Relatório do Trabalho Prático 02

Sistemas Operacionais

Thiago Martin Poppe, Giovanna Louzi Bellonia

11/10/2020

1 Uma breve descrição

1.1 Problema proposto

Este trabalho prático foi dividido em duas partes. A primeira consiste em implementar um escalonamento de filas multinível na arquitetura xv6, onde cada fila terá um escalonador Round-Robin com quantum igual à 5 ticks de clock; já a segunda, consiste em realizar uma série de testes computando a média dos tempos de espera, execução, pronto e turnaround para cada processo criado através da chamada de sistema **fork**.

1.2 Observações

- O desenvolvimento de ambas as partes foi feito utilizando o WSL. Posteriormente, os mesmos códigos foram testados em uma máquina virtual Linux com distribuição Ubuntu 20.04 LTS.
- Foi utilizado o manual do próprio xv6 para obter mais conhecimento sobre o sistema, além de outros links e programas disponibilizados pelo monitor da disciplina.

2 Parte 1 - Escalonamento de filas multinível

2.1 Sistema operacional xv6

Essa seção será reservada para falarmos um pouco mais sobre a arquitetura que iremos modificar.

Originalmente, o xv6 utiliza uma política de escalonamento baseada em um Round-Robin com valor de quantum igual à 1 tick de clock. Por ser um escalonador simples de ser implementado, e livre de inanição, é uma escolha perfeita para esse sistema que foi projetado para ser didático, acima de tudo.

Essa política de escalonamento realiza a preempção de processos à cada intervalo, chamado de quantum, definido pelo S.O. Por esse motivo, o mesmo, como dito anteriormente, é livre de inanição, i.e quando processos nunca executam devido à outros que geralmente possuem

prioridade de execução maior. Em outras palavras, para esse escalonador, todos os processos possuem a mesma prioridade e compartilham, de forma igualitária, o acesso à CPU.

2.2 Conceito de filas multinível e prioridade

Um escalonamento baseado em filas multinível consiste em criarmos N filas com prioridades distintas. Para esse trabalho prático, filas com maior valor representavam maior prioridade. Além disso, cada fila possui sua própria política de escalonamento para os processos contidos na mesma, que no nosso caso será um Round-Robin com quantum igual à 5, como dito anteriormente.

Nessa política, devemos sempre dar prioridade para aquela fila que possui maior prioridade definida pelo S.O. No caso do trabalho prático, teremos 3 filas de prioridade, sendo elas para processos CPU-Bound, processos fortes em processamento - prioridade máxima; SCPU-Bound, pequenas tarefas fortes em processamento - prioridade mediana; e I/O-Bound, processos fortes em entrada/saída - prioridade mínima. Sendo assim, apenas pegaremos processos de uma fila F se todas as filas com prioridade superior a F estiverem vazias, ou sem processos em estado pronto.

Diferentemente do Round-Robin, essa política trás consigo a desvantagem de causar inanição no sistema. Em outras palavras, podemos ter o caso onde processos de filas com menor prioridade nunca sejam executados, pois as filas de maior prioridade nunca estarão vazias. Iremos deixar uma seção própria para explicar mais sobre a solução construída para esse problema.

2.3 Mudando o intervalo de preempção

Dentro do arquivo param.h, temos a definição de várias constantes que o sistema operacional irá utilizar, como por exemplo: o número máximo de processos que poderão ser executados em paralelo, que é representado por NPROC. Nesse arquivo, definimos uma constante chamada INTERV que assumirá valor 5. Usaremos essa constante para mudar o intervalo de preempção do Round-Robin.

A preempção dos processos é feita dentro da função trap pertencente ao trap.c. Apenas adicionamos uma condição a mais nessa função para se realizar a chamada de sistema **yield**, que é responsável por retirar o processo da CPU.

Originalmente, adicionamos um campo na estrutura proc, presente em proc.h, que servia como um "countdown" para sabermos quanto tempo o processo estava na CPU. Posteriormente, com a implementação do campo "rutime", optamos por mudar essa ideia, apenas verificando se $proc.rutime \equiv 0 \pmod{INTERV}$, o que indica que se passaram "INTERV ticks" desde sua entrada na CPU.

