Sistemas Operacionais
Trabalho Prático 2

Luís Felipe Ramos Ferreira Igor Lacerda Faria da Silva

lframos_ferreira@outlook.com

1 Introdução

O Trabalho Prático 2 da disciplina de Sistemas operacionais teve como proposta o estudo e modificação dos algoritmos de escalonamento de processos presente no *kernel* do sistema operacional XV6.

O repositório onde está armazenado o código utilizado durante o desenvolvimento desse projeto pode ser encontrado neste endereço.

2 Respostas

1. Qual a política de escalonamento é utilizada atualmente no XV6?

   A política de escalonamento utilizada no XV6 é uma política de

   Round Robin sem prioridades, ou seja, o escalonador irá checar continuamente a lista de processos disponíveis para serem executados e irá fornecer um tempo de processamento a cada um deles, sem que exista uma política de priorização dentre um processo a outro. O algoritmo é simples de se implementar, simples de compreender e não causa inanição aos processos, embora possua pontos negativos como os gargalos causados pela constante troca de contexto, a depender do tempo que cada processo terá para execução na CPU. É um algoritmo preemptivo, uma vez que força a saída de um processo da CPU caso o limite de tempo tenha sido atingido.

2. Quais processos essa política seleciona para rodar?

   A política citada seleciona os processos que estão disponíveis para serem executados conforme eles são checados na lista de processos disponíveis. Não há um tipo de prioridade estabelecida em cima sobre os processos, ou seja, os processos terão uma certa quantidade de tempo a cada momento que o escalonador encontrá-los na lista de processos disponíveis. É importante frisar que o escalonador irá checar `apenas` os processos marcados como disponíveis para serem executados, ou seja, processos dormindo ou esperando algum I/O não receberão tempo de processamento da CPU a menos que estejam prontos para serem executados e marcados como tal na lista de processos do sistema.

3. O que acontece quando um processo retorna de uma tarefa de I/O?

   O processo é marcado como *RUNNABLE*, isto é, está pronto para executar e entra para a lista de processos que podem ser executados. Assim, ele eventualmente será escolhido pelo escalonador para começar a rodar.

4. O que acontece quando um processo é criado e quando ou quão frequente o escalonamento acontece?

   Quando um processo é criado, uma referência para ele é criada no espaço de memória do sistema operacional, e a esse processo deve ser alocado um espaço de memória de usuário onde irá estar armazenado seu identificador, código, dados, pilha de execução e *heap*. Um processo pode ser criado no XV6 por meio da chamada de sistema *fork()*, que irá criar uma cópia do processo que fez a chamada da função. Para executar um novo programa, a chamada de sistema *exec()* deve ser utilizada.

O processo de escalonamento acontece, na implementação original do XV6, a cada 1 tick do clock. No entanto, esse parâmetro pode ser modificado para alterar o período entre as preempções realizadas pelo escalonador.

3 Algoritmos implementados

3.1 Escalonador

Uma das requisições do trabalho foi a de implementar uma modificação na política de escalonamento do *kernel* do XV6. Em particular, foi proposta a implementação de um escalonamento por meio de filas multinível. Para isso, a função *scheduler()* do arquivo *proc.c* foi alterada de modo a satisfazer essa política. Em suma, a nova implementação agora irá checar toda a lista de processos em busca daquele processo que possui a maior prioridade e, ao fim da busca, esse processo será o próximo a ser executado. Não foi utilizada nenhuma estrutura de dados customizada para facilitar essa checagem, como uma fila de prioridades. A busca pelo processo de maior prioridade é feita de forma linear na lista de processos, para facilitar a implementação.

3.2 Chamadas de sistema

Ao todo, quatro novas chamdas de sistema foram implementadas para facilitar o desenvolvimento do trabalho, e o propósito de cada uma delas está desrito a seguir.

3.2.1 change_prio()

Conforme especificado no enunciado, a chamada de sistema *change_prio()* deve ter utilizada para mudar a prioridade do processo atual. Sua implementação é simples, e basta trafegar por toda a lista de processos, encontrar o proesso com o identificador correto e mudarsua prioridade conforme o parâmetro desejado.

3.2.2 wait2()

Conforme também especificado no enunciado, a chamada de sistema *wait2()* deve ser utilizada como uma extensão da chamada de sistema *wait()* mas com algumas ações a mais. Em particular, ela deve também atribuir a três posições de memória os valores totais de tempo que o processo passou nos estados *READY*, *RUNNING* e *SLEEPING*, de modo que tais valores possam ser utilizados posteriormente na análise de dados das modificações propostas para o escalonador.

3.2.3 yield2()

A chamada de sustema *yield2()* se trata apenas de uma solução paliativa para que a função *yield()* seja utilizada nos programas do tipo S-CPU. Sem defini-lá, não era possível utiliar *yield()* em um programa de usuário pois não existia uma interface definida para seu uso em espaço de usuário no *kernel* do XV6.

Em suma, a chamada *yield2()* apenas chama a função *yield()*.

