Universidade LaSalle

Bacharelado em Ciência da Computação

Sistemas Operacionais

Aula começa as
10h30min

Prof. Me. Filipo Novo Mór

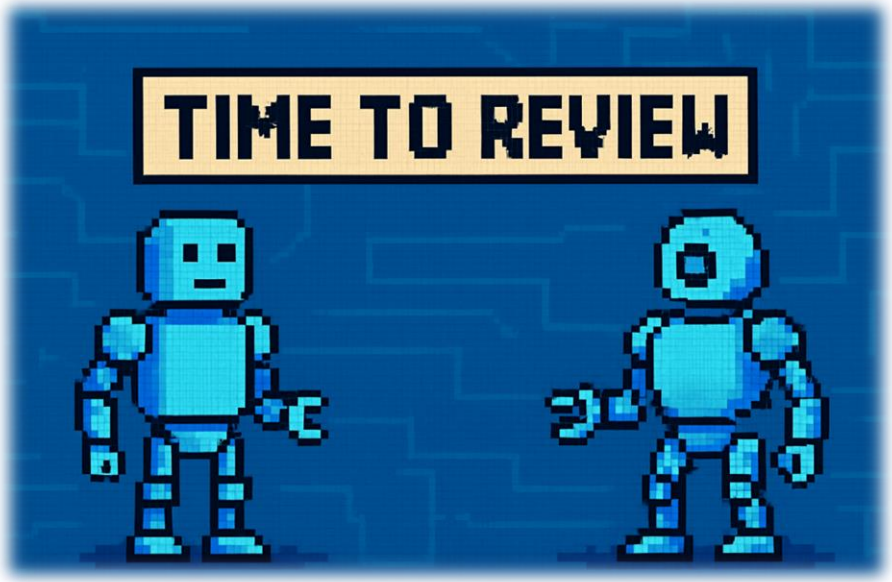
filipo.mor *at* unilasalle *dot* edu *dot* br

Conhecendo vocês!

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

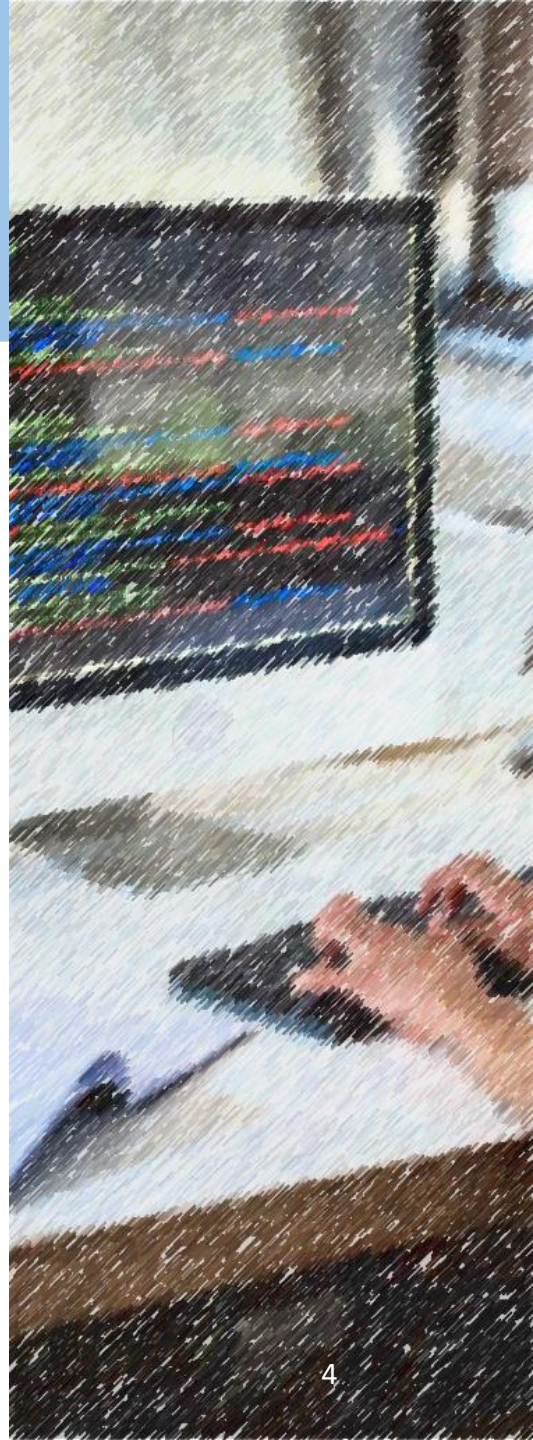
https://padlet.com/filipomor/sisop2025II

Revisando



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 (*segfault*) **não derruba os outros processos**	Uma thread que cai pode comprometer todo o processo
Uso de recursos	Criar processos é **mais pesado** (maior *overhead*)	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)



