Sistemas Operacionais
Relatório do Trabalho Prático 2

Vinicius Silva Gomes e Mirna Mendonça e Silva

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brasil

26 de novembro de 2023

1 Introdução

O objetivo principal desse trabalho é compreender como funciona o escalonamento de processos no sistema operacional xv6 e, em seguida, estendê-lo, alterando o tempo em que os processos sofrem preempção, implementando uma nova política de escalonamento, entre outras tarefas.

Sendo assim, a primeira parte do trabalho será destinada a entender e explorar como o escalonador padrão do XV6 funciona, quais são suas políticas, particularidades e como ele se comporta em determinadas situações: o que acontece quando um novo processo é criado, quando um processo volta de uma tarefa de I/O, como ele decide qual será o próximo processo a executar, etc.

Em seguida, algumas modificações serão realizadas. Dentre elas, destaca-se o aumento do tempo entre preempções no processador, a implementação de uma nova política de escalonamento e de novas chamadas de sistema.

Por fim, as modificações realizadas serão testadas através de programas que utilizem as chamadas de sistema implementadas e coloquem o SO em situações onde as extensões realizadas no escalonamento sejam evidentes.

2 O escalonador do XV6

O primeiro passo foi entender como o escalonador de processos do XV6 funciona. Para realizar essa tarefa, quatro principais perguntas devem ser respondidas: qual política de escalonamento é utilizada atualmente no XV6, quais processos essa política seleciona para executar, o que acontece quando um processo retorna de uma tarefa de I/O e o que acontece quando um processo é criado e quando ou o quão frequentemente o escalonamento acontece.

Para tanto, o arquivo `proc.c` foi a principal parte do código a ser analisada. Esse arquivo reúne o código do escalonador e o código de funções auxiliares, como a função que realiza a troca de contexto do processo que vai entrar ou sair de execução, a função que realiza a criação e alocação de processos na memória, etc.

Sendo assim, algumas observações foram feitas sobre o escalonador do XV6:

- A política de escalonamento implementada por padrão no XV6 é o *Round Robin*. A ideia dessa abordagem é varrer a fila de processos no estado de PRONTO e selecionar processos de forma sequencial para serem executados. No entanto, ela permite que eles realizem processamento apenas por uma quantidade finita de tempo, dando lugar a outro processo quando esse tempo acaba. Ou seja, o *Round Robin* é um algoritmo de escalonamento preemptivo e que não atribui prioridades aos processos.

- O escalonamento padrão do XV6 ocorre a cada tick de clock. Ou seja, a cada tick do processador, o processo que está executando deixa a CPU e o escalonador escolhe um novo processo para ser executado. Essa abordagem é vantajosa pois permite que processos menores não fiquem muito tempo esperando por processos maiores (oferecendo um uso mais eficiente da CPU), mas o tempo de preempção deve ser muito bem pensado pois as trocas de contexto podem se tornar grandes *overheads* no custo do escalonamento.

- A política implementada pelo XV6 seleciona apenas processos no estado `RUNNABLE` para serem executados. De fato, processos nesse estado são processos que se encontram na fila de PRONTO e estão aptos a receberem acesso à CPU, para continuarem sua execução. Como não há critério para a escolha, ao varrer a tabela de processos, o primeiro processo encontrado que esteja no estado `RUNNABLE` receberá acesso à CPU.

- Quando um processo sai da CPU para realizar uma tarefa de I/O, ele vai para o estado `SLEEPING`. Enquanto está nesse estado, esse processo não será escolhido pelo escalonador. Quando a tarefa de I/O é finalizada, o processo retorna para o estado `RUNNABLE` e, portanto, se torna novamente um candidato a ser escolhido pelo escalonador para executar na CPU.

