Resumo para a Prova de Sistemas Operacionais I

Contents

1 Introdução aos Sistemas Operacionais

1.1 Papel do SO

- 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.

1.2 Virtualização

O SO transforma recursos físicos em abstrações úteis, permitindo que cada processo acredite ter controle total da CPU e memória, compartilhando o hardware de forma segura e isolada.

1.3 Concorrência

Gerencia múltiplos processos/threads. Problemas como atualização de variáveis compartilhadas exigem mecanismos de sincronização.

1.4 Persistência

Garante armazenamento duradouro de dados, mesmo após desligamento, via chamadas de sistema para E/S com arquivos.

1.5 Objetivos de Projeto do SO

Abstração, Eficiência, Proteção e Confiabilidade.

2 Processos e Execução Direta Limitada

2.1 Programa vs. Processo

- **Programa**: Conjunto de instruções armazenadas em disco.
- **Processo**: Programa em execução, com estado e contexto próprios.

2.2 Estados do Processo

- **Running (Executando)**: Utilizando a CPU.
- **Ready (Pronto)**: Aguardando a CPU.
- **Blocked (Bloqueado)**: Esperando por um evento (ex.: I/O).

2.3 API de Processos

- `fork()`: Cria uma cópia do processo atual. Retorna 0 no filho e o PID do filho no pai.
- `exec()`: Substitui o processo atual por um novo programa. Usado após `fork()`.
- `wait()`: Bloqueia o processo pai até que um filho termine, evitando processos zumbis.
- `waitpid()`: Espera por um filho específico com opções avançadas.

2.4 Execução Direta Limitada

Permite execução direta na CPU para eficiência, com restrições para segurança:

- **User Mode**: Execução restrita, sem acesso direto ao hardware.
- **Kernel Mode**: Acesso total para o SO.
- **Trap**: Transição segura do User Mode para o Kernel Mode.

2.5 Retomada do Controle da CPU pelo SO

- **Execução Cooperativa**: O SO confia que o programa devolverá o controle via chamadas como `read()`, `write()`, `exit()`. Programas mal-intencionados podem não cooperar.
- **Timer Interrupt (Preempção)**: Temporizador transfere controle ao kernel ao expirar.
- **Context Switch**: Salva o estado do processo atual e restaura outro.

3 Escalonamento de CPU

3.1 Objetivos

Compreender políticas, avaliar vantagens/limitações, explorar justiça/adaptação e escalonamento em multiprocessadores.

3.2 Políticas de Escalonamento

- **FIFO (First-In, First-Out)**:
  - Preemptiva? Não.
  - Otimiza: Simplicidade.
  - Fragilidade: Convoy effect.
- **SJF (Shortest Job First)**:
  - Preemptiva? Não.
  - Otimiza: Turnaround.
  - Fragilidade: Requer tempo de execução conhecido.
- **STCF (Shortest Time-to-Completion First)**:
  - Preemptiva? Sim.
  - Otimiza: Turnaround.
  - Fragilidade: Tempo de execução futuro difícil de prever.
- **RR (Round Robin)**:
  - Preemptiva? Sim.

  - Otimiza: Tempo de resposta.
  - Fragilidade: Overhead de troca de contexto.
- **MLFQ (Multilevel Feedback Queue)**:
  - Heurística para processos com tempo variável.
  - Múltiplas filas com prioridades decrescentes.
  - Feedback: Desce se usa muita CPU, sobe se faz I/O.

3.3 Problemas e Soluções

- **Starvation**: Processos de baixa prioridade podem não executar. **Solução**: Boost periódico.
- **Não-falsificabilidade**: Evitar manipulação do escalonador.

3.4 Escalonamento por Loteria

- Natureza: Probabilística.
- Processos recebem bilhetes; mais bilhetes, maior chance de execução.
- Justiça: Longo prazo.
- Simplicidade: Alta.

3.5 Escalonamento por Stride

- Natureza: Determinística.
- Taxa de avanço (stride) proporcional à prioridade.
- Justiça: Curto e longo prazo.
- Simplicidade: Moderada.

3.6 Multiprocessamento

- Desafios: Múltiplos núcleos, coerência de cache, afinidade de CPU, balanceamento de carga.
- **SQMS (Single Queue Multiprocessor Scheduling)**:
  - Escalabilidade: Baixa.
  - Cache Affinity: Ruim.
  - Balanceamento: Simples.
- **MQMS (Multiple Queue Multiprocessor Scheduling)**:
  - Escalabilidade: Alta.
  - Cache Affinity: Boa.

  - Balanceamento: Requer migração (work stealing).
- Técnicas: Migração controlada, Work stealing.
- Agendadores Linux: O(1), CFS, BFS.

4 Espaços de Endereçamento e Alocação de Memória

4.1 Espaço de Endereçamento Virtual

Abstração que dá a ilusão de memória exclusiva por processo.

- Benefícios: Transparência, Eficiência, Proteção.
- Estrutura:
  - **Código**: Instruções do programa.
  - **Heap**: Alocação dinâmica (malloc, free). Cresce positivamente.
  - **Stack**: Variáveis locais, argumentos, retornos. Cresce negativamente.

4.2 Tipos de Memória em C

- **Stack**: Memória automática, liberada ao fim da função (ex.: `int x;`).
- **Heap**: Memória manual (`malloc`, `free`). Flexível, mas propensa a erros.

4.3 Alocação com `malloc()` e `free()`

- `malloc()`: Aloca memória no heap, retorna ponteiro.
- `free()`: Liberta memória alocada.
- Erros:
  - Acesso sem alocar (ponteiro NULL).
  - Estouro de buffer.
  - Memory Leak (não chamar `free()`).
  - Liberação dupla.
  - Uso após liberação (Dangling Pointer).

4.4 Ferramentas de Diagnóstico

- `valgrind`: Detecta vazamentos, acessos inválidos, uso após `free()`.
- `gdb`: Depuração interativa.
- `free`: Memória livre/usada no sistema.
- `ps`: Lista processos ativos.
- `pmap`: Mapeamento de memória de um processo.

4.5 Níveis de Gerenciamento de Memória

- **Biblioteca**: `malloc`, `free` no espaço virtual do processo.
- **SO**: `brk`, `mmap` ajustam o heap ou mapeiam memória.