

# Projeto Prático de Sistemas Operacionais

## Minix 3.4.0rc6 - Equipe 5

**Arthur Moreira Craveiro<sup>1</sup>, Bruna Surur Bergara<sup>1</sup>, Emanuella Oliveira S. Balieiro<sup>1</sup>,  
Isabella Mariana C. Pinto<sup>1</sup>, Katherine Mombach V. Mosselaar<sup>1</sup>, Luis Felipe C. Menezes<sup>1</sup>**

<sup>1</sup> Instituto de Ciência e Tecnologia – Universidade Federal de São Paulo (ICT/Unifesp)  
São José dos Campos – SP – Brazil

{arthur.moreira, bruna.surur, emanuella.baliero,  
isabella.cardoso, kmvd.mosselaar, luis.menezes}@unifesp.br

<https://github.com/arthur-craveirol/minix.git>

**Resumo.** *Este relatório apresenta o projeto prático de Sistemas Operacionais, feito no Minix 3.4.0rc6, visando aplicar conceitos como gerenciamento de processos, escalonamento e arquitetura de microkernel. As atividades incluíram instalação, configuração e recompilação do kernel para modificações, além da implementação dos algoritmos de escalonamento Round Robin, Shortest Remaining Time e Loteria. Em seguida, foi feita uma avaliação comparativa do desempenho dos algoritmos implementados com diferentes cargas de trabalho (CPU-bound e IO-bound). O projeto possibilitou a consolidação do conhecimento teórico em práticas reais de Sistemas Operacionais, além do uso de Git para versionamento e organização.*

### 1. Introdução

O projeto prático de Sistemas Operacionais teve como objetivo proporcionar uma vivência prática dos conceitos teóricos abordados em sala de aula, através da interação e modificação do sistema operacional Minix 3.4.0rc6. Durante o desenvolvimento, foram realizadas atividades fundamentais para o entendimento do funcionamento inteiro de um sistema operacional.

O Minix é um sistema operacional baseado em uma arquitetura de microkernel, o que o diferencia de kernels tradicionais por sua arquitetura modular e compacta. Desta forma, a maior parte dos serviços do SO, como gerenciamento de arquivos e de processos, é executada como processos de usuário, e não como parte do próprio kernel, o que entrega ao Minix a possibilidade de que a falha de um serviço de usuário não comprometa o kernel inteiro, além da facilidade de manutenção graças à sua estrutura. O escalonamento de processos no Minix é realizado por meio de diversas filas de prioridade, com 16 níveis distintos, onde cada fila opera em um esquema de round-robin. As partes do escalonamento estão localizadas tanto no espaço de núcleo (/src/minix/kernel/proc.\*) quanto no espaço de usuário (/src/minix/servers/sched/\*), onde um servidor de escalonamento ajusta dinamicamente as prioridades de cada processo.

O projeto pode ser separado em três etapas distintas, abrangendo desde a familiarização inicial com o ambiente do Minix, incluindo sua instalação, configuração e recompilação do kernel, até a implementação de modificações no código-fonte para personalização de mensagens e processos do sistema. A primeira etapa focou na

instalação e configuração do sistema, bem como em modificações de banners e listagem de comandos (resultado da modificação do arquivo `exec.c`, que monitora a execução de comandos no sistema, exibindo seus nomes e caminhos).

Na segunda etapa, o projeto colocou em prática e aprofundou-se na implementação de algoritmos de escalonamento de processos, permitindo a análise comparativa entre o algoritmo padrão do Minix e os algoritmos propostos pelo grupo, – sendo eles o *Round Robin*, o escalonamento da Loteria e o *Shortest Remaining Time*[Tanenbaum and Bos 2016]. Essa abordagem prática possibilitou a consolidação do conhecimento sobre gerenciamento de processos, filas de prioridade e políticas de escalonamento, além de fortalecer o uso de versionamento com Git e organização de ambientes de desenvolvimento virtualizados. Para cada algoritmo implementado, o kernel foi recompilado, e avaliações de desempenho foram sendo conduzidas, incluindo a execução de uma aplicação de teste `teste_processos.c` com diferentes cargas de trabalho, variando o número de processos e a natureza de suas operações (IO-bound e CPU-bound).

Por fim, a terceira etapa, que consistiu na compilação e detalhamento de todo o trabalho realizado nas fases anteriores. Este relatório apresenta as etapas realizadas no projeto, descrevendo desde a ambientação inicial com o Minix até o desenvolvimento dos algoritmos de escalonamento, bem como os resultados obtidos, discussões e considerações finais sobre a experiência prática adquirida.

