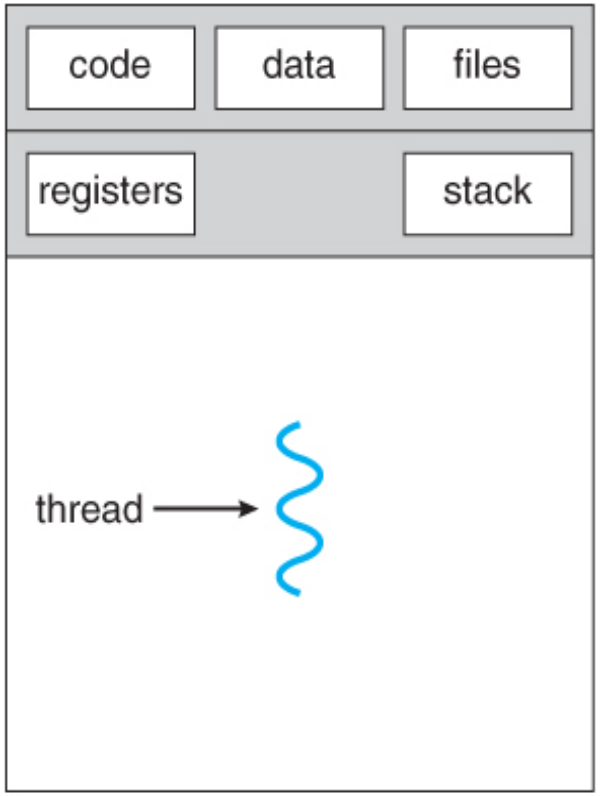
Programação Concorrente

Marcelo Veiga Neves

Marcelo.neves@pucrs.br

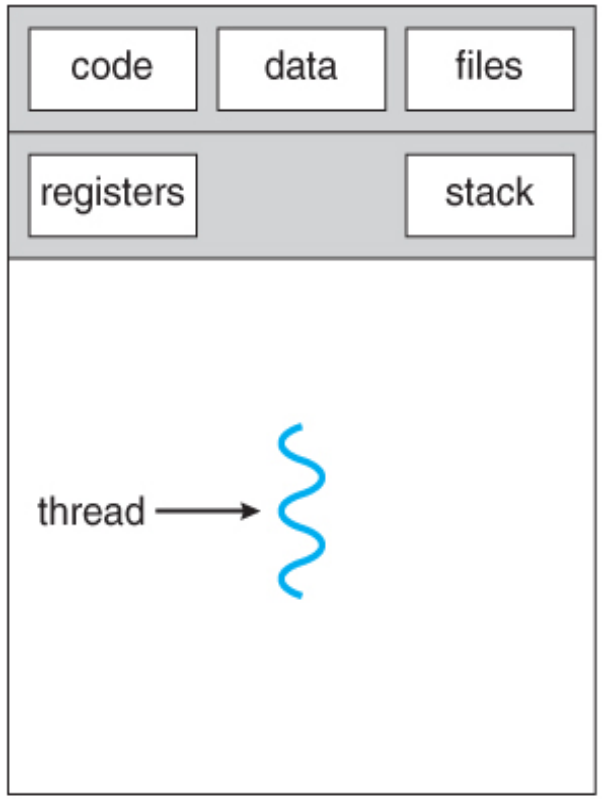
Processos

- Em Sistemas Operacionais, um processo é uma instância de programa em execução
- Um programa pode "gerar" vários processos
- Cada processo contém diferentes seguimentos:
  - Instruções que serão executadas (code/text)
  - Variáveis globais e estruturas de dados dinâmicas (data/heap)
  - Memória usada para variáveis locais (stack)
  - Registradores usados para execução das instruções (registers)
  - E outros recursos como arquivos, sockets, etc.
- Todo processo tem pelo menos um fluxo de execução



