Universidade de Brasília
Sistemas Operacionais
Aletéia Favacho

Projeto Final SO

Eduardo Schuabb Duarte - 11/0010876
Maximillian Fan Xavier - 12/0153271
Rafael Dias da Costa - 12/0133253

6 de Dezembro de 2017

1 Descrição das ferramentas/linguagens utilizadas

Para o projeto final da disciplina de Sistemas Operacionais, que consiste no desenvolvimento de um pseudo SO, os integrantes concordaram em desenvolver o código em ambiente Unix no sistema operacional Ubuntu 16.04 (64 bits) com a linguagem Python em sua versão 3.5.2.

O código foi desenvolvido nos editores de texto Sublime Text (versão 3.0) e Atom (versão 1.18), e o versionamento do código foi realizado utilizando a plataforma GitHub.

2 Descrição teórica/prática da solução dada

Neste projeto tínhamos o objetivo de implementar um pseudo sistema operacional, operando com diferentes módulos em sua estrutura, sendo eles um gerenciador de memória, gerenciador de arquivos, gerenciador de E/S, gerenciador de filas e gerenciador de recursos.

Para podermos operar com diferentes gerenciadores, implementamos cada módulo de forma separada, sendo aglomerados em duas classes (Despachante e Gerenciador de Filas) que realizam a maioria das operações no projeto. Cada módulo consiste de uma classe que é instanciada e que possui métodos que operam sobre os processos ou arquivos, de acordo com a sua finalidade.

A estrutura do programa está representada na Figura 1:

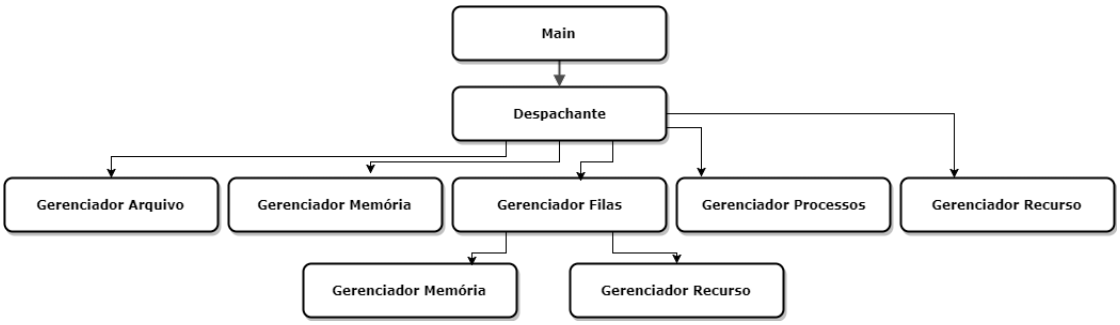

Figura 1: Estrutura Geral

Em nosso arquivo de execução principal (*main*), temos um objeto da classe Despachante, que possui uma instância de cada um dos módulos para poder inicializar o ambiente com os valores de execução (memória, blocos no disco, etc).

Esta classe possui um método denominado *startSO* que realiza a inicialização do software. Neste método realiza-se, na seguinte ordem, a leitura dos arquivos .txt contendo os processos e os arquivos a serem processados (alocando os dados lidos em listas), instancia-se as *threads* que realizam a movimentação dos processos de tempo real e usuário entre a fila global e as filas dedicadas a cada classe de processo, a chamada do método de escalonamento/execução dos processos e a chamada do método que realiza a execução das operações sobre os arquivos.

Na classe Gerenciador Filas realiza-se os procedimentos de escalonamento e instanciamento das threads representando os processos em execução. Por esse motivo, a mesma possui duas instâncias específicas para essas execuções, sendo elas um módulo Gerenciador de Memória e um módulo Gerenciador de Recursos, ambas sendo utilizadas no momento de escalonar os processos. Essas instâncias realizam as operações, enquanto as instâncias pertencentes à Despachante servem para inicialização do ambiente de execução.

