Linguagem de Programação II

CARLOS EDUARDO BATISTA

CENTRO DE INFORMÁTICA - UFPB

BIDU@CI.UFPB.BR

Roteiro

- Introdução
- O modelo de Threads
- O uso de Threads
- Threads POSIX
- Threads Java
- Threads STD (C++11)

Introdução

- Em sistemas operacionais tradicionais, cada processo tem um único ponto de execução em um momento particular.
- Assim como um processo, uma *thread* tem um início, uma sequência e um fim.

Uma *thread* é um fluxo único de controle sequencial dentro de um programa.

Threads

- **Podemos enxergar um processo sob dois aspectos:**

Conjunto de recursos necessários para a execução de um programa.

Linha ou contexto de execução (Thread)

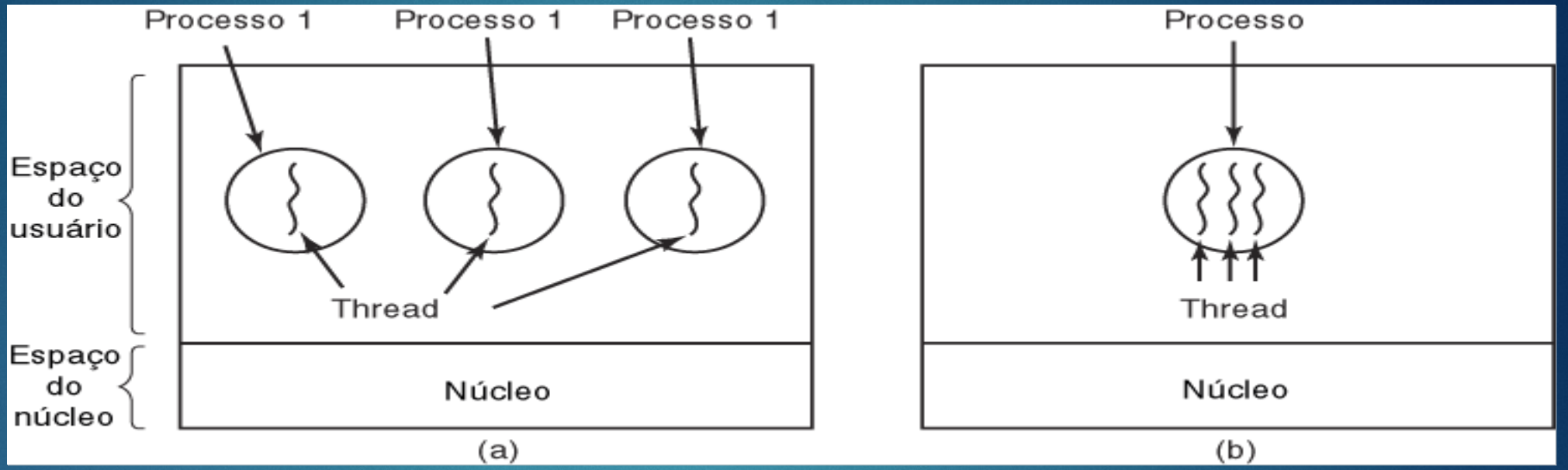
Threads

- **Conjunto de recursos necessários para execução de um programa:**
  - Espaço de endereçamento.
  - Tabela de descritores para arquivos abertos.
  - Informações sobre processos filhos.
  - Código para tratar sinais (signal handlers).

Threads

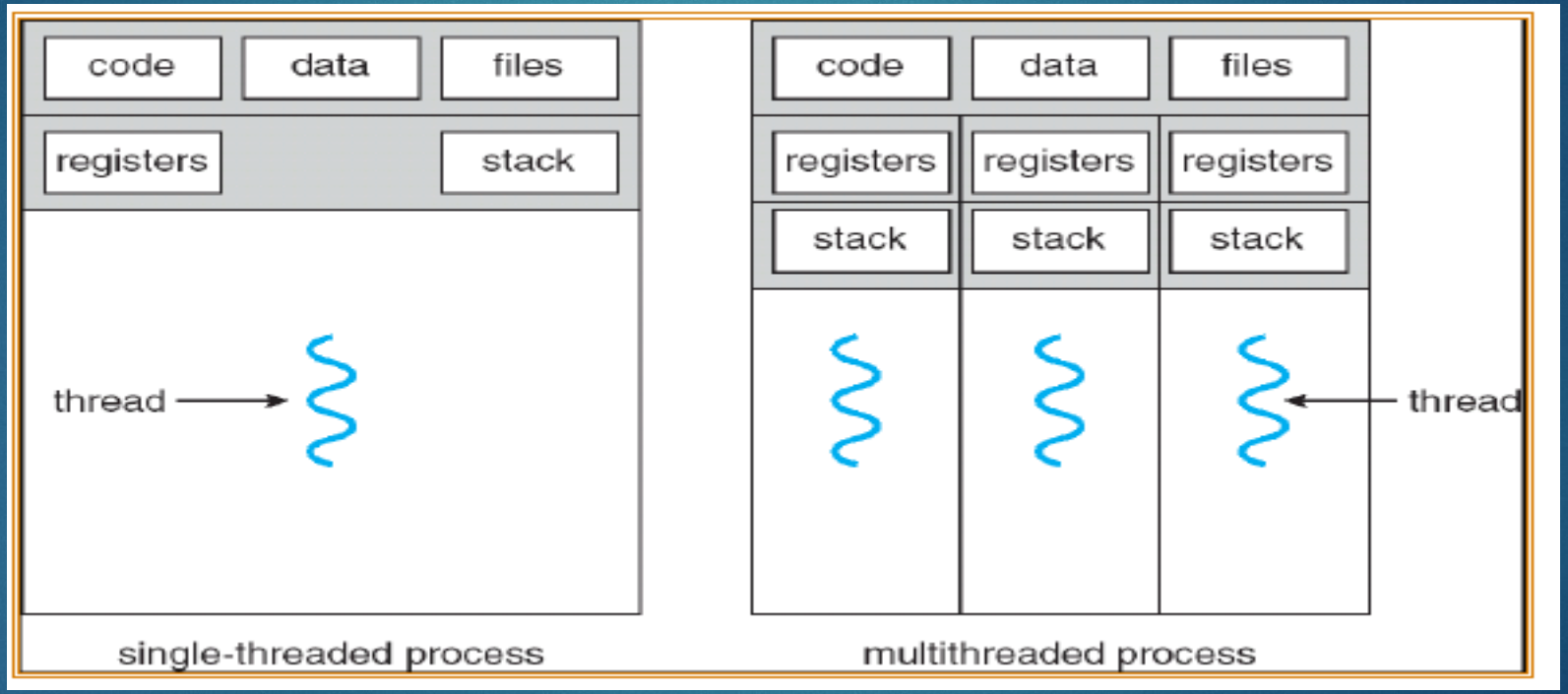
- **Linha ou contexto de execução (Thread)**
  - Contador de programa
  - Registradores
  - Pilha de execução
- Threads são entidades escalonadas para a execução sobre a CPU. Também chamados de **processos leves**.

