Threads

MCTA026-13 - Sistemas Operacionais

Emilio Francesquini
e.francesquini@ufabc.edu.br

2019.Q1

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de **Sistemas Operacionais na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Este material foi baseado nas ilustrações e texto preparados por Silberschatz, Galvin e Gagne [SGG] e Tanenbaum e Bos [TB] detentores do copyright.
- Estes slides contém alguns textos e figuras preparados por Dror Bereznitsky.

Introdução

Roteiro

UFABC

- Visão geral
- Programação para multicores
- Modelos de multithreading
- Bibliotecas de threads
- Threading implícito
- Problemas com threads
- Exemplos

Objetivos

UFABC

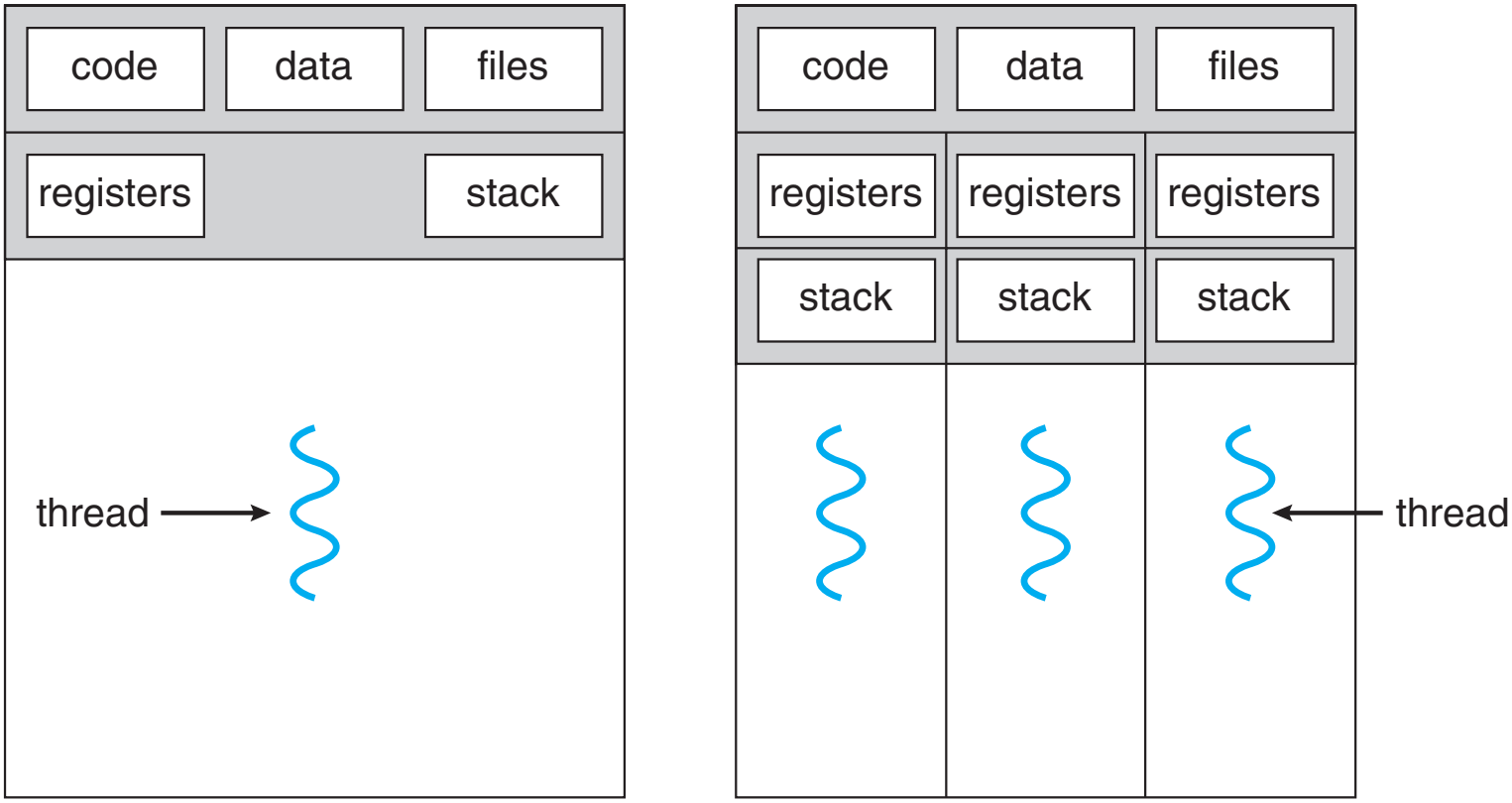
- Apresentar o conceito de thread - uma unidade fundamental para utilização da CPU, base para sistemas multithreaded
- Discutir a API de Pthreads, Windows e Java
- Explorar diversas alternativas para o uso de threading implícito
- Mostrar o uso de threads no Windows e Linux

Motivação

UFABC

- A maior parte das aplicações modernas são multithreaded
- Threads rodam dentro de uma aplicação
- Múltiplas tarefas em uma mesma aplicação podem ser implementadas como threads
  - Atualização de tela
  - Buscar por dados
  - Verificação ortográfica
  - Responder a uma requisição de rede
  - ...
- A criação de um novo processo é pesada enquanto a criação de um thread é leve
- Pode aumentar o desempenho do código e, em alguns casos, até simplificá-lo
- Kernels são, normalmente, multithreaded

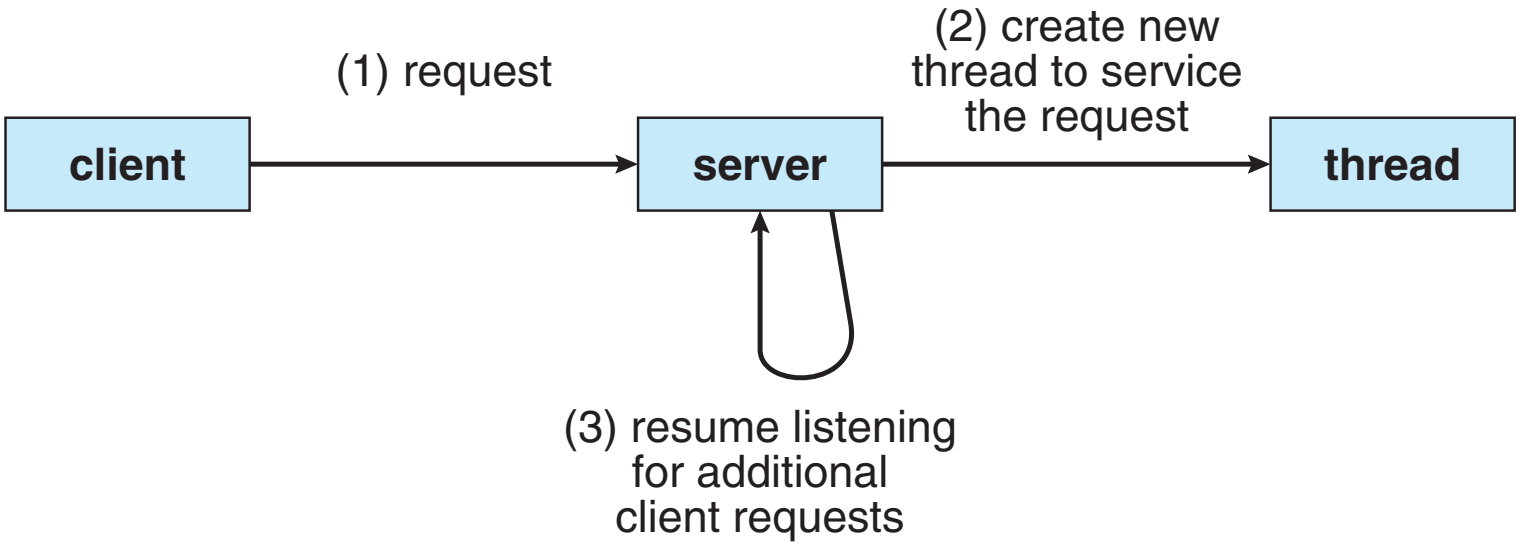
Processos *single* e *multithreaded*



single-threaded process

multithreaded process

Arquitetura de um servidor *multithreaded*



Benefícios

UFABC

- Responsividade (*responsiveness*) - Permite que o processo prossiga sua execução ainda que parte dele esteja bloqueada esperando por alguma resposta. Especialmente importante no caso de interfaces com o usuário
- Compartilhamento de recursos (*resource sharing*) - Threads compartilham recursos do processo. Há quem diga que é mais simples e fácil do que memória compartilhada e troca de mensagens
- Eficiência/Economia - É mais barato (em tempo e recursos) que criar um processo, e tem menos overhead para fazer trocas de contexto
- Escalabilidade (*scalability*) - Permite ao processo fazer um uso eficiente de processadores multicore

