

Bacharelado em Ciência da Computação *Sistemas Operacionais*

Aula começa as
10h30min


Prof. Me. Filipo Novo Mór
filipo.mor at unilasalle dot edu dot br



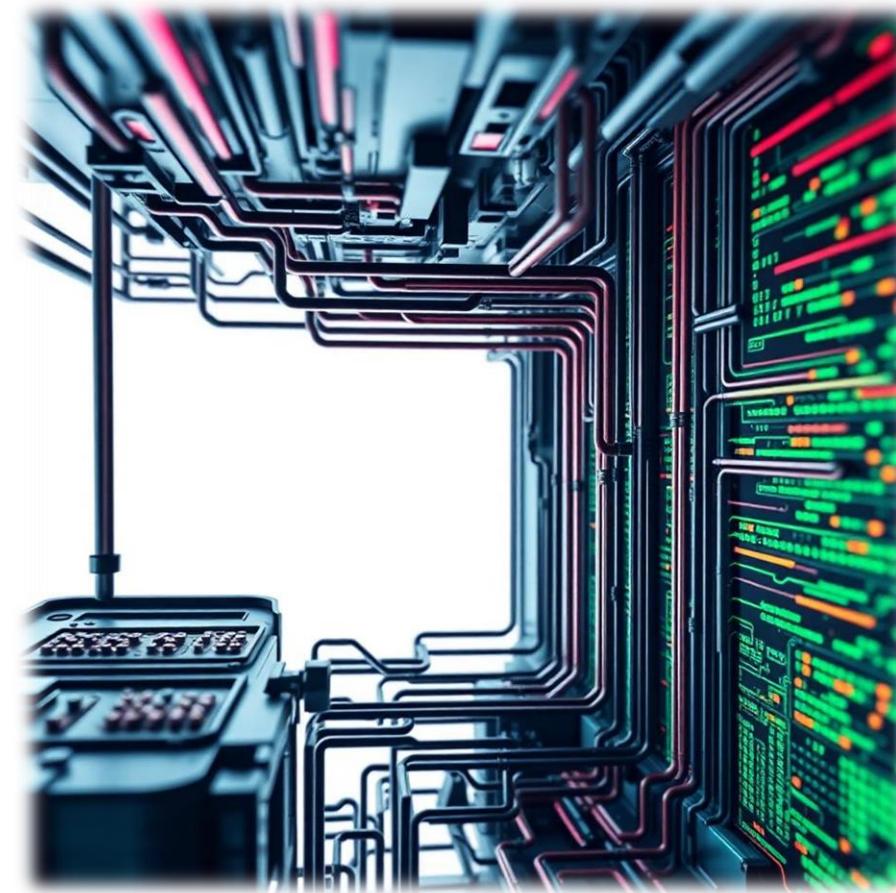
Conhecendo vocês!

- Nome, curso e semestre
- Experiência profissional (se tiver)
- Possui conhecimentos prévios em programação? Quais?
- Quais as expectativas para esta disciplina?

<https://padlet.com/filipomor/sisop2025II>



Gerenciamento de Processos



O que é um Processo?

- Uma instância de um programa em execução
- Não é um processo quando um programa está apenas localizado em um disco ou na memória, como um dado.
- Se dois usuários estiverem executando o mesmo programa simultaneamente, existirão dois processos separados



Operações sobre um Processo

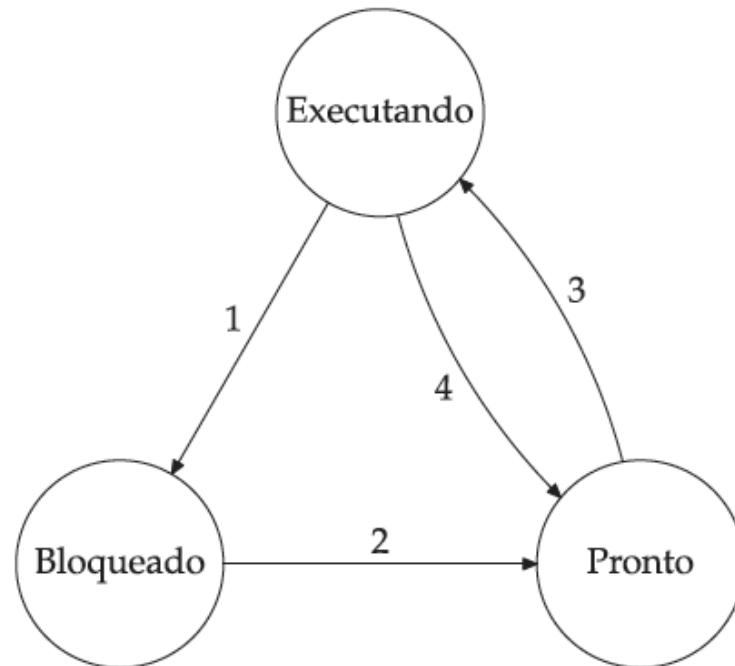
- Criar/terminar processo
- Mudar programa
- Definir/obter parâmetros de processo
- Bloquear processo
- Despertar processo
- Trocar processo
- Escalonar processo



Estado do Processo

- **Executando:** A CPU está correntemente executando um código que faz parte do processo
- **Pronto:** Processos no estado de Pronto não estão esperando por evento algum, mas esperam por sua vez na CPU.
- **Bloqueado:** Identificamos processos à espera de algum evento (com frequência operações de E/S) como bloqueado

