INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA SUL-RIO-GRANDENSE - IFSUL, CÂMPUS PASSO FUNDO
CURSO DE TECNOLOGIA EM SISTEMAS PARA INTERNET

ÁDLER JONAS GROSS

GROSS: UM *FRAMEWORK* PARA EXPERIMENTAÇÃO E ENSINO DE GERÊNCIAS DE SISTEMAS OPERACIONAIS

Prof. Me. Élder Francisco Fontana Bernardi

PASSO FUNDO, 2014

ÁDLER JONAS GROSS

GROSS: UM *FRAMEWORK* PARA EXPERIMENTAÇÃO E ENSINO DE GERÊNCIAS DE SISTEMAS OPERACIONAIS

Monografia apresentada ao Curso de Tecnologia em Sistemas para Internet do Instituto Federal Sul-Rio-Grandense, *Campus* Passo Fundo, como requisito parcial para a obtenção do título de Tecnólogo em Sistemas para Internet.

Orientador: Prof. Me. Élder Francisco Fontana Bernardi

PASSO FUNDO, 2014

ÁDLER JONAS GROSS

GROSS: UM *FRAMEWORK* PARA EXPERIMENTAÇÃO E ENSINO DE GERÊNCIAS DE SISTEMAS OPERACIONAIS

Trabalho de Conclusão de Curso aprovado em ____/____/____ como requisito parcial para a obtenção do título de Tecnólogo em Sistemas para Internet

Banca Examinadora:

_____

Prof. Me. Élder Francisco Fontana Bernardi (Orientador)

_____

Prof. Esp. José Antônio Oliveira Figueiredo (Convidado)

_____

Prof. Me. Roberto Wiest (Convidado)

_____

Prof. Dr. Alexandre Tagliari Lazzaretti
Coordenação do Curso

PASSO FUNDO, 2014

Aos meus pais e ao meu irmão,
pelo apoio e o incentivo
em todos os momentos.

"A única forma de fazer software seguro,
confiável e rápido é faze-lo pequeno"
Andrew Stuart Tanenbaum

RESUMO

Este trabalho apresenta um *framework* de código fonte aberto que permite o usuário programar, testar e comparar seus próprios algoritmos que controlam a gerência de processador de um sistema operacional simulado. Sua principal utilidade é facilitar a experimentação de novos algoritmos de escalonamento, como também auxiliar no ensino das gerências de sistemas operacionais, visando atividades práticas de programação. Serão apresentados detalhes de desenvolvimento, funcionamento e de implementação do *framework.*

Palavras-chave: Simulação; *Framework*; Sistemas Operacionais; Escalonamento; Gerência de Processador.

ABSTRACT

This paper presents an open source framework that allows the user to program, test and compare their own algorithms that control the processor management of a simulated operating system. Its main use is to facilitate the testing of new scheduling algorithms, but also assist in the education of operating systems managers, aimed at practical programming activities. This paper also will presents the details of development, operation and implementation of the framework.

Keywords: Simulation; Framework; Operating Systems; Scheduling; Processor Management.

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

CPU - *Central Processing Unit, p. 19*

DMA – *Direct Memory Access, p. 33*

FIFO - *First In First Out, p.18*

HD - *Hard Disk, p. 19*

HRN - *Highest Response-Ratio Next, p. 18*

HPF - *Highest Priority First, p. 18*

I/O - *Input / Output, p. 18*

MMU - *Memory Management Unit, p. 19*

RAM - *Random Access Memory, p. 19*

SJF - *Shortest Job First, p. 18*

SO - Sistema Operacional, p. 12

TLB - *Translation Lookaside Buffer, p. 19*

SUMÁRIO

1 INTRODUÇÃO

O número de máquinas de propósito geral e “inteligentes”, como computadores e *smartphones*, está crescendo cada vez mais, e o software que deixa essa máquina operacional para o usuário é chamado de Sistema Operacional (SO).

Na área de SO existem inúmeros estudos que objetivaram o aumento do desempenho desses sistemas, seja por questões de mercado ou de evolução da tecnologia, ao mesmo tempo em que houve a necessidade e o interesse de aumentar a usabilidade e a segurança contra possíveis erros dos usuários.

Dessa maneira, o entendimento de mecanismos utilizados nos SOs, como a gerência do processador e a gerência de memória, é necessário para qualquer pessoa que busca programar, testar e validar como estes mecanismos reagem, porém é difícil de entender e experimentar de forma prática novos algoritmos e abordagens, da mesma maneira que comparar as técnicas a fim de verificar qual foi a melhor, utilizando-se de SOs para uso final. Portanto, como experimentar algoritmos de um SO sem a necessidade de compilar e instalar o SO a cada teste e sem demandar o conhecimento avançado requerido para fazer isso em um SO para uso final? Como avaliar as implementações criadas?

Com a utilização de simuladores e *frameworks* na realização de testes e na implementação de mecanismos de gerência do SO, tende-se a facilitar o processo, se comparado ao estudo a partir de um sistema de uso final. Porém, existem poucas ferramentas de ensino de SO que permitam a experimentação prática de programação. Algumas ferramentas são como SOs de uso final, mas simplificados para facilitar o entendimento, no entanto sofrem os mesmos problemas para se experimentar novos algoritmos. Outras ferramentas são como simuladores, mas muitas não possuem o código fonte aberto para ser possível alterá-lo.

Nesse sentido, este trabalho busca criar um ambiente em que possam ser experimentados algoritmos essenciais de um SO, de maneira simples e acessível ao programador. O método escolhido e que visa essa interação facilitada é a utilização de um simulador, o qual foi implementado. A implementação permite a troca de algoritmos do SO e que os programas que executam na simulação sejam customizados. No final da simulação é gerado um resultado que pode ser utilizado para comparar as execuções de algoritmos diferentes e verificar quais foram os mais eficiente.

1.1 MOTIVAÇÃO

A principal motivação para esta iniciativa é a ausência de *frameworks* ou simuladores de sistemas operacionais, em especial mas não unicamente, de tecnologia nacional que sejam de código fonte aberto, onde incentiva-se a programação e não apenas a visualização.

Em sistemas operacionais tradicionais, modificar o mecanismo de escalonamento ou outros mecanismos básicos de funcionamento demandaria um conhecimento extremamente avançado e tanto tempo investido, que tais mudanças poderiam ser consideradas improváveis. Para isso objetivou-se que a simulação seja mais simples se comparada com sistemas operacionais de mercado, e que a quantidade de código fonte fosse relativamente menor, facilitando a leitura e a modificação deste.

O *framework* deste trabalho busca facilitar o entendimento interno de sistemas operacionais por alunos de informática e outras pessoas interessadas, pois partes do código foram projetadas para serem modificadas enquanto outras partes para serem expandidas.

1.2 OBJETIVOS

Nesse capítulo é apresentado o objetivo geral e objetivos específicos do trabalho.

1.2.1 Objetivo geral

Desenvolver um *framework* para experimentação e ensino da gerência de processador, o qual permita a manipulação de seus algoritmos, visando avaliar o desempenho dos algoritmos.

1.2.2 Objetivos específicos

- Estudar sobre as gerências de sistemas operacionais, especificamente seus algoritmos e como os mesmos funcionam.
- Analisar outros simuladores já existentes, como também o sistema operacional MINIX, a fim de verificar aspectos importantes que devem ser observados na simulação.
- Projetar a lógica do *framework*, como a ordem de eventos e funcionalidades.
- Propor um mecanismo que permita a troca de algoritmos da simulação.

- Propor uma maneira de visualizar a simulação para comparar resultados, sendo possível a comparação de algoritmos.
- Desenvolver o protótipo.

2 REFERENCIAL TEÓRICO

Este capítulo apresenta sistemas relacionados com este trabalho e conceitos de Sistemas Operacionais necessários para entender a gerência de memória e a gerência de processador.

2.1 SISTEMAS OPERACIONAIS

De uma maneira geral, Oliveira et al. (2010, p. 22) define sistema operacional como "uma camada de *software* colocada entre o *hardware* e os programas que executam tarefas para os usuários" e que tem como objetivos: autorizar que a operação de um computador por um usuário ocorra da forma mais conveniente possível; permitir o acesso e compartilhamento dos recursos do equipamento de forma organizada e protegida; fazer o baixo nível de máquina ficar transparente através de uma série de abstrações.

Os SOs possuem um núcleo (nome que se origina do inglês *kernel*) que possui o controle primário do hardware. Segundo Null (2006, p. 442), o núcleo fornece serviços comuns, coordena diretamente toda a entrada e saída, carrega todos os controladores de dispositivos, inclusive faz o gerenciamento da memória e é responsável pelo escalonamento dos processos, o sincronismo, a segurança e o tratamento das interrupções.

2.1.1 Chamada de Sistema

Se as aplicações em modo usuário (que não possuem a mesma prioridade do núcleo) tentarem fazer uma operação direta de hardware, um erro será informado, por não possuírem autorização para fazê-la, isso graças às camadas de abstração do SO.

Por isso, quando uma aplicação precisa fazer uma operação deste tipo, que exija modo privilegiado, ela envia uma "chamada de sistema" que solicita que o núcleo faça a operação por ela, enquanto a aplicação poderá ficar bloqueada. Se o usuário dono do processo tiver a autorização devida para a operação, o núcleo a efetuará, mas, se o usuário não tiver a autorização, poderá retornar um erro. Segundo Silberschatz (2008 p. 556), a chamada de sistema "envolve a transferência do controle do modo de usuário não-privilegiado para o modo do *kernel* privilegiado" e ainda que "os detalhes dessa transferência variam de uma arquitetura para outra."

2.1.2 Multiprogramação

Multiprogramação é uma técnica que permite a um SO criar a ilusão de que vários processos estão sendo executados simultaneamente. Esta técnica permite que o SO se torne ativamente multi-usuário, com mais de uma pessoa usando o SO ao mesmo tempo, além de melhorar a eficiência do uso do hardware, pois durante a inatividade de um processo, outro pode ser executado, melhorando muito a experiência de utilização dos usuários.

