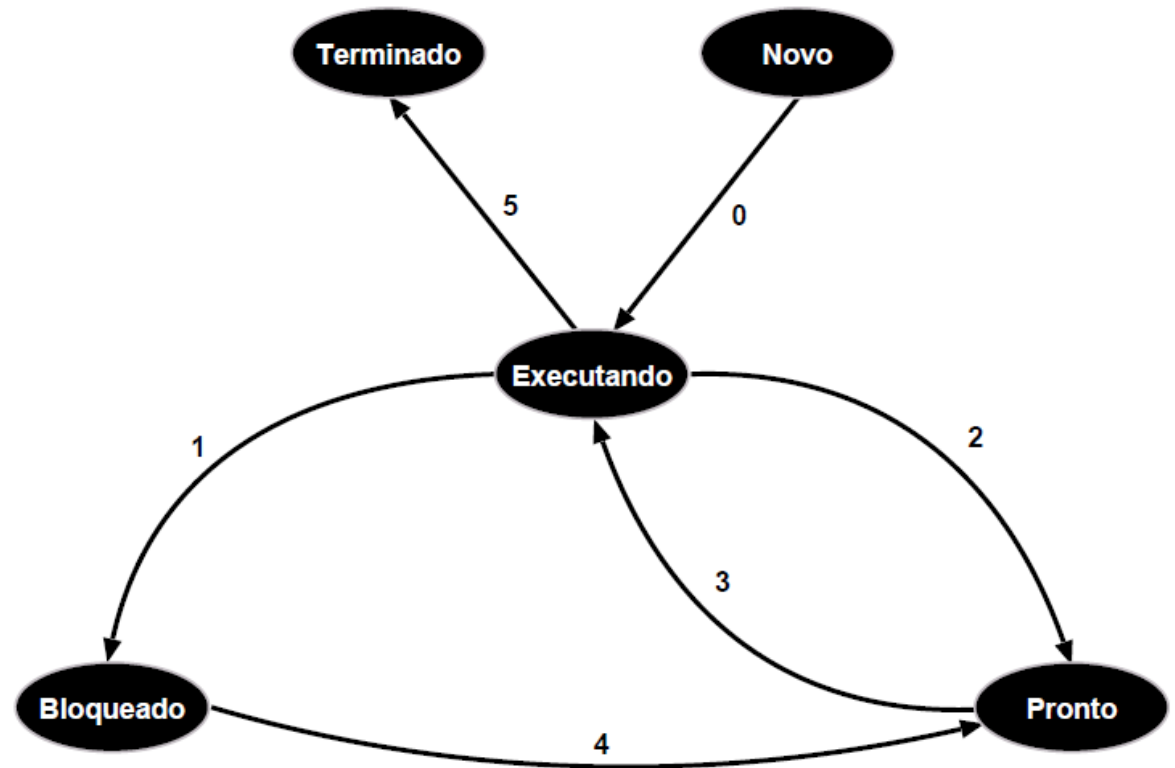


# Mudança de Estado – 5 estados

- Um processo está executando e faz uma chamada ao sistema pra execução de uma operação de E/S
  - Execução → Bloqueado
- Um processo está esperando por um evento (E/S, por exemplo) que ocorre
  - Bloqueado → Pronto
- Um processo é criado no sistema e se torna pronto para executar
  - Novo → Pronto

# Mudança de Estado – 5 estados

- Justifique se este diagrama de mudanças de estado está correto.



## Transições

- 0: O novo processo inicia a sua execução.
- 1: O escalonador escolhe um outro processo para executar.
- 2: O processo bloqueia esperando por algum evento.
- 3: O processo é desbloqueado pois o evento já ocorreu.
- 4: O processo volta a executar no processador.
- 5: O processo termina a sua execução.