Típica Máquina de Estados do Processo



Estados do Processo no Linux

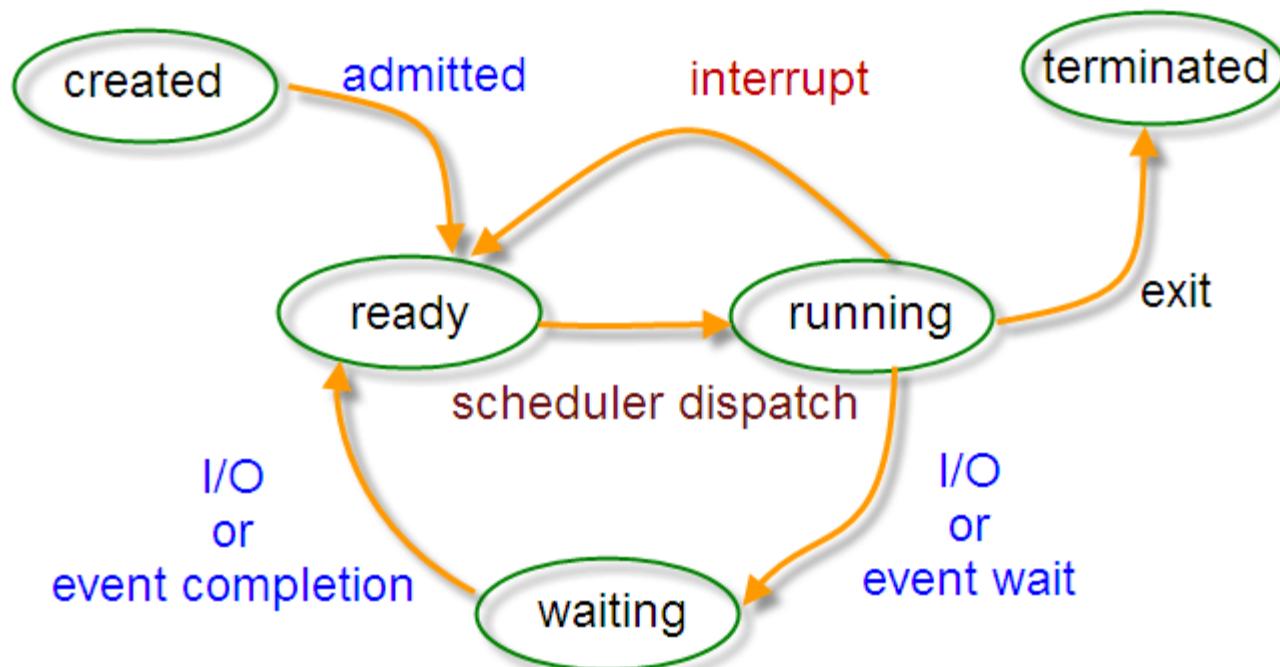
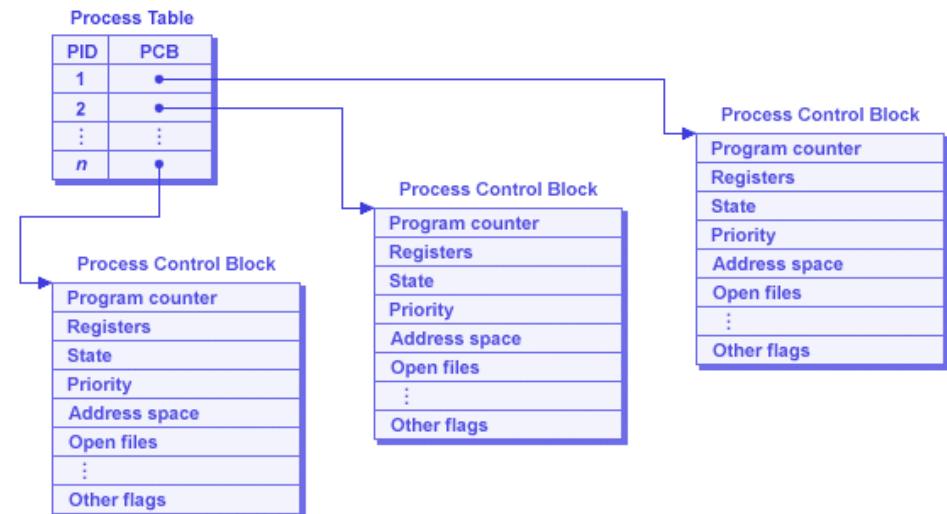


Tabela de Processos

- Uma entrada para cada processo
- Alguns itens típicos:
 - registradores salvos;
 - estado do processo;
 - número ID do processo;
 - número ID do proprietário;
 - número ID do grupo;
 - prioridade;
 - utilização e mapeamento da memória
 - status dos arquivos abertos;
 - tempo de execução cumulativo



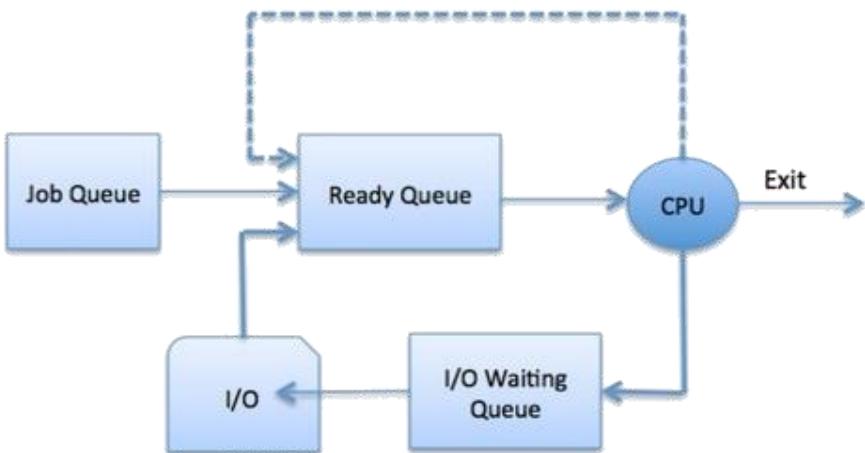
Threads

- Elementos do processo pai:
 - Memória
 - Arquivos abertos
 - Espaço do nome
 - Ambiente
- Processo do filho:
 - Compartilha
 - Herda uma cópia
 - Recebe uma substituição

Característica	Processo	Thread
Definição	Instância de um programa em execução.	Unidade de execução dentro de um processo.
Espaço de Memória	Espaço de memória isolado.	Compartilha o espaço de memória do processo pai.
Recursos	Recursos próprios (arquivos, sockets, etc.).	Compartilha recursos com outras threads do mesmo processo.
Criação	Mais pesado, requer duplicação de recursos.	Mais leve, criação mais rápida.
Troca de Contexto	Mais lenta, envolve troca de memória.	Mais rápida, troca apenas de contexto de execução.
Comunicação	Requer mecanismos IPC (pipes, sockets, etc.).	Comunicação direta através da memória compartilhada.
Isolamento	Maior isolamento, falhas afetam menos outros processos.	Falhas podem afetar todas as threads do processo.
Exemplo	Navegador web (cada aba pode ser um processo).	Editor de texto (auto-salvar, verificação ortográfica em threads).

Escalonamento

- Seleciona que processo é o próximo a obter a CPU
- Mantém uma estrutura de dados de processos prontos para executar



Algoritmos de Escalonamento de Processos

Gerenciando a Execução de Processos na CPU

- O sistema operacional Linux utiliza diversos algoritmos para escalar processos, com o objetivo de otimizar o desempenho do sistema.
- A escolha do algoritmo e seus parâmetros são cruciais para garantir justiça, eficiência e capacidade de resposta.
- Veremos os principais algoritmos de escalonamento no Linux: CFS, FIFO, RR e Deadline.

Algoritmos de Escalonamento de Processos

CFS (Completely Fair Scheduler):

- Algoritmo padrão para processos normais (não tempo real).
- **Objetivo:** Alocar tempo de CPU justo para cada processo.
- **Mecanismo:** Utiliza uma árvore vermelha-preta para priorizar processos que menos executaram.
- **Parâmetros:** *nice value* (prioridade do usuário), *sched_latency_ns*, *sched_min_granularity_ns*.

Escalonadores de Tempo Real:

- Para aplicações que exigem respostas em tempo real (ex: multimídia, controle de sistemas).
- Políticas:
 - **FIFO (First-In, First-Out):** Execução em ordem de chegada, sem preempção (exceto por maior prioridade).
 - **RR (Round Robin):** Cada processo recebe um *quantum* de tempo.