O modelo de Threads



(a) Três processos cada um com uma *thread*.
(b) Um processo com três *threads.*

O modelo de Threads



O modelo de Threads

Itens por processo	Itens por thread
Espaço de endereçamento	Contador de programa
Variáveis globais	Registradores
Arquivos abertos	Pilha
Processos filhos	Estado
Alarmes pendentes	
Sinais e tratadores de sinais	
Informação de contabilidade	

- Na primeira coluna constam os itens compartilhados por todas as threads em um processo.
- A segunda é composta pelos itens privativos de cada thread.

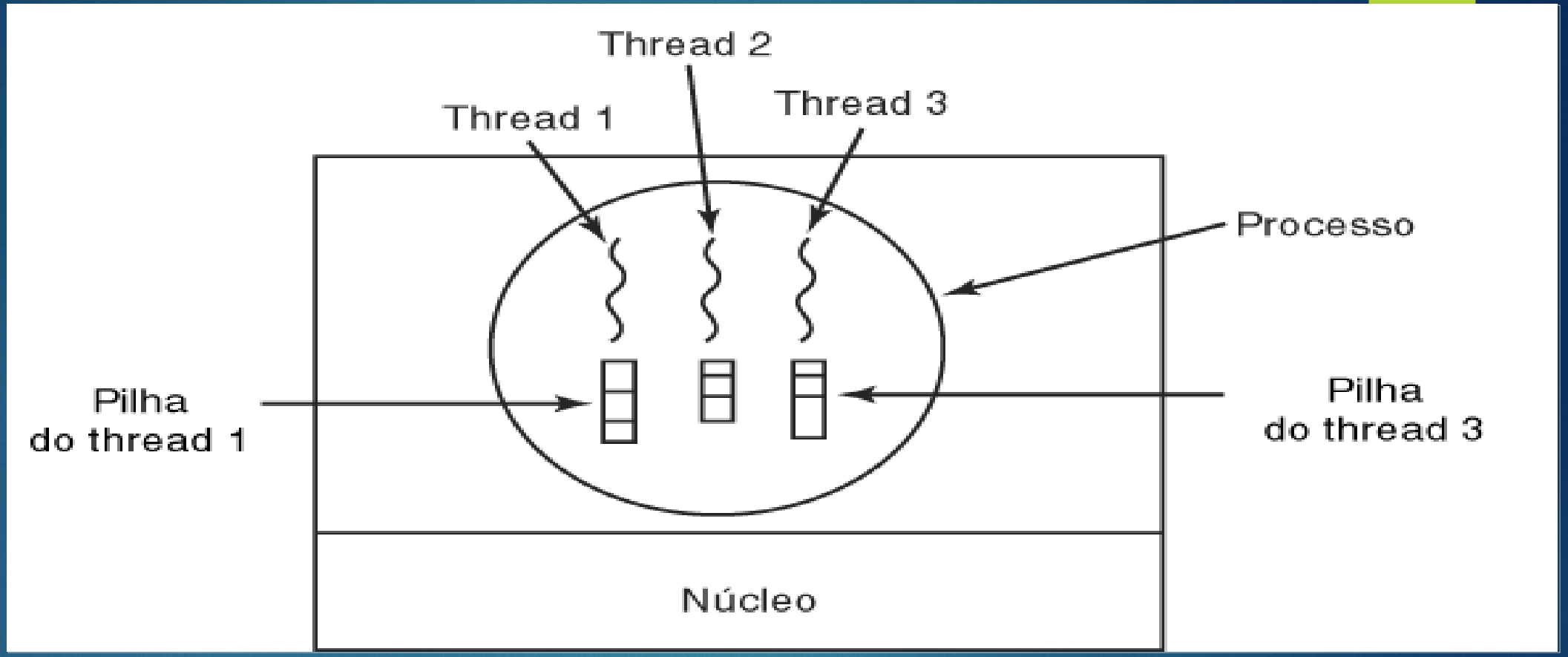
O modelo de threads

- Cada thread tem a sua pilha própria, mas compartilha o mesmo espaço de endereçamento do processo em que foi criada
- Se duas threads executam o mesmo procedimento/método, cada uma terá a sua própria cópia das variáveis locais
- As threads podem acessar todas os dados globais do programa, e o *heap* (memória alocada dinamicamente)
- Nesse acesso a dados globais (i.e. quando o acesso inclui mais do que uma instrução de máquina), as threads precisam ter acesso em regime de exclusão mútua (p.ex. usando locks)

O modelo de Threads

- Uma thread, assim como os processos, pode estar em um dos seguintes estados: em **execução**, **bloqueado**, **pronto** ou **finalizado**.
- É importante frisar que cada thread tem sua própria pilha, compartilhando, assim, apenas as variáveis globais.

