Relatório do Projeto 3
MC504 - Grupo 12

Matheus Domingues Casanova Nogueira (185135)
André Rodrigues Alves da Silva (231392)
Daniel de Sousa Cipriano (233228)
Guilherme Gomes Gonçalves (170927)

***Resumo*—Neste relatório de projeto, apresentamos mudanças que realizamos no sistema operacional Minix3 para tentarmos adicionar novos comportamentos e políticas de escalonamento para o sistema. Para processos-pai, que não vêm de um `fork()`, tentamos implementar um escalonador Round Robin (RR), e para processos-filho, que vêm de um `fork()`, tentamos implementar um escalonador First Come First Served (FCFS).**

Primeiramente, na introdução, apresentamos uma pequena visão geral sobre escalonadores, com seus objetivos e funcionamentos básicos. Após isso, explicamos como o escalonamento é feito no Minix3, incluindo as escolhas de *design* adotadas, as chamadas de sistema utilizadas e os servidores que são responsáveis pelo escalonamento.

Após isso, explicamos como pensamos em implementar a nova política de escalonamento descrita acima utilizando um gerenciador de escalonadores. Explicamos nossa abordagem e decisões que tomamos para implementar esse gerenciador.

Após explicarmos o gerenciador, apresentamos como foi feita a criação dos novos escalonadores RR e FCFS que implementamos, e quais foram as especificidades de cada um. Abordamos também algumas dificuldades que tivemos para implementarmos esses escalonadores e as soluções que conseguimos.

Por fim, explicamos como pensamos em formas de testar nossas alterações, para tentarmos verificar que elas funcionariam do jeito que esperávamos. Após isso, compilamos todas as alterações que realizamos no sistema e apresentamos os resultados que obtemos e uma conclusão acerca deles.

I. Introdução

QUANDO usamos um sistema operacional (S.O.) multiprogramado, frequentemente temos mais de uma aplicação (processo) no estado pronto (*ready*) ao mesmo tempo. Se o processador (CPU) não possui *cores* suficientes para executar esses processos em paralelo, ele deverá escolher qual processo será executado primeiro e quais processos serão executados após novos *cores* ficarem disponíveis. Essa decisão de escolher qual processo será executado é feita por um processo chamado de escalonador (*scheduler*). A função básica de qualquer escalonador é permitir que processos sejam executados de maneira "justa" (isso é, que todos possam ser executados e possam realizar suas funções em tempos razoáveis) e eficiente (que as escolhas do escalonador façam com que os recursos do computador sejam utilizados de maneira ótima). No geral, processos têm "ciclos de vida" definidos por operações de entrada e saída (I/O) e por operações de CPU. Alguns processos utilizam mais operações de entrada e saída, e são chamados de processos limitados por entrada e saída (I/O *bound*), e outros processos utilizam mais operações de CPU, e são chamados de processos limitados por CPU (CPU *bound*). De maneira geral, é interessante que os processos I/O *bound* sejam executados antes dos processos CPU *bound*, pois em operações de I/O os processos são bloqueados, dando chance para que outros processos sejam executados enquanto o disco trabalha com as requisições feitas. Além disso, é interessante que processos que raramente utilizam chamadas de I/O (CPU *bound*) sejam limitados pela CPU. Como eles não serão bloqueados frequentemente, eles podem consumir muito tempo da CPU, fazendo com que os outros processos esperem muito tempo para executar. Os escalonadores podem definir um tempo limite para que esses processos sejam executados, e esse tempo é chamado de *quantum*. No geral, essas priorizações e limitações são os objetivos mais desejados e realizados por escalonadores, mas existem várias outras funções e políticas de escalonamento que também podem ser utilizadas por um escalonador, cada uma com seus pontos positivos e negativos.