Um ponto negativo da multiprogramação é o aumento da complexidade do núcleo devido à questão de segurança, pois é necessário que o núcleo faça a proteção através da memória virtual, para que um processo não acesse os mesmos dados ou região de memória que outro, a menos que ambos declarem compartilhar as áreas envolvidas, para que não cause corrupção dos dados, o que pode causar erros em ambos os processos envolvidos e instabilidade no próprio SO, já que o próprio núcleo é um processo que está sendo executado e por esse motivo também está na memória primária do computador.

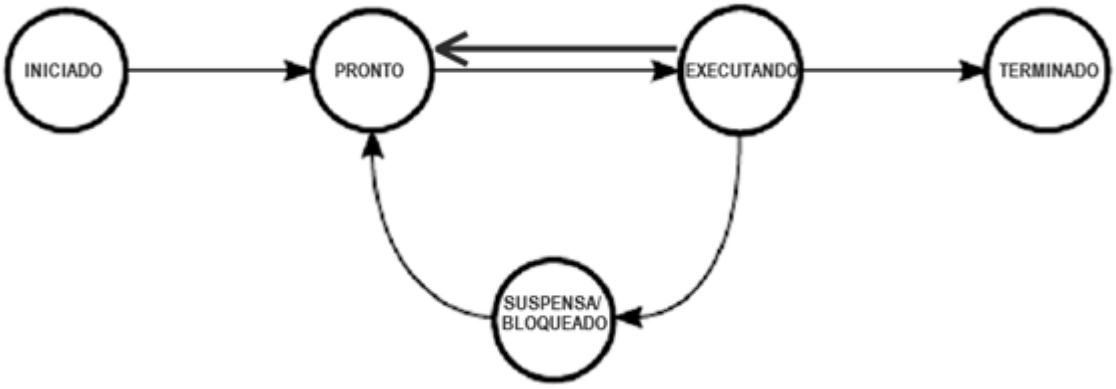
2.1.2.1 Processos

Processo é um programa em execução, sendo assim, o mesmo programa pode estar executando várias vezes ao mesmo tempo em um SO. É importante perceber que enquanto o programa é um elemento inativo, o "processo é um elemento ativo que altera o seu estado, à medida que executa um programa" (OLIVEIRA et al., 2010, p. 39).

2.1.2.2 Estados de um processo

Como dito por Oliveira et. al. (2010), os processos alternam por estados durante a execução, estes estados e suas transições são mostradas na Figura 1.

Figura 1 - Estados de um processo



Fonte: Autor

Cada estado e suas transições são explicadas por Sharma (2010, p. 104) da seguinte maneira:

1. **Iniciado**: O Processo está sendo criado, com o programa e bibliotecas compartilhadas sendo copiados na memória, e estruturas de dados do *kernel* sendo alteradas para permitir a execução;
2. **Terminado**: O processo terminou a execução, processo removido da memória, estruturas de dados temporárias do *kernel* são apagadas;
3. **Pronto**: Processo aguardando disponibilidade do processador, atividade que é comandada pelo algoritmo de escalonamento;
4. **Executando**: Fluxo de instruções sendo executados no processador;
5. **Bloqueado**: Processo que não pode continuar, pois está esperando algum evento acontecer antes de continuar, como o término de uma entrada ou saída;
6. **Iniciado=>Pronto**: O processo já foi carregado na memória;
7. **Pronto=>Executando**: Processo escolhido pelo escalonador, carregando contexto;
8. **Executando=>Pronto**: Escalonador decidiu remover o acesso do processo ao processador, salvando o contexto. Esta etapa normalmente ocorre quando se esgota o tempo que o escalonador determinou ao processo;
9. **Executando=>Bloqueado**: Processo precisa de recurso que não possui, e é bloqueado. Quando o processo solicita um recurso que está disponível naquele momento, ele ficará bloqueado até o recurso se tornar disponível;
10. **Bloqueado=>Pronto**: Processo obteve os recursos que precisava, e está pronto para entrar na fila de espera do escalonador;
11. **Executando=>Terminado**: O processo pode ser morto ou emitir aviso (Chamada de Sistema) de término de execução.

2.1.3 Gerência de Processador

Nesse subcapítulo são apresentados os principais algoritmos de escalonamento e o conceito de preemptividade.

2.1.3.1 Preemptividade

Os algoritmos de escalonamento podem ser preemptivos ou não-preemptivos, segundo Tanenbaum (2008, p. 232) a preemptividade se baseia no fato de que o processador e a memória são recursos que podem ser retirados de um processo e devolvidos posteriormente sem prejuízo, enquanto que outros recursos não podem sofrer preempção, como impressoras e

arquivos, pois muitas vezes um arquivo não pode ser retirado de um processo sem ocorrer problemas. Em algoritmos de escalonamento não-preemptivos, o processo não pode perder acesso ao processador até sua finalização ou bloqueio por operação de I/O. Alguns algoritmos de escalonamento são explicados por Li, Li e Shih (2014, p 112), estes que são apresentados a seguir.

2.1.3.2 Round Robin

O escalonamento Round-Robin efetua um rodízio entre os processos em execução, atribuindo tempos iguais (*Quantum*) para cada um deles. Se mais de um processo iniciar simultaneamente, o algoritmo escolherá arbitrariamente um dos processos para ter acesso ao processador. A prioridade não é atender os novos processos que são executados, mas a prioridade é atribuir um tempo igual para cada um.

2.1.3.3 *First In First Out* (FIFO)

Este é o algoritmo de escalonamento mais simples, o primeiro a entrar é o primeiro a sair, ou seja, o primeiro processo que entrar em execução será o primeiro a sair, para dar lugar ao próximo. Ponto positivo: Justiça com os processos que chegaram primeiro. Ponto Negativo: Monopoliza os processos, havendo demora na troca dos mesmos.

2.1.3.4 *Shortest Job First* (SJF)

Este algoritmo de escalonamento considera que primeiro deve ser feito o trabalho mais curto, escolhendo o processo que demora menos tempo como prioridade de execução. Ponto positivo: Grande vazão e dinâmica dos processos. Ponto negativo: Impossibilidade de implementação 100% correta, pois os processos podem demorar tempos inesperados, e este erro de cálculo pode levar o sistema ao desequilíbrio.

2.1.3.5 *Highest Response-Ratio Next* (HRN)

O algoritmo de escalonamento é uma variante do SJF, e considera que o próximo processo deve ser aquele que tiver a maior razão de resposta. Esta razão de resposta é calculada utilizando-se o tempo de espera e o tempo esperado para execução.

2.1.3.6 *Highest Priority First* (HPF)

Este tipo de escalonamento é uma variante do FIFO por colocar todos os processos em uma fila, além disso, existe um valor de prioridade nos processos, sendo que serão atendidos

primeiramente os processos de maior prioridade. Ponto Positivo: Favorece processos com prioridade. Ponto Negativo: Podem ocorrer situações de processos de baixa prioridade nunca serem atendidos.

O algoritmo possui uma variante com o objetivo de corrigir seu ponto negativo, criando o conceito de envelhecimento (*Aging*); à medida que um processo é processado, a prioridade dos demais, que não estão em execução, é incrementada.

2.1.4 Gerência de Memória

Nesse subcapítulo são apresentados os conceitos envolvidos com a gerência de memória.

2.1.4.1 Endereços Lógicos e Endereços Físicos

Segundo Jadhav (2009, p. 100) os endereços lógicos são criados pela CPU a pedido do núcleo, são visualizados pelo processo e sempre iniciam em zero. Os endereços físicos são locais reais na memória (registradores, *cache*, RAM, HD).

A Unidade de Gerenciamento de Memória (*Memory Management Unit*, MMU) faz a conversão de endereços lógicos para endereços físicos através da *Translation Lookaside Buffer (*TLB) que é uma memória cache que contém a tabela de paginação. Se algum processo não estiver registrado na TLB, não é feita a tradução, e o endereço lógico é interpretado como físico, assim como o próprio núcleo.

2.1.4.2 Modalidades de Gerenciamento de Memória

Segundo Null (2006, p. 343-344), existem 3 modalidades de gerenciamento de memória:

1. **Partições Fixas**: Divide a memória em partes fixas, e o endereço lógico é convertido em físico pela adição do valor de início da partição onde se encontra;
2. **Alocação por Segmentos**: O processo é particionado em segmentos de endereços lógicos, que são distribuídos nos endereços físicos, cuja tradução ocorre com a necessidade das tabelas de segmentação;
3. **Alocação por Páginas**: Os endereços lógicos são divididos em blocos (páginas) de tamanho igual, e os endereços físicos em quadros, cuja tradução é feita pela MMU através da TLB.

2.1.4.3 Fragmentação Interna e Fragmentação Externa

Existem dois tipos de fragmentação que podem ocorrer no gerenciamento de memória, que são explicadas por Null (2006, p. 344):

1. **Fragmentação Externa**: Ocorre durante a alocação de memória por segmentos, devido a não ocupação útil dos espaços entre os segmentos, fazendo com que qualquer programa que seja maior que estes espaços não possa ser alocado;
2. **Fragmentação interna**: Ocorre durante a alocação de memória por páginas, quando as mesmas não são completamente preenchidas pelos endereços lógicos do processo em execução, deixando áreas inutilizadas dentro dos quadros.

Na prática, nenhuma modalidade de alocação de memória está livre de problemas.

2.1.4.4 Estratégias de Alocação de Memória

Existem algoritmos usados pelos SO para diminuir os problemas da fragmentação de memória, entre os quais Samanta (2009, p. 80) explica:

1. ***First-Fit***: Escolhe o primeiro espaço vago de memória em que caiba o processo a ser alocado;
2. ***Best-Fit***: Varre a memória procurando pelo espaço vago mais justo para o processo a ser alocado;
3. ***Worst-Fit***: Varre a memória procurando pelo espaço vago menos justo para o processo a ser alocado, esperando que o processo peça mais memória durante a execução, e que possua espaço livre para tal;
4. ***Next-Fit***: Escolhe o primeiro espaço vago em que caiba o processo a ser alocado, iniciando a varredura pela última posição da varredura anterior.

No protótipo desenvolvido não foi implementada a gerência de memória, a qual fica para um trabalho futuro. A seguir apresentam-se os sistemas relacionados com o presente trabalho.

