Relatório: Implementação do Pseudo-SO

Andressa Valadares, Luigi Gil e Roberto Ono

I. Introdução

Implementamos um pseudo-SO multiprogramado em linguagem Python (v2.7), composto por um Gerenciador de Processos, por um Gerenciador de Memória e por um Gerenciador de Recursos.

O gerenciador de processos é capaz de agrupar os processos em quatro níveis de prioridades. O gerenciador de memória assegura que um processo não acesse as regiões de memória de um outro processo. E o gerenciador de recursos é o responsável por administrar a alocação e a liberação de todos os recursos disponíveis, garantindo uso exclusivo dos mesmos. Os detalhes da implementação desse pseudo-SO são descritos nas próximas seções.

II. Pseudo-SO

O pseudo-so desenvolvido pelo grupo possui cinco classes, sendo quatro delas módulos gerenciadores de memória, fila, recursos e processos. A quinta classe corresponde à classe dos processos do sistema.

Mais dois arquivos compõem o código desse trabalho, o *dispatcher* é o arquivo de execução do SO e o *FileReader* é um pacote com funções auxiliares para leitura de arquivos.

A. ProcessController.py

O módulo *ProcessController.py* realiza o gerenciamento dos processos. É nele que estão as classes e estruturas de dados relacionadas ao processo. Basicamente, mantém funções específicas do processo como mudança de estados(`restartProcessBlock`, `startProcessBlock`, `suspendProcessBlock`, `termProcessBlock`), inserção (`pushProcessBlock`) e retirada (`popProcessBlock`)de um processo da lista de processo. Além da criação do bloco do processo (`createProcessBlock`) e os métodos de exibição dos dados do processo (`updateTable`).

Nos baseamos no comando *top* para criar um interface simplificada e que atendesse aos nossos requisitos:

B. ListManager.py

Classe responsável pela gerência de filas de processos no sistema. A partir das prioridades de cada processo é feita uma alocação destes em filas de prioridades.

As rotinas da lista de despacho são realizadas por esta classe. A rotina da lista principal simula a lista de processos a serem executadas a partir de um arquivo de texto. Os processos são retirados da lista principal, lidos e, de acordo com a prioridade, colocado na lista de processos de tempo real ou lista de processos de usuários.

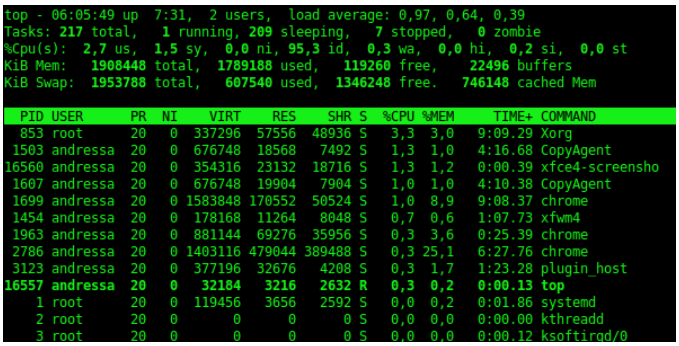

Fig. 1. Screenshot da execução do *top*

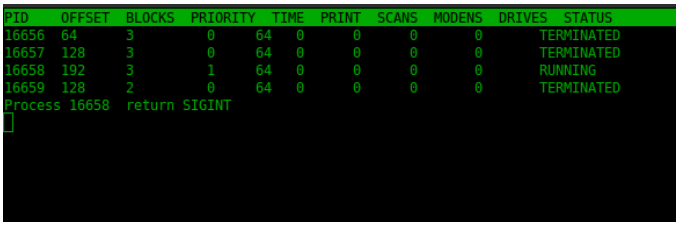

Fig. 2. Screenshot da execução do pseudo-SO

Para a implementação da rotina da lista principal encontrou-se o problema de chegarem processos requerendo mais recursos do que a máquina podia suportar. Caso tal situação ocorra, estes processos são ignorados e retirados da fila de processos a serem executados.

A rotina de lista usuário funciona de modo análogo à rotina anterior. Primeiramente, tenta-se alocar um bloco de memória para o processo. Caso a máquina possua recursos suficientes para a alocação do processo, a alocação é feita. O processo é então multiplexado para a fila de prioridades respectiva.

C. MemoryManager.py

Esse módulo provê uma interface de abstração de memória RAM foi implementado em `MemoryManager.py`.

O gerenciador de memória é uma classe que define os métodos a serem utilizados pelo $dispatcher$ e $ListManager$ para alocar (`allocate`), verificar se existe espaço disponível (`check`), liberar espaço de memória (`free`).

D. ResourceManager.py

Está implementado em `ResourceManager.py`, é uma classe que possui os métodos para verificação de erros (`check`), alocação (`alloc`) e liberação (`free`) dos recursos de E/S para os processos. Seus métodos também são utilizados em $ListManager.py$.

E. FileReader.py

Arquivo que contém função auxiliar para realizar leitura de arquivo contendo informações dos processos a serem executados pelo sistema.

A função `process_list_file` recebe uma string contendo o nome de um arquivo a ser aberto. Cada linha do arquivo é salva numa lista e então a lista é retornada.

F. Process.cpp

Arquivo responsável por prover a implementação de uma classe de processo a fim de simular um sistema operacional.

Uma função `main` engloba o ciclo de vida de um processo. O processo mostra uma mensagem na tela avisando de seu início. Em seguida as instruções a serem executadas são carregadas no processo.

Feito esse processo, faz-se um gancho de alguns sinais a uma função de *callback* que multiplexa o sinal e seta-se uma *flag* correspondente ao sinal enviado.

Por fim, o processo executa uma função que verifica as *flags* de sinal, a fim de identificar se algum sinal foi acionado. Caso não tenha ocorrido acionamento de *flag*, é feita a impressão na tela a execução de uma instrução.

Não houve muita dificuldade na implementação desse módulo, uma vez que tais conceitos foram apresentados e treinados em laboratório no início do semestre, inclusive implementação.

G. dispatcher.py

Esse módulo é onde está a `main()`, é quem realiza efetivamente o escalonamento dos processos.

Inicialmente define quais os valores para cada recurso, instancia os módulos, lê o arquivo que possui os dados de cada processo (`FileReader`) e inicializa a memória. Verifica-se a cada iteração do loop, que dura um $quantum$, a existência de processo em execução. Caso exista checamos quanto tempo ainda lhe resta para terminar - caso tenha concluído e o tempo de uso da CPU é menor que 0, os recursos são desalocados e a memória liberada. Ao contrário verifica a sua prioridade, adiciona uma penalidade (+1) e insere na lista de processos (`list_manager`). Se não houver nenhum processo executando, as listas são liberadas por ordem de prioridade, quanto menor o valor maior a preferência. Em seguida comparamos se o processo foi suspenso e o reiniciamos ou se é preciso apenas de um início comum. Uma das grandes dificuldades foi colocar em prática esse algoritmo de escolamento, já que deveria garantir a execução do processo (sem starvation), e existiam muitos fatores a se considerar como os privilégios.

REFERENCIAS

[1] R. W. Lucky, Automatic equalization for digital communication, Bell Syst. Tech. J., vol. 44, no. 4, pp. 547D588, Apr. 1965.