No S.O. Minix3, o escalonamento é feito tanto pelo núcleo (*kernel*) quanto por um serviço executado em modo de usuário chamado de *sched*. Em versões antigas do Minix, o escalonamento era feito apenas pelo *kernel*, mas, por ser um S.O. microkernel (com núcleo mínimo), foi feita uma atualização para tentar mover o escalonador para o modo de usuário, usando um novo serviço [2]. No geral, um *kernel* de um S.O. que faz apenas o que é necessário (estritamente o mínimo possível) garante maior confiabilidade no sistema, pois *bugs* a nível do núcleo são perigosos para o sistema como um todo. Como o Minix3 tem como objetivo garantir esse aspecto para seus usuários [2], grande parte do escalonamento, atualmente, é feita pelo serviço *sched*.

Alguns modelos de escalonadores são baseados em eventos (*event-driven*), e outros são pró-ativos (*pro-active scheduling*). No Minix3, o serviço do escalonador é baseado no primeiro modelo, e ele recebe mensagens de serviços como o gerenciador de processos (PM) e também mensagens do *kernel*, e age de acordo com as informações delas. Dentre as mensagens bloqueadas pelo escalonador, podemos destacar as mensagens de controle (processo foi encerrado/criado/bloqueado), e mensagens de tempo esgotado (*out of quantum*). A resposta do escalonador varia de mensagem para mensagem. A definição do serviço pode ser encontrada no diretório `usr/src/minix/servers/sched`, junto com as ações específicas dele para cada tipo de mensagem.

Como o serviço do escalonador é um serviço em modo usuário, alguns outros processos podem ser inicializados antes dele, e podem exigir um escalonamento antes do próprio escalonador ser inicializado, o que pode gerar até possíveis *deadlocks*. Tendo isso em vista, o *kernel* do Minix3 possui

também um simples escalonador para evitar que isso aconteça. O escalonador do *kernel* é um escalonador simples utilizando filas de prioridades, e mais informações acerca dele podem ser encontradas em [1]. A seguir, explicaremos resumidamente como podemos implementar novos escalonadores utilizando algumas chamadas de sistema (*system calls*) definidas pelo *kernel*, e entraremos também em mais detalhes sobre a implementação do serviço *sched*.

O *kernel* do Minix3 e o gerenciador de processos (PM) possuem acesso à várias informações relacionadas aos processos do sistema. No *kernel*, na tabela de processos, cada processo possui informações relacionadas ao *scheduler* responsável pelo escalonamento daquele processo, no campo `p_scheduler`. Cada processo também possui, nessa tabela do *kernel*, um campo chamado `p_quantum_size_ms`, que diz o tamanho do *quantum* do processo. Caso o campo `p_scheduler` de um processo tenha o valor `NULL`, o *kernel* será responsável pelo escalonamento desse processo. Do contrário, o escalonador definido nesse campo será considerado pelo *kernel*, e esse o enviará mensagens relacionadas ao escalonamento, como mensagens de tempo esgotado (*out of quantum messages*). O gerenciador de processos (PM), também possui informações, a nível de usuário, relacionadas ao escalonamento de processos. Através do campo `mp_scheduler`, é possível obter o escalonador utilizado para um processo dado, e com o campo `mp_nice` é possível obter um valor que representa a ”prioridade” do processo.

Sabendo dessas informações, é possível que, no Minix3, criemos serviços além do serviço padrão *sched* que sejam responsáveis pelo escalonamento de um conjunto de processos. Tudo que precisamos fazer é definir os campos citados acima para cada um desses processos, e assim fazemos com que o *kernel* reconheça um novo escalonador. Para realizarmos isso, podemos utilizar algumas *system calls* definidas pelo próprio S.O. A chamada `sys_schedctl` pode ser utilizada por um serviço para tomar conta do escalonamento de um processo. O *kernel* reconhecerá a chamada e ”anotará” que o novo escalonador do processo é o escalonador que realizou a chamada. A *syscall* `sys_schedule` pode ser usada para um escalonador definir novas prioridades para um processo, e também definir novos valores de *quantum* associados a ele.

