

UNIVERSIDADE FEDERAL DE MINAS GERAIS

Instituto de Ciências Exatas

Graduação em Sistemas de Informação

Gabriel Oliveira Sant'Ana

Mariana Assis Ramos

Trabalho Prático #1 – Escalonador de Processos

Belo Horizonte

2024

1. Introdução

O xv6 é um sistema operacional educacional baseado na 6ª versão do Unix(V6), desenvolvido pelo MIT. Projetado com uma estrutura modular, ele serve como uma ferramenta fundamental no ensino dos princípios fundamentais de sistemas operacionais de forma acessível e compreensível. Um dos componentes-chave de todo sistema operacional é o escalonador de processos, que determina quais processos serão executados e por quanto tempo. No caso do xv6, ele adota a política de escalonamento round-robin (RR), na qual os processos são executados em ordem circular, garantindo uma distribuição justa do tempo de CPU entre eles evitando a inanição.

Este trabalho tem como objetivo aplicar modificações no escalonador do xv6, explorando diferentes políticas de escalonamento de processos. A principal mudança consiste na implementação de uma fila multinível de prioridade, na qual cada um dos quatro níveis possui uma política de escalonamento específica: round-robin (RR), loteria, shortest job first(SJF) e first come first served (FCFS). A entrega também conta com outras modificações, como funções auxiliares e mecanismos para prevenir inanição, e uma série de testes comparando as diferentes políticas propostas e juntamente com a versão original do xv6, com o intuito de avaliar o desempenho e a eficácia de cada abordagem.

2. Implementações

2.1. INTERV

A primeira modificação feita no xv6 envolveu a compreensão da política de escalonamento atual para possibilitar sua alteração. Por padrão, o xv6 utiliza a política de round-robin (RR), que será discutida em mais detalhes na seção subsequente (2.2. *Políticas Utilizadas*). A alteração proposta consistia em ajustar o intervalo de preempção dessa política, que, por padrão, é de um tick do relógio.

Para implementar essa alteração, introduzimos uma constante chamada **INTERV**, que determina o valor do novo intervalo de preempção. Além disso, adicionamos um novo atributo na estrutura **proc** do arquivo *proc.h*, chamado **clock**. Inicializado como 0, este atributo conta por quantos ticks um processo está em execução, sendo incrementado em 1 a cada tick que o processo está rodando (em

estado de RUNNING). Utilizamos uma função auxiliar chamada *updateClock()* para realizar esse aumento.

A função *updateClock()* é invocada a cada tick do relógio pelo *trap.c* e atualiza não apenas o clock, mas também outras métricas de todos os processos. Com a presença do clock em cada processo, calculando o tempo do quantum, adicionamos uma condição adicional no *trap.c* que controla a preempção dos processos. Assim, a preempção ocorre somente quando o valor do **clock** é igual ao definido no **INTERV**.

2.2. Escalonador de Filas Multinível

A próxima modificação requerida foi a implementação de uma fila multinível com 4 filas de prioridade, na qual todos os processos são inicializados por padrão com prioridade 2. O escalonamento por fila multinível ocorre quando a fila de prontos é dividida em diferentes filas, cada uma com seu nível de prioridade. Os processos de maior prioridade são colocados em filas com maior nível e os de menor prioridade nas filas de menor nível. As prioridades podem ser definidas com base em características específicas, importância ou, como é o caso desta tarefa, uma prioridade padrão para todos os processos.

O xv6 padrão opera com uma pseudo-fila de prioridade, contendo um array com todos os processos do sistema armazenado na estrutura **ptable**. Cada processo possui diferentes estados e o escalonador padrão percorre todo o array em busca dos processos no estado de pronto (RUNNABLE) para escaloná-los.

Para implementar as 4 filas de prontos, adotamos uma abordagem em que teríamos efetivamente 4 filas de prontos. Criamos 4 filas em formato de matriz dentro da estrutura **ptable**, contendo ponteiros para os processos chamada **queue_ready**. Essas serão nossas filas de prontos para cada um dos níveis de prioridade. Para facilitar a manipulação de cada uma dessas filas, também introduzimos o array **count_queue**, que armazena a quantidade de processos em cada uma delas. Essa implementação proporciona uma estrutura fácil de entender e trabalhar. Além disso, ao utilizar ponteiros, economizamos memória e conseguimos alterar diretamente os processos a que apontamos.

