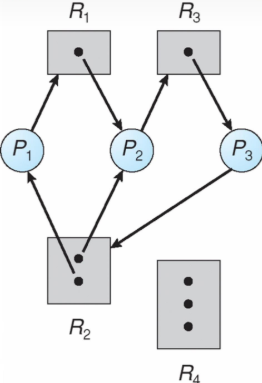
# Kahoot

## Sincronização de Processos

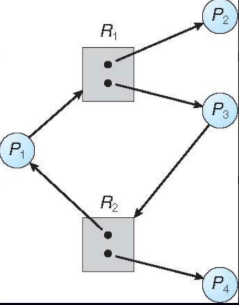
1. O acesso concorrencial a dados partilhados pode resultar em **Inconsistência de Dados**
2. Se um processo está na secção critica e outros não podem aceder, esta condição chama-se **exclusão mútua**;
3. **Semáforo** é um **mecanismo de sincronização**;
4. Um semáforo é uma variável inteira que **não pode ser menor que 0**;
5. Um semáforo tem o valor de 7, após 20 waits (-1) e 15 posts (+1) o seu valor é **2**. (Valor – (Waits – Posts)) – (7 – (20-15)) = 7-5 = **2.**

## Deadlocks

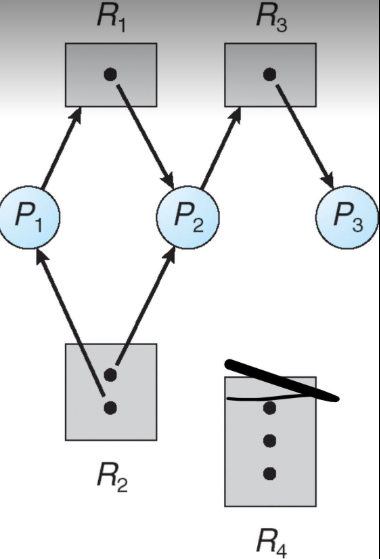
1. Um dos motivos que é uma condição necessária para a ocorrência de deadlocks é a **Exclusão mútua**;
2. **Existe** deadlock nesta imagem;



1. **Não existe** deadlock nesta imagem;

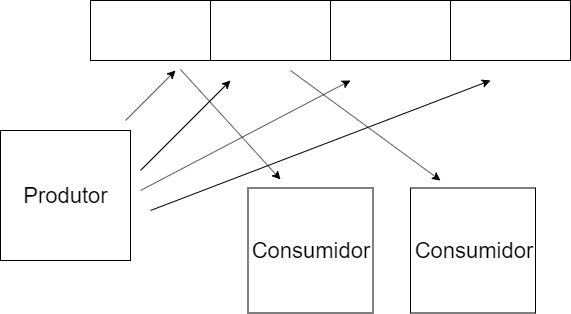


1. **Não existe** deadlock nesta imagem.



# Apontamentos

## Sincronização de Processos



*Quando estiver* ***cheio****,* ***o produtor*** *não pode colocar nada. Quando estiver* ***vazio****, o* ***consumidor*** *não pode ir buscar nada.*

**Empty** -> Nº de lugares vazios.

**Full** -> Nº de Items.

## **Escalonamento**

**Escalonador** -> **Algoritmo** que decide **quando e qual processo** deve passar a **execução**.

**Preemptivo** -> Algo que é **forçado.**

**Não-Preemptivo** -> **Livre escolha** ou **cooperativo**.



O tempo médio entre estes 3 processos é calculado através da simples operação de **média.** Ou seja – **Tempo Médio** = 0 (Processo A) + 24 (Processo B) + 27 (Processo C) / 3 (número de processos) = **17ms**.

O algoritmo **Priority Scheduling** consiste numa **execução** por **ordem de prioridade**.

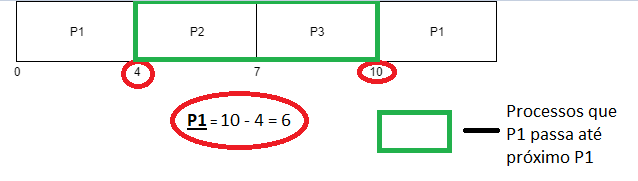
O problema neste algoritmo (Priority Scheduling) denomina-se de **fome**.

**Fome** -> **Processos** de **baixa** prioridade podem **nunca ser executados**.

A solução para esse problema (fome) denomina-se de **idades**.

**Idades** -> **Aumentar a prioridade** à medida que o **tempo** passa.

O algoritmo **Round-Robin** é onde ocorre mais falhas no miniteste (sairá).

Esta imagem serve para perceber o **slide 14 de SO-7 Escalonamento (Round-Robin)**.

Sendo que os valores para os processos são:

* P1 -> 0
* P2 -> 4ms
* P3 -> 7ms
* P1(2º) -> 10ms

Os valores assumidos são os que aparecem anteriormente aos processos (conforme imagem).

