
# Fundamentos de Sistemas Operacionais

## Especificação do Trabalho Prático (Implementação)

## Profa.: Aletéia Patrícia Favacho de Araújo

### *Orientações Gerais*

O trabalho de implementação da disciplina de FSO, a ser desenvolvido em grupo com três (03) componentes, os mesmos do seminário, compreenderá as seguintes fases:

- Estudo teórico relacionado ao assunto do trabalho;
- Apresentação da solução teórica dada ao problema;
- Implementação da solução proposta;
- Apresentação e explicação detalhada do código-fonte implementado;
- Relatório explicando o processo de construção e o uso da aplicação.

### *Orientações Específicas*

#### 1 Problema

Implementação de um **pseudo-SO** multiprogramado, composto por um **Gerenciador de Processos,** por um **Gerenciador de Memória,** por um **Gerenciador de E/S** e por um **Gerenciador de Arquivos**. O gerenciador de processos deve ser capaz de agrupar os processos em quatro níveis de prioridades. O gerenciador de memória deve garantir que um processo não acesse as regiões de memória de um outro processo. O gerenciador de E/S deve ser responsável por administrar a alocação e a liberação de todos os recursos disponíveis, garantindo uso exclusivo dos mesmos. E o gerenciador de arquivos deve permitir que os processos possam criar e deletar arquivos, de acordo com o modelo de alocação determinado. Os detalhes para a implementação desse pseudo-SO são descritos nas próximas seções.

#### 1.1 Estrutura das Filas

O programa deve ter duas filas de prioridades distintas: a fila de processos de tempo real e a fila de processos de usuários. Processos de tempo real entram para a fila de maior prioridade, sendo gerenciados pela política de escalonamento FIFO (*First In First Out*), sem preempção.

Processos de usuário devem utilizar múltiplas filas de prioridades com realimentação. Para isso, devem ser mantidas **três** filas com prioridades distintas. Para evitar *starvation*, o sistema operacional deve modificar a prioridade dos processos executados mais frequentemente e/ou utilizar uma técnica de envelhecimento (*aging*). **Quanto menor** for o valor da prioridade atribuída a um processo, **maior** será a sua

prioridade no escalonamento. Dessa forma, os processos de tempo real, que são os mais prioritários, terão prioridade definida como **0 (zero)**. Além disso, é importante destacar que processos de usuário podem ser preemptados e o *quantum* deve ser definido de **1 milissegundo**.

As filas devem suportar no máximo **1000 processos**. Portanto, recomenda-se utilizar uma fila "global", que permita avaliar os recursos disponíveis antes da execução e que facilite classificar o tipo de processo. A Figura 1 ilustra o esquema indicado.

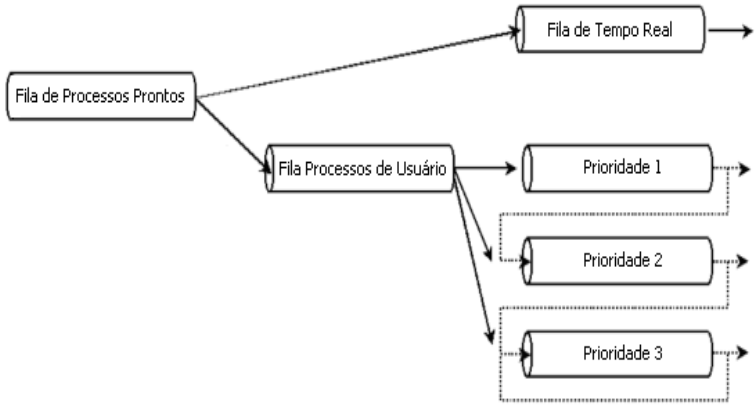


Figura 1 – Estrutura de Filas para o Pseudo-SO.

### 1.2 Estrutura de Memória

A alocação de memória deve ser implementada como um conjunto de blocos contíguos, onde cada bloco equivale uma palavra da memória real.

Cada processo deve alocar um segmento contíguo de memória, o qual permanecerá alocado durante toda a execução do processo. Deve-se notar que não é necessário a implementação de memória virtual, *swap*, nem sistema de paginação. Portanto, não é necessário gerenciar a memória, apenas verificar a disponibilidade de recursos antes de iniciar um processo.

Deve ser utilizado um tamanho fixo de memória de 1024 blocos. Dessa quantidade, 64 blocos devem ser reservados para processos de tempo-real e os 960 blocos restantes devem ser compartilhados entre os processos de usuário. A Figura 2 ilustra o caso onde cada bloco de memória possui 1 MB.

