Concorrência

Prof. Hugo de Paula

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Departamento de Ciência da Computação

Sumário

Principais componentes de um SO

- *Process manager*: trata da criação e operação dos processos.
- *Thread manager*: trata da criação, sincronização e escalonamento das threads.
- *Communication manager*: trata da comunicação entre processos e threads num mesmo computador.
- *Memory manager*: trata do gerenciamento da memória física e virtual.
- *Supervisor*: dispara interrupções, *traps* de chamadas de sistema e outras exceções.

Kernel

- Processo constantemente em execução com acesso total a todos os recursos físicos e privilégios de acesso no computador.
- Possui espaço de endereçamento protegido.
- Acesso ao espaço de endereçamento do kernel através de exceções como uma interrupção ou *traps* de chamadas de sistema.
  - Quando uma *trap* é executada, assim como qualquer outro tipo de exceção, o hardware força o processador a executar o tratador do kernel para evitar acesso ilícito ao endereçamento do kernel.

Processos e Threads

Processo

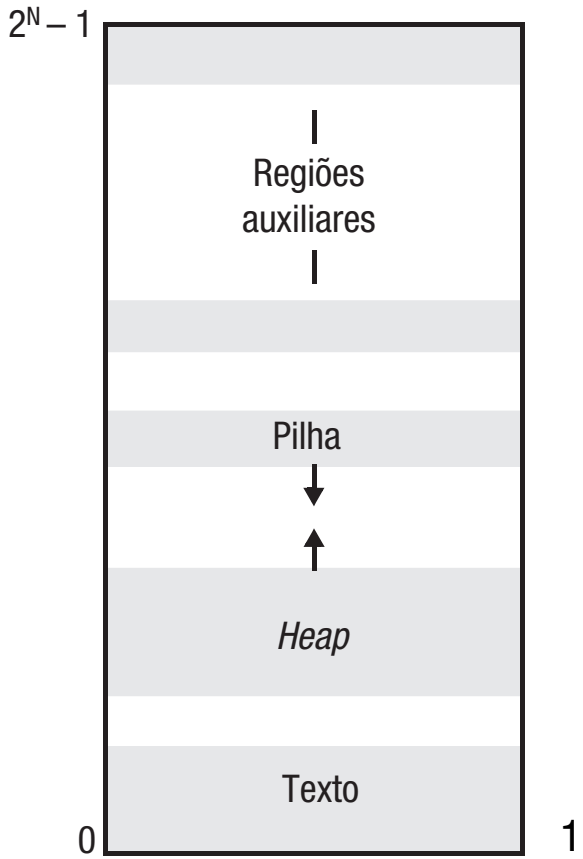
Consiste de um ambiente de execução, a unidade de gerenciamento de recursos. Possui seu espaço de endereçamento próprio.

Thread

É uma atividade, ou linha de execução.

- Um espaço de endereçamento é uma região contínua de memória virtual que é acessível para as threads do processo que a possui.

Espaço de endereçamento



[1] Coulouris, Figura 7.3

Estados de um Processo

Processos podem coexistir em um desses três estados:

Em execução

processo está de posse do processador e está utilizando tempo de CPU.

Bloqueado

processo está suspenso, esperando alguma atividade, como I/O, completar.

Em espera

Processo está pronto, esperando pelo processador.

Sistemas com memória compartilhada

Quando múltiplos processos ou *threads* modificam a mesma memória, podem ocorrer situações inexperadas:

Condição de corrida (*race condition*)

Múltiplos processos competindo pela mesma região de memória. É necessário implementar um mecanismo de exclusão mútua, em que um processo é bloqueado usando um mecanismo de trava (*lock*) enquanto o outro processo estiver acessando a seção crítica.

Deadlock

Ocorre quando múltiplos processos permanecem bloqueados esperando uns aos outros.

Categorias de Concorrência

Concorrência física

Múltiplos processadores independentes (controle de múltiplas *threads*)

Concorrência lógica

Simula concorrência física com o compartilhamento de processamento em *slots* de tempo.

Corotinas (*quasi-concorrência*)

Única thread de controle, que é uma sequencia de pontos de programa que controlam o fluxo de execução.

Níveis de concorrência

Nível de instrução de máquina

pipelining e previsão são utilizadas, e duas ou mais instruções de máquina podem executar simultaneamente.

Nível de instrução de linguagem de programação

dois ou mais comandos da linguagem podem executar simultaneamente.

Nível de unidade

dois ou mais sub-programas podem executar simultaneamente, e é o caso das *threads* e co-rotinas.

Nível de programa

dois ou mais programas podem executar simultaneamente.

Níveis de concorrência

Nível de instrução de máquina *Relacionado ao hardware.*

pipelining e previsão são utilizadas, e duas ou mais instruções de máquina podem executar simultaneamente.

