Universidade Federal de São João del Rei
Campus Tancredo de Almeida Neves
Departamento Ciência da Computação

Trabalho Sistema Operacional Gerenciamento de processos

Aluno: Gabriel Carneiro
Aluno: Felipe Samuel
Aluno: Andre Luiz
Professor: Rafael Sachetto

30 Outubro
2020

Conteúdo

1 Introdução

O problema proposto foi desenvolver um simulador de gerenciamento de processos. Para atender todas as requisições propostas, implementamos o trabalho da seguinte maneira:

- 2 tipos de processos principais: commander, processo manager.
- 2 tipos de processos "secundários": o processo simulado e o processo reporter.
- 5 funções principais de gerenciamento: Criação de processos, substituição do processo atual por outro processo, transição de estados de um processo, escalonamento e troca de contexto.
- 4 chamadas de sistema do Linux: fork(), wait(), pipe(), sleep().

1.1 Como compilar e executar o programa

Para compilar e executar o programa basta usar estes comandos num terminal

```
Para ir até o diretório do trabalho
$ cd caminho/para/a/pasta

Para compilar
$ make

Para usar o teclado como entrada principal:
$ ./process_commander

Para usar um arquivo de texto:
$ ./process_commander < arquivo.txt
```

2 Funcionalidades

O programa consiste em dois elementos principais: o processo commander e o processo manager, a execução do simulador é iniciada pelo processo comandar, e o controle dos processos simulados é feito pelo processo manager.
Dito isso faremos uma breve análise das funções implementadas a seguir, posteriormente aprofundando um pouco mais em sua funcionalidade.

2.1 Processos

2.1.1 Commander

O processo commander controla como será o andamento do processo manager e do processo simulado, ele primeiramente cria um pipe e um processo do tipo manager, após isso ele lê 1 comando por segundo da entrada principal e envia para o processo manager através do pipe, os comandos são os seguintes:

- **Q**: Encerra uma unidade de tempo e aciona a próxima instrução do processo simulado
- **U**: Desbloqueia o primeiro processo na fila de processos bloqueados
- **P**: Cria um processo do tipo reporter que imprime o estado atual do sistema
- **T**: Imprime o tempo de retorno médio, cria um processo do tipo reporter e finaliza o simulador

São usadas 4 chamadas de sistema no processo commander:

- **fork():** Cria uma cópia do processo em execução:
- **wait():** Espera o processo filho terminar a execução
- **pipe():** Envia uma variável para o processo filho ou algum outro processo.
- **sleep():** Faz o processo ficar "parado", por uma quantidade n de segundos

2.1.2 Manager

Simula 5 funções de gerenciamento de processos:

- **Criação de um processo:** Lê um arquivo com o programa e guarda as instruções, tamanho do arquivo e nome do arquivo na struct CPU e insere-a na tabela PCB (que é o array list pbc_list).
- **Substituição do processo atual por outro processo(troca de imagem):** Substitui o código do programa atual pelo código do programa especificado
- **Transição de estados de um processo:** Um processo sempre está em um dos três possíveis estados:
  - **Executando(E):** É a condição do processo que está atualmente na CPU.
  - **Pronto(P):** É a condição no qual um processo não esta em execução mas pode ser escalonado a qualquer momento.
  - **Bloqueado(B):** É o estado de um processo que estava em execução e executou a instrução B, fazendo com que sua execução seja interrompida e seu estado seja salvo na memoria e o processo seja enviado para a fila de bloqueados.
- **Escalonamento:** O escalonador escolhe quem deve executar a partir da lista de prontos com uma fatia de tempo a cada 2 segundos por processo.
- **Troca de contexto:** A troca de contexto ocorre quando o processo simulado usa a instrução F (seção 2.1.3 item 6), ela cria uma cópia do processo em execução (pai) e insere na tabela PCB, e começa a executar esse processo filho que é a exata cópia do processo pai.

2.1.3 Simulado

Representa cada processo simulado em um programa, que opera o valor de uma única variável inteira, ou seja o estado de uma variável em um determinado instante de tempo T equivale ao valor da sua variável inteira e o valor de seu contador de programa. Neste caso o processo é armazenado uma sequencia de sete instruções:

1. **S** n: Atualiza o valor da variável inteira para n.
2. **A** n: Soma n na variável inteira

3. **D** n: Subtrai n Na variável inteira
4. **B**: Bloqueia o processo simulado
5. **E**: Termina o processo simulado
6. **F** n: Cria um novo processo simulado, sendo este a copia exata do processo pai.
7. **R** arquivo: Substitui o processo atual pelo programa no arquivo com nome arquivo.