## **2. Etapa 1 - Instalação e Ambientação**

### **2.1. Instalação e Configuração do Minix[MIX 3 Project 2025]**

A primeira atividade do projeto consistiu na instalação e configuração do sistema operacional Minix 3.4.0rc6 em uma máquina virtual. Para isso, foi utilizado o VirtualBox, uma vez que a compatibilidade com a versão do Minix indicada no projeto já havia sido previamente validada nesse ambiente. A VM foi configurada com 1 GB de memória RAM, disco rígido de 4 GB (dinamicamente alocado), e adaptador de rede em modo bridge, conforme sugerido no enunciado do projeto.

Após o download da imagem `.iso.bz2` do site oficial do Minix, foi realizada a descompactação do arquivo e o associada à unidade de CD da VM. A instalação do Minix foi realizada via terminal, utilizando o comando `"setup"`. Durante o processo, foram criadas as partições no disco, configurado o hostname e ativada a configuração automática de rede, garantindo que a máquina estivesse conectada corretamente à rede local da máquina hospedeira.

Durante o processo de instalação, também foi configurado o teclado, parte do grupo encontrou dificuldades por ter instalado a versão portuguesa do teclado, sem o suporte para as teclas `"/` e `"_"`. Para contornar esse problema, foi necessário recomençar o processo de instalação da máquina virtual e se atentar na escolha de `"abnt2"` para teclado ou então utilizar o teclado virtual fornecido pela VirtualBox. Alguns integrantes conseguiram contornar utilizando os seguintes comandos recomendados pelo site do Minix: `loadkeys /usr/lib/keymaps/abnt2.map`.

Com o sistema instalado, foi removida a imagem `.iso` da unidade da VM, e criado snapshots com o progresso salvo. Posteriormente, foi realizada a criação da senha para o usuário root, seguida da atualização dos pacotes do sistema via `pkgin update`,

instalação de utilitários como o `nano`, e por fim, o comando `pkgin-sets`, que instala pacotes adicionais para desenvolvimento no ambiente Minix. Finalizada a preparação inicial do ambiente Minix, foi utilizado o comando “git fork” para criar uma cópia independente do repositório apresentado pelo professor no perfil de um dos integrantes. Então, na VM foi utilizado os comandos para navegar até a pasta de `/usr/src/` e copiar o diretório em um novo diretório já no ambiente Minix:

```
cd ../usr/src/  
git clone https://github.com/arthur-craveiro1/minix.git
```

Por padrão o comando acima não tem sucesso pois apresenta um erro com certificados SSL. Foram encontradas diversas formas de contornar esse desafio, a adotada pelo grupo foi desativar a macro de verificação de certificados SSL para operações com Git por meio do comando abaixo:

```
export GIT_SSL_NO_VERIFY=1
```

Apesar de não ser ideal por abrir brechas de segurança, essa opção foi priorizada visto que realizar clone por SSH demandaria um esforço adicional de copiar as chaves SSH manualmente do GitHub para a máquina virtual, após tentativas falhas este método foi substituído pelo comando que desativa a verificação de certificados SSL para o Git. Por fim, foi feita a primeira compilação do kernel utilizando o comando `\make build` na pasta do repositório da VM.

## 2.2. Mudança dos banners do kernel

Com o ambiente devidamente preparado, foram realizadas as modificações visuais do sistema. Três banners diferentes foram editados no código-fonte do kernel para exibir informações relacionadas à equipe e ao projeto, um aviso no *boot*, outro após o login e um anterior ao desligamento. Para alteração dos banners no boot e no desligamento, foi necessário alterar o arquivo “*minix/kernel/main.c*” que contém a rotina principal do Minix assim como a do seu desligamento.

A rotina “*announce(void)*” imprime o banner de inicialização do sistema ao fim do *boot* e foi alterada para imprimir o banner de inicialização com a mensagem do grupo, como mostra o código abaixo:

```
printf("=====\n");  
printf("| Minix 3.4.0rc6 - SO - UNIFESP 1s2025 |\n");  
printf("|      Projeto Pratico (I) - Equipe 5      |\n");  
printf("=====\n");
```

A figura 1 explicita a mensagem impressa no funcionamento do sistema.

```

Loading /boot/minix_latest/mod11_mfs.gz
Loading /boot/minix_latest/mod12_init.gz

MINIX 3.4.0. Copyright 2016, Urije Universiteit, Amsterdam, The Netherlands
MINIX is open source software, see http://www.minix3.org
=====
! Minix 3.4.0rc6 - SO - UNIFESP 1s2025 !
!   Projeto Pratico (I) - Equipe 5   !
=====
Started UFS: 9 worker thread(s)
Root device name is /dev/c0d0p0s0
/dev/c0d0p0s0: clean
/dev/c0d0p0s0 is mounted on /
none is mounted on /proc
/dev/c0d0p0s2: clean
/dev/c0d0p0s1: clean
size on /dev/imgrd set to 0kB
Multiuser startup in progress ...
Starting hotplugging infrastructure... done.
Starting services: random lance pty lwip uds ipc log printer vbox.
Starting daemons: update cron.
^[[DStarting network.
IPv6 mode: host
Configuring network interfaces: le0^[[D^[[D^[[D^[[D=