2.2 SISTEMAS RELACIONADOS

Como o objetivo deste trabalho foi desenvolver um simulador de SO, para uma melhor implementação buscou-se estudar outros sistemas que foram construídos com propósitos similares.

2.2.1 SOsim

SOsim é um simulador de sistemas operacionais criado por Maia com o propósito de ser educacional, o qual permite alterar algoritmos de escalonamento e operações do SO através da interface gráfica. O principal objetivo desse sistema é servir como uma ferramenta para aulas das disciplinas de SO, pois estudar os sistemas operacionais para uso final poderia dificultar o aprendizado do aluno devido à sua complexidade.

Um dos maiores problemas encontrados nesse simulador é a falta do código fonte. Outro problema é que, sem emulação ou virtualização, só funciona nos sistemas operacionais do tipo Windows.

O simulador emula os principais subsistemas de um SO multiprogramável, como gerência de processos, escalonamento e memória virtual por paginação (MAIA, 2001, p. 4).

Segundo Maia (2001, p. 62-65, 71, 74, 76, 79), a aplicação possui diversas janelas com recursos, como:

1. **Console**: Possui controles de toda a simulação, como iniciar, pausar e parar o modelo.
2. **Gerência de Processador:** Janela que apresenta graficamente a mudança de estados dos processos e do escalonador.
3. **Gerência de Memória**: Janela que apresenta graficamente a memória, permitindo a visualização de páginas de memória livres e alocadas, e opções de gerência de memória virtual.
4. **Log de mensagens**: Janela que contêm detalhes específicos do sistema, que são gerados dentro de períodos de tempo pré-determinados.
5. **Estatísticas**: Janelas que exibem diversas informações estatísticas em tempo real.
6. **Criação de processos**: Permite atribuir os detalhes dos processos que serão criados, inclusive os identifica-los por cores.
7. **Visualização de processos:** Exibe a lista de processos criados e seus detalhes.
8. **Visualização dos PCBs**: Apresenta as informações do bloco de controle de processos da simulação.

9. **Gerência do Processador**: Mostra uma animação gráfica representativa do processador, e permite controles como o do *quantum*, *clock* e tempo de espera.
10. **Opções de Escalonamento**: Permite mudar o tipo de escalonamento e prioridades da simulação.
11. **Opções de Memória Virtual**: Janela que permite modificar opções da simulação referentes à memória, como quantidade de páginas livres e o tipo de busca.
12. **Arquivo de paginação**: Janela similar à gerência de memória, mas que exibe detalhes do arquivo virtual de paginação, como, por exemplo, os blocos reservados.

2.2.2 BACI

Ben-Ari Concurrent Interpreter (BACI) é um sistema que foi criado para o ensino de Programação Paralela e Distribuída. Ele vem sendo desenvolvido desde 1995 e possuí executáveis para Linux, Windows, entre outros, e ainda existe uma versão em Java. Possui um compilador de linguagem de programação similar ao C++, o C--. Ele também "simula a execução de processos concorrentes, suportando monitores e semáforos." (STALLINGS, 2012, p. 762).

Para interação com o sistema, é necessário trocar os arquivos de configuração e programar os programas que serão executados, verificando o que acontece.

Aparenta ser de fácil utilização e possui exemplos de programas para testar, todos com um passo a passo do que fazer.

O interpretador possui diversos recursos, como janelas de exibição de entradas, exibição de saída, debug, monitoração das variáveis nos processos e tabela de processos, também permitindo criar pontos de parada e inspeção de programas em execução (BACI, s.d., p. 3-7).

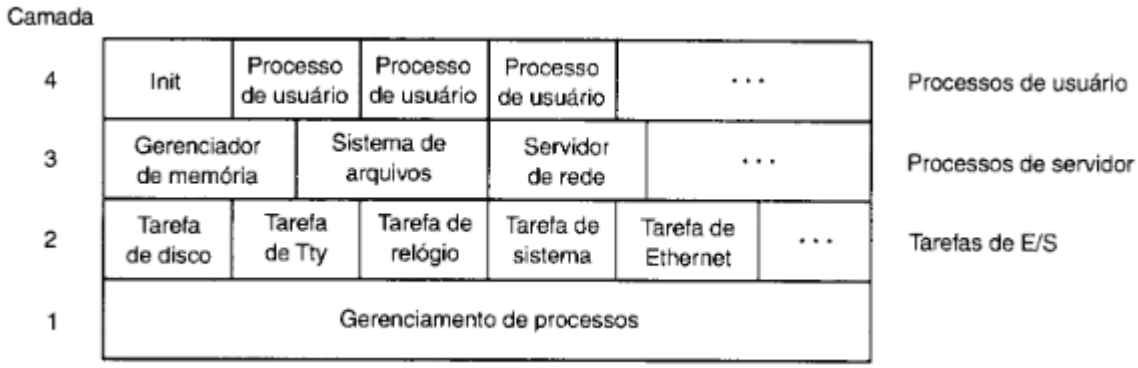
2.2.3 MINIX

Sistema operacional considerado um clone mínimo do UNIX escrito por Andrew S. Tanenbaum, para permitir o ensino de sistemas operacionais, 100% livre da propriedade intelectual da AT&T, já que a AT&T resolveu banir o ensino a partir do UNIX. MINIX é baseado na ideia de *microkernel* (*kernel* modularizado), e não em *kernel* monolítico (em bloco único) como o Linux.

Segundo Tanenbaum (2002, p. 76), a escolha de um *microkernel*, foi para ter um projeto flexível, justamente por que é possível substituir partes inteiras do sistema sem a necessidade de recompilar o *kernel*, devido a sua natureza modularizada.

O MINIX é dividido em 4 camadas conforme a Figura 2, que são explicadas por Tanenbaum (2002, p. 76), a camada 1 sendo a responsável pela captura de todas as interrupções, fornece para as camadas mais altas um modelo de processos sequenciais independentes, que se comunicam utilizando mensagens, e “esta parte da camada que lida com nível mais baixo do gerenciamento de interrupções é escrito em liguagem assembly. O resto da camada e todas as camadas mais altas são escritos em C” Tanenbaum (2002, p. 76).

Figura 2 - Estrutura do MINIX em 4 camadas



Fonte: Tanenbaum (2002, p. 77)

A camada 2 contém processos de E/S, um por tipo de dispositivo, Tanenbaum prefere chamá-los de tarefas, mas explica que "em muitos sistemas, as tarefas E/S são chamadas de drivers de dispositivo". (2002, p. 76)

Uma tarefa é necessária para cada tipo de dispositivo, com exceção da 'tarefa de sistema', que oferece serviços, como copiar entre diferentes regiões de memória para processos que não possuem a permissão de fazê-lo. Em processadores mais antigos, tais restrições não podem ser impostas.

Dessa maneira, Tanenbaum (2002, p. 76) define que "todas as tarefas na camada 2 e todo o código na camada 1 estão vinculados entre si em único programa binário chamado Kernel."

Os processos da camada 3 sempre estão ativos enquanto o MINIX está ligado. A seguir Tanenbaum explica o funcionamento da camada 3:

> "A camada 3 contém processos que fornecem serviços úteis para os processos de usuário. Estes processos de servidor executam em um

> nível menos privilegiado que o Kernel e as tarefas, e não podem acessar portas de E/S diretamente. Eles também não podem acessar memória fora dos segmentos atribuídos a eles. O gerenciador de memória (MM) executa todas as chamadas de sistema do MINIX que envolvem gerenciamento de memória, como FORK, EXEC e BRK. O sistema de arquivos (FS) executa todas as chamadas de sistema de arquivos, como READ, MOUNT e CHDIR". Tanenbaum (2002, p. 76-77).

As camadas 1 e 2 executam com privilégio modo *kernel*, enquanto as camada 3 e 4 executam com privilégio modo usuário. A camada 4 é composta por processos do usuário.

Tanenbaum explica o que é um *daemon* e sua diferença em relação a processos de usuários:

> "Um daemon é um processo de segundo plano que executa periodicamente ou sempre espera algum evento, como a chegada de um pacote de rede. Em certo sentido, um daemon é um servidor que é iniciado independentemente e executa como um processo de usuário. Entretanto, diferentemente dos servidores verdadeiros instalados em entradas privilegiadas, esses programas não podem receber o tratamento especial do kernel que os processos servidores de memória e de arquivos recebem". Tanenbaum (2002, pág 77).

2.2.4 SimulaRSO

SimulaRSO é uma demonstração on-line, do cálculo dos tempos de escalonamento de processos, disco e paginação de memória, cujos resultados são apresentados para o usuário através de gráficos em uma interface web.

São apresentados os algoritmos para cada subsistema:

1) Escalonamento de processos (FCFS, SJF, SRT, Round Robin)

2) Requisições de entrada e saída (I/O) de um disco rígido: (FCFS, SSTF, SCAN, C-SCAN, C-LOOK)

3) Substituição de páginas de uma memória virtual (FIFO, LRU, OPT, MRU).

O projeto é de código fonte aberto, o qual pode ser baixado[1].

[1] Disponível em https://github.com/caio-ribeiro-pereira/SimulaRSO

2.2.5 Comparativo entre os sistemas relacionados

Uma comparação entre GROSS e as ferramentas comentadas nas subseções anteriores pode ser visualizada na Tabela 1. Como pode ser visto na Tabela 1, dentre as ferramentas, GROSS é a mais portável (no sentido de não precisar de executáveis externos, como servidores) com o ideal de programação.

Por MINIX ser um sistema operacional para uso final, pela hipótese pode ser considerado complexo se comparado com qualquer uma das ferramentas que são simulações.

SimulaRSO segue alguns padrões de programação (como a generalização dos algoritmos) que implica em facilitar a adição de novos recursos, porém, essas facilidades foram voltadas para o desenvolvimento da ferramenta em si, isso pode ser observado já que esta ferramenta foi construída no intuito de auxiliar em aulas de sistemas operacionais a distância, portanto, seu objetivo não é ser expandida.

O SOsim não possui código fonte disponível (no tempo de criação deste documento), por mais que o site[2] do projeto indique uma futura divulgação.

O sistema BACI foi construído para aprendizagem de sistemas paralelos e distribuídos, portando não entra diretamente no escopo do trabalho, mas com o mesmo foi possível se obter várias ideias de implementação como a utilização de arquivos de configuração e programas customizáveis.