Programação multicore

UFABC

- Sistemas com múltiplas CPUs (*multi-CPU systems*) - Múltiplas CPUs em um mesmo sistema para prover um maior desempenho
- Sistemas multicore (*multicore systems*) - Múltiplos núcleos de processamento (*processing cores*) em um mesmo chip e cada um dos cores aparece como uma CPU independente para o sistema operacional
- A programação multithreaded oferece um modelo e mecanismo para utilizarmos estas arquiteturas computacionais de maneira eficiente e concorrente

- Considere uma aplicação com 4 threads
  - Em um sistema com um único núcleo de processamento, concorrência significa que **a execução dos threads vai ser intercalada ao longo do tempo**
  - Em um sistema com múltiplos núcleos, concorrência significa que **alguns threads vão executar em paralelo** pois o sistema poderá atribuir um núcleo diferente para cada thread

Concorrência *vs.* paralelismo

UFABC

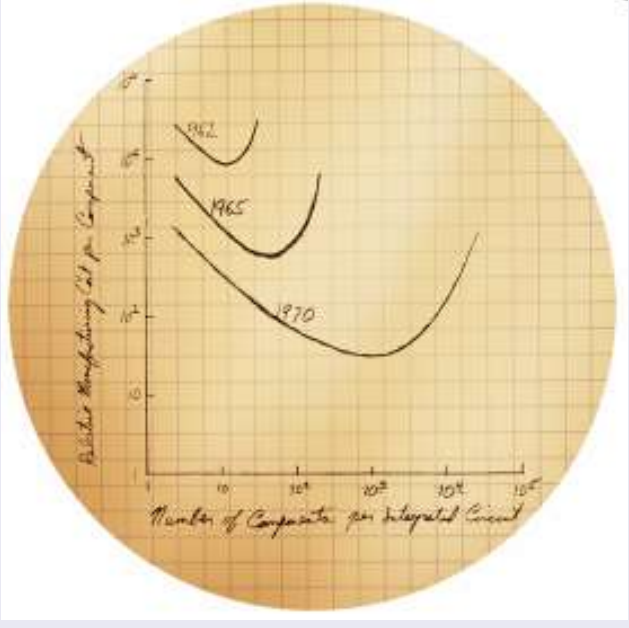
- Há uma distinção bem tênue entre o que é concorrente e o que é paralelo
- Um sistema concorrente permite que mais de uma tarefa progrida em sua execução ao longo do tempo
  - Uma tarefa não precisa esperar a outra finalizar para avançar
- Um sistema paralelo permite que mais de uma tarefa avance na sua execução ao mesmo tempo
- Assim, é possível que um sistema seja concorrente mas não paralelo! (Por que?)

UFABC

A Lei de Moore

O número de transistores em um chip vai dobrar aproximadamente a cada 18 meses.

Gordon E. Moore, 1965



É o fim da lei de Moore?

Andy Giveth, and Bill Taketh away

- Não importa o quão rápido os processadores se tornem, o software sempre encontra novas maneiras para consumir o ganho de desempenho.

The Free Lunch

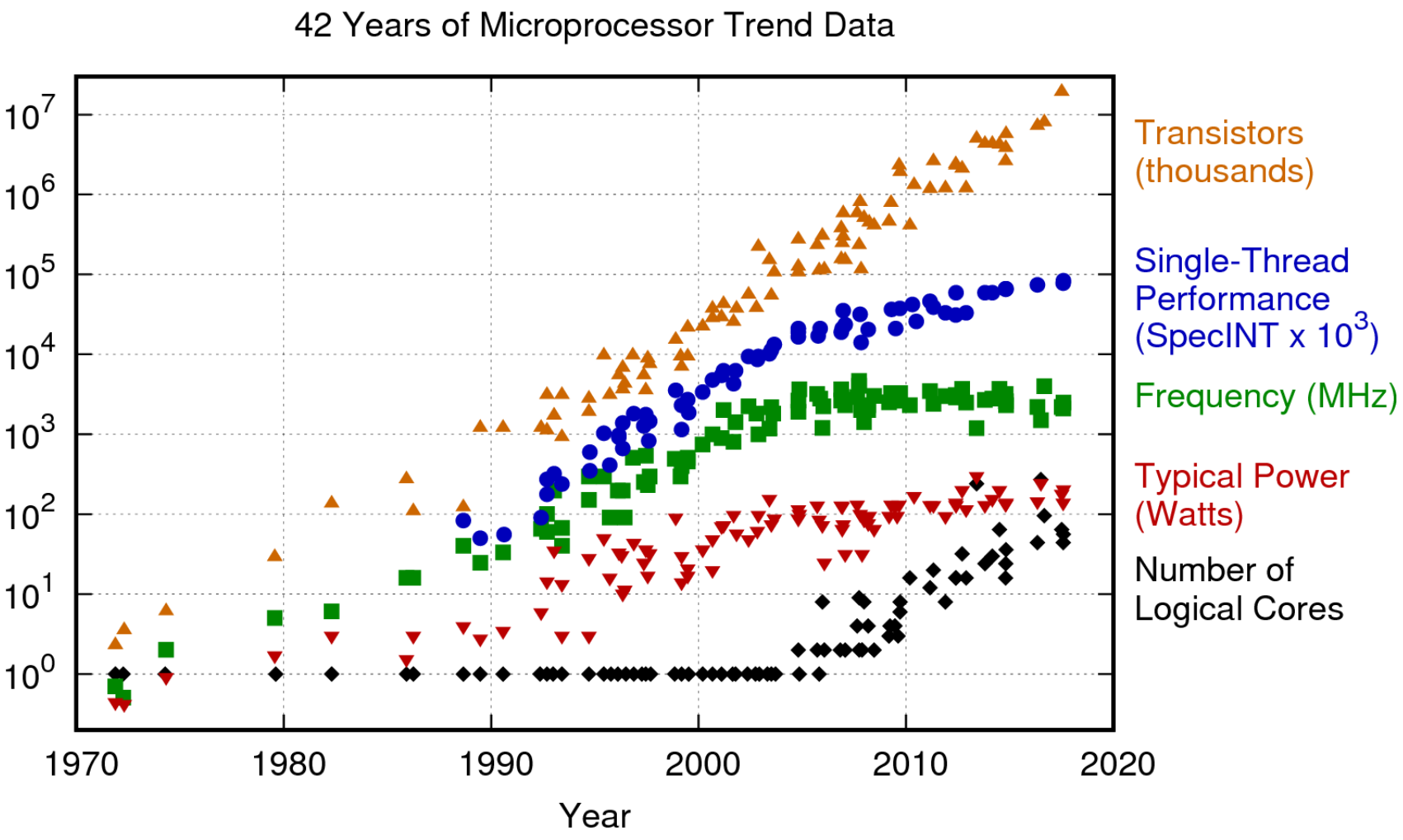
Ganhos de desempenho regulares, baratos e ”gratuitos”sem nem mesmo ter a necessidade de lançar novas versões.

