Universidade Federal de Minas Gerais **Sistemas Operacionais**

Instituto de Ciências Exatas **1° Semestre de 2017**

Departamento de Ciência da Computação **Prof. Flávio Figueiredo**

Alunos: Caíque Bruno Fortunato **Matrícula:** 2013062731

Pâmela Carvalho da Silva 2013073474

Tiago Melo Tannus 2012079762

Documentação do TP2: Memória

VMM + AhLoka! + Garbage Collection

Introdução

Um Sistema Operacional é uma interface entre o hardware e software que realiza troca de contextos, gerência de interrupções, chamadas de sistemas, entre outros.

Um Processo é abstração base de Sistemas Operacionais, que possuem atividades como escalonamento, paralelismo e compartilhamento de dados dentro de um processo e sincronização.

No entanto, ainda não foi falado nada sobre gerência de memória de um SO. A maioria dos computadores atuais trabalham com a ideia de memória hierárquica, possuindo uma pequena memória muito rápida chamada de Cache, uma quantidade considerável de memória principal (RAM) e muito espaço para armazenamento de disco (HD), que é considerada lenta.

O problema básico para o gerenciamento de memória é que os programas atuais são muito grandes para rodarem inteiramente no cache. Logo, deve existir um

gerenciador de memória que deve ser capaz de controlar parte da memória que está em uso (e quais não estão), alocar memória para processos quando eles necessitam e desalocar quando eles terminam e, principalmente, gerenciar a troca entre a memória principal e o disco, quando a memória principal é muito pequena para armazenar todos os processos.

Existem dois tipos de memória principal: Lógica e Física. A Memória Física recebe o próprio Sistema Operacional e os aplicativos que o usuário está executando no momento. O restante, o que o Sistema Operacional acredita que não será utilizado com urgência, ocupa espaço na Memória Virtual.

Para fazer a “transição” entre as memórias é necessário realizar a tradução de endereços lógicos para endereços físicos. Esse processo de tradução de endereços lógicos em endereços físicos é realizado por uma unidade de gerência de memória chamada MMU (Memory Management Unit).

Este trabalho prático tem como principal propósito fixar o conteúdo de Memória, aprendido na disciplina de Sistemas Operacionais. Em particular, o foco está no entendimento de gerência de memória de um kernel é sua política de reposição de páginas, gerenciamento de alocações e um coletor de memória simples por contagem de referências.

Organização

Este trabalho está dividido em três partes principais.

Parte 1: VMM

A primeira parte consiste na simulação de uma Memória Virtual e a política de substituição de páginas para o disco. Ou seja, implementação de algoritmos que decidem que páginas da memória serão gravadas no disco quando uma nova página precisa ser alocada. A paginação ocorre quando todas as molduras de página (page frames) estão ocupadas e é necessário adicionar uma nova página para atender um

page fault, a política de reposição determina qual das páginas atualmente em memória deve ser excluída.

Parte 2: AhLoka!

A segunda parte consiste em implementar uma biblioteca de gerenciamento de alocações utilizando a biblioteca mmap para alocar e desalocar espaços de memória. A ideia é encaixarmos programas na memória utilizando os quatro algoritmos para encaixe: Primeiro, Próximo, Melhor e Pior encaixe.

Parte 3: Garbage Collection

Por fim, a terceira e última parte consiste em simular um coletor de memória simples por contagem de referências sem se preocupar com ciclos. O simulador deverá ter operações novas de dependências entre ids de alocação.

Implementação

Todo o trabalho foi implementado em linguagem C em três máquinas máquinas: Duas com sistema operacional Linux na versão 4.4.0-72-generic com processador Intel ® Core™ i5 5° geração possuindo 8GB de memória RAM e 1TB de memória de disco. Já a outra máquina possui sistema operacional Linux na versão 4.4.0-72-generic com processador Intel® Core™ i7 5° geração possuindo 8GB de memória RAM e 500GB de memória de disco.

Parte 1: VMM

Implementação dos algoritmos

Para implementar a primeira parte do trabalho prático consideramos quatro algoritmos de substituição de páginas, conforme solicitado na especificação, são eles:

- **Fifo:** O Sistema Operacional mantém uma fila de páginas correntes na memória de modo que a página no início da fila é a mais antiga e a

página no final é a mais nova. Quando ocorre um Page Fault a página no início é removida e a nova é inserida no final da fila.

- Embora seja simples, o algoritmo pode se tornar bastante ineficiente, principalmente se uma página que está em uso constante pode ser retirada.