Tabela 1 - Comparação entre as ferramentas relacionadas e o protótipo do GROSS

Ferramenta / Recurso	GROSS	SOsim	BACI	MINIX	SimulaRSO
Código fonte aberto	Sim	Não	Sim	Sim	Sim
Gerência de Processador	Sim	Sim	Não	Sim	Sim
Gerência de Memória	Não	Sim	Não	Sim	Sim
Simulação	Sim	Sim	Sim	Não	Sim
Arquivos configuram a simulação	Sim	Não	Sim	Não é simulação	Não
Programas customizáveis	Sim	Não	Sim	Sim	Não
Facilita novos recursos	Sim	Não é possível	Não	Não	Sim
Necessidade de Servidor	Não	Não	Não	Não	Sim

Fonte: do Autor.

Conhecidas as principais diferenças com os sistemas relacionados, no capítulo 3 é apresentado vários aspectos do trabalho proposto.

[2] Disponível em http://www.training.com.br/sosim/

3 TRABALHO PROPOSTO

Propõe-se a arquitetura de um *framework* para experimentação e implementação da gerencia de processador de sistemas operacionais, oferecendo modelos para a implementação de algoritmos customizados para esta gerência e que possibilite alternância entre os algoritmos criados. Além disso, que forneça a possibilidade de executar programas em uma linguagem simplificada no simulador, permitindo a comparação de execuções destes em diferentes configurações da simulação, tais como, em uma primeira configuração, processos apenas I/O e FIFO (como o algoritmo de escalonamento do processador) e em uma segunda configuração, processos apenas I/O, mas com o algoritmo Round Robin. A comparação deve ser possibilitada por resultados mostrando dados relevantes da configuração simulada.

Os algoritmos FIFO e Round Robin deverão estar programados com o *framework*, conforme no referencial teórico, para que possa ser utilizado como exemplo em caso de dúvidas, foram escolhidos esses algoritmos devido a maneira como operam ser de fácil entendimento e pelo fato do FIFO não utilizar troca de contexto (algoritmo não-preemptivos), enquanto o Round Robin utilizar troca de contexto (algoritmo preemptivos).

3.1 METODOLOGIA

Inicialmente foi feito uma pesquisa bibliográfica a fim de apresentar os principais conceitos envolvidos na pesquisa, e para fazer uma revisão de literatura sobre o tema. Na sequência, foi feito um estudo aplicado, através da criação da modelagem e de um protótipo de simulador de SO para ser possível a experimentação e implementação de gerências de processador.

3.2 LICENÇA

Buscou-se uma licença que tivesse as características mais permissivas possíveis, e que fosse de amplo conhecimento da comunidade de software.

Foi escolhida a licença MIT, disponível pela Open Source Iniciative (2014), pois não possui as restrições com relação a autoria (como a licença BSD), e que possui apoio da comunidade de software livre.

A necessidade da Licença surgiu, pois segundo Google Code (2014) há países como a França, em que o autor de qualquer software em domínio público recebe responsabilização por quaisquer atos decorrentes do uso do mesmo.

3.3 FUNCIONALIDADES DESEJADAS

Na implementação deve ser possibilitada a troca de algoritmos do SO, por exemplo trocar o algoritmo de escalonamento SJF para o Round Robin e como cada algoritmo é diferente, esperam-se resultados diferentes. Os resultados da execução poderão ser utilizados para medir e comparar os algoritmos ou o conjunto de algoritmos.

A simulação deve permitir criação de cenários únicos (configurações), sendo possível a escolha dos programas que executarão na simulação e suas respectivas prioridades. Para uma execução mais dinâmica o *framework* deverá permitir que os programas que executem na simulação também possam ser alterados, através da edição do código fonte destes. Dessa maneira será possível criar programas específicos para testar os algoritmos, como programas que somente utilizam o processador, que somente utilizam entrada e saída ou que utilizem ambos, alterar a ordem de execução dos programas e a prioridade dos processos.

3.4 POSSIBILIDADES DE UTILIZAÇÃO

Pode ser utilizado com fins educativos, para verificar os efeitos da mudança do escalonamento, além de abrir a possibilidade para eventuais testes de desempenho.

A aplicação é de código fonte aberto e a divulgação pública na Internet permite que o leitor interessado reprograme trechos e faça experiências com suas próprias versões de escalonamento.

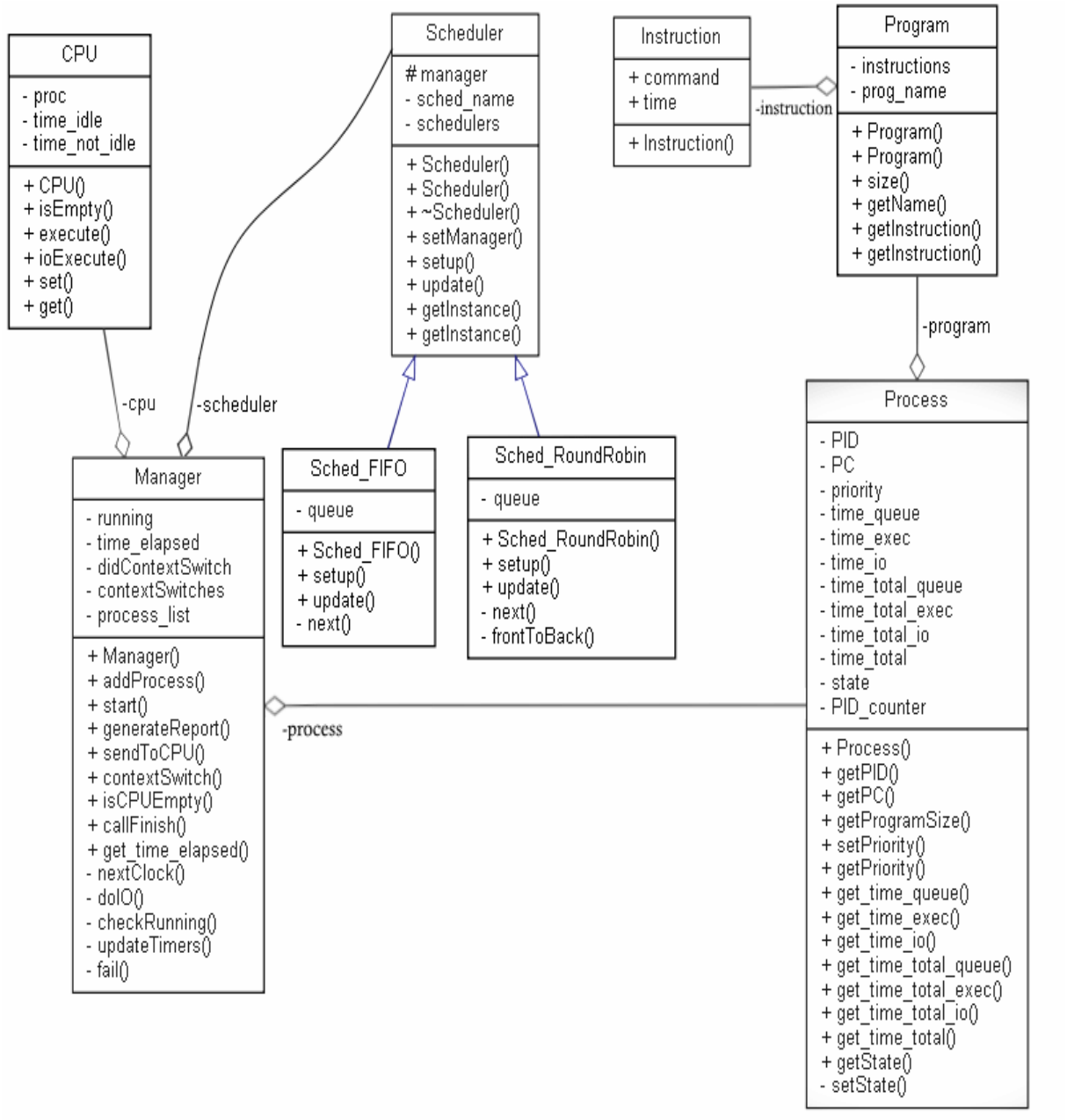
Também é possível a criação de escalonadores genéricos, como um Round Robin Genérico que recebe por parâmetro o *quantum*, onde então subclasses constroem a superclasse genérica com determinados parâmetros. Desse modo podem ser adicionados níveis de abstração para melhorar o aprendizado de SO.

Pode ser útil em aulas de sistemas operacionais onde deseja-se lições práticas de programação, nessa situação o professor pode pedir para que alunos programem um escalonador seguindo determinadas regras.

4 MODELAGEM

A aplicação seguiu a modelagem do diagrama de classes apresentado na Figura 3.

Figura 3 - Modelagem UML do Framework



Fonte: do Autor

Assim como o BACI, preferiu-se que a aplicação deve receber arquivos de entradas, um que configure a simulação (gerente de processador - nome generalizado para escalonador -

e a listagem de programas) e um arquivo que indique as ações de cada programa de modo que cada programa possa ser único.

As classes Program e Instruction estão relacionadas com o carregamento dos programas. O arquivo com as ações de um programa é carregado e dividido em várias Instructions, que por sua vez formam um Program. Desse modo, o arquivo fonte de um programa só precisa ser carregado uma única vez e então ser copiado quando for colocado em execução (transformado em um processo).

Depois de carregar os arquivos e estar com os programas preparados para executar, deve-se criar um processo para cada pedido de execução originário do arquivo de configuração e informar para o Manager a adição de um novo processo (através de addProcess). A adição de todos os processos deve ser feita antes do início da simulação, para que todos processos iniciem na simulação com estado ‘pronto’.

Process funciona como um descritor de processo, contém todas informações sobre cada processo (prioridade, tempos, registrador PC que indica a instrução para executar, o tamanho do programa e seu estado). Caso algum algoritmo de escalonamento precise de qualquer informação de um processo, deverá utilizar-se dos getters da classe, medida de segurança para que um algoritmo de escalonamento não altere as informações de um processo.