Bastava esperar...

- Este tempo já passou.
- Ótimo artigo do Herb Sutter:

`http://www.gotw.ca/publications/concurrency-ddj.htm`

The free lunch is over



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

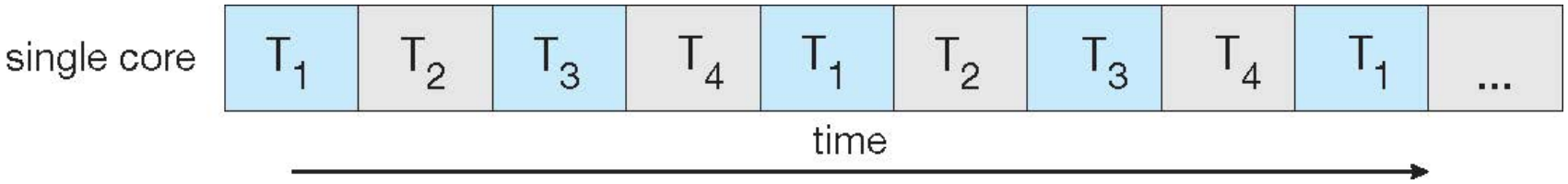
The free lunch is over

UFABC

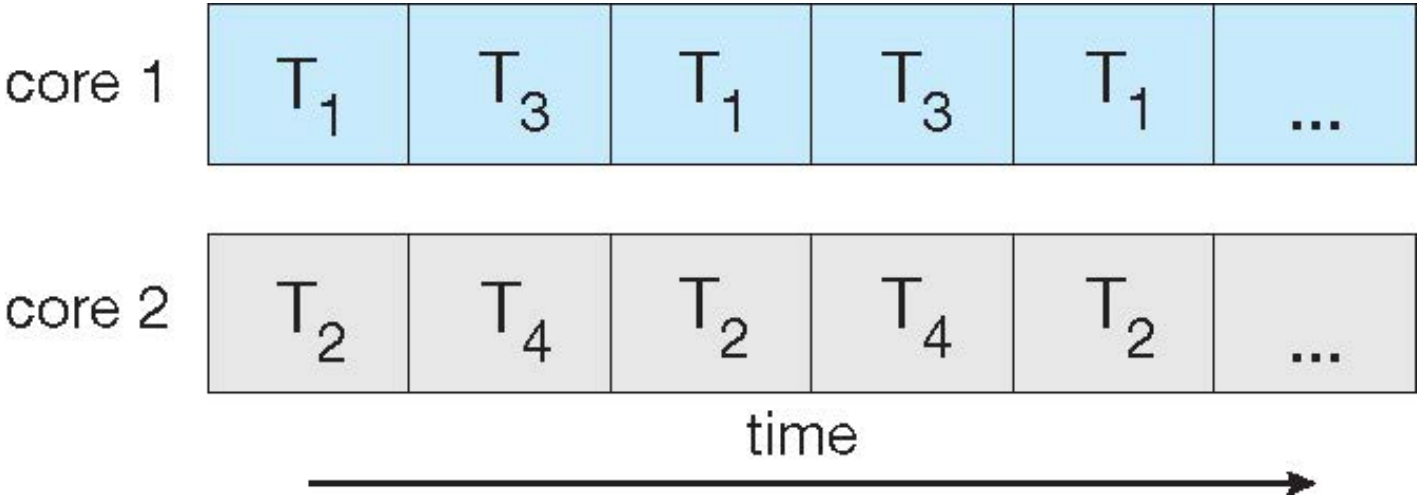
- Sistemas multicore e multiprocessadores estão colocando pressão nos desenvolvedores de software
  - Dividir as atividades
  - Equilibrar a carga
  - Dividir os dados
  - Tratar as dependências de dados
  - Testes e depuração
- **Paralelismo** - implica a execução de mais de uma tarefa ao mesmo tempo
- **Concorrência** - permite que mais de uma tarefa avance ao longo do tempo
  - Em um processador com um único núcleo - A concorrência é fornecida pelo escalonador

UFABC

- Execução concorrente em um sistema com um núcleo



- Paralelismo em um sistema multicore



Lei de Amdahl

UFABC

- Identifica ganhos de desempenho pela adição de núcleos de processamento para a execução de uma aplicação com componentes sequenciais e paralelos
- Considere N núcleos de processamento e S a proporção sequencial do código. Então:

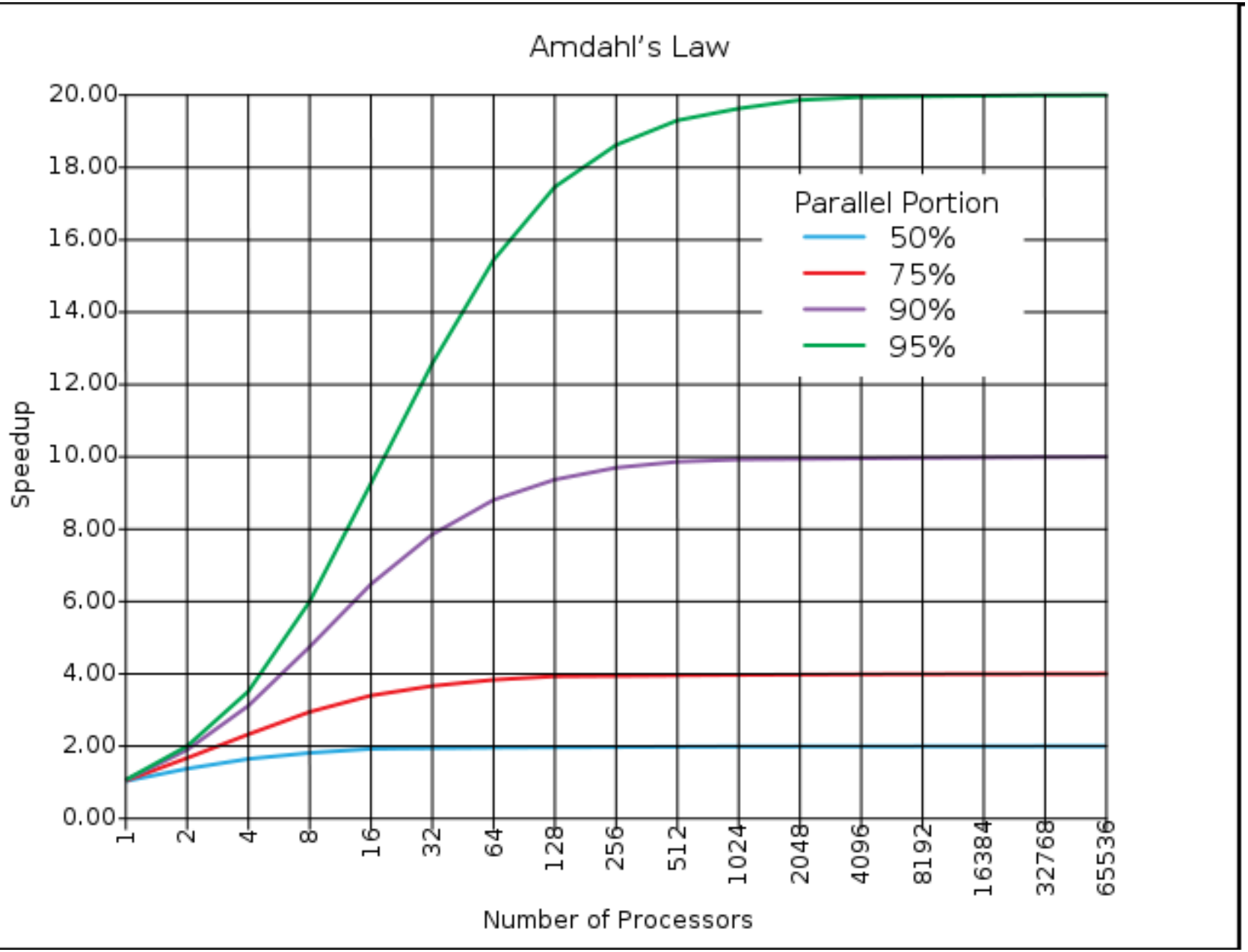
$$\text{speedup} \leq \frac{1}{S + \frac{1-S}{N}}$$