Algoritmos de Escalonamento de Processos

Deadline Scheduler (SCHED_DEADLINE):

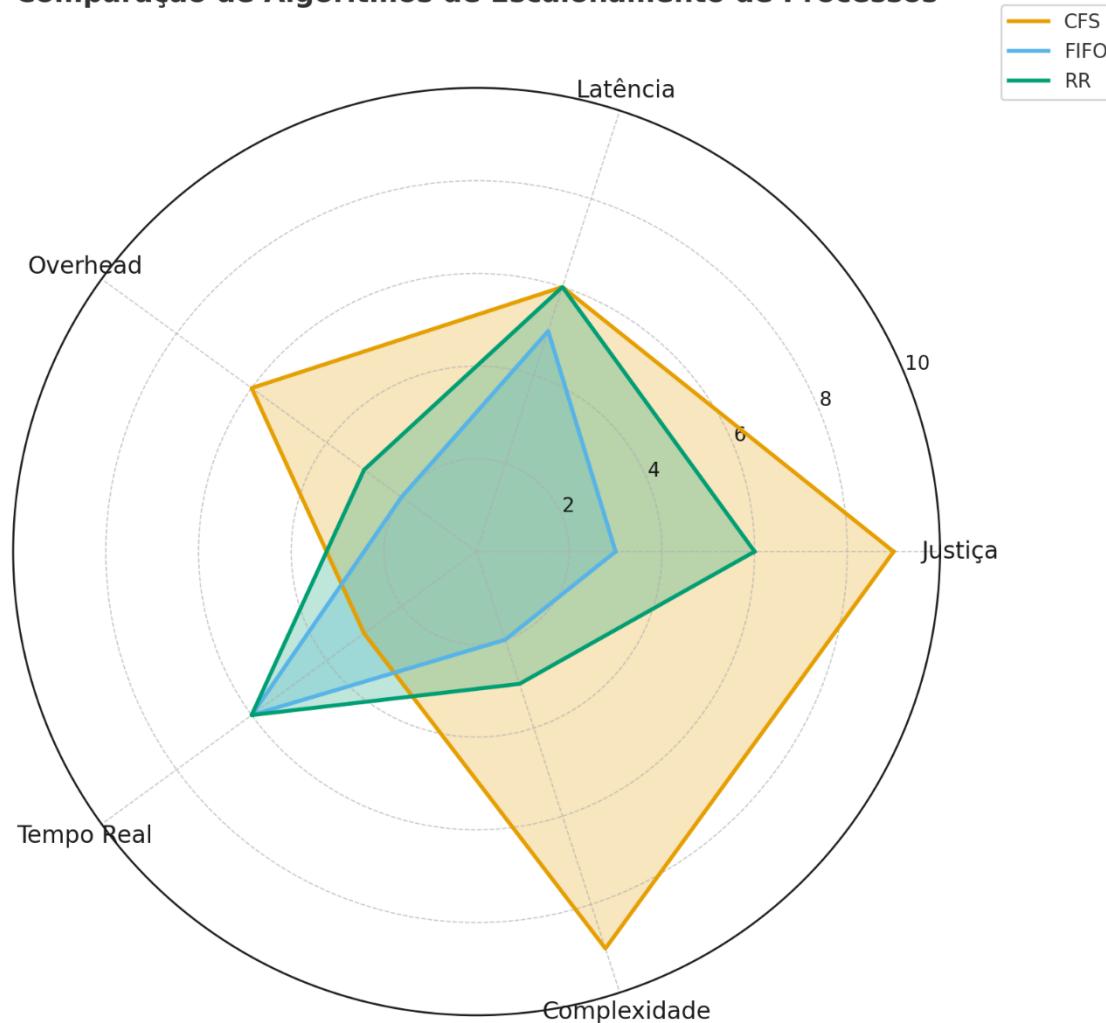
- Permite que as aplicações especifiquem um prazo (deadline) para a conclusão de suas tarefas.
- O escalonador tenta garantir que cada tarefa termine antes de seu prazo.
- Utilizado para aplicações de tempo real mais exigentes.

Aspectos adicionais:

- **Preempção:** Linux é preemptivo, permitindo interromper processos de menor prioridade.
- **Prioridades:** Influenciam a ordem de escalonamento (*nice value* no CFS, prioridades explícitas em tempo real).
- **Grupos de Controle (cgroups):** Permitem alocar recursos para grupos de processos.

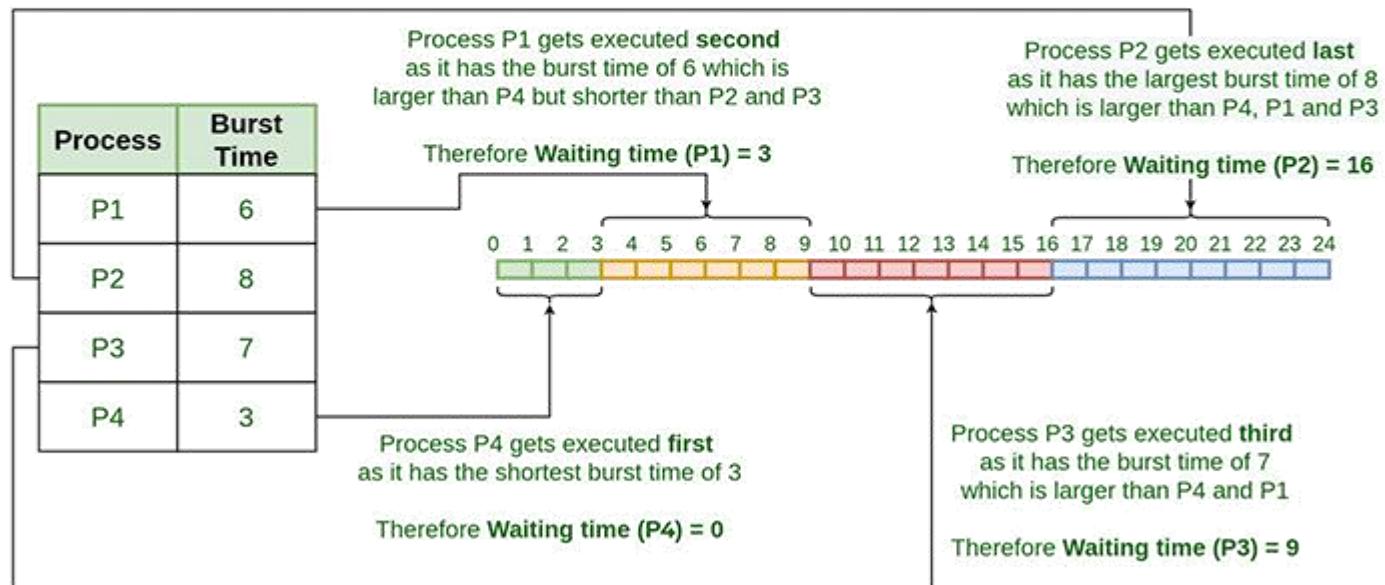
Algoritmos de Escalonamento de Processos

Comparação de Algoritmos de Escalonamento de Processos



Job Mais Curto Primeiro

Shortest Job First (SJF) Scheduling Algorithm



- Processo de job em ordem crescente de tempo de execução
- Otimização: tempo de retorno médio é minimizado
- Exige que conheçamos o tempo que o job levará para ser executado
- É uma abordagem “gulosa”

