ieeta universidade de aveiro ua

Sistemas Operativos

Licenciatura Engenharia Informática
Licenciatura Engenharia Computacional

Ano letivo 2024/2025

Nuno Lau (nunolau@ua.pt)

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);
```

- 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()
- Cancelamento de *threads*
- Atendimento de sinais
- *Thread Pools*

fork() e exec()

- fork() duplica todas as *threads* ou apenas aquela em que foi executado
  - 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*

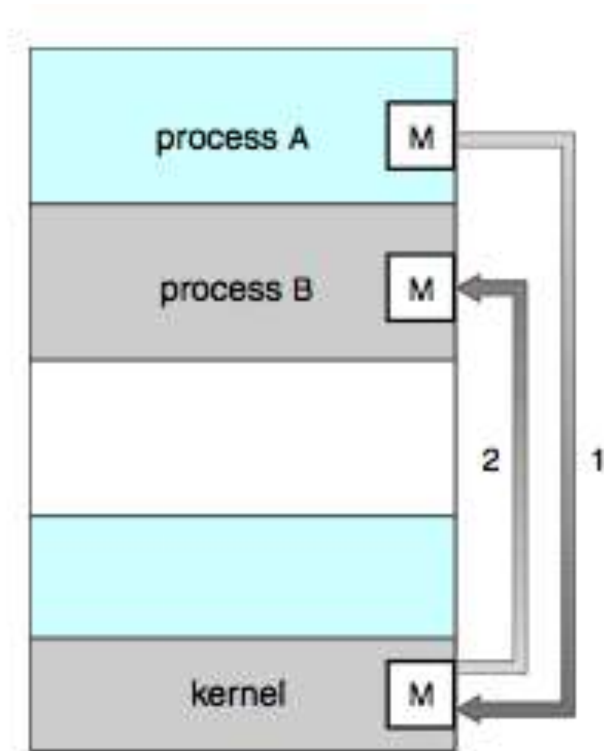
- Cancelar uma *thread* antes desta terminar por si
- 2 abordagens
  - Cancelamento assíncrono
    - *Thread* é terminada imediatamente
