Universidade Federal de Minas Gerais
Departamento de Ciência da Computação

Sistemas Operacionais — Trabalho Prático 2

Memória Virtual

Alexander Thomas Mol Holmquist

23 de janeiro de 2022

1 Introdução

Quando implementando um sistema operacional, um dos princípios que deve ser observado é o de **transparência**. Operações de baixo nível, como comunicação com dispositivos periféricos, proteção e segurança, *timesharing* e manipulação direta da memória (primária e secundária) devem ser abstraídas pelo sistema operacional, de forma a dar ao usuário uma experiência transparente. É com esse objetivo que todos os sistemas operacionais de PC atuais implementam um tipo de memória virtual, impedindo que o usuário acesse diretamente a memória física. Geralmente, essa memória virtual procura minimizar a fragmentação interna e externa da memória.

Tabela de páginas virtual é um exemplo de implementação que trabalha nesse sentido. Ainda que utilizando esse esquema, o sistema operacional precisa lidar com algumas questões. A principal delas é que a tabela virtual precisa ter tamanho limitado. Sendo assim, é necessário estabelecer uma estratégia de substituição de páginas. Quando tal limite for atingido, a estratégia entra em ação.

Estratégias de reposição de páginas já foram um tópico aquecido de pesquisa. Atualmente, existem várias heurísticas conhecidas. Exemplos incluem: *Least Recently Used* (LRU), *Least Frequently Used* (LFU), *First In First Out* (FIFO), *Most Recently Used* (MRU), *Adaptive Replacement Cache* (ARC), etc. Neste trabalho, construimos um simulador de memória virtual, além de três estratégias de reposição distintas: LRU, FIFO, e "newalg", que se assemelha ao Least Frequently Used.

2 Implementação e Desafios

Dividimos o programa em quatro módulos: *Alg*, *Interface*, *Memory* e *Utils*.

- *Alg* contém os algoritmos de reposição e funcionalidades relacionadas com a leitura do arquivo de entrada e registro de estatísticas de uso.
- *Interface* é responsável por receber as entradas do usuário, transformá-las para o formato certo, e então passar para o emulador.
- *Memory* contém a implementação da tabela de páginas virtual.
- *Utils* contém utilidades não relacionadas a uma parte específica do programa.

A memória virtual foi implementada em *Memory* como um tipo template–classe *vtable*. Cada estratégia de reposição deve fornecer a implementação de um comparador customizado. Internamente, *vtable* passa esse comparador como argumento de uma *heap* de Fibonacci, que vai ser responsável por sempre manter no elemento *top* a página que deve ser removida naquele momento. Isto traz o custo de encontrar tal página para $O(log\ n)$.

Além disso, a tabela virtual mantém um vetor chamado *status*. Este vetor nos ajuda a acessar o estado de uma página (ativa ou não, suja ou não) rapidamente, partindo de seu identificador. Esta separação de duas estruturas de dados foi feita para acomodar mais facilmente as diferentes estratégias de reposição. Ao chegar a hora de implementar "newalg", contudo, encontramos dificuldades.

No algoritmo *newalg*, buscamos mesclar várias métricas, incluindo o número de acessos a uma página. Para isso, precisamos de atualizar uma página específica da *heap* a cada leitura ou escrita. Porém, é muito custoso percorrer uma estrutura de dados *heap* buscando por uma

página específica. A ideia da heap é ser acessada somente pelo topo. Por isso, decidimos atualizar o número de acessos somente quando uma nova página é inserida: mantemos um histórico de acesso. Ao inserir uma página, se aquela página havia sido alocada anteriormente, lhe fornecemos o número de acessos que ocorreram durante estadia anterior no momento em que ela volta para a tabela.

Partimos então para testar várias mesclagens de métricas para *newalg*. Inicialmente, pensamos em $log(lastUsedAt) * historyUsed$ e $\sqrt{lastUsedAt} * historyUsed$, onde *lastUsedAt* é o tempo do último acesso àquela página, e *historyUsed* é a quantidade de acessos que a página teve última vez que esteve na tabela. Testou-se sobre os exemplos fornecidos na especificação do trabalho. Estas duas estratégias, porém, aumentaram o tempo de execução. Não se mostraram eficientes na prática. Além destas, muitas outras combinações foram testadas sem sucesso, incluindo normalização, exponenciação, e tomar uma média com pesos.