```

**Figura 1. Banner de inicialização alterado**

Para imprimir o banner de desligamento, foi alterada a rotina `prepare_shutdown(int how)`, que prepara o desligamento. Dentro da rotina, antes de chamar a função `set_kernel_timer()`, que configura um temporizador de 1 segundo para o escalonador terminar os processos e depois chama a função de desligamento, foi impressa a mensagem do grupo como mostra abaixo:

```

printf("MINIX will now be shut down ...\n");
printf("=====\n");
printf("| ATE A PROXIMA - SO - UNIFESP 1s2025 |\n");
printf("| Projeto Pratico (I) - Equipe 5       |\n");
printf("=====\n");

```

A figura 2 mostra a modificação em execução.

```

Copyright (c) 1982, 1986, 1989, 1991, 1993
The Regents of the University of California. All rights reserved.

For post-installation usage tips such as installing binary
packages, please see:
http://wiki.minix3.org/UsersGuide/PostInstallation

For more information on how to use MINIX 3, see the wiki:
http://wiki.minix3.org

We'd like your feedback: http://minix3.org/community/

Executando: /bin/sh
Executando: /bin/hostname
Executando: /bin/test
minix# poweroff
Executando: /sbin/poweroff
Executando: /libexec/dhccpd-run-hooks
Executando: /bin/sh
MINIX will now be shut down ...
=====
! ATE A PROXIMA - SO - UNIFESP 1s2025 !
! Projeto Pratico (I) - Equipe 5       !
=====

```

**Figura 2. Banner de desligamento alterado**

Para impressão do *banner* após o login, foi necessário alterar o arquivo de *motd*, conhecido como “*message of the day*”, no caminho do repositório “*etc/motd*”. No entanto,

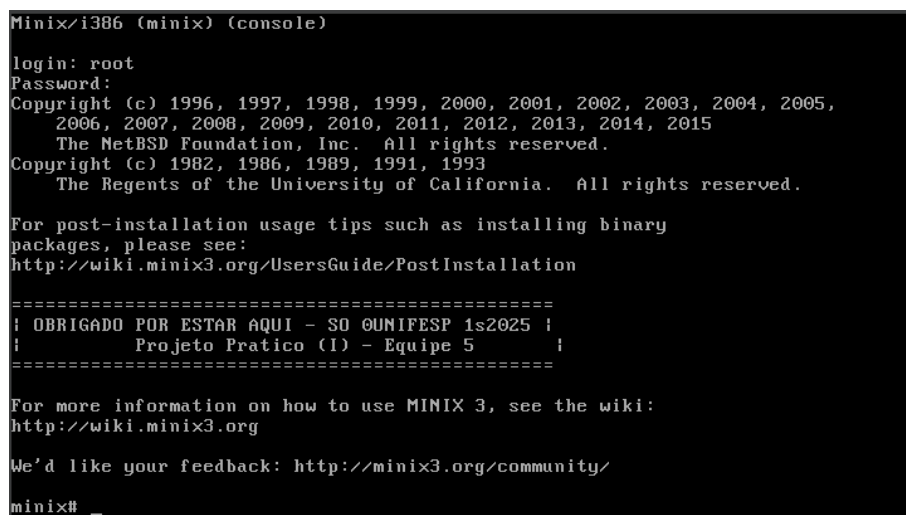
para o usuário *root* o arquivo de *motd* não se altera a cada recompilação ou reboot e é necessário realizar a alteração no arquivo *motd* dentro das pastas exteriores do Minix, fora do repositório. Para isso, partindo do repositório, foi necessário utilizar os seguintes comandos:

```
cd ../../../../etc
nano motd
```

Dentro deste arquivo “*motd*” foi inserido a mensagem do grupo como abaixo:

```
printf(\=====\\n");
printf(\| OBRIGADO POR ESTAR AQUI - SO - UNIFESP 1s2025 |\\n");
printf(\|           Projeto Pratico (I) - Equipe 5           |\\n");
printf(\=====\\n");
```

A figura 3 explicita a mensagem impressa pelo sistema após o login.



```
Minix/i386 (minix) (console)
login: root
Password:
Copyright (c) 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005,
2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015
The NetBSD Foundation, Inc. All rights reserved.
Copyright (c) 1982, 1986, 1989, 1991, 1993
The Regents of the University of California. All rights reserved.

For post-installation usage tips such as installing binary
packages, please see:
http://wiki.minix3.org/UsersGuide/PostInstallation

=====
| OBRIGADO POR ESTAR AQUI - SO @UNIFESP 1s2025 |
|           Projeto Pratico (I) - Equipe 5           |
=====

For more information on how to use MINIX 3, see the wiki:
http://wiki.minix3.org

We'd like your feedback: http://minix3.org/community/

minix# _
```