A lógica de funcionamento é simples: sempre que um processo tem seu estado alterado para pronto (RUNNABLE), ele é adicionado à fila de prioridade

correspondente, e a contagem dessa fila é incrementada. Da mesma forma, quando um processo deixa o estado de execução (RUNNING), ou seja, é escalonado, ele é removido da fila e a contagem é decrementada. Garantindo que dentro de cada fila tenha somente os processos no estado de pronto (RUNNABLE)

2.3. Políticas usadas

2.3.1 - Round Robin

A política de escalonamento Round Robin é a política padrão utilizada pelo xv6 originalmente. Para implementar essa política na fila de prioridade escolhida, mantivemos a implementação antiga e a utilizamos exclusivamente para essa fila. De forma geral, o algoritmo Round Robin é preemptivo, o que significa que cada processo recebe a CPU por um período de tempo predefinido, conhecido como quantum, determinado pela constante INTERV. Após esse período, o processo é interrompido e volta para sua fila de prontos, aguardando sua próxima vez de execução. Isso ocorre de maneira cíclica, garantindo que todos os processos tenham a oportunidade de utilizar a CPU e evitando problemas de starvation. No entanto, esse algoritmo gera um overhead significativo de troca de contexto devido à frequente interrupção dos processos. Portanto, é crucial analisar cuidadosamente o valor do quantum. Se for muito alto, os processos podem ser executados por períodos prolongados, semelhante ao algoritmo FCFS, o que pode resultar em longos tempos de espera para processos no final da fila. Por outro lado, se o quantum for muito baixo, o overhead de troca de contexto aumenta, tornando o algoritmo menos eficiente. Na seção de TESTES, exploraremos diferentes valores do INTERV para avaliar os resultados obtidos com o algoritmo Round Robin.

Além disso, escolhemos esse algoritmo para ser executado na fila de prioridade 2, onde a maioria dos processos é alocada, devido à sua capacidade de oferecer um equilíbrio entre resposta rápida e justiça na distribuição de tempo de CPU entre os processos. Como essa fila é destinada a processos de prioridade padrão, é essencial garantir que todos os processos recebam oportunidades equitativas de execução, evitando assim problemas de starvation e garantindo um bom desempenho geral do sistema.

2.3.2 - First-Come, First-Served

O algoritmo First Come First Served (FCFS) foi escolhido para ser implementado na fila de prioridade 4, a de mais alta prioridade. O FCFS é um algoritmo simples que prioriza os processos com base na ordem de chegada, permitindo que o primeiro processo a entrar na fila seja o primeiro a ser executado. Uma vez que um processo inicia sua execução, ele continuará sua execução até ser concluído, sem ser interrompido por outros processos. Essa abordagem é adequada para a fila de maior prioridade, pois garante que os processos críticos ou urgentes sejam atendidos imediatamente, sem a necessidade de aguardar ou ser interrompido por outros processos. No entanto, é importante observar que o FCFS pode gerar um grande tempo de espera para os processos que estão no final da fila, especialmente se houver muitos processos na fila de prioridade 4.

2.3.3 - Lottery

Para a fila de menor prioridade, optou-se pelo algoritmo de loteria. Neste algoritmo, cada processo é atribuído uma quantidade de tickets e um ticket é escolhido aleatoriamente para selecionar o próximo processo a ser executado. Para implementar isso, foi adicionado um atributo chamado tickets na estrutura proc, representando a quantidade de tickets que cada processo possui, sendo inicializado como 10.

No escalonador, a função `loteria_total` é chamada para somar a quantidade total de tickets de todos os processos na fila. Em seguida, é utilizada a função `randomInRange`, que foi implementada para gerar um número aleatório dentro de um intervalo especificado, para selecionar um número que será o ticket escolhido, chamado de `golden_ticket`. Em seguida, todos os processos na fila são percorridos até que seja encontrado o processo que possui o `golden_ticket`, que será então selecionado para execução.

Essa abordagem de loteria garante uma distribuição aleatória de tempo de CPU entre os processos na fila de menor prioridade, proporcionando uma forma equitativa de acesso aos recursos do sistema. Isso pode ser útil para garantir que todos os processos recebam oportunidades justas de execução, mesmo em uma fila de baixa prioridade.

2.3.4 - Shortest Job First

Para a fila de prioridade 3, foi implementado o algoritmo de shortest job first (SJF). Nesse algoritmo, é essencial calcular e prever o tempo estimado de burst de cada processo para selecionar aquele com o menor tempo de burst esperado. Para isso, foram introduzidos os atributos `estimatedburst` e `previousburst` na estrutura `proc`, juntamente com o atributo `bstime`, usado para calcular o burst durante a execução do processo. Durante a execução de um processo, o atributo `bstime` é incrementado a cada ciclo de clock, representando o tempo de CPU utilizado pelo processo. Quando o processo é interrompido ou concluído, o `bstime` é redefinido para zero.

Foi criada uma função `updateBurst` que é responsável por calcular o `estimatedburst` de um processo após cada burst de CPU, utilizando uma média exponencial que leva em consideração tanto o último burst calculado quanto os bursts anteriores. Isso garante uma estimativa mais precisa do tempo de burst esperado.

No escalonador, é chamada a função `shortestBurst` que percorre todos os processos na fila selecionada e compara seus `estimatedburst`, que são sempre atualizados pela função `updateBurst`. O processo com o menor `estimatedburst` é selecionado para execução.

