Aula 01: Introdução aos Sistemas Operacionais

Papel do Sistema Operacional

- Virtualiza recursos físicos (CPU, memória, disco).
- Gerencia hardware de forma eficiente.
- Fornece APIs (chamadas de sistema) para interação com o hardware.
- Atua como máquina virtual e gerenciador de recursos.

Virtualização

O Sistema Operacional transforma recursos físicos em abstrações mais úteis, fazendo com que cada processo "acredite" ter controle total da CPU e memória, compartilhando o hardware de forma segura e isolada.

Concorrência

Lida com múltiplos processos/threads. Problemas podem surgir, como na atualização de variáveis compartilhadas, exigindo mecanismos de sincronização.

Persistência

Garante que os dados sejam armazenados de forma duradoura, mesmo após o desligamento do sistema, através de chamadas de sistema para E/S com arquivos.

Objetivos de Projeto do SO

Abstração, Eficiência, Proteção e Confiabilidade.

Aula 02: Processos e Execução Direta Limitada

Programa vs. Processo

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

Estados do Processo

- **Running (Executando)**: O processo está utilizando a CPU.
- **Ready (Pronto)**: O processo está pronto para ser executado e aguarda a CPU.
- **Blocked (Bloqueado)**: O processo está esperando por um evento (ex: I/O).

API de Processos

- **fork()**: Cria uma cópia do processo atual (o processo filho). Retorna 0 no filho e o PID do filho no pai.
- **exec()**: Substitui o processo atual por um novo programa. Geralmente usado após `fork()`.
- **wait()**: Bloqueia o processo pai até que um de seus filhos termine, evitando processos zumbis.
- **waitpid()**: Permite esperar por um filho específico ou com opções mais avançadas.

Execução Direta Limitada

Permite que programas de usuário executem diretamente na CPU para eficiência, mas com restrições para segurança e controle do SO.

- **User Mode (Modo Usuário)**: Execução restrita, sem acesso direto a hardware.
- **Kernel Mode (Modo Kernel)**: Acesso total para o SO.
- **Trap**: Transição segura do User Mode para o Kernel Mode, geralmente via chamada de sistema.

Retomada do Controle da CPU pelo SO

- **Execução Cooperativa (Limitação)**: O SO confia que o programa fará chamadas de sistema (`read()`, `write()`, `exit()`) para devolver o controle. Um programa mal-intencionado pode não cooperar.
- **Timer Interrupt (Preempção)**: O hardware possui um temporizador configurado pelo SO. Ao expirar, uma interrupção automática transfere o controle para o kernel, que pode então salvar o contexto e trocar de processo.
- **Context Switch (Troca de Contexto)**: Processo de salvar o estado do processo atual e restaurar o estado de outro processo para que este possa executar na CPU.

Aula 03: Escalonamento de CPU

Objetivos do Escalonamento

Compreender políticas, avaliar vantagens/limitações, explorar mecanismos de justiça e adaptação, e entender o escalonamento em sistemas multiprocessadores.

Políticas de Escalonamento

- **FIFO (First-In, First-Out)**:

  - Preemptiva? Não.
  - Otimiza: Simples.
  - Fragilidades: Convoy effect (processos curtos esperam por longos).
- **SJF (Shortest Job First)**:
  - Preemptiva? Não.
  - Otimiza: Turnaround (tempo total para completar o processo).
  - Fragilidades: Requer tempo de execução conhecido antecipadamente.
- **STCF (Shortest Time-to-Completion First)**:
  - Preemptiva? Sim.
  - Otimiza: Turnaround.
  - Fragilidades: Idealizado (tempo de execução futuro é difícil de prever).
- **RR (Round Robin)**:
  - Preemptiva? Sim.
  - Otimiza: Tempo de resposta.
  - Fragilidades: Overhead de troca de contexto (frequente).
- **MLFQ (Multilevel Feedback Queue)**:
  - Heurística para lidar com processos de tempo de execução variável.
  - Múltiplas filas com prioridades decrescentes.
  - Feedback baseado no comportamento do processo (ex: desce de fila se usar muito a CPU, sobe se fizer muito I/O).