# Mudança de Estado – 5 estados

- Mudanças possíveis, mas não usuais:
- Um processo pai está executando e termina. O que acontece com o estado dos seus processos filhos.
  - Pronto → Terminado
  - Bloqueado → Terminado

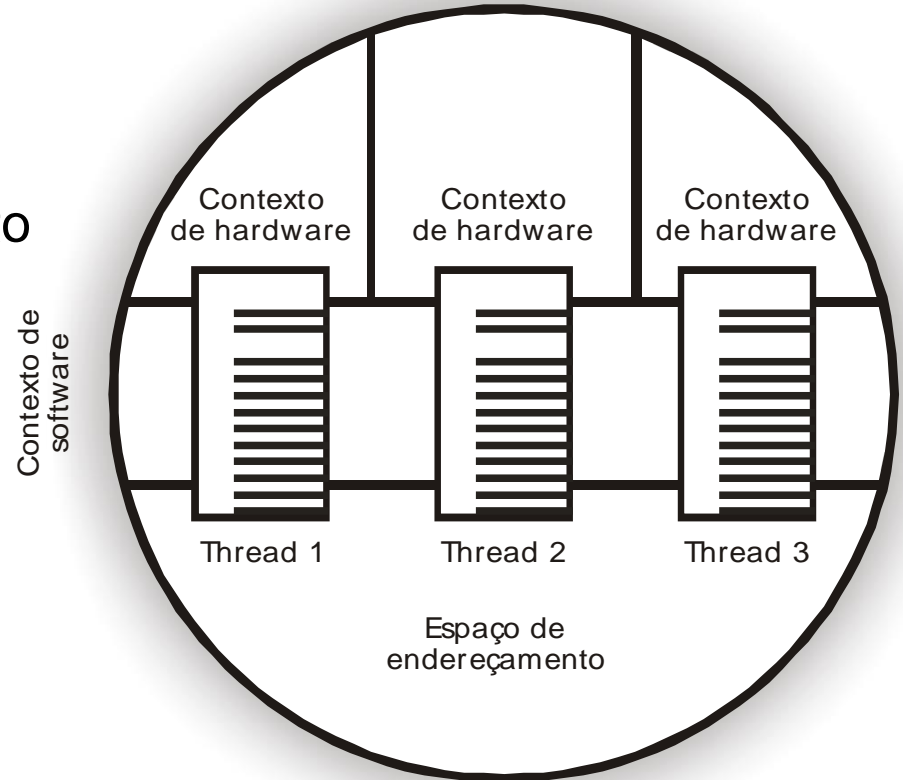
# Mudança de Estado – 3 estados

- Em um diagrama com 3 estados, na teoria, seria possível fazer 6 tipos de transições diferentes, contudo apenas 4 transições são mostradas.
- Existe alguma circunstância na qual alguma transição não ilustrada possa ocorrer?



# Threads

- No conceito de *multithread* um processo pode possuir vários fluxos de controle (*threads*)
  - *Threads* de um processo compartilham o mesmo espaço de endereçamento





# Threads

- Com múltiplos *threads* é possível projetar e implementar aplicações concorrentes de forma eficiente.
  - Um mesmo processo pode ter partes diferentes do seu código sendo executadas concorrentemente ou em paralelo.
- Os *threads* compartilham o processador da mesma forma que os processos e passam pelas mesmas mudanças de estado
  - Do mesmo modo, a CPU alterna rapidamente entre os *threads*



# Threads

- Vantagens no uso de *threads*
  - Menos tempo para criar um *thread* do que um processo filho
  - Menos tempo para finalizar um thread do que um processo
  - A troca de contexto é mais rápida entre *threads* do mesmo processo
    - Algumas CPUs possuem suporte de HW direto para *multithread* e permitem um chaveamento mais rápido
  - Mais eficiência no compartilhamento de dados através da memória compartilhada dentro de um mesmo processo



# *Threads*

## ■ Exemplos

### □ Navegador Web

- Um thread para exibir imagens ou texto
- Outro thread para recuperar dados da rede

### □ Processador de textos

- Um thread para exibir gráficos/reformatar um texto
- Outro para ler sequência de teclas do usuário
- Outro para verificação ortográfica e gramatical
- Outro para salvamento automático



# Threads

- O espaço de endereçamento dos *threads* de um processo é compartilhado
  - Um *thread* pode ler escrever ou apagar a pilha de execução de outro *thread*
  - *Threads* podem compartilhar um conjunto de recursos
  - Não há proteção entre *threads*
  - *Threads* devem cooperar e não competir
- Cada *thread* possui seu próprio contexto de hardware