Para começar o escalonamento de um processo, os serviços do gerenciador de processos (PM) e do servidor de reencarnação (RS), por exemplo, mandam mensagens para o escalonador padrão pedindo para que ele comece a escalonar novos processos originais ou advindos de um `fork()`. Se, ao invés de tratarmos essas mensagens no próprio escalonador padrão, enviarmos elas para outros escalonadores (por meio de comunicação entre processos, por exemplo), outros escalonadores podem ter a oportunidade de usar as chamadas de sistema explicadas acima para tomarem conta do escalonamento desses processos, e podemos assim definir novos domínios de escalonamento, transformando o escalonador padrão em um gerenciador de escalonadores. Tudo isso pode ser feito com alterações mínimas no serviço existente *sched*, apenas precisamos nos preocupar com a implementação dos novos escalonadores e com as políticas de escalonamento deles. Nas próximas seções, discutiremos com mais detalhes como tentamos implementar essa ideia para criarmos dois novos escalonadores no Minix3.

II. Criação do Gerenciador de Escalonadores

O escalonador padrão a nível usuário do Minix3, *sched*, funciona recebendo e bloqueando mensagens relacionadas ao escalonamento que são recebidas de servidores como o PM e do próprio kernel. Dentre todas as mensagens tratadas pelo escalonador padrão, as mensagens `SCHEDULING_INHERIT` e `SCHEDULING_START` são as únicas mensagens relacionadas à inicialização do escalonamento de processos. O servidor PM envia essas mensagens quando algum processo precisa ser escalonado pelo escalonador. A mensagem `SCHEDULING_INHERIT` está relacionada à processos advindos de um `fork()`, e a mensagem `SCHEDULING_START` à processos-pai originais.

O escalonador padrão implementado trata essas duas mensagens com uma chamada a função `do_start_scheduling()`, que registra algumas informações sobre o processo em uma estrutura local do escalonador e começa o escalonamento usando a *syscall* discutida na introdução `sys_schedctl()`.

Se modificarmos o tratamento dessas duas mensagens do *sched*, e fizermos ele repassar essas mensagens para outros serviços tratarem, transformamos ele no gerenciador de escalonadores discutido anteriormente. Sendo assim, modificamos o tratamento da mensagem `SCHEDULING_INHERIT` e `SCHEDULING_START`. Agora, o escalonador padrão *sched* envia uma mensagem para outros dois escalonadores tratarem essas mensagens. A primeira mensagem, relacionada à herança de escalonadores (`fork()`), será tratada e escalonada por um escalonador FCFS (First Come First Served), e a segunda, relacionada à processos-pai normais, será escalonada por um escalonador RR (Round Robin).

Esses dois escalonadores também foram criados nesse projeto, e são servidores do Minix3. Eles podem ser encontrados no diretório `/usr/src/minix/servers`, e mais detalhes sobre a implementação deles serão discutidos nas próximas seções. Para podermos nos comunicar facilmente com esses servidores, registramos um *endpoint* fixo para eles nos registros de *endpoints* do sistema. Esse registro pode ser encontrado no arquivo `/usr/src/minix/include/minix/com.h`. Chamamos os endpoints dos escalonadores FCFS e RR de, respectivamente, `SCHED_FCFS_PROC_NR` e `SCHED_RR_PROC_NR`.

Com esses *endpoints* implementados, conseguimos finalizar a construção do gerenciador de escalonadores no *sched*. Utilizamos a *syscall* `_taskcall()` para repassarmos as mensagens relacionadas à cada um dos escalonadores para eles poderem tratá-las. O novo tratamento da mensagem `SCHEDULING_INHERIT` utilizando essas ideias está representado na figura 1.