Scheduler é a classe base para os algoritmos de gerência do processador, todo algoritmo que pretende gerenciar a CPU deverá herdar de Scheduler, assim como Sched_FIFO e Sched_RoundRobin. Portanto para o usuário do *framework* a única classe que será necessário implementação é uma classe que herde Scheduler. Herdando de Scheduler, o algoritmo é automaticamente cadastrado para ser utilizado na simulação.

Manager é a classe gerente da simulação (controla a simulação), e por questões de organização e controle das ações no *framework*, apenas Manager tem controle sobre a CPU e sobre a simulação, desse modo, tudo o que envolve a lista de processos, a CPU e dados da simulação, devem ser pedidos para Manager, de modo que qualquer algoritmo de gerência de processador deve fazer requisições para Manager, que então repassa para CPU.

A classe CPU é a parte do sistema responsável em executar as instruções dos programas e realizar a entrada e saída, isso para que toda a troca de estados dos processos seja centralizada na classe CPU. Os métodos set e get, colocam e tiram respectivamente um processo na CPU. O método ‘isEmpty’ retorna o estado da CPU, se algum processo está executando ou não. execute e ioExecute são métodos utilizados respectivamente por Manager para realizar a execução do processo que está na CPU e de I/O de determinado processo.

5 DESENVOLVIMENTO DO PROTÓTIPO

Neste capítulo é explicado como foi desenvolvido o protótipo do sistema, como ocorre o funcionamento e os métodos de programação utilizados.

5.1 FERRAMENTAS UTILIZADAS

O protótipo foi programado na linguagem de programação C++, tendo como requisito mínimo que o compilador suporte a padronização C++11. Utilizou-se a IDE Code::Blocks 13.12 e o compilador GCC 4.9.2 que é disponibilizado pelo MinGW-W64 4.9.2. As versões específicas do MinGW-W64 para compilar o protótipo para Windows nas arquiteturas i686 e x86-64, como segue na Tabela 2.

Tabela 2 - Versões do MinGW-W64 utilizadas na compilação do protótipo

Arquitetura	Modelo de Threads	Tratamento de Exceções	Revisão
i686	Posix	dwarf	0
x86-64	Posix	seh	0

Fonte: do Autor.

Os compiladores disponibilizados em conjunto com o Code::Blocks podem não suportar determinados recursos do C++11, devido a isto, recomenda-se a utilização de um compilador com suporte, como o citado anteriormente. O protótipo também foi testado em um Debian 7.7 com G++ 4.9, embora, no tempo de escrita deste trabalho, para baixar uma versão recente do G++ seja necessário alterar os repositórios padrões para os repositórios ainda em teste, como o repositório do Debian 8.0 (jessie).

5.2 FUNCIONAMENTO

Baseado na modelagem, a aplicação recebe um arquivo de entrada que contém as informações sobre qual escalonador será utilizado na simulação e os programas que deverão ser executados com suas respectivas prioridades. Como o protótipo é uma aplicação para terminal, o arquivo de entrada é informado após o nome do programa. O Quadro 1 mostra um exemplo de como é executado a aplicação.

Quadro 1 - Exemplo de chamada pelo terminal da aplicação

```
>GROSS exemplo.txt
```

Fonte: do Autor.

5.2.1 Estrutura do Arquivo de Configuração de Execução

O arquivo de entrada segue o seguinte padrão:

- É obrigatório a primeira palavra ser “escalonador” ou “scheduler”;
- Após a palavra obrigatória deve haver o nome de um escalonador existente. Dois escalonadores já vem programados e se chamam “sched_fifo” e “sched_roundrobin”.
- Depois do escalonador estar definido, coloca-se a lista de programas que devem ser executados e suas respectivas prioridades.

Um exemplo de arquivo de configuração (exemplo.txt) de execução está no Quadro 2:

Quadro 2 - Exemplo de arquivo de configuração

```
1    escalonador sched_roundrobin
2    cpuIntensive 8
3    cpuIntensive 2
4    cpuIO 5
5    ioIntensive 3
6    cpuIO 4
7    cpuIntensive 2
8    ioIntensive 4
9    ioIntensive 1
10   ioIntensive 3
11   cpuIO 7
```

Fonte: do Autor.

Para definir um programa é necessário seguir a seguinte lógica:

1. Cada programa é definido por um arquivo que deve estar no diretório “programs”.
2. O diretório “programs” deve estar no mesmo diretório que a aplicação GROSS.

3. O nome do arquivo onde está definido o programa é o nome do programa com a extensão “.prog”, por exemplo, o programa “cpuIntensive” está definido no arquivo “cpuIntensive.prog”.

5.2.2 Estrutura do Arquivo de um Programa

Os programas também possuem uma estrutura específica, esta estrutura pode ser observada a seguir:

- Cada linha é uma instrução;
- Linhas em branco são ignoradas;
- Uma instrução é definida por uma operação e o número de *clocks* (tempo de simulação) necessário para realizar esta operação;
- As operações válidas são “io” e “process”;
- Para o número de *clocks* apenas serão considerados números positivos até 32767, e ao se encontrar algum carácter inválido, o restante é ignorado, sendo considerado apenas o trecho válido;
- A operação “process” indica a quantidade de *clocks* que o programa precisa para realizar algum processamento na CPU da simulação.
- A operação “io” indica a quantidade de *clocks* que o programa ficará em I/O, devido a isso, coloca o processo em estado bloqueado pelo tempo da operação. Essa operação de maneira implícita utiliza 1 *clock* de processador, o equivalente à instrução “process 1”, um custo para realizar I/O.

Um exemplo de programa está no Quadro 3:

Quadro 3 - Exemplo de programa

```
1  process 10
2  io 5
3  io 5
4  process 8
```

Fonte: do Autor.

Como os tempos de instrução são bem definidos, é possível saber quantos *clocks* o processo que estiver executando um programa ficará utilizando o processador e quantos

clocks ficará bloqueado. O processo que executar o programa do exemplo do Quadro 3 utilizará 20 *clocks* de processamento - 10 da primeira instrução, mais 1 (*clock* implícito) da segunda instrução, mais 1 (*clock* implícito) da terceira instrução, e por fim, mais 8 da quarta instrução – e ficará 10 *clocks* bloqueado – 5 da segunda instrução mais 5 da terceira instrução.

5.2.3 Execução da Simulação

Depois do usuário informar o arquivo de entrada, a aplicação valida o arquivo e carrega os programas necessários para a execução. Caso algum erro ocorra nessa etapa, este é informado ao usuário. Após carregar os programas, ocorre a etapa de criação dos processos. Nessa etapa, é criado um processo na simulação para cada chamado de execução que foi especificado pelo usuário no arquivo de entrada.

Antes de iniciar a simulação, o Gerente da Simulação inicializa o escalonador escolhido com a lista de processos que estão executando, para que o escalonador possa reorganizar a lista caso necessário – como armazenar em uma fila ou pilha.

Em seguida a simulação é iniciada e a cada *clock* são executadas várias tarefas até o fim da simulação. Essas tarefas são apresentadas a seguir:

1) A execução do escalonador para que o mesmo tome as decisões que foram programadas. Todas estatísticas de *clocks* da simulação são altamente influenciado pelas decisões tomadas pelo escalonador.
2) Atualizar os tempos da simulação.
3) Execução do I/O de cada processo, pois como ocorre a simulação de uma DMA e o I/O de cada processo é considerado independente, o I/O pode ocorrer em paralelo.
4) Execução da CPU, que pode executar uma instrução de um processo ou não executar nada, depende das decisões tomadas pelo escalonador.
5) Verificações se a simulação deve terminar.

5.2.4 Resultados

Assim que a simulação termina, os resultados são gerados e apresentados na tela, mas também podem ser redirecionados para um arquivo. Caso existam muitos processos é altamente recomendado redirecionar a saída do terminal para um arquivo de texto, para não correr o risco do terminal cortar partes dos resultado.

Todos os valores relacionados a tempo estão em *clocks* (unidade de tempo de simulação). São apresentados 4 informações gerais antes das informações de cada processo, e são elas:

- *Total Simulation Time*: o tempo total da simulação.
- CPU *Idle Time*: a quantidade de *clocks* em que não houve processamento, em uma comparação entre escalonadores, quanto maior este valor, pior o escalonador.
- CPU *Not Idle Time*: a quantidade de *clocks* que o processador da simulação processou.
- *Context Switches*: mostra a quantidade de trocas de contextos que foram executadas durante a simulação.

Para cada processo é exibido um cabeçalho e os seguintes itens. O cabeçalho contém o PID e o nome do programa entre colchetes. Os dados são compostos na seguinte ordem:

- *Time Waiting*: Tempo total que o processo ficou esperando pelo processamento.
- *Time Running*: Tempo que o processo ficou sendo executado.
- *Time Blocked*: Tempo que o processo ficou bloqueado em I/O.
- *Life Span*: Tempo de vida do processo.
- *State*: Apresenta o estado que o processo se encontra ao terminar a simulação. O padrão é DESTROYED, pois um término bem sucedido significa que todos os processos devem ter sido finalizados, mas é possível o escalonador pedir o fim da simulação por algum motivo, por isso a necessidade de apresentar o estado.

No Quadro 4 um exemplo de resultado com os dois escalonadores que vêm programados com o *framework*, à esquerda o “sched_roundrobin” e à direita o “sched_fifo”, ambos com mesma lista de programas para executar.

