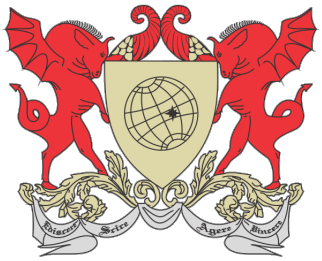


UNIVERSIDADE FEDERAL DE VIÇOSA
CAMPUS FLORESTAL

Trabalho Prático I
Sistemas Operacionais

Artur Souza Papa - 3886
Luciano Belo de Alcântara Júnior - 3897
Maria Theresa Henriques - 3486

Trabalho Prático I apresentado à disciplina de Sistemas Operacionais do curso de Ciência da Computação da Universidade Federal de Viçosa.

Florestal
20 de Junho de 2022

CCF 451 - Sistemas Operacionais

Trabalho Prático I

Artur Souza Papa - 3886
Luciano Belo de Alcântara Júnior - 3897
Maria Theresa Henriques - 3486

20 de Junho de 2022

Contents

1 Introdução

O presente trabalho tem como objetivo o entendimento e aplicação através de simulação, do que consiste um gerenciamento de processos em ambiente Linux. Para isso serão implementadas algumas funções, sendo as principais delas: (1) criação de processo; (2) substituição da imagem vigente no processo por uma imagem nova; e (3) realização de troca de contexto.

Dessa forma, o trabalho foi dividido em duas etapas, construção conceitual do que se espera e aplicação prática do que foi estudado. Assim, em um primeiro momento apresentaremos os principais conceitos que serão implementados em nossa simulação. Em seguida, serão apresentadas explicações ilustradas das funcionalidades codificadas no projeto.

2 Construção conceitual

2.1 Sistema operacional

Neste trabalho, construiremos nosso programa de simulação sobre o sistema operacional (SO) Linux. Linux é caracterizado por ser um SO de código aberto baseado em padrões Unix, desenvolvido por programadores voluntários e distribuído sob a licença pública GPL (General Public License).

O SO Linux é baseado em três componentes primários, sendo eles o kernel, o sistema de bibliotecas e o sistema de utilidades. Seu kernel possui arquitetura monolítica e é responsável pela maioria das atividades do sistema operacional. Já o sistema de bibliotecas contém os programas que decidem qual aplicação ou utilidade acessar os recursos do kernel. Por fim, o sistema de utilidades são programas responsáveis por realizarem atividades específicas.

2.2 Processos

Segundo Tanenbaum e Austin [1] um processo pode ser considerado um programa em execução, contendo informações como: (1) identificador de processo; (2) identificador de seu processo pai; (3) contador do programa; (4) prioridade do processo; (5) estado em que se encontra; (6) tempo de início; (7) tempo gasto na CPU; e o (8) tipo da operação que o processo deverá realizar.

Para além, processos estão sujeitos a mudanças de contexto, que ocorre quando há uma troca do estado de execução de um processo por outro. Diz respeito ao contexto do processo as atribuições supracitadas de identificação do processo, identificação do processo pai, privilégios no sistema, e limites de alocação do processo.

Quanto ao estado de um processo, o mesmo poderá assumir apenas um valor de estado por vez, sendo eles: (1) em execução; (2) pronto; e (3) bloqueado. Caso esteja em execução, o processo estará sendo utilizado pela CPU. Já em estado pronto, o processo fica no aguardo para ser executado. Por fim, no estado bloqueado o processo espera por algum evento externo para prosseguir para o estado pronto.

2.3 Chamadas de sistema

Chamadas de sistema são funções usadas por aplicações de modo a solicitar a execução de determinado serviço ao kernel do sistema operacional, ou seja, são instruções de maior privilégio que requerem serviços ao núcleo do sistema operacional sobre o qual está executando.

Ao executar uma chamada de sistema, o aplicativo solicita permissão ao kernel, através de um pedido de interrupção, que pausa o atual processo e transfere o controle para o modo núcleo. Assim, o sistema operacional salva todo o contexto do processo interrompido, verifica as permissões envolvidas no pedido e autoriza, caso a resposta seja afirmativa, o processo a executar o serviço pedido. Após terminar a execução, o sistema retorna o controle para o

processo, colocando-o novamente na fila de processos prontos para a execução [2]. Neste trabalho utilizaremos quatro chamadas de sistema presentes no sistema operacional Linux, sendo elas: (1) fork(); (2) wait(); (3) pipe(); e (4) sleep(). Será dada uma explicação breve de cada uma dessas funções abaixo.