- Um processo é criado a partir da função `fork()`. Essa função chama a função `allocproc()` para alocar um espaço para o novo processo na tabela de processos. Essa alocação se dá através de uma busca por processos no estado `UNUSED`. Quando um processo nesse estado é encontrado, ele é retornado pela `allocproc()` e a função `fork()` termina de inicializar o processo, inserindo as informações do processo que a chamou (processo pai).

3 Alterando o tempo de preempção

No XV6, a preempção ocorre a cada tick de clock. O objetivo é aumentar esse tempo de preempção para 5 ticks. Esse valor foi mapeado para uma constante `INTERV` no SO. Para tanto, um atributo `quantum` será incluído na estrutura de dados do processo. Quando um processo está executando na CPU, esse atributo será incrementado a cada tick numa função que atualiza os tempos que o processo despendeu em cada fila do escalonador (veja a seção 5). Além disso, quando o próximo processo a ser executado é decidido e antes dele iniciar sua execução, esse atributo é zerado, para que ele possa executar novamente pelos 5 ticks definidos.

Por fim, uma última alteração foi realizada no arquivo `trap.c`. Esse arquivo define uma série de TRAP's que são executadas em determinadas condições. Ele define, também, um teste que preempta o processo corrente a cada tick de clock, usando a função `yield()`. Esse teste foi modificado para executar a função `yield()` apenas quando o `quantum` do processo que está executando for maior ou igual a quantidade de ticks desejada. Assim, o processo executa pela quantidade de ticks definida e é corretamente preemptado após o tempo de CPU dele acabar.

```
...

// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
{
  // Se o processo ja rodou por INTERV ticks, preempta ele usando a funcao yield
  if(myproc()->quantum >= INTERV)
  {
    yield();
  }
}

...
```

4 Escalonamento de filas multinível (MLQS)

A próxima tarefa é modificar a política de escalonamento do XV6. Para tanto, a nova implementação será uma política de escalonamento baseada em filas de multinível (*Multi-Level Queue Scheduling* ou MLQS).

Para realizar a implementação dessa política, cujos detalhes são melhor descritos na especificação do projeto, um novo atributo `priority` será incluído na estrutura de dados do processo. Esse atributo será inicializado com o valor 2 nas funções `allocproc()` e `fork()`, o que determina 2 como a prioridade inicial de todos os processos.

Além disso, para permitir a manipulação das prioridades por parte do usuário, duas chamadas de sistema diferentes foram criadas: `change_prio()` e `set_prio()`. A primeira recebe como parâmetro um valor de

prioridade e altera o atributo `priority` do processo que está executando no momento para a prioridade passada como parâmetro. A segunda, de forma semelhante, também altera a prioridade, mas do processo com o `pid` igual ao recebido como parâmetro. Para tanto, a tabela de processos será percorrida, procurando pelo processo com `pid` fornecido. Ambas as chamadas de sistema retornam 0 caso a operação seja bem-sucedida e -1 caso contrário (valor de prioridade não está entre 1 e 3 ou `pid` não encontrado, por exemplo).

```
...

int
set_prio(int priority, int pid)
{
  int priority_changed = 0;

  if (priority <= 0 || priority > 3)
    return -1;

  acquire(&ptable.lock);
  struct proc *p;

  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid == pid)
    {
      p->priority = priority;
      priority_changed = 1;
      break;
    }
  }

  release(&ptable.lock);

  if (priority_changed == 0)
    return -1;

  return 0;
}

int
change_prio(int priority)
{
  if (priority <= 0 || priority > 3)
    return -1;

  acquire(&ptable.lock);
  struct proc *p = myproc();

  p->priority = priority;

  release(&ptable.lock);

  return 0;
}

...
```

Com as prioridades atribuídas, o escalonador do XV6 será modificado para procurar não mais pelo primeiro processo pronto para ser executado, mas pelo processo com a maior prioridade na fila de PRONTO. Uma vez que esse processo seja encontrado, ele será escalonado para a CPU e irá executar até deixar o processamento por vontade própria ou ser preemptado. Assim, os processos com maior prioridade serão escolhidos primeiro e os com menor prioridade serão escolhidos posteriormente, assim como a política do escalonamento define.

