Documentação - Trabalho Prático 1 de Sistemas Operacionais: Gerenciamento de Processos

Arthur Vieira Silva

24 de Julho de 2024

Sumário

1 Introdução

1.1 Gerenciando Processos

Um processo nada mais é do que a abstração de um programa em execução, isto é, inclui valores do contador de programa, registradores, estado do processo, prioridade, variáveis, entre outros. Dessa forma, é tarefa do Sistema Operacional lidar com a criação, a parada, o término e o escalonamento dos processos.

A criação de um processo pode ocorrer, principalmente, devido a quatro fatores principais: o sistema foi inicializado, um processo em execução executa uma chamada de sistema para a criação de um processo, o usuário requisitou a criação de um novo processo ou devido à inicialização de tarefas em lote. Por outro lado, a finalização de um processo pode ocorrer de maneira voluntária (uma saída normal ou por erro) ou involuntária (um erro fatal ou o cancelamento por outro processo).

Sendo assim, o gerenciamento de processos figura como uma parte fundamental de qualquer Sistema Operacional pois, será ele quem irá controlar a maneira pela qual os processos serão executados, definindo qual processo executar, quando interromper a execução de um determinado processo e a transição de estados de um processo.

Essa transição de estados de um determinado processo pode ser definida da seguinte forma: um processo em execução (está realmente utilizando a CPU naquele instante) pode ser bloqueado (está incapaz de executar até que algum evento externo ocorra) ou pode alterar o seu estado para pronto (pode ser executado mas, está parado para que outro processo seja executado). Além disso, um processo bloqueado pode transitar para o estado pronto e, um processo pronto, pode transitar tanto para o estado em execução quanto para o estado bloqueado, como mostra a Figura 1 abaixo.

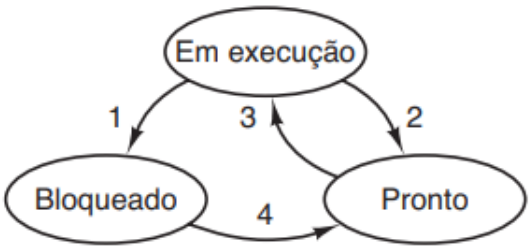

Figura 1: Transição de estados de um processo

1.2 Funcionamento do Programa

Diante desse contexto, o objetivo desse trabalho prático é implementar um simulador que será capaz de simular algumas funções de gerenciamento de processos: criação de um processo, substituição do processo atual por outro processo, transição de estados de um processo, escalonamento e troca de contexto. Sendo assim, teremos três tipos de processos: *commander*, *manager* e *reporter*.

A princípio, o processo *commander* será encarregado de dar início a simulação, ele irá, primeiramente, criar um *pipe* e depois um processo filho do tipo *process manager*. Com isso, o *commander* lerá comandos da entrada padrão a cada 1 segundo e vai escrevê-los no *pipe* para que o *process manager* seja encarregado de lê-los. Esses comandos são:

1. **Q**: Fim de uma unidade de tempo;
2. **U**: Desbloqueie o primeiro processo simulado que está na fila de bloqueados;
3. **P**: Imprima o estado atual do sistema;
4. **T**: Imprima o tempo de retorno médio e finalize o simulador.

Além disso, teremos a execução de vários **processos simulados** que irão manipular o valor de uma única variável inteira. O programa de cada processo simulado será formado por uma sequência de instruções e elas serão armazenadas em um vetor (cada posição para uma instrução). As instruções dos processos simulados incluem a atualização da variável inteira, o bloqueio do processo, o término do processo, a criação de um novo processo e a substituição do programa do processo simulado.

O *process manager* (PM), por sua vez, será responsvável pela criação e gerenciamento dos processos simulados. Primeiramente, o *process manager* cria o primeiro processo simulado (id = 0) e o programa desse processo será lido de um arquivo com o nome *init* e armazenado em um *array*. Esse processo simulado será o único criado pelo *process manager*, os outros serão criados caso haja uma instrução de criação de um novo processo no *array* do programa do processo simulado.