2.1.4 Reporter:

Imprime o estado em que o sistema se encontra.

```
********************************************************************************
********************************************************************************
TEMPO ATUAL:
PROCESSO EXECUTANDO:
pid     ppid    valor   tempo inicio    CPU usada ate agora     nome processo

PROCESSO BLOQUEADOS:
PROCESSO PRONTOS:
********************************************************************************
```

3 Estruturas e funções

3.1 Estruturas de dados

3.1.1 Programa

```
typedef struct {
    char comando;
    char *valor;
} programa;
```

O código do programa fica armazenado na struct programa, a variável char comando guarda o comando e a variável char *valor guarda os argumentos do comando.

3.1.2 CPU

```
typedef struct{
    programa *processo;
    char *nome_arquivo;
    int tamanho;
    int chave;
    int contador;
    int tempo_total;
    int tempo_atual;
} CPU;
```

Essa estrutura armazena as informações cruciais para o funcionamento correto do processo, e a execução do processo simulado fica confinada à ela. O código do programa fica armazenado na struct programa *processo, o nome do arquivo na variável char *nome_arquivo, o número de instruções do programa fica na variável tamanho, o valor inteiro que representa o estado atual num instante de tempo é a variável chave, a variável int contador guarda qual a instrução atual do programa, o int tempo_total armazena a quantidade de tempo total que o processo ficou na CPU e a variável inteiro tempo_atual armazena o tempo que o programa está executando na fatia de tempo atual.

3.1.3 Tabela PCB

```
typedef struct{
    CPU *programa;
    int pid;
    int ppid;
    int estado;
    int *contador;
    int tempo_inicio;
} tabela_pcb;
```

Essa estrutura armazena todas as informações do processo, inclusive a estrutura CPU que contém o programa. A tabela_pcb armazena uma struct CPU que aponta para onde está armazenado um processo, o inteiro pid que é o id do processo, o inteiro ppid que é o id do processo pai, um inteiro estado que representa o estado atual do precesso, um inteiro contador que aponta para o contador de programa do processo e o inteiro que armazena o tempo de inicio do processo.

3.1.4 Array list

```
typedef struct {
    tabela_pcb *pcb;
    int tamanho;
    tabela_pcb *pcb_atual;
} array_list;
```

Essa estrutura armazena um lista de arranjo de todos os programas que estão nos 3 possíveis estados de execução (executando, pronto e bloqueado), foi usado um array por ser a melhor opção quando se tem que fazer várias consultas por causa princípio de localidade.

3.1.5 Fila

```
typedef struct {
     CPU *chave;
     struct fila *prox;
} fila;
```

Essa estrutura armazena uma fila de programas, ela é usada no programa para armazenar a fila de processos prontos e de processos bloqueados.

3.2 Variáveis Globais

3.2.1 tempo

É um inteiro inicializado com 0 que armazena o tempo de execução do simulador.

3.2.2 executando

É uma struct CPU que armazena o processo que está atualmente em execução.

3.2.3 estado_pronto

É uma fila que armazena os processos que estão no estado pronto.

3.2.4 estado_bloqueado

É uma fila que armazena os processos que estão no estado bloqueado.

3.2.5 pcb_list

É uma lista de arranjos que armazena a tabela PCB.

3.2.6 multiplicador_pcb

É um inteiro que serve para ser multiplicado pelo tamanho da tabela PCB quando ela estiver cheia.

3.3 Funções

3.3.1 void printa_processo()

Printa informações básicas do processo. A complexidade desta função é O(1).

3.3.2 void insere_pcb(CPU *c)

Insere o processo colocado como parâmetro da função na tabela PCB, caso ela esteja vazia ele é inserido na posição 0, caso ela n esteja vazia mas também não esteja cheia, o processo é inserido na primeira posição vaga do vetor, e no caso da tabela PCB estar cheia, ela realoca seu tamanho usando a seguinte formula: tamanho_atual * multiplicado_pcb (que aumenta em 1

toda vez que a tabela é realocada). A complexidade desta função em seu pior caso é O(1).