# Threads

- Informações que não são estritamente necessárias para gerenciar múltiplos *threads* em geral são ignoradas
- Logo, proteger dados contra acesso inadequado por *threads* dentro de um único processo fica a cargo do desenvolvedor da aplicação
  - Requer esforço intelectual adicional
- Em muitos casos o *multithreading* resulta em ganho de desempenho
  - Por exemplo, as “trocas de contexto” são mais rápidas





# Threads – Implementação

1. Construir uma biblioteca de *threads* que é executada inteiramente em modo usuário:
  - ❑ Criar e terminar *threads* é barato
  - ❑ Chaveamento entre *threads* é rápido → basicamente somente os registradores de CPU precisam ser armazenados
  - ❑ Escalonamento é feito internamente
  - ❑ Uma chamada bloqueadora → bloqueia todo o processo ao qual o *thread* pertence

# Threads – Implementação

## 2. Implementação de threads no núcleo do sistema operacional

- Toda a operação de *thread* (criação, encerramento, sincronização, etc) terá que ser executada pelo núcleo
  - Requer chamadas ao sistema
  - O custo é mais alto
- Chavear contexto de *threads* pode ser tão caro quanto chavear processos
- Escalonamento feito pelo SO
- Chamada bloqueadora → bloqueia apenas o *thread*



# Threads – Implementação

- Abordagem híbrida: LWP (*Lightweight Process*)
  - Um LWP executa no contexto de um único processo (pesado)
    - Podem existir vários LWPs por processo
  - O sistema fornece um pacote de *threads* de nível de usuário que oferece às aplicações operações usuais de *threads*
    - Criação, término, exclusão mútua
    - Todas operações em *threads* são realizadas sem intervenção do núcleo
  - Cada LWP (nível núcleo) pode executar um *thread* (nível usuário)



# Threads – Implementação

- Abordagem híbrida: LWP (*Lightweight Process*)
  - A designação de um *thread* a um LWP é implícita e oculta ao programador
  - Quando o LWP encontra um *thread* executável ele chaveia o contexto para aquele *thread*
    - Quando um *thread* precisa bloquear devido a uma exclusão mútua, ela faz a administração necessária e chama a rotina de escalonamento
    - É feito um chaveamento para outro *thread* executável
    - O chaveamento neste caso é implementado no espaço do usuário



# Threads – Implementação

- Abordagem híbrida: LWP (*Lightweight Process*)
  - Quando um *thread* faz uma chamada bloqueadora de sistema
    - A execução muda de modo usuário para modo núcleo, mas continua no contexto do LWP corrente.
    - Se o LWP corrente não puder continuar, o SO pode chavear o contexto para outro LWP
  - Uma chamada bloqueante bloqueia um LWP, mas não os outros LWPs, que compartilham a tabela de *threads* entre si



# *Threads* – Sistemas Distribuídos

- Uma das vantagens do uso de *threads* é:
  - Proporcionar um meio conveniente para permitir chamadas bloqueadoras de sistema sem bloquear o processo inteiro no qual o *thread* está executando
- Esta propriedade torna os *threads* atraentes para o uso em sistemas distribuídos
  - Facilitam a comunicação, podendo manter múltiplas conexões lógicas ao mesmo tempo



# Clientes *Multithreads*

- Usados para ocultar latências de comunicação, separando *threads* de envio/recebimento de dados com *threads* de processamento da interface.
- Torna possível o recebimento de vários arquivos de uma página WEB ao mesmo tempo
- Torna possível o acesso a vários servidores (redundantes), que servirão os dados independentemente, gerando maior velocidade.



