

Aula 02

Processos, API's e Execução Direta Limitada



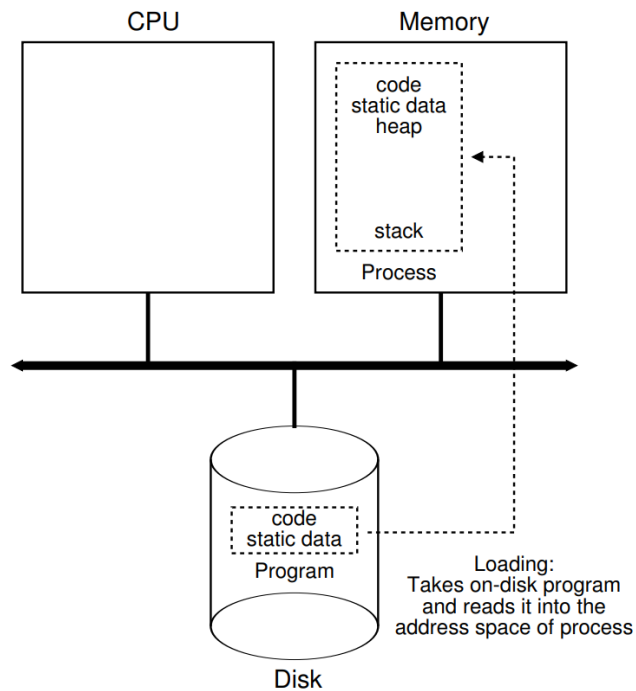
Objetivos

- Compreender o conceito de processo e sua importância como abstração do sistema operacional.
- Conhecer o funcionamento das principais chamadas de sistema: `fork()`, `exec()` e `wait()`.
- Entender como o sistema operacional gerencia a CPU entre vários processos.
- Aprender os mecanismos utilizados para retomada de controle da CPU pelo SO (ex: modo kernel, interrupções, troca de contexto).



Programa vs Processo

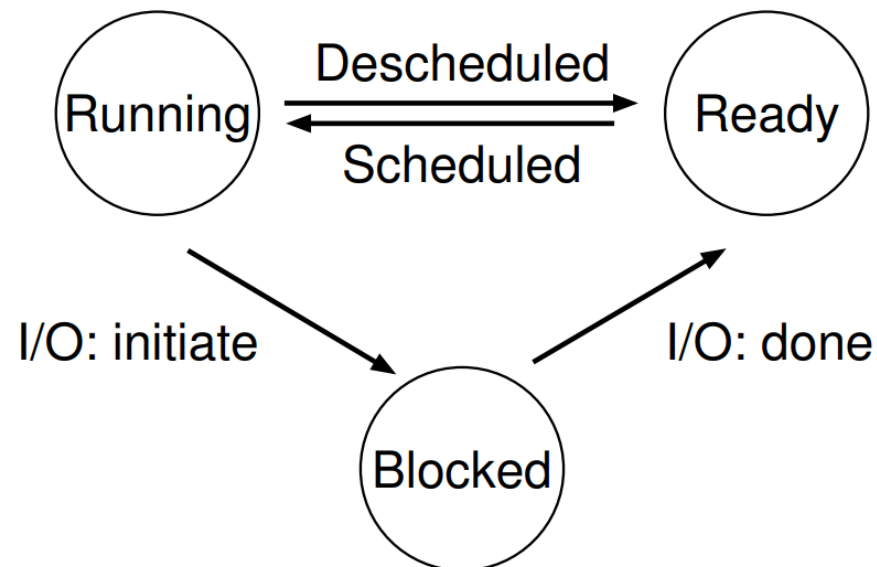
- Um **programa** é um conjunto de instruções armazenadas em disco
- Um **processo** é um programa em execução, com estado e contexto



Estados do Processo

Principais estados:

- **Running:** executando
- **Ready:** pronto para executar
- **Blocked:** esperando por evento



Time	Process ₀	Process ₁	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process ₀ initiates I/O
4	Blocked	Running	Process ₀ is blocked,
5	Blocked	Running	so Process ₁ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process ₁ now done
9	Running	–	
10	Running	–	Process ₀ now done



API de Processos - fork()

- fork() cria uma cópia do processo atual
- Retorno:
 - 0 no processo filho
 - PID do filho no pai

```
int main() {  
    int pid = fork();  
    if (pid == 0) {  
        // Filho  
    }  
    else {  
        // Pai  
    }  
    return 0;  
}
```



API de Processos - exec() e wait()

- exec() substitui o processo atual por outro
- Comumente usado após fork()

```
execl("/bin/ls", "ls", NULL);
```

- wait() bloqueia o pai até que o filho termine
 - Evita processos zumbis

```
int rc = fork();  
if (rc == 0) {  
    execvp(argv[0], argv);  
} else {  
    wait(NULL);  
}
```



Execução Direta Limitada

O sistema operacional quer que programas de usuário sejam executados rapidamente.