Todas essas modificações ocorreram na função principal do escalonamento, `scheduler()`, no arquivo `proc.c`.

```
...

// Escalonador percorre toda a lista de processos
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
```

```
{
  // Processo nao esta pronto para executar -> vai para o proximo
  if (p->state != RUNNABLE)
    continue;

  high = p;

  // Percorre a tabela de processos prontos para executar
  // e escolhe aquele com maior prioridade
  for (q = ptable.proc; q < &ptable.proc[NPROC]; q++)
  {
    // Processo nao esta pronto para executar -> vai para o proximo
    if(q->state != RUNNABLE)
      continue;

    // Processo com prioridade maior que a maior prioridade
    // observada ate o momento
    if(q->priority > high->priority)
      high = q;
  }

  // Define ele como proximo processo a executar
  p = high;

  ...
}

...
```

No entanto, é possível que situações de inanição (ou *starvation*) aconteçam, isto é, por ter prioridade baixa, um processo nunca é escalonado para a CPU por sempre existirem processos de prioridade maior disponíveis para o escalonador.

Para resolver esse problema, será implementada uma estratégia de *aging*. A ideia é aumentar a prioridade de processos que estão há muito tempo no estado de PRONTO. Na implementação realizada, uma quantidade fixa de ticks de clock será definida para que um processo transite entre as filas de prioridade. Mais especificamente, da fila de prioridade 1 para a 2, um processo deve ficar 200 ticks de clock no estado de PRONTO e da fila de prioridade 2 para a 3, um processo deve ficar 100 ticks de clock. Esses valores foram mapeados para as constantes `Q1TOQ2` e `Q2TOQ3`, respectivamente, para que seja possível alterá-los de maneira prática e impacto do mecanismo de *aging* possa ser avaliado.

Além disso, será necessário um atributo para determinar há quanto tempo um processo está na mesma fila de PRONTO. Para tanto, o atributo `queue_time` será criado na estrutura de dados do processo. Esse atributo irá iniciar como 0 e é incrementado a cada tick de clock que um processo passa no estado `RUNNABLE`. Assim como o *quantum* do processo, esse atributo é incrementado por uma função que executa a cada tick de clock e atualiza as métricas medidas e os atributos de tempo definidos (veja a seção 5).

Assim, é possível implementar o código que realiza o envelhecimento dos processos. Esse código será executado após um processo largar a CPU e antes do próximo ser escolhido e irá percorrer toda a tabela, procurando por processos no estado `RUNNABLE` com `queue_time` maior que as constantes definidas. Caso um processo com prioridade 1 e `queue_time` maior que `Q1TOQ2` seja encontrado, ele terá sua prioridade aumentada para 2 e o `queue_time` resetado. O mesmo acontece para um processo transitar da fila de prioridade 2 para a 3, usando a constante `Q2TOQ3` como parâmetro. Sempre que um processo recebe acesso à CPU, ele tem seu `queue_time` resetado.

```
...

// Mecanismo de aging para evitar inanicao
for (q = ptable.proc; q < &ptable.proc[NPROC]; q++)
{
  if (q->state != RUNNABLE)
    continue;

  if (q->priority == 1 && q->queue_time >= Q1TOQ2) {
    q->priority = 2;
    q->queue_time = 0;
```

```
  } else if (q->priority == 2 && q->queue_time >= Q2TOQ3) {
    q->priority = 3;
    q->queue_time = 0;
  }
}

...
```

5 Medindo o tempo despendido em cada fila

Para testar a nova política escalonamento, o processo de *aging* e o aumento no tempo de CPU dos processos, 3 novos atributos (métricas) serão incluídos na estrutura de dados do processo:

- `ctime`: marca o tempo em que o processo foi criado;
- `stime`: marca o tempo total que o processo gastou no estado `SLEEPING`;
- `retime`: marca o tempo total que o processo gastou no estado `RUNNABLE` (fila de PRONTO);
- `rutime`: marca o tempo total que o processo executou na CPU.

