Universidade Federal de Lavras
PPGCC
PCC508 – Sistemas Operacionais

Tópico 11 Lista Avaliativa

Douglas Aquino T. Mendes
28 de janeiro de 2025

Sumário

1 Introdução

Este documento tem como objetivo apresentar o desenvolvimento das atividades avaliativas para o tópico 11 da disciplina de Sistemas Operacionais, focando na implementação de códigos em linguagem C. Serão apresentadas as questões, a resolução, os códigos desenvolvidos, seguido da apresentação dos resultados da execução do código.

2 Questões

2.1 1

Pergunta: 1) Explique os diferentes estados que um processo do Linux pode estar. Também fale sobre as possíveis transições entre os estados.

Resposta: Os estados principais de um processo no Linux são:

- Em Execução (Running): O processo está atualmente utilizando a CPU e executando suas instruções [1, p. 72].
- Pronto (Ready): O processo está apto a ser executado, mas está temporariamente parado pois a CPU está alocada para outro processo. O processo está aguardando sua vez para utilizar a CPU. A transição para o estado de execução ocorre quando o escalonador de processos decide que o processo pronto deve ser executado [1, p. 72].
- Bloqueado (Blocked): O processo não pode continuar sua execução até que um evento externo ocorra. Esse evento pode ser a conclusão de uma operação de I/O, a chegada de dados em um pipe ou socket, ou a liberação de um recurso. O processo permanece neste estado até que o evento esperado aconteça, momento em que ele passa para o estado pronto [1, p. 72].

As transições entre esses estados são as seguintes:

- Execução → Bloqueado: Ocorre quando o processo precisa esperar por um evento externo, como uma operação de I/O, a leitura de um pipe vazio, ou um recurso não disponível. A transição é iniciada pelo próprio processo ao realizar uma chamada de sistema que causa o bloqueio [1, p. 64].
- Execução → Pronto: Ocorre quando o escalonador do sistema operacional decide que o tempo de execução do processo atual expirou ou que um outro processo com prioridade mais alta deve ser executado. O processo é interrompido e colocado na fila de processos prontos [1, p. 64].
- Pronto → Execução: Ocorre quando o escalonador de processos escolhe o processo pronto para executar na CPU. Essa escolha é baseada em algoritmos de escalonamento que tentam equilibrar eficiência e justiça entre os processos [1, p. 64].
- Bloqueado → Pronto: Ocorre quando o evento pelo qual o processo estava esperando acontece [1, p. 64].

2.2 2

Pergunta: 2) Explique como funciona a criação dos processos no Linux, quais as chamadas de sistema envolvidas e como a chamada fork é implementada.

Resposta: O processo de criação envolve uma série de chamadas de sistema e estruturas de dados, com a chamada fork sendo a principal maneira de se criar um novo processo [1, p. 37]. Chamadas de Sistema Envolvidas:

- fork(): Esta é a chamada de sistema essencial para criar um novo processo no Linux. Ela cria uma cópia exata do processo original, incluindo descritores de arquivos, registradores e tudo mais. O processo original é chamado de processo pai, e o novo processo é chamado de processo filho. Cada um tem suas próprias imagens de memória privadas. Após o fork, o pai e o filho seguem seus próprios caminhos. O valor de retorno de fork é zero no processo filho e o PID (Process Identifier) do filho no processo pai.
- waitpid(): Usada pelo processo pai para esperar que um processo filho termine a sua execução.
- execve() (ou exec): Após um fork, o processo filho geralmente precisa executar um código diferente do processo pai. A chamada execve substitui a imagem de núcleo do processo filho pelo arquivo nomeado como parâmetro, permitindo que ele execute um novo programa. Na prática, a chamada de sistema em si é exec, mas várias rotinas de biblioteca a chamam com parâmetros diferentes.
- exit(): Essa chamada é usada por um processo para terminar sua execução e retornar um status de saída para o processo pai através do waitpid.

