Sistemas Operacionais
Escalonamento no MINIX3

Gabriel de Mello Cambuy Ferreira - RA: 142641
Gabriela Carregari Verza - RA: 148756
Melissa Frigi Mendes - RA: 133693
Raissa Barbosa dos Santos - RA: 148551

Universidade Federal de São Paulo
São José dos Campos

Agosto de 2024

1. Implementação

Neste trabalho, o grupo optou por implementar o método de escalonamento por loteria, considerando que quanto mais prioritário o processo, mais bilhetes ele recebe. Como a implementação desse método depende das filas de prioridade, não foram feitas modificações no servidor de escalonamento sched. A alteração realizada pelo grupo foi no arquivo proc.c, mais especificamente na função pick_proc(), responsável por escolher o processo a ser executado.

No Anexo A, temos o trecho de código correspondente a função pick_proc() original. O funcionamento dessa função se dá da seguinte forma: O algoritmo percorre as filas de processos prontos (run_q_head), começando pela fila de maior prioridade (q = 0). Para cada fila, verifica se há processos prontos por meio de “rdy_head[q]”. Se encontrar um processo, identificado pela variável “rp”, verifica se ele está em estado executável pela chamada da função “proc_is_runnable(rp)”. Se o processo for "billable" (cobrável), ele é marcado (get_cpulocal_var(bill_ptr) = rp). O primeiro processo com maior prioridade encontrado é selecionado para execução, sendo o retorno da função.

O Anexo B apresenta o trecho de código implementado pelo grupo para a função pick_proc(). Essa versão da função funciona da seguinte maneira: Primeiro, o total de bilhetes é calculado com base na prioridade das filas. Cada processo na fila de prioridade “q” recebe “NR_SCHED_QUEUES - q” bilhetes. O valor definido para “NR_SCHED_QUEUES” é 16, e “q” itera de 0 a 15, de forma que os processos da fila 0 recebem 16 bilhetes, os da fila 1 recebem 15 bilhetes, e assim sucessivamente, até os processos da fila 15, que recebem apenas 1 bilhete cada. Um número aleatório (winning_ticket) entre 0 e o total de bilhetes calculado é gerado. As filas são percorridas e os bilhetes somados novamente; quando a soma dos tickets excede o valor de “winning_ticket”, o processo correspondente é selecionado. Se o processo selecionado for "billable", ele é marcado (get_cpulocal_var(bill_ptr) = rp) e retornado pela função.

2. Execução de testes

A fim de comparar os desempenhos dos métodos de escalonamento citados anteriormente, o grupo planejou alguns experimentos. Para tal, foi necessário realizar a instalação do MINIX3 em uma máquina virtual por meio do software Oracle VM VirtualBox, e clonagem do código fonte do MINIX3 no caminho /usr/src. Também foi configurada uma pasta compartilhada pelo VirtualBox, e montada no MINIX através do comando **# mount -t vbfs -o share=NAME none /mnt**, onde NAME foi substituído pelo nome da pasta selecionada para essa tarefa no host. Essa pasta compartilhada permitiu a transferência de arquivos entre o SO host e o MINIX, e era acessada no MINIX através do comando **# cd /mnt**.

Os testes foram baseados na execução da aplicação test.c, através da qual foi possível parametrizar o número de processos (sendo 50% destes de IO, e os outros 50%

de CPU) e, para cada processo, a quantidade de operações de CPU e IO. O grupo planejou três cenários de execução: todos com 100 processos, mas variando a quantidade de operações entre carga leve, média e pesada.

O primeiro passo foi incluir o arquivo test.c na pasta compartilhada e gerar seu executável, por meio do comando **# clang test.c -o test** no terminal. Então, considerando que nesse momento estávamos lidando com o código fonte original do MINIX, foram executados três comandos:

1) # **./test 100 100000 10000000**;
2) # **./test 100 1000000 100000000**;
3) # **./test 100 3000000 300000000**.

