Sistemas Operacionais

Processos e threads

Guilherme Meira

Agenda

1. Processos

2. Threads

Processos

- **Processos** são o conceito mais central de qualquer sistema operacional
- Um processo representa um programa em execução
- Processos permitem que um computador realize varias tarefas ao mesmo tempo, mesmo quando apenas uma CPU está disponível

Processos

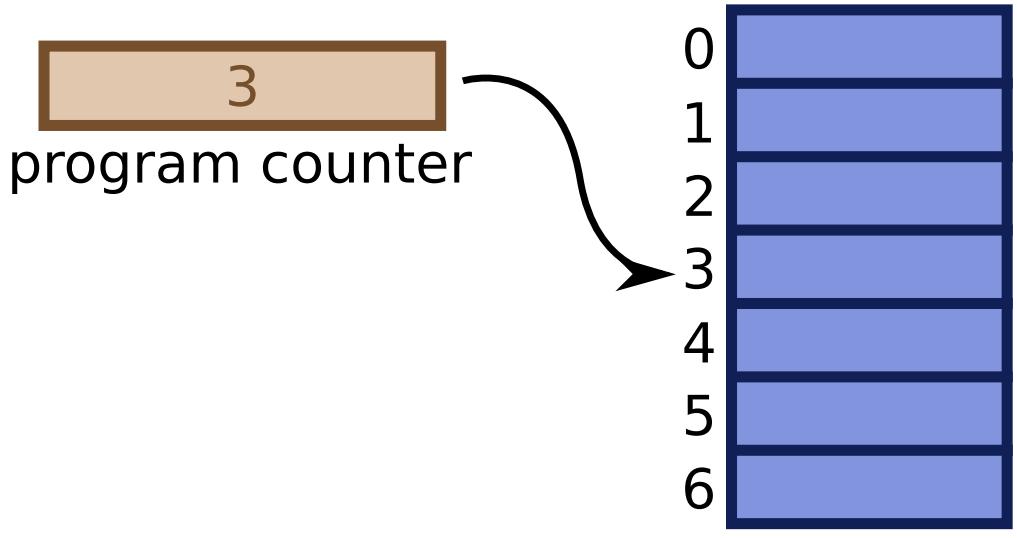
- Computadores modernos estão sempre realizando diversas tarefas ao mesmo tempo:
  - Navegando na web
  - Tocando música
  - Checando novos e-mails
  - Buscando por vírus
  - Copiando arquivos
  - Dentre várias outras
- O sistema operacional faz com que a CPU troque muito rapidamente entre processos, dando a impressão de que estão executando em paralelo (**pseudoparalelismo**)
- Sistemas com vários processadores serão estudados mais adiante

Processos

- Computadores modernos estão sempre realizando diversas tarefas ao mesmo tempo:
  - Navegando na web
  - Tocando música
  - Checando novos e-mails
  - Buscando por vírus
  - Copiando arquivos
  - Dentre várias outras
- O sistema operacional faz com que a CPU troque muito rapidamente entre processos, dando a impressão de que estão executando em paralelo (**pseudoparalelismo**)
- Sistemas com vários processadores serão estudados mais adiante