Lei de Amdahl

- Em outras palavras, se uma aplicação é 75% paralela e 25% sequencial então mudar de 1 para dois núcleos resulta em um speedup de 1.6×
- Conforme N tende a infinito, o speedup tende a $\frac{1}{S}$
- A parte sequencial tem um efeito fortíssimo no desempenho máximo que pode ser obtido pela adição de novos núcleos
- Essa lei leva em conta os sistemas multicores atuais?

Lei de Amdahl

UFABC



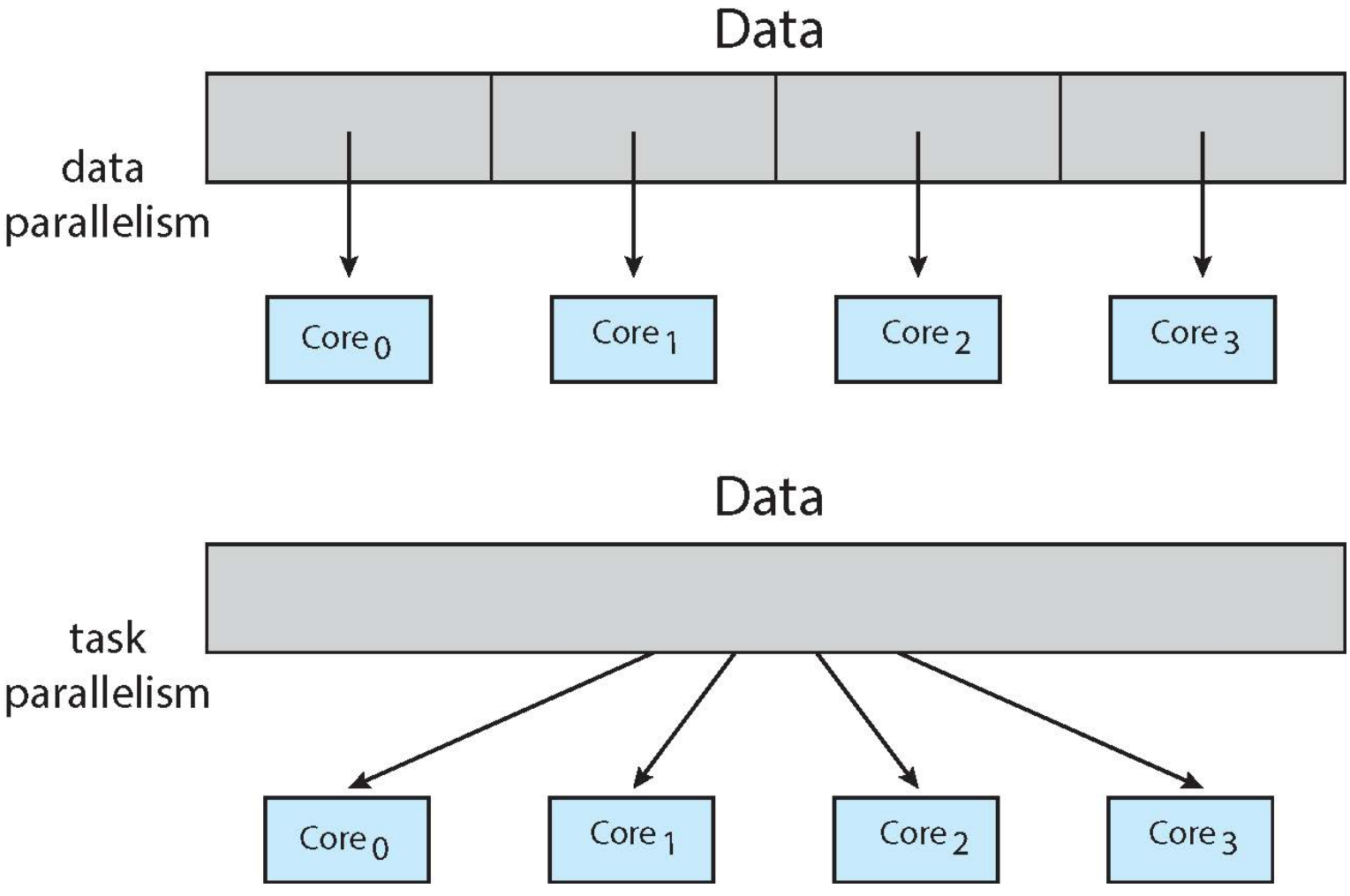
Tipos de paralelismo

UFABC

- Comumente dividimos o paralelismo em duas categorias
  - Paralelismo de dados (*data parallelism*) - distribui subconjuntos dos dados a serem processados entre múltiplos núcleos que, por sua vez, efetuam a mesma operação
  - Paralelismo de tarefas (*task parallelism*) - distribui threads enter os núcleos onde cada um deles efetua uma operação distinta. Threads distintos podem estar ou não operando sobre o mesmo conjunto de dados
- Conforme o número de threads cresce, o suporte de hardware também acompanha
  - É comum encontrar processadores convencionais com 32+ threads
  - Alguns processadores como o MPPA-256 tem 256 cores
  - GPUs modernas alcançam facilmente 2048+ núcleos

Tipos de paralelismo

UFABC

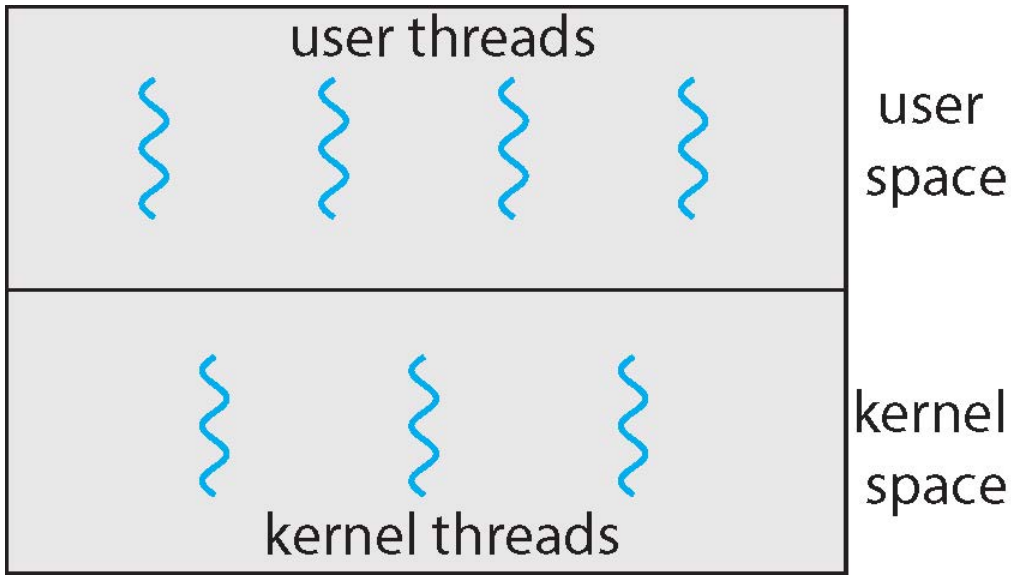


Modelos de multithreading

Threads de usuário ou do kernel

UFABC

- O suporte a threads pode estar disponível em dois níveis
  - Threads de usuário (*user threads*) - São oferecidos no nível do usuário e são administrados sem interferência do kernel, normalmente através de bibliotecas
  - Threads do kernel (*kernel threads*) - São oferecidos e administrados diretamente pelo sistema operacional
- Praticamente todos os sistemas operacionais modernos (Windows, Linux, Mac OS, ...) oferecem threads do kernel