Além de modificarmos o tratamento dessas duas mensagems, modificamos uma chamada que o *sched* faz durante a inicialização do sistema, que define um *timer* relacionado ao balanceamento das prioridades dos processos que estão sendo escalonados por ele (política do escalonador implementada). Nossos escalonadores implementados não usarão a mesma política, e esse balanceamento periódico é prejudicial para eles, pois, como explicaremos nas próximas seções, usaremos uma fila (prioridade) fixa para todos os processos escalonados. Dessa forma, removemos os códigos relacionados a essa chamada do servidor padrão. Também removemos o

tratamento de todas as outras mensagens relacionadas ao escalonamento, já que o escalonador padrão não vai escalonar nenhum processo com nossas modificações e será um mero gerenciador de escalonadores. Assim, ele nunca receberá outra mensagem além das duas que já tratamos. Essas foram todas as modificações realizadas no serviço *sched*, e podem ser vistas com mais detalhes nos *commits* do GitLab no diretório `/usr/src/minix/servers/sched`.

III. Criação dos Escalonadores Round Robin e FCFS

Para criarmos os novos escalonadores do sistema, criamos dois novos serviços relacionados a eles. Decidimos criar dois serviços pois o escalonador *sched* é implementado como um serviço, e mantemos o padrão do S.O. Como dito anteriormente, registramos também um *endpoint* fixo para esses novos serviços. Os serviços foram criados segundo a documentação do Minix3 [3], sem modificações.

Os escalonadores funcionam que nem o escalonador *sched*, reagem a eventos que ocorrem no sistema relacionados a eles. Para implementarmos o código dos escalonadores, reutilizamos grande parte do código do escalonador normal, e realizamos pequenas alterações. O tratamento das mensagens de inicialização de escalonamento agora não é mais delegado, e é tratado por funções locais que criamos chamadas de `do_start_RR_scheduling()` e `do_start_FCFS_scheduling()`. Além disso, como dito na seção anterior, também não inicializamos e tratamos a mensagem `CLOCK` do *timer* do escalonador normal, que é utilizada para balanceamento de prioridades, pois não precisamos desse balanceamento. Por fim, as outras mensagens que tratamos, `SCHEDULING_STOP`, `SCHEDULING_SET_NICE` e `SCHEDULING_NO_QUANTUM`, são resolvidas, respectivamente, por funções locais que criamos, `do_stop_XX_scheduling()`, `do_nice_XX()` e `do_noquantum_XX()`, com `XX` sendo `FCFS` ou `RR`, dependendo do escalonador.

Como o serviço *sched* precisa mandar mensagens para esses escalonadores, e como esses escalonadores não são inicializados automaticamente com o sistema (apenas o escalonador padrão *sched*), também tentamos fazer com que esses escalonadores se inicializassem com o sistema. Analisando a bibliografia [1], decidimos que deveríamos adicionar novas entradas no arquivo `/usr/src/etc/system.conf` com os serviços que criamos, para que eles pudessem ser inicializados junto com o sistema e pudessem estar disponíveis assim que *sched* precisasse deles. As entradas registradas podem ser vistas na figura 2.

IV. Especificidades do Escalonador Round Robin

A política Round Robin, em sua forma mais simplificada, define um valor de *quantum* e organiza todos os processos que são escalonados em uma fila normal. Primeiramente, o processo no começo da fila é escalonado e obtém acesso à CPU. Quando ele é bloqueado, ou quando seu *quantum* acaba, ele vai para o final da fila, e o próximo na fila é executado. Isso se repete para todos os processos, até que a fila fique vazia.

Para implementarmos essa política, utilizamos as próprias filas que o Minix3 implementa para o escalonamento, e implementamos tratamentos alternativos para algumas mensagens para que elas implementassem a política de escalonamento Round Robin. O tratamento da mensagem `SCHEDULING_SET_NICE` agora não altera mais a prioridade de um processo, e retorna apenas `OK`. A mensagem `SCHEDULING_NO_QUANTUM` também não diminui mais a prioridade de processos que usaram todo seu *quantum* e não finalizaram, ela simplesmente coloca o processo no fim de sua fila novamente. O tratamento da mensagem `SCHEDULING_START` define que a maior prioridade do processo é fixa com valor 7 (`rmp->max_priority = 7;`), que sua prioridade atual também é fixa e é 7 (`rmp->priority = 7;`), e que seu *quantum* tem 40ms (`rmp->time_slice = 40;`). As *syscalls* discutidas na introdução são utilizadas e registram essas prioridades, escalonando assim os processos. Finalmente, podemos receber também uma mensagem `SCHEDULING_INHERIT` nesse escalonador. Quando isso acontece, enviamos uma mensagem para o escalonador FCFS, pois ele deve tratar todos os processos vindos de um `fork()`. Para mandarmos essas mensagens usamos o mesmo método adotado para o escalonador *sched*.