Por fim, o processo *reporter* também será criado pelo *process manager* sempre que o usuário digitar o comando **P** para imprimir o estado atual do sistema ou **T** para imprimir o tempo de retorno médio e finalizar o simulador.

2 Implementação

2.1 Estruturas de Dados

Na implementação do simulador, foram utilizadas algumas estruturas de dados fundamentais para o seu funcionamento. Assim sendo, o *process manager* possui 6 estruturas de dados:

1. **Tempo**: Um inteiro que inicia com o valor 0;
2. **CPU**: Simula a execução de um processo simulado que está executando em um dado momento. CPU é uma estrutura que contém alguns dados importantes, tais como:
    - **ponteiroPrograma**: É um ponteiro para o *array* do programa do processo simulado;
    - **contadorPrograma**: Representa o valor atual do contador de programa;

- **valor**: É o valor de uma variável inteira;
- **fatiaTempo**: Representa a fatia de tempo do processo simulado;
- **tempoAtual**: Representa o número de unidades de tempo usadas até o momento.

3. **TabelaPcb**: O Sistema Operacional precisa manter uma tabela (um *array* de estruturas) chamada tabela de processos, na qual cada linha dessa tabela representa um determinado processo e cada coluna possui os campos fundamentais dos processos. Os campos utilizados na estrutura TabelaPcb são: idProcesso, idProcessoPai, ponteiroContadorPrograma, valor, prioridade, estado ('E' para um processo executando, 'P' para pronto ou 'B' para bloqueado), tempoInicio e cpuUsada. Além disso, também é possível visualizá-los na Figura 2, na qual os campos na primeira coluna estão relacionados justamento ao gerenciamento de processos.

Gerenciamento de processo	Gerenciamento de memória	Gerenciamento de arquivo
Registros	Ponteiro para informações sobre o segmento de texto	Diretório-raiz
Contador de programa		Diretório de trabalho
Palavra de estado do programa	Ponteiro para informações sobre o segmento de dados	Descritores de arquivo
Ponteiro da pilha		ID do usuário
Estado do processo	Ponteiro para informações sobre o segmento de pilha	ID do grupo
Prioridade		
Parâmetros de escalonamento		
ID do processo		
Processo pai		
Grupo de processo		
Sinais		
Momento em que um processo foi iniciado		
Tempo de CPU usado		
Tempo de CPU do processo filho		
Tempo do alarme seguinte		

Figura 2: Campos de uma entrada na tabela de processos

4. **estadoPronto**: Uma lista que irá guardar os índices do processos na TabelaPcb que estão no estado pronto (P);

5. **estadoBloqueado**: Uma lista que irá guardar os índices do processos na TabelaPcb que estão no estado bloqueado (B);

6. **estadoExecutando**: Uma lista que irá guardar os índices do processos na TabelaPcb que estão no estado executando (E).

Além disso, será utilizado um vetor de estruturas para guardar o programa de um processo simulado, os seus membros são uma variável do tipo caractere para a instrução, uma variável inteira n e uma string novo_arquivo para guardar o nome do arquivo de um outro processo simulado. E, para finalizar, foi usada uma estrutura chamada No que possui o indice de um processo e um ponteiro para o próximo nó, para representar os nós das listas utilizadas na estrutura do *process manager*.

2.2 Arquivo *commander*.c

O processo *commander* (processo principal) possui duas funções: **lerComando()** e **executarCommander()**. Primeiramente, chamaremos uma função para armazenar o programa do primeiro processo simulado (com o nome *init*) e, em seguida, inicializaremos as estruturas de dados utilizadas (ambas as funções serão explicadas a seguir). Assim, a função **executarCommander()** é chamada.

Na função **executarCommander()**, um *pipe* será criado para a passagem dos comandos para o *process manager*, com isso, criamos o *process manager* (id = 0) . Dessa forma, definimos o estado do processo simulado com o valor 'E' e inserimos o seu índice (nesse caso, 0) na fila de estadoExecutando. Por fim, chamamos a função **lerComando()**.