Quadro 4 - Exemplo de saída do protótipo

```
1   Total Simulation Time: 190
2   CPU Idle Time: 64
3   CPU Not Idle Time: 126
4   Context Switches: 0
5
6   Process 0 [cpuIntensive]
7     Time Waiting = 0
8     Time Running = 50
9     Time Blocked = 0
10    Life Span = 50
11    State = DESTROYED
12  Process 1 [cpuIO]
13    Time Waiting = 50
14    Time Running = 20
15    Time Blocked = 10
16    Life Span = 80
17    State = DESTROYED
18  Process 2 [ioIntensive]
19    Time Waiting = 80
20    Time Running = 3
21    Time Blocked = 27
22    Life Span = 110
23    State = DESTROYED
24  Process 3 [ioIntensive]
25    Time Waiting = 110
26    Time Running = 3
27    Time Blocked = 27
28    Life Span = 140
29    State = DESTROYED
30  Process 4 [cpuIntensive]
31    Time Waiting = 140
32    Time Running = 50
33    Time Blocked = 0
34    Life Span = 190
35    State = DESTROYED
```

```
1   Total Simulation Time: 126
2   CPU Idle Time: 0
3   CPU Not Idle Time: 126
4   Context Switches: 21
5
6   Process 0 [cpuIntensive]
7     Time Waiting = 71
8     Time Running = 50
9     Time Blocked = 0
10    Life Span = 121
11    State = DESTROYED
12  Process 1 [cpuIO]
13    Time Waiting = 51
14    Time Running = 20
15    Time Blocked = 10
16    Life Span = 81
17    State = DESTROYED
18  Process 2 [ioIntensive]
19    Time Waiting = 24
20    Time Running = 3
21    Time Blocked = 27
22    Life Span = 54
23    State = DESTROYED
24  Process 3 [ioIntensive]
25    Time Waiting = 25
26    Time Running = 3
27    Time Blocked = 27
28    Life Span = 55
29    State = DESTROYED
30  Process 4 [cpuIntensive]
31    Time Waiting = 76
32    Time Running = 50
33    Time Blocked = 0
34    Life Span = 126
35    State = DESTROYED
```

Fonte: do Autor.

5.3 DETALHES DE IMPLEMENTAÇÃO

Esta seção descreve as ideias e técnicas utilizadas nas principais áreas do código fonte referente à simulação e ao framework.

5.3.1 Recursos Modernos de Linguagem

Nas versões mais novas do C++ não é necessário a utilização explícita de ponteiros, new e delete para fazer um programa eficiente e que utilize alocação dinâmica. O protótipo como prova disso não utiliza nenhum destes três recursos de maneira explícita.

5.3.2 Saída para o usuário

As mensagens, erros, resultados e operações que envolvem escrever na saída padrão do sistema é feita somente pelo main e pelas funções auxiliares de console.h, estas funções auxiliam na saída com cor para o terminal. Elas foram programadas e testadas para ambientes Linux e Windows, com versões únicas de setColor() e unsetColor() para cada sistema.

5.3.3 Carregamento de programas

Para otimizar o carregamento dos programas, cada programa é carregado uma única vez, e o algoritmo que faz esse carregamento está no Quadro 5. Isto é feito através de um map, o qual relaciona o nome do programa como chave e o Program como valor (linha 39). Na linha 46 ocorre a verificação se o programa já foi carregado. A leitura do arquivo com as instruções do programa ocorre no construtor da classe Program, e a construção de um objeto de Program ocorre na linha 47 com a chamada de emplace.

Quadro 5 - Código do carregamento dos programas

```
39      std::unordered_map<std::string, Program> loaded_programs;
  . . .
45      while(in >> str_program >> str_priority){
46          if(!loaded_programs.count(str_program)){
47              loaded_programs.emplace(str_program, str_program);
                                        // chave, arquivo com o programa
48          }
```

Fonte: do Autor.

5.3.4 Otimização de Espaço para os Programas e Instruções

Cada processo executado possui uma cópia de seu programa correspondente, isso porque a CPU utiliza diretamente as instruções da cópia do programa para executar e atualizar o tempo restante das instruções. Se o processo tiver uma referência para o programa, quando a

CPU alterar um processo, estará alterando o programa e como consequência todos os processos que utilizam este programa.

Devido a isso, cada programa possuí um nome e um vetor de instruções, cujo tamanho é exatamente o número de instruções. O nome do programa é utilizado ao mostrar os resultados no fim da simulação, e as instruções são utilizadas para a simulação. Como um programa possuí diversas instruções, então cada processo possuí uma cópia de todas as instruções do programa, independente se é o mesmo programa para vários processos.

Tudo isso significa que o tamanho que todos os processos irão ocupar na memória RAM do computador é diretamente proporcional ao tamanho de cada instrução, considerando isto, as instruções foram otimizadas para cada uma ocupar 3 bytes, mas de fato ocupam 4 bytes devido ao alinhamento efetuado pelo compilador. A estrutura da instrução que está em instruction.h pode ser visualizada no Quadro 6.

Quadro 6 - Estrutura Instruction

```
 6  struct Instruction
 7  {
 8      Instruction(std::string const& str);
 9
10      enum opcode : char {PROCESS, IO} command{};
11      unsigned short time{0};
12  };
```

Fonte: do Autor.

Os 3 bytes são compostos por:

- *command* – variável do tipo opcode, que é um enum com tamanho de um char, e por definição é 1 byte;
- *time* – variável do tipo unsigned short, usualmente 2 bytes.

A escolha de *time* ser do tipo unsigned short foi por questões de espaço e que deve ser um valor positivo, como efeito tem-se que o número máximo de *clocks* suportado por cada instrução é o valor máximo suportado pelo tipo unsigned short (65.535).

5.3.5 Ciclo da simulação

Após fazer toda a leitura do arquivo de entrada, programas, validações e a criação dos processos, a simulação é iniciada pelo main ao chamar o método start de Manager (no Quadro

7). Este método, conforme foi modelado e disponível em manager.cpp, inicializa o escalonador chamando o método setup (linha 31) e então inicia o ciclo de *clock* da simulação (linha 33).

Quadro 7 – Método start de Manager

```
21  void Manager::start(){
  . . .
30      running = true;
31      scheduler.setup(process_list);
32
33      while(running){
34          nextClock();
35      }
36  }
```

Fonte: do Autor.

Como visto no Quadro 7, o método nextClock (no Quadro 8) é o responsável em realizar cada *clock* da simulação. No final de cada *clock* da simulação, é feito a execução do método checkRunning, o qual verifica se a simulação deve continuar ou terminar. Como o escalonador somente será chamado caso a simulação não terminar, é importante que quem vai programar o escalonador conheça em quais condições isso pode ocorrer.

Quadro 8 – Método nextClock de Manager

```
38  void Manager::nextClock(){
39      scheduler.update();
40      updateTimers();
41      doIO();
42      cpu.execute();
43      running = checkRunning();
44      didContextSwitch = false;
45  }
```

Fonte: do Autor.

Existem três condições, e são elas:

1) O escalonador executou o método callFinish de Manager, solicitando o fim da simulação.
2) O tempo de simulação é igual ao maior número positivo suportado nativamente, consultar Tabela 3.
3) Todos os processos estão no estado destruído.

Tabela 3 - Maior tempo de simulação possível para sistemas 32 bits e 64 bits

64 bits	18.446.744.073.709.551.615 *clocks*
32 bits	4.294.967.296 *clocks*

Fonte: do Autor.

A execução do escalonador ocorre se as três condições forem todas falsas, portanto um escalonador não precisa verificar se todos os processos estão destruídos, por outro lado, o escalonador é responsável por manter sua cópia local com a lista dos processos, que pode ser uma pilha, fila ou alguma outra estrutura, e desse modo, atualizá-la conforme o necessário, como efetuar a remoção de processos destruídos.

5.3.6 Utilização de ProcPtr

ProcPtr foi criado para facilitar a manipulação de ponteiros para processos, já que estes são encapsulados pela classe shared_ptr, garantindo posse compartilhada do objeto, desse modo o processo só é destruído quando ninguém mais precisa do objeto.

5.3.7 Implementação Sched_RoundRobin e Sched_FIFO

Para facilitar o entendimento de como o framework funciona são fornecidas duas classes de exemplo, uma implementando um escalonador FIFO e outra implementando um escalonador Round Robin. Ambos herdam da classe Scheduler, que se trata de uma classe genérica, para informações de como criar um novo escalonador, consulte o Apêndice A.

5.3.7.1 Sched_FIFO

O algoritmo de escalonamento FIFO tenta seguir a lista de processos por ordem de chegada, e para isso, internamente Sched_FIFO armazena a lista de processos como uma fila, utilizando a classe std::queue do C++, e implementa um método auxiliar chamado ‘next’ para escolher o próximo elemento, ambos os métodos setup e update são obrigatórios.

No método setup, no Quadro 9, cada elemento do vetor de processos é adicionado na fila (linha 9). O tipo da variável i é identificada pelo compilador, sendo uma referência para um ProcPtr.

Quadro 9 - Métodos principais de Sched_FIFO

```
 7  void Sched_FIFO::setup(std::vector<ProcPtr> const&  p){
 8      for(auto& i : p){
 9          q.push(i);
10      }
11  }
12
13  void Sched_FIFO::update(){
14      ProcPtr p = next();
15
16      if(p->getState() == READY && manager->isCPUEmpty()){
17          manager->sendToCPU(p);
18      }
19  }
```

Fonte: do Autor.

No método update, no Quadro 9, é chamado o método next (linha 14) para obter o próximo da fila, então é feito uma verificação se o processo está preparado para execução (linha 16) e se a CPU está vazia (linha 16), caso essas duas condições sejam satisfeitas, o escalonador envia o processo para execução, senão não faz nada. É possível enviar para a CPU um processo somente se estas duas condições forem verdadeiras, senão o *framework* mostrará um erro durante a execução.

Quadro 10 – Método next de Sched_FIFO

```
21  ProcPtr Sched_FIFO::next(){
22      if(!q.empty()){
23          if(q.front()->getState() == DESTROYED){
24              q.pop();
25              return next();
26          } else {
27              return q.front();
28          }
29      } else {
30          return nullptr;
31      }
32  }
```

Fonte: do Autor.

O método next (disponível no Quadro 10) aplica as regras padrões de um escalonador FIFO, remove o primeiro elemento (linha 24) se o processo já terminou a execução, senão retorna o primeiro elemento. Seguindo esse algoritmo, o processo ficará sendo o próximo a executar até que termine sua execução, mesmo que entre em estado de I/O.

Cabe aqui destacar que as linhas 22, 29, 30 e 31 não são necessárias, pois nunca o escalonador irá ser executado com todos os processos em estado destruído, portanto, como o algoritmo só remove elementos destruídos, sempre haverá alguém na fila.