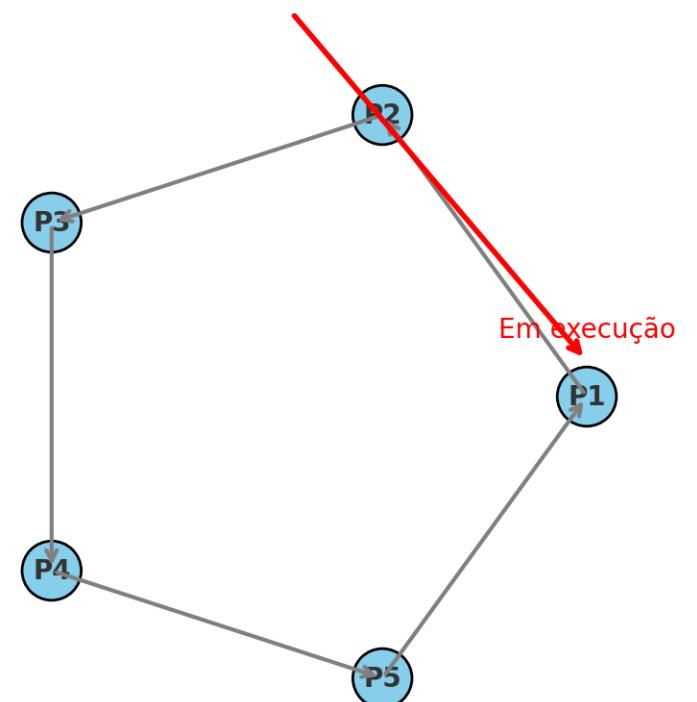
Circular (Round Robin)

- Um dos algoritmos de escalonamento mais simples e comuns em sistemas de tempo compartilhado.
- Objetivo: Garantir que todos os processos em estado de pronto recebam uma fatia justa de tempo da CPU.
- Ideal para sistemas interativos, proporcionando boa capacidade de resposta.
- Técnica aplicável ao compartilhamento de tempo
- No escalonamento circular, os processos se alternam igualmente.
- Uma forma de defini-lo é que o próximo processo selecionado para ser executado é o que esperou por mais tempo.
- Os processos que passam por uma preempção são adicionados ao final da fila, assim como os novos processos
- Processos prontos são mantidos em uma fila circular

Circular (Round Robin)

- **Fila de Prontos:** Processos prontos são mantidos em uma fila circular.
- **Quantum de Tempo (*Time Slice*):** Cada processo recebe uma pequena unidade de tempo da CPU.
- **Execução:** O escalonador seleciona o primeiro processo da fila e permite que ele execute por um quantum.
- **Preempção:** Se o processo não terminar no quantum, é preemptado e movido para o final da fila.
- **Próximo Processo:** O escalonador seleciona o próximo processo da fila, repetindo o ciclo.

Round Robin: Fila Circular de Processos



Circular (Round Robin)

Vantagens:

- Simples e fácil de implementar.
- Justo, garantindo tempo de CPU para todos.
- Boa capacidade de resposta para sistemas interativos.

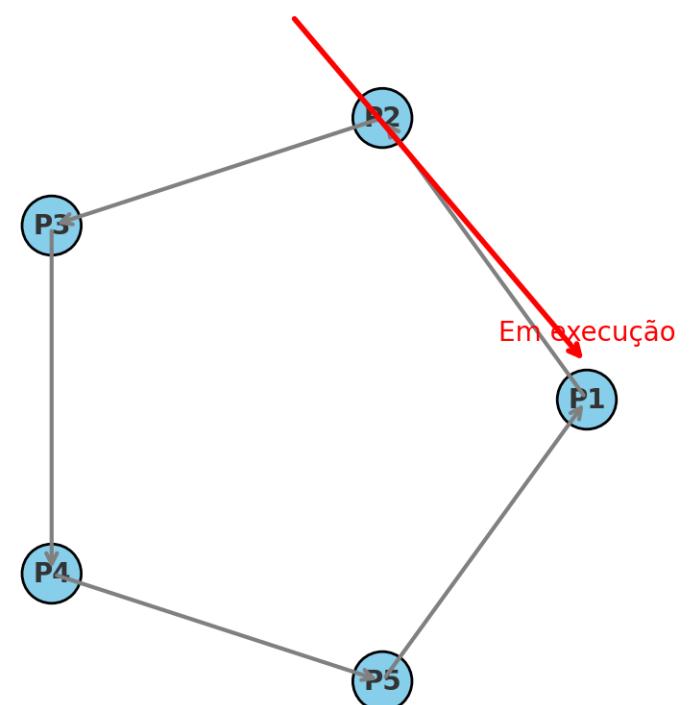
Desvantagens:

- Overhead de troca de contexto.
- Pode não ser o mais eficiente em *throughput*.
- Não prioriza processos.

Considerações:

- A escolha do quantum de tempo é crucial para o desempenho.
- Quantum muito pequeno aumenta o *overhead*.
- Quantum muito grande se aproxima do comportamento FIFO.

Round Robin: Fila Circular de Processos



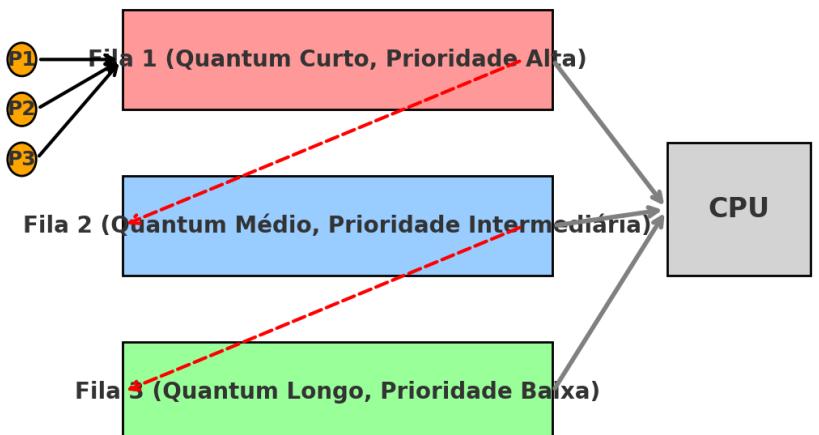
Escalonamento por Prioridades

- Cada processo tem uma prioridade
- O processo com maior prioridade é selecionado
- Prioridades dinâmicas e estáticas
- Um método comum é elevar a prioridade de processos bloqueados

Filas de Retorno Multinível – (MLFQ)

- Uma implementação particularmente significante de prioridades dinâmicas
- Uma fila por nível de prioridade
- Os processos não migram para prioridades mais altas no curso normal de escalonamento. Em vez disso, podem receber um aumento na prioridade por uma série de razões

Escalonamento por Filas de Retorno Multinível (MLFQ)



Processos começam na fila de maior prioridade.
Se não terminam no quantum, descem para a fila seguinte.
Fila mais baixa tem quantum maior (menos prioridade).

Filas de Retorno Multinível – (MLFQ)

No **MLFQ (Multilevel Feedback Queue)**, as prioridades dos processos mudam dinamicamente conforme o comportamento deles. Dois exemplos típicos:

1. Processo que consome muito tempo de CPU

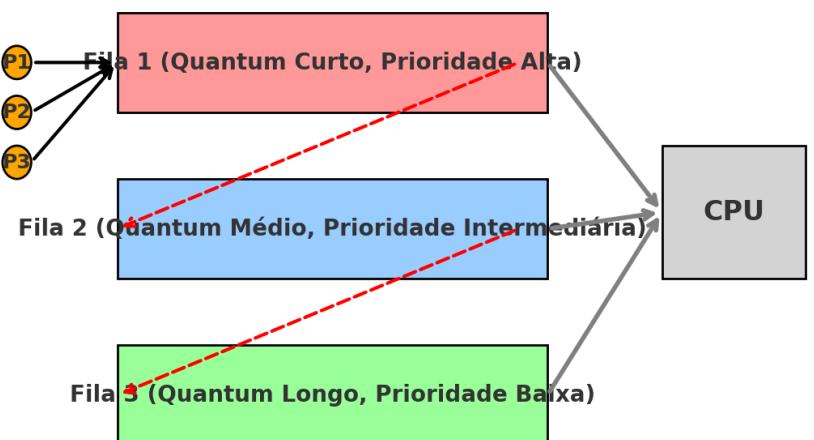
1. Se um processo **não termina dentro do quantum** da sua fila, ele é **rebaixado** para uma fila de prioridade mais baixa.
2. Isso evita que processos CPU-bound monopolizem a CPU.