Problemas e Soluções

- **Starvation**: Processos de baixa prioridade podem nunca executar. Solução: Boost periódico (move todos os jobs para a fila de maior prioridade periodicamente).
- **Não-falsificabilidade**: Evitar que processos "enganem" o escalonador.

Escalonamento por Loteria (Lottery Scheduling)

- Natureza: Probabilística.
- Processos recebem "bilhetes" de loteria; quanto mais bilhetes, maior a chance de serem selecionados.
- Justiça: Atingida a longo prazo.
- Simplicidade: Alta.

Escalonamento por Stride (Stride Scheduling)

- Natureza: Determinística.
- Atribui uma taxa de avanço (stride) a cada processo, garantindo um avanço proporcional à sua prioridade de forma justa e previsível.
- Justiça: Curto e longo prazo.
- Simplicidade: Moderada.

Multiprocessamento

- **Novos Desafios**: Vários núcleos, compartilhamento de memória/cache, coerência de cache, afinidade de CPU, balanceamento de carga.
- **SQMS (Single Queue Multiprocessor Scheduling)**:
  - Uma única fila global para todos os CPUs.
  - Escalabilidade: Baixa.
  - Cache Affinity: Ruim (processos podem migrar entre CPUs e invalidar cache).
  - Balanceamento: Simples (naturalmente balanceado).
- **MQMS (Multiple Queue Multiprocessor Scheduling)**:
  - Múltiplas filas, uma para cada CPU.
  - Escalabilidade: Alta.
  - Cache Affinity: Boa (processos tendem a permanecer na mesma CPU).
  - Balanceamento: Requer migração de jobs (work stealing) para balancear carga.
- **Técnicas Avançadas**: Migração controlada de jobs, Work stealing.
- **Agendadores Linux**: O(1), CFS (Completely Fair Scheduler), BFS (Brain Fuck Scheduler).

Aula 04 e Atividade 3.1, 3.2: Espaços de Endereçamento e Alocação de Memória

Espaço de Endereçamento Virtual

Abstração criada pelo SO que dá a cada processo a ilusão de possuir sua própria memória exclusiva.

- **Benefícios**: Transparência, Eficiência e Proteção.

Estrutura Típica

- **Segmento de Código**: Onde as instruções do programa residem.
- **Heap**: Para alocação dinâmica de memória (`malloc`, `free`). Cresce positivamente.
- **Stack (Pilha)**: Para variáveis locais, argumentos de funções e valores de retorno. Cresce negativamente.

O endereço que o programa "vê" é virtual; o SO e o hardware o traduzem para endereços físicos.

Tipos de Memória em C

- **Stack (Pilha)**: Memória automática, liberada ao final da função. Ex: `int x;`.
- **Heap (Monte)**: Memória manual, alocada com `malloc` e liberada com `free`. Tem vida longa, é flexível, mas propensa a erros se não gerenciada corretamente.

Alocação com malloc() e free()

- `malloc()`: Aloca memória no heap e retorna um ponteiro para a área alocada.
- A memória alocada com `malloc()` deve ser liberada com `free()`.

Erros Comuns

- **Acesso sem alocar**: Usar ponteiro NULL ou não inicializado.
- **Estouro de buffer**: Escrever além do limite alocado.
- **Não liberar (Memory Leak)**: Esquecer de chamar `free()`.
- **Liberar duas vezes**: Chamar `free()` mais de uma vez para o mesmo ponteiro.
- **Uso após liberação (Dangling Pointer)**: Usar o ponteiro depois de `free()`.

Ferramentas de Diagnóstico

- **valgrind**: Detecta vazamentos de memória, acessos inválidos e uso após `free()`.
- **gdb**: Ferramenta de depuração interativa para análise passo a passo.
- **free**: Exibe a quantidade de memória livre e usada no sistema.
- **ps**: Lista os processos ativos, permitindo monitorar o uso de recursos.
- **pmap**: Mostra o mapeamento de memória de um processo específico.

Níveis de Gerenciamento de Memória

- **Camada de Biblioteca (`malloc`, `free`)**: Gerenciada pelo programa (usuário) dentro do espaço de endereçamento virtual do processo.
- **Chamadas de SO (`brk`, `mmap`)**: Permitem ao Sistema Operacional ajustar o tamanho do heap ou mapear regiões de memória.