O modelo de Threads



Pilhas das threads

O uso de Threads

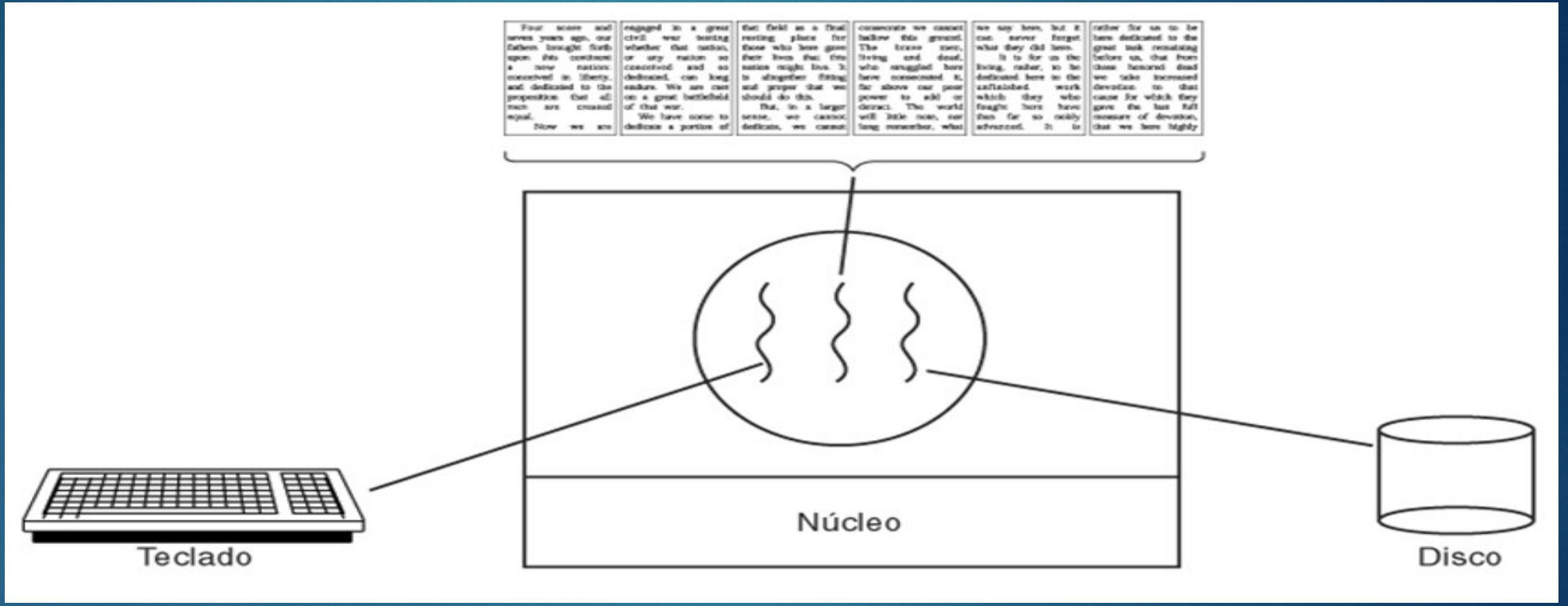
Quais as diferenças das threads em relação aos processos?

O uso de Threads

- **Threads**
  - Algumas aplicações necessitam de uma paralelização que ofereça compartilhamento de recursos.
  - Threads são criadas e destruídas mais rapidamente que processos, visto que não possuem recursos associados.
  - A implementação da cooperação entre threads é mais simples e seu desempenho é melhor.