A seguir, o arquivo proc.c, com as alterações realizadas pelo grupo, foi copiado para a pasta compartilhada, e movido para o caminho do arquivo proc.c do código fonte original pelo comando **# mv /mnt/proc.c /usr/src/minix/kernel**. Esse comando substitui o arquivo que encontra no diretório destino pelo arquivo do diretório origem. Após isso, para recompilar os fontes, foram executados os comandos **# make** e **# make install**.

E, mais uma vez, foram executados os três comandos para testes citados anteriormente.

3. Resultados obtidos e discussões

Foram utilizados gráficos e cálculos de variância (quanto os valores de um conjunto de dados se desviam da média desses dados) para comparar os desempenhos dos escalonadores.

Tabela 1 - Variâncias calculadas para escalonador Round-Robin com múltiplas filas de prioridade

Caso de Teste / Processos	**CPU-bound**	**IO-bound**
Carga Leve	0,0001	0,0233
Carga Média	0,0001	6,0911
Carga Pesada	1,3491	33,2298

Tabela 2 - Variâncias calculadas para escalonador por loteria

Caso de Teste / Processos	**CPU-bound**	**IO-bound**
Carga Leve	0,0001	0,0460
Carga Média	0,0001	6,5133
Carga Pesada	1,4223	21,9444

3.1 Carga leve: 10 milhões de operações de CPU e 100 mil operações de IO

No cenário de carga leve, para processos CPU-bound, ambos os escalonadores apresentaram tempos de retorno similares, baixos e com pouca variância, com alguns processos sendo executados imediatamente após a criação (tempos nulos).

Para processos IO-bound, o escalonador round-robin teve tempos de retorno com menos variância, mas que foram mais altos em comparação com o escalonador por loteria. Além disso, para ambos escalonadores, os tempos de retorno desses processos foram muito maiores do que os processos CPU-bound, o que indica inanição.

A execução com o escalonador round-robin durou 23 minutos, enquanto a execução com o escalonador por loteria durou 22 minutos. Os arquivos gerados nas execuções estão no Anexo C.

Figura 1 - Gráficos com tempos de retorno por processo em Carga Leve - CPU-bound

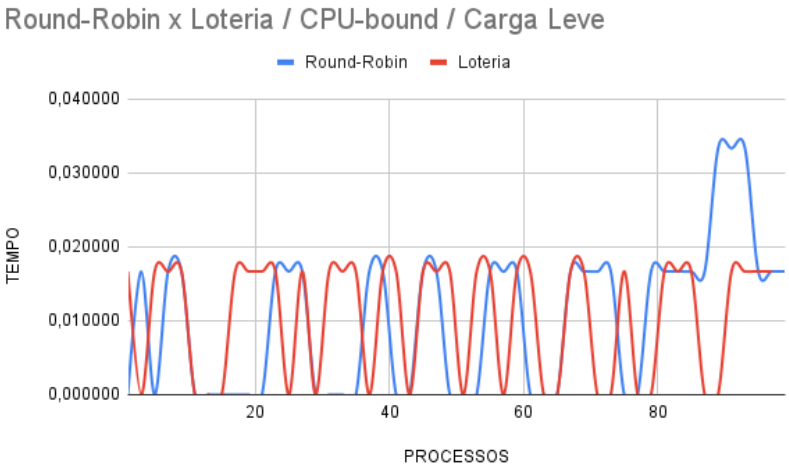
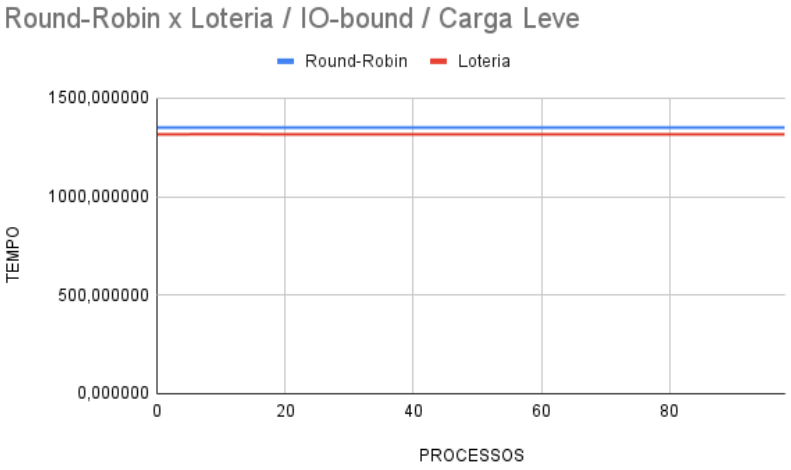


