Sistemas Operativos

Licenciatura Engenharia Informática
Licenciatura Engenharia Computacional

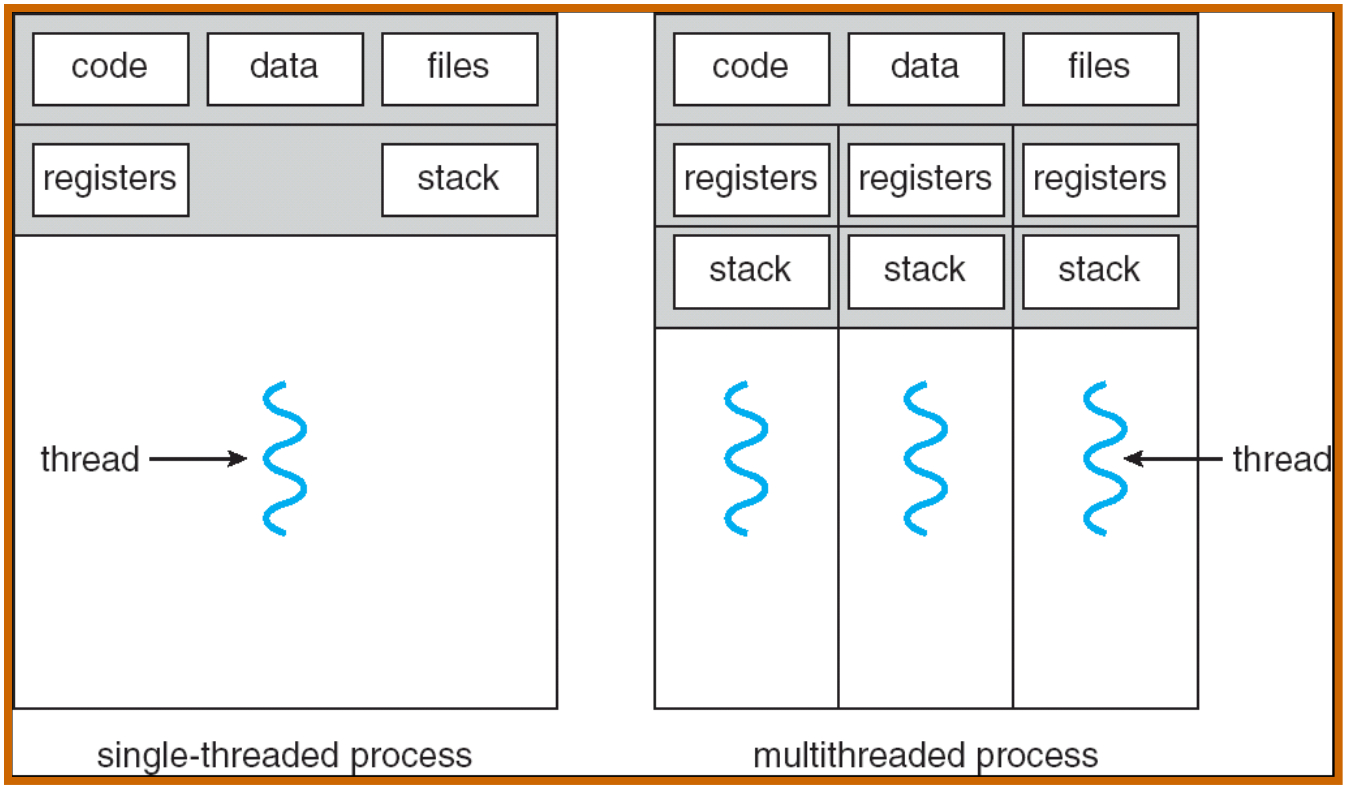
Ano letivo 2023/2024

Nuno Lau (nunolau@ua.pt)

Threads

- Programas têm geralmente de executar diversas atividades distintas
- Usando *threads,* o programador pode desenvolver o programa como um conjunto de fluxos de execução sequenciais, um para cada atividade
- Cada *thread* comporta-se como tendo o seu processador próprio.
- Todas as *threads* do mesmo processo partilham espaço de endereçamento (memória)