**P1** só **esperou** por P2 **e P3**, até voltar a ser executado.

## **Deadlocks**

Condições para deadlocks:

1. **Exclusão mútua**;
2. **Bloquear e esperar;**
3. **Sem preepção;**
4. **Espera circular**.

Métodos para prevenir deadlocks:

1. Assegurar que o sistema nunca entra num deadlock;
2. Permitir ao sistema recuperar de um deadlock;
3. Ignorar o problema e responsabilizar o programador.

Evitar deadlocks:

* Cada processo declara o número máximo de recursos que vai necessitar, de cada tipo;
* O algoritmo examina os recursos de modo a evitar estados com ciclos.

**Estado Seguro** -> **Não** acontece deadlocks;

**Estado Inseguro** -> **Pode** ocorrer deadlocks.

Recuperação deadlocks:

* Terminação de processos;
* Preempção de recursos.

Sumário (transcrição slide 15 – Sumário):

* Deadlocks ocorrem quando dois ou mais processos bloqueiam à espera de recursos libertados por outros processos, também eles bloqueados;
* Existem 3 formas de líder com deadlocks;
* Ignorar o problema e responsabilizar o programador é o mais comum.

## Threads

**Thread** -> É uma **linha de execução** dentro de um **processo**.

É mais rápido **várias Threads** do que vários processos**.**

**Thread.start()** – Inicia a thread.

Numa **thread pool** as tarefas vão utilizando **Threads criadas inicialmente**.

# Sumário Slides (Bullet Style)

## Sincronização

* Em Mutex (MUTual EXclusion), os processos devem adquirir a “fechadura” (lock) antes de entrar na secção critica e devem libertar a “fechadura” após sair da secção critica;
* Semáforos são uma **variável inteira** n, acedida através de **wait()** (-1 ao valor n do semáforo) e **post()** (+1 ao valor n do semáforo);
* **Mutex** (**Exclusão Mútua**) garante **acesso único** a **secções críticas**;
* Na **prática** usa-se essencialmente mutexes e semáforos;
* Os Sistemas Operativos fornecem estes mecanismos.

## Problemas Clássicos

* No problema do **Produtor**-**Consumidor** existe um **array de tamanho fixo**, em que o **produtor** **coloca valores no array**, sendo que o **consumidor** **remove os valores do array**. O **produtor não pode** colocar valores se o **array estiver cheio**, e o **consumidor** **não pode remover** valores se o array estiver vazio (referido anteriormente);
* No problema dos **Leitores-Escritores**, uma base de dados entre **vários processos concorrentes**, em que os **leitores** apenas **leem dados** e os **escritores** **podem ler** e **escrever** dados. Neste problema, **dois processos** **leitores** podem ler ao mesmo tempo, mas **um processo escritor** deve ter **acesso exclusivo** à Base de Dados. Os **leitores** só devem **esperar** se um **escritor** já estiver **obtido permissão para escrever**;
* No problema do **jantar dos filósofos**, quando o **filosofo** tem fome, deve usar os **dois garfos**, quando **terminar** de comer, deve **pousar os dois garfos**, sendo que o **filosofo** não pode usar um garfo que já esteja a ser usado. Como solução podemos **permitir** que apenas **4 filósofos** se sentem à mesa, **permitir** que **um filósofo** pegue nos garfos apenas se os dois estiverem disponíveis ou os **filósofos pares** pegam **primeiro no garfo esquerdo** e só **depois no direito**, e os **ímpares** fazem ao contrário.
* Sumarizando, os **problemas clássicos** são utilizados para **testar novos mecanismos de sincronização**.