A métrica `ctime` é inicializada nas funções `allocproc()`, para que ela seja corretamente definida para o primeiro processo, e `fork()`, para os demais processos que são criados durante a execução.

Já as métricas `stime`, `retime` e `rutime` são atualizadas na função `update_time_attrs()`, que é executada a cada tick de clock. Essa função percorre toda a tabela de processos e verifica o estado de cada processo, incrementando uma unidade na métrica correspondente ao estado. Além disso, ela também atualiza o tempo de CPU do processo que está executando, incrementando o atributo `quantum` para processos no estado `RUNNING`, e o tempo que um processo esteve na fila de PRONTO numa mesma fila de prioridade, incrementando o atributo `queue_time` para processos no estado `RUNNABLE`.

```
...

void
update_time_attrs()
{
  struct proc* p;

  acquire(&ptable.lock);

  for(p=ptable.proc;p < &ptable.proc[NPROC];p++)
  {
    if(p->state == RUNNABLE)
    {
      p->retime++;
      p->queue_time++;
    }

    if(p->state == RUNNING)
    {
      p->rutime++;
      p->quantum++;
    }

    if(p->state == SLEEPING)
    {
      p->stime++;
    }
  }

  release(&ptable.lock);
}

...
```

6 Testes

Para testar as mudanças realizadas, três novas chamadas de sistema foram definidas:

- `wait2()`: possui praticamente a mesma implementação que a função `wait()` original. Entretanto, recebe três ponteiros como parâmetro e atribui o tempo total gasto pelo processo nos estados `RUNNABLE`, `RUNNING` e `SLEEPING` para cada um desses ponteiros. Isso será útil para avaliar o desempenho do escalonamento de maneira efetiva.
- `yield2()`: atua apenas como um *wrapper* para permitir que o usuário também seja capaz de que chamar a função `yield()` original, que inicialmente não está definida no espaço de chamadas de sistema acessíveis ao usuário.
- `wait3()`: atua da mesma forma que a função `wait2()`, mas ao invés de passar as métricas de tempo que são medidas para os processos, passa a prioridade com que ele finalizou sua execução, o que facilita a verificação do mecanismo de *aging*.

Com essas chamadas de sistema, dois programas de teste foram implementados: `sanity.c` e `priotest.c`. O primeiro recebe um inteiro n como parâmetro e cria $3 \cdot n$ processos, sendo cada um deles sendo um processo *CPU-Bound*, *S-Bound* ou *IO-Bound*. O programa pai, então, aguarda esses processos terminarem, usando a chamada de sistema `wait2()` e, ao final, exibe as métricas de tempo de cada um deles, juntamente das médias que podem ser calculadas a partir dessas métricas.

O segundo programa, por sua vez, cria 20 processos do tipo *CPU-Bound* e aguarda cada um deles terminar usando a chamada de sistema `wait3()`. Cada um desses processos receberá uma prioridade diferente, baseada em seu `pid`, e, ao final, será possível verificar a prioridade com que cada um deles terminou, o que possibilita a análise do mecanismo de *aging* e do escalonamento baseado em prioridades que foi implementado.

Além disso, esses dois programas também foram executados sob configurações diferentes dos parâmetros, com variações no tempo necessário para que um processo suba de prioridade, no tempo de preempção, etc. Por fim, o escalonamento MLQS implementado será comparado com o escalonamento original do XV6.

A seção 7 detalha os resultados obtidos para cada um desses testes.

7 Resultados

Para o programa `sanity.c`, o principal resultado obtido foram as médias para o tempo despendido em cada fila, para cada tipo de processo criado. Como era de se esperar, os processos *CPU-Bound* foram aqueles que terminaram primeiro, uma vez que deixavam a CPU apenas quando eram preemptados pelo escalonador e, portanto, progrediam mais na sua execução enquanto estavam usando a CPU.