# Clientes *Multithreads*

- Como implementar *multithreading* em um cliente Web:
  - Um documento Web consiste em um grande número de objetos
  - A busca de cada objeto de uma página HTML será feita após estabelecimento de uma conexão TCP
    - Requisições são feitas sem que os objetos precedentes tenham chegado no cliente
    - Cliente é capaz de manipular diversos fluxos em paralelo através da utilização de *Threads*





# Clientes *Multithreads*

- Caso os dados estejam espalhados por diversas réplicas de servidores...
- A utilização de *threads* possibilita aos clientes estabelecerem diversas conexões, em paralelo, com o objetivo de disponibilizar um único documento
- Determina efetivamente que o documento Web inteiro seja totalmente exibido em tempo muito menor do que com um servidor não replicado

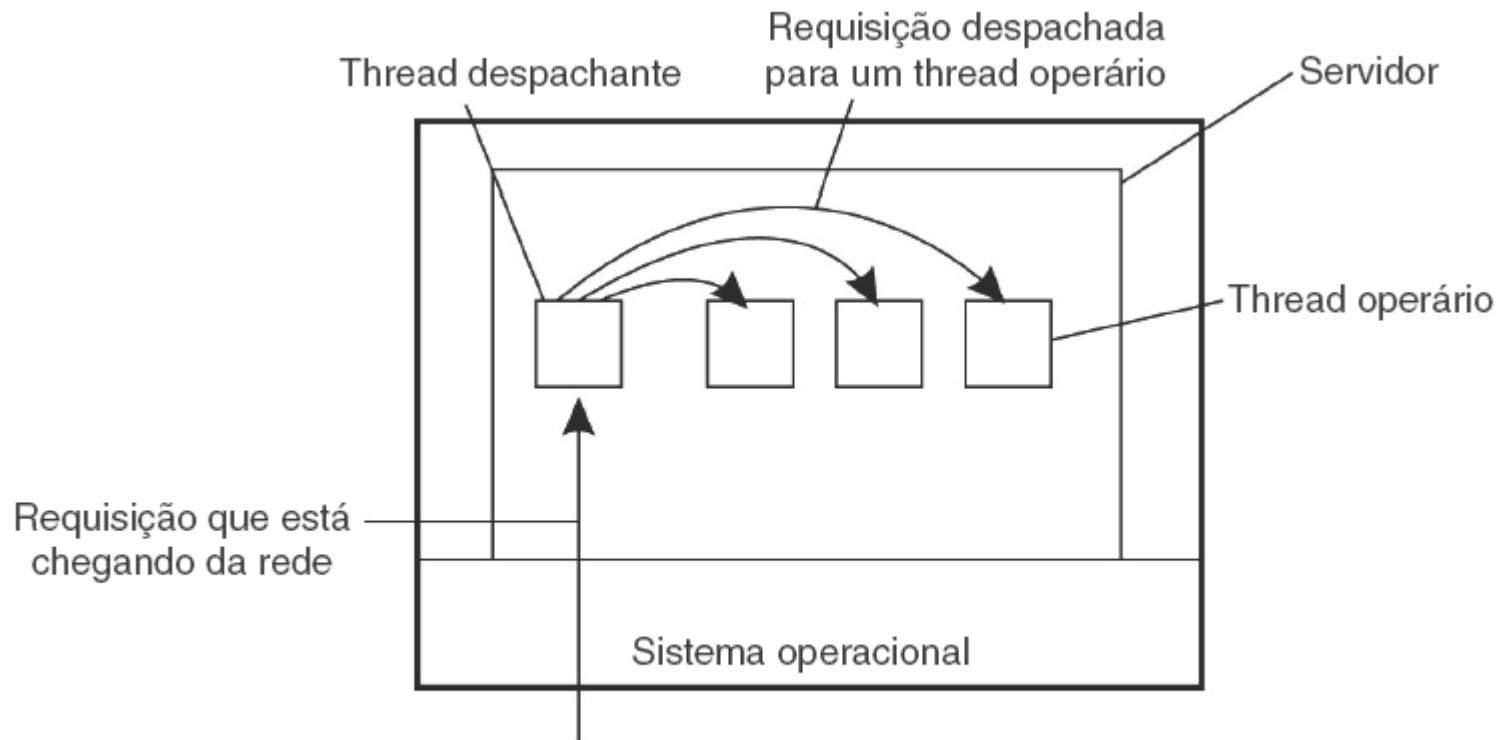


# Servidores *Multithreads*

- Além de simplificar o código do servidor, explora paralelismo para obter alto desempenho, mesmo em sistemas monoprocessadores
- Servidor de arquivos:
  - Espera uma requisição, executa e devolve a resposta
  - Uma organização possível é usar o modelo despachante/operário
    - Um *thread* despachante deve ler requisições que entram para operações de arquivos
    - O servidor escolhe um *thread* operário ocioso e lhe entrega a requisição