Figura 3. Banner após o login

### 2.3. Modificação do código `exec.c`

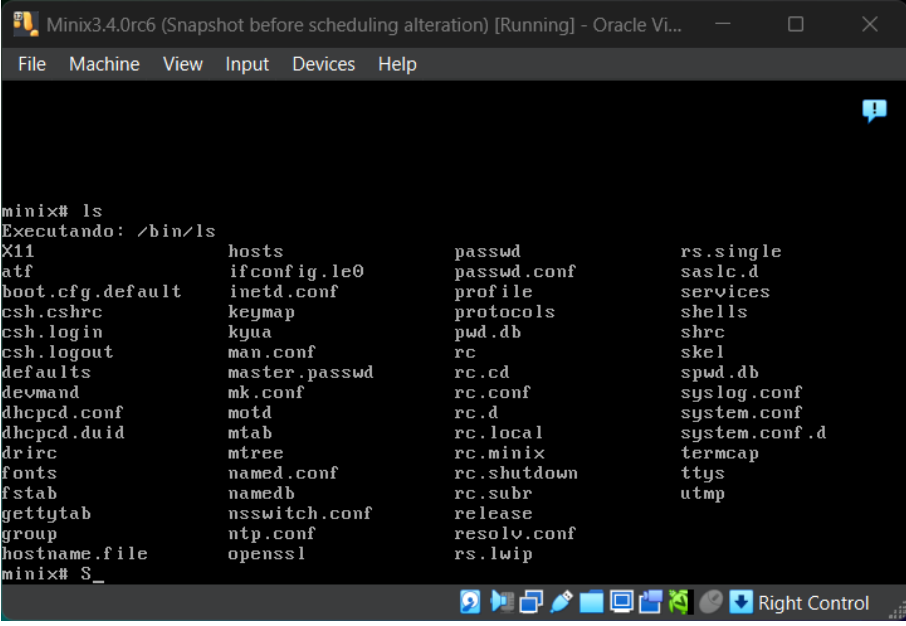
Antes de iniciar a implementação do novo algoritmo de escalonamento, realizamos uma modificação simples no arquivo `/usr/src/minix/servers/pm/exec.c` com o objetivo de verificar se a recompilação do sistema estava sendo feita corretamente e se o procedimento de criação de novos processos estava funcionando como esperado. Na linha 49 da função `do_exec()`, inserimos a linha:

```
printf("Executando: %s\\n", m_in.m_lc_pm_exec.name);
```

Essa linha imprime no terminal o caminho do executável que está sendo carregado, utilizando a variável `m_in.m_lc_pm_exec.name`. Essa variável já estava presente no código e contém o caminho completo do arquivo binário que o processo irá executar.

Essa alteração serviu como teste preliminar de recompilação do kernel, antes da modificação do algoritmo de escalonamento. Com ela, pudemos confirmar visualmente que:

- A recompilação do sistema com `make build` estava funcionando corretamente;



```
Minix3.4.0rc6 (Snapshot before scheduling alteration) [Running] - Oracle Vi...
File Machine View Input Devices Help

minix# ls
Executando: /bin/ls
X11          hosts          passwd         rs.single
atf          ifconfig.le0   passwd.conf    sasl.c.d
boot.cfg.default  inetd.conf     profile        services
csh.cshrc    keymap         protocols      shells
csh.login    kyua           pwd.db         shrc
csh.logout   man.conf       rc             skel
defaults     master.passwd  rc.cd          spwd.db
devmand      mk.conf        rc.conf        syslog.conf
dhcpcd.conf  motd           rc.d           system.conf
dhcpcd.duid  mtab          rc.local       system.conf.d
dirc         mtrees         rc.minix       termcap
fonts        namedb         rc.shutdown    ttys
fstab        nsswitch.conf  rc.subr        utmp
gettytab     ntp.conf       release
group        openssl        resolv.conf
hostname.file openssl         rs.lwip

minix# _
```

Figura 4. Função do `_exec` sendo executada

- O sistema estava reconhecendo e inicializando os processos normalmente;
- A função `do_exec()` estava sendo de fato executada durante a criação de novos processos, como podemos ver na Figura 4.

Essas verificações foram essenciais para garantir que o ambiente de desenvolvimento e testes estava corretamente configurado, fornecendo uma base sólida para as etapas posteriores do projeto.

### 3. Etapa 2 - Algoritmos de Escalonamento de Processos

#### 3.1. Visão Geral

Esta etapa consistiu no desenvolvimento e a análise de desempenho de algoritmos de escalonamento *Minix 3.4.0rc6*, com suporte a sistemas multicore. O trabalho foi dividido em duas frentes: a implementação de diferentes algoritmos de escalonamento no *scheduler* do Minix e a realização de testes com quantidade variada de processos CPU-bound e IO-bound.