Os processos *S-Bound* e *IO-Bound*, por sua vez, deixavam a CPU por vontade própria ou iam para o estado `SLEEPING` para simular processos de I/O usando o comando `sleep(1)`, respectivamente. Por isso, foram aqueles que mais demoraram para finalizar, ficando a maior parte do tempo no estado `RUNNABLE`. A tabela 1 apresenta as médias obtidas para cada tipo de processo. Nessa execução, foi utilizado o parâmetro $n = 15$, portanto, o número total de processos era 45. As métricas são medidas em ticks de clock.

Tipo	# de processos	*RUNNING*	*READY*	*SLEEPING*	*TURNAROUND*
CPU-Bound	15	26	219	0	245
S-Bound	15	12	543	0	556
IO-Bound	15	1	448	100	550

Tabela 1: Resultado retornado pelo programa `sanity.c`.

É possível perceber que a característica dos processos, como era de se esperar, impacta bastante no tempo de *turnaround*, de modo que os processos *CPU-Bound* foram aqueles que possuíram o menor valor observado para essa métrica. Os processos *S-Bound* e *IO-Bound*, por sua vez, foram os que apresentaram o maior tempo de *turnaround*, o que é justificado pelo grande tempo despendido no estado `RUNNABLE` e/ou no estado `SLEEPING` para os *IO-Bound*.

Já para o programa `priotest.c`, a análise que pode ser feita é que, por causa da política do escalonamento, os processos que inicialmente receberam prioridade 3 ficaram entre os primeiros a terminar. É possível reparar também que os processos de prioridade inicial 2 transitaram para a prioridade 3 e os processos de prioridade inicial 1 foram para a prioridade 2.

Além disso, uma observação interessante é que, num primeiro momento os processos de prioridade 3 são preferenciados mas, rapidamente, os primeiros processos de prioridade 2 sobem para a prioridade 3. Como os processos, após serem executados, vão para o final da fila de prioridade em que se encontram, os primeiros processos de prioridade 2 que ascendem para a prioridade 3 serão intercalados com esses processos que iniciaram com prioridade mais alta, de modo que muitos deles terminem juntos.

A tabela 2 apresenta o PID dos processos, a ordem com que cada um deles finalizou, a prioridade inicial e a prioridade final.

PID	Ordem de finalização	Prioridade inicial	Prioridade final
4	2	2	3
5	1	3	3
6	15	1	2
7	3	2	3
8	4	3	3
9	16	1	2
10	5	2	3
11	6	3	3
12	17	1	2
13	10	2	3
14	7	3	3
15	18	1	2
16	11	2	3
17	8	3	3
18	19	1	2
19	12	2	3
20	9	3	3
21	20	1	2
22	14	2	3
23	13	3	3

Tabela 2: Prioridade inicial e final dos processos disparados.

Ademais, os programas também foram executados sob as seguintes condições:

7.1 Variando as constantes de escalonamento

7.1.1 `INTERV`

A constante `INTERV` foi variada tanto para cima quanto para baixo. A situação mais interessante foi aquela onde o valor de `INTERV` foi baixo. O que foi possível notar é que houve um aumento geral nas médias obtidas para todos os tipos de processo.

Isso aconteceu pois a escolha do próximo processo a executar no escalonamento MLQS é mais complexa do ponto de vista computacional. Assim, as trocas de contexto constantes e as operações mais complexas para selecionar o próximo processo se tornam *overheads* impactantes na execução.

A tabela 3 apresenta as médias obtidas com `INTERV` = 1.

Tipo	# de processos	*RUNNING*	*READY*	*SLEEPING*	*TURNAROUND*
CPU-Bound	15	25	246	0	273
S-Bound	15	23	742	0	765
IO-Bound	15	1	578	100	679

Tabela 3: Resultado retornado pelo programa `sanity.c` com $n = 15$ e `INTERV` = 1.