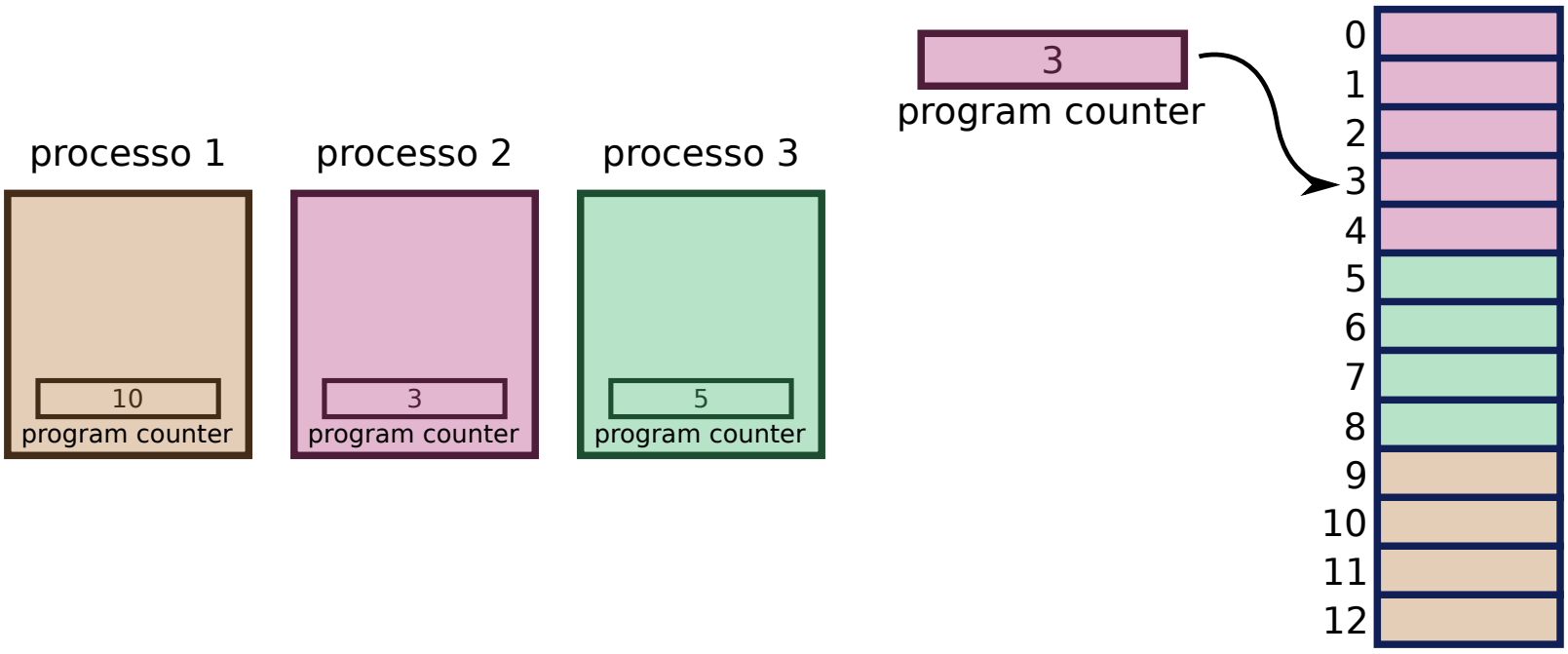
Processos

- O **program counter (PC)** é um registrador da CPU que aponta para a posição de memória que contém a próxima instrução a ser executada



Processos

- Uma CPU tem apenas um program counter
- O sistema operacional precisa armazenar o program counter de cada processo na memória
- Quando um processo vai receber um tempo da CPU, seu program counter é carregado no program counter real da CPU



Processos

- O sistema operacional determina quando um processo vai executar e por quanto tempo ele vai ter a posse da CPU
- Para a maioria dos processos isso não é relevante
- Isso pode afetar processos com requerimentos de tempo real (ex: tocar áudio e video em sincronia)

Processos

- Qual a diferença entre um **processo** e um **programa**?

Processos

- Qual a diferença entre um **processo** e um **programa**?
- Uma analogia:
  - Uma pessoa está fazendo um bolo seguindo uma receita
  - A pessoa é o **processador**, a receita é o **programa**, a ação de fazer o bolo é o **processo**
  - A pessoao é interrompida pelo filho que foi picado por uma abelha
  - Ele anota onde parou na receita e pega um livro de primeiros socorros para ajudar o filho
  - Aqui, trocamos de um processo para o outro
  - Quando ele termina de cuidar do filho, continua o processo anterior de onde parou

Processos

- Um **programa** é uma sequência de passos que pode ficar guardada em disco sem fazer nada
- Um **processo** é a atividade de executar as instruções de um programa
- O mesmo programa pode ser executado mais de uma vez ao mesmo tempo por processos diferentes

Processos

Criação de processos

- Durante a execução de um sistema operacional, processos podem ser criados por diversos motivos:
  - Inicialização do sistema
  - Execução de uma chamada de sistema por um processo que já esteja rodando
  - O usuário requisita a criação de um processo
  - Inicio de um lote

Processos

Criação de processos

- Quando o sistema operacional é iniciado, vários processos são criados
- Alguns são processos de primeiro plano, que interagem com o usuário
- Outros rodam em segundo plano, e não estão associados a um usuário em específico, mas a uma tarefa
- Processos que ficam rodando em plano de fundo são chamados de **daemons**

Processos

Criação de processos

- Em UNIX, processos são criados pela chamada de sistema `fork`
- Essa chamada cria um clone do processo que a chamou
- O novo processo, então, pode executar a chamada `execve` para carregar um novo programa
- No Windows, processos são criados pela chamada `CreateProcess`
- Tanto em Windows como em UNIX, cada processo tem seu próprio espaço de endereçamento (não é possível acessar a memória de outro processo)

Processos

Término de processos

- Após serem criados, processos eventualmente são terminados, por diversos motivos:
  - Saída normal (voluntário)
  - Saída com erro (voluntário)
  - Erro fatal (involuntário)
  - Morto por outro processo (involuntário)

Processos

Hierarquia de processos

- Em UNIX, processos possuem uma hierarquia
- Um processo e seus filhos formam um **grupo**. Quando o usuário envia um sinal a um processo, todos os processos do grupo recebem o sinal (veremos sinais com mais detalhes adiante)
- Quando o UNIX inicializa, um processo especial chamado `init` cria todos os demais processos. O `init` é o pai de todos os processos no UNIX
- O Windows não possui o conceito de hierarquia de processos