Figura 2 - Gráficos com tempos de retorno por processo em Carga Leve - IO-bound



3.2 Carga média: 100 milhões de operações de CPU e 1 milhão de operações de IO

No cenário de carga média, para processos CPU-bound, ambos os métodos de escalonamento apresentaram pouca variância nos tempos de retorno, sendo os resultados bastante próximos.

Para processos IO-bound, novamente tanto o escalonador round-robin quanto o por loteria apresentaram tempos de retorno muito maiores do que os processos CPU-bound, sendo os tempos do escalonador por loteria maiores e com mais variância.

A execução com o escalonador round-robin durou 3 horas e 33 minutos, enquanto a execução com o escalonador por loteria durou 3 horas e 36 minutos. Os arquivos gerados nas execuções estão no Anexo C.

Figura 3 - Gráficos com tempos de retorno por processo em Carga Média - CPU-bound

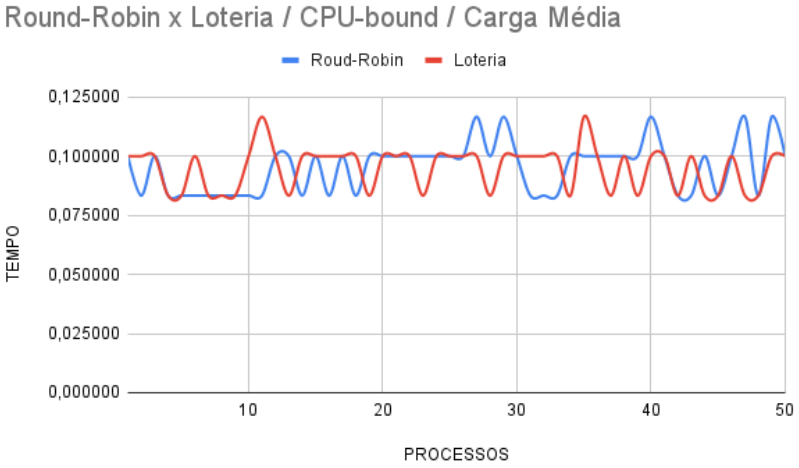
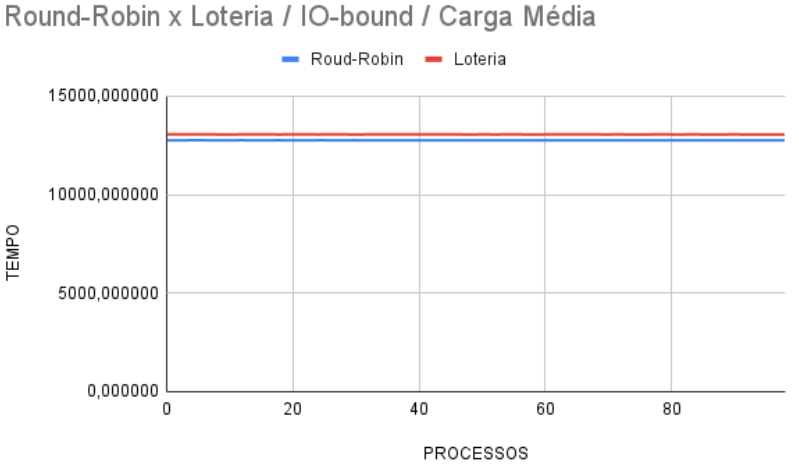


Figura 4 - Gráficos com tempos de retorno por processo em Carga Média - IO-bound



3.3 Carga pesada: 300 milhões de operações de CPU e 3 milhões de operações de IO

No cenário de carga pesada, para processos CPU-bound, ambos escalonadores apresentaram tempos de retorno semelhantes, porém a variância desses tempos no escalonador por loteria foi maior.