V. Especificidades do Escalonador *FCFS*

A política FCFS (First Come First Served) organiza todos os processos numa fila, mas não define nenhum valor de *quantum* para eles. Primeiramente, o processo no início da fila é escalonado e começa a executar. Como não existe um valor de *quantum*, ele só deixará de executar quando for bloqueado ou quando finalizar todo o seu trabalho. Isso acontecerá até que não haja mais nenhum processo para ser executado.

Para implementarmos essa política, também usamos as filas que o *kernel* do Minix3 implementa para realizarmos o escalonamento. Os tratamentos das mensagens `SCHEDULING_SET_NICE` e `SCHEDULING_NO_QUANTUM` ficaram iguais aos tratamentos do escalonador Round Robin, as prioridades não são alteradas com essas mensagens. Como os processos virão de um `fork()`, teremos, para iniciarmos o escalonamento, uma mensagem `SCHEDULING_INHERIT`. Para tratarmos essa mensagem, aplicando a política FCFS, queremos dar um valor de *quantum* infinito para cada processo, para que eles se executem até finalizarem ou até serem bloqueados, e também queremos dar uma prioridade fixa e igual para todos eles. Como não temos a noção do valor de *quantum* infinito, demos o valor de `INT_MAX` para os processos, da mesma forma que foi discutida para o escalonador Round Robin. Mesmo não sendo infinito, esse valor é muito grande, representando mais de 3 semanas. Os processos terão tempo suficiente para se executarem. Para as prioridades fixas, atribuímos aos processos escalonados por esse serviço a prioridade 8, também usando o mesmo método que usamos para o escalonador Round Robin. As mesmas *syscalls* da introdução são chamadas e os processos são escalonados pelo escalonador FCFS.

VI. Testes das implementações

Criamos os seguintes programas auxiliares para testar os escalonadores, `track_process_test.c`, `child_test.c`, `normal_test.c` e `simulate_work.c`. Precisavamos verificar se, após implementados os novos escalonadores, os

processos estavam sendo iniciados e terminados corretamente, portanto, fizemos o arquivo `track_process_test.c` que simula uma execução e escreve em um arquivo .txt os status de cada processo ao iniciar e terminar. Além disso, queríamos testar a performance dos nossos escalonadores, para isso fizemos o código do arquivo `simulate_work.c`, que executa uma simples manipulação de bits para simular um processo que tem longo tempo de execução, verificar como a CPU e os escalonadores lidam com essa carga de trabalho, calcular o tempo total de execução do processo e quanto tempo o processo ficou em `waiting`. Por fim, criamos os arquivos `child_test.c` e `normal_test.c` que executam `simulate_work.c` para processos filhos e pais, respectivamente.

Depois de criarmos os programas, primeiramente executamos todos eles no escalonador padrão *sched* para analisarmos os resultados. O resultado do programa `track_process_test` pode ser visto na figura 3. O resultado dos programas `child_test.c` e `normal_test.c` podem ser vistos nas figuras 4, 5, respectivamente. Com esses resultados, podemos identificar em alguns processos uma diferença nos tempos de execução e `waiting` para os processos filhos e pais, e verificamos que o escalonador normal realmente realiza um tratamento diferenciado (como pode ser visto na função original `do_start_scheduling()` para diferentes tipos de processos (originais ou filhos). Verificamos assim que nossos testes de escalonadores estão funcionais e seguindo o que lhes foi definido.