Por fim, definiu-se a estratégia de *newalg* como *historyUsed*, sem adições (veja Listing 1). Esta é a estratégia que rendeu maior desempenho. Note que o algoritmo é diferente do conhecido *LFU*, pois o número de acessos a uma página só é atualizado no momento de inserção.

```
struct newalgPageCompare {
  bool operator()(const Memory::pageT& p1, const Memory::pageT& p2) const {
    // We want page with minimum historyUses to be replaced
    return p1.historyUses > p2.historyUses;
  }
};
```

Listing 1: Estratégia de reposição de páginas para *newalg*

3 Avaliação

Para compararmos cada algoritmo de reposição de páginas desenvolvido, executamos cada um sobre o conjunto de testes fornecido na especificação do trabalho. Os resultados foram todos registrados, mas nesta análise utilizaremos somente `matriz.log` (acessos a matrizes representam padrão muito comum) e `simulador.log` (demonstrou ser o conjunto de entradas mais desafiador).

Page\Mem		128	512	2048	8192	16384
4	LRU	996K—36	985K—32	892K—158	277K—663	3K—0
	FIFO	998K—24	987K—42	893K—172	277K—667	3K—0
	NALG	991K—38	968K—68	850K—195	257K—577	3K—0
16	LRU	989K—47	985K—83	974K—60	818K—196	477K—380
	FIFO	990K—53	987K—63	973K—73	819K—192	478K—384
	NALG	992K—41	992K—45	979K—64	804K—205	466K—392
64	LRU	974K—181	974K—181	974K—186	959K—206	900K—249
	FIFO	974K—181	974K—182	974K—186	959K—206	900K—249
	NALG	974K—181	974K—183	973K—186	956K—212	897K—255

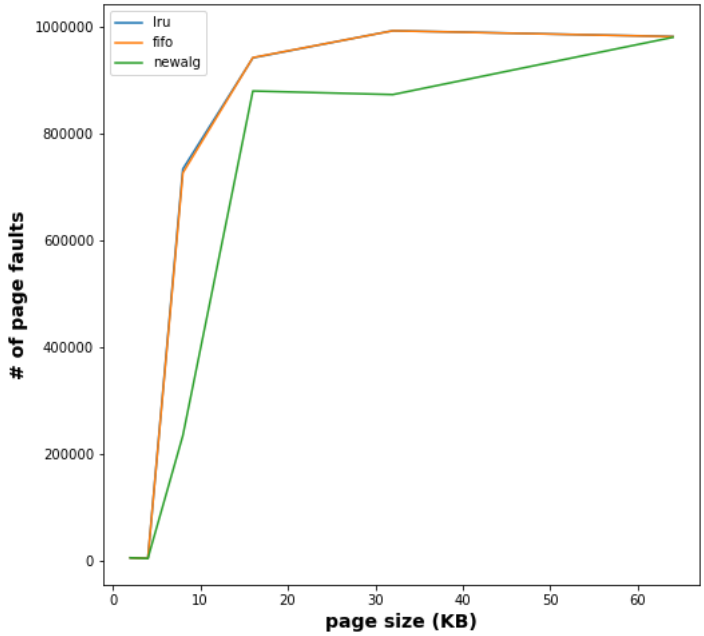
Tabela 1: Resumo dos resultados para a benchmark `matriz.log`. *nnnK—n* indica a quantidade de *page faults* seguido pela quantidade de *page writebacks*.

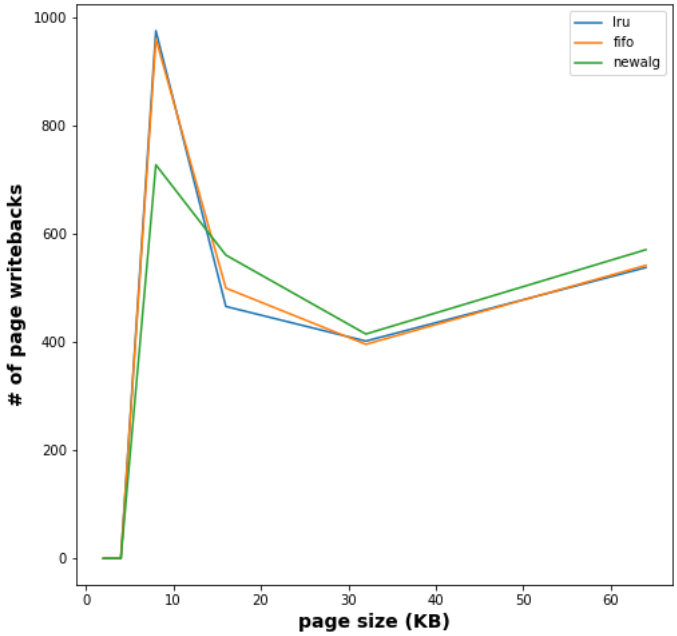
Pode-se observar na Tabela 1 que não há grande diferença de desempenho entre os algoritmos, neste caso. Em geral, a pior estratégia parece ser *FIFO*. É interessante notar que *newalg*,