Processos

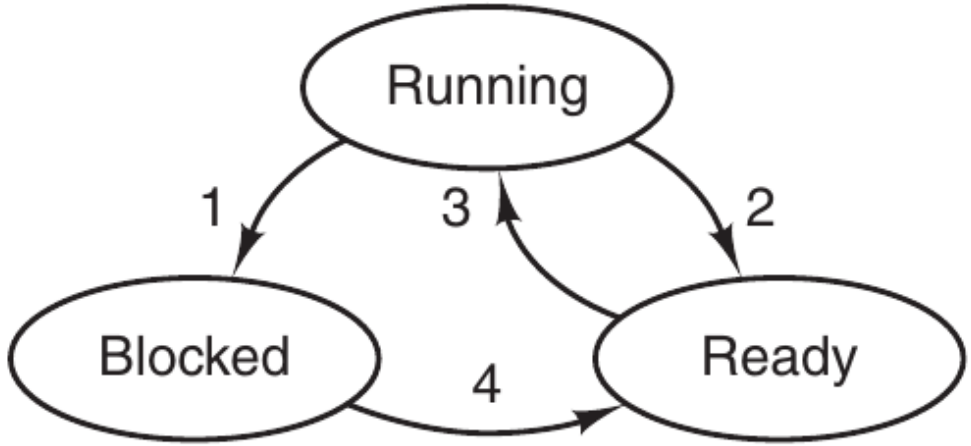
Estados de um processo

- Um processo pode estar em um de três estados:

Executando o processo está utilizando a CPU no momento

Pronto o processo está pronto para rodar, mas o sistema operacional não o deu a posse da CPU

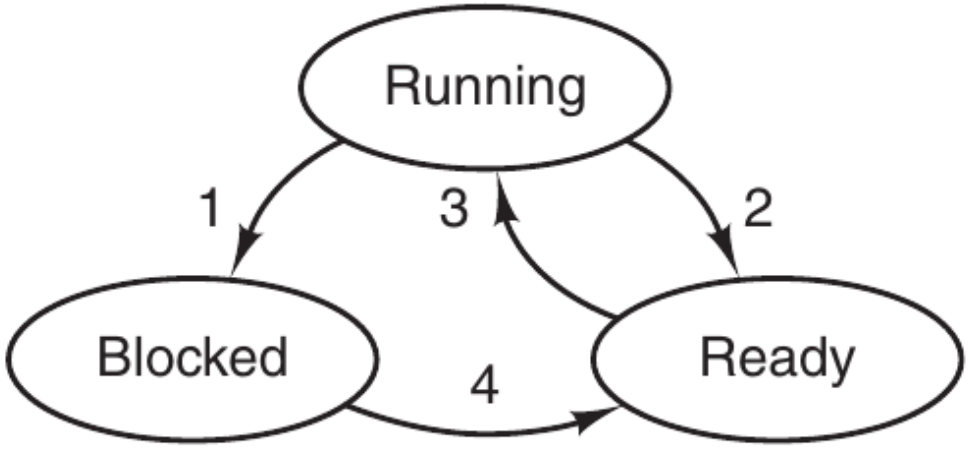
Bloqueado o processo está esperando algum evento externo para poder rodar



Processos

Estados de um processo

- As transições no diagrama representam:
  1. Processo bloqueia para esperar por algum evento
  2. Escalonador do sistema escolhe outro processo para colocar na CPU
  3. Escalonador escolhe este processo
  4. O evento que estava sendo aguardado acontece



Processos

Estados de um processo

- O estado **bloqueado** é fundamentalmente diferente dos demais
- Enquanto nos estados **executando** e **pronto** o processo deseja ser executado, no estado **bloqueado**, mesmo com a CPU disponível o processo não pode rodar
- Um processo pode entrar nesse estado, por exemplo, para esperar uma entrada vinda do teclado ou de outro processo
- As transições entre os estados **executando** e **pronto** são geradas por uma parte do sistema operacional chamada de escalonador (veremos mais detalhes adiante)

Processos

Implementação de processos

- O sistema operacional mantém uma tabela de processos contendo todas as informações sobre o estado do processo
- Essas informações são salvas quando o processo vai de **executando** para **pronto** ou **bloqueado** e são restauradas quando o processo for voltar a rodar
- As informações armazenadas na tabela de processos variam de sistema para sistema, mas em geral possuem informações sobre o próprio processo, sobre gerência de memória e sobre gerência de arquivos

Processos

Modelando multiprogramação

- **Multiprogramação** é o uso da CPU por vários processos
- Ela permite que aumentemos a utilização da CPU
  - Se um processo faz computação 20% do tempo, 5 processos executando em multiprogramação manteriam a CPU ocupada 100% do tempo
  - Modelo pouco realista! Assume que os processos nunca estarão aguardando por I/O ao mesmo tempo