#### 3.2. Conceitos de Escalonamento no Minix

O escalonador padrão do Minix é baseado em múltiplas filas de prioridade, numeradas de 0 a 15, onde valores menores indicam maior prioridade. Cada fila sob a política de *round-robin*, em que os processos recebem fatias de tempo iguais e são alternados de forma circular. Os processos de usuário são inseridos entre as prioridades definidas por `USER_Q(7)` e `MIN_USER_Q(14)`, enquanto tarefas de sistema ocupam filas de maior prioridade.

O Minix adota um modelo *microkernel*, no qual parte do gerenciamento de escalonamento é delegada ao servidor de usuário *sched*. Neste servidor, funções como `enqueue()`, `dequeue()` e `pick_proc()` são empregadas para decidir a ordem de

execução dos processos com base nas prioridades dinâmicas e no seu comportamento de execução.

No código-fonte do Minix, os trechos relacionados ao escalonamento estão localizados em:

- **Espaço de Núcleo:** `/src/minix/kernel/proc.*`
- **Espaço de Usuário:** `/src/minix/servers/sched/*`

### 3.3. Desenvolvimento do Algoritmo Lottery Scheduler[HashTag Purple Team 2025]

Implementamos o algoritmo *Lottery Scheduling Static* como alternativa ao modelo padrão baseado em múltiplas filas e prioridades dinâmicas. No modelo de loteria, cada processo recebe um número fixo de *tickets*, e um sorteio aleatório entre todos os processos prontos determina qual será executado, com probabilidade proporcional ao número de *tickets*.

No arquivo `schedproc.h`, adicionamos o campo `unsigned num_tickets`, definindo 20 bilhetes por processo. Também utilizamos o campo `unsigned nice` na função `do_nice()` para ajustar o número de bilhetes.

No arquivo `schedule.c` (em `servers/sched`), a função `loteria()` calcula a soma total de bilhetes dos processos na fila 15 e realiza um sorteio. O processo sorteado é promovido à fila 14. Isso garante que nenhum processo da fila 15 permaneça indefinidamente em espera, e que o promovido perca prioridade ao consumir seu quantum.

As principais funções do arquivo `schedule.c` incluem:

- `do_noquantum()`: redireciona processos que consumiram seu quantum e chama a função `loteria()`;
- `do_start_scheduling()`: define prioridade, quantum e endpoint do processo;
- `do_stop_scheduling()`: encerra processos finalizados ou que consumiram totalmente o quantum;
- `do_nice()`: ajusta o número de tickets, impactando a chance de sorteio.

No arquivo `config.h`, alteramos o valor de `MAX_USER_Q` para 14, mantendo `MIN_USER_Q` como 15. Também alteramos `schedule.c` (em `servers/pm`) para que o campo `nice` seja repassado ao escalonador como base para o número de *tickets*.

Além disso, foram necessárias modificações para adaptar o código ao Minix 3.4.0rc6. Na função `loteria()`, incorporamos o uso de flags com `schedule_process()` para atualizar a prioridade, quantum e CPU explicitamente.

Em `do_noquantum()`, passamos a usar `schedule_process_local()`, alinhando à nova interface para modificações locais. A função `do_start_scheduling()` foi ajustada para lidar com multicore, incluir a função `pick_cpu()` e lidar com a nova estrutura de mensagens.

Ainda, na `do_stop_scheduling()`, foi ajustado o tratamento de CPU específica. Em `do_nice()`, modificamos a chamada ao escalonador com base no endpoint, e por fim, a função `balance_queues()` foi ajustada para utilizar o

serviço nativo do kernel em vez de `set_timer()`, permitindo rebalanceamento a cada 5 segundos de forma compatível com a estrutura atual.

### 3.4. Desenvolvimento do Algoritmo Shortest Remaining Time (SRT)

Para o projeto de escalonamento, também foi implementado o algoritmo **Shortest Remaining Time (SRT)**, que tem como principal objetivo priorizar o processo com menor tempo restante de execução, certificando que exista um sistema mais reativo para tarefas curtas. Para isso, foram necessárias modificações nos arquivos `proc.c` e `proc.h`, ambos localizados no diretório `/src/minix/kernel` do Minix. No arquivo `proc.h`, foi adicionada uma nova variável inteira na struct `proc`:

```
int remaining_time; /* tempo restante estimado para SRT */
```

Essa variável armazena o tempo restante de cada processo, usado como critério de ordenação.

Já no arquivo `proc.c`, a função `enqueue()` foi modificada quase por completo para suportar o algoritmo SRT. Originalmente, essa função apenas inseria novos processos no final da fila de prioridade correspondente, seguindo a política padrão do Minix. Com a implementação do SRT, a função começou a inserir cada processo em ordem na fila, considerando seu `remaining_time`.