## Escalonamento de processos

* **Escalonamento de processos** é **central** ao **sistema operativo;**
* *Porquê escalonamento de processos? –* **Utilização máxima** de CPU e **Ciclos de CPU/IO** (**Conjunto** de **instruções CPU** seguido de um **conjunto de instruções de I/O**);
* Para **maximizar** o uso do **CPU**, **um processo** pode ser **trocado** enquanto espera por **I/O**;
* **Escalonador de CPU –** Decisões sobre **escalonamento do CPU** podem ocorrer quando um processo muda de **“running”** para **“waiting”** (exemplo: *wait()*), muda de **“running”** para **“ready”** (exemplo: *interrupt*), muda de **“waiting”** para **“ready”** (exemplo: *fim de I/O*), **termina** (exemplo: *exit(0)*) – Sendo que **“running” -> “waiting”** e quando um processo **termina** são **não-preemptivos ou cooperativos**, e os restantes são **preemptivos**;
* **Não-preemptivo ou cooperativo** – Os **processos libertam** o **processador ao terminarem** ou **passam para espera** (wait);
* **Preemptivo –** Os **processos libertam o processador** por **meios externos** (interrupts,time-slots,etc.);
* **Existem 5 critérios de escalonamento**:
  + **Utilização de CPU** – Manter o **CPU ocupado**;
  + **Taxa de transferência** – **Número de processos** que **completam execução** por **unidade de tempo**;
  + **Tempo de execução** – **Tempo** que **um processo** **demora** a **executar;**
  + **Tempo de espera - Tempo** que **um processo aguarda** na **fila “ready”;**
  + **Tempo de resposta** – **Tempo** desde que **um processo** foi **iniciado** até **começar a executar**.
* Algoritmo **First-Come, First-Served** – **Execução** por **ordem de chegada;**
* Algoritmo **Shortest-Job** **First** – **Execução** por **ordem de** **CPU burst** (existem algoritmos de estimativa);
* Algoritmo **Priority Scheduling** – **Execução** por **ordem de prioridade.** Neste algoritmo, pode ocorrer um **problema** denominado de **fome** que indica que **processos** com **prioridade baixa** podem **nunca ser executados**. A **solução** para o **problema de fome** denomina-se de **idades**, que indica, o **aumento de prioridade** à medida que o **tempo passa**;
* Algoritmo **Round-Robin** - Algoritmo **preemptivo** para **sistemas com partilha de tempo**;
* Se um **time quantum** for **maior que** o **process time** (exemplo: quantum=12ms e process time = 10ms) **não existem contente switches**. Se o **time quantum** for **menor que** o **process time**, deverá se fazer a conta de quantos **time quantum** cabem dentro do **process time**, o que indicará quantos **contexto switches** existem;
* Algoritmo **Multilevel Queue Scheduling** – Serve para **processos em diferentes grupos** (exemplo: *processos interactivos vs background*);
* Algoritmo **Multilevel Feedback Queue Scheduling** – Semelhante ao anterior, sendo que este **permite que processos transitem** **entre filas de espera**;
* Algoritmo **Sistemas em Tempo Real** – Serve para **minimizar latência**, sendo que **os processos** devem cumprir requisitos **em termos de tempos**;
* **Em Linux CFS –** Completely Fair Scheduler, é **preemptivo** e baseado em **prioridades**;
* **Vruntime** = **Tempo de execução** até ao momento;
* **Em Windows** – **Também preemptivo e baseado em prioridades**, existem **6 classes de prioridade** e **7 níveis de prioridade relativa**;
* **Sumário:**
  + **Escalonamento** é a escolha do **próximo processo** a **ser executado**;
  + **First-come** **First-served** é o mais simples;
  + **Restantes** algoritmos tentam **melhorar**;
  + **Round-Robin** é **mais apropriado** para **sistemas de partilha de tempo**;
  + **Time-Quantum** pode dar origem a **muitos contexto-switches**.

## Deadlocks

* **Num deadlock** os **processos não terminam** e os **recursos** ficam **bloqueados**, **não permitindo** que **outros processos iniciem;**
* Existem **4 condições para deadlocks**:
  + **Exclusão mútua –** Pelomenos **um recurso** deve **ser bloqueado** por **um único processo**;
  + **Bloquear e esperar** – **Um processo** deve manter **pelo menos um recurso** e **esperar** por **adquirir outros recursos**;
  + **Sem preempção –** **Os recursos** só **podem ser libertos** quando o **processo assim o permitir**;
  + **Espera circular** – **Deve existir um conjunto** {P0,P1, … ,Pn} (P de processo), de **processo à espera**, em que **P0** **espera por** **P1**, **P1 por P2**, etc.
* **Sem ciclos não há deadlocks;**
* **Ciclos não implicam deadlocks**;
* Existem **3 métodos para prevenir deadlocks**:
  + **Assegurar** que o **sistema nunca entra num deadlock;**
  + **Permitir** ao **sistema** recuperar de **um deadlock;**
  + **Ignorar** o **problema** e responsabilizar o programador.
* Para **evitar deadlocks**:
  + **Cada processo** **declara** o **número máximo de recursos** que vai necessitar, de cada tipo;
  + **O algoritmo** **examina os recursos** de modo a **evitar estados com ciclos.**
* **Estado seguro** – **O** sistema com **processos** {P1, … , Pn} está num **estado seguro** se os **recursos que Pi** (Processo i) **necessita não estão disponíveis**, pode esperar por **processos Pj**, quando **Pj termina, Pi consegue obter recursos,** quando **Pi termina, Pi+1** consegue **obter recursos de Pi**. (Para j<i);
* **Estado seguro** – **Sem deadlocks**;
* **Estado inseguro** – **Possibilidade de deadlocks**;
* **Recuperação de deadlocks**:
  + **Terminação de processos** – **Abortar** todos os **processos bloqueados** ou **abortar** **um processo** de **cada vez** até **eliminar o ciclo de deadlocks;**
  + **Preempção de recursos** – **Selecionar processo** – minimizar curto; **Rollback** – retornar a **um estado seguro**; **Fome –** o **mesmo processo** pode **ser escolhido continuamente**.
* **Sumário**:
  + **Deadlocks** ocorrem **quando dois ou mais processos bloqueiam** à **espera de recursos libertados por outros processos**, também eles **bloqueados**;
  + Existem **3 formas de lidar com deadlocks**;
  + **Ignorar** o problema e responsabilizar o programador é o mais comum.