Nível de instrução de linguagem de programação

dois ou mais comandos da linguagem podem executar simultaneamente.

Nível de unidade

dois ou mais sub-programas podem executar simultaneamente, e é o caso das *threads* e co-rotinas.

Nível de programa *Relacionado ao Sistema operacional.*

dois ou mais programas podem executar simultaneamente.

Modelos de concorrência

Threads e *locks*

Threads se comunicam por memória compartilhada e lock, semáforos e monitores, gerenciam as condição de corrida. Ex.: Java

Atores

Atores são entidades independentes que se comunicam por troca de mensagem. São leves e evitam a condição de corrida. Ex.: Erlang

CSP (*Communicating Sequential Processes*)

Comunicação se dá por canais que podem ser criados, escritos, lidos ou passados entre processos de forma independente. Ex.: Go

Sincronização

Sincronização

é o mecanismo que controle a ordem em que as tarefas serão executadas.

Sincronização Cooperativa

Sincronização cooperativa é necessária entre a Task A e a Task B quando a Task A deve esperar que a Task B termine alguma atividade para continuar seu processamento.

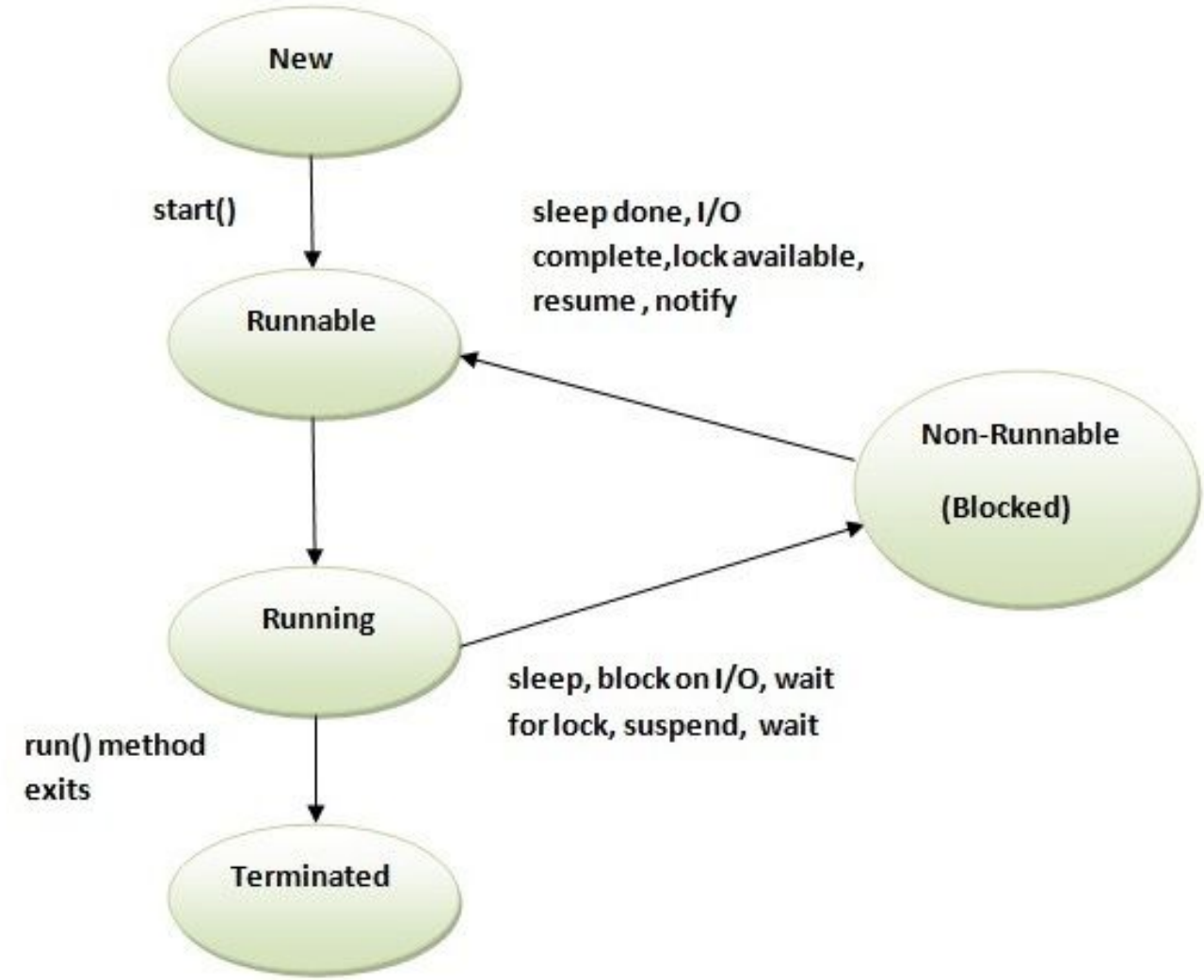
Sincronização Competitiva

Sincronização competitiva é necessária quando Task A e Task B precisam acessar o mesmo recurso simultaneamente.