Processos

Modelando multiprogramação

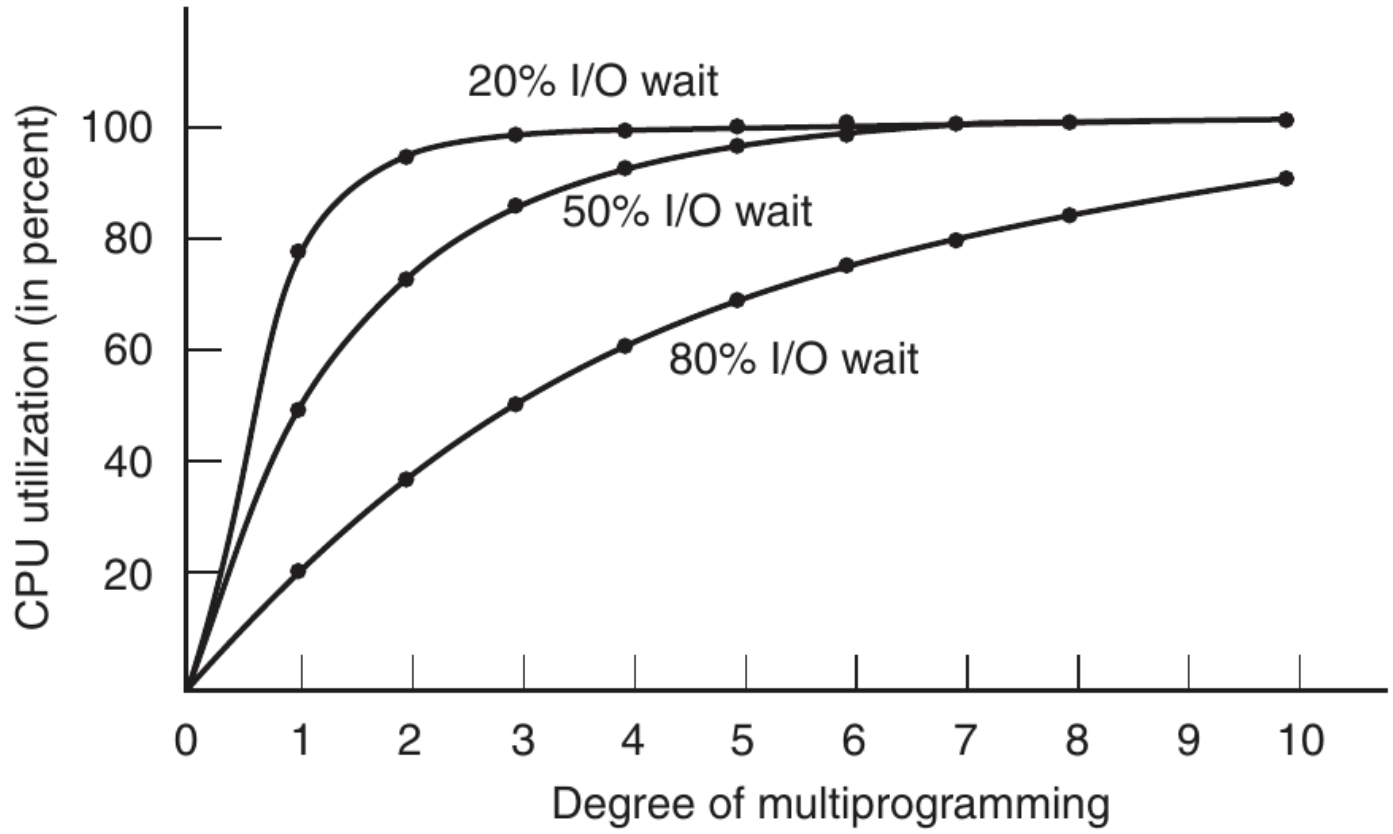
- Podemos utilizar probabilidade para encontrarmos um modelo melhor:
  - Um processo passa uma fração p do tempo esperando por I/O
  - Com n processos na memória, a probabilidade de que todos os processos estejam esperando por I/O (isto é, a CPU está ociosa) é p^n
  - A utilização da CPU é, então, calculada por:

$$utilizacao_{CPU} = 1 - p^n$$

Processos

Modelando multiprogramação

- Assim, podemos plotar a utilização estimada da CPU em função do número de processos em execução:



Processos

Modelando multiprogramação

- Este modelo também é uma aproximação, pois assume que os processos são independentes entre sí
- Quando dois processos estão no estado **pronto**, apenas um pode utilizar a CPU e o outro terá que esperar, portanto, existe uma dependência entre os processos
- Poderíamos chegar a modelos mais complexos utilizando Teoria das Filas, mas esse modelo é suficiente para fazermos algumas predições

Processos

Modelando multiprogramação

Suponha que um computador tenha 8GB de memória. O sistema operacional ocupa 2GB e cada processo executando na máquina também ocupa 2GB.

- Quantos processos podem executar simultaneamente nesta máquina?

Processos

Modelando multiprogramação

Suponha que um computador tenha 8GB de memória. O sistema operacional ocupa 2GB e cada processo executando na máquina também ocupa 2GB.

- Quantos processos podem executar simultaneamente nesta máquina?
  - Três
- Se o tempo de espera por I/O é de 80%, qual é a utilização da CPU?

Processos

Modelando multiprogramação

Suponha que um computador tenha 8GB de memória. O sistema operacional ocupa 2GB e cada processo executando na máquina também ocupa 2GB.