Essa abordagem assegura que os processos com tempos de burst mais curtos sejam priorizados para execução, otimizando o tempo de resposta e minimizando o tempo de espera médio na fila de prioridade 3. Ao utilizar o algoritmo SJF nessa fila, espera-se uma melhor eficiência em termos de utilização de CPU e tempo de resposta para os processos.

2.4. Starvation e Aging

No escalonamento por fila multinível, a possibilidade de starvation é uma desvantagem conhecida, ocorrendo quando um processo aguarda por muito tempo no estado de pronto sem ser escalonado. Para evitar esse problema, introduzimos um sistema de envelhecimento (aging) no qual adicionamos um novo atributo para cada processo no `proc.h`, chamado **readyTimeAging**. Esse atributo registra quanto tempo cada processo permanece no estado de pronto sem ser escalonado. O valor é zerado sempre que o processo é escalonado e incrementado a cada tick enquanto

o processo está no estado de pronto (RUNNABLE), utilizando a função auxiliar *updateClock()* para atualizar as estatísticas de todos os processos.

Em seguida, implementamos a função *upgradePriority_Aging()*, que verifica se algum processo nas três filas de prioridade mais baixa está em estado de starvation. Caso positivo, o processo recebe um upgrade na prioridade e é movido para a fila de prioridade mais alta seguinte, sendo retirado da fila atual. O estado de starvation é definido com base em três constantes do *param.h*, "**_1TO2**", "**_2TO3**" e "**_3TO4**", que indicam o tempo máximo que um processo pode permanecer em cada fila de pronto sem ser escalonado. Quando o **readyTimeAging** atinge o valor definido pela constante referente a fila relacionada ao processo, ele recebe o upgrade de prioridade.

Para garantir uma troca de filas robusta e eficiente, a remoção do processo da fila atual e a adição na nova fila de prioridade, bem como a atualização do contador de processos de cada fila, são realizadas pela função auxiliar *auxTrocarFilaPriority()*.

2.5 - Change_prio

Construímos uma system call chamada *change_prio()* que permite a alteração manual da prioridade de um processo. Essa system call recebe como argumento a nova prioridade desejada e realiza a troca da prioridade do processo. Logo em seguida, é executado um yield, garantindo que o processo seja reescalonado na nova prioridade.

3. Testes

Para analisar as mudanças realizadas, conduzimos uma série de testes detalhados abaixo. Para facilitar esses testes, implementamos a função *wait2()*, uma system call que retorna atributos cruciais do processo, como o tempo de sleep, running e pronto. Além disso, implementamos também a *change_prio()*, outra system call que possibilita a mudança manual da prioridade do processo corrente, ajustando também sua fila de escalonamento. Essas ferramentas foram essenciais para facilitar a execução e a análise dos testes.

Os testes foram conduzidos com o auxílio de dois arquivos principais: o *sanity.c* e o *tp_teste.c*.

O *sanity.c* foi projetado para receber um número 'n' de processos e criar três vezes 'n' processos. Esses processos replicados incluem um de cada um dos tipos: CPU, S-CPU e I/O. Os processos CPU executam continuamente sem interrupções de IO, os S-CPU realizam tarefas de CPU mais curtas e os I/O são interativos. O *sanity.c* utiliza a função *wait2()* para calcular a duração de cada um dos processos criados e, ao final, retorna a média de ready, running, sleep e turnaround time.

Por outro lado, o *tp_teste.c* é um arquivo dinâmico que é constantemente modificado para diferentes situações de teste. Utilizamos a *change_prio()* para inicializar processos em filas específicas e a *wait2()* para obter estatísticas detalhadas durante os testes.

3.1. Tempo de Interv

Como mencionado, o valor do INTERV é usado para modificar e determinar o quantum de um processo preemptivo, afetando diretamente o desempenho da política de escalonamento preemptiva, o Round-Robin (RR). Para investigar o impacto de diferentes valores de INTERV, configuramos o código para que todos os processos sejam da prioridade da fila do RR e desativamos o mecanismo de aging, garantindo que fossem escalonados exclusivamente com o RR. Em seguida, realizamos testes utilizando o arquivo *sanity.c*, que nos permitiu calcular os tempos de execução para três tipos de processos (CPU-bound, S-CPU-bound e IO-bound). Rodamos cinco execuções para cada um dos valores de INTERV: 1, 5, 10 e 15. Calculamos a média dessas cinco execuções para obter uma estimativa mais precisa dos tempos de execução para cada processo com cada valor de INTERV.