  - Cancelamento síncrono
    - *Thread* verifica periodicamente se deve terminar

Atendimento de sinais

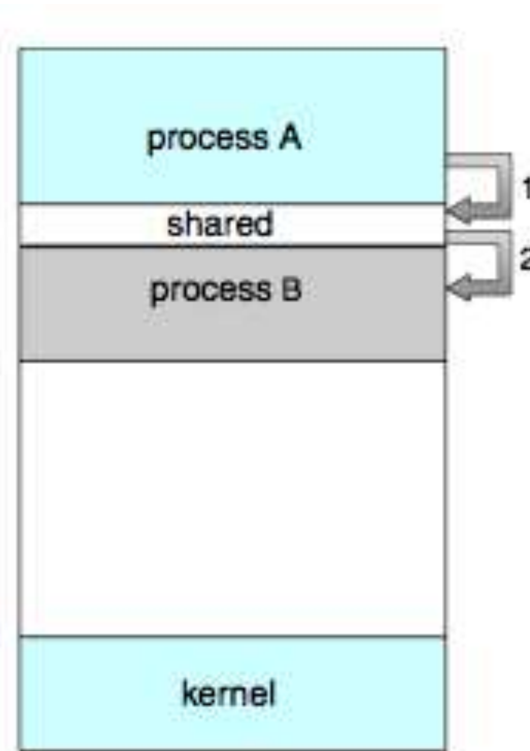
- Sinais são usados em UNIX para notificar processos de certos eventos
- Opções
  - Sinal é enviado apenas para a thread a que o sinal de aplica (ex: divisão por zero, etc)
  - Sinal enviado para todas as threads
  - Sinal enviado para subconjunto das threads
  - Thread específica recebe todos os sinais
    - 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



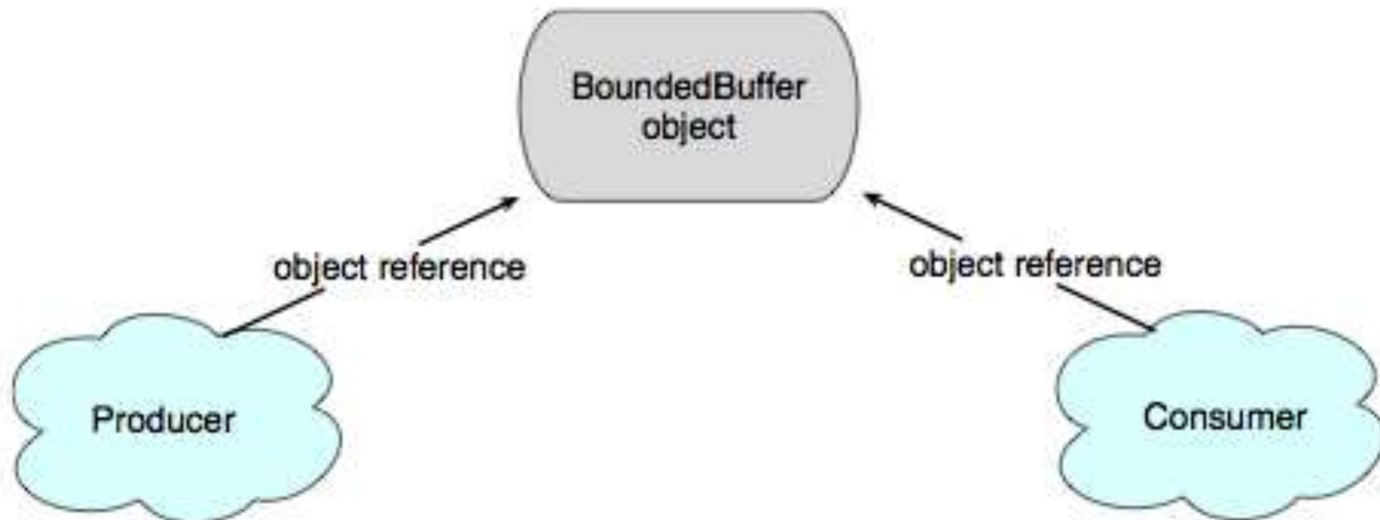
Shared Memory



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

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

ieeta universidade de aveiro ua