Por fim, para testarmos o funcionamento de nossos escalonadores, compilamos seus códigos e as mudanças que realizamos no sistema. Primeiramente, definimos novos *endpoints* para os escalonadores no próprio sistema (além do repositório que clonamos `/usr/src`), para registrarmos nossas alterações. Copiamos o mesmo arquivo alterado, `/usr/src/minix/include/minix/com.h` para o arquivo `/usr/include/minix/com.h`. Além disso, também definimos os novos registros do arquivo local `/etc/system.conf` para tentarmos inicializar os novos escalonadores junto com o sistema. Copiamos o arquivo `/usr/src/etc/system.conf` para definirmos esses registros. Registramos todas as nossas alterações executando, em `/usr/src/minix/commands/minix-service`, o comando `make` seguido do comando `make install`. Após isso, fomos ao diretório `/usr/src/releasetools` e executamos os comandos, em ordem, `make includes` e `make hdboot`. Finalmente, reiniciamos o sistema e concluímos as compilações necessárias.

Ao ligarmos o sistema após as compilações, nos deparamos com uma mensagem *panic* enviada pelo próprio sistema (figura 6). Essa mensagem vagamente dizia que o serviço RS recebeu uma resposta inválida de algum serviço do sistema, e apresentava uma *stacktrace* com vários endereços de memória. Para tentarmos resolver essa mensagem, primeiramente pensamos que quando nossa comunicação entre processos falhava (pensamos que nossos serviços `sched_RR` e `sched_FCFS` acabaram não se inicializando com o sistema), estávamos retornando um valor incorreto (-1). Alteramos esse valor para `EBADEPT`, que é um valor típico do sistema e tentamos recompilar, mas não conseguimos sucesso. Por fim, tentamos alterar a inicialização do *system event framework* (SEF) para os escalonadores que criamos, e copiamos a inicialização do escalonador original, que chama uma função `sef_local_startup()`, mas que não registra o *timer* de balanceamento de prioridades. Infelizmente, ainda assim não conseguimos resolver o erro.

Cada compilação, em nossas máquinas, demorava cerca de 20 minutos, o que tornava inviável que continuássemos tentando por muito mais tempo. Não conseguimos compreender exatamente o erro que recebemos, pois o sistema só nos entregava endereços de memória. Dessa forma, não conseguimos testar os nossos escalonadores novos.

VII. Conclusão

Por fim, mesmo que nossas alterações não tenham surtido efeito, acreditamos que conseguimos andar no caminho correto para implementarmos um gerenciador de escalonadores e dois escalonadores novos, alterando o serviço existente e criando dois novos serviços. O projeto foi extremamente importante para aprendermos mais a fundo como funciona o sistema Minix3. Estudamos e conseguimos entender como um S.O. real pode realizar o escalonamento de processos, e conseguimos entender a política padrão do sistema, que é mais complexa que as que tentamos implementar e utiliza balanceamentos periódicos de prioridades.

Além disso, pudemos reforçar nossos entendimentos acerca da comunicação entre processos (IPC) e como ela é implementada no Minix3. Revisamos conceitos como *system event framework* (SEF), *endpoints* e *messages*. Também aprendemos como os serviços são inicializados junto com o sistema: descobrimos, como dissemos no relatório, que o servidor RS percorre um arquivo chamado `system.conf` para ligar os serviços que ali estão, e realizamos nossos novos registros nesse arquivo. Finalmente, conseguimos modelar e pensar em maneiras de testar um escalonador, analisando métricas de eficiência e criando processos a partir de um `fork()` e executando processos com um `execlp()`, relembrando como funciona a criação de processos em Unix. Assim sendo, mesmo que não tenhamos conseguido testar nossos escalonadores, entendemos que o projeto foi benéfico para nosso aprendizado no geral.