# Servidores *Multithreads*

- Modelo despachante/operário



**Figura 3.3** Servidor multithread organizado segundo modelo despachante/operário.



# Servidores *Multithreads*

- Suponha que o servidor de arquivos tenha sido implementado com ausência de *threads*
  - O servidor obtém uma requisição, examina e executa até a conclusão
  - Servidores *Monothread* não poderiam atender a um segundo usuário enquanto lêem disco!
  - A CPU fica ociosa, enquanto o servidor de arquivos estiver esperando pelo disco



# Exercício 1

Nesse problema você deverá fazer uma comparação entre ler um arquivo usando um servidor de arquivos monothread ou um servidor multithread.

Obter uma requisição para trabalho, despachá-la e fazer o resto do processamento necessário demora 15ms, considerando que os dados necessários estejam em uma cache na MP.

Se for preciso uma operação de disco, como acontece 1/3 das vezes, serão necessários mais 75ms, durante os quais a thread dorme.

Quantas requisições por segundo o servidor pode manipular se for monothread? E se for multithread?



## Exercício 2

*Tem sentido limitar a quantidade de threads  
em um servidor?*



## Exercício 3

*Há alguma circunstância na qual um servidor monothread pode ser melhor que um servidor multithread?*

# Resposta 1

- Cache hit – 15ms
- Cache miss – 90ms
- Servidor monothread:  $1/3 \times 90 + 2/3 \times 15 = 40$ 
  - Cada requisição gasta 40ms na média
  - O servidor poderá executar 25 requisições em 1 segundo
- No servidor multithread não precisa haver espera
  - Cada requisição gastará na média 15ms
  - O servidor poderá executar 66 requisições em 1 segundo





# Resposta 2

- SIM
- Apesar de serem mais leves do que os processos, os threads também requerem memória para implementar a sua pilha de execução → Threads em excesso consomem muita memória, prejudicando o trabalho do servidor
- Muitos threads podem levar a degradação do sistema, resultando em “*page thrashing*” → muito cache miss → muita troca de contexto



# Resposta 3

- SIM
- Se o servidor for totalmente CPU-bound, não há necessidade de múltiplos *threads*
- Exemplo:
  - Considere uma lista para assistência de números telefônicos para uma área com 1 milhão de pessoas (nome e telefone)
  - Se cada registro possui 64 caracteres → o Banco de Dados total possuirá 64Mbytes e pode ser mantido na memória do servidor para agilizar a consulta