Para valores de `INTERV` altos, foi possível observar uma redução no tempo de *turnaround* dos processos *CPU-Bound*, ocasionada por uma redução na média do tempo que esses processos ficaram no estado *READY*.

Em contrapartida, o tempo no estado *READY* dos processos *S-Bound* e *IO-Bound* aumentou de forma considerável. Isso acontece pois, com a maior demora para a preempção ocorrer, os processos *CPU-Bound* são capazes de realizar praticamente toda a sua execução sem largar a CPU e por isso, terminam mais rapidamente e são pouco impactados na espera pelos próximos processos.

Já os outros tipos de processo, por deixarem a CPU após poucos momentos executando, devem aguardar um período longo para que possam executar novamente. Isso ocasionou no aumento considerável no tempo de espera para ter acesso a CPU novamente.

A tabela 4 apresenta as médias obtidas com `INTERV` = 10.

Tipo	# de processos	*RUNNING*	*READY*	*SLEEPING*	*TURNAROUND*
CPU-Bound	15	19	130	0	149
S-Bound	15	31	800	0	831
IO-Bound	15	4	729	100	833

Tabela 4: Resultado retornado pelo programa `sanity.c` com $n = 15$ e `INTERV` = 10.

Uma observação que pode ser feita é que, como em ambas as situações os processos *S-Bound* e *IO-Bound* tiveram suas médias no estado *READY* incrementadas, a prioridade desses processos aumentou graças ao mecanismo de *aging*, alcançando a prioridade máxima definida.

7.1.2 `Q1TOQ2` e `Q2TOQ3`

A variação das constantes `Q1TOQ2` e `Q2TOQ3` levou a cenários interessantes. Quando elas assumem valores baixos, os processos rapidamente assumem a prioridade máxima, de modo que, após uma certa quantidade tempo, o escalonamento feito pelo MQLS nada mais é que um *Round Robin* na fila de prioridade máxima.

Para evitar situações como essa, é interessante que, primeiro, os tempos para mudança de fila sejam bem nivelados e ajustados e segundo, que existam, por exemplo, formas de um processo perder prioridade de forma sistemática, igual acontece para ganhar prioridade através do envelhecimento.

A tabela 5 apresenta o PID dos processos, a ordem com que cada um deles finalizou, a prioridade inicial e a prioridade final para `Q1TOQ2` = 50 e `Q2TOQ3` = 50.

Para valores altos, os processos dificilmente sobem de prioridade. Isso faz com que o processo de envelhecimento praticamente não aconteça e, com isso, os processos sejam escalonados e finalizem de acordo com suas prioridades iniciais, indo da maior prioridade para a menor. A tabela 6 apresenta o PID dos processos,

PID	Ordem de finalização	Prioridade inicial	Prioridade final
4	2	2	3
5	1	3	3
6	5	1	3
7	4	2	3
8	3	3	3
9	6	1	3
10	7	2	3
11	8	3	3
12	9	1	3
13	10	2	3
14	11	3	3
15	12	1	3
16	13	2	3
17	14	3	3
18	15	1	3
19	16	2	3
20	17	3	3
21	18	1	3
22	19	2	3
23	20	3	3

Tabela 5: Prioridade inicial e final dos processos disparados quando `Q1TOQ2` = 50 e `Q2TOQ3` = 50.

a ordem com que cada um deles finalizou, a prioridade inicial e a prioridade final para `Q1TOQ2` = 1000 e `Q2TOQ3` = 1000.