5.3.7.2 Sched_RoundRobin

O algoritmo Round Robin busca alternar o acesso ao processador a todos os processos dando-lhes tempos regulares. A implementação de Sched_RoundRobin utiliza as ideias da implementação de Sched_FIFO, como remover processos destruídos e utilizar uma fila para armazenar os processos. A principal diferença com Sched_FIFO é que Sched_RoundRobin utiliza troca de contextos. O método frontToBack é utilizado para enviar o primeiro da fila para o fim da fila.

No método update, no Quadro 11, é feito a verificação se o processador está livre (linha 14), e se estiver, o algoritmo seleciona o próximo processo da fila que estiver preparado para executar (estado READY) e o envia para execução (linha 17). Caso o processador estiver ocupado e o tempo consecutivo que o processo está executando atingir cinco (linha 24), então é feito a troca de contexto (linha 25), enviado o processo que estava executando para o fim da fila (linha 26), e selecionado um próximo processo para executar (linha 27 e 28).

Quadro 11 - Método update de Sched_RoundRobin

```
13  void Sched_RoundRobin::update(){
14      if(manager->isCPUEmpty()){
15          ProcPtr p = next();
16          if(p != nullptr){
17              manager->sendToCPU(p);
  . . .
22      }
23
24      if(q.front()->get_time_exec() == 5){
25          manager->contextSwitch();
26          frontToBack();
27          ProcPtr p = next();
28          manager->sendToCPU(p);
29      }
30  }
```

Fonte: do Autor.

O método next, no Quadro 12, percorre a fila até encontrar alguém para executar (linha 34), colocando os elementos no final da fila caso não possam executar (linhas 40 e 41), e se não encontrar nenhum processo (todos em I/O), retorna nullptr (por isso a verificação na linha 16).

Quadro 12 - Método next de Sched_RoundRobin

```
32  ProcPtr Sched_RoundRobin::next(){
33      unsigned long i=0;
34      while(i<q.size()){
35          if(q.front()->getState() == READY){
36              return q.front();
37          } else if(q.front()->getState() == DESTROYED){
38              q.pop();
39              continue;
40          } else {
41              frontToBack();
42              i++;
43          }
```

Fonte: do Autor.

6 CONSIDERAÇÕES FINAIS

O objetivo principal foi alcançado, a construção de um *framework* para experimentação e ensino de gerências de sistemas operacionais, que facilite a construções de algoritmos para a gerência de processador.

Com a utilização deste *framework* espera-se que a prototipação e construção de algoritmos para as gerências de um sistema operacional seja muito mais agradável e fácil do que a construção direta em um sistema para uso final. Também que aulas de disciplinas como Sistemas Operacionais não sejam apenas teóricas, mas que possam ter assuntos de programação envolvida, com a possibilidade de implementação dos algoritmos.

Por mais que no protótipo não foi feita a implementação da Gerência de Memória, com a base já existente a adição dessa gerência precisará menos esforço se comparado com a criação a partir do zero. Um exemplo seria a adição de comandos como “alloc” e “free” para os programas, onde toda a estrutura para as instruções já está construída. Outro ponto é que toda a lógica envolvida na troca de ciclos da simulação já está programada.

No protótipo buscou-se seguir técnicas e mecanismos que ofereçam maior desempenho para a aplicação como um todo, e também boas práticas de programação em C++, como a não utilização de ponteiros, new e delete.

Além de ser uma ferramenta para a gerência de processador, o *framework* pode ser utilizado para aprendizado da linguagem de programação C++, como também para criação de estruturas de dados que armazenam a lista de processos com determinadas regras especiais de modo que estas estruturas possam ser reutilizadas entre os algoritmos.

Embora a realização dessa simulação por software tenha sido uma simplificação da realidade, o desenvolvimento mostra toda a complexidade que envolve a área de Sistemas Operacionais, inclusive nas áreas mais básicas como gerência de processador e de um escalonador.

Na subseção de trabalhos futuros é deixado ideias para a continuação do desenvolvimento.

6.1 TRABALHOS FUTUROS

Seria de grande interesse o aumento da complexidade da simulação, visando representar um sistema operacional com maiores detalhes e não apenas a gerência de processador, portanto algumas sugestões de adição à implementação são:

- Uma gerência de memória, que inclua o acesso à memória RAM;
- Possibilitar a entrada dinâmica de novos processos durante a simulação;
- Custo da troca de contexto de cada processo, durante o escalonamento;
- Internacionalização do programa;
- Saídas oferecendo vários formatos, como xml, csv, visando facilitar o uso da saída para ferramentas de comparação e geração de gráficos;
- Criação de uma interface gráfica amigável para usuários mais leigos.

REFERÊNCIAS

BACI, BACI Docs. Disponível em: <http://inside.mines.edu/~tcamp/baci/bacidocpdf.tar.gz> Acesso em: 5 mar. 2014.

GOOGLE CODE. FAQ. **Google Code**, 2014. Disponível em: <https://code.google.com/p/support/wiki/FAQ>. Acesso em: 9 nov. 2014.

JADHAV, S. Advanced Computer Architecture and Computing. Ed. Technical Publications Pune, 2009.

LI, K., LI, Q. e SHIH, T. K. Cloud Computing and Digital Media: Fundamentals, Techniques, and Applications. CRC Press, 2014.

MAIA, Luiz P. MACHADO, F. B. Um framework construtivista no aprendizado de Sistemas Operacionais – uma proposta pedagógica com o uso do simulador SOsim. Site de Luiz Paulo Maia. Disponível em: <http://www.training.com.br/sosim/sbcwei04.pdf> Acesso em: 22 mar. 2014.

MAIA, Luiz P. SOsim: **Simulador para o ensino de sistemas operacionais**. 2001. 97f. Dissertação (Mestrado em Ciências em Informática) – Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2001. Disponível em: <http://infocao.dominiotemporario.com/doc/sosim_tese.pdf> Acesso em: 22 mar. 2014.

NULL, L.; LOBUR, J. **Princípios Básicos de Arquitetura e Organização de Computadores**. 2. ed. Porto Alegre: Bookman, 2006.

OLIVEIRA, R. S.; CARISSIMI, A. S.; TOSCANI, S. S. **Sistemas Operacionais**. Série Livros Didáticos Informática UFRGS. Ed. Porto Alegre: Bookman, v. 11, 2010.

OPEN SOURCE INICIATIVE. MIT License. **Open Source Iniciative**, 2014. Disponível em: <http://opensource.org/licenses/MIT>. Acesso em: 17 nov. 2014.

SAMANTA D. Classic Data Structures. 2 ed. New Delhi. PHI Learning Pvt, 2009.

SHARMA V., VARSHNEY M. SHARMA S. Design and Implementation of Operating System. New Delhi ed. Laxmi Publications: 2010.

SILBERSCHATZ, A. **Sistemas Operacionais com Java**. São Paulo: Elsevier Brasil, 2008.

STALLINGS, W. Operating Systems – Internals and Design Principle. 7 Edição, Pearson Education, 2012.

TANENBAUM, A. S. **Sistemas Operacionais**: Projetos e Implementação. Porto Alegre: Ed. Bookman, 2002.

TANENBAUM, A. S. **Sistemas Operacionais**: Projetos e Implementação. Porto Alegre: Ed. Bookman, 2008.

APÊNDICES

APÊNDICE A – MANUAL DE UTILIZAÇÃO DO FRAMEWORK

O QUE É O FRAMEWORK?

GROSS é um framework para construir e experimentar algoritmos, de uma maneira facilitada, para gerência de processador, existente em sistemas operacionais. Esses algoritmos criados funcionam e interagem em uma simulação simplificada, e são responsáveis por decidir qual processo deve utilizar o processador. Os algoritmos criados são compilados junto com o framework, e são escolhidos em tempo de execução por um arquivo de configuração que é fornecido ao executar a aplicação. Estes algoritmos serão chamados de escalonadores, pois sua principal função é escolher qual processo na simulação deve executar, mas são escalonadores de alto nível, pois devem se preocupar também em como armazenar a lista de processos, verificar se os processos são válidos para execução e fazer requisições de troca de contexto.

GUIA PARA UTILIZAR A APLICAÇÃO

O framework é compilado em uma aplicação para terminal, o framework precisa de um arquivo de configuração para saber como irá realizar a simulação (qual escalonador irá utilizar e quais programas serão executados junto de sua prioridade). Este arquivo de configuração deve ser fornecido ao executar a aplicação pelo terminal. Suponha o seguinte arquivo de configuração chamado ‘exemplo.txt’ onde 5 é a prioridade do programa1 e 7 a prioridade do programa2, para simular esse cenário, execute pelo terminal “GROSS exemplo.txt”.

```
escalonador nome_do_escalonador
programa1 5
programa2 7
```

O framework busca cada programa em um arquivo, no qual, o nome é o nome do programa e a extensão “.prog”, na pasta “programs” que deverá estar junto da aplicação.

Os programas são formados por instruções, cada uma deve estar em uma linha separada, e é composta por uma operação e o tempo necessário, em *clocks* de simulação, para ser realizada. É possível executar 2 tipos de operações, uma é “process” que indica que o

programa precisa de processamento, e a outra é “io” que indica que o programa precisa realizar uma atividade de I/O. Abaixo um exemplo de programa.

```
process 20
io 10
```

Carregados os programas segundo o arquivo de configuração fornecido, a aplicação executa a simulação e mostra um resultado após terminar. Com este resultado pode-se saber se o escalonador escolhido obteve um bom desempenho no cenário. O framework não é capaz de afirmar se o desempenho foi bom ou não, é mostrado todas as métricas disponíveis (tempos da CPU parada ou processando, trocas de contexto realizadas, tempos de cada processo no processador, esperando ou em I/O e o estado de cada processo), e então cabe a nós decidirmos se atendeu as expectativas.

ENTENDIMENTO DO ESCALONADOR

Antes de programar um escalonador é importante saber qual sua função e a relação dele com outras classes do código fonte. Manager é a classe que controla a simulação, conhece os processos que existem, atualiza os tempos, se relaciona com a CPU, inicializa o escalonador e chama o escalonador a cada *clock* para tomar alguma decisão. Manager deixa implícito qual decisão o escalonador deve tomar, ou seja, o escalonador pode fazer o que bem entender.