A função **lerComando()**, por sua vez, terá um laço *while* que será executado enquanto o comando 'T' não for digitado. Dentro do *while*, o processo *commander* lerá comandos da entrada padrão a cada 1 segundo e utilizará o *pipe* para escrevê-los para o outro processo. Desse modo, o *process manager* lê os comandos que são enviados pelo *pipe* e chama uma função para verificar cada um desses comandos (detalhada logo abaixo). Após um comando ser digitado, os processos são escalonados seguindo um algoritmo descrito posteriormente.

2.3 Arquivo *manager*.c

Neste arquivo, temos as seguintes funções: **inserirNaFila()**, **removerDaFila()**, **inicializaEstruturasDeDados()**, **escalonar()**, **executarProximaInstrucao()**, **desbloquearProcesso()**, **criaProcessoReporter()** e **verificaComandoPipe()**.

As funçoes **inserirNaFila()** e **removerDaFila()** servem para manipular as filas estadoPronto, estadoBloqueado, estadoExecutando presentes na estrutura de dados *ProcessManager*. A primeira terá um índice de um determinado processo como parâmetro e irá adicioná-lo no final de uma das filas (dependendo de qual for passada como parâmetro). Já a segunda, irá remover o índice de um processo simulado do ínicio da fila (também depende de qual foi passada como parâmetro).

Para inicializar as estruturas de dados para o *process manager*, utilizamos a função **inicializaEstruturasDeDados()**. Sendo assim, definimos o seu tempo com *0*, na cpu, o ponteiroPrograma aponta para o processo simulado que está em execução, contadorPrograma, valor, fatiaTempo e tempoAtual também iniciam em *0*. Na pcb, dada um determinado indice de um processo simulado, definimos idProcesso e idProcessoPai, inicializamos o ponteiroContadorPrograma, valor inica com *0*, prioridade receberá o valor 20, estado é definido como pronto ('P'), tempo início recebe o valor armazenado em tempo e cpuUsada também inicia com *0*. Além disso, a filas estadoPronto, estadoBloqueado e estadoExecutando também são inicializadas.

A função **escalonar()** lidará com o escalonamento dos processos, visto que ao criar um novo processo, terminar um processo, bloquear um processo ou ocorrer uma interrupção de E/S uma decisão para escalonar esses processos deverá ser tomada. Para isso, foi-se utilizado o Escalonamento por prioridades.

Nesse algoritmo de escalonamento, como cada processo possui um valor que define a sua prioridade, a ideia é que o processo que será executado é aquele que possui prioridade mais alta. Entretanto, para evitar que um processo com uma alta prioridade seja executado durante um grande intervalo de tempo, o escalonador irá abaixar a prioridade do processo que está sendo executado a cada *1* segundo.

Dessa forma, esta função primeiro verifica se há algum processo pronto para ser executado. Se houver, procuramos na fila de estadoPronto um processo que possui prioridade maior que a prioridade do processo que está executando no momento. Se encontrarmos, definimos o estado do processo que estava executando como 'P', inserimos o seu índice em estadoPronto e removemos de estadoExecutando, o novo processo que será executado terá o estado definido como 'E', o seu índice adiconado em estadoExecutando e removido de estadoPronto. Caso não haja nenhum processo com prioridade mais alta que o processo que está executando, apenas diminuímos a prioridade do processo atual em uma unidade.

A função **executarProximaInstrucao()** será chamada em resposta ao comando 'Q'. Com isso, chamos uma outra função para verificar qual a instrução no *array* do programa do processo simulado na posição que o contador de programa está no momento, essa função será explicada quando as funções do arquivo simulado.c forem mencionadas. Após isso, o contadorPrograma, tempo e tempoAtual são incrementados em uma unidade.

Se o comando 'U' for digitado, a função **desbloquearProcesso()** será chamada. Removemos o primeiro processo que está na fila de processos bloqueados. Porém, verificamos se há ou não processos bloqueados.

Em resposta ao comando 'P', temos a função **criaProcessoReporter()**, que irá criar um processo *reporter* (processo filho) e ele será responsável por imprimir o estado atual do sistema.