A implementação do Fifo foi simples: A tabela é percorrida na coluna do endereço da memória física. Se a página na posição da iteração for igual ao parâmetro fifo_frm, ou seja, a primeira que entrou, então ela será retornada.

```
for(int i = 0; i < num_pages; i++)
        if(page_table[i][PT_FRAMEID] == fifo_frm)
                return i;
```

- **Second Chance:** Possui a mesma estrutura do FIFO, mas com uma importante modificação: Antes de retirar uma página bit de referência (R) é inspecionado. Se o bit for 0, então a página é velha e não foi usada recentemente, sendo trocada. Caso contrário, o bit 1 se torna 0, a página é colocada no final da fila e o tempo de carga é modificado, fazendo parecer que recém chegou na memória, recebendo, portanto, uma segunda chance.

A implementação é similar à do algoritmo Fifo, mas com uma condição a mais para verificar o bit de referência e realizar o algoritmo descrito acima.

```
        for(int i = 0; i < num_pages; i++) {
                if(page_table[i][PT_FRAMEID] == fifo_frm) {
                        if(page_table[i][PT_REFERENCE_BIT] == 0)
                                return i;
         else {
            page_table[i][PT_REFERENCE_BIT] = 0;
```

```
            fifo_frm += 1;
            fifo_frm = fifo_frm % num_frames;
            i = -1;
        }
                }
        }
```

- **NRU:** O algoritmo Not Recently Used Page Replacement, ou seja, Não Recentemente Usado serve para substituir páginas através de coleta de estatísticas de uso:
    - 02 bits associados a cada página: R e M, Referenciada (lida ou escrita) e Modificada (escrita), respectivamente.
        - Classe 0 (00): Não referenciada, não modificada;
        - Classe 1 (01): Não referenciada, modificada;
        - Classe 2 (10): Referenciada, não modificada;
        - Classe 3 (11): Referenciada, modificada.

  Para ajudar a coletar estatísticas, os bits R e M são atualizados a cada referência à memória e armazenados em cada entrada da tabela de página.

  Periodicamente, o bit R é limpo para marcar as páginas que não foram referenciadas recentemente. No entanto, o bit M nunca é limpo, já que o SO precisa saber se deve escrever a página no disco.

  A implementação segue abaixo:

```
int val1 = 0, val2 = 0, cont = 0;

// Se gerou um cliclo de clock
if(clock == 1)
      for(int i = 0; i < num_pages; i++)
            page_table[i][PT_REFERENCE_BIT] = 0;
```

```
        while(1) {

                // Prioridades
                if(cont == 0) val1 = 0, val2 = 0;
                if(cont == 1) val1 = 0, val2 = 1;
                if(cont == 2) val1 = 1, val2 = 0;
                if(cont == 3) val1 = 1, val2 = 1;

                for(int i = 0; i < num_pages; i++) {
                        // Filtro as paginas que estao na tabela
                        if(page_table[i][PT_FRAMEID] != -1)
                                // Procuro se tem algum R=0 e M=0
                                if(page_table[i][PT_REFERENCE_BIT] ==
val1 &&
                                        page_table[i][PT_DIRTY] == val2)
                                        return i;
                }
                cont++;

        } // Fim do while
```

- **Aging:** Por fim, o Aging combina o algoritmo LFU (Não discutido neste trabalho) com o tempo, evitando assim Starvation. Em vez de apenas incrementar os contadores de páginas referenciadas, colocando ênfase na igualdade de referências de página, independentemente do tempo, o contador de referência em uma página é primeiro deslocada para a direita (dividido por 2), antes de adicionar o bit referenciada à esquerda desse número binário. Tal técnica garante que as páginas referenciadas mais recentemente, embora menos frequentemente referenciada, terão maior prioridade sobre as páginas mais freqüentemente mencionados no passado. Assim, quando uma página precisa ser trocado, a página com o menor contador será escolhido.

    A implementação segue abaixo:

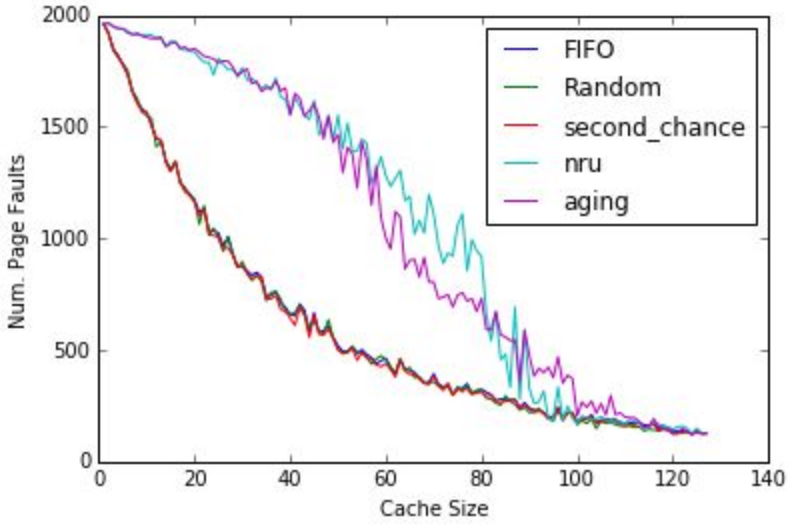
```
   int menor = 1, tira;

for(int i = 0; i < num_pages; i++) {

      page_table[i][PT_AGING_COUNTER] >>= 0x1;

      if(page_table[i][PT_REFERENCE_BIT] == 1) {
            page_table[i][PT_AGING_COUNTER] |= 0x1 << 7;
      }

} // End of FOR cicle

// Acha o menor agora...
for(int i = 0; i < num_pages; i++) {
      if(page_table[i][PT_FRAMEID] != -1) {
            if (page_table[i][PT_AGING_COUNTER] < menor) {
                  menor = (page_table[i][PT_AGING_COUNTER]);
                  tira = i;
            }
      }

}

return tira;
```