política de escalonamento	tipo de processo	tempos			
		READY	RUNNING	SLEEP	TURNAROUND
Round-Robin: INTERV 1	CPU	451	45	0	496
	S-CPU	1109	9	0	1118
	IO	1139	0	100	1239
Round-Robin: INTERV 5	CPU	435	45	0	480
	S-CPU	1108	9	0	1117
	IO	1143	0	100	1243
Round-Robin: INTERV 10	CPU	428	43	0	471
	S-CPU	1073	9	0	1082
	IO	1098	0	100	1198
Round-Robin: INTERV 15	CPU	475	46	0	521
	S-CPU	1137	9	0	1146
	IO	1163	0	100	1263

figura 1: tabela com diferença de tempo de escalonamento dado diferentes valores de INTERV

Ao analisar os resultados obtidos, observamos que os tempos de execução são bastante semelhantes, especialmente em relação ao tempo de execução em estado de RUNNING. No entanto, as diferenças significativas ocorrem nos tempos em estado de pronto, que influencia diretamente o TURNAROUND time. Ao buscar o valor ótimo para o INTERV visando o menor TURNAROUND, identificamos que esse valor gira em torno de 10. Isso se deve a dois motivos principais: quando o INTERV é muito alto (a partir de 15), o algoritmo RR se assemelha ao First-Come-First-Served, no qual não ocorre preempção e o TURNAROUND time não é tão eficiente. Por outro lado, quando o INTERV é muito baixo, a preempção é muito frequente, resultando em mais trocas de contexto e maior tempo de espera na fila de pronto. Portanto, com base nos testes realizados, identificamos o valor ótimo para o INTERV, visando minimizar o TURNAROUND time, como sendo 10.

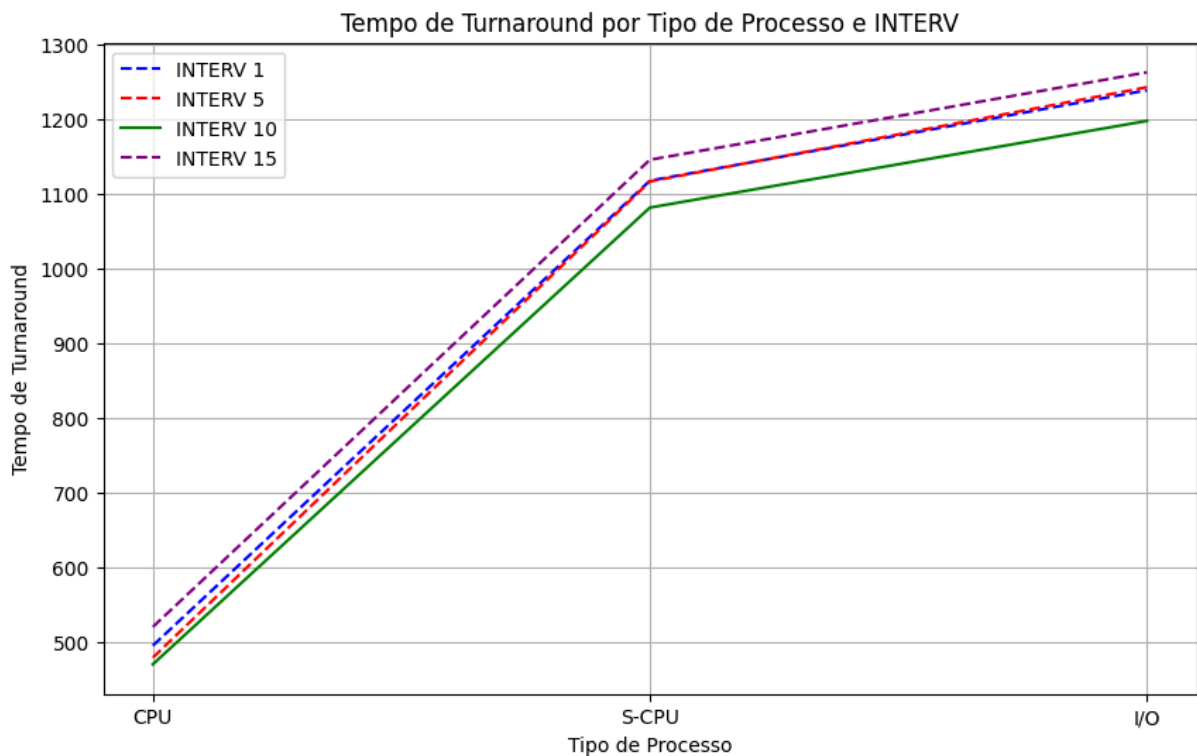
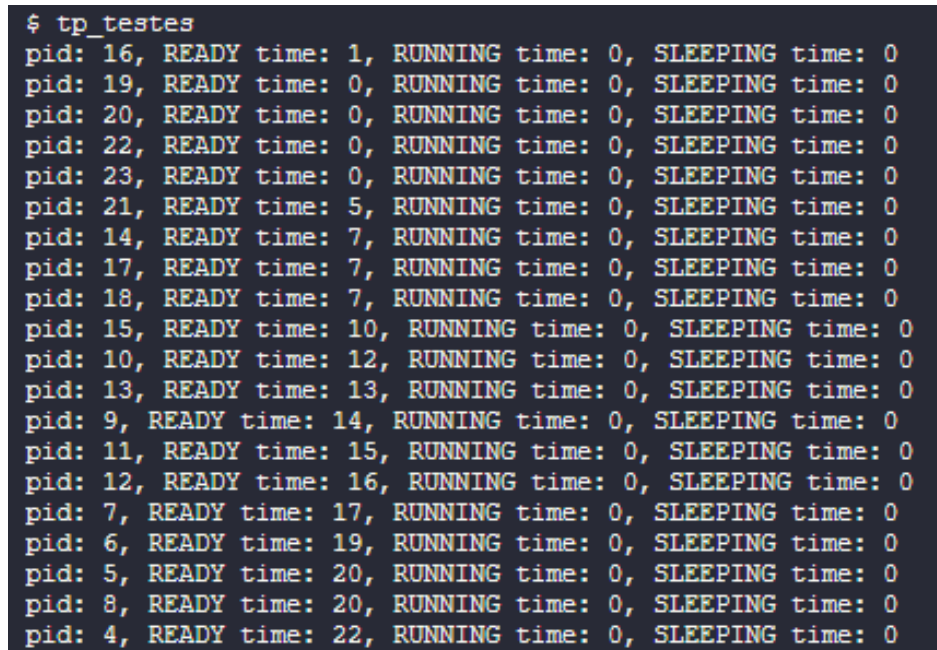