2.3.1 fork()

O fork() é a função utilizada para um novo processo em sistemas Linux. Ao ser executada, a função cria um processo filho com muitas atribuições semelhantes ao processo pai, possuindo as mesmas variáveis, registros, descritores de arquivos e tudo mais. Entretanto, há duas diferenças importantes entre os processos pai e filho que são as informações de controle, que estão presentes no bloco de controle do processo filho.

O fork() é uma das chamadas de sistema mais usadas durante o processo de gerenciamento de processos, pois permite que um novo processo seja criado sem que um novo programa seja executado. O novo subprocesso executa o mesmo programa que o processo pai estava a executar.

2.3.2 wait()

A função wait() possui a atribuição de suspender o processo pai que está em execução, colocando-o em um estado de bloqueio no sistema operacional, até que seu processo filho seja terminado, registrando a mudança de estado do programa. Assim, caso haja um processo filho, a função wait() retorna o identificador do processo filho após o seu término.

2.3.3 pipe()

Já o pipe() é uma função de conexão entre dois processos, de tal forma que a saída padrão de um processo se torna a entrada padrão do outro processo, sendo assim, ela é utilizada para comunicação entre processos relacionados.

O pipe é uma uma comunicação unilateral, podendo ser usado pelo processo de criação bem como os seus demais processos filhos ou o canal relacionado a ele.

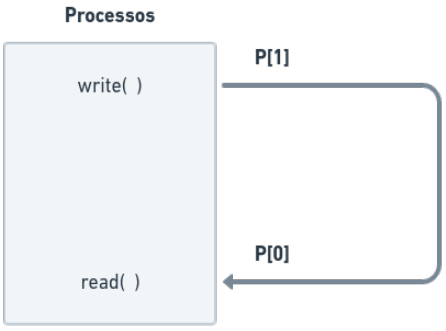


Figure 1: Chamada Pipe

2.3.4 sleep()

Por fim, a função sleep() é responsável por colocar o processo em modo de espera por um tempo determinado de segundos. Após esse tempo especificado ser finalizado, o programa retorna ao processo que antecede a chamada dessa função.

3 Desenvolvimento

O primeiro passo na criação da aplicação de gerenciamento de processo foi definir quais seriam os elementos básicos para o conjunto de atividades que este trabalho se propõe. Em seguida, com as estruturas prontas, foram implementadas as funções objeto deste trabalho e as funções complementares necessárias.

Para que fosse possível entender e simular o funcionamento de um gerenciador de processos em um sistema operacional Linux, o trabalho foi desenvolvido em cinco etapas, sendo elas: delimitações e escolhas de implementação, construção das estruturas de dados, construção das funções pedidas no projeto e suas auxiliares, realização de testes e resultados do programa.

3.1 Implementação

Para o desenvolvimento dos códigos, foi utilizado o sistema operacional Ubuntu, a linguagem de programação C, o compilador GCC e o editor de código-fonte Visual Studio Code.

Tanto o sistema operacional quanto a linguagem de programação foram escolhidos devido às possibilidades que os mesmos oferecem por ser, respectivamente, um sistema de código aberto com ampla documentação e por ser uma linguagem de propósito geral com alta flexibilidade de uso.

Ademais, vale ressaltar que o trabalho todo foi feito em inglês, desta maneira, temos que o nome das variáveis, funções, arquivos e prints estão todos na língua inglesa.

3.2 Estrutura de projeto

O trabalho foi desenhado com uma estrutura organizacional de pastas de modo a separar os arquivos em: dados a serem usados no programa; executável; biblioteca do programa e, por fim, os arquivos código-fonte.

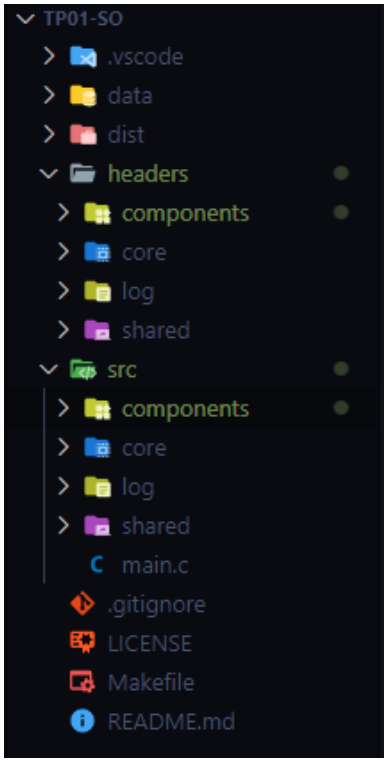


Figura 2: Estrutura de projeto

Temos quatro principais diretórios: *components*, *core*, *shared* e *log* (tanto no **src** como em **headers**). No diretório **components** temos as estruturas *cpu*, *tabela de processos*, *processo*