Threads em Java

- Classe Thread pertence ao pacote java.lang.
- Palavra chave **synchronized** é usada para estabelecer exclusão mútua.
- Java Runtime não detecta ou previne *deadlocks*.
- Recursos podem ser compartilhados passando-se objetos como parâmetro no construtor.

Ciclo de vida de uma thread em Java



Métodos de gerenciamento de threads Java

Thread(ThreadGroup group, Runnable target, String name)
Cria uma nova *thread* no estado *SUSPENDED*, a qual pertencerá a *group* e será identificada como *name*; a *thread* executará o método *run( )* de *target*.

setPriority(int newPriority), getPriority( )
Configura e retorna a prioridade da *thread*.

run( )
A *thread* executa o método *run( )* de seu objeto de destino, caso ele tenha um; caso contrário, ela executa seu próprio método *run( )* (*Thread* implementa *Runnable*).

start( )
Muda o estado da *thread* de *SUSPENDED* para *RUNNABLE*.

sleep(long millisecs)
Passa a *thread* para o estado *SUSPENDED* pelo tempo especificado.

yield( )
Passa para o estado *READY* e ativa o escalonamento.

destroy( )
Termina (destrói) a *thread*.

2

[2] Coulouris, Figura 7.8

Chamadas de sincronização de thread Java

thread.join(long millisecs)
Bloqueia até a *thread* terminar, mas não mais que o tempo especificado.

thread.interrupt( )
Interrompe a *thread*: a faz retornar de uma invocação a método que causa bloqueio, como *sleep( )*.

object.wait(long millisecs, int nanosecs)
Bloqueia a *thread* até que uma chamada feita para *notify( )*, ou *notifyAll( )*, em *object*, ative a *thread*, ou que a *thread* seja interrompida ou, ainda, que o tempo especificado tenha decorrido.

object.notify( ), object.notifyAll( )
Ativa, respectivamente, uma ou todas as *threads* que tenham chamado *wait( )* em *object*.

3

[3] Coulouris, Figura 7.9

Exemplo de pilha compartilhada

- Todo objeto em java possui um *lock*.
- Quando uma *Thread* tenta executar um método **synchronized**, ele primeiro precisa adquirir o *lock* do objeto.
  - Se o *lock* está alocado a outra *thread*, ela deve esperar.
  - Ao sair do método, o *lock* deve ser liberado.
- `wait()` é usado para travar uma *thread manualmente*.
- `notify()` ou `notifyAll()` são usados para acordar uma *Thread* em espera.

Concorrência em Java

- Suporte à concorrência desenvolveu-se enormemente desde a versão do Java 8.
  - Classe Semaphore nativa.
  - Funcionalidade adicional em objetos em *Lock/Condition*.
- Interface Executor provê diferentes mecanismos de suporte para Threads:
  - Alocação de Threads em diferentes núcleos em uma máquina *multicore*.
  - Retorna resultados de *futures* de uma tarefa assíncrona.
  - ForkJoin: usado para distribuir threads entre múltiplos núcleos.
  - *Concurrent Annotations* com muitas aplicações como em JAX-WS para o gerenciamento de POJO para a geração de *Web Services*.

Concurrent Annotations

- É boa prática de programação utilizar anotações para documentar o código.
- Anotações são processadas em tempo de compilação ou em tempo de execução ou em ambos.
- Alguns frameworks e bibliotecas utilizam anotações como no caso do JAX-WS, por exemplo, para transformar um, POJO em *Web Services Resources*.

```
public class ContaBancaria {
    private Object credencial = new Object();

    /* Saldo guardado por uma credencial porque o acesso
    * só é possível se estiver com o lock da credencial.
    */
    @GuardedBy("credencial")
    private int saldo;
}
```