3.2.4 set_prio()

A chamada de sistema *set_prio()* deve ser utilizada para modificar a prioridade de um processo. Seu principal intuito é de ajudar nos testes do escalonador e checar se ele está funcionando da forma que deveria.

Da maneira que interpretamos, uma boa forma de implementar o *set_prio()* seria como uma extensão da chamada de sistema *fork()*, mas que permita a escolha dinâmica daprioridade do novo processo criado. Dessa forma, sua implementação foi feita exatamente como uma cópia de *fork()*, mas a prioridade *prio* dos processos criados a partir dela não é 2 por padrão, mas seguem a seguinte equação a partir de seus próprios identificadores *pid*:

$$prio = (pid \mod 3) + 1$$

Portanto, um processo com identificador 15 teria prioridade 1, e um processo com identificador 29 teria prioridae 3, por exemplo.

3.3 Programas implementados

Três tipos de programa deveriam ser criados para execuções no trabalho, sendo eles os programas *CPU-Bound*, *S-CPU* e *IO-Bound*, e a especificação de como cada um eles funciona está no enunciado.

Outros dois programas também foram implementados, sendo eles o progrma *sanity*, cujo objetivo é testar e analisar os tempos que cada processo passou em cada estado, e um programa extra, que nós denominamos *myprogram*, cujo objetivo principal é testar o correto funcionamento do novo escalonador. As especificações do programa *sanity* estão definidas no enunciado. Alguns detalhes relativos ao programa *myprogram*, no entanto, são detalhados abaixo.

3.3.1 myprogram.c

O programa chama, dentro de um *loop* de tamanho 20, a chamada de sistema *set_prio()*, que irá criar os novos processos e atribuir uma prioridade a eles com base no identificador que receberem. Após isso, o programa chama dentro de um laço *while* a chamada de sistema *wait()* e, sempre que um de seus processos filhos finaliza, imprime em tela seu identificador e sua prioridade inicial, isto é,prioridade estabelecida no moment da criação. O laço e o programa finalizam quando todos os processos filhos forem finalizados.

3.4 Mudança no intervalo de preempção

Para alterar o intervalo de preempção, bastou alterar uma linha do arquivo *trap.c* e adicionar um campo à estrutura *proc*. O novo campo, denominado *time_slice*, é sempre inicializado com o valor de `INTERV` quando um processo inicia sua execução na CPU, e contêm quantos *ticks* de *clock* cada processo ainda tem.

No arquivo *trap.c*, a preempção só ocorre quando o processo que está na CPU tem um valor de *time_slice* igual a 0, ou seja, já esgotou o tempo que tinha na CPU.

4 Análise de resultados

4.1 Tempo médio em cada estado

O tempo médio em cada estado é uma das formas de se compreender como cada um dos tipos de processo (CPU-Bound, S-CPU e IO-Bound) se comportam dentro do sistema. Para testar isso, utilizamos o programa *sanity*. Abaixo, podemos ver o resultado da execução do *sanity* para 30 processos, isto é, 10 processos de cada tipo.

Estado	CPU-Bound	S-CPU	IO-Bound
SLEEPING	8	8	446
READY	15	337	211
TURNAROUND	24	380	691

Table 1: Tempo médio em cada estado para n = 10

Podemos ver que os processos do tipo CPU-Bound são os de menor *turnaround*, o que faz todo sentido, uma vez que esses processos são dominados por tempo executando algo na CPU, o que fará com que eles terminem rapidamente. Faz todo sentido também que esse tipo de processo passe pouco tempo nos estados *SLEEPING* e *READY* quando comparados aos outros.

Os processos do tipo IO-Bound, por sua vez, são os que mais passaram tempo no estado *SLEEPING*, o que também é esperado, dado que nesse tipo de programa a chamada de sistema *sleep()* é feita constantemente, simulando as esperas por execuções de IO. Com isso, vale notar também que os processos desse tipo são os que possuem o maior tempo de *turnaround* disparadamente, uma vez que requisições de IO naturalmente são bem demoradas e fazem com que os processos que as utilizam demorem mais para serem finalizados a partir de sua primeira entrada na CPU.

Os programas do tipo S-CPU, também como imaginado, estão no meio do caminho entre os outros dois tipos no quesito tempo de *turnaround*. Nota-se, entretanto, que eles são os processos que mais passam tempo no estado *READY*. Isso condiz com o esperado dado que após executar a chamada de sistema *yield()* diversas vezes, esses processos abrem mão da CPU e voltam à um estado de espera com muita frequência.

Uma nova tabela foi também gerada,mas agora com um número maior de processos sendo criados, 50 de cada tipo, com o intuito de obter uma valida estatística ainda mais pecisa.

Estado	CPU-Bound	S-CPU	IO-Bound
SLEEPING	5	5	370
READY	18	345	178
TURNAROUND	23	365	563

Table 2: Tempo médio em cada estado para n = 50

Nota-se que os padrões observados anteriormente se mantêm para essa nova tablea, o que indica que o escalonador mantêm seu comportamente com o aumento do número de processos sendo executados.