simulado e *programa* a ideia foi então agrupar os componentes principais da parte central do gerenciador de processos. No **core** encontra-se, como já dito anteriormente, a parte central do gerenciador de processos sendo os arquivos *escalonador*, *processo controle* e *gerenciador de processos*. Já no diretório **shared** encontra-se a implementação da estrutura de dados *fila*. E por fim, no diretório **log** encontra-se os arquivos do *logger* da aplicação, que tem por finalidade gerar uma cor de print diferente para três situações principais: *debug*, *erro* e *comum*.

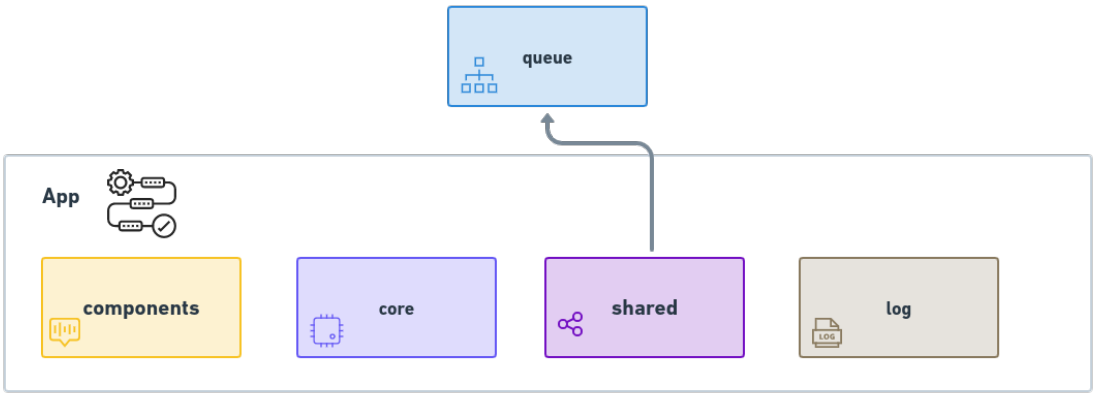


Figura 3: Arquitetura do projeto

Ademais, nas pasta *dist* e *data* encontra-se o executável e os arquivos *.txt* da aplicação, respectivamente.

3.3 Estrutura de dados

Para as estruturas de dados foram definidos 6 *structs* ao todo, sem contar a estrutura da fila, sendo assim, foram definidos os seguintes TAD's.

3.3.1 Program

A estrutura *program* se trata de uma *string* em que será lido o nome do arquivo a ser executado durante a simulação.

```
typedef struct {
    char file_name[30];
} program;
```

3.3.2 Simulated process

A estrutura *processo simulado* consiste do programa ao qual o processo pertence:

```
typedef struct
{
    int *memory;
    int number_of_vars;
    int pc;
    program programa;
} simulated_process;
```

3.3.3 Process table

A tabela de processos, por sua vez, fica a cargo de guardar as informações de cada processo em específico como o identificador do próprio processo, identificador do processo pai, a sua

prioridade, o estado em que se encontra (em execução, bloqueado e pronto), o seu tempo inicial, o tempo em que se encontrava a cpu em seu momento de criação, a referência sobre qual processo simulado pertence, o valor de memoria do processo simulado - variaveis que ele tem e foram declaradas na minha instrução do processo - e referência ao contador do programa simulado.