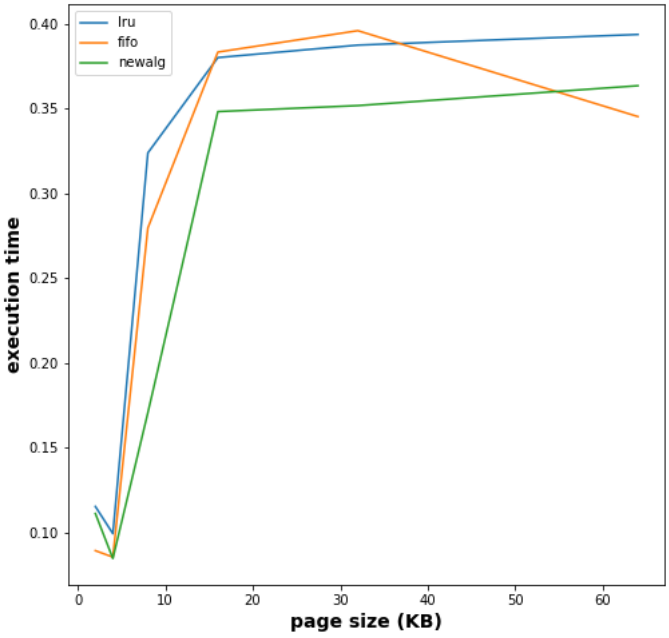
apesar de seguir uma heurística estranha de "reputação" das páginas, apresenta desempenho bom. Veja os número de *page faults* registrados na primeira linha da tabela.

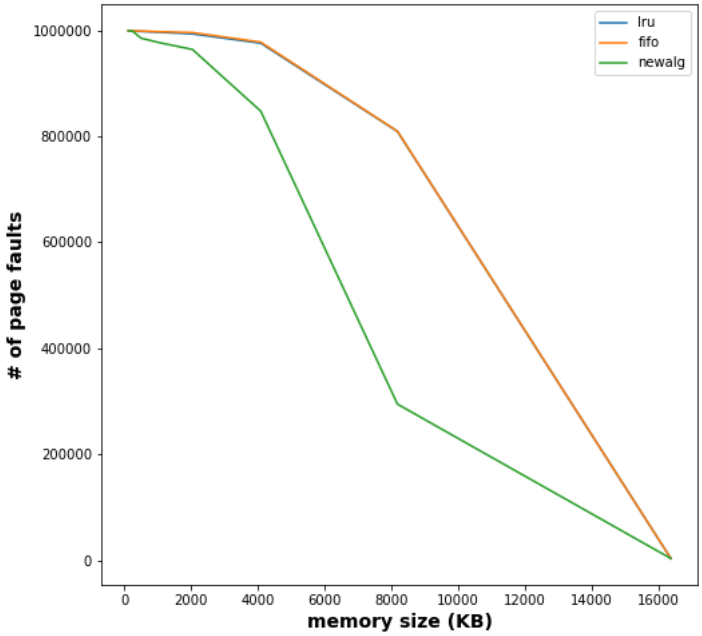
A seguir, apresentamos gráficos para cada métrica existente: *page faults*, *page writebacks* e tempo de execução. Inclui-se gráficos com o tamanho da página fixo em *4KB*, variando-se somente o tamanho total da memória, e gráficos para memória constante em *16384KB* com o tamanho da página variando. A benchmark utilizada é a `simulador.log`. O tempo de execução é dado em segundos.