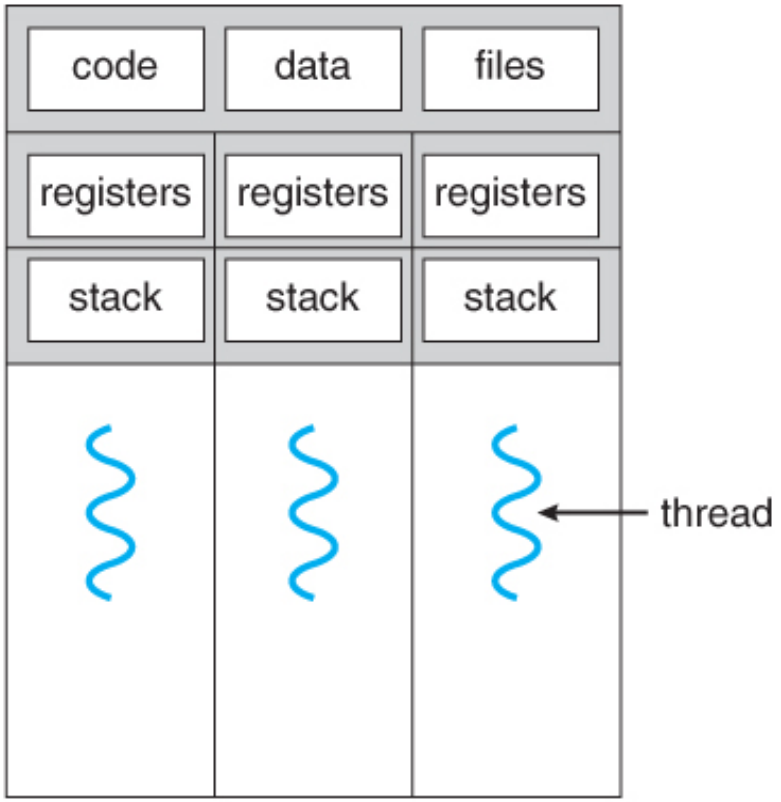
Processo e Thread

- Um **thread** representa um fluxo de execução e contém um contador de programa (*program counter*), uma *stack* e um conjunto de registradores)
- Um **processo pesado** é um processo com uma única thread
- Podemos criar programas concorrentes formados por múltiplos processos pesados
  - Processos pesados não compartilham a memória diretamente
  - Necessidade de comunicação entre processos (IPC – Inter-processing communication)



Processo com múltiplas Threads

- Podemos criar programas concorrentes formados por múltiplas threads
- Podemos criar programas concorrentes formados por múltiplas threads
- Cada thread possui *program counter*, *stack* e registradores próprios (somente o que é necessário para o fluxo de execução local)
- Todo o resto é compartilhado (memória global, arquivos abertos, etc.)
- Neste caso, chamamos de **processos leves**



Processos vs Threads

Aspecto	Processos Pesados	Threads (Processos Leves)
Isolamento	Isolados; espaço de memória independente.	Compartilham o mesmo espaço de memória do processo.
Criação	Relativamente custoso; operações de criação e terminação são mais lentas.	Mais rápidos de criar e terminar.
Comunicação	Usa Comunicação Entre Processos (IPC), como pipes, sockets, arquivos, etc.	Direta; através de variáveis globais compartilhadas.
Falha/Erro	Se um falhar, não afeta outros diretamente.	Se um falhar (como um acesso inválido à memória), pode afetar todos no mesmo processo.
Recursos	Cada um possui recursos separados (registradores, PC, espaço de memória).	Compartilham recursos do processo, exceto registradores e stack.
Uso	Ideal para tarefas independentes e/ou que requerem isolamento.	Ideal para tarefas que necessitam de comunicação e coordenação frequentes.

Como programar com Threads?

- Linguagens Java, C e Go
- Como criar um thread
- Como esperar pela conclusão de um thread
- Como controlar acesso a uma seção crítica

Threads em C

- Em C, a biblioteca POSIX Threads (pthreads) é o método padrão para criação de threads
- As threads são implementadas como funções
  - Posso ter múltiplas funções executando de forma concorrente
- Código precisa incluir os cabeçalhos da biblioteca pthread:

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

- Compilação precisa ligar com a biblioteca pthread:

```
gcc -o prog programa.c -pthread
```

C: Criação de Thread

- A função **pthread_create()** é usada para criar uma nova thread
  - Recebe um ponteiro de função com parâmetro
- **pthread_t** é o tipo de dado usado para identificar uma thread.

```c
void* minha_thread(void* arg) {
    // código da thread
    return NULL;
}

pthread_t thread_id;
pthread_create(&thread_id, NULL, minha_thread, NULL);
```

C: Esperando a Conclusão de Uma Thread

- As threads são interrompidas quando a main() termina
- Precisamos utilizar a **pthread_join()** para esperar que uma thread termine

```
pthread_t thread_id;
pthread_create(&thread_id, NULL, minha_thread, NULL);
pthread_join(thread_id, NULL);
printf("Executando função principal!\n");
```

C: Passando Argumentos para Threads

- A passagem de argumentos para as funções que executam como thread é feito por ponteiro

```c
void* imprimir_numero(void* numero) {
    printf("Número: %d\n", *(int*)numero);
    return NULL;
}

int main() {
    int valor = 5;
    pthread_t thread_id;
    pthread_create(&thread_id, NULL, imprimir_numero, &valor);
    pthread_join(thread_id, NULL);
    return 0;
}
```

C: Controlando Acesso a Seções Críticas

- Mutex (**exclusão mútua**) é uma ferramenta de sincronização para prevenir que múltiplas threads acessem a seção crítica

```c
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* minha_thread() {
    pthread_mutex_lock(&mutex);
    // Seção crítica...
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

Exemplo de código em C

- Compilar e executar o programa contador.c
  - gcc -o contador contador.c -pthread
  - ./contador

Threads em Java

- A criação de threads em Java pode ser feita por herança ou interface
- Método 1: Extendendo a classe Thread:
  - Criar uma classe que estenda **Thread** e sobrescrever o método **run()**.
  - Iniciar a thread com o método **start()**.
- Método 2: Implementando a interface Runnable:
  - Criar uma classe que implementa **Runnable** e definir o código dentro do método **run()**.
  - Passar uma instância do Runnable para o construtor de uma nova Thread e iniciá-la.

Java: Extendendo a Classe Thread