figura 2: gráfico de linhas para o TURNAROUND time dado um INTERV

3.2. Ordem de Prioridade

Para verificar se o algoritmo de escalonamento entre as filas multiníveis está obedecendo a ordem de prioridades, realizamos um teste simples que envolve a criação de 20 processos. Estes são distribuídos em quatro grupos de prioridade, sendo os cinco primeiros de prioridade 1 (a menor), seguidos por cinco processos de

prioridade 2 e assim por diante, até cinco processos de prioridade 4, criados exatamente nessa ordem. O objetivo do teste é confirmar que, mesmo que um processo tenha sido criado posteriormente, se possuir uma prioridade maior, será escalonado primeiro.

A terminal window with a dark background and light-colored text. The prompt is '\$ tp_testes'. Below it, 20 lines of process status information are listed, each on a new line. Each line contains a process ID (pid), its state (READY, RUNNING, or SLEEPING), and its time values. The processes are ordered by their READY time, which increases from 1 to 22. The states are: pid 16 is RUNNING; pid 19, 20, 22, 23, 21, 14, 17, 18, 15, 10, 13, 9, 11, 12, 7, 6, 5, 8, and 4 are all in the READY state.

```
$ tp_testes
pid: 16, READY time: 1, RUNNING time: 0, SLEEPING time: 0
pid: 19, READY time: 0, RUNNING time: 0, SLEEPING time: 0
pid: 20, READY time: 0, RUNNING time: 0, SLEEPING time: 0
pid: 22, READY time: 0, RUNNING time: 0, SLEEPING time: 0
pid: 23, READY time: 0, RUNNING time: 0, SLEEPING time: 0
pid: 21, READY time: 5, RUNNING time: 0, SLEEPING time: 0
pid: 14, READY time: 7, RUNNING time: 0, SLEEPING time: 0
pid: 17, READY time: 7, RUNNING time: 0, SLEEPING time: 0
pid: 18, READY time: 7, RUNNING time: 0, SLEEPING time: 0
pid: 15, READY time: 10, RUNNING time: 0, SLEEPING time: 0
pid: 10, READY time: 12, RUNNING time: 0, SLEEPING time: 0
pid: 13, READY time: 13, RUNNING time: 0, SLEEPING time: 0
pid: 9, READY time: 14, RUNNING time: 0, SLEEPING time: 0
pid: 11, READY time: 15, RUNNING time: 0, SLEEPING time: 0
pid: 12, READY time: 16, RUNNING time: 0, SLEEPING time: 0
pid: 7, READY time: 17, RUNNING time: 0, SLEEPING time: 0
pid: 6, READY time: 19, RUNNING time: 0, SLEEPING time: 0
pid: 5, READY time: 20, RUNNING time: 0, SLEEPING time: 0
pid: 8, READY time: 20, RUNNING time: 0, SLEEPING time: 0
pid: 4, READY time: 22, RUNNING time: 0, SLEEPING time: 0
```

figura 3: terminal do xv6 mostrando a ordem de escalonamento dos processos

Ao executar este teste, observamos que os processos 4 ao 8 têm prioridade 1, os processos 9 ao 13 têm prioridade 2, os processos 14 ao 18 têm prioridade 3 e os processos 19 ao 23 têm a maior prioridade (4). Embora o processo 16 tenha sido o primeiro a ser escalonado, isso somente ocorreu porque não houve a criação dos de prioridade mais alta ainda, de forma que o fluxo segue conforme esperado: os processos de prioridade mais alta são finalizados antes dos de prioridade mais baixa, mesmo que tenham sido criados anteriormente. Isso confirma a precisão na sequência da ordem de nossa fila de prioridade.