O uso de Threads

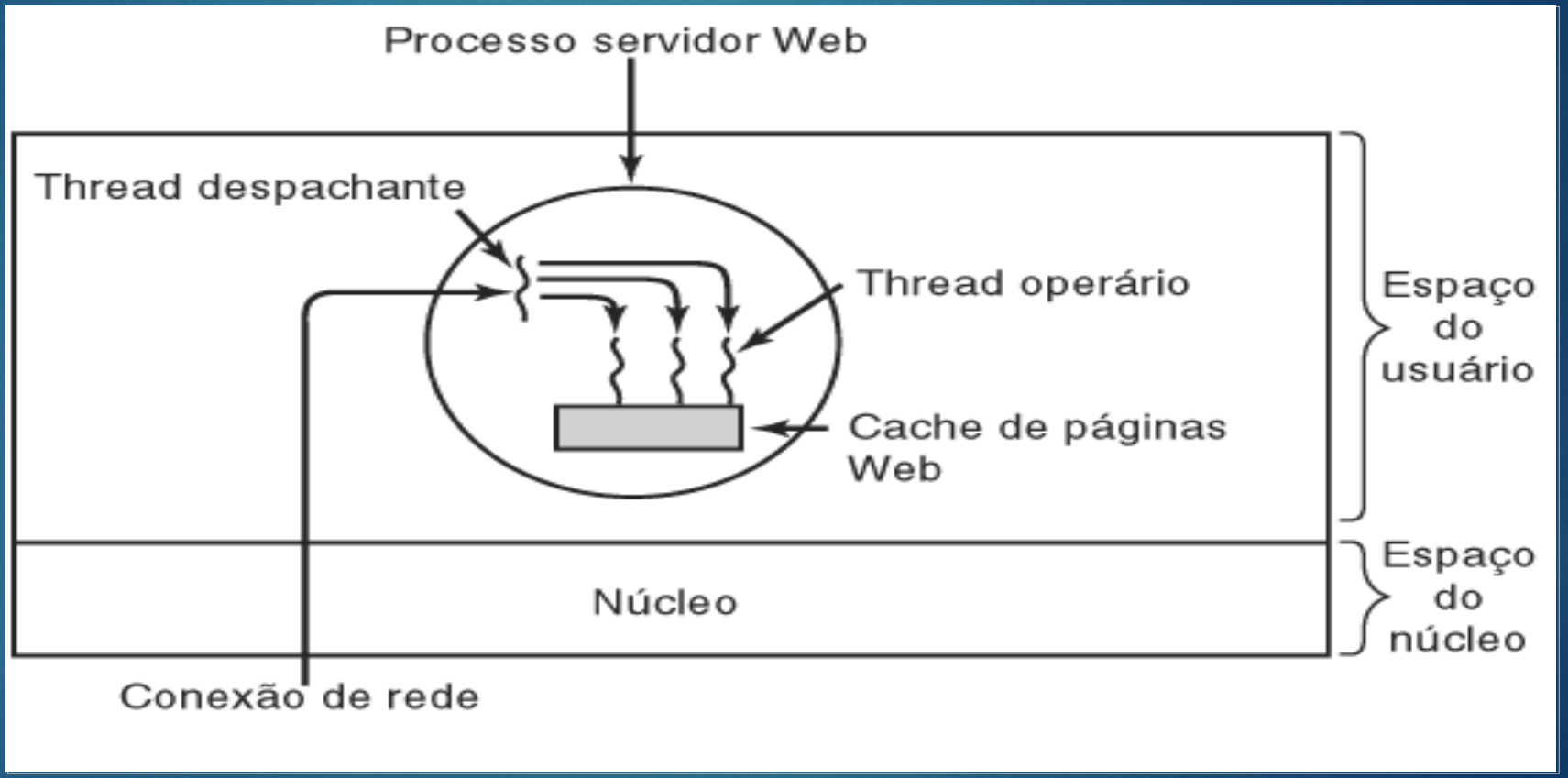
- **Threads**



Um processador de texto com três threads.

O uso de Threads

- **Threads**



Um servidor web com múltiplas threads.

O uso de Threads

- **Threads**

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if(page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

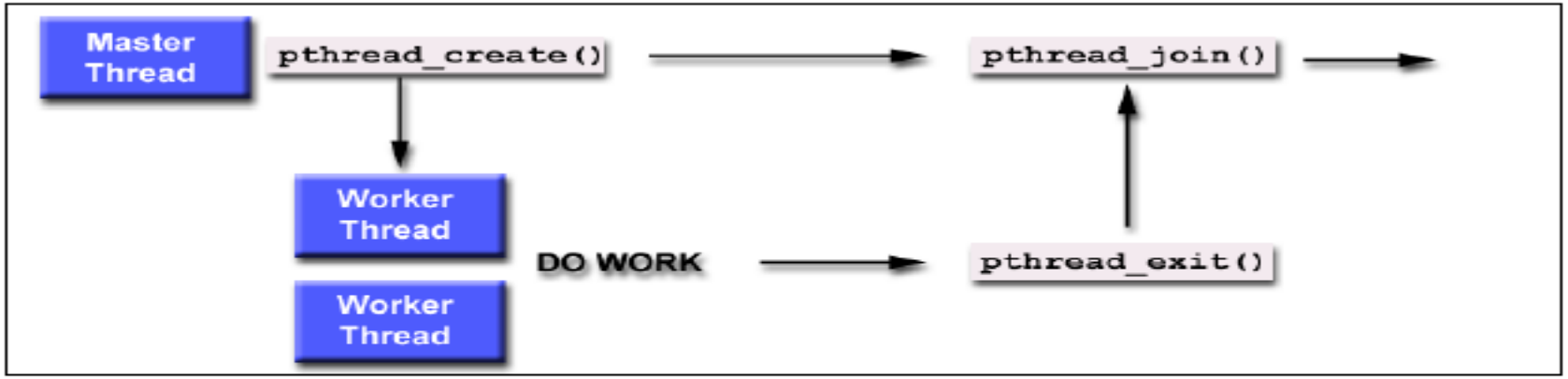
- Código simplificado para o slide anterior
  - (a) Thread despachante
  - (b) Thread operária

O uso de Threads

int pthread_join(pthread_t tid, void* status)

// a thread invocadora é bloqueada até que a thread tid termina

- tid A threadID pela qual deseja-se esperar;
- status O valor de retorno da thread execurando o exit(), será copiado para status




```
void main() {
        pthread_t tid;
        int status;
                pthread_create(&tid,NULL,thread_main,NULL);
        ....
        pthread_join(tid, (void*) &status);
        printf("Return value is: %d\n", status);
}
```

```
void *thread_main( ){
        int result;
        ....
        Pthread_exit((void*) result);
}
```