3.3.3 void remove_pcb(CPU *c)

Procura pelo enderenço do processo usado colocado como parâmetro da função na tabela PCB, e caso seja encontrado, ele é removido, a memoria alocada é liberada e os elementos seguintes na tabela são movidos para a posição correta. A complexidade desta função em seu pior caso é O(n).

3.3.4 tabela_pcb *busca_pcb(CPU *c)

Procura pelo endereço do processo colocado como parâmetro da função na tabela PCB, e caso seja encontrado, é retornado o elemento, caso não seja encontrado, é retornado o programa que está atualmente em execução. A complexidade desta função em seu pior caso é O(n).

3.3.5 int line_count(char *fileName)

Conta o número de linhas no arquivo, para ser armazenado na variável tamanho na struct CPU. A complexidade desta função em seu pior caso é O(n)

3.3.6 void enfilera(fila **f, CPU *chave)

Insere um processo em uma fila. A complexidade desta função em seu pior caso é O(n).

3.3.7 fila *desenfilera(fila **f)

Remove o primeiro elemento de uma fila e retorna ele. A complexidade desta função em seu pior caso é O(1).

3.3.8 void printa_fila(fila *f)

Imprime os elementos que estão em uma fila. A complexidade desta função em seu pior caso é O(n).

3.3.9 void printa_comando(programa p)

Imprime uma determinada instrução de um programa. A complexidade desta função em seu pior caso é O(1).

3.3.10 CPU cria_processo(char *prog)

Lê o arquivo com o nome especificado como argumento da função, e armazena informações numa struct CPU, mais informações sobre o que é guardado nessa estrutura estão na seção 3.1.2. A complexidade desta função em seu pior caso é O(n^2), sendo n o número de linhas no arquivo e m o tamanho do argumento de cada instrução

3.3.11 void bloqueia_processo()

Bloqueia o processo que está atualmente em execução e insere ele na fila de bloqueados. A complexidade desta função em seu pior caso é O(n).

3.3.12 void desbloqueia_processo()

Remove o primeiro processo na fila de bloqueados e insere na fila de processos prontos. A complexidade desta função em seu pior caso é O(1).

3.3.13 void troca_de_imagem(CPU *c)

Substitui o programa em execução pelo processo especificado como parâmetro da função. A complexidade desta função em seu pior caso é O(n).

3.3.14 void troca_de_contexto()

Faz uma cópia do processo em execução e insere na tabela PCB, essa nova cópia é então executada e o processo pai é inserido na fila de processos prontos. A complexidade desta função em seu pior caso é O(n).

3.3.15 void escalona_processo()

Caso o processo acabe sua execução ou exceda o limite de tempo de sua fatia, ele é inserido na fila de processos prontos e o primeiro processo na fila de processos prontos é executado. A complexidade desta função em seu pior caso é O(n).

3.3.16 void encerra_processo()

Encerra a execução do processo atual, desaloca a memoria, remove o processo da tabela PCB e chama a função de escalonamento para que um novo processo seja executado. A complexidade desta função em seu pior caso é O(n).

3.3.17 void reporter()

Imprime o estado atual do sistema, como descrito na seção 2.1.4. A complexidade desta função em seu pior caso é O(n).

3.3.18 void substitui_processo()

Substitui o código do processo atual por outro código, a diferença dessa função para a troca de imagem é que essa função troca o código do programa até mesmo na tabela PCB, enquanto a troca de imagem apenas troca o processo que está em execução. A complexidade desta função em seu pior caso é O(n^2), pois ela chama a função cria processo.

3.3.19 void executa_processo()

Essa é a função responsável pela execução de um processo simulado, ela que interpreta os comando e chama as respectivas funções. A complexidade desta função em seu pior caso é O(n^2), pois ela pode executar todas as funções do programa.

3.3.20 int main()

Essa é a função principal, no main é que é feito a leitura do comando enviado no pipe pelo processo comander e é nele que este comando é interpretado. O main chama a função executa_processo em resposta à instrução Q, chama a função desbloqueia_processo em resposta à instrução U, cria um processo do tipo reporter em resposta à instrução P e em resposta à instrução T ele cria um processo reporter e finaliza o simulador após o fim deste processo.