A nova lógica atualizada percorre a fila de processos prontos até encontrar a posição correta para ser inserida, garantindo que processos com menor tempo restante sejam executados antes. Essa verificação é feita através do trecho:

```
while (curr && curr->remaining_time <= rp->remaining_time) {
    prev = curr;
    curr = curr -> p_nextready;
}
```

Após essa verificação, o processo é inserido antes do processo com tempo maior:

```
if (prev == NULL) {
    rp->p_nextready = rdy_head[q];
    rdy_head[q] = rp;
} else {
    rp->p_nextready = curr;
    prev->p_nextready = rp;
}
```

Caso o novo processo possua tempo restante menor do que o processo que está atualmente em execução, uma preempção é forçada através da chamada `RTS_SET(p, RTS_PREEMPTED)`, substituindo imediatamente o processo em execução pelo novo. Por fim, o *tail* da fila é atualizado caso o processo tenha sido inserido no final:

```
if (curr == NULL) rdy_tail[q] = rp;
```

### 3.5. Desenvolvimento do Algoritmo Round-Robin

O último algoritmo de escalonamento desenvolvido foi o *Round Robin*. Neste algoritmo todos os processos possuem a mesma quantidade de tempo de CPU, chamado de quantum. Os processos são ordenados em uma fila por ordem de chegada. Quando um processo excede seu quantum ou é bloqueado ele vai para o final da fila, trazendo a ideia da circularidade dos processos.



Para esta implementação, usamos como base o algoritmo padrão do Minix, que já utiliza o *Round Robin*, porém empregando-o em diferentes filas de prioridades. Dessa forma, a ideia geral era apenas deixar todos os processos com o mesmo nível de prioridade e eliminar todos os métodos nos quais um processo aumenta ou diminui de prioridade. O arquivo principal que cuida do escalonamento no Minix é o *schedule.c*. Nele foram modificadas duas funções: *do\_noquantum* e *do\_start\_scheduling*.

A primeira trata do momento no qual um processo excede seu tempo de CPU e precisa ser bloqueado. Quando um processo é bloqueado, seu nível de prioridade aumenta no algoritmo padrão, portanto as linhas apresentadas no código abaixo foram retiradas:

```
if (rmp->priority < MIN_USER_Q) {  
    rmp->priority += 1;  
}
```

Para a segunda função, temos toda a inicialização de um processo na fila de escalonamento. Para garantir que todos esses processos tenham o mesmo nível de prioridade, foram alteradas as seguintes variáveis para a estrutura *rpm*:

```
rmp->max_priority = USER_Q;  
rmp->priority      = USER_Q;  
rmp->time_slice    = DEFAULT_USER_TIME_SLICE;
```

Sendo *USER\_Q* a prioridade base dada para todos os processos e *DEFAULT\_USER\_TIME\_SLICE* o quantum de cada processo. Além dessas modificações, outras partes dessa mesma função foram retiradas pela redundância na definição das mesmas variáveis já definidas anteriormente, como em:

```
case SCHEDULING_START:  
    rmp->priority      = rmp->max_priority;  
    rmp->time_slice    = m_ptr->m_lsys_sched_scheduling_start.quantum;  
    Break;
```

e também em:

```
rmp->priority = schedproc[parent_nr_n].priority;  
rmp->time_slice = schedproc[parent_nr_n].time_slice;
```

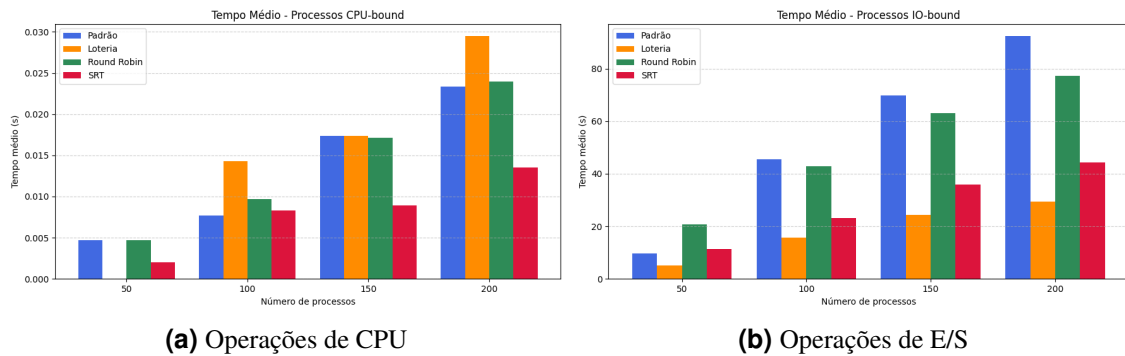
### 3.6. Testes dos algoritmos

Para testar os algoritmos, foi utilizado o programa *teste\_processos.c*, feito pelo professor. Foram realizados dois testes distintos, alterando a quantidade de operações de CPU e de Entrada/Saída. Para cada um destes dois testes foram modificados os números de processos de 50, 100, 150 e 200 a fim de comparar a escalabilidade dos algoritmos. Para o primeiro teste foram feitas 2.000 operações de E/S e 2.000.000 de CPU, enquanto que no segundo foram realizadas 20.000 de E/s e 20.000.000 de CPU.