VIII. Divisão do trabalho

- A equipe inteira colaborou com discussões gerais sobre o andamento do projeto e erros que encontrávamos pelo caminho
- Matheus e André: discussões mais aprofundadas sobre os escalonadores e criação do gerenciador de escalonadores, dos escalonadores sched_RR, sched_FCFS e das respectivas seções do relatório.
- Daniel e Guilherme: discussões mais aprofundadas sobre os métodos de teste, e criação dos testes. Realização das respectivas seções do relatório, além do resumo e conclusão.
- Mais detalhes podem ser encontrados nos *commits* do Gitlab do grupo12 (Arquivos de teste estão no repositório projeto3-aux).

APÊNDICE A
FIGURAS

```
case SCHEDULING_INHERIT:
{
        /* Send msg to sched_FCFS */
        endpoint_t sched_FCFS_ep; /* sched_FCFS endpoint */
        if (minix_rs_lookup("sched_FCFS", &sched_FCFS_ep) != 0)
        {
                printf("Server sched_FCFS is not up\n");
                return EBADEPT;
        }
        result = _taskcall(sched_FCFS_ep, SCHEDULING_INHERIT, &m_in);
        break;
}
```

Figura 1: Novo tratamento da mensagem `SCHEDULING_INHERIT` em *sched*

```
service sched_RR
{
        uid     0;
        ipc     ALL_SYS;        # All system ipc targets allowed
        system  ALL;            # ALL kernel calls allowed
        vm      BASIC;          # Only basic VM calls allowed
        io      NONE;           # No I/O range allowed
        irq     NONE;           # No IRQ allowed
        sigmgr          rs;     # Signal manager is RS
        scheduler   KERNEL;     # Scheduler is KERNEL
        priority        4;      # priority queue 4
        quantum       500;      # default server quantum
};

service sched_FCFS
{
        uid     0;
        ipc     ALL_SYS;        # All system ipc targets allowed
        system  ALL;            # ALL kernel calls allowed
        vm      BASIC;          # Only basic VM calls allowed
        io      NONE;           # No I/O range allowed
        irq     NONE;           # No IRQ allowed
        sigmgr          rs;     # Signal manager is RS
        scheduler   KERNEL;     # Scheduler is KERNEL
        priority        4;      # priority queue 4
        quantum       500;      # default server quantum
};
```

Figura 2: Entradas adicionadas em system.conf

```
# ./track
Parent process with PID: 4706. Started.
Child process with PID: 4707 and PPID: 4706. Started.
Parent process with PID: 4706. Finished.
Child process with PID: 4707 and PPID: 4706. Finished.
Parent process with PID: 4706. Started.
Child process with PID: 4708 and PPID: 4706. Started.
Parent process with PID: 4707. Started.
Parent process with PID: 4706. Finished.
Child process with PID: 4709 and PPID: 4707. Started.
Parent process with PID: 4707. Finished.
Child process with PID: 4708 and PPID: 4706. Finished.
Child process with PID: 4709 and PPID: 4707. Finished.
#
```

Figura 3: Resultado da execução do programa track_process_test

```
# ./child
-------- CHILD PROCESS --------
Process:  1
Process:  1
Process:  2
Process:  2
Process:  3
Process:  3
Process:  4
Process:  4
Process:  5
Process:  5
Process:  6
Process:  6
Process:  7
Process:  7
Process:  8
Process:  8
Process:  9
Process:  9
Process: 10
Process: 10
** Process  1 finished
** Process  1 turnaround time: 1.500000 seconds
** Process  1 waiting time: 0.633350 seconds
** Process  2 finished
** Process  2 turnaround time: 1.500000 seconds
** Process  3 finished
** Process  2 waiting time: 0.749975 seconds
** Process  3 turnaround time: 1.500000 seconds
** Process  3 waiting time: 0.583300 seconds
** Process  4 finished
** Process  4 turnaround time: 1.516675 seconds
** Process  5 finished
** Process  4 waiting time: 0.416675 seconds
** Process  5 turnaround time: 1.500000 seconds
** Process  5 waiting time: 0.950025 seconds
** Process  6 finished
** Process  6 turnaround time: 1.500000 seconds
** Process  7 finished
** Process  6 waiting time: 0.716650 seconds
** Process  7 turnaround time: 1.500000 seconds
** Process  7 waiting time: 0.633325 seconds
** Process  8 finished
** Process  8 turnaround time: 1.500000 seconds
** Process  9 finished
** Process  8 waiting time: 0.466650 seconds
** Process  9 turnaround time: 1.500000 seconds
** Process  9 waiting time: 0.883300 seconds
** Process 10 finished
** Process 10 turnaround time: 1.483350 seconds
** Process 10 waiting time: 0.766625 seconds
#
```