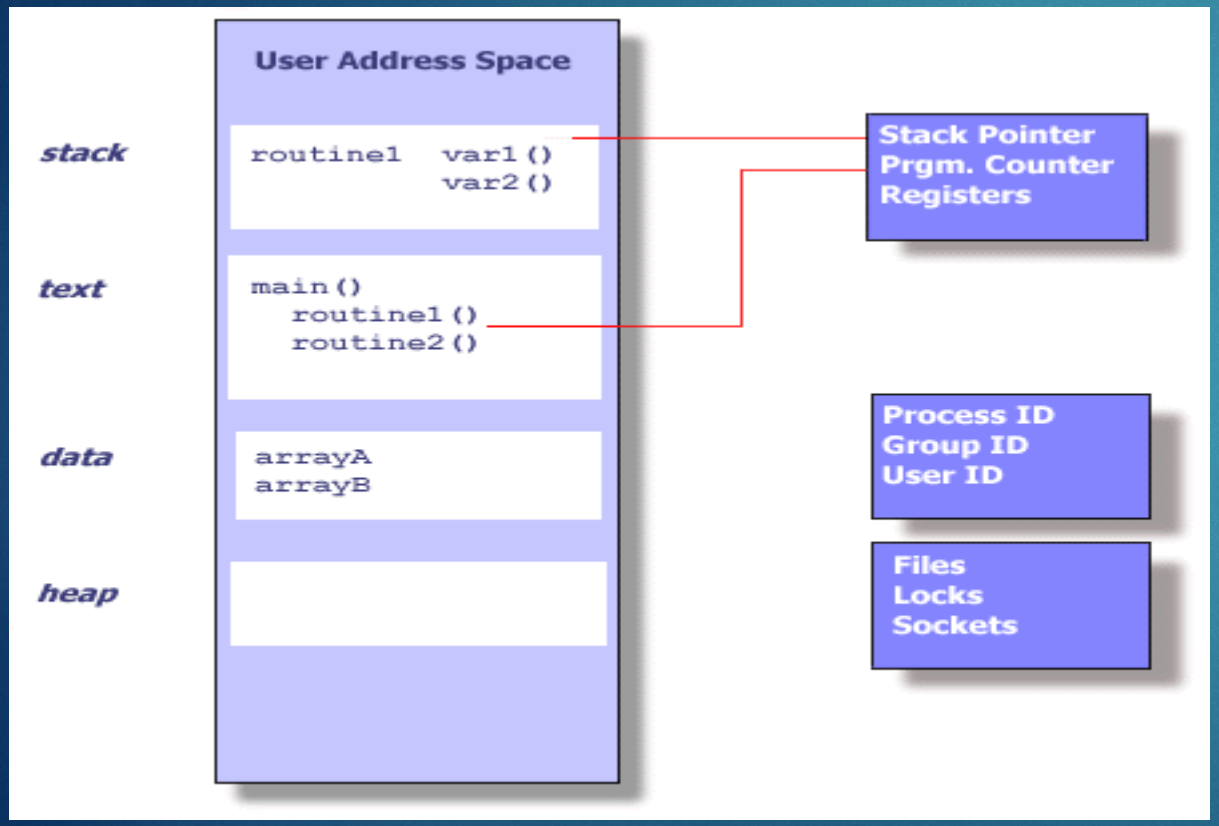
Threads POSIX

- Norma internacional IEEE POSIX1 1003.1 C
  - Apenas a API é normalizada
  - é fácil trocar a biblioteca que implementa a norma em tempo de compilação, mas não em tempo de execução
- Threads POSIX podem implementar threads em nível de usuário, em nível de kernel ou misto

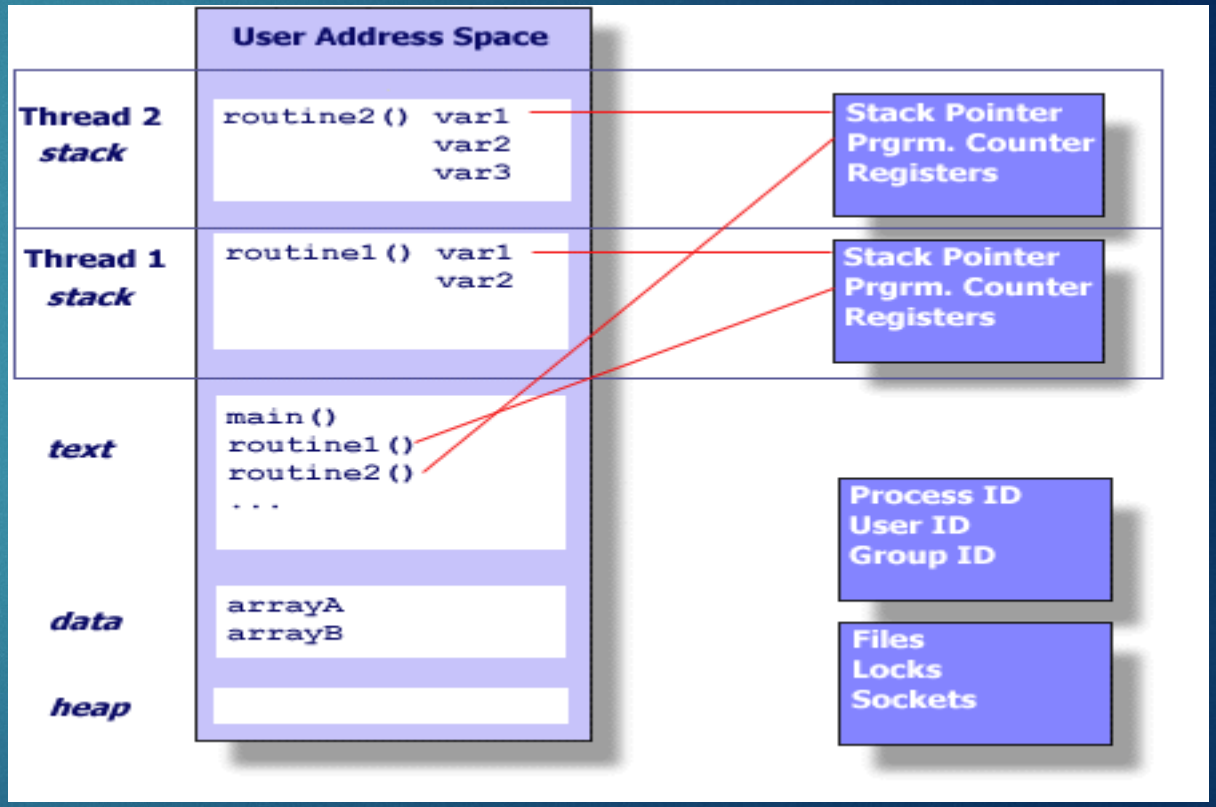
Pthreads

- POSIX 1003.1-2001 – pthreads
- API para gerenciamento de threads
  - >more /usr/include/pthread.h
  - > man pthread_create
- Threads co-existem num mesmo processo, compartilhando vários recursos, mas são escalonadas separadamente pelo sistema operacional
- Somente o mínimo necessário de recursos é replicado entre duas threads