Além disso, vale notar que nesse arquivo temos um tipo *enum states* em que são enumerados os estados do processo, sendo eles *EXECUTING, BLOCKED, READY*.

```
typedef struct
{
    int pid;
    int ppid;
    int *pc;
    int *data_structure;
    int priority;
    states state;
    int inital_time;
    int cpu_time;
    simulated_process *process_reference;
} process_table;
```

3.3.4 CPU

Responsável por guardar as informações da instrução em execução, armazena dados como o valor do contador do programa, o tempo de programação, o tamanho necessário para armazenar os dados da instrução, o arquivo onde a instrução se encontra e os dados da instrução.

```
typedef struct
{
    program *program;
    int pc;
    int *memory;
    int *size_memory;
    int time;
} CPU;
```

3.3.5 Management

Para a estrutura *management* temos elementos para a controlar a transição dos estados, além de conter componentes para os algoritmos de escalonamento, como o *enum scheduler_policy*, vale notar também que há variáveis para calcular o tempo de execução, além daquelas do tipo *CPU* e *process_table*.

```
typedef struct
{
    CPU cpu;
    process_table *process_table;
    int executing_state;
    queue ready;
    queue blocked;
    scheduling scheduler;
    int time;
```

```
    scheduller_policy type_escalation_policy;
} management;
```

3.3.6 Schedulling

Por fim, tem-se a estrutura *schedulling* que foi usada para executar o escalonamento de múltiplas filas com classes de prioridade, onde temos uma fila pra cada nível de prioridade além de suas respectivas funções.

```
typedef struct
{
    queue *first;
    queue *second;
    queue *third;
    queue *fourth;
} scheduling;
```

3.4 Funções

3.4.1 Criar processo

A funçao criar processo (**create_new_process**) recebe como parâmetro o *gerenciador de processos*, o *número de instruções*, o *tamanho da tabela de processos* e o *pid*. Primeiramente cria-se um processo simulado, em seguida copia-se as informações de contador de programa, nome do arquivo (vetor de programa) e memória (vetor com variáveis) para o novo processo simulado, caso o gerenciador de processo já tenha itens na memoria do processo o novo processo simulado também as recebe. O próximo passo é ajustar os valores dos contadores de pograma e em seguida a tabela de processos é criada/realocada conforme a necessidade. Por fim, o processo filho criado (já com estado pronto), é adicionado na estrutura do tipo de escalonador escolhido.

3.4.2 Substituição da imagem atual por uma nova

A funçao substituição da imagem atual por uma nova (**replace_current_image_process**) recebe por parâmetro o *gerenciador de processos* e uma string com as instruções. Nesta função copia-se o caminho do arquivo para o programa na cpu do gerenciador de processos, além disso, o contador de programa e a memória do cpu são resetados.

3.4.3 Troca de contexto

A funçao de trocar contexto (**change_context**) recebe por parâmetro o *gerenciador de processos*, o index no próximo processo e estado inicial do processo. Essa função tem como finalidade copiar o estado do processo simulado em execução da CPU para a tabela de processos e copiar o estado do recém escalonado processo simulado da tabela de processos para a CPU.

3.5 Politícas de escalonamento

Ademais, vale citar que além da politíca de escalonamento de **Múltiplas Filas com Classes de Prioridade**, o algoritmo escolhido pelo grupo foi a **Primeiro a Chegar Primeiro a Ser Servido**. Com esse algoritmo, a CPU é atribuída aos processos na ordem em que a requisitam. Basicamente, há uma fila única de processos prontos. Quando a primeira tarefa entrar no sistema de manhã, ela é iniciada imediatamente e deixada executar por quanto tempo ela quiser. Ela não é interrompida por ter sido executada por tempo demais. À medida que as outras tarefas chegam,

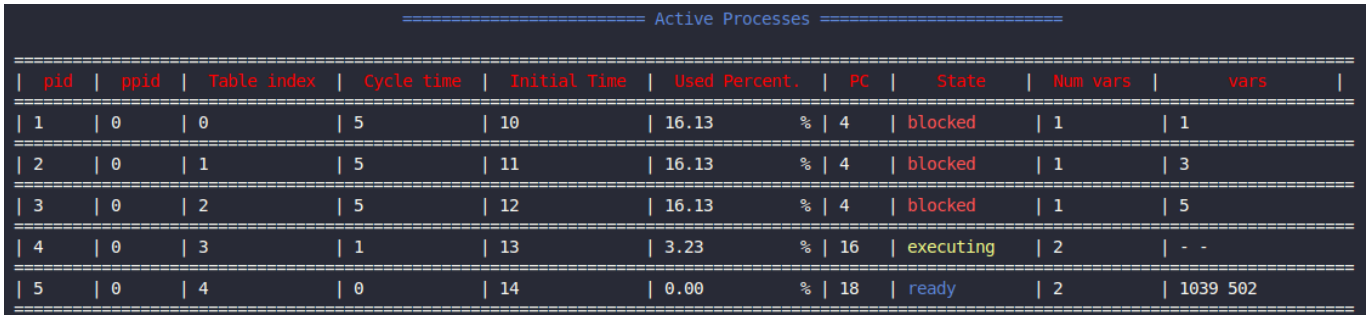
elas são colocadas no fim da fila. Quando o processo que está sendo executado é bloqueado, o primeiro processo na fila é executado em seguida. Quando um processo bloqueado fica pronto — assim como uma tarefa que chegou há pouco — ele é colocado no fim da fila, atrás dos processos em espera[1].Outrossim, pode-se dizer que um dos motivos para a escolha desse algoritmo pelo grupo foi devido ao fato de ser uma politíca fácil de entender e igualmente fácil de programar.