```
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
   }
}
```

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

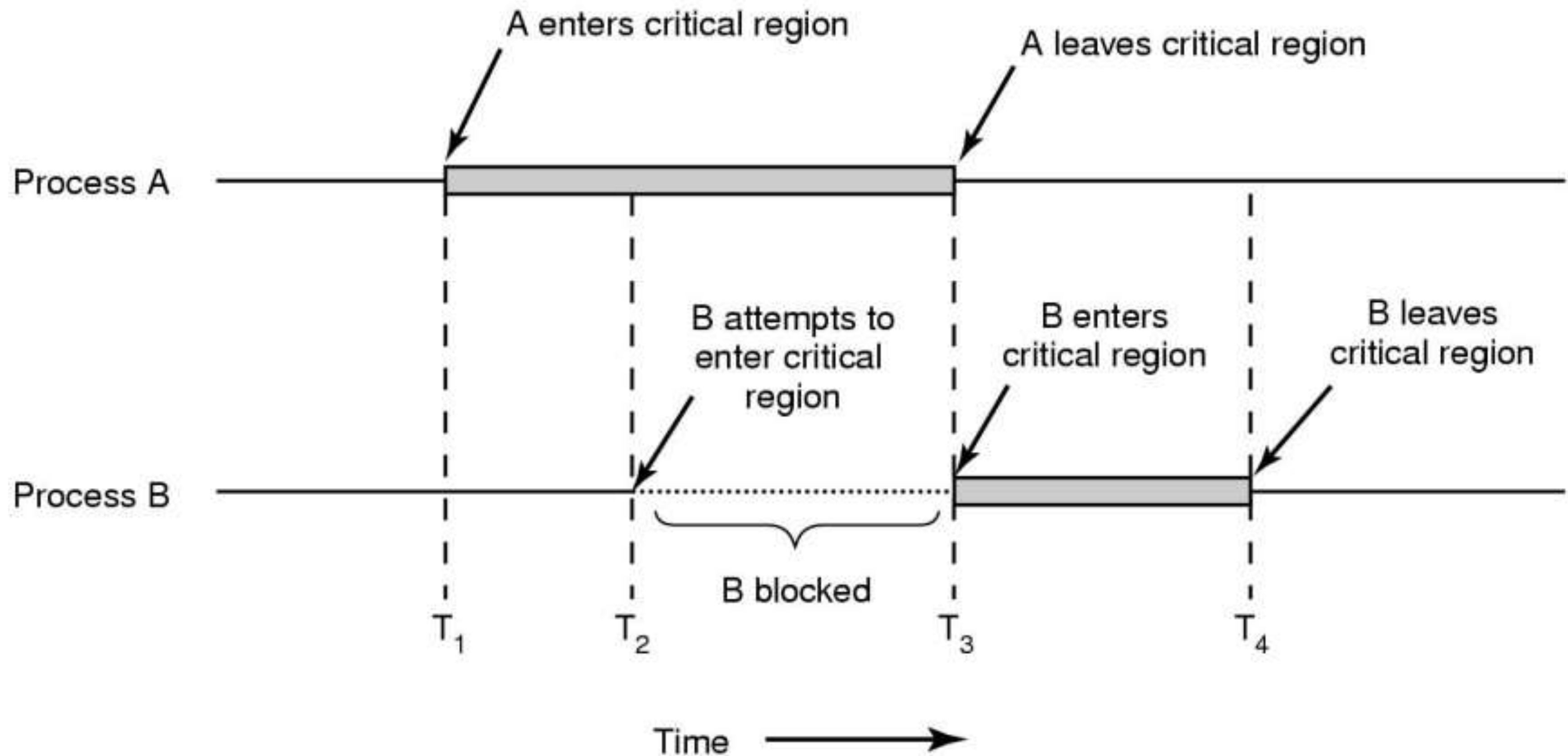
- A solução apresentada anteriormente não é segura!
- Se um produtor e um consumidor executarem `insert()` e `remove()` ao mesmo tempo, `count` pode não ser actualizado correctamente
- Para manter a consistência do buffer é necessário que estes métodos sejam sempre executados por apenas 1 *thread* de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

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

Definições

- **Condição de corrida**
  - Quando vários processos/*threads* acedem a dados partilhados e o resultado final depende de forma inesperada da ordem de execução
- **Região crítica**
  - **Zona de código que manipula dados partilhados** e que não pode ser executada concorrentemente por mais do que um processo/*thread*

Região Crítica



Condições para Região Crítica

ieeta

universidade de aveiro

ua

- **Exclusão Mútua**
  - Se um processo P_i está a executar na sua região crítica então nenhum dos outros processos pode estar em execução nas suas regiões críticas
- **Progresso**
  - Se nenhum processo está em execução em regiões críticas e pelo menos um processo pretende o acesso à região crítica então a seleção do processo que deverá ter acesso a esta região não pode ser adiada indefinidamente
- **Espera limitada**
  - Deve existir um limite ao número de vezes que é concedido o acesso a outros processos à região crítica, após um determinado processo ter pedido esse acesso e até que esse pedido seja satisfeito
- **Não há nenhum pressuposto sobre a velocidade ou número de CPUs**

Definições

- **Condição de corrida**
  - Quando vários processos/*threads* acedem a dados partilhados e o resultado final depende de forma inesperada da ordem de execução
- **Região crítica**
  - **Zona de código que manipula dados partilhados** e que não pode ser executada concorrentemente por mais do que um processo/*thread*

- **Região de entrada**
  - Código que realiza o pedido de acesso à região crítica
- **Região de saída**
  - Código executado após a saída da região crítica

Estrutura típica

```
while (true) {

    entry section

        critical section

    exit section

        remainder section

}
```

Soluções por software

- Variável partilhada de *lock*
  - Valor=0 se Região Crítica não está a ser usada
  - Valor=1 se Região Crítica está a ser usada
  - Não funciona se implementado apenas em software
- Alternância estrita
- Algoritmo de Dekker (1964)
- Algoritmo de Peterson (1981)

Alternância estrita

- Variável **turn** controla acesso à região crítica

Processo 0

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

Processo 1

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```