Visão geral



Sem Threads



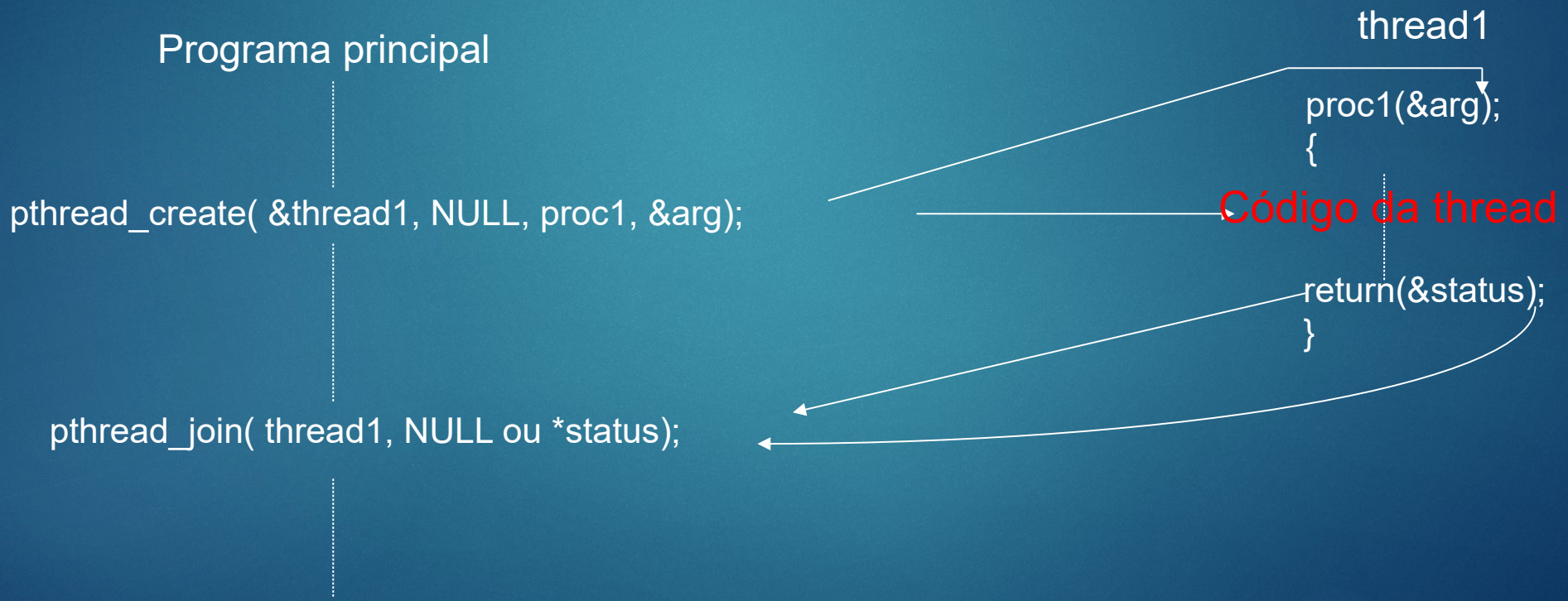
Com Threads

Threads POSIX

```
#include <pthread.h>
#include <stdlib.h>
#define NUM_THREADS 5
void *PrintHello(void *threadid)
{
  long tid = (long)threadid;
  printf("Olá! Sou a thread #%ld!\n", tid);
  pthread_exit(NULL);
}
int main (int argc, char *argv[])
{
 pthread_t threads[NUM_THREADS];
 int rc;
 long t;
 for(t=0; t<NUM_THREADS; t++){
  printf("main: criando thread %ld\n", t);
  rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
  if (rc){
   printf("ERRO; pthread_create() devolveu o erro %d\n", rc);
   exit(-1);
  }
 }
 for(t=0; t<NUM_THREADS; t++){
 pthread_join(threads[t], NULL);
 }
}
```

Pthreads

Interface portável do sistema operacional, POSIX, IEEE



PThread API

Prefix	Funcionality
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects
pthread_cond_	Condition variables
pthread_cond_attr	Condition attributes objects
pthread_key_	Thread-specific data keys

Como programar

- Usar
  - `#include "pthread.h"`
- Compilar com a opção –pthread
  - `gcc -pthread ex_pthread.c -o ex_pthread`

Criando uma thread

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

Encerrando uma thread

- Além da alternativa de simplesmente encerrar a função, é possível também
  - `void pthread_exit(void *retval);`