Thread Pool Pattern

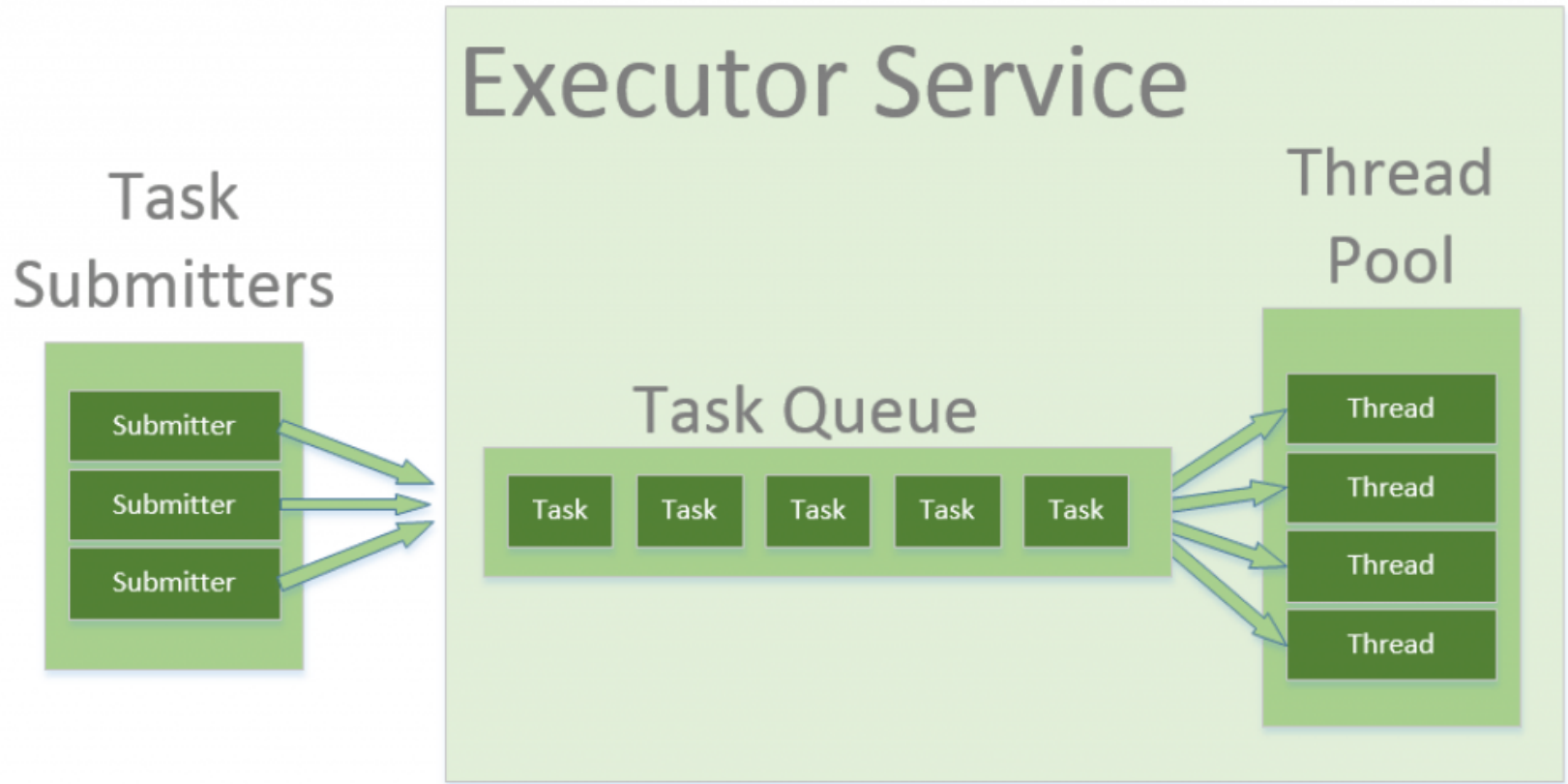
- Motivação: *Thread* por tarefa ou request possui problemas de performance.
  - *Overhead* de recursos computacionais na criação/destruição das *Threads*.
  - Excesso de *Threads* também afeta desempenho do escalonador e *overhead* de mudança de contexto.

Thread Pool Pattern

O padrão *Thread Pool* permite otimizar os recursos do sistema operacional, limitando o paralelismo e reaproveitando recursos de Thread.

- *Thread Pool* gerencia um conjunto de trabalhadores (*worker pool*) de Threads.

Thread Pool



4

[4] Fonte: https://www.baeldung.com/thread-pool-java-and-guava

Elementos de um *Thread Pool*

Executor Framework possui três elementos essenciais:

- `Executor`: interface funcional com um único método para enviar instâncias de *Runnable* para execução.
- `ExecutorService`: sub-interface de `Executor` que contém vários métodos para controlar o andamento das tarefas e gerenciar o término dos serviços.
- `ThreadPoolExecutor`: Implementa as interfaces anteriores e provê uma implementação concreta de um *Thread Pool*.

Elementos de um *Thread Pool*

Acoplamento entre uma *Thread* e uma tarefa

```
void executeTasks() {
    while (hasTasks()) {
        new Thread(new RunnableImpl()).start();
    }
}
```

Executor desvincula a tarefa da criação da *Thread*

```
void executeTasks() {
    Executor executor = new ExecutorImpl();
    while (hasTasks()) {
        executor.execute(new RunnableImpl());
    }
}
```