PID	Ordem de finalização	Prioridade inicial	Prioridade final
4	8	2	2
5	1	3	3
6	15	1	1
7	9	2	2
8	2	3	3
9	16	1	1
10	10	2	2
11	3	3	3
12	17	1	1
13	11	2	2
14	4	3	3
15	18	1	1
16	12	2	2
17	5	3	3
18	19	1	1
19	13	2	2
20	6	3	3
21	20	1	1
22	14	2	2
23	7	3	3

Tabela 6: Prioridade inicial e final dos processos disparados quando `Q1TOQ2` = 1000 e `Q2TOQ3` = 1000.

7.2 Comparação entre o MLQS e o escalonador original

Por fim, algumas execuções foram realizadas com o algoritmo de escalonamento original, para que ele pudesse ser comparado com o MLQS. Testando o programa `priotest.c`, o que foi possível observar é que o escalonamento com prioridades permite que os processos terminem em qualquer ordem (veja a tabela 2), a depender das prioridades atribuídas inicialmente e de como o processo de envelhecimento atua.

Por outro lado, no escalonamento original, como todos os processos recebem acesso a CPU de forma sequencial, os processos finalizam na ordem em que foram criados, assim como era esperado para um escalonamento *Round Robin* onde todos os processos são iguais. A tabela 7 apresenta o PID dos processos e a ordem com que cada um deles finalizou utilizando o algoritmo de escalonamento original do XV6.

PID	Ordem de finalização
4	1
5	2
6	3
7	4
8	5
9	6
10	7
11	8
12	9
13	10
14	11
15	12
16	13
17	14
18	15
19	16
20	17
21	18
22	19
23	20

Tabela 7: Ordem de finalização dos processos usando o escalonador padrão do XV6

Se comparado com as informações na tabela 2, é possível perceber diferenças claras na ordem com que os processos finalizam. No entanto, avaliar precisamente se existe alguma forma de escalonamento que é melhor ou pior e qual seria ela não é uma tarefa simples.

De fato, é necessário pensar bem para qual finalidade principal o escalonador será usado, quais são os principais tipos de processos que serão executados nele e com qual frequência, que tipo de recursos o SO pode precisar ou o usuário pode querer, entre muitas outras coisas.

Sendo assim, é preciso ponderar bem todos esses pontos e decidir aquele que favorece a métrica mais crítica do sistema, visto que todos eles apresentam vantagens e desvantagens, como é o caso da implementação mais simples do escalonador padrão do XV6 e, em contrapartida, a falta de possibilidade de definir alguns processos como mais importantes que outros, como o MLQS permite, e assim por diante.

8 Conclusões

De modo geral, o segundo trabalho da disciplina foi bastante interessante pois incentivou que os estudantes entendessem e modificassem um sistema operacional que é, de fato, implementado. Isso fez com que muitas dúvidas relacionadas a codificação e manipulação de algumas estruturas do SO, que antes tinham sido estudadas apenas na teoria, pudessem ser solucionadas de maneira prática.

Além disso, as modificações solicitadas no escalonamento também auxiliaram na fixação dos algoritmos vistos em sala de aula e contribuíram para um entendimento mais profundo das vantagens e desvantagens dos algoritmos de escalonamento que tivemos contato, a importância dos mecanismos de *aging* para escalonamentos com prioridade, entre muitos outros.

Referências

[1] A. S. Tanenbaum e H. Bos, *Modern Operating Systems*, 4ª edição.

[2] Silberschatz, Galvin e Gagne, *Fundamentos de Sistemas Operacionais*, 8ª edição.

[3] Remzi H. Arpaci-Dusseau e Andrea C. Arpaci-Dusseau, *Operatng Systems: Three Easy Pieces*, Versão 1.0.

[4] *Xv6, a simple Unix-like teaching operating system*, disponível em `https://pdos.csail.mit.edu/6.828/2012/xv6.html`.

[5] *Xv6 Operating System -adding a new system call*, disponível em `https://www.geeksforgeeks.org/xv6-operating-system-adding-a-new-system-call/`.

[6] *Adding System call in xv6*, disponível em `https://medium.com/@mahi12/adding-system-call-in-xv6-a5468ce1b463`.