Para entender ainda melhor como os diferentes tipos de processos funcionam, algumas constantes foram alteradas para testar como isso impactaria os tempos médios de cada tipo de processo em cada estado. Com essas variações podemos entender ainda melhor como a nova política do escalonador funciona e como cada tipo de processo é tratado pelos mecanismos implementados. Abaixo, temos os resultados obtidos para cada uma dessas variações, onde foram criados 10 processos de cada tipo.

4.1.1 Aumento do tempo de *sleep* em processos do tipo IO-Bound

Neste exemplo, o tempo de *sleep* foi aumentado de 1 para 5, com o intuito de compreender como isso impactaria o tempo de *turnaround* dos processo do tipo IO-Bound.

Estado	CPU-Bound	S-CPU	IO-Bound
SLEEPING	7	8	1028
READY	16	333	783
TURNAROUND	24	376	1974

Table 3: Tempo médio em cada estado para n = 10

Fica claro que o impacto é grande. O tempo de *turnaround* de processos do tipo IO-Bound aumentou muito, juntamente, de maneira óbvia, com os tempos nos estados *SLEEPING*e *READY*. Isso contribui ainda mais para a confirmação de que processos de tipo IO-Bound são grandes gargalos no sistema.

4.1.2 Aumento do número de laços em processos do tipo CPU-Bound

Aumentamos também o número de laços externos nos procesos de tipo CPU-Bound de 100 para 1 milhão,para entender se os novos resultdos indicariam alguma coisa. A nova tabela está disposta abaixo.

Estado	CPU-Bound	S-CPU	IO-Bound
SLEEPING	7	8	443
READY	15	329	214
TURNAROUND	24	372	691

Table 4: Tempo médio em cada estado para n = 10

Os resultados, no entanto,não variam muito dos obtidos com a quantidade original de laços. Isso indica que processos do tipo CPU-Bound são realmente muito rápidos e possuem baixo tempo de turnaround quando desconsideramos filas de prioridades.

4.2 Escalonamento

Para testar a nova política de escalonamento, conforme citado, foi utilizado o programa *myprogram*, que cria 20 processos do tipo CPU-Bound, mas suas prioridades são estabelecidas conforme seus identificadores. A tabela abaixo mostra a ordem de finalização dos processos criados com base nisso.

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

Table 5: Ordem de finalizaçao - Fila multinível

Podemos ver claramente que os processos foram finalizados na ordem correta, isto é,aquele com a maior prioridade (3) finalizaram primeiro e, da mesma maneira, aqueles com a menor prioridade (2) finalizaram por último. Isso traz uma boa inclinação de que o escalonador está funcionando de maneira correta. A título de comparação, temos abaixo uma tabela que foi gerada pelo mesmo código de *myprogram*, mas com o escalonador padrão do XV6.

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

Table 6: Ordem de finalização - Escalonador padrão

Podemos ver que nesse caso, como a prioridade é ignorada, já que o escalonador padrão do XV6 não leva ela em consideração, os processos são finalizados sem considerar elas. Em particular, podemos ver que nesse caso os processos finalizaram basicamente na ordme em que foram criados.

4.3 Tratamento de inanição

Com o uso de filas multinível, processos podem vir a sofrer de inanição no sistema, e por isso alguns mecanismos foram utilizados, conforme especificado no enunciado, para evitar esse problema. Em particular, utilizamos um processo de tratamento de inanição que aumenta a prioridade de um processo caso ele não tenha sido executado em muito tempo. Para testar isso, utilizamos uma variação do programa *sanity* que, ao invés de obter os tempos médios de cada tipo de processo em cada estado, obtêm a variação dasprioridades dos processos conforme eles são executados.

5 Conclusão

Em suma, o segundo trabalho prática da disciplina permitiu um aprendizado ainda mais aprofundado acerca de como um sistema operacional funciona, uma vez que abriu portas para explorar, modificar e analisar como um *kernel* realmente é implementado.

Fazer as modificações próprias para alterar as políticas de escalonamento e preempção tornaram a compreensão de como eles operam mais simples, e ampliaram a visão de como um processo realmente é manipulado dentro do sistema.

6 Referências

- Livros:
  - Tanenbaum, A. S. & Bos, H. (2014), Modern Operating Systems, Pearson, Boston, MA.
  - Abraham Silberschatz, Peter Baer Galvin, Greg Gagne: Operating System Concepts, 10th Edition. Wiley 2018, ISBN 978-1-118-06333-0
  - Arpaci-Dusseau, Remzi H., Arpaci-Dusseau, Andrea C.. (2014). Operating systems: three easy pieces.: Arpaci-Dusseau Books.

- Web:
  - *xv6: a simple, Unix-like teaching operating system*
- Youtube:
  - Jacob Sorber
  - Code Vault
  - hhp3 xv6 *kernel playlist*