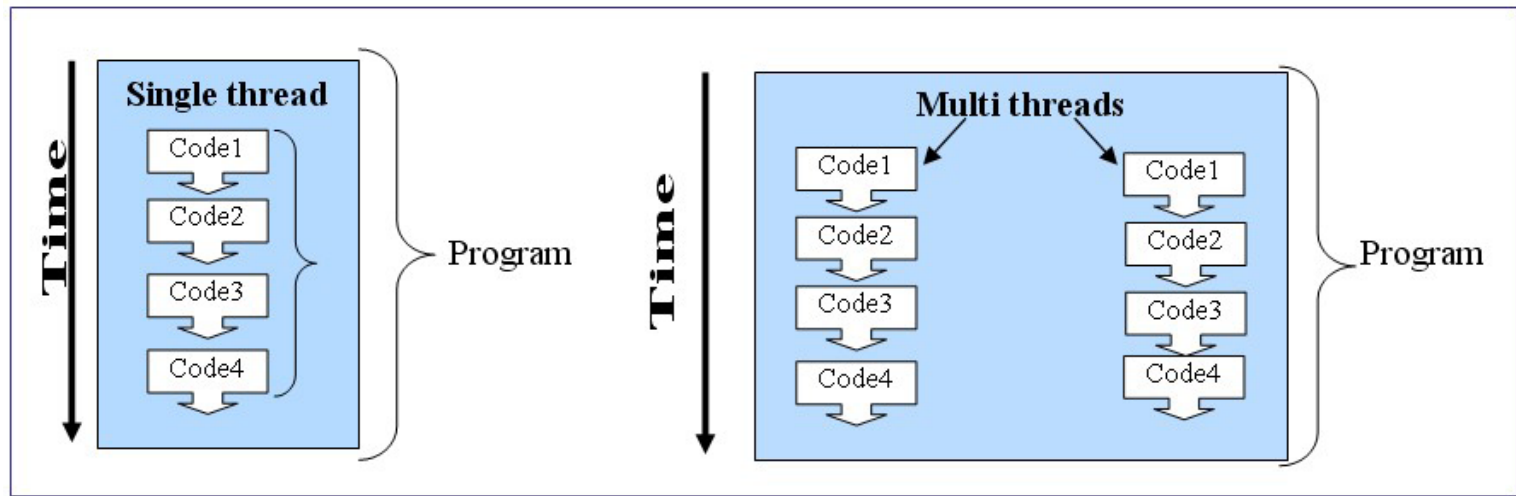
A decorative graphic on the left side of the slide, consisting of a grid of squares in shades of purple, teal, and dark green, arranged in a pattern that tapers to the right.

# Programação Concorrente

– THREADS –

# Programação Concorrente

- Atividade de construir programas que incluem linhas de controle distintas que podem executar simultaneamente
  - As diferentes linhas de controle cooperam para a execução da tarefa principal





# Programação Concorrente – Por quê?

- Computadores modernos são multicore → vários processadores em um único chip
  - O sistema operacional gerencia a atribuição das diferentes linhas de execução para os diferentes processadores
- Para explorar todo o potencial dessas novas arquiteturas o programador precisa desenvolver programas concorrentes (com várias linhas de execução)



# *pthread*s

- Criando programas concorrentes em C, usando a biblioteca *pthread*.c

- `int pthread_create (thread,  
attr,  
start_routine,  
arg);`

- `void pthread_exit (status);`

- `int pthread_join (thread,  
status);`



# Exemplo

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

#define NTHREADS 10

void *PrintHello (void *arg) {

    printf("Hello World\n");
    pthread_exit(NULL);

}
```

# Exemplo

```
int main(void) {

    pthread_t tid_sistema[NTHREADS]; //identificadores das threads
    int i;

    for(i=0; i<NTHREADS; i++) {
        printf("--Cria a thread %d\n", i);
        if (pthread_create(&tid_sistema[i], NULL, PrintHello, NULL)) {
            printf("--ERRO: pthread_create()\n");
            exit(-1);
        }
    }

    for(i=0; i<NTHREADS; i++) {
        printf("--Termina a thread %d\n", i);
        if (pthread_join(tid_sistema[i], NULL)) {
            printf("--ERRO: pthread_join()\n");
            exit(-1);
        }
    }

    pthread_exit(NULL);
}
```





# Exemplo – Com tomada de Tempo

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

#include "timer.h"

#define NTHREADS 10

void *PrintHello (void *arg) {

    printf("Hello World\n");
    pthread_exit(NULL);

}
```

# Exemplo – Com tomada de Tempo

```
int main(void) {

    pthread_t tid_sistema[NTHREADS]; //identificadores das threads
    int i;
    double inicio, fim;

    GET_TIME(inicio);
    for(i=0; i<NTHREADS; i++) {
        printf("--Cria a thread %d\n", i);
        if (pthread_create(&tid_sistema[i], NULL, PrintHello, NULL)) {
            printf("--ERRO: pthread_create()\n");
            exit(-1);
        }
    }
    for(i=0; i<NTHREADS; i++) {
        printf("--Termina a thread %d\n", i);
        if (pthread_join(tid_sistema[i], NULL)) {
            printf("--ERRO: pthread_join()\n");
            exit(-1);
        }
    }
    GET_TIME(fim);
    printf("--Tempo total: %f\n", fim-inicio);
    pthread_exit(NULL);
}
```