A chamada de sistema fork() é implementada com uma série de passos no núcleo do Linux, a Implementação da Chamada fork():

- O núcleo Linux representa internamente os processos através da estrutura task_struct. Ao executar a chamada fork(), o núcleo cria uma nova estrutura task_struct para o processo filho, assim como outras estruturas de dados de acompanhamento, como a pilha do modo núcleo e a estrutura thread_info [1, p. 306].
- Uma área de usuário é criada para o processo filho e é preenchida principalmente com dados do pai. O filho recebe um PID, seu mapa de memória é configurado e ele recebe acesso compartilhado aos arquivos do seu pai [1, p. 306].
- O mapa de memória do processo filho é configurado. Em sistemas que suportam arquivos mapeados, as tabelas de páginas são configuradas para indicar que nenhuma página está na memória, exceto talvez uma página de pilha, mas que o espaço de endereçamento tem o suporte do arquivo executável no disco [1, p. 512].
- Os arquivos abertos pelo processo pai são compartilhados com o filho. Mudanças feitas nos arquivos por um processo são visíveis para o outro.
- Os registradores do processo filho são configurados.
- A chamada fork() retorna um valor diferente de zero (o PID do filho) para o processo pai e zero para o processo filho. Isso permite que os processos saibam qual deles é o pai e qual é o filho e executar diferentes blocos de código.

2.3 3

Pergunta: 3) Como o Linux trata as threads de programas de usuário? Como funciona a criação das mesmas com a chamada de sistema clone?

Resposta: No Linux, threads de programas de usuário são tratadas de forma semelhante a processos, compartilhando diversos recursos. O núcleo do Linux utiliza a estrutura `task_struct` para representar tanto processos quanto threads [1, p. 511].

Cada entidade de execução (processo ou thread) é tratada como uma tarefa distinta pelo sistema. A estrutura `task_struct` representa qualquer contexto de execução, e tanto processos quanto threads possuem instâncias dessa estrutura. Threads de usuário operam no modo usuário, mas chamadas de sistema são executadas no modo núcleo. O escalonador trata threads e processos com a mesma política, permitindo execução independente.

A chamada de sistema `clone()` é utilizada para criar threads, permitindo controle sobre o compartilhamento de recursos. Sua sintaxe é:

```
pid = clone(function, stack_ptr, sharing_flags, arg);
```

- `function`: Ponteiro para a função que o novo thread executará.
- `stack_ptr`: Ponteiro para a pilha do novo thread.
- `arg`: Argumento passado para a função.
- `sharing_flags`: Define quais recursos serão compartilhados.

Flags de Compartilhamento:

- **CLONE_VM:** Compartilha espaço de endereçamento (memória virtual).
- **CLONE_FS:** Compartilha diretórios raiz e de trabalho.
- **CLONE_FILES:** Compartilha descritores de arquivos.
- **CLONE_SIGHAND:** Compartilha tratadores de sinais.
- **CLONE_PARENT:** Mantém o mesmo processo pai do thread criador.

A granularidade no compartilhamento de recursos é possível devido à separação de estruturas de dados no kernel.

O Linux distingue entre **PID (Process Identifier)** e **TID (Thread Identifier)**. O PID identifica um processo, enquanto o TID identifica uma tarefa dentro de um processo. Todos os threads de um mesmo processo compartilham o mesmo PID. Quando `clone()` cria um novo processo sem compartilhamento, um novo PID é atribuído [1, p. 515].

2.4 4

Pergunta: 4) Quais são as tarefas realizadas pela chamada do_exit()?

Resposta: A chamada do_exit() é uma função interna do kernel do Linux que é executada quando um processo termina, seja por uma saída normal, por um erro ou por ser morto por outro processo. Esta função é responsável por liberar os recursos usados pelo processo e notificar o processo pai [1, p. 62]. As principais tarefas realizadas pela função do_exit() são:

- Liberar a Memória
- Liberar Recursos do Sistema
- Notificação do Processo Pai
- Remover a Estrutura task_struct
- Chamada de callbacks

Referências

[1] A. S. Tanenbaum e H. Bos, *SISTEMAS OPERACIONAIS MODERNOS*, 4ª ed. Boston: Pearson, 2021.