Figura 4: Resultado da execução do programa child_test

```
# ./normal_test
-------- NORMAL PROCESS --------
Process:  1
Process:  1
Process:  2
Process:  2
Process:  3
Process:  3
Process:  4
Process:  4
Process:  5
Process:  5
Process:  6
Process:  6
Process:  7
Process:  7
Process:  8
Process:  8
Process:  9
Process:  9
Process: 10
Process: 10
** Process  1 finished
** Process  1 turnaround time: 1.500000 seconds
** Process  1 waiting time: 0.549950 seconds
** Process  2 finished
** Process  2 turnaround time: 1.500000 seconds
** Process  3 finished
** Process  2 waiting time: 0.466600 seconds
# ** Process  3 turnaround time: 1.516650 seconds
** Process  4 finished
** Process  3 waiting time: 0.716575 seconds
** Process  4 turnaround time: 1.516650 seconds
** Process  4 waiting time: 0.916575 seconds
** Process  5 finished
** Process  5 turnaround time: 1.516650 seconds
** Process  5 waiting time: 0.749975 seconds
** Process  6 finished
** Process  6 turnaround time: 1.500000 seconds
** Process  6 waiting time: 0.866700 seconds
** Process  7 finished
** Process  7 turnaround time: 1.500000 seconds
** Process  8 finished
** Process  7 waiting time: 0.416775 seconds
** Process  8 turnaround time: 1.500000 seconds
** Process  9 finished
** Process  8 waiting time: 0.600075 seconds
** Process  9 turnaround time: 1.516675 seconds
** Process 10 finished
** Process  9 waiting time: 0.666725 seconds
** Process 10 turnaround time: 1.516675 seconds
** Process 10 waiting time: 0.716600 seconds
```

Figura 5: Resultado da execução do programa normal_test

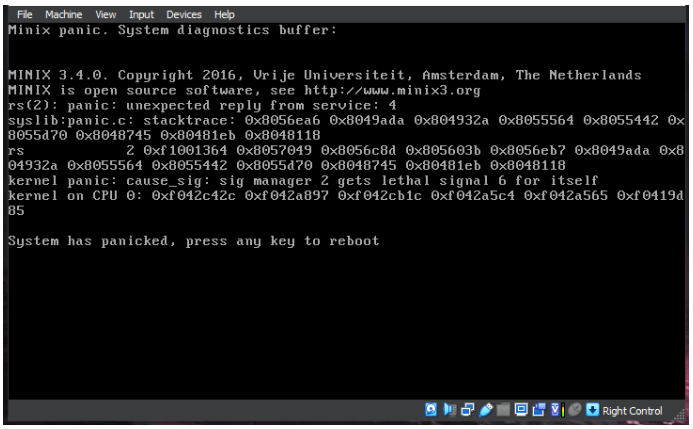

Figura 6: Mensagem apresentada pelo Minix3 na inicialização do sistema após as alterações

REFERÊNCIAS

[1] Tanenbaum, Andrew S.; Woodhull, Albert S. (1987). Operating Systems: Design and Implementation. ISBN 9780131429383.

[2] Minix3 - IPA about the creation of the user mode scheduler, url = https://www.minix3.org/docs/scheduling/report.pdf

[3] Minix3 developers guide - driver programming, url = https://wiki.minix3.org/doku.php?id=developersguide:driverprogramming