Já a função **verificaComandoPipe()**, que foi chamada dentro da função **lerComando()** em *commander*.c, apenas verifica qual comando foi lido no *pipe* pelo *process manager* e executa algumas da funções já explicadas acima. Por outro lado, se o comando 'T' for digitado, o processo *reporter* será criado e o simulador será finalizado. Por fim, fatiaTempo e cpuUsada são incrementadas em uma unidade.

2.4 Arquivo simulado.c

Neste arquivo estão presentes funções responsáveis por executar determinadas funções de acordo com uma dada instrução, armazenar o pro-

grama de um processo simulado e verificar qual instrução será executada no momento, são elas: **atualizaVariavel()**, **somaVariavel()**, **subtraiVariavel()**, **bloqueiaProcessoSimulado()**, **terminaProcessoSimulado()**, **criaNovoProcessoSimulado()**, **armazenarPrograma()**, **substituiPrograma()** e **verificarInstrucao()**.

Para manipular o valor da variável inteira de um processo simulado, **atualizaVariavel()** é chamada caso a instrução 'S' for lida e define o seu valor com o valor presente em n, **somaVariavel()** é chamada caso a instrução 'A' for lida e soma o valor de n à ela e **subtraiVariavel()** é chamada caso a instrução 'D' for lida e subtrai o valor de n à ela. O valor tanto da instrução quanto de n estão presentes no vetor do programa do processo simulado na posição do valor do contador de programa atual.

A função **bloqueiaProcessoSimualado()** será chamada em resposta a instrução 'B' e, inicialmente, verifica se a fila de estadoPronto está vazia. Se estiver, o usuário precisa digitar a instrução 'U' para desbloquear o primeiro processo da fila de processo bloqueados. Caso contrário, define o estado do processo em execução como 'B' e remove o índice desse processo de estadoExecutando. O escalonamento é realizado e um novo processo será executado, resultando em uma troca de contexto.

A função **terminaProcessoSimulado()** será chamada caso a instrução 'E' seja executada. Com isso, o processo atual será terminado, a memória será desalocada e a TabelaPcb será atualizada.

Em resposta a instrução 'F', a função **criaNovoProcessoSimulado()** será executada. Nela, o tamanho da tabela pcb será incrementado, e um novo processo simulado (cópia do pai) será criado. O contador de programa do processo pai será definido como n + contadorPrograma. Já o processo filho terá a sua entrada na tabela de processos e será executado uma instrução após 'F', o tempo de início será igual ao tempo atual e o seu índice será adicionado na fila estadoPronto.

A função **armazenarPrograma()** é responsável por armazenar o programa de um processo simulado em um vetor, com uma instrução para cada posição. Os arquivos dos programas que serão executados estarão na pasta

inputs, desse modo, o nome do arquivo que será aberto deverá ser "./inputs/"+ o nome do arquivo passado como argumento para a função. Sendo assim, lemos do arquivo de entrada até o fim, armazenando o valor da instrução no vetor e os valores da variável n e do nome do novo arquivo também, dependendo de qual é a instrução em uma determinada posição do *array*.

Em resposta a instrução 'R', a função **substituiPrograma()** será chamada. Armazenamos o progama desse novo arquivo e fazemos o ponteiroPrograma apontar para esse novo vetor. Além disso, diminuímos o contadorPrograma em uma unidade porque ele será incrementado novamente na função **executaProximaInstrucao()** e definimos o valor da variável inteira como 0.

Por fim, a função **verificarInstrucao()** apenas verifica qual instrução deverá ser executada no momento e chama as funções definidas acima.

2.5 Arquivo *reporter*.c

Finalmente, este arquivo possui apenas a função **imprimeEstadoAtual()** que será chamada caso o comando 'P' ou 'T' seja digitado. Ela apenas imprime o estado atual do sistema. Dessa forma ela irá mostrar alguns dos campos da tabela de processos, separando o processo que está executando e os processos que estão na fila de processos bloqueados e na fila de processos prontos.

3 Referências

TANENBAUM, A. S. **Sistemas Operacionais Modernos** 3 ed. São Paulo: Pearson, 2010.