2.4 Implementação de filas multinível

Optamos por implementar, de forma simples, uma fila de tamanho fixo que possui operações de inserção, remoção, criação e "busca". A complexidade de cada operação é dada a seguir:

	Melhor caso	Pior caso
Criação	$O(n)$	$O(n)$
Inserção	$O(1)$	$O(1)$
Remoção	$O(1)$	$O(n)$
Busca	$O(1)$	$O(n)$

Dentro da nossa estrutura queue_t mantivemos um vetor de ponteiros, de tamanho NPROC, para processos que pertencem àquela fila de prioridades. Durante a remoção, optamos por apenas realizar um "shift à esquerda" dos valores começando do processo que queremos remover da fila. A inserção sempre ocorre no final da mesma, sendo assim, processos que chegam primeiro são os primeiros a serem escalonados pelo Round-Robin.

A operação de busca consiste em apenas retornar um valor True (1) caso tenha processos em estado pronto em uma determinada fila e False (0) caso contrário. Com essa operação, conseguimos, de forma simples, aliar o escalonador Round-Robin já implementado pelo xv6 com essa nova política de escalonamento baseada em filas multinível, de forma que dado um processo P, verificamos em qual fila ele pertence. Se, por exemplo, ele pertencer à fila de prioridade 1, ele apenas será "despachado" para a CPU se a fila de prioridade 2 não conter nenhum processo em estado pronto; caso contrário, passamos para o próximo processo pertencente à tabela de processos já implementada pelo xv6. Note que o processo P também deve estar em estado PRONTO para ser "despachado" para a CPU.

A implementação dessa estrutura de dados foi feita diretamente no arquivo proc.c para ser mais simples de ser chamada dentro da função scheduler, que executa a lógica do escalonamento.

2.5 Implementação de envelhecimento

Para lidar com o problema de inanição, implementamos um envelhecimento (aging) nos processos com estado PRONTO. O envelhecimento consiste em basicamente aumentar a prioridade de um processo baseada no tempo que o mesmo esteve esperando para ser executado. Para isso, no arquivo param.h definimos duas constantes de envelhecimento, sendo elas: P0TO1 que fala quanto tempo um processo deve esperar para ser transferido da fila de prioridade 0 para 1; e P1TO2 que se baseia na mesma lógica, porém da fila de prioridade 1 para 2.

Aliado a essas duas constantes, dentro da estrutura proc adicionamos um campo chamado ticks, que guarda a informação de quantos ticks de clock um processo ficou em estado PRONTO. Sempre que esse valor ultrapassar uma das constantes, realizamos o processo de remoção de uma fila de prioridade mais baixa e inserimos em uma de prioridade mais alta, resetando o valor de ticks e alterando a prioridade do processo.

3 Parte 2 - Testes

3.1 Arquivo sanity.c

Para a implementação dos testes, foi criado um arquivo chamado sanity.c que recebe um parâmetro **n** indicando o número de vezes que chamaremos a chamada de sistema **fork**,

já presente no xv6. Para cada processo terminado iremos printar seu pid, tipo e métricas calculadas. Esse arquivo irá realizar 5 vezes **n** chamadas **fork**, computando no final de tudo a média dos tempos de sleeping, pronto, executando e turnaround para cada tipo de processo.

3.2 Obtendo as métricas necessárias

Para guardar as métricas a serem computadas, criamos 4 campos na estrutura proc, sendo eles: ctime, que indica o tempo em ticks de clock que o processo foi criado; retime, que indica o tempo total em estado pronto; rutime, que indica o tempo total em estado executando; e stime, que indica o tempo total em estado sleeping, que iremos utilizar para simular I/O.

A atualização desses campos se dá através da implementação da chamada de sistema **update_metrics**, onde, a cada tick de clock, a chamada irá percorrer por todos os processos atualizando seus campos conforme seu estado atual. Essa chamada é feita na função trap dentro do arquivo trap.c logo antes de atualizarmos o número de ticks de clock global.

Para recuperar tais métricas no arquivo sanity.c, implementamos a chamada de sistema **wait2**, que estende a chamada **wait** recebendo como argumentos um ponteiro para o valor de retime, rutime e stime. Dentro dessa chamada, ao encontramos o processo pai da chamada **fork**, passamos para esses ponteiros os valores armazenados nesses campos e logo em seguida retiramos o processo filho da sua fila de prioridades, visto que o mesmo irá voltar a ter seu estado "setado" para UNUSED, ou seja, como se não existisse mais.