Para processos IO-bound, o escalonador round-robin obteve tempos de retorno mais altos e com mais variância em relação ao escalonador por loteria. Os tempos de retorno de IO-bound foram novamente bem mais altos do que os de CPU-bound para ambos escalonadores, porém essa diferença foi acentuada nesse teste.

A execução com o escalonador round-robin durou 12 horas e 20 minutos, enquanto a execução com o escalonador por loteria durou 12 horas e 42 minutos. Os arquivos gerados nas execuções estão no Anexo C.

Figura 5 - Gráficos com tempos de retorno por processo em Carga Pesada - CPU-bound

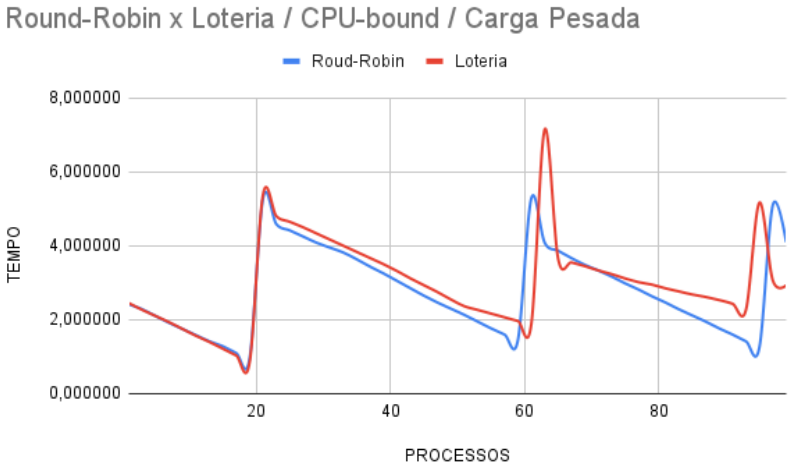
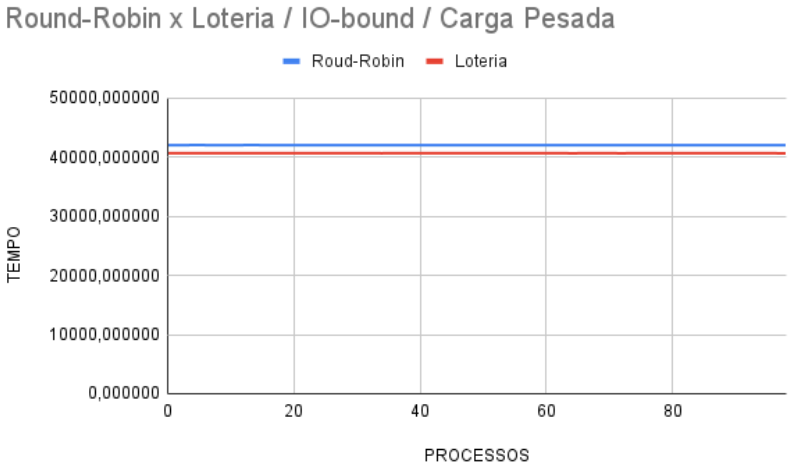


Figura 6 - Gráficos com tempos de retorno por processo em Carga Pesada - IO-bound



3.4 Conclusão sobre os testes

Ao comparar os tempos de retorno e variância desses nos escalonadores round-robin de múltiplas filas de prioridade e por loteria, observou-se que ambos apresentaram tempos e variâncias muito próximas nos três cenários de teste para processos CPU-bound. Já para processos IO-bound, o escalonador round-robin apresentou menor variância nos tempos de retorno nos cenários de carga leve e média, mas teve tempos de retorno mais altos em comparação ao escalonador por loteria nos testes de carga leve e pesada. O escalonador por loteria apresentou tempos de retorno e variância bem menores nos testes de carga pesada.