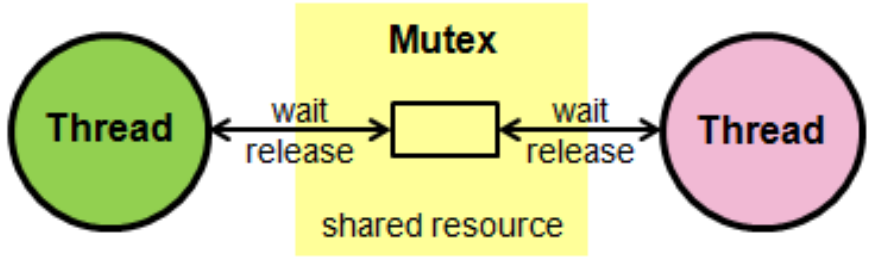
Seções Críticas

- 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)

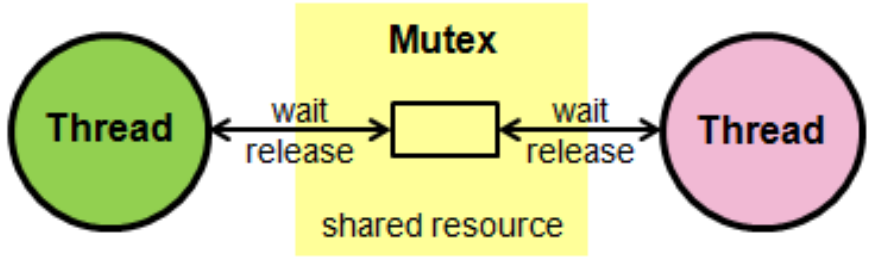
- Controle de interrupção
  - Podemos bloquear uma seção crítica, evitando as interrupções que poderiam nos causar uma preempção enquanto estivermos na seção crítica
  - Um dos problemas mais óbvios de manter as interrupções desligadas por muito tempo é que poderíamos perder as interrupções do relógio
  - 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 exclusivop ara 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

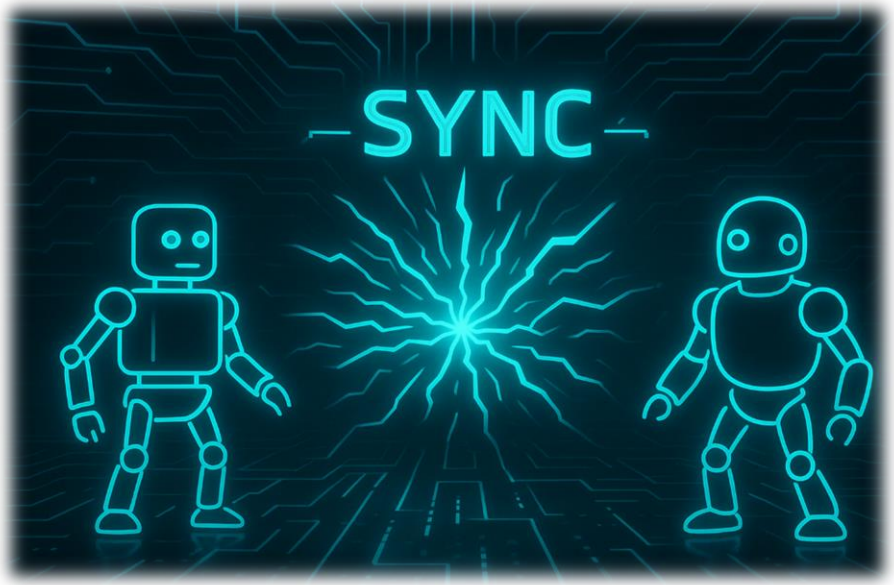
Deadlock (cont.)

- 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

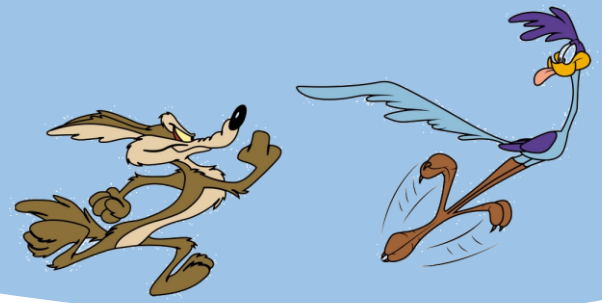
Deadlock (cont.)

- 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

Sincronização de Threads



Condição de Corrida



Uma **condição de corrida** ocorre quando o comportamento de um programa depende da ordem ou do *timing* em que *threads* são executadas, resultando em comportamentos imprevisíveis ou incorretos.

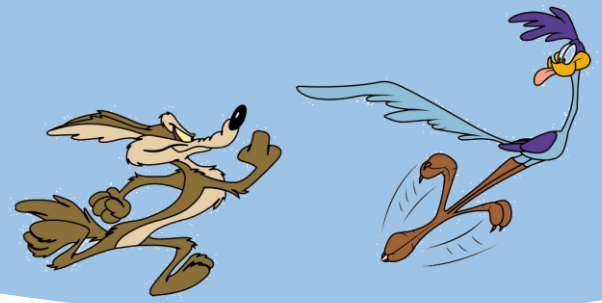
Contexto:

- **Threads**: Múltiplas threads executam tarefas concorrentemente.
- **Compartilhamento de Recursos**: Threads compartilham recursos como variáveis, memória ou dispositivos.
- **Sincronização**: A falta de mecanismos adequados de sincronização pode levar a condições de corrida.