3.3 Tipos de processos

Como dito anteriormente, teremos processos de 3 tipos distintos, sendo eles: CPU-Bound, SCPU-Bound e I/O-Bound. Para determinar seu tipo, realizamos o seguinte cálculo:

- Se $pid \equiv 0 \pmod 3$, o processo será do tipo CPU-Bound.
- Se $pid \equiv 1 \pmod 3$, o processo será do tipo SCPU-Bound.
- Se $pid \equiv 2 \pmod 3$, o processo será do tipo I/O-Bound.

Após conhecermos o tipo do processo, iremos atribuir as prioridades 0 para o tipo I/O-Bound, 1 para o tipo SCPU-Bound e 2 para o tipo CPU-Bound através da chamada de sistema **setprio** implementada por nós.

3.4 Atribuição de prioridade

Criamos uma chamada de sistema **setprio** que permite com que um programa em modo de usuário chame o sistema, solicitando que a prioridade do processo atual seja modificada para a prioridade passada como parâmetro da chamada.

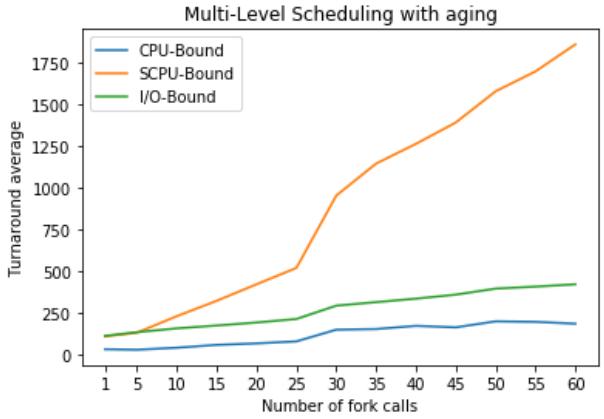
Tendo seu corpo implementado no arquivo proc.c, a chamada de sistema **setprio** além de modificar a prioridade do processo atual também realiza a mudança entre as filas de prioridade para não tornar a política de escalonamento incoerente. Caso o usuário passe uma prioridade inválida, no caso menor que 0 ou maior que 2, a chamada irá lançar uma interrupção para o S.O, alertando o mesmo sobre esse erro cometido pelo usuário, que será mostrado através da função **panic** já implementada pelo xv6.

3.5 Resultados observados

No total, fizemos 5 experimentos utilizando o arquivo sanity.c. Para cada experimento, coletamos os dados e utilizamos a linguagem Python para plotar gráficos, facilitando assim a análise dos resultados. Para tornar os gráficos menos poluídos, apenas observamos a métrica de turnaround que é igual a rutime + retime + stime. Para os experimentos que não envolviam a variação de **n**, fixamos o mesmo em 20.

3.5.1 Experimento 01

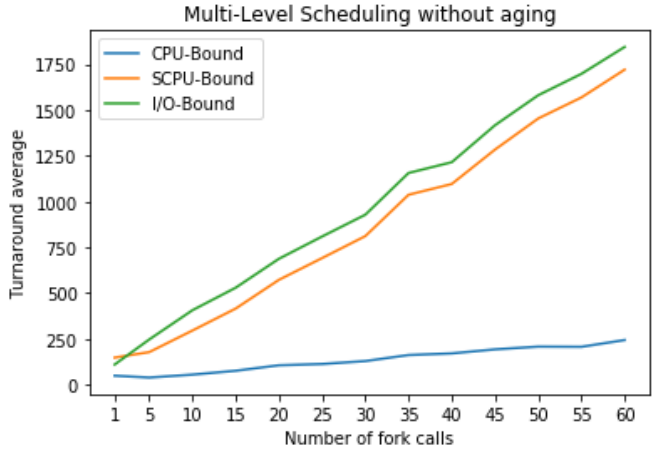
- O que ocorre quando variamos o valor de **n** utilizando aging?



Aqui podemos observar que com o aumento de **n** teremos um aumento quase linear do tempo médio de turnaround para os processos SCPU-Bound e menor que linear para os demais tipos.