Processos *Single* e *Multi threaded*



Processos e *Threads*

Per-process items	Per-thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Criar POSIX *Threads*

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

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

#define NUM_THREADS 5

void *PrintMsg(void *threadid) {
   long tid;
   tid = (long)threadid;
   printf("Hello World! Thread ID, %d\n", tid);
   pthread_exit(NULL);
}

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

   for( i = 0; i < NUM_THREADS; i++ ) {
      printf( "main() : creating thread, %d\n",i);
      rc = pthread_create(&threads[i], NULL, PrintMsg, (void *)i);

      if (rc) {
         printf("Error: unable to create thread, %d\n", rc);
         exit(1);
      }
   }
   pthread_exit(NULL);
}
```

Java *Threads*

- Java *threads* são geridas pela JVM ⇝ Foi pensado desde o início !//
- Implementação usa muitas vezes uma biblioteca disponível no sistema, mas isso é transparente
  - Pthreads API (Linux e Solaris)
  - Win32 API (Windows)
- *Threads* em Java podem ser criadas através de:
  - Classes com suporte para a **`Runnable`** *interface*

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

  - Classes derivadas de **`Thread`**

Criar Java *Thread*

- Java *thread* a partir de **`Runnable`**

```
public class RunHello implements Runnable {
 public void run() {
    System.out.println("Hello!");
 }
}
```

```
public class mainThread {
  public static void main(String args[]) {
     Thread th =new Thread(new RunHello());
     th.start();
  }
}
```

↳ Precisamos de fazer isto!

↳ podia ser feita

```
() → {
  ... → código
}
```

Criar Java *Thread*

- Java *thread* a partir de **`Thread`**

```
public class ThHello extends Thread {
 public void run() {
    System.out.println("Hello!");
 }
}
```