Pthread Join

- A rotina *pthread_join*() espera pelo término de uma thread específica

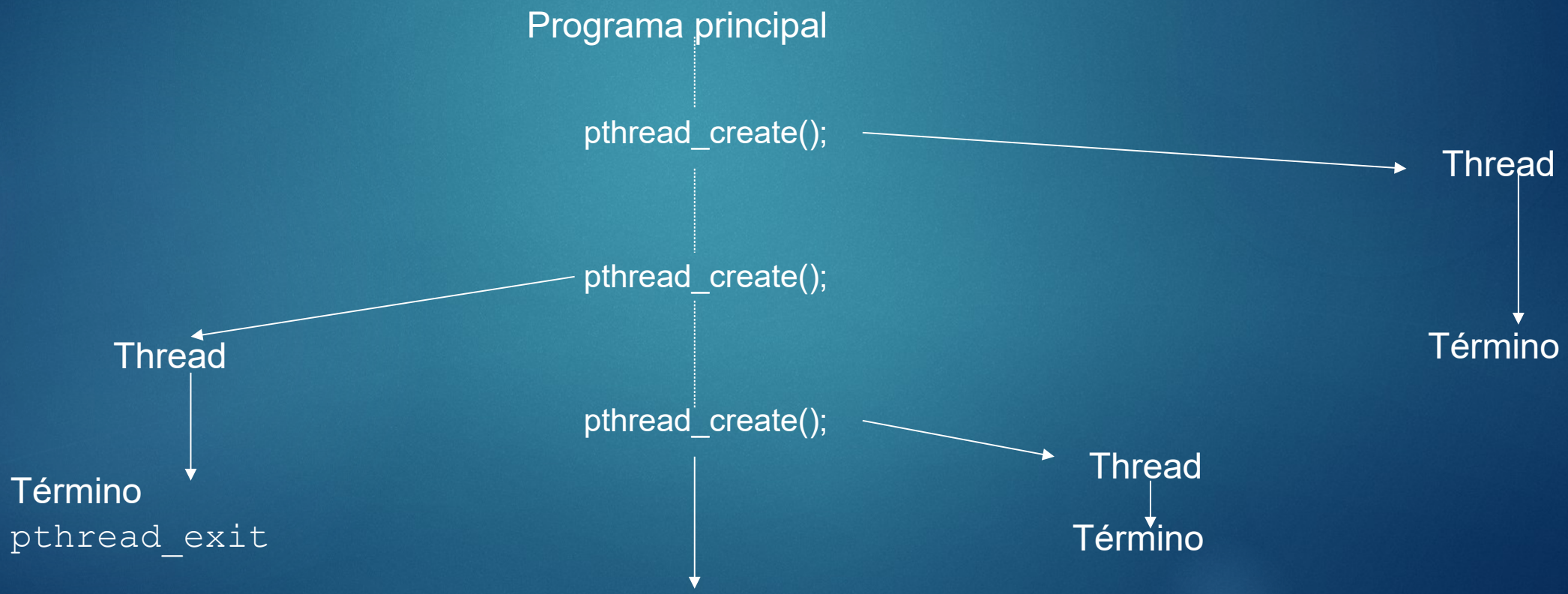
```
for (i = 0; i < n; i++)
        pthread_create(&thread[i], NULL, (void *) slave, (void *) &arg);

...thread mestre

for (i = 0; i < n; i++)
        pthread_join(thread[i], NULL);
```

Detached Threads (desunidas)

Pode ser que uma thread não precise saber do término de uma outra por ela criada, então não executará a operação de união. Neste caso diz-se que o thread criado é *detached* (desunido da thread pai progenitor)



Pthreads

- Quando dois ou mais threads podem simultaneamente alterar às mesmas variaveis globais poderá ser necessário sincronizar o acesso a este variavel para evitar problemas.
- Código nestas condição diz-se “uma seção critica”
  - Por exemplo, quando dois ou mais threads podem simultaneamente incrementar uma variável x
  - /* codigo – Seção Critica */
  - x = x +1 ;
- Uma seção crítica pode ser protegida utilizando-se pthread_mutex_lock() e pthread_mutex_unlock()

Lock/Mutex

```
/* Note scope of variable and mutex are the same */
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int counter=0;

/* Function C */
void functionC()
{
   pthread_mutex_lock( &mutex1 );
   counter++
   pthread_mutex_unlock( &mutex1 );
}
```

Fontes de problemas

- Condição de corrida
  - Não assuma uma ordem específica para a execução das threads
  - Um código pode funcionar muito bem em determinados momentos e gerar sérios problemas em outros
- Thread safe code
  - Use bibliotecas que possuam chamadas “thread safe”
  - Várias chamadas à mesma função devem ser permitidas
- Deadlock

Leitura complementar

- http://pages.cs.wisc.edu/~travitch/pthreads_primer.html
- https://computing.llnl.gov/tutorials/pthreads/exercise.html
- http://students.cs.byu.edu/~cs460ta/cs460/labs/pthreads.html

Threads STD (C++11)

- Biblioteca padrão STD passou a incluir suporte a threads
- Criação e gerência do ciclo de vida de threads
- #include <thread>
- Compilador deve suportar C++11
  - Opções -std=c++0x" ou "-std=c++11" no GCC para ativar suporte

Threads STD (C++11)

```
#include <iostream>       // std::cout
#include <thread>         // std::thread

void foo() { std::cout << "foo!" << endl; }

void bar(int x) {std::cout << "bar: " << x << endl; }

int main()
{
  std::thread first (foo);     // spawn new thread that calls foo()
  std::thread second (bar,10);  // spawn new thread that calls bar(10)

  std::cout << "main, foo and bar now execute concurrently...\n";

  // synchronize threads:
  first.join();                // pauses until first finishes
  second.join();               // pauses until second finishes

  std::cout << "foo and bar completed.\n";

  return 0;
}
```

Threads STD (C++11)

```
class thread {
 public:
    class id;
    typedef /*implementation-defined*/ native_handle_type;

    thread() noexcept;
    template <class F, class ...Args> explicit thread(F&& f, Args&&... args);
    ~thread();
    thread(const thread&) = delete;
    thread(thread&&) noexcept;
    thread& operator=(const thread&) = delete;
    thread& operator=(thread&&) noexcept;

    void swap(thread&) noexcept;
    bool joinable() const noexcept;
    void join();
    void detach();
    id get_id() const noexcept;
    native_handle_type native_handle();

    static unsigned hardware_concurrency() noexcept;
};
```

Threads STD (C++11)

`yield` (C++11)	(function)
`get_id` (C++11)	(function)
`sleep_for` (C++11)	(function)
`sleep_until` (C++11)	(function)

Threads STD (C++11)

```
#include <thread>
#include <iostream>
#include <vector>

void hello(){
    std::cout << "Hello from thread ";
    std::cout << std::this_thread::get_id() << std::endl;
}

int main(){
    std::vector<std::thread> threads;
    for(int i = 0; i < 5; ++i){
        threads.push_back(std::thread(hello));
    }
    for(auto& thread : threads){
        thread.join();
    }
    return 0;
}
```

Threads STD (C++11)

```
#include <thread>
#include <iostream>
class TestClass {
    int i;
public:
    TestClass(int n) { i = n; }
    void greeting(std::string const& message) const {
        std::cout << message << std::endl;
        std::cout << "My number: " << n << std::endl;
    }
};
int main() {
    TestClass x(10);
    std::thread t(& TestClass::greeting,&x,"Hello! ");
    t.join();
}
```