Causas Comuns:

- **Acesso Concorrente:** Múltiplas threads acessam o mesmo recurso simultaneamente.
- **Falta de Sincronização:** Ausência de mecanismos como *locks*, semáforos ou monitores.
- **Ordem de Execução:** A ordem em que as threads são executadas afeta o resultado.

Condição de Corrida



Uma **condição de corrida** ocorre quando o comportamento de um programa depende da ordem ou do *timing* em que *threads* são executadas, resultando em comportamentos imprevisíveis ou incorretos.

Impactos:

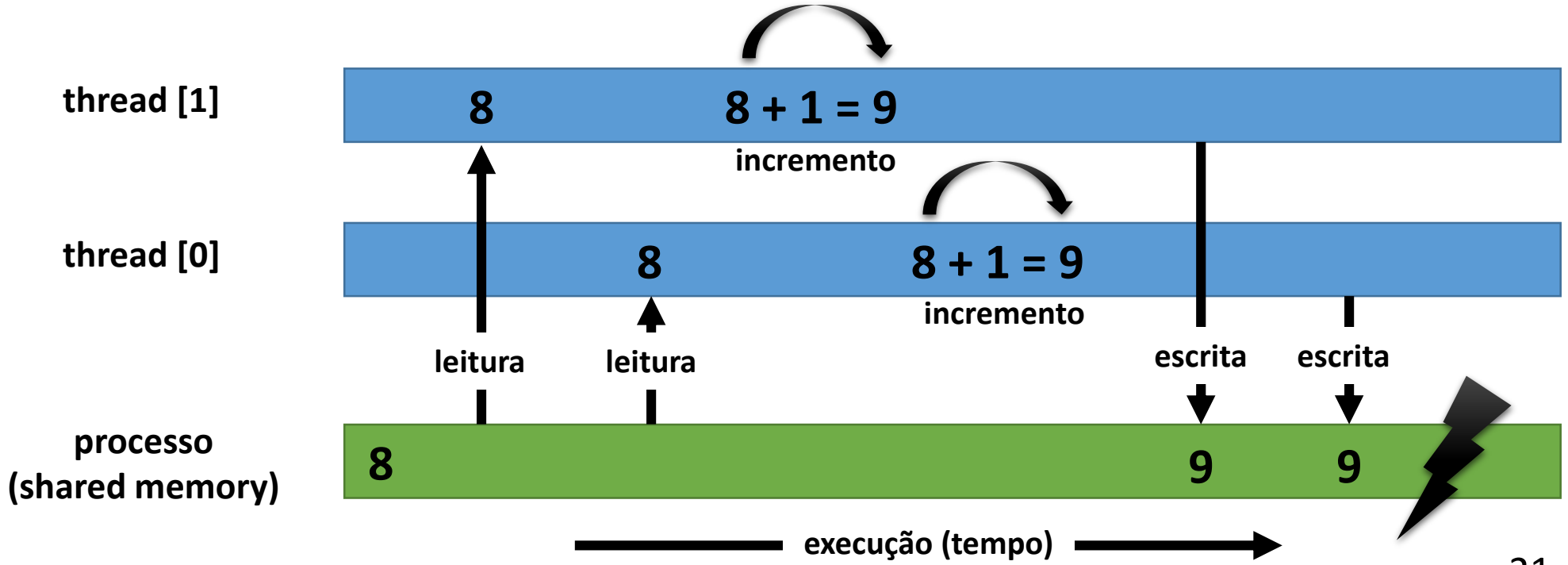
- **Inconsistência de Dados:** Valores ou estados incorretos.
- **Comportamento Imprevisível:** Resultados diferentes em execuções diferentes.
- **Bugs Difíceis de Reproduzir:** Problemas que ocorrem apenas sob condições específicas de timing.

Soluções:

- **Locks (Mutex):** Garantir que apenas uma thread acesse o recurso por vez.
- **Semáforos:** Controlar o acesso a recursos compartilhados.
- **Monitores:** Estruturas de alto nível para sincronização.
- **Variáveis Atômicas:** Operações indivisíveis que evitam condições de corrida.

Condição de Corrida

Uma **condição de corrida** ocorre quando o comportamento de um programa depende da ordem ou do *timing* em que *threads* são executadas, resultando em comportamentos imprevisíveis ou incorretos.



Senta que lá vem história

- Demonstração Prática

  https://github.com/ProfessorFilipo/Aulas_SisOp/blob/main/sincronizacao/exemploThreadsSemafarosBarreirasMutex.c

Hora do Desafio Prático!

- 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/

Para esse desafio, você precisará de uma VM ou sistema físico rodando Linux ou MacOS.
IDEs online não servem.

Universidade LaSalle

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, 1ª edição, ISBN-13 : 978-8522107339.