```
public class mainThread {
  public static void main(String args[]) {
     ThHello th = new ThHello();
     th.start();
  }
}
```

Ficamos com 2 threads em execução

se fosse th.run() iriam correr sequencialmente (o normal...)

Alguns métodos da classe **Thread**

- `static Thread currentThread();`
- `void interrupt();`
- `static boolean interrupted();`
- `boolean isInterrupted();`
- `void join();`
- `void join(long milis);`
- `static void sleep(long milis);`
- `static void yeld()`

Java *Threads* - estados

- **New**

  Thread foi criada mas start() ainda não foi chamado

- **Runnable** (Running / Ready)

  A chamada a start() aloca memória para a *thread* e chama run() (num novo fluxo de execução); Neste estado a *thread* pode ser escolhida pela JVM para executar no CPU; Java não tem estado Running

- **Blocked**

  Thread espera por adquirir um ***lock***

- **Waiting**

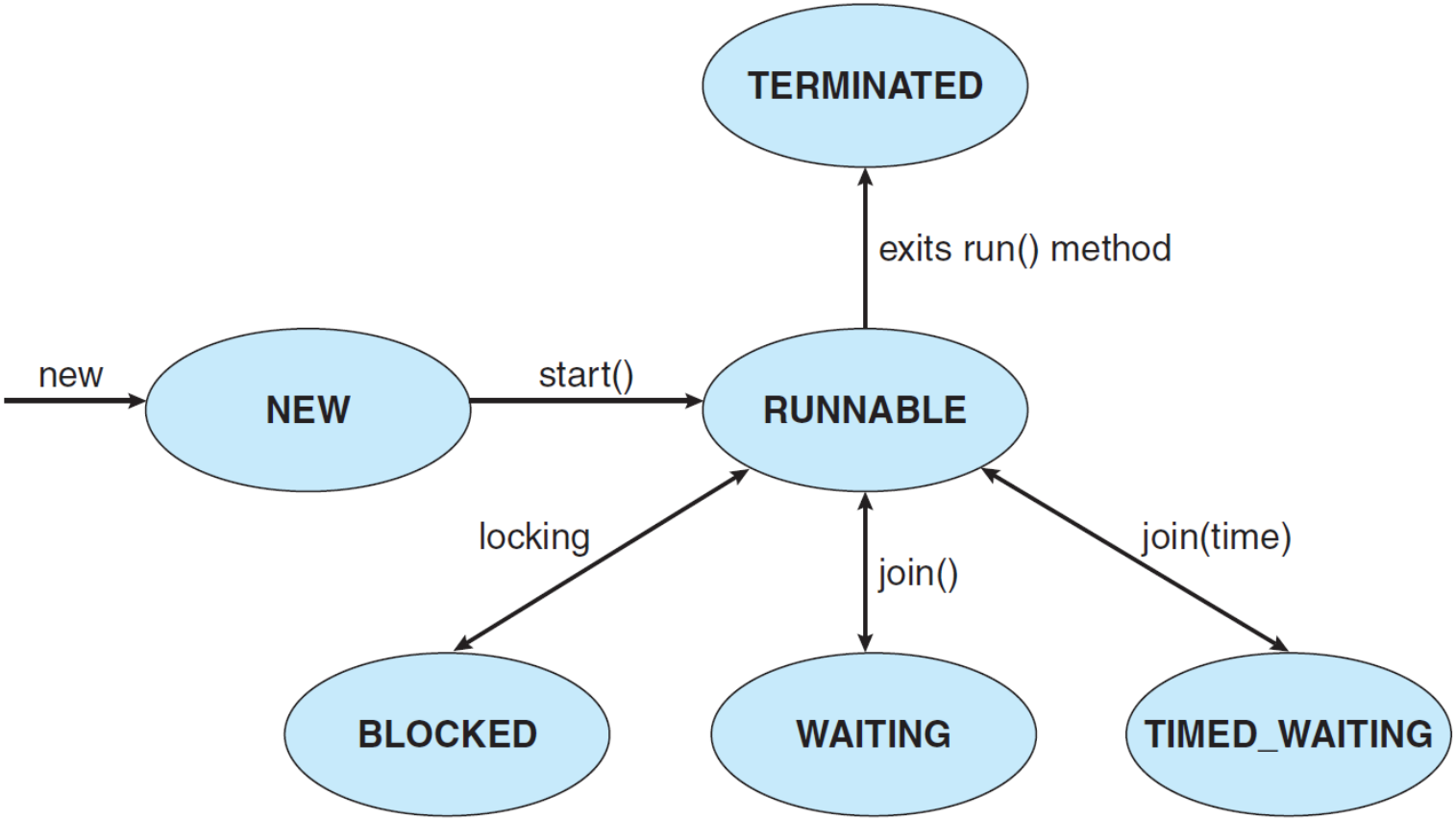
  ... Iremos falar + para a frente!

  Thread espera por ação de outra *thread* (ex: **join()**)

- **Timed Waiting**

  Idêntico a **Waiting** mas com um tempo máximo de espera

Java *Threads* - estados



Python *Threads*

- Módulo **`threading`**
- Criar *Threads*

```
import threading
import time

def th_func(id):
    time.sleep(3)
    print('Thread',id,'finishing')

for i in range(2):
    t= threading.Thread(target=th_func, args=(i,))
    t.start()
```

- Python GIL (*Global Interpreter Lock*) não permite que duas *threads* Python possam executar simultaneamente

Thread Pools

- Gerir um conjunto de *threads* previamente criadas, atribuindo trabalho à medida que for necessário => Pode melhorar a eficiencia!
- Vantagens:
  - Potencialmente mais rápido do que criar *threads* à medida que for necessário
  - Limita/controla o número de *threads* do sistema
- Em Java podem ser geridas através de *Executor interface*

Executor interface

- A interface Executor permite um nível de abstracção superior ao criar e manter várias *threads* em execução.
- Se **`r`** é um Runnable, então o código:

```
(new Thread(r)).start();
```

- Pode ser substituído, usando o executor **`e`** por:

```
e.execute(r);
```

↳ Queremos executar, mas a decisão de quando executar vem do "executor" (e)

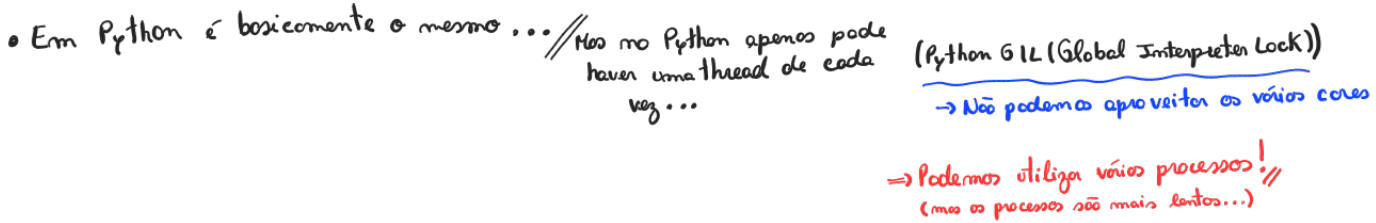
- A execução do método **`run()`** de **`r`** pode não ser imediata e depende da política associada ao executor

Static Executors

- O Java contém algumas implementações de *Thread Pools* prontas a ser usadas
- Estas *Thread Pools* podem ser criadas através de métodos `static` de `java.util.concurrent.Executors`
- Alguns exemplos:
  - `newSingleThreadExecutor()`
    - Executa uma tarefa de cada vez
  - `newFixedThreadPool(int nThreads)`
    - *Thread Pool* com um número fixo de *Threads*
  - `newCachedThreadPool()`
    - *Threads* sobrevivem durante algum tempo após tarefa terminar, mas são descartadas se não forem reutilizadas após esse tempo

Tópico prático: *Thread Pools* em Java

- **`Countdown` *threads*** usando
  - `newSingleThreadExecutor()`
    - Executa uma tarefa de cada vez
  - `newFixedThreadPool(int nThreads)`
    - *Thread Pool* com um número fixo de *Threads*
  - `newCachedThreadPool()`
    - *Threads* sobrevivem durante algum tempo após tarefa terminar, mas são descartadas se não forem reutilizadas após esse tempo
- Ficheiro **`ThreadPool2.java`**



Novas Questões

- Semântica de fork() e exec()
  - ↳ Com vários threads ...
- Cancelamento de *threads*
- Atendimento de sinais
- *Thread Pools*

fork() e exec()

- fork() duplica todas as *threads* ou apenas aquela em que foi executado → Duplica apenas a thread que executou!"
  - Comportamento normal é que, após o fork(), o processo filho só tem uma *thread*
  - Usar com cuidado pois podem surgir vários problemas
    - memória inconsistente, semáforos bloqueados, …
    - Depois de fork() apenas **funções async-safe** devem ser usadas
      - Ex: Não usar malloc() ou printf()
  - int pthread_atfork(void (*prepare)(void), void (*parent)(void), void (*child)(void));