3.5.2 Experimento 02

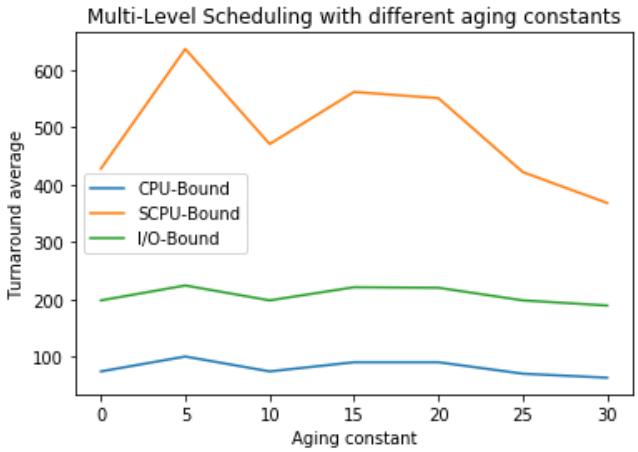
- O que ocorre quando variamos o valor de **n** não utilizando aging?



Aqui podemos observar claramente que executamos primeiro todos os processos do tipo SCPU-Bound e depois os do tipo I/O-Bound, ilustrando de forma gráfica o processo de inanição dos processos que possuem menor prioridade.

3.5.3 Experimento 03

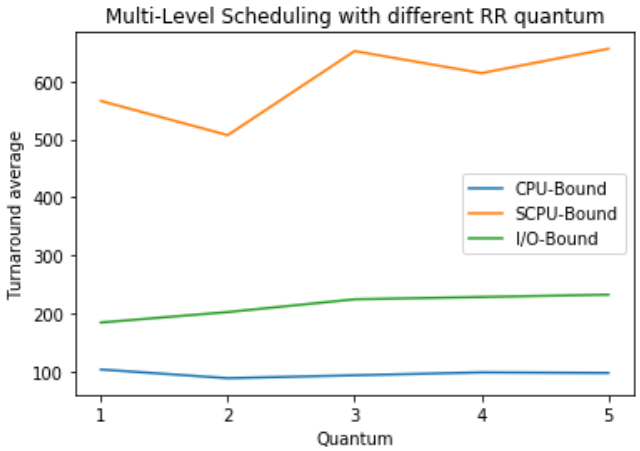
- O que ocorre quando variamos as constantes de aging? (Por simplificação P0TO1 = P1TO2)



Esse experimento consistia em encontrar um valor "ótimo" para as constantes de envelhecimento usadas na política de escalonamento. Observamos que o valor 30 foi o que mais minimizou o turnaround médio.

3.5.4 Experimento 04

- O que ocorre quando alteramos o tempo de preempção do Round-Robin utilizando a política de filas multinível?

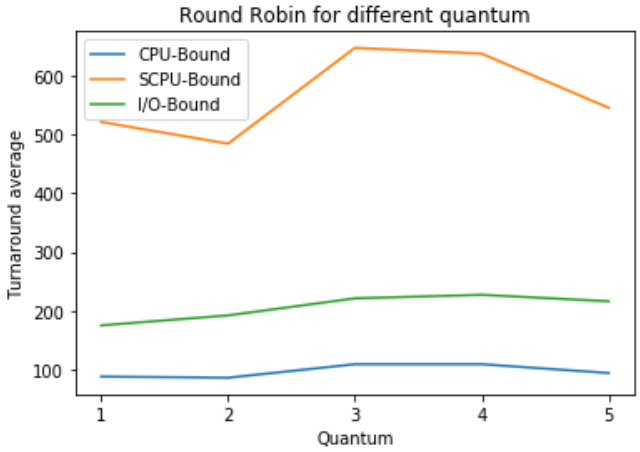


Esse experimento consistia em encontrar um valor "ótimo" para o intervalo de preempção do escalonador Round-Robin quando utilizamos filas multinível. Observamos que o valor 2 foi o que mais minimizou o turnaround médio. Note que há uma queda entre 1 e 2 devido à diminuição do overhead associado com a troca de contexto.

Obs.: Utilizamos P0TO1 = 10 e P1TO2 = 5, que foram os valores colocados inicialmente antes do experimento 03.

3.5.5 Experimento 05

- O que ocorre quando alteramos o tempo de preempção do Round-Robin sem a política de filas multinível?



Esse experimento consistia em encontrar um valor "ótimo" para o intervalo de preempção do escalonador Round-Robin original do xv6, além de servir como uma comparação para a política implementada no trabalho. Observamos que o valor 2 foi o que mais minimizou o turnaround médio, atingindo um valor menor do que aquele obtido no experimento 04.