Relacionamento entre threads dr usuário e do kernel

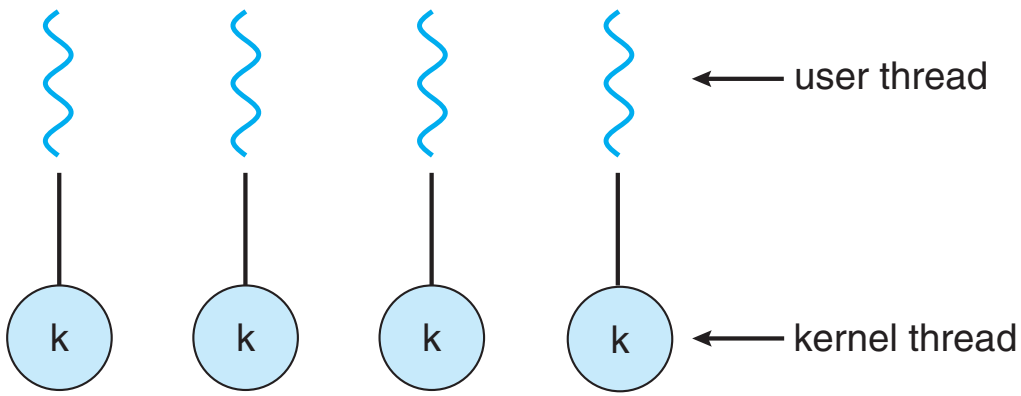
UFABC

- Um para um
- Muitos para um
- Um para muitos

Modelo um para um

UFABC

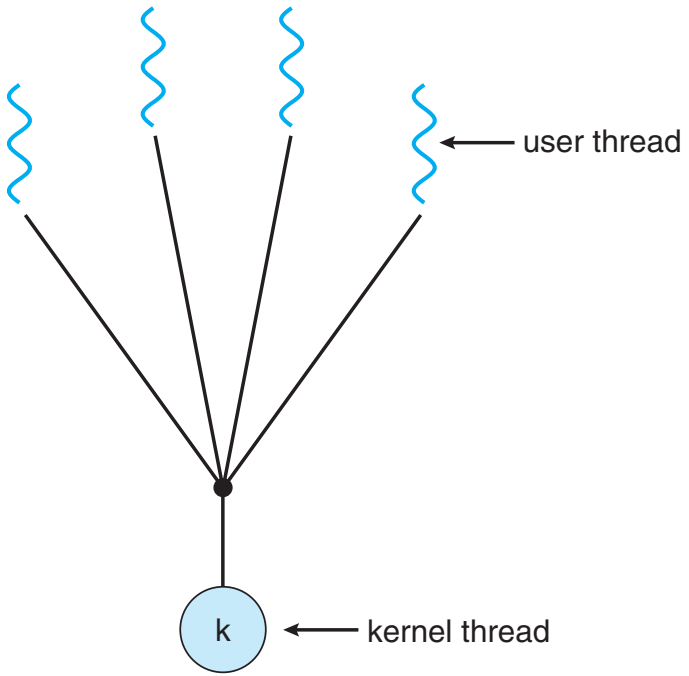
- Cada thread no nível do usuário tem um correspondente no nível do kernel
- Criar um thread no nível do usuário é o mesmo que criar um nível do kernel
- Mais concorrência do que ”muitos para um”
- Número de threads por processo pode ser limitado devido à sobrecarga
- Exemplos: Linux, Windows



Modelo muitos para um

UFABC

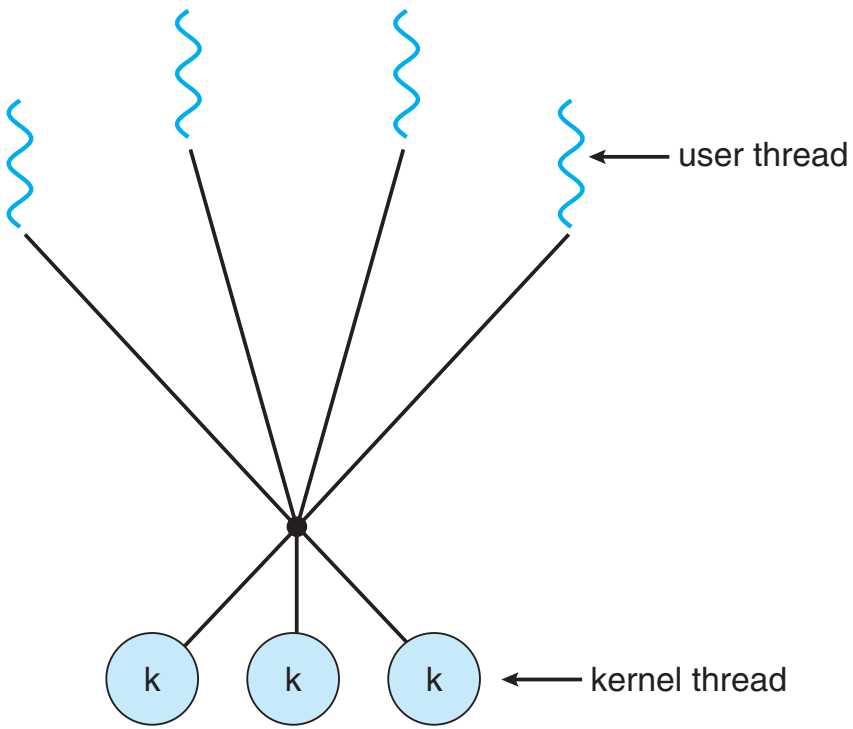
- Muitos threads de usuário são mapeados para um único thread do kernel
- Se um thread bloquear, todos ficam bloqueados
- Múltiplos threads podem deixar de rodar em paralelo em um sistema multicore pois há apenas um thread do kernel
- Poucos sistemas usam este sistema atualmente
  - Solaris Green Threads, GNU Portable Threads