- Quantos processos podem executar simultaneamente nesta máquina?
  - Três
- Se o tempo de espera por I/O é de 80%, qual é a utilização da CPU?
  - $1 - 0.8^3 \approx 49\%$
- Se adicionarmos mais 8GB de memória, qual será a nova utilização da CPU?
  - Poderemos ter 7 processos em paralelo: $1 - 0.8^7 \approx 79\%$

Processos

Modelando multiprogramação

Ou seja, adicionar 8GB de memória aumentou a ocupação da CPU em 30%. E se adicionássemos mais 8GB?

Processos

Modelando multiprogramação

Ou seja, adicionar 8GB de memória aumentou a ocupação da CPU em 30%. E se adicionássemos mais 8GB?

- Podemos ter até 11 processos: $1 - 0.8^{11} \approx 91\%$
- Ganhamos apenas mais 12% de ocupação da CPU

Agenda

1. Processos

2. Threads

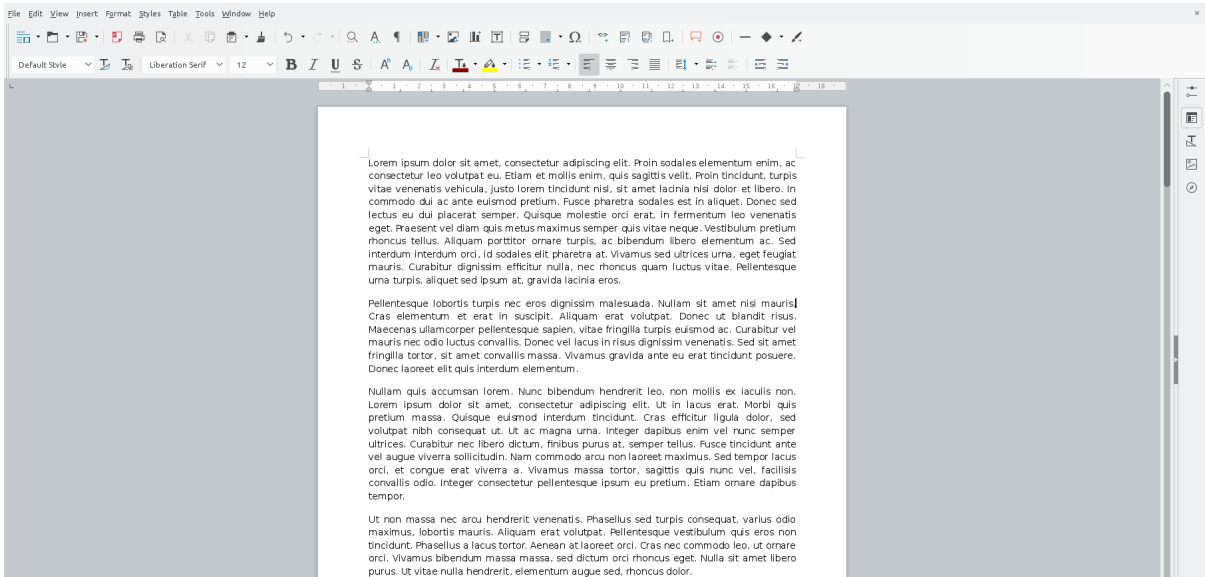
Threads

- Até então, todo o código dentro de um processo executava sequencialmente
- Threads permitem realizar paralelismo dentro de um mesmo processo
- O paralelismo é feito da mesma forma que em processos: alternando qual thread tem a posse da CPU
- Principal diferença: **espaço de endereçamento**
  - Cada processo tem seu próprio espaço de endereçamento e não pode acessar a memória dos demais processos
  - Todas as threads dentro de um processo compartilham o mesmo espaço de endereçamento e podem acessar os mesmos dados

Threads

Por que usar threads?

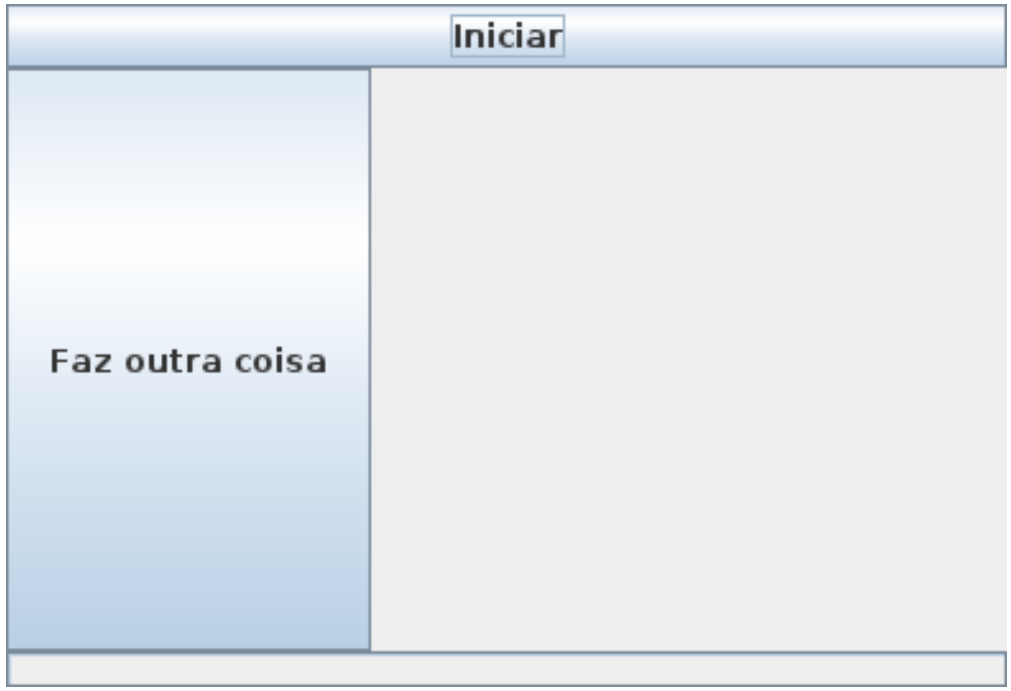
- Valem os mesmos argumentos para o uso de processos:
  - Aumentar a utilização da CPU
  - Em máquinas com mais de uma CPU, há um ganho de performance com o paralelismo
- Além disso, threads são mais fáceis de criar e destruir
  - Criar uma thread pode ser de 10 a 100 vezes mais rápido que criar um processo