4 Analise de dados

A seguir, os dados avaliados de acordo com o tempo de execução (lembrando que cada processo tem uma fatia de tempo de 2 segundos):

```
Arquivo init
S 1000
A 23
F 1
R programa_a
A 21
D 20
E

Arquivo programa_a
S 100
A 19
A 20
B
A 21
E


Q
P
********************************************************************************
Estado do sistema:
********************************************************************************
TEMPO ATUAL: 1
PROCESSO EXECUTANDO:
pid     ppid    valor   tempo inicio    CPU usada ate agora     nome processo
0       0       1000    0               1                       init
PROCESSO BLOQUEADOS:
PROCESSO PRONTOS:
********************************************************************************

Q
Q - Nesse momento o algoritmo troca de contexto em decorrência da instrução F
P
********************************************************************************
Estado do sistema:
********************************************************************************
TEMPO ATUAL: 3
PROCESSO EXECUTANDO:
pid     ppid    valor   tempo inicio    CPU usada ate agora     nome processo
1       0       1023    3               3                       init
PROCESSO BLOQUEADOS:
PROCESSO PRONTOS:
0       0       1023    0               3                       init
********************************************************************************
```

```
Q - Substitui o código do programa init (có
P
********************************************************************************
Estado do sistema:
********************************************************************************
TEMPO ATUAL: 4
PROCESSO EXECUTANDO:
pid     ppid    valor   tempo inicio    CPU usada ate agora     nome processo
1       0       0       3               0                       programa_a
PROCESSO BLOQUEADOS:
PROCESSO PRONTOS:
0       0       1023    0               3                       init
********************************************************************************

Q
Q
Q
Q
Q
Q - Neste momento o processo do programa_a é bloqueado
P

********************************************************************************
Estado do sistema:
********************************************************************************
TEMPO ATUAL: 10
PROCESSO EXECUTANDO:
pid     ppid    valor   tempo inicio    CPU usada ate agora     nome processo
0       0       1024    0               5                       init
PROCESSO BLOQUEADOS:
1       0       139     3               4                       programa_a
PROCESSO PRONTOS:
********************************************************************************

Q - Termina a execucão do programa init
P
********************************************************************************
Estado do sistema:
********************************************************************************
TEMPO ATUAL: 11
PROCESSO EXECUTANDO:
pid     ppid    valor   tempo inicio    CPU usada ate agora     nome processo
PROCESSO BLOQUEADOS:
1       0       139     3               4                       programa_a
PROCESSO PRONTOS:
********************************************************************************
```

```
U - É desbloqueado o primeiro processo na fila de bloqueados
P
********************************************************************************
Estado do sistema:
********************************************************************************
TEMPO ATUAL: 11
PROCESSO EXECUTANDO:
pid     ppid    valor   tempo inicio    CPU usada ate agora     nome processo
PROCESSO BLOQUEADOS:
PROCESSO PRONTOS:
1       0       139     3               4                       programa_a
********************************************************************************

Q - O programa_a sai da fila de prontos e volta a ser executado
P
********************************************************************************
Estado do sistema:
********************************************************************************
TEMPO ATUAL: 12
PROCESSO EXECUTANDO:
pid     ppid    valor   tempo inicio    CPU usada ate agora     nome processo
1       0       160     3               5                       programa_a
PROCESSO BLOQUEADOS:
PROCESSO PRONTOS:
********************************************************************************

Q - O programa_a termina sua execução
P
********************************************************************************
Estado do sistema:
********************************************************************************
TEMPO ATUAL: 13
PROCESSO EXECUTANDO:
pid     ppid    valor   tempo inicio    CPU usada ate agora     nome processo
PROCESSO BLOQUEADOS:
PROCESSO PRONTOS:
********************************************************************************

T - Fim do simulador de processos
********************************************************************************
Estado do sistema:
********************************************************************************
TEMPO ATUAL: 13
PROCESSO EXECUTANDO:
pid     ppid    valor   tempo inicio    CPU usada ate agora     nome processo
PROCESSO BLOQUEADOS:
PROCESSO PRONTOS:
********************************************************************************
```

5 Conclusão

O objetivo deste trabalho foi construir um algoritmo de gerenciamento de recursos, porém houve etapas de difícil execução, principalmente na troca de contexto onde não é tão trivial a abstração e implementação na linguagem C, no decorrer do desenvolvimento conseguimos atingir o objetivo e fazer análise dos dados obtidos através de testes previamente estipulados. Portanto, este trabalho abre uma uma oportunidade de simular e compreender os processos executados no Linux.

6 Bibliografia

Todo material utilizado foi obtido através do manual do LINUX, nenhum outro material foi consultado para o desenvolvimento do trabalho.