3.3. Efeito de Aging

Para os testes sobre o efeito de aging, foi proposta uma análise da ordem de execução dos processos à medida que as constantes de aging são ajustadas. É crucial destacar que encontrar o tempo ideal para o aging é desafiador. Por um lado, ele não pode ser excessivamente curto, caso contrário, não resolveria a inanição quando muitos processos estão chegando. Por outro lado, não pode ser muito

rápido, pois os processos acabariam alcançando a mesma prioridade rapidamente e perderia o sentido a fila de prioridade.

Diante dessa complexidade, realizamos o seguinte teste: criamos 150 processos usando uma versão modificada do *sanity.c*. Inicializamos um *sanity* com 50 processos, que foi então expandido para gerar uma variedade de tipos de processos (CPU, S-CPU e IO). Entre esses 50 processos, atribuímos manualmente prioridades, com os 20 primeiros sendo designados para a fila de maior prioridade, outros 20 para a fila de prioridade 3 e os 10 restantes para a fila de menor prioridade (1). O objetivo é encontrar o tempo ideal para as constantes de aging de cada fila, de modo que os processos da fila 1 ascendam para as filas subsequentes sem serem escalonados antes dos processos de prioridade máxima, enquanto mantêm um TURNAROUND time satisfatório.

valores constante aging	tipo de processo	tempos			
		READY	RUNNING	SLEEP	TURNAROUND
1TO2 = 80 2TO3 = 60 3TO4 = 40	CPU	1386	55	0	1441
	S-CPU	3415	10	0	3426
	IO	3541	0	100	3641
1TO2 = 200 2TO3 = 150 3TO4 = 100	CPU	1506	58	0	1564
	S-CPU	3551	11	0	3562
	IO	3705	0	100	3805
1TO2 = 600 2TO3 = 500 3TO4 = 400	CPU	1265	43	0	1308
	S-CPU	2325	9	0	2334
	IO	2748	0	100	2848

figura 4: tabela com diferença de tempo de escalonamento dado diferentes valores das constantes de aging para 150 processos

Com esse teste e ao acompanhar os logs do terminal, formulamos algumas hipóteses que decidimos investigar mais a fundo. Refizemos os testes, reduzindo o número de processos para um *sanity* de 10, mantendo a mesma proporção de prioridades para os processos.

valores constante aging	tipo de processo	tempos			
		READY	RUNNING	SLEEP	TURNAROUND
1TO2 = 80 2TO3 = 60 3TO4 = 40	CPU	206	44	0	250
	S-CPU	541	9	0	550
	IO	543	0	100	643
1TO2 = 200 2TO3 = 150 3TO4 = 100	CPU	251	45	0	296
	S-CPU	436	10	0	446
	IO	541	0	100	641
1TO2 = 600 2TO3 = 500 3TO4 = 400	CPU	239	43	0	283
	S-CPU	351	9	0	360
	IO	515	0	100	615

figura 5: tabela com diferença de tempo de escalonamento dado diferentes valores das constantes de aging para 30 processos

Analizando os resultados desses testes, chegamos às seguintes conclusões: o valor ideal para o aging varia principalmente com a carga de trabalho do sistema. No cenário com 150 processos em execução, o aging com valores altos demonstrou ser extremamente eficaz, elevando os processos para filas de maior prioridade conforme necessário e evitando a inanição. No entanto, nos testes com menos processos, as métricas permaneceram satisfatórias porque nenhum processo precisou envelhecer, indicando que, devido ao baixo volume de processos, o sistema operacional não teve que gerenciar o custo de migrá-los entre filas. Assim, ocorreu um cenário em que o aging não foi necessário.

Concluimos que o aging se mostra mais crucial em situações de alta carga de trabalho e com um grande número de processos dinâmicos. No entanto, quando há poucos processos, é mais eficiente evitar o aging, pois esses processos podem ser escalonados de forma adequada sem a necessidade de gastar recursos da CPU com a migração entre filas. Portanto, apesar de termos ajustado as quantidades de processos aceitas no xv6 para nossos testes, consideramos a quantidade padrão de processos, 64, ao escolhermos os valores das constantes de aging. Isso garante um equilíbrio delicado entre otimizar a prevenção da inanição e evitar o uso excessivo de recursos da CPU, respeitando a fila de prioridade do sistema.