Threads STD (C++11)

```
#include ...

int main()
{
    std::shared_ptr<SayHello> p(new SayHello);
    std::thread t(&SayHello::greeting,p,"goodbye");
    t.join();
}

// What if you want to pass in a reference to an
// existing object, and a pointer just won't do?
// That is the task of std::ref.
```

Threads STD (C++11)

```
#include <thread>
#include <iostream>

void write_sum(int x,int y)
{
    std::cout<<x<<" + "<<y<<" = "<<(x+y)<<std::endl;
}

int main()
{
    std::thread t(write_sum,123,456);
    t.join();
}
```

Threads STD (C++11)

```
#include <thread>
#include <iostream>
#include <functional> // for std::ref
class PrintThis
{
public:
    void operator()() const
    {
        std::cout<<"this="<<this<<std::endl;
    }
};
int main()
{
    PrintThis x;
    x();
    std::thread t(std::ref(x));
    t.join();
    std::thread t2(x);
    t2.join();
}
```

```
/*
this=0x7fffb08bf7ef
this=0x7fffb08bf7ef
this=0x42674098
*/
```

Threads STD (C++11)

```
#include <thread>
#include <iostream>
#include <functional>

void increment(int& i)
{
    ++i;
}

int main()
{
    int x=42;
    std::thread t(increment,std::ref(x));
    t.join();
    std::cout<<"x="<<x<<std::endl;
}
```

Threads STD (C++11)

- Cenas dos próximos capítulos
- std::mutex
- std::lock_guard<>
- std::unique_lock<>
- Evitando deadlocks com múltiplos mutexes...

Threads STD (C++11)

- Leitura complementar
- http://cppwisdom.quora.com/Why-threads-and-fork-dont-mix
- http://www.linuxprogrammingblog.com/threads-and-fork-think-twice-before-using-them

Threads Java

- Java incorporou ao seu núcleo as primitivas de simultaneidade.
- Outras linguagens, como C e C++, necessitam de bibliotecas específicas para permitir que seus programas sejam executados com múltiplas threads.
- Threads podem ser criadas através da classe Thread e da interface Runnable.

Threads em Java

- **Classe Thread**
  - Faz parte do pacote java.lang.
  - Uma classe que herda de Thread tem a possibilidade de ser executada de forma paralela.
  - É necessário sobrescrever o método **run()**.
  - Para iniciar a execução de um objeto Thread, seu método **start()** deve ser invocado.

Threads em Java

- **Interface Runnable**
  - Presente no pacote java.lang.
  - Possui apenas um método para ser implementado: o método **run()**.
  - Um objeto **Thread** deve ser criado, passando como parâmetro para seu construtor um objeto **Runnable**.
  - A invocação do método start() do objeto Thread executará as instruções do método run() do objeto Runnable.

Threads em Java

- **Thread versus Runnable**
  - A classe Thread implementa a interface Runnable.
  - A classe Thread é mais fácil de ser utilizada e é a mais indicada para aplicações simples.
  - A implementação da interface Runnable é requerida quando precisamos dar suporte a *multithreading* em uma classe que já é derivada de outra diferente de Thread.
  - **Java não suporta herança múltipla.**

Threads em Java

- **Escalonamento**
  - A JVM possui um escalonador de threads responsável por decidir que thread vai executar em determinado momento.
  - A linguagem Java não estabelece uma forma padronizada para o escalonamento de threads, não garantindo, portanto, que haja justiça em sua execução.
  - Dois modelos de escalonamento norteiam a implementação da JVM: ***green-thread model*** **e** ***o native-thread model*****.**

Threads em Java

- ***Green-Thread Model***
  - As threads são escalonadas pela própria JVM.
  - A JVM é a responsável por salvar o contexto das threads.
  - O sistema operacional desconhece a existência das threads controladas pela JVM.
- ***Native-Thread Model***
  - As threads são escalonadas pelo sistema operacional sobre o qual a JVM está sendo executada.
  - O SO não faz distinção seus processos e as threads da JVM.

Threads em Java

- **Prioridades**
  - Diz respeito à prioridade de execução de cada thread.
  - Toda thread possui uma prioridade no intervalo de Thread.MIN_PRIORITY a Thread.MAX_PRIORITY (intervalo de 1 a 10).
  - Quando uma thread é criada, ela recebe, por padrão, uma prioridade igual a cinco (Thread.NORM_PRIORITY).
  - Manipulada através dos métodos getPriority e setPriority.

Threads em Java

- **Prioridades**
  - A forma como as prioridades são utilizadas depende do SO, da JVM e do modelo de escalonamento utilizados.
  - Nas implementações da JVM que seguem *green-thread model*, em geral não há preempção por tempo.
    - Threads perdem a CPU quando concorrem com outras de maior prioridade ou então quando são bloqueadas.
  - A plataforma Windows segue o escalonamento ***round-robin***.

Threads em Java

- **Método sleep()**

```
static void sleep(long millis)
static void sleep(long millis, int nanos)
```

  - Faz a thread em execução dormir por um período de tempo especificado.
  - A thread passa para o estado **Esperando**.
  - A precisão depende dos temporizadores do sistema e do escalonador.
  - Lança uma **InterruptedException.**

Threads em Java

- **Método yield()**

```
static void yield()
```

  - Faz a thread em execução voltar para o estado **Pronta**.
    - Permite que outras threads de mesma prioridade sejam executadas.
  - Não há como garantir que as outras threads serão escolhidas pelo escalonador.
    - O método yield() pode não funcionar!

Threads em Java

- **Método join()**

```
void join()
void join(long millis)
void join(long millis, int nanos)
```

- Exemplo:
  - A thread em execução será pausada até que a thread `t1` termine.
- Lança uma **InterruptedException.**

```
t1.join()
```

Threads em Java

- **Compartilhamento de Memória**
  - Através de membros de classe (*static*).
  - Utilização de objetos compartilhados.