Sendo assim, para a implementação da politíca de escalonamento FCFS temos que primeiramente é analisado se há algum processo em execução, caso não haja é criada uma variável que recebe o processo que está com o estado pronto na fila e caso o próximo processo não esteja vazio ele é carregado na CPU.

```
void first_come_first_serve(management management)
{
    if (management.executing_state == -1)
    {
        int next_process_ii;
        next_process_ii = dequeue(&(management.ready));
        if (next_process_ii != -1)
        {
            load_cpu_process(&(management), next_process_ii);
        }
    }
}
```

3.6 Impressão

Os principais processos de impressão são o de *processos ativos* e o *debug*. A tabela de processos ativos possui os campos principais já citados, além disso, vale ressaltar que para cada um dos estados temos uma cor diferente. O processo de debug é definido por *define* sendo que foi de grande auxílio para visualização do grupo durante os testes. Além disso, temos a impressão final do tempo médio por ciclo antes do finalizar a execução do programa (*commando M*) e a impressão de erros. Tais tipos de impressão podem ser visíveis nas imagens a seguir:



Active Processes

pid	ppid	Table index	Cycle time	Initial Time	Used Percent.	PC	State	Num vars	vars
1	0	0	5	10	16.13 %	4	blocked	1	1
2	0	1	5	11	16.13 %	4	blocked	1	3
3	0	2	5	12	16.13 %	4	blocked	1	5
4	0	3	1	13	3.23 %	16	executing	2	- -
5	0	4	0	14	0.00 %	18	ready	2	1039 502