3.4. Prioridade das Políticas

Para realizar uma análise sobre qual política de escalonamento deve ser aplicada em cada fila de prioridade, foram conduzidos testes utilizando os quatro algoritmos específicos. Esses testes consistiram na execução de cada algoritmo com conjuntos de 15, 60 e 300 processos, respectivamente. Os resultados obtidos forneceram insights sobre o desempenho de cada política em diferentes cenários, permitindo identificar em quais situações cada uma se destaca e em quais não é tão recomendada.

política de escalonamento	tipo de processo	tempos			
		READY	RUNNING	SLEEP	TURNAROUND
Round-Robin - Sanity 5	CPU	115	29	0	144
	S-CPU	171	4	0	176
	IO	163	0	100	263
Round-Robin - Sanity 20	CPU	476	29	0	505
	S-CPU	655	4	0	660
	IO	658	0	100	758
Round-Robin - Sanity 100	CPU	1890	31	0	1922
	S-CPU	3069	4	0	3073
	IO	3220	0	100	3320
política de escalonamento	tipo de processo	tempos			
		READY	RUNNING	SLEEP	TURNAROUND
SJF - Sanity 5	CPU	68	30	0	99
	S-CPU	184	6	0	190
	IO	154	0	100	254
SJF - Sanity 20	CPU	279	28	0	307
	S-CPU	685	6	0	691
	IO	562	0	100	662
SJF - Sanity 100	CPU	1432	28	0	1461
	S-CPU	3495	5	0	3500
	IO	2699	0	100	2794
política de escalonamento	tipo de processo	tempos			
		READY	RUNNING	SLEEP	TURNAROUND
FCFS - Sanity 5	CPU	56	27	0	84
	S-CPU	166	5	0	171
	IO	160	0	100	260
FCFS - Sanity 20	CPU	266	27	0	294
	S-CPU	693	6	0	699
	IO	698	0	100	798
FCFS - Sanity 100	CPU	1393	27	0	1421
	S-CPU	3443	5	0	3449
	IO	3529	0	100	3629
política de escalonamento	tipo de processo	tempos			
		READY	RUNNING	SLEEP	TURNAROUND
Lottery - Sanity 5	CPU	61	28	0	89
	S-CPU	173	6	0	179
	IO	164	0	100	264
Lottery - Sanity 20	CPU	293	29	0	323
	S-CPU	742	6	0	748
	IO	754	0	100	854
Lottery - Sanity 100	CPU	1623	28	0	1652
	S-CPU	3220	5	0	3226
	IO	3683	0	100	3783

figura 6: tabelas com diferença de tempo de escalonamento dado os diferentes algoritmos e quantidade de processos rodados

Os dados obtidos permitiram alcançar algumas conclusões significativas.

Primeiramente, observou-se que quando o número de processos é baixo, as políticas de escalonamento tendem a gerar resultados semelhantes, pois seus comportamentos distintos não impactam significativamente a execução dos processos. Entretanto, à medida que a quantidade de processos aumenta, as diferenças entre as políticas se tornam mais evidentes.

Para processos IO-Bound, o algoritmo Shortest Job First (SJF) demonstrou o melhor desempenho. Isso se deve à sua natureza, que sempre prioriza os processos com menor tempo de burst esperado. Por essa razão, implementamos essa política na fila de prioridade 3, onde seria ideal categorizar os processos de IO para otimizar suas execuções. Além disso, o SJF também apresenta um bom desempenho em relação aos outros tipos de processos.

Para os processos de S-CPU, percebe-se que a política de Round Robin teve um bom desempenho em comparação com os outros. Por ser um algoritmo preemptivo, apesar de gerar um possível overhead devido à troca de contexto, sua capacidade de dividir equitativamente o tempo de CPU entre os processos contribui para um bom tempo de resposta. Por esse motivo, tanto processos S-CPU quanto processos CPU-Bound poderiam ser atribuídos a essa política, o que também justifica o fato de optarmos por colocar esse algoritmo na fila de prioridade padrão, pelo fato de todo processo eventualmente ser escalonado e não haver nenhum processo tomando muito tempo da CPU podendo prejudicar outros.

Para processos de CPU, percebe-se que o FCFS teve um tempo de espera médio reduzido em comparação com outros. Como esse algoritmo apenas executa os processos em ordem de chegada, pode haver situações que esse algoritmo gaste muito tempo em certos processos e demore para chegar nos outros, gerando uma ineficiência, como é possível observar em seu tempo de espera nos processos de IO-Bound e de S-CPU. Ainda sim, o fato dele executar os processos por ordem de chegada pode ser útil quando se trata de processos de alta prioridade, que devem ser executados imediatamente sem ter que por exemplo ceder a CPU com interrupções e bloqueios, justificando o fato de escolhermos ele para ser o de maior prioridade.

Em relação ao algoritmo de Loteria, como é uma política que seleciona os processos de forma aleatória não é possível prever seu comportamento. Por esse motivo, toda vez que essa política for usada os resultados para os diferentes tipos de processo podem variar. No entanto, essa aleatoriedade pode ser benéfica, pois

garante a igualdade de tratamento entre os processos, ajudando a evitar o starvation. Considerando a falta de um padrão de comportamento previsível, optamos por aplicar essa política na fila de menor prioridade. Nesse contexto, onde os processos são menos críticos (caso contrário, estariam em uma fila de prioridade mais alta), a seleção aleatória do próximo processo a ser executado não compromete significativamente o desempenho do sistema.