Threads

Por que usar threads?

- Programas interativos em geral utilizam uma thread dedicada a tratar eventos do usuário. Tarefas lentas são feitas em uma thread separada.



Threads

Por que usar threads?

- Servidores que precisam atender usuários simultaneamente
  - Exemplo: servidor web
  - Uma thread recebe as requisições e atribui cada uma a uma thread disponível em um pool de threads

Threads

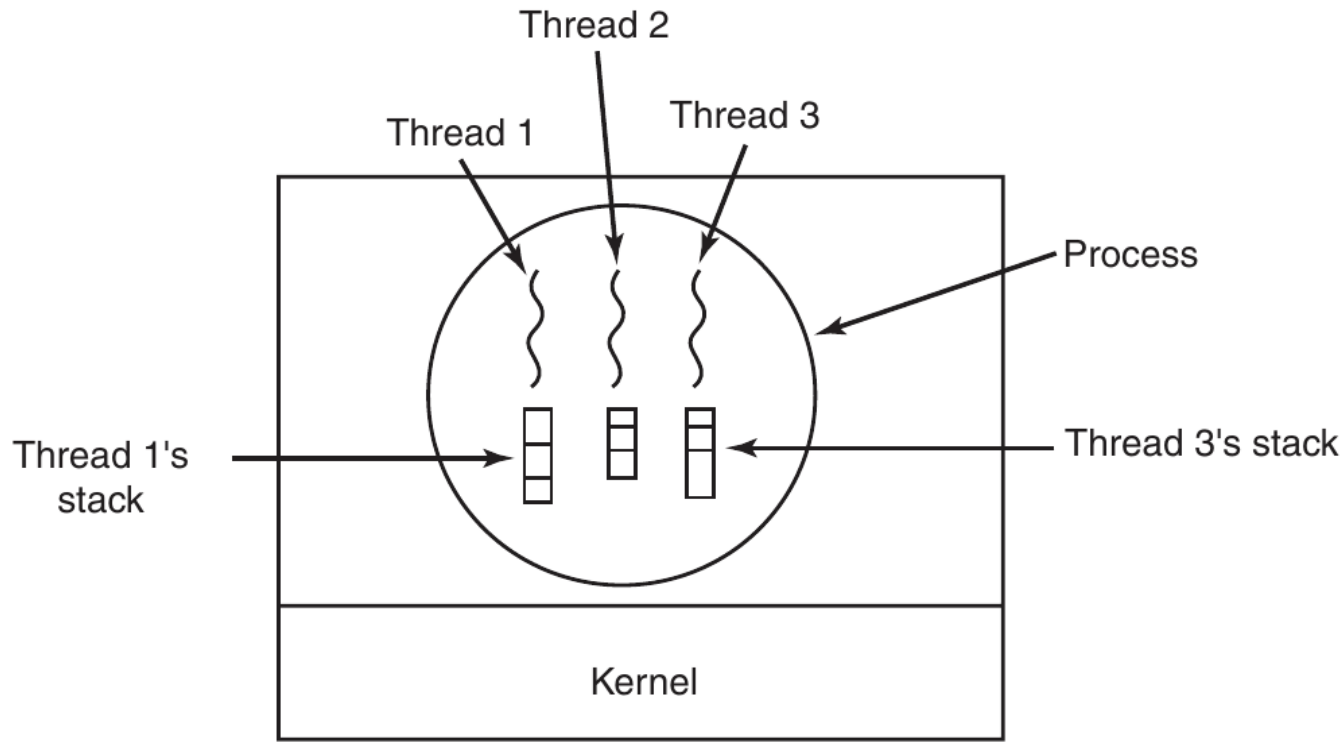
Por que usar threads?