Manager deixa a disposição alguns métodos para o escalonador, que são: sendToCPU, contextSwitch, isCPUEmpty, get_time_elapsed e callFinish. Os métodos são explicados abaixo:

- callFinish – quando este método é chamado, Manager irá terminar a simulação, portanto não haverá próximo *clock* e os resultados serão mostrados na tela. Um escalonador bem programado não deve chamar este método, já que todos os processos irão encerrar normalmente conforme o andamento da execução, mas ele existe para suprir alguns casos, como parar a simulação após n *clocks* e verificar se todos os processos tiveram acesso à CPU.
- get_time_elapsed – este método retorna quantos *clock* já se passaram na simulação. É possível utilizá-lo como um contador global para tomar decisões.

- isCPUEmpty – retorna verdadeiro se a CPU não estiver executando algum processo, e falso caso contrário.
- contextSwitch – método utilizado para retirar o processo atual que está em execução na CPU. Não é possível chamar duas vezes este método no mesmo clock.
- sendToCPU – deve ser usado para colocar um processo em execução na CPU, só pode ser chamado se o processador não estiver sendo utilizado.

Além da classe Manager, um escalonador deve se relacionar através de herança com a classe Scheduler, isto para que o escalonador seja visível ao framework.

A ultima classe importate para um escalonador é a classe Process, esta obter informações sobre os processos, como estado, tempo de cpu, tempo de io, tempo de espera, entre outros tempos e também para alterar a prioridade caso necessário (como algoritmos que implementam políticas de envelhecimento).

Portanto, um escalonador apenas se relaciona com Scheduler, Manager e Process. Entendendo com quais classes um escalonador se relaciona e como se relaciona, é possível acompanhar a criação de um escalonador a seguir.

GUIA PARA CRIAR UM ESCALONADOR – COM EXEMPLO.

Para criar um escalonador é necessário programar uma classe que em sua hierarquia de herança possua Scheduler no topo, isso significa que todos os escalonadores criados devem herdar a classe Scheduler ou alguma classe que herde de Scheduler.

A classe Scheduler está no arquivo scheduler.h enquanto a sua respectiva implementação está no arquivo scheduler.cpp. É necessário que o escalonador criado inclua scheduler.h para ser possível a herança e inclua manager.h para utilizar o objeto manager (Gerente da Simulação) que é disponibilizado por Scheduler através de um atributo protected.

Recomenda-se que os novos escalonadores, assim como os dois exemplos programados Sched_FIFO e Sched_RoundRobin, sejam divididos em cabeçalho (arquivo .h) e implementação (arquivo .cpp).

Para exemplificar o processo de criação de um escalonador, será criado um escalonador FIFO com prioridade, o qual manda executar primeiro os processos com maior prioridade.

Cabeçalho do novo Escalonador

O cabeçalho deve possuir um protetor de inclusão para evitar que seja incluído duas vezes, causando erros de compilação. E como explicado, o cabeçalho do novo escalonador deve incluir scheduler.h para efetuar a herança, além de manager.h. Seguindo esses passos o arquivo deverá estar como no quadro abaixo:

```
#ifndef FIFO_PRIORIDADE_H
#define FIFO_PRIORIDADE_H

#include "scheduler.h"
#include "manager.h"

#endif
```

Deve-se então declarar a herança

```
class FIFO_Prioridade : public Scheduler
```

Os dois métodos virtuais "setup" e "update" de Scheduler devem ser declarados e posteriormente implementados. A palavra-chave “override” é opcional, a adição dela ajuda a evitar erros de programação, pois o compilador verificará se realmente está sobrescrevendo um método de Scheduler.

```
        void setup(std::vector<ProcPtr> const& p) override;
        void update() override;
```

O método setup é executado logo antes do início da simulação, ele deve ser utilizado para o escalonador guardar a lista de processos da maneira que achar conveniente, e devido a isso, deve receber pôr parâmetro uma referência constante para um vector de ponteiros para processos, resumidamente um vetor de ponteiros para processos, e como é constante implica que não é possível alterar o parâmetro recebido e que deverá ser feito uma cópia nesse método da maneira que achar necessária.

O método update não recebe nada por parâmetro, e se trata do método que vai ser executado a cada *clock* da simulação, é onde o escalonador deve tomar as decisões. É importante notar que o método update não será chamado se a simulação terminar, e com isso podemos ter certeza de que pelo menos um processo estará vivo.

Esses são os únicos métodos obrigatórios para criar um escalonador.

No caso de nosso FIFO_Prioridade iremos utilizar uma fila (queue - que já está implementada no C++) para armazenar os processos. Abaixo está o fifo_prioridade.h.

```
#ifndef FIFO_PRIORIDADE_H
#define FIFO_PRIORIDADE_H

#include "scheduler.h"
#include "manager.h"
#include <queue>

class FIFO_Prioridade : public Scheduler
{
    public:
        FIFO_Prioridade();
        void setup(std::vector<ProcPtr> const& p) override;
        void update() override;
    private:
        std::queue<ProcPtr> fila{};
};

#endif
```

Implementação do novo Escalonador

Na implementação (arquivo cpp) deve-se instanciar o escalonador para que Scheduler registre a instância e o escalonador possa ser utilizado pela simulação.

```
FIFO_Prioridade fifo_prioridade_obj;
```

O construtor de Scheduler recebe o nome na qual o escalonador será registrado, portanto para registrar o escalonador como válido, em seu construtor faça como o seguinte:

```
FIFO_Prioridade::FIFO_Prioridade() : Scheduler("fifo_prioridade") {
```

Depois disso o escalonador já está preparado para ser chamado pela simulação, restando programar suas funcionalidades.

No setup de FIFO_Prioridade devemos guardar os processos na fila de acordo com sua prioridade, primeiro os processos com maior prioridade até os de menor prioridade. Para isso vamos criar uma variável chamada maiorPrioridade e descobrir a maior prioridade dentre os processos, como no seguinte código:

```
void FIFO_Prioridade::setup(std::vector<ProcPtr> const&  p){
    int maiorPrioridade = 0;
    for(ProcPtr i : p){
        if(i->getPriority() > maiorPrioridade){
            maiorPrioridade = i->getPriority();
        }
    }
```

Após este algoritmo executar, a variável maiorPrioridade possuirá a maior prioridade entre os processos. Agora é possível inserir na fila (método push de queue) os processos com prioridade mais alta, aqueles que possuem mesmo valor de prioridade que maiorPrioridade, até os processos de prioridade 1, portanto o seguinte código mostra essa inserção:

```
    for(int prioridade = maiorPrioridade; prioridade>0; prioridade--){
        for(ProcPtr i : p){
            if(i->getPriority() == prioridade){
                fila.push(i);
            }
        }
    }
```

Não há mais o que fazer em setup, pois a fila já está criada com os processos na ordem correta. Agora é necessário implementar update, o método que executará a cada *clock*. Primeiro precisamos remover o processo caso ele terminou a execução, passando para o próximo processo executar, seguindo assim a lógica FIFO. Abaixo o código dessa remoção:

```
if(fila.front()->getState() == DESTROYED){
    fila.pop();
}
```

Como o processo que terminou a execução foi removido, e na lógica de um FIFO o próximo processo nunca vai ter executado, temos certeza que ele estará pronto para execução, portanto apenas um ‘if’ já é o bastante.

Agora que a verificação se o primeiro processo da fila terminou a execução já foi feita, é necessário criar a lógica para enviar um processo para execução. Para um processo ir para execução é necessário que a CPU não esteja sendo utilizada por outro processo (nesta situação poderia ser utilizada troca de contexto, mas não é o caso para esse FIFO) e que o processo esteja preparado para executar (não esteja em I/O ou destruído), a seguir o código que coloca um processo em execução.

```
if(fila.front()->getState() == READY && manager->isCPUEmpty()){
    manager->sendToCPU(fila.front());
}
```

Com isso o escalonador FIFO com prioridade está pronto e já pode ser testado, abaixo fifo_prioridade.cpp. Para utilizar este escalonador, no arquivo de configuração coloque “escalonador fifo_prioridade”, observe que fifo_prioridade é o nome escolhido no construtor.

```
#include "fifo_prioridade.h"

FIFO_Prioridade fifo_prioridade_obj;

FIFO_Prioridade::FIFO_Prioridade() : Scheduler("fifo_prioridade") {}

void FIFO_Prioridade::setup(std::vector<ProcPtr> const&  p){
    int maiorPrioridade = 0;

    for(ProcPtr i : p){
        if(i->getPriority() > maiorPrioridade){
            maiorPrioridade = i->getPriority();
        }
    }

   for(int prioridade = maiorPrioridade; prioridade>0; prioridade--){
        for(ProcPtr i : p){
            if(i->getPriority() == prioridade){
                fila.push(i);
            }
        }
    }
}

void FIFO_Prioridade::update(){
    if(fila.front()->getState() == DESTROYED){
        fila.pop();
    }

    if(fila.front()->getState() == READY && manager->isCPUEmpty()){
        manager->sendToCPU(fila.front());
    }
}
```

EXEMPLOS DE COMO PODE SER UTILIZADO

A utilização mais trivial é a criação de algoritmos e compará-los de acordo com os resultados, mas além disso podemos observar outros casos como a seguir.

Em aulas de sistemas operacionais onde deseja-se tarefas práticas de programação, nessa situação o professor pode pedir para que alunos programem um determinado escalonador seguindo certas regras.

É possível criar várias camadas de abstração dentro do código fonte devido a liberdade que a programação oferece, exemplo: para facilitar o professor pode criar classes intermediárias que possuam determinados métodos programados, desse modo o aluno herda a

classe do professor, a qual herda Scheduler, um caso é fazer pequenas alterações em um algoritmo Round Robin, o professor pode programar toda a lógica e pedir para os alunos encontrarem o melhor *quantum* em um determinado cenário.

Também é possível para praticar criação de estruturas que recebem processos de entrada, como uma lista especial que se auto ordena de acordo com os processos que são inseridos.

Devido ao fato de ser possível fazer saídas para o console durante o método update, é possível que o escalonador mostre suas ações em tempo de execução, ou ainda que peça em tempo real alguma informação para o usuário.

De maneira implícita pode ser utilizado para aprendizagem de Orientação a Objetos, Estruturas de Dados, Algoritmos e C++.