2. Processo que passa a maior parte do tempo em espera por I/O

1. Quando um processo é bloqueado rapidamente (por exemplo, esperando entrada de usuário ou operação de disco) e depois volta para pronto, ele pode ser **promovido para uma fila de prioridade mais alta**.
2. Isso garante boa responsividade para processos interativos.

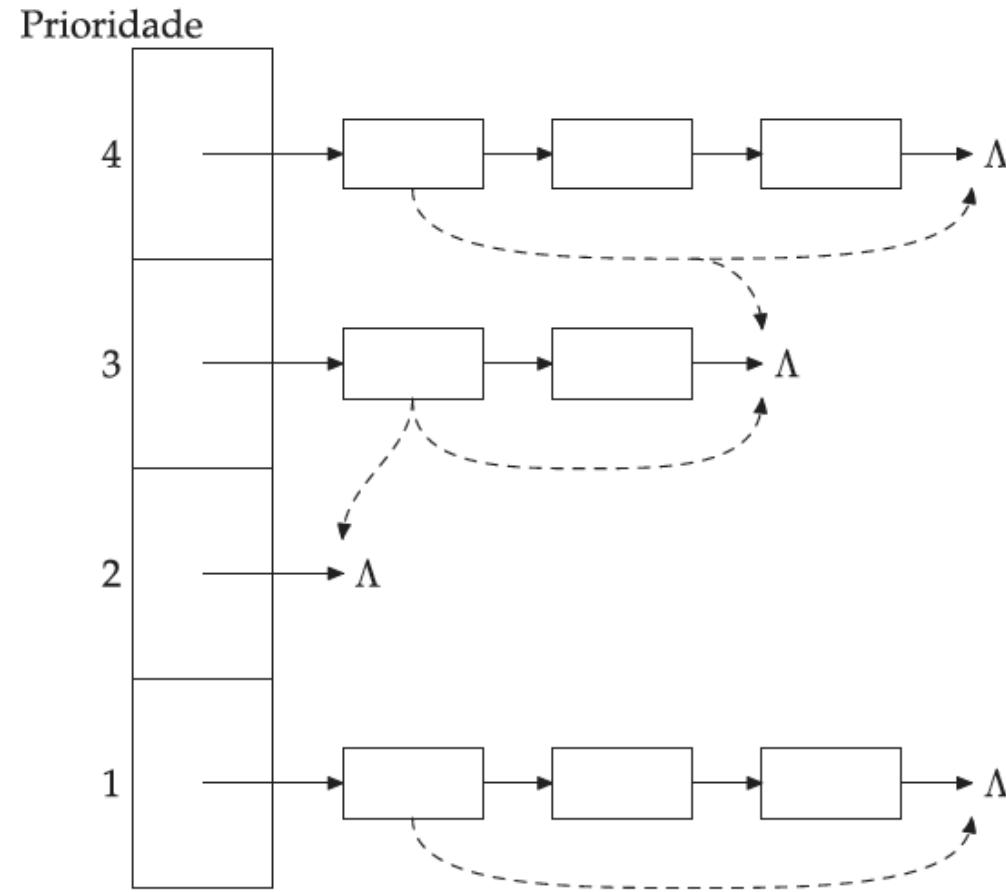
☞ Assim, o MLFQ adapta a prioridade conforme o processo se comporta: **quem exige muita CPU desce**, enquanto **quem interage com o usuário sobe**.

Escalonamento por Filas de Retorno Multinível (MLFQ)



Processos começam na fila de maior prioridade.
Se não terminam no quantum, descem para a fila seguinte.
Fila mais baixa tem quantum maior (menos prioridade).

Fila de Retorno Multinível (cont.)



Parâmetros de Escalonamento

- Prioridade não é o único parâmetro que podemos mudar para ajustar o comportamento do sistema
- O principal exemplo disso é a forma pela qual o *Compatible Time-Sharing System* (CTSS) ajusta o quantum do processo
- O CTSS utiliza uma fila de retorno multinível com uma fatia de tempo crescendo exponencialmente
- O CTSS dá prioridade para processos focados em E/S sobre processos focados em computação
- O Windows NT é exemplo de sistema que utiliza *quantum* de diferentes dimensões para dar preferência para alguns processos sobre outros.



Escalonamento de Dois Níveis

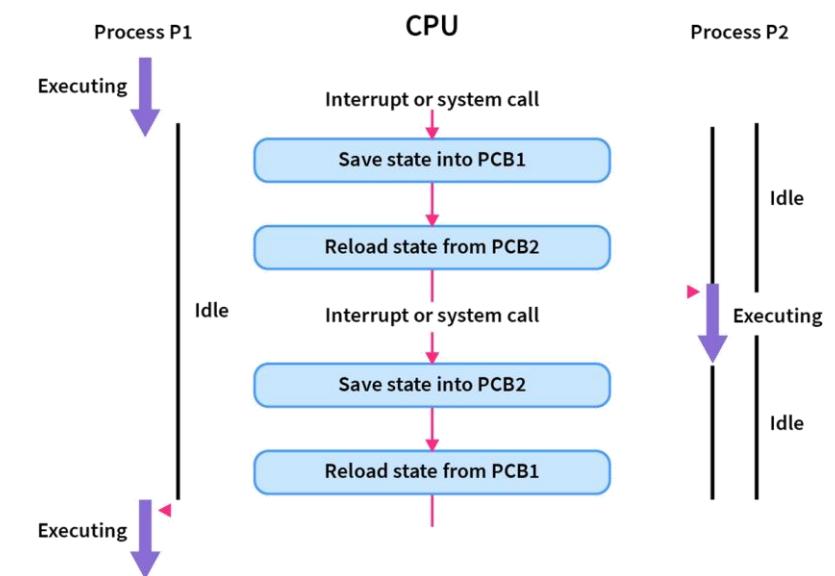
- Utilizando um escalonador de longa duração, de nível mais alto, que é executado mais lentamente, selecionamos o subconjunto de processos que são residentes.
- Então, um escalonador normal, de nível mais baixo e de curta duração, seleciona apenas entre os processos residentes na memória.