- Hoje, ganha popularidade o uso servidores que utilizam uma única thread para evitar o overhead do uso de threads

Threads

Modelo de uma thread

- Cada thread pode seguir uma linha de execução diferente das demais. Portanto, cada uma precisa ter sua própria pilha (stack)



Threads

Criando threads

- Padrão IEEE 1003.1c define a biblioteca de threads **Pthreads**.
- Para utilizá-lo, basta incluir a `pthread.h`:

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

- Para criar uma thread:

```
int pthread_create(pthread_t *thread,
        const pthread_attr_t *attr,
        void *(*start_routine) (void *),
        void *arg);
```

Threads

Criando threads

```
int pthread_create(pthread_t *thread,
        const pthread_attr_t *attr,
        void *(*start_routine) (void *),
        void *arg);
```

- `pthread_t` é uma estrutura que armazena informações sobre a thread. Passamos um ponteiro para ela
- A função retorna um inteiro (zero se tudo correu bem, ou outro valor caso contrário)

Threads

Criando threads

```
int pthread_create(pthread_t *thread,
        const pthread_attr_t *attr,
        void *(*start_routine) (void *),
        void *arg);
```

- `pthread_attr_t` Armazena atributos da thread:
  - Tamanho da pilha da thread, opções de escalonamento, prioridade, etc...
  - Em geral, passamos `NULL` para utilizar os valores padrão

Threads

Criando threads

```
int pthread_create(pthread_t *thread,
        const pthread_attr_t *attr,
        void *(*start_routine) (void *),
        void *arg);
```

- Ponteiro para função que a thread começará a executar
  - A função recebe como parâmetro um ponteiro `void` e também tem um ponteiro `void` como valor de retorno

Threads

Criando threads

```
int pthread_create(pthread_t *thread,
        const pthread_attr_t *attr,
        void *(*start_routine) (void *),
        void *arg);
```

- O argumento que será passado para a função de entrada da thread

Threads

Criando threads

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

#define NUM_THREADS 10

void* entradaThread(void *idThread) {
        long id = (long) idThread;
        printf("Olá, mundo! Eu sou a thread %ld\n", id+1);
        pthread_exit(NULL);
}
```

Threads

Criando threads

```
int main(int argc, char *argv[]) {
        pthread_t threads[NUM_THREADS];
        long status, i;

        for(i = 0; i < NUM_THREADS; i++) {
                printf("Criando a %lda thread...\n", i+1);
                status = pthread_create(&threads[i], NULL,
                        entradaThread, (void*) i);

                if(status != 0) {
                        printf("Erro ao criar a thread: %ld\n", status);
                        exit(-1);
                }
        }
        return 0;
}
```

Threads

Outras funções das pthreads

- Encerrar uma thread:

```
void pthread_exit(void *retval);
```

  - Se a função `main` retorna da forma usual (`return`) o processo é encerrado mesmo se outras threads estiverem rodando
  - Para continuar a execução do processo, podemos encerrar a `main` com `pthread_exit`
  - O valor passado como parâmetro é o valor de retorno da thread (veremos como obtê-lo)

Threads

Outras funções das pthreads

- Manipular a estrutura de atributos:

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

- Liberar a CPU para outras threads:

```
int pthread_yield(void);
```

Threads

Outras funções das pthreads

- Esperar uma thread terminar:

```
int pthread_join(pthread_t thread, void **value_ptr);
```

  - O valor retornado pela thread em `pthread_exit` pode ser colocado em `value_ptr`
  - A função retorna 0 se tudo correu bem ou outro valor caso contrário

Threads

Criando threads

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

#define NUM_THREADS 10

void* entradaThread(void *idThread) {
        long id = (long) idThread;
        long numero = rand()%100;
        printf("Olá, mundo! Eu sou a thread %ld. Eu sorteei %ld!\n",
                id+1, numero);
        pthread_exit((void*)numero);
}
```

Threads

Criando threads

```
int main(int argc, char *argv[]) {
        srand(time(NULL));
        pthread_t threads[NUM_THREADS];
        long status, i;

        for(i = 0; i < NUM_THREADS; i++) {
                printf("Criando a %lda thread...\n", i+1);
                status = pthread_create(&threads[i], NULL,
                        entradaThread, (void*) i);

                if(status != 0) {
                        printf("Erro ao criar a thread: %ld\n", status);
                        exit(-1);
                }
        }
```

Threads

Criando threads

```
        for(i = 0; i < NUM_THREADS; i++) {
            long sorteado;
            status = pthread_join(threads[i], (void**) &sorteado);

            if(status != 0) {
                printf("Erro ao esperar thread: %ld\n", status);
                exit(-1);
            }

            printf("A thread %ld sorteou %ld!\n", i+1, sorteado);
        }
        return 0;
}
```

Threads

Implementando threads

Onde é melhor implementar threads? Dentro ou fora do kernel?