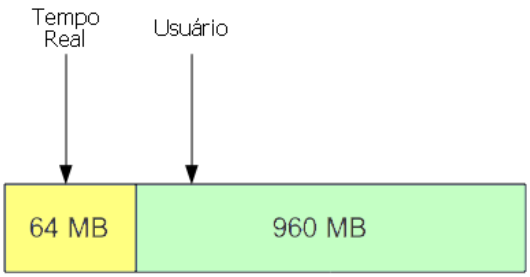


Figura 2 – Modelo de memória com região reservada aos processos de tempo real e aos processos de usuário.

### 1.3 Estrutura dos Recursos Disponíveis

O pseudo-SO deve, além de escalonar o processo no compartilhamento da CPU, e gerenciar o espaço de memória de acordo com as áreas reservadas, ele deve gerenciar os seguintes recursos:

- 1 *scanner*
- 2 impressoras
- 1 modem
- 2 dispositivos SATA

Todos os processos, com exceção daqueles de tempo-real podem obter qualquer um desses recursos. O pseudo-SO deve garantir que cada recurso seja alocado para um proceso por vez. Portanto, não há preempção na alocação dos dispositivos de E/S. Assim, processos de tempo-real não precisam de recursos de I/O.

### 1.4 Estrutura do Sistema de Arquivos

O pseudo-SO deve permitir que cada processo possa criar e deletar arquivos. Na criação de um arquivo, os dados devem ficar residentes no disco, mesmo após o encerramento do processo. O sistema de arquivos fará a alocação por meio do método de alocação contígua. Contudo, o arquivo deve ser tratado como uma unidade de manipulação. Além disso, para que possamos ter a mesma tomada de decisão, vamos considerar que o algoritmo a ser usado no armazenamento do disco (embora não seja usual ser usado na alocação do disco) seja o *first-fit* (sempre considerando a busca a partir do primeiro bloco do disco).

Além disso, o sistema de arquivos deve garantir que os processos de tempo real possam criar (se tiver espaço) e deletar qualquer arquivo (mesmo que não tenha sido criado pelo processo). Por outro lado, os processos comuns do usuário, só podem deletar arquivos que tenham sido criados por eles, e podem criar quantos arquivos desejarem, no tamanho que for solicitado (se houver espaço suficiente).

O sistema de arquivos terá como entrada um arquivo com extensão .txt, que contém a quantidade total de blocos no disco, a especificação dos segmentos ocupados por cada arquivo, as operações a serem realizadas por cada processo.

Assim, após o pseudo-SO executar todos os processos, ele deve mostrar na tela do computador um mapa com a atual ocupação do disco, descrevendo quais arquivos estão em cada bloco, e quais são os blocos vazios (identificados por 0).

## 2 Estrutura do Programa

Conforme descrito anteriormente, espera-se que o programa seja dividido em, pelo menos, cinco grandes módulos: processo, memória, filas, recursos e arquivos. Esses modelos devem ser:

- **Módulo de Processos** – classes e estruturas de dados relativas ao processo. Basicamente, mantém informações específicas do processo.
- **Módulo de Filas** – mantém as interfaces e funções que operam sobre os processos;

- **Módulo de Memória** – provê uma interface de abstração de memória RAM;
- **Módulo de Recurso** – trata a alocação e liberação dos recursos de E/S para os processos;
- **Módulo de Arquivos** – trata as operações *create* e *delete* sobre os arquivos.

É importante ressaltar também que outros módulos podem ser utilizados, caso sejam necessários.

### 2.1 Interface de Utilização

#### 2.1.1 Saída

O processo principal é o "despachante". Esse é o primeiro processo criado na execução do problema e deve ser o responsável pela criação de qualquer processo. Na criação de cada processo, o **despachante** deve exibir uma mensagem de identificação contendo as seguintes informações:

- PID (int);
- Prioridade do processo (int);
- *Offset* da memória (int);
- Quantidade de blocos alocados na memória (int);
- Utilização de impressora (bool);
- Utilização de *scanner* (bool);
- Utilização de *drivers* (bool).

O processo deve exibir alguma mensagem que indique sua execução. Contudo, isso não é obrigatório.

#### 2.1.2 Entrada

O programa deve ler dois arquivos. O primeiro é um arquivo contendo as informações dos processos a serem criados e gerenciados pelo pseudo-SO. O segundo arquivo traz a descrição das operações a serem realizadas pelo sistema de arquivos.