Escalonamento de Tempo Real

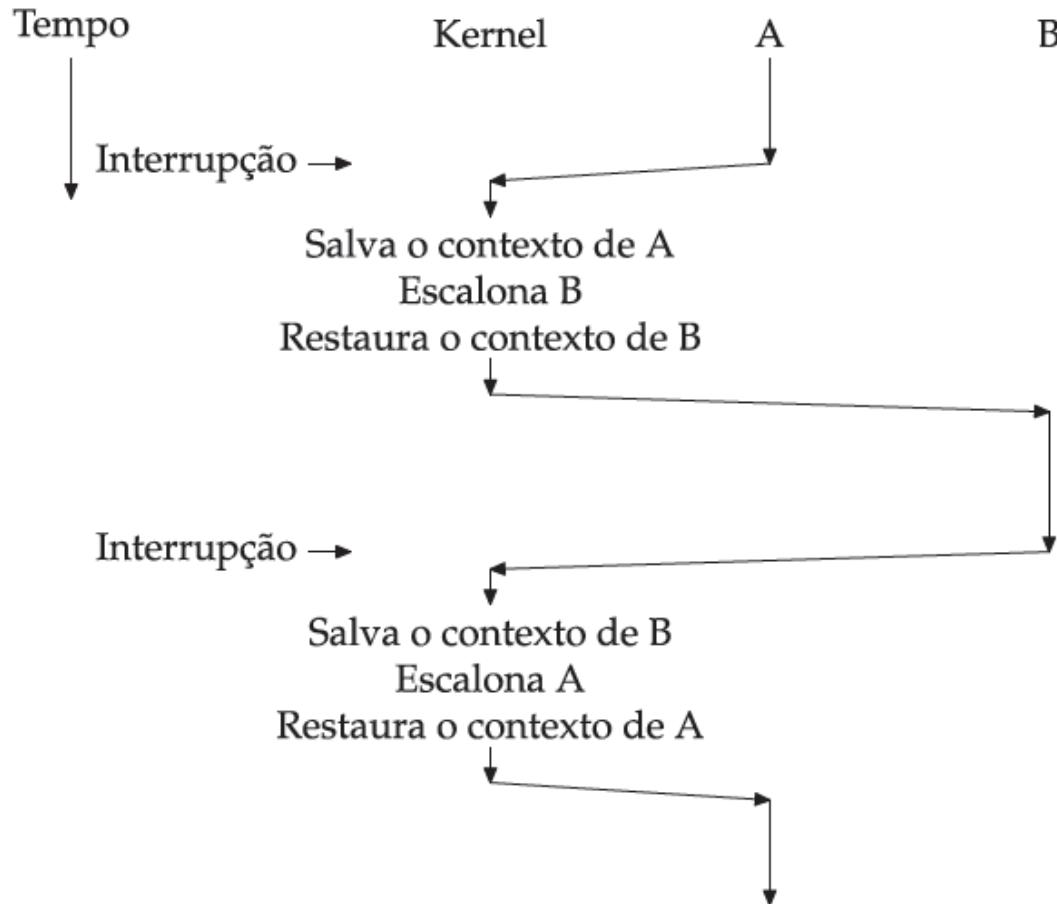
- Sistemas de processos específicos
- Eventos Automotivos
 - Exigências muito estritas no tempo de resposta para os eventos
 - Erros superiores a 1º de uma rotação podem ter efeito notavelmente prejudicial no desempenho do motor
- Se o código envolvido não completa sua função no tempo designado, então nenhuma parte do projeto de escalonamento ajudará.

Troca de Contexto

- Dê a CPU outro processo:
 - Transfere o fluxo de controle do processo corrente do usuário para o sistema operacional.
 - Salva a configuração do processo corrente.
 - Seleciona (escalona) o próximo processo.
 - Restaura a configuração (salva previamente) do próximo processo.
 - Retorna o fluxo de controle para o processo corrente (novo) do usuário.



Exemplo de Troca de Contexto



Criação de Processo

- Fork cria uma cópia do pai
- Spawn cria novo processo de execução para o novo programa
- Mecanismo:
 - Inicializar uma nova entrada na tabela de processos
 - Copiar a memória do pai (versão fork)
 - Carregar um novo programa (spawn)
 - Iniciar na configuração salva (spawn)

Processo de Finalização

- Iniciar pela finalização dos processos
- Mecanismos:
 - Fechar arquivos abertos
 - Reduzir o uso de recursos compartilhados
 - Contador de associação atinge zero
 - O pai (ou algum outro processo) consulta o *status do processo para determinar se ele terminou e por quê*. O SO deve registrar esse *status*.

Seções Críticas

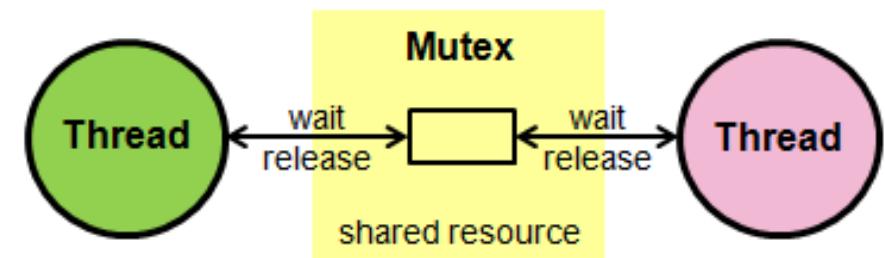
- Código que incorpora uma condição de disputa
- A exatidão do resultado depende da sincronização relativa do escalonamento entre dois ou mais processos que compartilham algum recurso.
- Exemplo:

```
if( head == NULL )  
    head = new_elem ;  
else  
    tail->next = new_elem ;  
tail = new_elem ;
```

Técnica de Exclusão Mútua – (MUTEX)

- Controle de interrupção

- Podemos bloquear uma seção crítica, evitando as interrupções que poderiam nos causar uma preempção enquanto estivermos na seção crítica
- Um dos problemas mais óbvios de manter as interrupções desligadas por muito tempo é que poderíamos perder as interrupções do relógio
- A natureza do controle de interrupções evita que múltiplos processos esperem pela seção crítica simultaneamente.
- Isso, por sua vez, não nos oferece qualquer possibilidade de priorizar entre processos que possam desejar obter acesso.

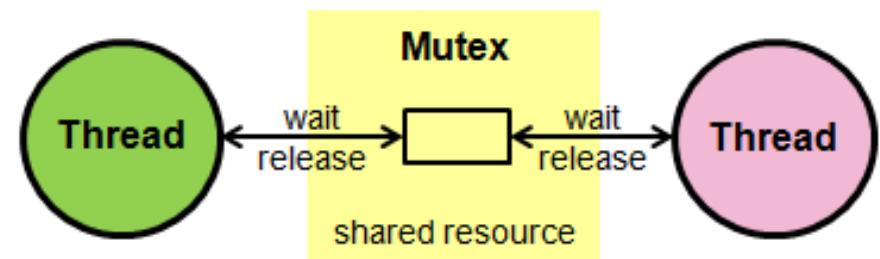


Técnica de Exclusão Mútua – (MUTEX)

- Manipular interrupções funciona bem para um único processador
- No entanto, se tivermos mais que um processador compartilhando a memória, impedir interrupções em um processador não impede que o outro acesse a memória
- Instrução *Test and set*
- Essa instrução testa o valor de uma localização de memória associada a instruções condicionais

- Exemplo:

```
mutex_lock:  
    tas _lock  
    blt mutex_lock  
    ret
```



Técnica de Exclusão Mútua – (MUTEX)

■ Algoritmo de Peterson

- Tem como base a ideia de que normalmente desejamos que dois processos concorrentes se alternem se um processo está interessado em utilizar o recurso compartilhado enquanto o outro não, o primeiro pode obtê-lo mesmo que não tenha chegado a sua vez.

```
void mutex_lock(int who) {  
    other = 1 - who;          // identifica o outro processo (se who=0 → other=1; se who=1 → other=0)  
    want[who] = 1;            // processo 'who' declara intenção de entrar na seção crítica  
    turn = other;             // dá preferência ao outro processo  
    while (want[other] && turn != who);  
        // espera ocupada ("busy wait");  
        // só sai do laço se o outro processo não quiser entrar  
        // OU se for a vez deste processo (turn == who)  
}
```

Técnica de Exclusão Mútua – (MUTEX)

- Semáforos

- Um dos modelos mais frequentemente estudados e implementados de exclusão mútua para os processos do usuário
- Definido por duas operações:
 - up(): Aumente o valor do semáforo em um
 - down(): Se o semáforo é 0, bloqueie até que se torne 0. Diminua o valor do semáforo em um
- Semáforo binário, em que o valor do semáforo possa assumir somente os valores 0 e 1.