Modelo muitos para muitos

UFABC

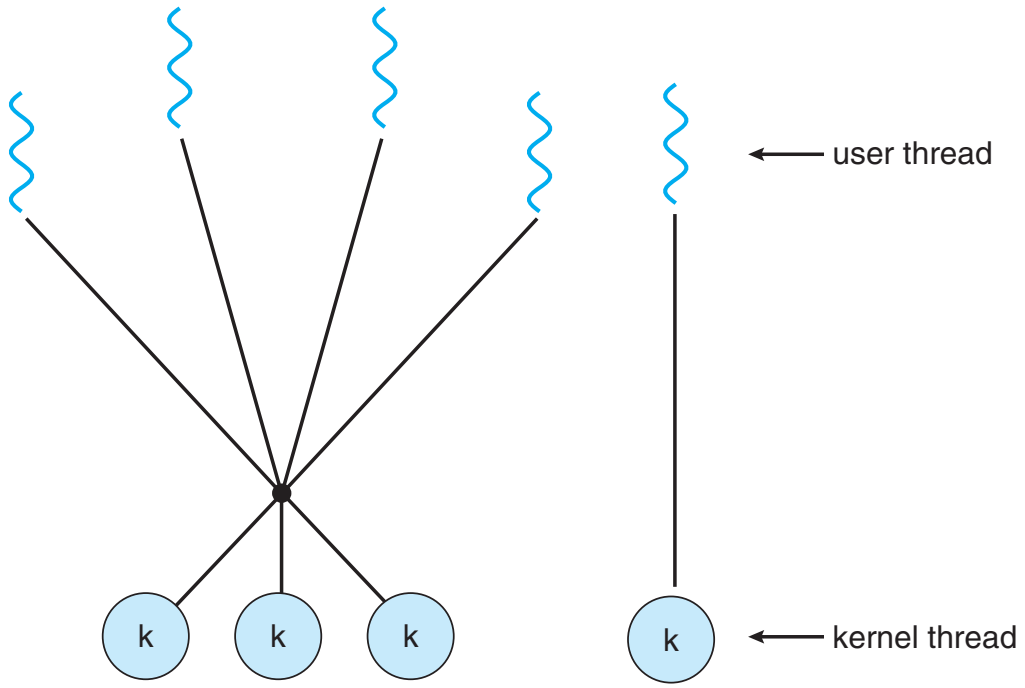
- Permite que diversos threads de usuário sejam mapeados para diversos threads do kernel
- Permite que o SO crie um número suficiente de threads
- Solaris antes da versão 9
- Windows com o pacote ThreadFiber



Modelo de dois níveis

UFABC

- Simular ao muitos para muitos, contudo permite ao usuário vincular (*bind*) um thread de usuário a um thread do kernel
- Exemplos: IRIX, HP-UX, Tru64 UNIX, Solaris ≤ 8



Bibliotecas de threads

Bibliotecas de threads

UFABC

- Uma biblioteca de threads (*thread library*) fornece ao programador uma API para criar e gerenciar threads
- Duas maneiras principais para implementar
  - Biblioteca contida completamente no espaço do usuário
  - Biblioteca implementada no nível do kernel fornecida pelo SO
- São 3 as mais famosas
  - POSIX Pthreads
  - Windows threads
  - Java threads

Pthreads

- Pode ser implementado tanto como no nível do usuário quanto no nível do kernel
- Padrão POSIX (IEEE 1003.1c) - API para criação, gerenciamento e sincronização de threads
- É uma especificação e não uma implementação
- A API especifica o comportamento da biblioteca de threads, a implementação é por conta do desenvolvedor da biblioteca
- Presente em sistemas estilo UNIX: Solaris, Linux, Mac OS X

Pthreads: Exemplo

UFABC

- Nos próximos slides apresentamos um código multithreaded em C que calcula a soma de um inteiro não negativo em um thread separado
- Em um programa Pthread, os novos threads começam a sua execução chamando uma função. No exemplo é a função `runner()`
- Quando o programa começa a execução, há apenas um thread em execução o thread principal (*main thread*).
- Depois da inicialização a função `main()` cria um novo thread que começa a executar na função `runner()`
- Ambos os threads **compartilham** a variável global `sum`

Pthreads: Exemplo

UFABC

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param);/*threads call this function*/

int main(int argc, char *argv[]) {
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */
    if (argc != 2) {
        fprintf(stderr,"usage: a.out <int>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%s must be >=0\n", argv[1]);
        return -1;
    }
    ...
```

UFABC

```
1       /* get the default attributes */
2       pthread_attr_init(&attr);
3       /* create the thread */
4       pthread_create(&tid,&attr,runner,argv[1]);
5       /* wait for the thread to exit */
6       pthread_join(tid,NULL);
7       printf("sum = %d\n",sum);
8   }
9   /* The thread will begin control in this function */
10  void *runner(void *param) {
11      int i, upper = atoi(param);
12      sum = 0;
13      for (i = 1; i <= upper; i++)
14          sum += i;
15      pthread_exit(0);
16  }
```

Pthreads - Joining

UFABC

- O código da soma cria apenas um thread
- Em arquiteturas multicore é comum criar bem mais threads
- Exemplo para fazer o *join* de mais threads

```
1  #define NUM THREADS 10
2  /* an array of threads to be joined upon */
3  pthread_t workers[NUM THREADS];
4  for (int i = 0; i < NUM THREADS; i++)
5      pthread_join(workers[i], NULL);
```

Windows - Exemplo multithreaded

UFABC

```
1  #include < windows.h>
2  #include < stdio.h>
3  DWORD Sum; /* data is shared by the thread(s) */
4
5  /* the thread runs in this separate function */
6  DWORD WINAPI Summation(LPVOID Param) {
7      DWORD Upper = *(DWORD*)Param;
8      for (DWORD i = 0; i <= Upper; i++)
9          Sum += i;
10     return 0;
11 }
12 ...
```

UFABC

```
1  int main(int argc, char *argv[]) {
2      DWORD ThreadId;
3      HANDLE ThreadHandle;
4      int Param;
5      if (argc != 2) {
6          fprintf(stderr,"An integer parameter is
           ↪  required\ n");
7          return -1;
8      }
9      Param = atoi(argv[1]);
10     if (Param < 0) {
11         fprintf(stderr,"An integer >= 0 is required\
           ↪  n");
12         return -1;
13     }
14     ...
```

Windows - Exemplo multithreaded

UFABC

```
1       ...
2       /* create the thread */
3       ThreadHandle = CreateThread(
4                   NULL, /* default security attributes */
5                   0, /* default stack size */
6                   Summation, /* thread function */
7                   &Param, /* parameter to thread function */
8                   0, /* default creation flags */
9                   &ThreadId); /* returns the thread id */
10      if (ThreadHandle != NULL) {
11          /* now wait for the thread to finish */
12          WaitForSingleObject(ThreadHandle,INFINITE);
13          /* close the thread handle */
14          CloseHandle(ThreadHandle);
15          printf("sum = %d\ n",Sum);
16      }
17  }
```

UFABC

- Threads do Java são gerenciados pela JVM
- Tipicamente são implementados usando o sistema de threads fornecido pelo SO
- Há duas maneiras principais de se criar um thread em Java
  - Criar uma classe derivada da classe `Thread` e sobrescrever o método `run()`
  - Implementar uma classe que implementa a interface `Runnable`

```
1 public interface Runnable {
2     public abstract void run();
3 }
```

- Quando uma classe implementa `Runnable` ela é obrigada a implementar o método `run()` que por sua vez é o método executado pelo thread criado

Um programa Java multithreaded

UFABC

- Os próximos slides mostram um programa multithreaded em Java
- A classe `Summation` implementa a interface `Runnable`
- A criação de threads é feita através da criação de um objeto da classe thread e passando como parâmetro para o construtor um objeto do tipo `Runnable`

Um programa Java multithreaded

UFABC

```
1  class Sum {
2      private int sum;
3      public int getSum() {
4          return sum;
5      }
6      public void setSum(int sum) {
7          this.sum = sum;
8      }
9  }
```

Um programa Java multithreaded

UFABC

```
1  class Summation implements Runnable {
2      private int upper;
3      private Sum sumValue;
4      public Summation(int upper, Sum sumValue) {
5          this.upper = upper;
6          this.sumValue = sumValue;
7      }
8      public void run() {
9          int sum = 0;
10         for (int i = 0; i <= upper; i++)
11             sum += i;
12         sumValue.setSum(sum);
13     }
14 }
```

Um programa Java multithreaded

UFABC

```
1  public class Driver {
2    public static void main(String[] args) {
3      if (args.length > 0) {
4        if (Integer.parseInt(args[0]) < 0)
5          System.err.println(args[0] + " must be >= 0.");
6        else {
7        ...
```

Um programa Java multithreaded

UFABC