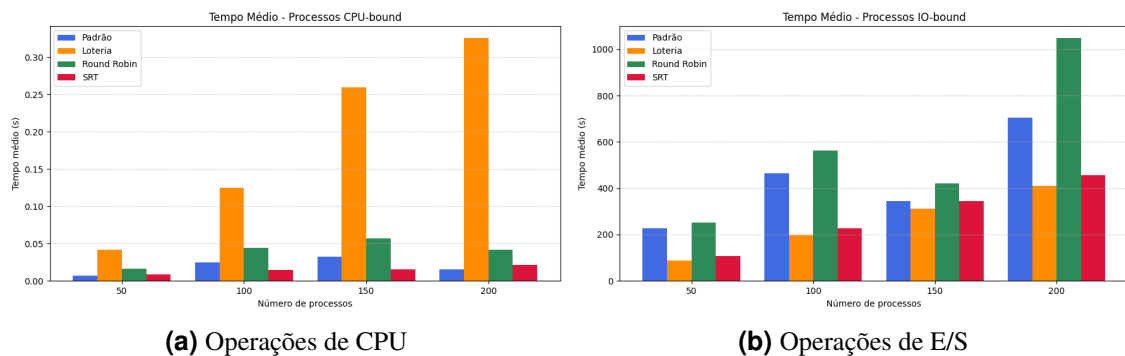
### 3.7. Resultados obtidos

Após a etapa de testes, os arquivos de texto gerados por cada um dos algoritmos foram analisados e foram construídos quatro gráficos no total.

O algoritmo *Shortest Remaining Time* (SRT) tem como principal objetivo a priorização dos processos com menor tempo restante de execução. Isso o torna especialmente eficaz em contextos onde existe uma alta concorrência por CPU, como observado



**Figura 5. Gráficos do primeiro teste.**



**Figura 6. Gráficos do segundo teste.**

nos testes com processos *CPU-bound*, nos quais o SRT apresentou os menores tempos médios de retorno entre todos os algoritmos comparados. Essa vantagem se dá pela sua política preemptiva, que reordena a fila de prontos sempre que um processo mais “curto” chega, forçando a preempção do atual se necessário.

Nos testes com processos *I/O-bound*, o SRT também teve bom desempenho, superando o escalonador padrão e o Round Robin. Embora a diferença não tenha sido tão grande quanto nos testes *CPU-bound*, o algoritmo manteve tempos médios consistentes, aproveitando sua capacidade de priorizar rapidamente processos curtos.

De forma geral, os resultados confirmam que o SRT é eficiente para reduzir o tempo médio de retorno e responder rapidamente a tarefas curtas. No entanto, por ser altamente preemptivo, pode ser inadequado em sistemas com alto custo de troca de contexto ou que exigem maior equidade entre os processos.

Com base na estrutura do algoritmo de Loteria e em estudos sobre sua implementação, espera-se maior equidade no acesso à CPU entre processos *CPU-bound* e *I/O-bound*. Cada processo tem uma chance proporcional de ser escolhido, evitando inanição. Como a escolha depende de sorteio aleatório, a ordem de execução pode variar a cada rodada, dificultando a reprodutibilidade dos testes. Em sistemas críticos ou de tempo real, isso se torna um problema, já que mesmo processos com poucos bilhetes podem ser sorteados, tornando o comportamento imprevisível.

O algoritmo Round Robin é simples e promove uso justo da CPU, garantindo acesso periódico a todos os processos. É indicado para sistemas interativos que exigem responsividade. Nos testes, teve desempenho semelhante ao escalonador padrão do Minix. No entanto, sob alta carga, foi superado por algoritmos mais sofisticados, como prioridades e SRT, por não diferenciar processos curtos ou urgentes.

### 3.8. Discussão

Com a escalabilidade dos processos em cada teste, foi possível observar o crescimento linear no tempo de execução e foram obtidos resultados proveitosos. Na elaboração dos testes, o grupo primeiramente idealizou o número de processos variando de 10 até 150, porém foi perceptível a dificuldade na análise dos algoritmos, visto que o tempo de execução era extremamente baixo e as discrepâncias entre cada algoritmo não eram mostradas. Além disso, inicialmente o número de processos seria a metade do que foi realmente utilizado, mas foi necessário dobrar o valor pelos mesmos motivos da quantidade de processos.

Primeiramente, analisando o algoritmo de *Shortes Remaining Time*, é perceptível que ele apresentou os melhores resultados. Ele obteve o menor tempo para os dois tipos de processo em todos os casos no primeiro teste, e manteve sua estabilidade no segundo, mesmo com o aumento expressivo de operações. Pode-se afirmar que, para tarefas de processos em grandes lotes e com pouca imprevisibilidade durante a execução, seu desempenho é o melhor do que os outros métodos analisados.