```
class MinhaThread extends Thread {
    @Override
    public void run() {
        // código da thread
    }
}

MinhaThread t = new MinhaThread();
t.start();
```

- Benefício: acesso direto aos métodos da Thread.
- Desvantagem: Limita a herança (Java não permite herança múltipla).

Java: Implementando a Interface Runnable

```
class MeuRunnable implements Runnable {
    @Override
    public void run() {
        // código da thread
    }
}

Thread t = new Thread(new MeuRunnable());
t.start();
```

- Flexibilidade: permite que sua classe herde de outras classes.
- É a abordagem preferida pela maioria dos desenvolvedores.

Java: Esperando a Conclusão de Uma Thread

- As threads continuam executando mesmo que a main() termine
- Mesmo assim, podemo utilizar **.join()** para esperar/bloquear que uma thread termine

```
MinhaThread t = new MinhaThread();
t.start();

try {
    t.join(); // Espera t terminar sua execução
} catch (InterruptedException e) {
    e.printStackTrace();
}
// Continua a execução
```

Java: Métodos Importantes da Classe Thread

- **start()**: Inicia a execução da thread.
- **run()**: Define o código que a thread irá executar.
- **join()**: Espera a thread terminar sua execução.
- **sleep(long milis)**: Pausa a thread pelo tempo especificado (estático).
- **interrupt()**: Interrompe uma thread.
- **isAlive()**: Verifica se a thread ainda está em execução.
- **getId()**: Retorna o identificador único da thread.

Consultar o Javadoc para mais:
https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

Java: Controlando Acesso a Seções Críticas

- **Seção crítica**: parte do código onde o acesso simultâneo por múltiplas threads pode causar uma **condição de corrida**
- Existe três mecanimos de controle de acesso à seção crítica:
  - Métodos synchronized
  - Blocos synchronized
  - ReentrantLock (lock/unlock)

Java: Modificador synchronized

- Garante que apenas uma thread acesse um método ou bloco de código por vez
- Método:

```java
// Método synchronized
public synchronized void meuMetodo() {
    // Seção crítica
}
```

- Bloco:

```java
// Blocos synchronized
Object lock = new Object();
synchronized(lock) {
    // Seção crítica
}
```

Java: Classe ReentrantLock

- Mecanimos mais flexivel para controle de seção crítica

```
ReentrantLock lock = new ReentrantLock();

lock.lock();
try {
    // Seção crítica
} finally {
    lock.unlock();
}
```

Exemplo de código em Java

- Compilar e executar o programa Contador.java
  - javac Contador.java
  - java Contador

Threads em Go

- Em Go, a concorrência é realizada através de **goroutines**.
- Uma goroutine é uma função leve que é executada concorrentemente com outras funções.
  - Não é exatamente um thread tradicional de sistema operacional
  - Implementação própria para ter múltiplos fluxos de execução
  - São ainda mais leves do que threas tradicionais

Go: Criação de goroutines

- Uma goroutine é lançada simplesmente usando a palavra-chave **go** antes de uma chamada de função.
- Sem a palavra go, ela funciona como uma função normal

```go
func minhaGoroutine() {
    // Código da goroutine
}

go minhaGoroutine() // Inicia uma goroutine
```

Go: Esperando a Conclusão de Uma Goroutine

- As goroutines são interrompidas quando a main() termina
- Para esperar pela conclusão, podemos usar WaitGroups ou Channels

- Exemplo WaitGroup:
  - wg.Add(1) incrementa o contador de goroutines.
  - wg.Done() decremente o contador.
  - wg.Wait() bloqueia até que o contador chegue a zero.

```go
var wg sync.WaitGroup

func minhaGoroutine() {
    defer wg.Done()
    // Faça algo...
}

func main() {
    wg.Add(1)
    go worker()
    wg.Wait()
}
```

Go: Controlando Acesso a Seções Críticas

- Go oferece recurso de exclusão mútua (mutex) para controlar acesso à seções críticas e evitar condições de corrida
- Mutex (Mutual Exclusion):
  - Lock(): Bloqueia o acesso.
  - Unlock(): Libera o acesso.
- RWMutex (Read-Write Mutex):
  - RLock(): Bloqueia acesso para escrita, mas permite múltiplas leituras.
  - RUnlock(): Libera leitura.
  - Lock(): Bloqueia tanto leitura quanto escrita.
  - Unlock(): Libera ambos.

```
var mutex sync.Mutex

func increment() {
    mutex.Lock()
    // Seção crítica
    mutex.Unlock()
}
```

Exemplo de código em Go

- Compilar e executar o programa Contador.java
  - go build contador.go
  - ./contador