Portanto, a escolha entre os dois escalonadores pode depender do tipo de carga e dos requisitos específicos do sistema. O escalonador round-robin pareceu ser mais eficiente em termos de tempo de execução geral, enquanto o escalonador por loteria pode oferecer vantagens em certos cenários de carga pesada para processos IO-bound.

4. Contribuições

Em termos das atividades práticas, o grupo optou por realizar o trabalho em conjunto, para que todos contribuíssem igualmente e estivessem a par do que foi realizado. Para isso, foram realizadas algumas reuniões via Discord para tratar os seguintes tópicos: a instalação do MINIX3 na máquina virtual e configuração do ambiente, decidir qual seria o método de escalonamento a ser implementado, realizar as alterações no código fonte do MINIX e planejar os testes a serem executados.

Os membros também aproveitaram as reuniões no Discord para compor os tópicos 1 e 2 deste relatório, além deste tópico 4 e inclusão dos anexos, gráficos e tabelas. Em relação ao tópico 3, Gabriel ficou responsável pela análise e discussão dos resultados do cenário de carga leve, Melissa pela análise e discussão dos resultados do cenário de carga média, Raissa pela análise e discussão dos resultados do cenário de carga pesada, e Gabriela pela conclusão sobre os testes.

Vale ressaltar que os membros estavam sempre em contato através de grupo no WhatsApp para organizar as tarefas de cada um na composição do relatório e vídeo apresentação.

Anexo A - Função pick_proc() original

```
static struct proc * pick_proc(void)
{
  register struct proc *rp;
  struct proc **rdy_head;
  int q;

  rdy_head = get_cpulocal_var(run_q_head);
  for (q=0; q < NR_SCHED_QUEUES; q++) {
       if(!(rp = rdy_head[q])) {
              TRACE(VF_PICKPROC, printf("cpu %d queue %d empty\n", cpuid, q););
              continue;
       }
       assert(proc_is_runnable(rp));
       if (priv(rp)->s_flags & BILLABLE)
              get_cpulocal_var(bill_ptr) = rp;
       return rp;
  }
  return NULL;
}
```

Anexo B - Função pick_proc() implementada pelo grupo

```
static struct proc * pick_proc(void)
{
    register struct proc *rp;
    struct proc **rdy_head;
    int q;
    int total_tickets = 0;
    int ticket_count = 0;

    rdy_head = get_cpulocal_var(run_q_head);
    for (q = 0; q < NR_SCHED_QUEUES; q++) {
        rp = rdy_head[q];
        while (rp) {
            total_tickets += (NR_SCHED_QUEUES - q);
            rp = rp->p_nextready;
        }
    }

    if (total_tickets == 0) {
        return NULL;
    }

    int winning_ticket = random() % total_tickets;

    for (q = 0; q < NR_SCHED_QUEUES; q++) {
        rp = rdy_head[q];
        while (rp) {
            ticket_count += (NR_SCHED_QUEUES - q);
            if (ticket_count > winning_ticket) {
                if (priv(rp)->s_flags & BILLABLE) {
                    get_cpulocal_var(bill_ptr) = rp;
                }
                return rp;
            }
            rp = rp->p_nextready;
        }
    }

    return NULL;
}
```

Anexo C - Resultados dos testes

Round-Robin com múltiplas filas de prioridade - Carga leve
Round-Robin com múltiplas filas de prioridade - Carga média
Round-Robin com múltiplas filas de prioridade - Carga pesada
Escalonamento por loteria - Carga leve
Escalonamento por loteria - Carga média
Escalonamento por loteria - Carga pesada