O algoritmo que obteve resultados mais diferentes dos demais foi o da Loteria, algo já esperado dada sua aleatoriedade intrínseca de implementação. Para o primeiro teste, pode-se observar que a Loteria conseguiu distribuir de forma proveitosa os bilhetes e garantiu que os processos de E/S fossem os mais rápidos entre os algoritmos. No segundo teste, a aleatoriedade prejudicou muito os processos CPU *bound*. De forma geral, é possível afirmar que este foi o algoritmo que melhor lidou com os processos de E/S.

Para o algoritmo de *Round Robin*, a ideia era perceber qual seria o impacto na execução de um algoritmo que aplica o RR com prioridade, o Minix padrão, e sem prioridade, RR puro. Como visto, para o ambiente de testes criado, a diferença não foi tão grande, mesmo com o aumento na quantidade de operações, o que mostra que a complexidade introduzida pelo algoritmo de prioridades não é tão necessária para aplicações simples. No entanto, é necessário apontar que, para o aumento muito grande de operações e um número de processos alto, como no caso do segundo teste, seu tempo médio foi muito superior aos outros algoritmos, até mesmo do padrão. Algo interessante a ser observado em estudos futuros seria na diferença de *quantum* para diferentes processos, que para este projeto não foi implementado, mas que poderia se diferenciar mais do algoritmo de prioridades padrão do Minix.

## 4. Conclusão

Com este trabalho, foi possível aplicar, na prática, a teoria estudada em sala de aula sobre Sistemas Operacionais. Através do uso da máquina virtual com o Minix, tivemos contato direto com a estrutura do sistema, manipulando arquivos essenciais para o funcionamento dos *banners*, a impressão do arquivo que está sendo executado pelo SO, entre outras tarefas fundamentais (como a recompilação do kernel e a criação de *snapshots*). Essas

atividades proporcionaram uma compreensão mais concreta dos mecanismos internos de um sistema operacional, especialmente no que se refere ao funcionamento do escalonador de processos.

A familiarização com o ambiente e com os arquivos críticos do sistema foi essencial para que pudéssemos implementar com sucesso três alterações nos algoritmos de escalonamento, sendo necessário modificar arquivos específicos da implementação do Minix. As alterações foram realizadas para os algoritmos *Shortest Remaining Time*, *Loteria* e *Round Robin* sem Prioridade, permitindo a realização de testes comparativos que revelaram o comportamento de cada estratégia frente a diferentes cargas de trabalho.

A etapa de testes demonstrou, de forma prática, como cada algoritmo se comporta em cenários compostos por processos *CPU-bound* e *IO-bound*. O *Shortest Remaining Time* (SRT) mostrou-se o mais eficiente em termos de tempo médio de retorno, mantendo desempenho estável mesmo com o aumento da carga de processos, o que confirma sua adequação para cenários com grande volume de tarefas previsíveis. Já o algoritmo de *Loteria* apresentou maior variabilidade nos resultados devido à sua natureza aleatória, sendo mais eficaz com processos de *I/O*, mas pouco confiável em ambientes críticos ou de tempo real, onde a previsibilidade é essencial. O *Round Robin* sem Prioridade, por sua vez, destacou-se por sua simplicidade e justiça, mantendo desempenho próximo ao escalonador padrão do Minix, embora tenha sido superado pelos demais em situações com alta demanda e necessidade de diferenciação entre processos.

Portanto, de modo geral, o projeto proporcionou uma experiência prática e esclarecedora sobre os impactos reais de diferentes políticas de escalonamento, além de reforçar a importância do entendimento teórico para o desenvolvimento e avaliação de soluções eficientes dentro de um Sistema Operacional.

## 5. Divisão do trabalho em equipe

O grupo conta com seis integrantes. Todos os membros fizeram as alterações dos banners em seus próprios computadores para se familiarizarem com a máquina virtual. Três duplas foram formadas para a implementação e geração dos resultados dos algoritmos de escalonamento. As duplas foram: Bruna e Emanuella, com o SRT; Arthur e Luis, com o RR; Katherine e Isabella, com a Loteria. Cada dupla criou sua própria *branch* no *Git* para preservar o versionamento do projeto. Além disso, a geração dos resultados foi feita em um computador por dupla. A redação do relatório foi feita por todos os integrantes.

## 6. References

### Referências

- HashTag Purple Team (2025). Minix 3.4.0rc6 - Lottery Scheduler. <https://github.com/HashTag-PurpleTeam/minix-lottery-scheduler/tree/master>. GitHub Repository. Acesso em: jul. 2025.
- MINIX 3 Project (2025). MINIX 3 Official Website. <https://www.minix3.org/>. Acesso em: jul. 2025.
- Tanenbaum, A. S. and Bos, H. (2016). *Sistemas Operacionais: Projeto e Implementação*. Pearson, São Paulo, 4 edition.