3 Descrição das principais dificuldades encontradas durante a implementação

3.1 Gerenciamento de Arquivos

- Problema 1 - A primeira dificuldade estava relacionada a decisão de como definir as estruturas que todo o gerenciamento de arquivos necessitaria.
- Problema 2 - Outra dificuldade estava relacionada a definição dos atributos da classe arquivos e como os arquivos se adequariam aquela classe.
- Problema 3 - Uma última dificuldade relacionada a gerência de arquivos era o fato de quando um arquivo era processado, precisava-se verificar se o processo daquele arquivo existia.

3.2 Gerenciamento de Filas

- Problema 1 - O principal e único problema referente ao gerenciamento das filas é de que forma podemos proceder para escalonar os processos de usuário com diferentes prioridades, e obedecendo aos processos de tempo real que possuem prioridade máxima, podendo parar a execução dos processos de usuário em troca de sua execução.

4 Soluções para as dificuldades encontradas

4.1 Gerenciamento de Arquivos

4.1.1 Solução 1

Foi necessário criar três vetores:

- vetor_arquivos_disco. Esse vetor é um vetor de instâncias de arquivos e, nesse caso, esse vetor é responsável por obter os arquivos que já estão salvos em disco.
- posicoesDisco. Esse vetor simula as posições do disco. Ou seja, se um disco possui 10 posições, então o tamanho máximo desse vetor é de 10 posições. Os arquivos que são salvos no disco acabam sendo inseridos nesse vetor nas posições em que eles "ocupariam" em disco.
- vetor_arquivos_processos. Esse vetor é semelhante ao vetor_arquivos_disco, porém nesse caso temos os arquivos que ainda serão processados.

4.1.2 Solução 2

Para esta situação, segue a solução:

- Arquivos que já estão salvos no disco quando o SO inicia, possuem ID do processo igual a -1.
- Arquivos que serão ainda salvos no disco possuem bloco inicial igual a -1.
- Arquivos que serão deletados do disco possuem bloco inicial igual a -2 e tamanho de bloco igual a -1.

4.1.3 Solução 3

Foi necessário obter o vetor que possui todos os processos que são processados pelo SO, logo, reorganizamos a lógica da gerência de arquivos para poder incluir a lista de processos.

4.2 Gerenciamento de Filas

4.2.1 Solução 1

Para resolvermos o problema de escalonar as *threads* de forma sincronizada estabelecemos três níveis de prioridade para os processos de usuário, sendo elas as prioridades 1, 2 e 3 (processos de prioridade 4 ou mais são ajustados para prioridade 3), lembrando que os processos de tempo real possuem prioridade 0. São executadas duas *threads* especiais que realizam a movimentação dos processos da fila global (que possuem memória livre e recursos para sua execução) para a fila específica da classe.

Ao chegar na fila global, o processo é movimentado para a sua fila específica. Ao chegar na fila específica, caso seja um processo de tempo real, o mesmo é instanciado em uma *thread* de execução com auxílio do objeto Lock (o objeto Lock é a implementação em alto nível de um monitor para o ambiente *multithreading* em Python). Esse objeto permite, dentre algumas operações, as operações *acquire* (similar ao UP do semáforo convencional) e *release* (similar ao DOWN do semáforo convencional). Ao inicializar a *thread* de execução, uma operação de *acquire* é realizada, podendo ser bloqueada, caso exista alguma *thread* de tempo real já em execução, ou será executada e bloqueará o acesso para outras *threads* até que a mesma realiza uma chamada *release*.