Técnica de Exclusão Mútua – (MUTEX)

- Monitores
 - Característica de linguagem
 - Bloqueios
 - Somente um processo por vez é permitido acessar o monitor
 - “Monitores são pelo menos tão poderosos quanto semáforos, no sentido de que podem ser utilizados para todas as aplicações de semáforos”.

Técnica de Exclusão Mútua – (MUTEX)

- Troca de mensagens
 - Pode ser usado para sincronização
 - Baseado em enviar e receber
 - **Princípio rendezvous:** o processo emissor é bloqueado até que um receptor apareça

Deadlock

- Também denominado “abraço fatal”
- Exemplo:
 - Digamos que o processo A ganhe um bloqueio exclusivo na impressora, mas, antes de tentar acessar a unidade de CD-ROM, o processo B é executado e bloqueia a unidade de CD-ROM. Em algum momento, B está bloqueado, esperando pelo acesso à impressora que A retém, e A está bloqueado no acesso à unidade de CD-ROM, que B retém
 - Sem esperança de fugir: os dois processos estão bloqueados

Deadlock (cont.)

- Condições necessárias e suficientes
 - **Exclusão mútua** – Não mais que um processo pode reter um recurso ao mesmo tempo.
 - **Retenção e espera** – Um processo não cede voluntariamente um recurso enquanto espera por outro.
 - **Não preempção** – Uma vez que concedemos a um processo acesso exclusivo para um recurso, não o retiramos à força.
 - **Espera circular** – O grafo de dependência é cíclico.

Deadlock (cont.)

- Lidando com deadlock
 - Ignorando
 - Pode ser raro
 - Pode não afetar o sistema operacional

Deadlock (cont.)

- Lidando com deadlock
 - Detectando
 - Detectar quando aconteceu e corrigi-lo
 - Verificamos há quanto tempo os processos foram bloqueados à espera de acesso exclusivo a um recurso
 - Devemos terminar um deles, na esperança de permitir que o outro continue

Deadlock (cont.)

- Lidando com deadlock

- Prevenindo

- Há momentos em que é possível estruturar o software envolvido de forma que o deadlock pode nunca ocorrer
 - A prevenção de deadlock normalmente resume-se à ideia de assegurar que nunca podemos criar um grafo de dependência cíclico.
 - A maneira mais simples de evitar ciclos de dependência é por meio da imposição de uma ordenação nos recursos

Deadlock (cont.)

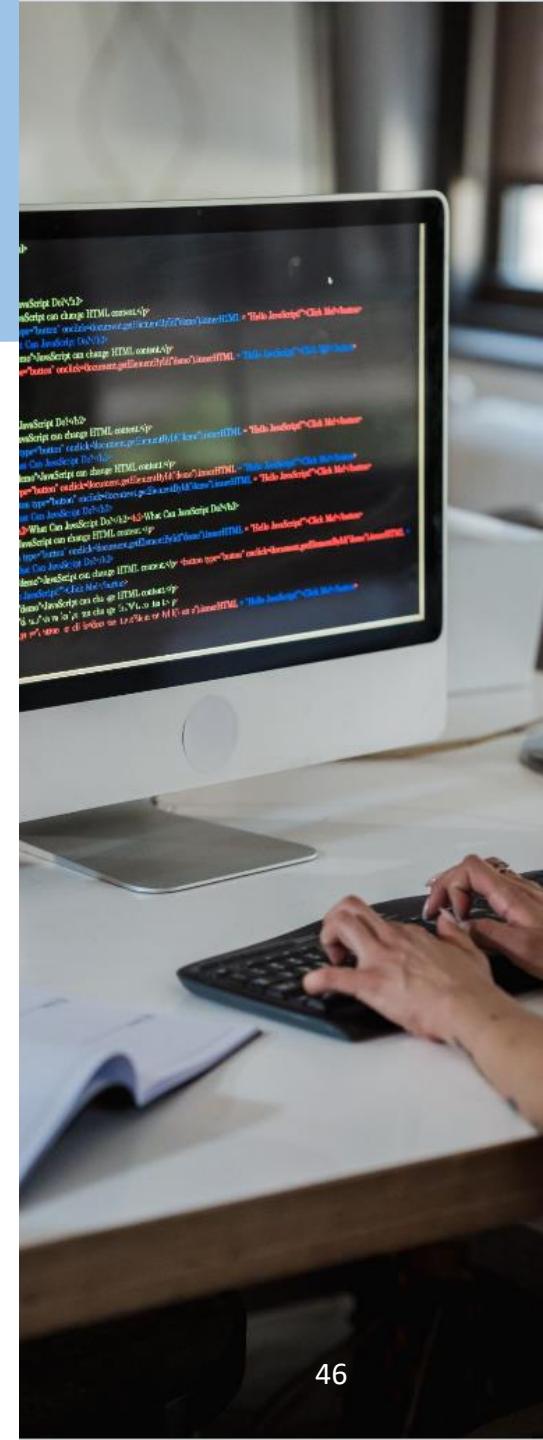
- Lidando com deadlock
 - Evitando
 - Analise cada solicitação
 - Algoritmo do Banqueiro é uma abordagem
 - Exigimos que os processos nos avisem com antecedência sobre quais serão suas necessidades de recurso

Algoritmo do Banqueiro

1. Temporariamente, diminua o valor $N(p,C)$ em um e aumente o $A(p,C)$ em um (implicitamente aumentando o elemento PC em um).
2. Selecione um processo p tal que todos os elementos da linha N_p sejam menores que ou iguais a $E - P$. Se essa linha não existir, então a nova configuração estaria insegura e o algoritmo termina com N e A sendo restaurados para seus valores anteriores ao passo 1.
3. Desconsidere as linhas p de A e N para as próximas verificações. (Aqui a linha A_p é implicitamente subtraída de P .)
4. Repita os passos 2 e 3 até encontrar uma configuração insegura ou até que todas as linhas sejam processadas. No último caso, o algoritmo verificou que a solicitação é segura e as alterações do passo 1 se tornam permanentes.