No primeiro arquivo, cuja extensão deve ser .txt, cada linha contém as informações de um único processo. Portanto, um arquivo com 2 linhas deve disparar 2 processos, outro arquivo com 5 linhas fará com que o programa dispare 5 processos, e assim por diante. Cada processo, portanto, cada linha, deve conter os seguintes dados:

<tempo de inicialização>, <prioridade>, <tempo de processador>, <blocos em memória>, <número-código da impressora requisitada>, <requisição do *scanner*>, <requisição do modem>, <número-código do disco>

No segundo arquivo, cuja extensão também deve ser .txt, haverá a quantidade total de blocos no disco, a quantidade de segmentos ocupados, a identificação de quais arquivos já estão gravados no disco, a localização dos blocos usados por cada arquivo, a identificação de qual processo efetuará cada operação, a

identificação das operações, sendo **código 0** para criar um arquivo, e **código 1** para deletar um arquivo. Além disso, para as operações de criação deve constar o nome do arquivo a ser criado, e a quantidade de blocos ocupada pelo arquivo. Por outro lado, na operação de deletar um arquivo, deve constar apenas o nome do arquivo a ser deletado. Dessa forma, a organização interna do arquivo de entrada .arq deve ser:

- Linha 1: Quantidade de blocos do disco;

- Linha 2: Quantidade de segmentos ocupados no disco (n);

- A partir da Linha 3 até Linha n + 2: arquivo (a ser identificado por uma letra), número do primeiro bloco gravado, quantidade de blocos ocupados por este arquivo;

- A partir da linha n + 3: cada linha representa uma operação a ser efetivada pelo sistema de arquivos do pseudo-SO. Para isso, essas linhas vão conter: <ID_Processo>, <Código_Operação>, <Nome_arquivo>, <se_operacaoCriar_numero_blocos>. Para padronizar, o <ID_Processo> deve sempre iniciar em 0 (zero), ou seja, os processos vão ser numerados de <0> até <Quantidade de Processos – 1>.

### 2.1.3 Exemplo de Execução do Pseudo-SO

Por exemplo, um arquivo *processes.txt* que contém as linhas:

2, 0, 3, 64, 0, 0, 0, 0
8, 0, 2, 64, 0, 0, 0, 0

E um arquivo *files.txt* que contém as linhas:

10
3
X, 0, 2
Y, 3, 1
Z, 5, 3
0, 0, A, 5
0, 1, X
2, 0, B, 2
0, 0, D, 3
1, 0, E, 2

Assim, esses arquivos, ao serem usados como entrada para o programa desenvolvido, devem mostrar na tela, após ser inicializado, algo como:

$ ./dispatcher processes.txt files.txt

```
dispatcher =>
    PID: 0
    offset: 0
    blocks: 64
    priority: 0
    time: 3
    printers: 0
    scanners: 0
    modems: 0
    drives: 0

process 0 =>
P0 STARTED
P0 instruction 1
P0 instruction 2
P0 instruction 3
P0 return SIGINT

dispatcher =>
  PID: 1
  offset: 65
  blocks: 64
  priority: 0
  time: 2
  printers: 0
  scanners: 0
  modems: 0
  drives: 0

process 1 =>
P1 STARTED
P1 instruction 1
P1 instruction 2
P1 return SIGINT

Sistema de arquivos =>
```

**Operação 1 => Falha**
O processo 0 não pode criar o arquivo A (falta de espaço).

**Operação 2 => Sucesso**
O processo 0 deletou o arquivo X.

**Operação 3 => Falha**
O processo 2 não existe.

**Operação 4 => Sucesso**
O processo 0 criou o arquivo D (blocos 0, 1 e 2).

**Operação 5 => Falha**
O processo 1 não pode deletar o arquivo E porque ele não existe.

**Mapa de ocupação do disco:**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| D | D | D | Y | | Z | Z | Z | | |

Sendo que **dispatcher =>** indica o momento em que o processo despachante (SO) leu as informações do processo a ser executado, imprimiu e que cedeu tempo de CPU para o processo. Além disso, deve ser o processo despachante o responsável por imprimir na tela as operações do sistema de arquivos. Dessa forma, no exemplo acima, **P0** e **P1** são processos que executam, exibindo as informações do que cada um estará fazendo.

## 3 Estudo Teórico para a Solução

Cada grupo deverá buscar a solução para o compartilhamento de recursos e a sincronização de processos (ou *threads*), quando se fizer necessário, de acordo com o problema proposto. É responsabilidade de cada grupo estudar a maneira mais eficiente para implementar o pseudo-SO. Contudo, é importante ressaltar que para o problema de compartilhamento de recursos não há soluções mágicas, as soluções possíveis são exatamente as mesmas vistas em sala de aula: semáforos (baixo nível), monitores (alto nível, mas devem ser implementados pela linguagem de programação, pois o compilador deve reconhecer o tipo monitor) e troca de mensagens (usadas preferencialmente para garantir a sincronização entre processos em máquinas diferentes, mas também podem ser usadas para processos na mesma máquina. Nesse caso, as mensagens trocadas passam por toda a pilha de protocolos como se estivessem sendo enviadas para outra máquina).