Para processos de usuário utilizamos um contador global de instruções exclusivo para processos de usuário. A cada instrução de processo de usuário executada esse contador é incrementado e a cada incremento os seguintes casos são analisados para realizar o escalonamento (na respetiva ordem, simulando uma prática de *aging*):

- Processo de prioridade 1 executado
  - Verifica-se se o contador é divisível por 3 e se for realiza-se a busca por algum processo de prioridade 2 que esteja esperando para ser executado
  - Caso não haja processos de prioridade 2 devido ao contador, verifica-se se o contador é divisível por 4 e se for realiza-se a busca por algum processo de prioridade 3 que esteja esperando para ser executado
  - Caso não haja processos de prioridade 2 ou 3 aguardando devido ao contador, busca-se por um processo de prioridade 1 que esteja aguardando
  - Caso não haja processos de prioridade 1 e o contador não seja divisível por 3, verifica-se se existe algum processo de prioridade 2 aguardando
  - Caso não haja processos de prioridade 2 e o contador não seja divisível por 4, verifica-se se existe algum processo de prioridade 3 aguardando
- Processo de prioridade 2 executado
  - Verifica-se se o contador é divisível por 4 e se for realiza-se a busca por algum processo de prioridade 3 que esteja esperando para ser executado
  - Caso o contador não esteja no valor certo para processo de prioridade 3, verifica-se se existe algum processo de prioridade 1 aguardando para ser executado
  - Caso não haja processos de prioridade 1 aguardando e o contador não esteja no valor certo para processo de prioridade 3, verifica-se se existe algum processo de prioridade 2 aguardando para ser executado
  - Caso não haja processos de prioridade 1 ou 2 aguardando e contador não está no valor correto para processos de prioridade 3, verifica-se somente agora se existe ainda algum processo de prioridade 3 aguardando para ser executado
- Processo de prioridade 3 executado
  - Verifica-se se existe algum processo de prioridade 1 aguardando para ser executado
  - Verifica-se se existe algum processo de prioridade 2 aguardando para ser executado
  - Verifica-se se existe algum processo de prioridade 3 aguardando para ser executado

5 Papel desempenhado por cada membro

O papel desempenhado por cada membro no trabalho está descrito nas subseções abaixo:

5.1 Eduardo

Toda a parte relacionada a gerenciamento e manipulação de arquivos. Desde a leitura dos arquivos de entrada até a execução dos arquivos em si. Todo o processo pode ser observado nas funções:

- Métodos lendoArquivosFiles e runFiles da `ClassDespachante.py`
- Toda a `ClassArquivo.py`
- Toda a `ClassGerenciadorArquivo.py`

5.2 Maximillian

As operações de escalonamento e execução dos processos em *threads* sincronizadas, alocação dos recursos (dispositivos) para processos de usuário, inicialização do software na classe Despachante e ajustes na lista de processos. Tais informações podem ser visualizadas em:

- Toda a `ClassGerenciadorFilas.py`
- Toda a `ClassGerenciadorRecurso.py`
- método startSO da `ClassDespachante.py`
- Métodos removeProcessosTempoInicializacaoIgual e updatePIDs na da `ClassDespachante.py`

5.3 Rafael

As operações relacionadas ao gerenciamento de memória, tanto verificação de disponibilidade, validação de requisição, e manipulação de *offsets*. Funções de validação de requisição de recursos. Montagem e separação de lista de processos, funcionalidade de *parse* de argumentos do programa e modularização do código. Atividades presentes nos seguintes fontes:

- Toda a `ClassGerenciadorMemoria.py`
- Método verificaRequisicaoRecursos em `ClassGerenciadorRecurso.py`
- Toda a `ClassGerenciadorProcesso.py`
- *Parse* de argumentos na `main.py` e impressões da `ClassInfo.py`

Obs.: Cada um dos integrantes contribuiu na etapa final do projeto realizando revisão de código (lógica, tratamentos de exceção e comentários).

6 Bibliografia

- Documentação da *lib* Python `threading`:

  https://docs.python.org/2/library/threading.html

- Referência de *Starvation* e *Aging*:

  http://www.geeksforgeeks.org/starvation-aging-operating-systems/

- Exemplos de uso de `Locks` em Python:

  https://www.laurentluce.com/posts/python-threads-synchronization-locks-rlocks-semaphores-conditions-events-and-queues/comment-page-1/

- Escalonamento de processos RT:

  http://ctd.ifsp.edu.br/ marcio.andrey/images/Escalonamento-Processos-IFSP.pdf