3.5. Multifilas VS xv6

SO	tipo de processo	tempos			
		READY	RUNNING	SLEEP	TURNAROUND
NOSSO	CPU	571	48	0	619
	S-CPU	1181	9	0	1190
	IO	1207	0	100	1307
xv6 original	CPU	1042	55	0	1097
	S-CPU	404	0	0	404
	IO	1093	0	100	1193

figura 7: tabela com diferença de tempo de escalonamento comparando o xv6 original e nossas modificações

A implementação do xv6 original se baseia na utilização da política de Round Robin, essa que também é utilizada na nossa versão. Por esse motivo, ao comparar os dois códigos, temos que levar em conta que boa parte dos processos também serão escalonados por nossa implementação do Round Robin, uma vez que ela foi utilizada na fila de prioridade padrão.

Ao analisar a figura, é notório que os processos de CPU apresentaram um tempo de espera significativamente menor em nossa implementação, demonstrando uma eficiência superior na alocação de recursos de CPU. Essa melhoria pode ser atribuída à otimização na forma como gerenciamos as filas de processos prontos em nossa versão do sistema. Enquanto na implementação original do xv6 é necessário percorrer toda a fila de processos para selecionar o próximo a ser executado, em nossa versão, as filas contêm apenas os processos no estado de pronto, evitando assim esse overhead de busca.

Por outro lado, observamos um desempenho superior dos processos S-CPU na versão original do sistema. Essa disparidade pode ser atribuída às escolhas de constantes que fizemos, como a definição do INTERV, que torna o algoritmo menos preemptivo ao aumentar o quantum. Como resultado, embora nossa implementação

escalone de forma mais eficiente, certos tipos de processos, como os S-CPU, podem experimentar um tempo de turnaround mais longo.

4. Conclusões

Durante a execução deste trabalho, foi necessário aprofundar-se no funcionamento do sistema operacional xv6, a fim de realizar as modificações solicitadas de maneira eficaz.

Ao explorar a implementação de uma fila multinível, percebemos a complexidade envolvida, porém também reconhecemos suas numerosas vantagens. Como exemplo, a utilização eficiente de recursos se destaca, uma vez que a estrutura permite a alocação de recursos de forma otimizada para grupos de processos similares, garantindo um melhor aproveitamento dos recursos do sistema. Além disso, a flexibilidade inerente à fila multinível possibilita adaptações conforme as necessidades específicas de implementação.

Ao analisar os diferentes algoritmos de escalonamento utilizados, constatamos que o Round Robin é uma ferramenta versátil, apta a ser aplicada em diversas prioridades. Sua capacidade de lidar com múltiplos processos sem gerar starvation e manter um bom tempo de resposta o tornam uma escolha adequada para a prioridade padrão. O algoritmo Shortest Job First, embora demande cálculos mais complexos para estimar o tempo de burst esperado, revelou-se uma opção eficaz para processos IO-Bound, ao prever com precisão quais processos podem ser executados rapidamente, sem atrasar outros. Por essa razão, optamos por implementá-lo na fila de prioridade 3, ideal para alocação de processos IO-Bound.

A política FCFS, embora não apresente um tempo de resposta ótimo, foi escolhida para a fila de maior prioridade, pois garante a execução dos processos conforme sua ordem de chegada, sem interrupções ou atrasos. Quanto ao algoritmo de Loteria, embora não seja o mais eficiente em termos de desempenho, sua justiça intrínseca e a garantia de oportunidades iguais para todos os processos o tornam adequado para a fila de menor prioridade, onde a eficiência não é tão crítica.

Por fim, este trabalho ressaltou a importância da análise e compreensão aprofundada de sistemas operacionais, evidenciando a vantagem de selecionar implementações adequadas para diferentes contextos. A partir desta experiência, tornou-se claro que a escolha cuidadosa das políticas de escalonamento e

estruturas de gerenciamento de processos pode influenciar significativamente o desempenho e a eficiência de um sistema operacional.

5. Instruções para uso

Após abrir o zip e certificar que o qemu esteja instalado na sua máquina:

- direcione para o diretório “xv6-public”
- rode o código com: **make clean && make && make qemu**
- na prompt do xv6 execute os programas que desejar
 - exemplo:
 - `sanity 20`
 - irá rodar o programa *sanity.c* com 20 processos

6. Referências

<https://www.cs.virginia.edu/~cr4bd/4414/F2019/slides/20190912--slides-1up.pdf>

<https://people.cs.rutgers.edu/~pxk/416/notes/07-scheduling.html>

<https://www2.cs.uregina.ca/~hamilton/courses/330/notes/scheduling/scheduling.html>

<https://pdos.csail.mit.edu/6.828/2012/xv6.html>