  - Alguns sistemas UNIX têm 2 versões que permitem escolher o comportamento (fork() e forkall())
- Em geral, exec() substitui todo o processo incluindo todas as *threads*

Cancelamento de *threads*

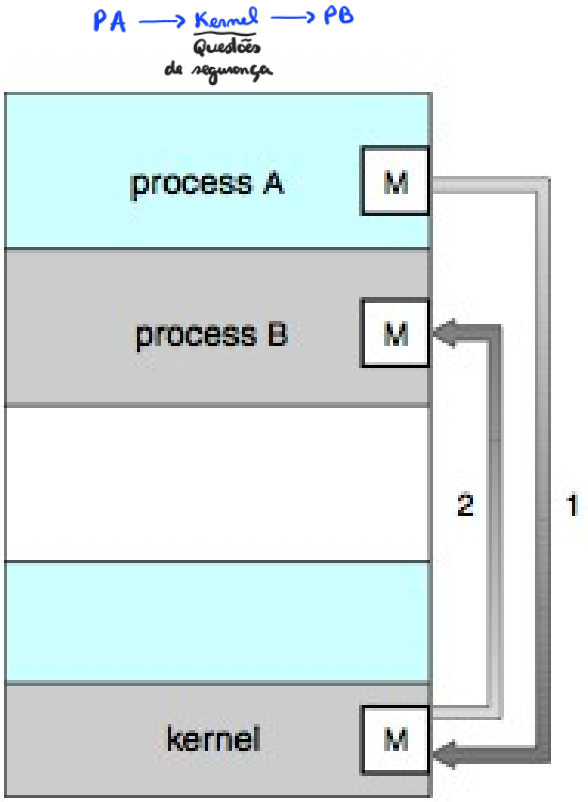
- Cancelar uma *thread* antes desta terminar por si
- 2 abordagens
  - Cancelamento assíncrono
    - *Thread* é terminada imediatamente
  - Cancelamento síncrono
    - *Thread* verifica periodicamente se deve terminar

Atendimento de sinais

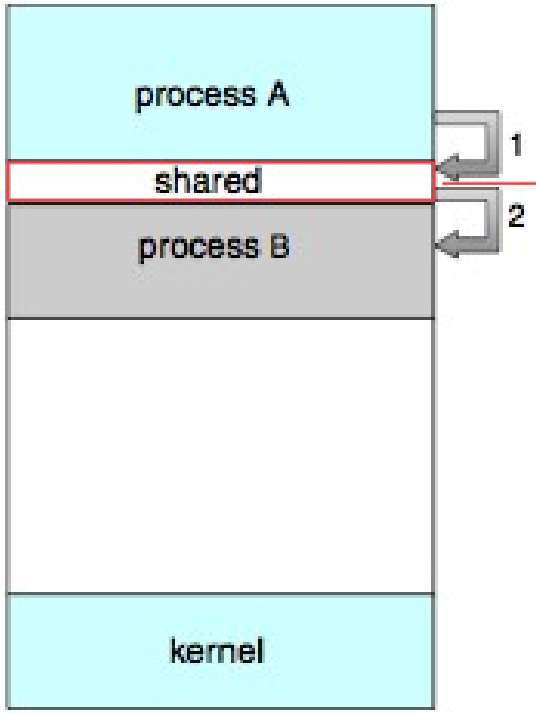
- Sinais são usados em UNIX para notificar processos de certos eventos
- Opções
  - Sinal é enviado apenas para a thread a que o sinal de aplica (ex: divisão por zero, etc)
  - Sinal enviado para todas as threads
  - Sinal enviado para subconjunto das threads
  - Thread específica recebe todos os sinais
    - pthread_sigmask() permite definir quais os sinais que cada thread pode receber. Assim a aplicação pode bloquear os sinais para todas as threads excepto uma (que fica com essa responsabilidade)

Comunicação entre processos/threads

Message Passing



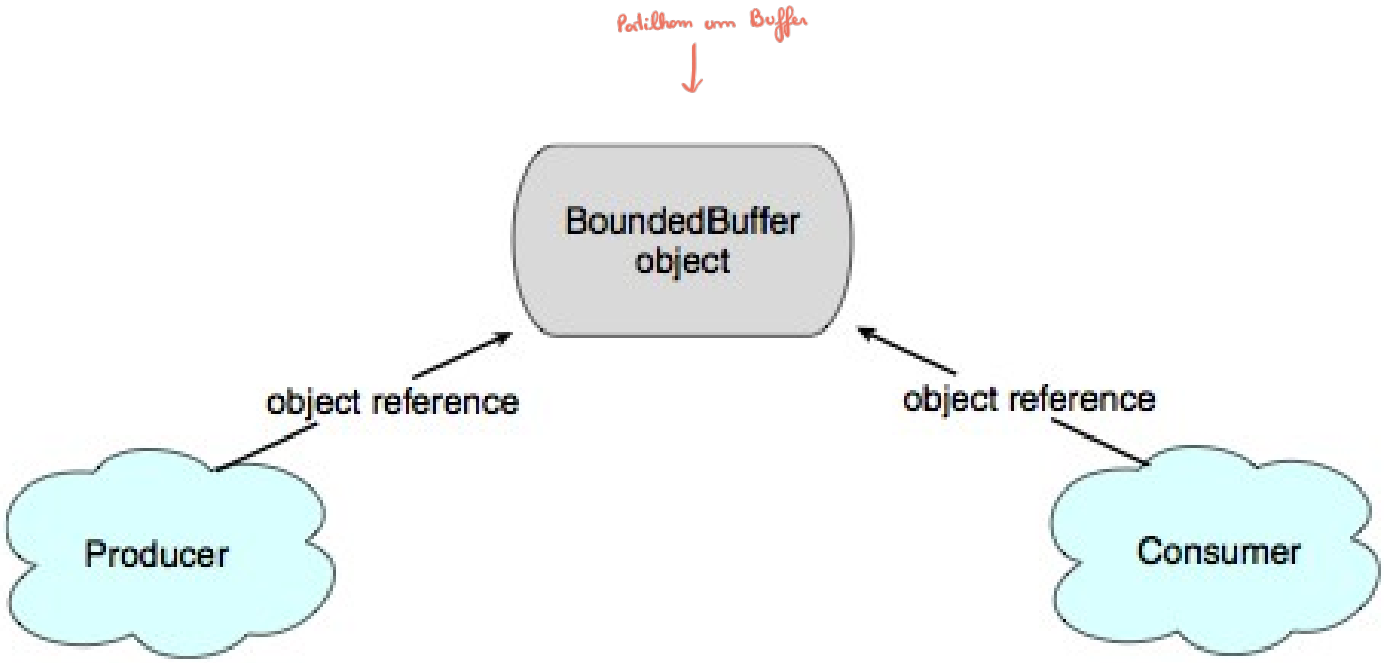
Shared Memory



Problema do produtor-consumidor

- Paradigma para processos cooperativos
  - Processo produtor produz informação
  - Informação é consumida pelo processo consumidor
- Um *buffer* partilhado armazena a informação em trânsito
  - *Buffer* sem limites (*unbounded buffer*) indica que não existe limite no tamanho do *buffer*
  - *Buffer* limitado considera que o *buffer* tem um tamanho fixo → Produtor não pode ser mais rápido que o consumidor

Problema do produtor-consumidor



Problema do produtor-consumidor
Solução Java com memória partilhada

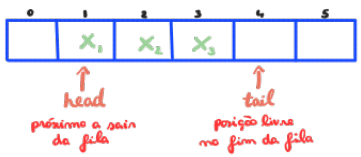
```
public interface Buffer
{
   // producers call this method
   public abstract void insert(Object item);