```
1           ...
2           Sum sumObject = new Sum();
3           int upper = Integer.parseInt(args[0]);
4           Thread thrd = new Thread(new Summation(upper,
            ↪  sumObject));
5           thrd.start();
6           try {
7             thrd.join();
8             System.out.println
9                 ("The sum of "+upper+" is
                  ↪  "+sumObject.getSum());
10          } catch (InterruptedException ie) { }
11        }
12      } else
13        System.err.println("Usage: Summation <integer
          ↪  value>"); }
14  }
```

Threading implícito

Threading implícito

UFABC

- Tem ficado cada vez mais na moda pois conforme o número de threads cresce, mais difícil fica fazer o seu gerenciamento
- Nesta modalidade a criação e o gerenciamento de threads são feitos pelos compiladores e por bibliotecas de execução e não pelos programadores
- Alguns métodos bem conhecidos
  - Thread Pools
  - Fork-Join
  - OpenMP
  - Intel Thread Building Blocks
  - Grand Central Dispatch
  - Microsoft Thread Building Blocks
  - Pacote `java.util.concurrent` do Java

Thread Pools

UFABC

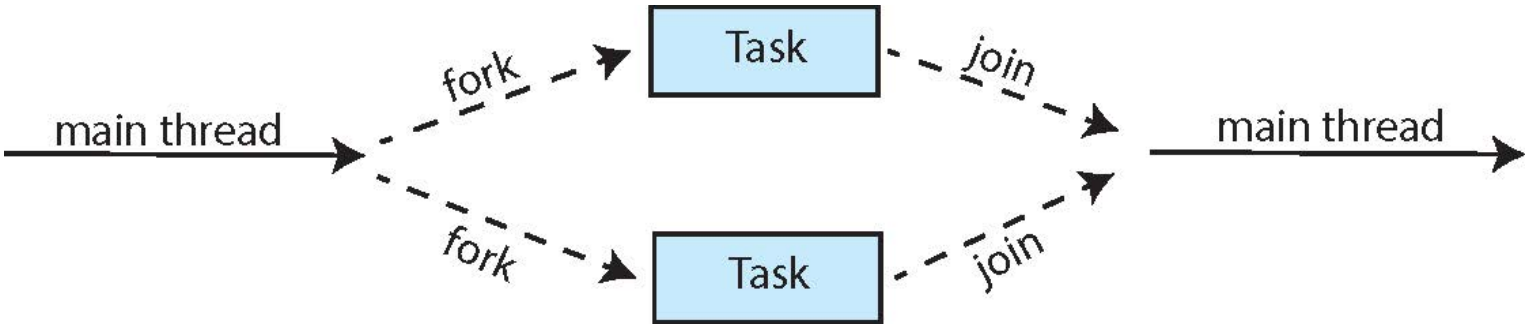
- Cria um número de threads e os mantém em um *pool* onde eles ficam esperando para ser executados
- Vantagens
  - É normalmente um pouco mais rápido usar um thread já criado para atender uma requisição do que criar um novo thread
  - Permite que o número de threads de uma aplicação seja limitado pelo tamanho do pool
  - Separar a tarefa a ser executada da maneira que ela é criada permite o uso de diferentes estratégias a depender da tarefa
    - Por exemplo, tarefas podem ser periódicas
  - A API do Windows tem suporte a thread pools

```
1 DWORD WINAPI PoolFunction(AVOID Param) {
2    /* this function runs as a separate thread */
3 }
```

Criando uma thread pool em Java

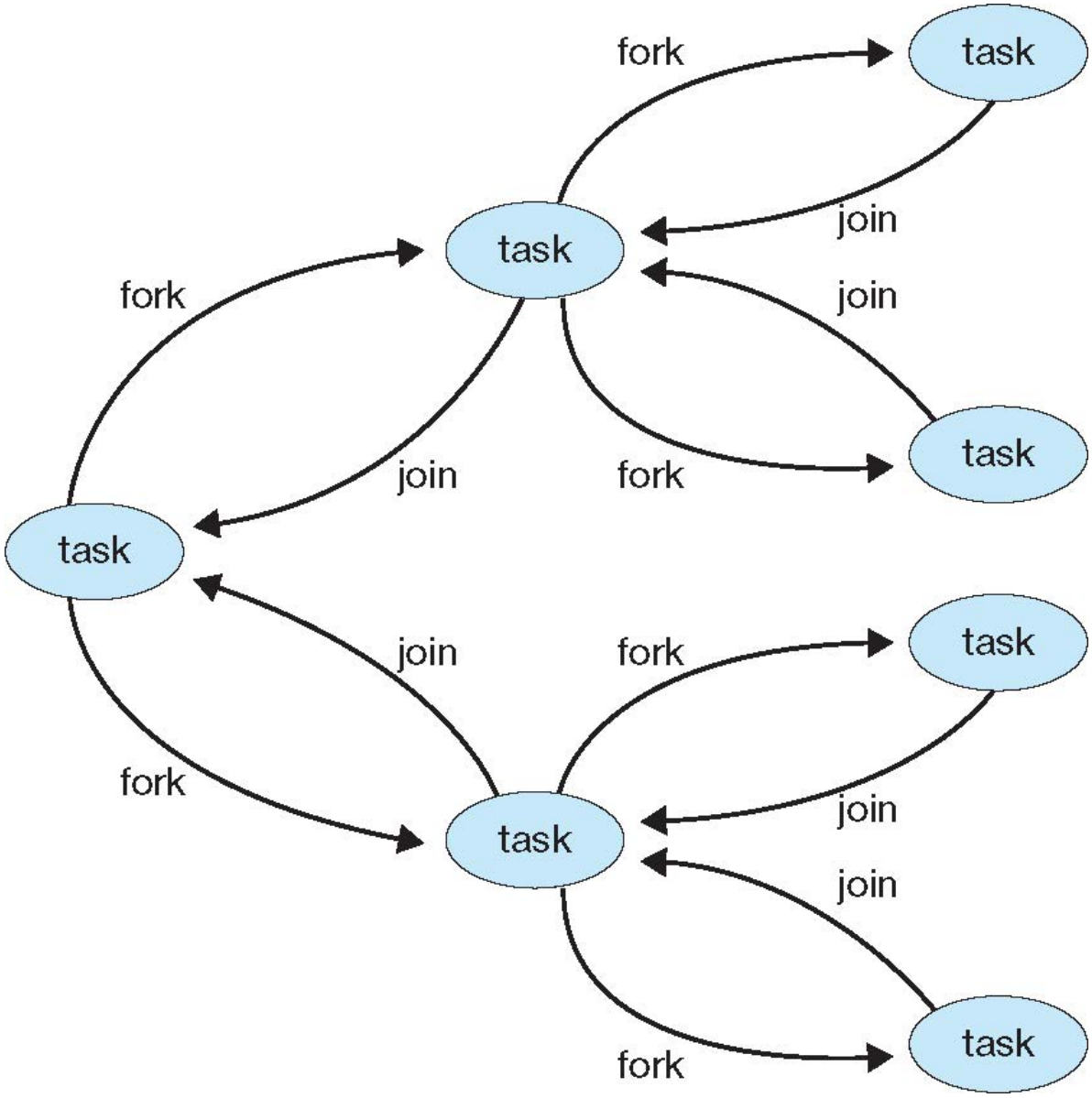
UFABC

```
1  import java.util.concurrent.*;
2
3  public class ThreadPoolExample {
4    public static void main(String[] args) {
5      int numTasks = Integer.parseInt(args[0].trim());
6      /* Create the thread pool */
7      ExecutorService pool
       ↪  =Executors.newCachedThreadPool();
8      /* Run each task using a thread in the pool */
9      for (int i = 0; i < numTasks; i++)
10       pool.execute (new Task());
11     /* Shut down the pool once all threads have
       ↪  completed */
12     pool.shutdown();
13   }
14 }
```


main thread
fork
Task
join
main thread
fork
Task
join

Fork-Join em Java

UFABC



Fork-Join: Exemplo em Java

UFABC