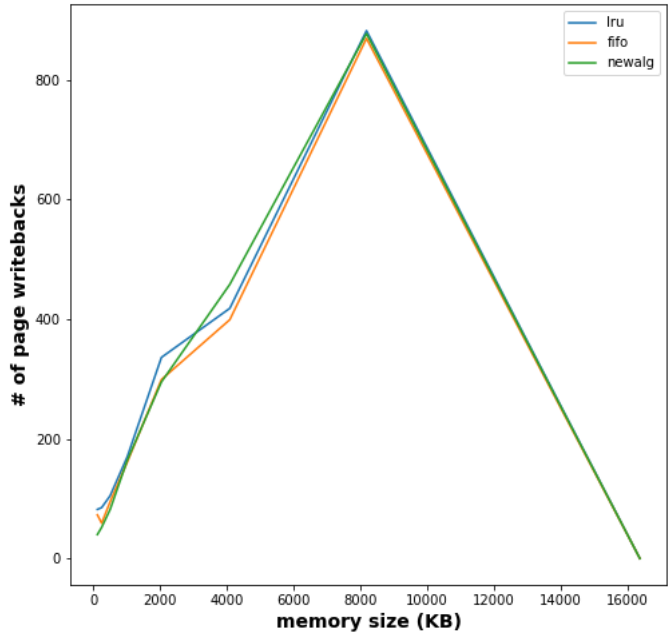
Como no caso de `matriz.log`, aqui se evidencia que *newalg* tem uma performance melhor com respeito a *page faults*, em geral, porém perde um pouco em *page writebacks*. *newalg* parece ser mais eficiente em tempo de execução, em alguns casos alcançando ganho de 50%.

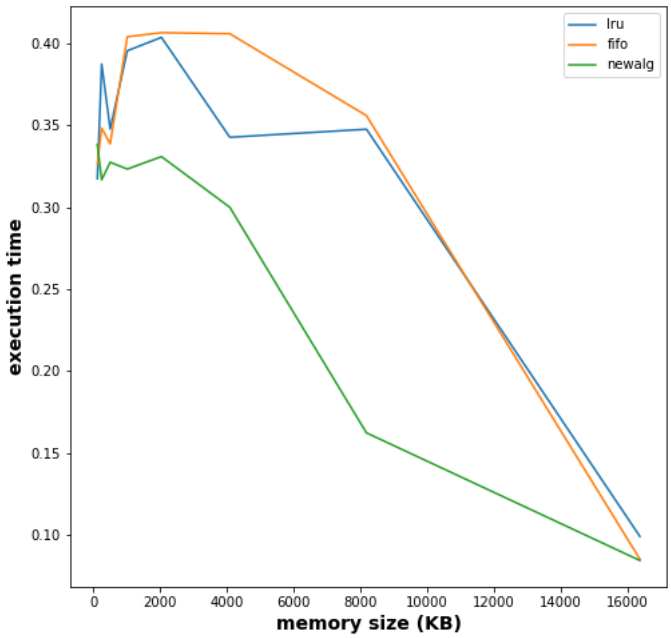



lru
fifo
newalg
# of page writebacks
page size (KB)
0
200
400
600
800
1000
0
10
20
30
40
50
60


lru
fifo
newalg
execution time
page size (KB)
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0
10
20
30
40
50
60


lru
fifo
newalg
1000000
800000
600000
400000
200000
0
# of page faults
0
2000
4000
6000
8000
10000
12000
14000
16000
memory size (KB)


lru
fifo
newalg
800
600
400
200
0
# of page writebacks
0
2000
4000
6000
8000
10000
12000
14000
16000
memory size (KB)



4 Instruções de compilação

Para compilar o programa, basta digitar `make` enquanto no diretório principal do projeto. É necessário ter instalada a biblioteca Boost [1]. Note que o único ambiente testado é o sistema operacional Ubuntu (ver Apêndice A) com versão 1.71 da biblioteca. A interface de linha de comando do programa segue a especificação. Existe um parâmetro opcional para que o programa imprima saída de fácil leitura (passe "test" como quinto parâmetro). Um repositório remoto com todo o projeto pode ser encontrado na ferramenta GitHub: [2].

4.1 Configurações

É possível encontrar alguns parâmetros ajustáveis nos arquivos `lib/make/global-var/global-var-general.mk` e `lib/main/header/Utils/defs.hpp`. Por exemplo, para que o programa imprima o tempo de execução do algoritmo, deve-se definir o macro *PRINT_EXECUTION_TIME* no segundo arquivo mencionado.

A Detalhes da máquina local

- **Sistema operacional:** Ubuntu 20.04.3 LTS.
- **CPU:** Intel(R) Core(TM) i7-1065G7 @ 1.30GHz-3.90GHz. 1 processador físico, 4 núcleos, 8 threads virtuais.
- **RAM:** 16GB tecnologia DDR4 a 3200MHz.

Referências

[1] Boost library. https://www.boost.org/. Acessado em 23 de janeiro de 2022.

[2] Git repository. https://github.com/Yowgf/virtual-memory-emulator. Acessado em 23 de janeiro de 2022.