   // consumers call this method
   public abstract Object remove();
}
```

Problema do produtor-consumidor

Solução Java com memória partilhada

```
public class BoundedBuffer implements Buffer
{
    private static final int BUFFER_SIZE = 5;
    private int count; // number of items in the buffer
    private int in; // points to the next free position
    private int out; // points to the next full position
    private Object[] buffer;

    public BoundedBuffer() {
       // buffer is initially empty
       count = 0;
       in = 0;
       out = 0;

       buffer = new Object[BUFFER_SIZE];
    }

    // producers calls this method
    public void insert(Object item) {
       // Figure 3.16
    }

    // consumers calls this method
    public Object remove() {
       // Figure 3.17
    }
}
```

FIFO em array circular:



Problema do produtor-consumidor
Solução Java com memória partilhada

```
public void insert(Object item) {
   while (count == BUFFER_SIZE)
        ; // do nothing -- no free buffers

   // add an item to the buffer
   ++count;
   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
}
```

Problema do produtor-consumidor

Solução Java com memória partilhada

```
public Object remove() {
   Object item;

   while (count == 0)
        ; // do nothing -- nothing to consume

   // remove an item from the buffer
   --count;
   item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;

   return item;
}
```

Problema do produtor-consumidor
Solução Java com memória partilhada

- A solução apresentada anteriormente não é segura!
- Se um produtor e um consumidor executarem **`insert()`** e **`remove()`** ao mesmo tempo, **`count`** pode não ser actualizado correctamente
- Para manter a consistência do buffer é necessário que estes métodos sejam sempre executados por apenas 1 *thread* de cada vez!

▲ **Exclusão Mútua no acesso a estes métodos**

Iremos ver soluções ...

Race Condition: Dependendo da forma como o escalonador executar o código o resultado pode ser diferente