```
1  import java.util.concurrent.*;
2  public class SumTask extends RecursiveTask<Integer> {
3      static final int THRESHOLD = 1000;
4      private int begin;
5      private int end;
6      private int[] array;
7      public SumTask(int begin, int end, int[] array) {
8          this.begin = begin;
9          this.end = end;
10         this.array = array;
11     }
```

Fork-Join: Exemplo em Java

UFABC

```
1   protected Integer  compute() {
2     if (end - begin < THRESHOLD) {
3       int sum = 0;
4       for (int i = begin; i <= end; i++)
5         sum += array[i];
6       return sum;
7     } else {
8       int mid = begin + (end - begin) / 2;
9       int mid = (begin + end) / 2;
10      SumTask leftTask = new SumTask(begin, mid, array);
11      SumTask rightTask = new SumTask(mid + 1, end,
        ↪  array);
12      leftTask.fork();
13      rightTask.fork();
14      return rightTask.join() + leftTask.join();
15    }
16  }
17  }
```

UFABC

- Conjunto de diretivas para o compilador
- Disponível em C, C++ e FORTRAN
- Prove suporte a programação paralela em ambientes com memória compartilhada
- O programador demarca regiões paralelas de código

```c
#include < omp.h>
#include < stdio.h>
int main(int argc, char *argv[]) {
    /* sequential code */
#pragma omp parallel
    {
        /*Cria tantos threads quanto há cores
          no sistema*/
        printf("I am a parallel region.");
    }
    /* sequential code */
    return 0;
}
```

UFABC

```
1  /* Divide o for entre vários threads
2     e executa em paralelo */
3  #pragma omp parallel for
4  for (i = 0; i < N; i++) {
5      c[i] = a[i] + b[i];
6  }
```

Grand Central Dispatch

UFABC

- Tecnologia da Apple utilizada no Mac OS X e no iOS
- Extensões às linguagens C e C++, API e biblioteca de execução
- Permite a demarcação de regiões paralelas
- Controla a maior parte dos detalhes relativos aos threads
- O delimitador do bloco é `^{}`:
  - `^{printf("Sou um bloco\n");}`
- Blocos são colocados na fila de despacho
  - Blocos são escalonados em um pool de threads

UFABC

- Dois tipos de filas de despacho para os blocos
  - Sequencial: blocos são retirados da fila em ordem de chegada (FIFO), cada processo tem uma fila chamada de `main queue`
    - Programadores têm a liberdade de criar filas sequencias adicionais em seus programas
  - Concorrente: blocos são retirados da fila em ordem de chegada (FIFO) contudo **podem ser executados concorrentemente**
    - Há três filas globais do sistema com prioridades baixa, média e alta:

```
1 dispatch_queue_t queue = dispatch_get_global_queue(
2     DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
3 dispatch_async(queue, ^{ printf("I am a block."); });
```

Problemas com threads

Problemas com threads

UFABC

- Semântica das syscalls `fork()` e do `exec()`
- Tratamento de sinais
  - Síncrono e assíncrono
- Cancelamento de threads
  - Assíncrono ou diferido
- *Thread-local storage*
- Ativações do escalonador (veja [SGG 4.6.5])

Semântica das syscalls `fork()` e do `exec()`

UFABC

- O `fork()` duplica apenas o thread que chamou ou todas os threads?
  - Alguns sistemas UNIX tem duas versões de `fork`
- O `exec()` funciona normalmente: substitui o processo em execução incluindo todas as threads

Tratamento de sinais

UFABC

- Sinais são usados nos sistemas UNIX para notificar processos que um evento ocorreu
- Um tratador de sinais (*signal handler*) é usado para processar os sinais
  1. Sinal é gerado por um evento em particular
  2. Sinal é entregue ao processo
  3. Sinal é tratado por um dos seguintes tratadores
     1. Padrão
     2. Definido pelo usuário
- Todo sinal tem um tratador padrão (*default handler*)
  - Um tratador de sinais definido pelo usuário (*user-defined signal handler*) pode sobrescrever o padrão
  - Para processos com um único thread, o sinal é entregue para o processo (=thread principal)

Tratamento de sinais

UFABC

- E em programas com diversos threads?
  - Entrega o sinal ao thread para o qual o sinal se aplica
  - Entrega o sinal para todos os threads
  - Entrega o sinal para alguns threads
  - Atribui-se um thread que tratará todos os sinais do processo

Cancelamento de threads

UFABC

- Finalizar a execução de um thread de maneira forçada
- O thread a ser finalizado é chamado de thread alvo (*target thread*)
- O cancelamento de um thread alvo pode ser feito de duas maneiras principais:
  - Cancelamento assíncrono (*asynchronous cancellation*) - Finaliza o thread alvo imediatamente
  - Cancelamento diferido (*deferred cancellation*) - Permite ao thread que verifique periodicamente se ele foi cancelado

Cancelamento de threads

UFABC

- A dificuldade de cancelar threads reside nas situações onde:
  - Recursos que estão associados a um thread cancelado
  - Um thread é cancelado enquanto está atualizando dados compartilhados com outros threads
- O SO tipicamente vai recuperar (*reclaim*) alguns dos recursos do sistema de um thread cancelado, mas não todos
- O cancelamento de um thread assincronamente não necessariamente libera um recurso global do sistema pois pode ser necessário para outros threads

- O cancelamento diferido não sofre desses problemas
  - Um thread indica que o outro deve ser cancelado
  - O cancelamento ocorre apenas quando o thread alvo tiver verificado a flag de cancelamento
  - O próprio thread alvo pode tomar as atitudes necessárias para a liberação de recursos e decidir um ponto onde seu cancelamento não causará problemas

Cancelamento em Pthread

UFABC

- O cancelamento de Pthread pode ser feito pela função `pthread_cancel()`
  - O identifador do thread é passado como parâmetro
- Código em Pthread para criar e cancelar um thread:

```
1  pthread t tid;
2  /* create the thread */
3  pthread_create(&tid, 0, worker, NULL);
4  ...
5  /* cancel the thread */
6  pthread_cancel(tid);
```

Cancelamento em Pthread

UFABC

- A chamada de `pthread_cancel()` indica apenas um pedido para que o thread seja cancelado
- O cancelamento só ocorrerá dependendo de como o thread alvo estiver configurado
- Pthread tem suporte a 3 modos de cancelamento. Cada modo é definido como um estado e um tipo. Um thread pode ajudar o seu estado e tipo através de uma API
- Se um thread estiver com o cancelamento desabilitado, os cancelamentos pendentes são armazenados até serem processados

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

UFABC

- O tipo padrão de cancelamento é o diferido
  - O cancelamento só ocorre quando o thread alcança um ponto de cancelamento (*cancellation point*)
  - Uma das maneiras para estabelecer um ponto de cancelamento é através da chamada da função `pthread_testcancel()`
  - Se um cancelamento estiver pendente, a função conhecida como tratador de limpeza (*cleanup handler*) é chamada. Essa função permite que quaisquer recursos alocados possam ser seguramente liberados pelo próprio thread antes de termine a sua execução
  - No Linux, o cancelamento de threads é tratado através de sinais

UFABC

- Thread-Local Storage (TLS) permite que cada thread tenha sua própria cópia dos dados
- Útil quando não se tem controle sobre a criação dos threads (por exemplo quando se utiliza um thread pool)
- Diferentes de variáveis locais
  - Variáveis locais só são visíveis enquanto a função que a declara estiver na pilha
  - TLS mantém-se visível mesmo entre diversas chamadas de função
- Semelhante a dados `static`
  - TLS é único para cada thread