A execução direta na CPU é eficiente, mas **pode ser perigosa**:

- Programas poderiam monopolizar a CPU.
- Acessar recursos críticos indevidamente.
- Comprometer a segurança do sistema.
- **Desafio:** Como permitir execução rápida **sem perder o controle**?



Execução Direta Limitada



O programa de usuário executa **diretamente na CPU**, mas com restrições.



Ao iniciar, o processo está em **User Mode** (modo restrito).



Acesso a hardware e operações críticas só é possível via **chamada de sistema**.



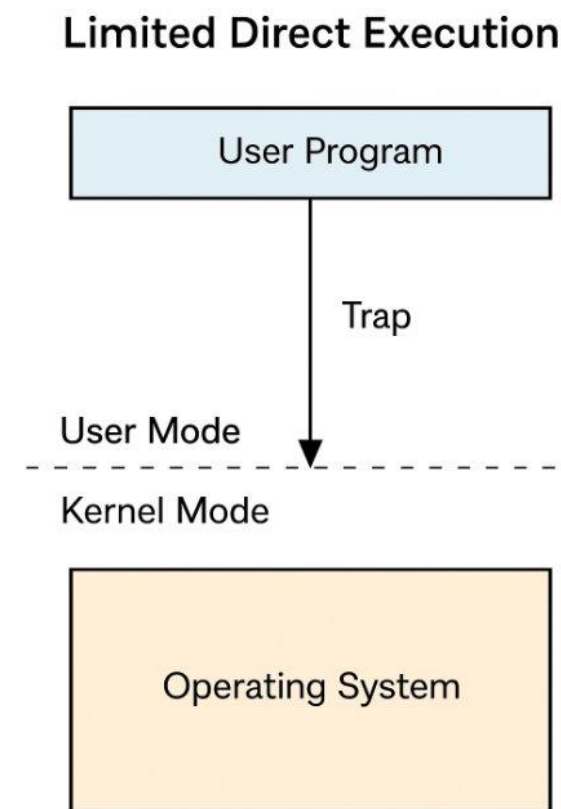
Uma chamada de sistema gera uma **trap**: transição segura para o **Kernel Mode**.



Resultado: Eficiência com segurança — o SO mantém o controle!

User Mode vs Kernel Mode

- **User Mode:** execução restrita, sem acesso direto a hardware
- **Kernel Mode:** acesso total para o SO
- Troca de modo feita por chamadas de sistema




Problema: Retomar o Controle da CPU

- E se o programa **nunca** fizer chamada de sistema?
 - Exemplo: `while(1) {}` → laço infinito
- O sistema operacional **não consegue** intervir facilmente.
- O **SO precisa de um meio de retomar o controle**, mesmo que o processo não coopere.



Solução 1: Execução Cooperativa

- O SO **confia no programa** para devolver o controle.
 - O processo faz chamadas como `read()`, `write()`, `exit()`...
 - Durante a chamada, ocorre uma **trap para o Kernel Mode**.
-  Limitação:
 - Um processo malcomportado pode **nunca fazer chamadas de sistema**.
 - Ineficiente e inseguro para sistemas com múltiplos usuários.



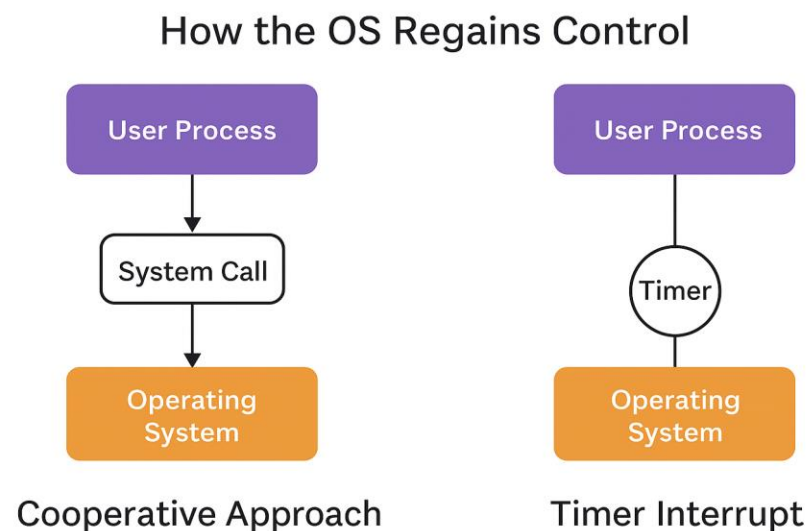
Solução 2: Timer Interrupt (Preempção)

- O hardware tem um **temporizador configurado pelo SO**.
- Quando o tempo se esgota, ocorre uma **interrupção automática**.
- O controle passa ao kernel:
 - O contexto atual é salvo.
 - O escalonador pode decidir trocar o processo.
- **O SO garante retomada periódica do controle!**



Preempção e Timer Interrupts

- **Cooperativo:** processo entrega controle
- **Preemptivo:** SO usa timer para interromper



Context Switch

Salva estado do
processo atual

Restaura estado
de outro
processo