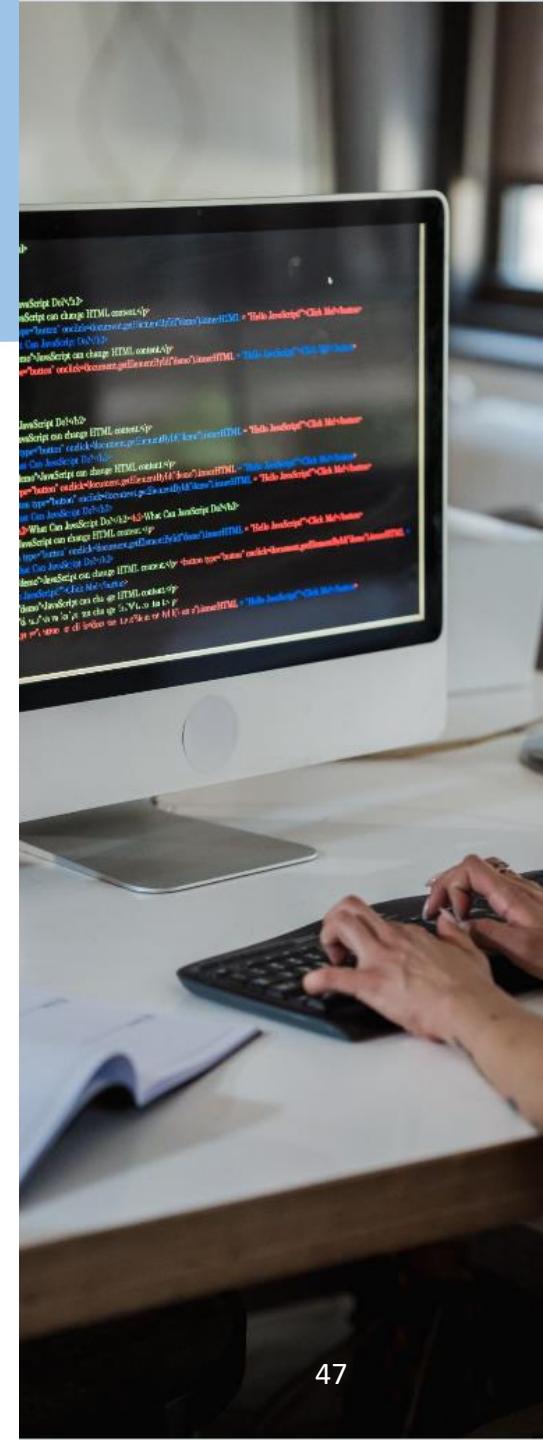
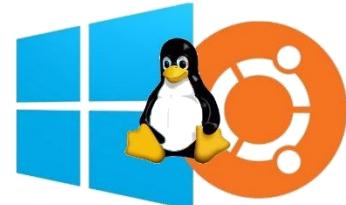
Processos vs Threads

Aspecto	fork()	pthread_create()
O que cria	Um novo processo	Uma nova thread dentro do mesmo processo
Espaço de memória	Cada processo tem sua própria cópia da memória (isolada, copy-on-write)	Todas as threads compartilham o mesmo espaço de memória (variáveis globais, heap, código)
Identificador	Retorna o PID do processo filho (inteiro)	Retorna um TID (Thread ID), geralmente armazenado em um tipo pthread_t
Execução	Pai e filho executam em paralelo , a partir da instrução seguinte ao fork()	A thread criada executa uma função separada passada como argumento
Comunicação	Processos diferentes precisam de IPC (pipes, sockets, memória compartilhada, etc.)	Threads compartilham memória, podendo comunicar-se via variáveis comuns (precisam de sincronização: mutexes, semáforos)
Isolamento	Um processo que cai (<i>segfault</i>) não derruba os outros processos	Uma thread que cai pode comprometer todo o processo
Uso de recursos	Criar processos é mais pesado (maior <i>overhead</i>)	Criar threads é mais leve (menos overhead)
Uso típico	Programas que precisam de isolamento forte ou rodar programas diferentes (ex.: shells, servidores que criam subprocessos)	Programas que precisam de tarefas paralelas cooperando (ex.: servidores multithread, cálculos paralelos)



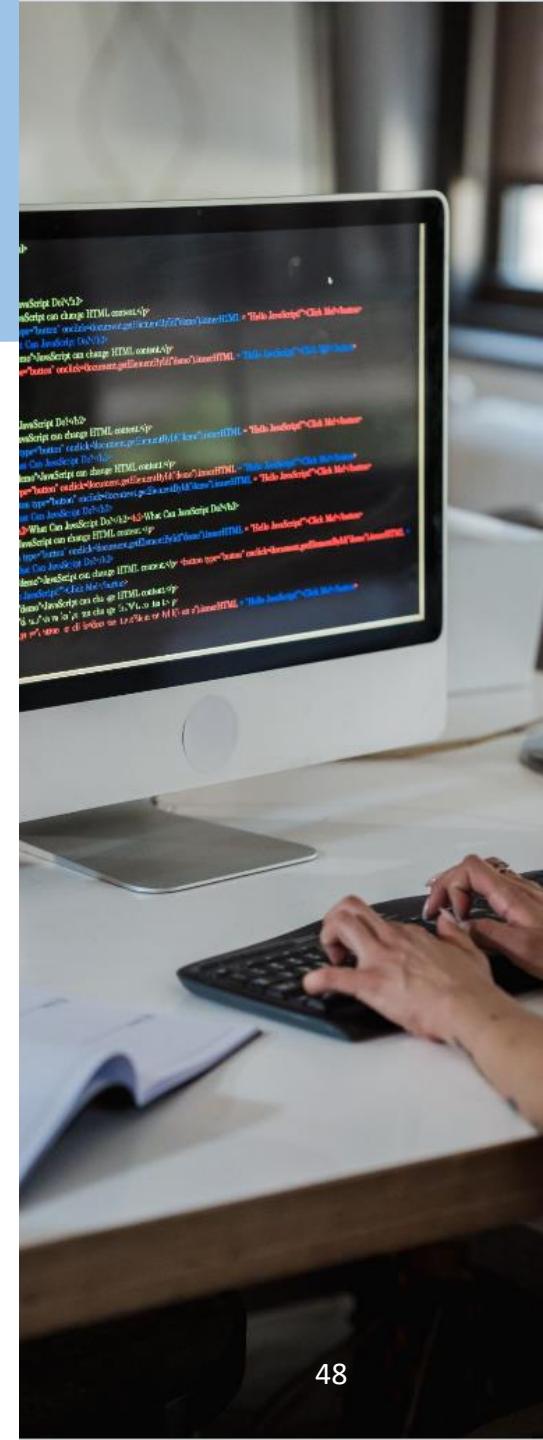
Demonstração Prática

- Criação de processos no Linux
 - Você vai precisar:
 - De um ambiente executando Linux:
 - VM
 - WSL2
 - Mac
 - Linux hospedeiro



Demonstração Prática

- Códigos de exemplo:
 - https://github.com/ProfessorFilipo/Aulas_SisOp
- Sugestão de IDEs:
 - <https://www.codeblocks.org/>
 - <https://code.visualstudio.com/>
 - <https://www.jetbrains.com/clion/>



Bacharelado em Ciência da Computação *Sistemas Operacionais*

Muito obrigado!

Prof. Me. Filipo Novo Mór

filipo.mor at unilasalle dot edu dot br

Parte desse material foi baseado no livro *Princípios de Sistemas Operacionais* de Brian L. Stuart, 1^a edição, ISBN-13 : 978-8522107339.