Figura 4: Impressão processos ativos

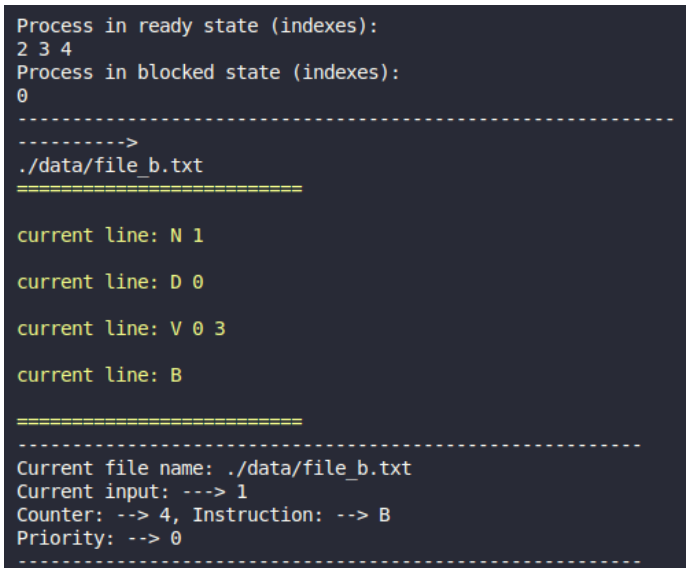


Figura 5: Impressão debug

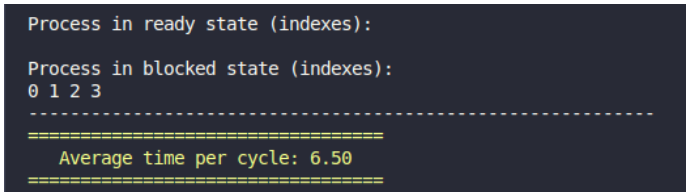


Figura 6: Impressão tempo médio por ciclo

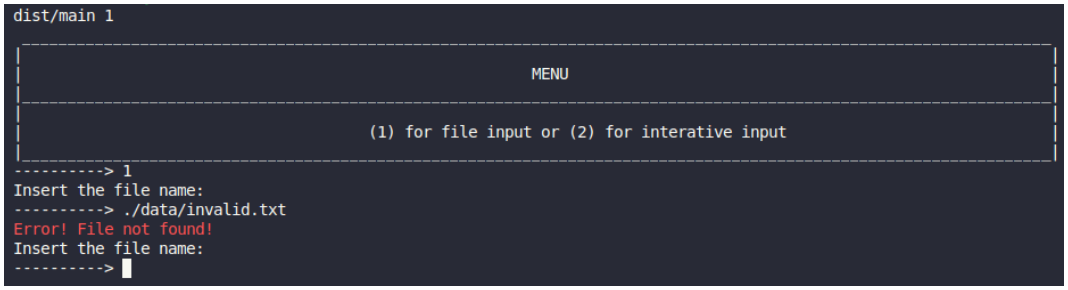


Figura 7: Impressão erro

Por último, temos o *logger* que assim como já elucidado anteriormente, tem como foco a impressão com diferentes cores.

3.7 Testes

A fim de verificarmos a assertividade dos resultados da simulação foram feitos testes com arquivos **.txt**. Para a parte interativa foi executado um comando **U** seguido de um **I** logo após pegarmos a instrução que estava sendo lida naquela linha, dessa maneira, conseguimos perceber se a simulação estava executando corretamente ou se o resultado estava inviesado. Já para a parte em que o usuário fazia a entrada por arquivo, esperávamos a simulação acabar e depois conferíamos o resultado através de testes de mesa. Vale ressaltar que os arquivos utilizados para os testes foram àqueles disponibilizados na documentação do trabalho.

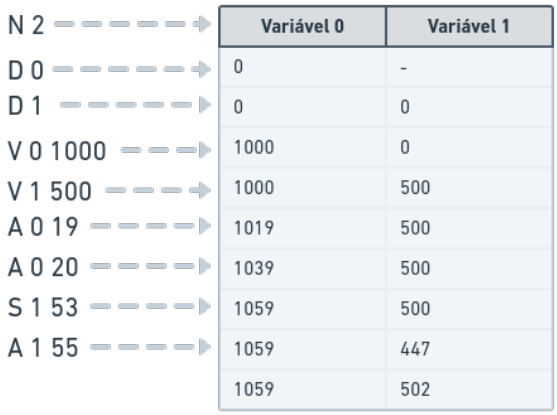


Figura 8: Teste de mesa

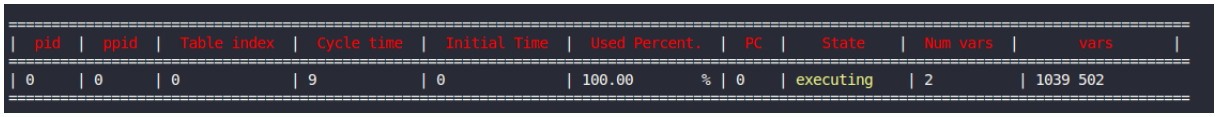


Figura 9: Resultado obtido após o teste de mesa

Posto isto, como pode ser visto nas duas figuras anteriores, o resultado obtido após a execução dos comandos de entrada foi o esperado.

4 Conclusão

Posto isso, conclui-se que o trabalho em questão foi desenvolvido conforme o esperado, atingindo as especificações requeridas na descrição do mesmo em implementar uma simulação de gerenciamento de processos, criar processo, substituir a imagem atual do processo com uma imagem nova do processo, transição de estado do processo, escalonamento de processos e troca de contexto.

Por conseguinte podemos ressaltar que o material apresentado na disciplina foi de suma importância para o desenvolvimento do projeto, haja vista que as explicações dadas pelo professor nos ajudaram bastante a entender as etapas a serem executadas, na construção dos códigos e disponibilização de alguns sites para serem sanadas certas dúvidas acerca da simulação de processos.

Em adição, é válido dizer que apesar das dificuldades na implementação do código, o trio foi capaz de superar e corrigir quaisquer erros no desenvolvimento do algoritmo. Por fim, verificou-se a assertiva para o objetivo do projeto em implementar uma simulação de gerenciamento de processos, criar processo, substituir a imagem atual do processo com uma imagem nova do processo, transição de estado do processo, escalonamento de processos e troca de contexto.

References

[1] Tanenbaum A. *Sistemas Operacionais Modernos*. Pearson Prentice Hall, 2015.

[2] Chamadas de sistema. `https://guialinux.uniriotec.br/chamadas-de-sistema/`. (Accessed on 27/05/2022).