Threads

Implementando threads

- Fora do kernel (espaço de usuário):
  - Implementadas na forma de uma biblioteca dentro do próprio processo
  - Não depende de suporte do sistema operacional (o processo é visto como tendo apenas uma thread)
  - A biblioteca de threads é responsável por decidir qual thread vai executar, sem utilizar funções do kernel

Threads

Implementando threads

- Fora do kernel (espaço de usuário):
  - **Vantagens:**
    - » Mais eficiente (não precisa chamar o kernel)
    - » Maior customização
    - » Escalam melhor (não dependem de tabelas internas do kernel)

Threads

Implementando threads

- Fora do kernel (espaço de usuário):
  - **Desvantagens:**
    - » Chamadas bloqueantes (ex: `read`) bloqueiam todo o processo. Isso pode ser resolvido transformando as chamadas em não bloqueantes (envolve alterar o sistema operacional)
      Também podemos utilizar uma chamada como a `select` para verificar se a chamada irá bloquear. Envolve criar um **wrapper** para todas as chamadas bloqueantes
    - » Page faults
    - » A thread precisa voluntariamente ceder a CPU (não temos interrupção de clock)

Threads

Implementando threads

- Dentro do kernel:
  - O kernel cuida do processo de criação e destruição de threads
  - Quando um processo quer criar uma thread, simplesmente realiza uma chamada de sistema
  - Podemos contornar o maior custo da criação de threads com uma **thread pool**

Threads

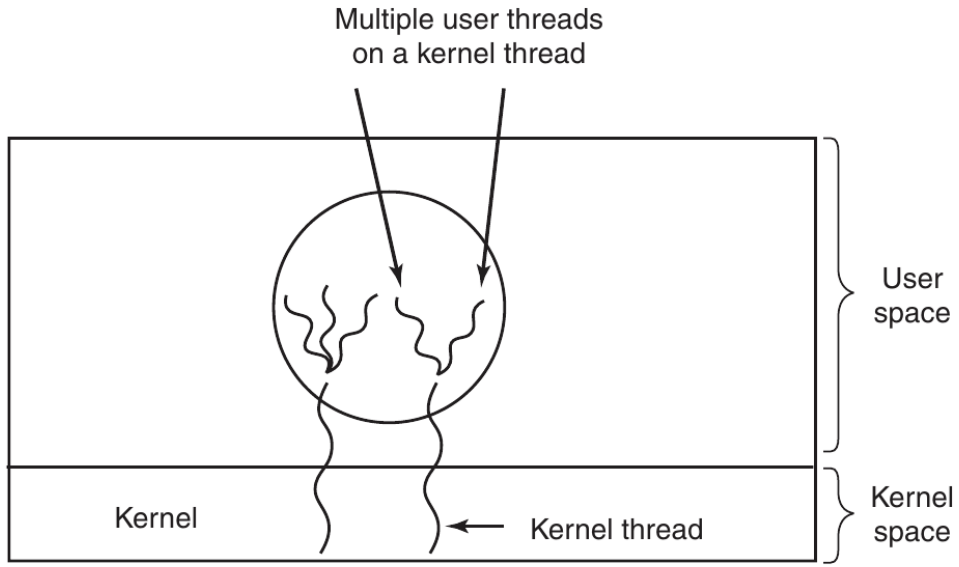
Implementando threads

- Dentro do kernel:
  - **Vantagens:**
    - » Não é necessário alterar as chamadas de sistema
    - » O kernel pode lidar corretamente com os page faults
  - **Desvantagens:**
    - » Mais custosas de criar e destruir (podemos contornar com thread pools)

Threads

Implementando threads

- Modelos híbridos:
  - Tentam combinar as vantagens dos dois modelos e reduzir as desvantagens
  - Utilizam threads do kernel, e dentro de cada uma delas podem existir várias threads de usuário



Threads

Implementando threads

- Qual modelo é utilizado pelas pthreads?
  - **Depende!** pthreads são uma interface, cada sistema operacional pode implementar de uma maneira
  - Em Linux, são implementadas com threads de kernel
- No passado, a máquina virtual Java (JVM) utilizava threads de usuário (green threads), mas hoje são utilizadas threads de kernel

Threads

Programando com múltiplas threads

- O uso de múltiplas threads introduz problemas que não existem quando se usa uma única thread
- **Problema 1:** variáveis globais

```
int saque(int valor) {
  if(saldo >= valor) {
    saldo = saldo-valor;
    return AUTORIZADO;
  } else {
    return RECUSADO;
  }
}
```

```
void deposito(int valor) {
  saldo = saldo + valor;
}
```

Threads

Programando com múltiplas threads

- **Problema 2:** reentrância
  - Quando uma thread é interrompida no meio de uma função e a mesma função é chamada por outra thread

```
int saque(int valor) {
  if(saldo >= valor) {
    saldo = saldo-valor;
    return AUTORIZADO;
  } else {
    return RECUSADO;
  }
}
```

Threads

Programando com múltiplas threads

- **Problema 3:** sinais
  - Qual thread deve tratar determinado sinal?
- **Problema 4:** gerência de pilha
  - Se um processo estoura a pilha, o sistema operacional pode alocar mais espaço
  - Se o kernel não tem conhecimento das pilhas de cada thread, ele não pode realocá-las