## 4 Implementação

O trabalho valerá 10 pontos, sendo 9 pontos atribuídos para a precisão dos algoritmos (de acordo com a especificação definida nas seções anteriores), e 1 ponto dado à qualidade do código. Portanto, o código-fonte deve seguir as boas práticas de programação: nomes de variáveis auto-explicativos, código comentado e identado. As métricas de medida a serem usadas para avaliar a qualidade do código serão baseadas naquelas descritas em [2]. Dessa forma, funções grandes, classes com baixa coesão, código mal-identado e demais práticas erradas acarretarão no prejuízo da nota.

Contudo, é fundamental que apenas os padrões das linguagens, sem bibliotecas de terceiros, sejam utilizados. Ou seja, se você for executar um programa escrito em Java no Linux, que eu precise apenas do SDK padrão para recompilá-lo, se o programa for escrito em C, que eu possa compilá-lo com o padrão -ansi ou c99, etc. As especificações de padrão devem ser explicitadas. Isto é para evitar problemas de executar na máquina de vocês e não executar na minha. Padronização é fundamental!

A implementação poderá ser feita em qualquer linguagem e utilizando qualquer biblioteca, desde que o código seja compatível com ambiente UNIX. Programas que utilizarem bibliotecas ou recursos adicionais deverão conter no relatório informações detalhadas das condições para a reprodução correta do programa. Além disso, bibliotecas proprietárias não são permitidas.

## 5 Responsabilidades

Cada grupo é responsável por realizar, de maneira exclusiva, toda a implementação do seu trabalho e entregá-lo funcionando corretamente. Além disso, o grupo deverá explicar detalhadamente todo o código-fonte desenvolvido. É fundamental que ***todo o grupo*** esteja presente no dia/horário definido para apresentar e explicar o código-fonte.

## 6 Material a ser entregue

O grupo deverá, além de apresentar o código-fonte executando, entregar todos os arquivos do código desenvolvido e entregar um relatório (mínimo de 2 e máximo de 5 páginas) contendo, obrigatoriamente, **no mínimo**, os seguintes itens:

- Descrição das ferramentas/linguagens usadas;
- Descrição teórica e prática da solução dada;
- Descrição das principais dificuldades encontradas durante a implementação;
- Para todas as dificuldades encontradas, explicar as soluções usadas;
- Descrever o papel/função de cada aluno na realização do trabalho;
- Bibliografia.

Assim, cada grupo deverá submeter no Aprender 3 dois arquivos. Um arquivo .zip com todos os arquivos necessários para executar o código (incluindo um arquivo com o passo-a-passo para a execução), e um arquivo .pdf com o relatório sobre o desenvolvimento realizado.

## 7 Cronograma

O trabalho deverá ser entregue/apresentado no dia especificado para cada equipe. Para garantirmos que todos os grupos vão conseguir apresentar no dia marcado as apresentações vão iniciar no horário padrão da aula (às 8:00h), e será feito de maneira virtual, via Zoom, em salas virtuais especificas. Cada grupo terá **20 minutos** para a apresentação do trabalho realizado. É importante ressaltar que nenhum trabalho será recebido fora do prazo, por qualquer que seja o motivo. Além disso, é obrigatória a presença de todos os membros da equipe para a explicação do código-fonte desenvolvido e das decisões tomadas. No dia da apresentação todos os alunos da equipe devem estar com o **microfone e a câmera ligados** para que seja comprovada a sua real participação no trabalho. Assim sendo, é fundamental que o aluno teste antes os seus equipamentos, e que os problemas sejam contornados previamente, com dias de antecedência, para evitar justificativas (que não serão aceitas) na hora da apresentação. Para cada dia, a ordem de apresentação seguirá a ordem de numeração dos grupos, a qual ocorrerá em *link* específico, criado antecipadamente para cada equipe, na plataforma ZOOM. Assim, a data de apresentação/entrega dos trabalhos deve seguir as datas abaixo:

- **Dia 06/02/2023 – Grupos 01, 02, 03 e 04**
- **Dia 08/02/2023 – Grupos 05, 06 e 07**
- **Dia 13/02/2023 – Grupos 08, 09 e 10**

## 8 Referências

**[1]** Stallings, W. *Operating Systems Internals and Design Principles*, Pearson Education, 2009.

**[2]** Martin, R. C. *Clean Code. A Handbook of Agile Software Craftsmanship*, Prentice Hall, 2008.