Experimentos rodados

Para demonstrar a validade dos algoritmos, foram rodados experimentos de acordo com os casos de testes enviados na pasta da documentação. Segue abaixo a visualização dos dados, feita seguindo orientações de utilização da ferramenta Jupyter, da Anaconda 3.

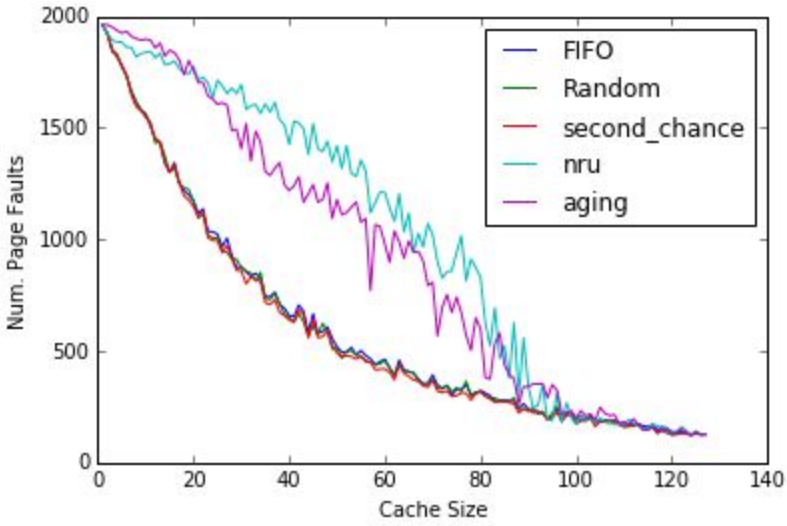
- **Pasta 8020:**

algorithm	faults	cachesize	clockfreq	input
aging	1035.007874	64.0	1.0	8020.0
fifo	597.283465	64.0	1.0	8020.0
nru	1060.338583	64.0	1.0	8020.0
random	590.377953	64.0	1.0	8020.0
second_chance	588.165354	64.0	1.0	8020.0



Com o ciclo de clock maior, temos:

algorithm	faults	cachesize	clockfreq	input
aging	918.409449	64.0	3.0	8020.0
fifo	597.283465	64.0	3.0	8020.0
nru	1003.960630	64.0	3.0	8020.0
random	591.149606	64.0	3.0	8020.0
second_chance	582.551181	64.0	3.0	8020.0



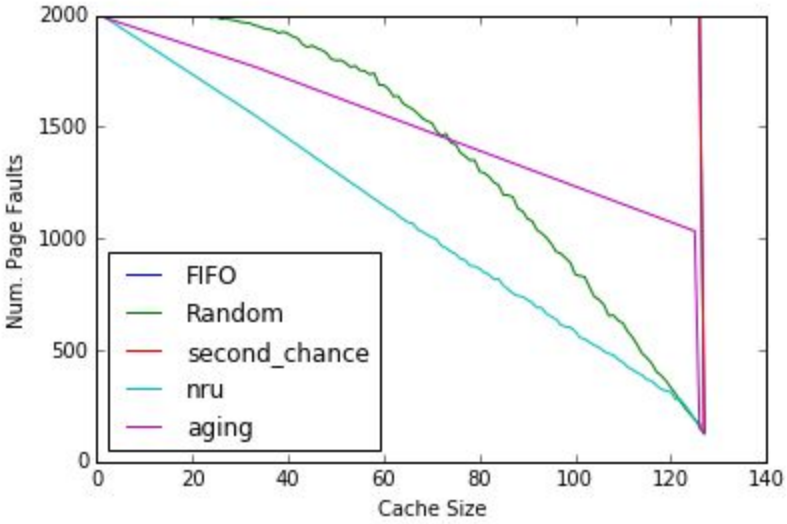
Os algoritmos de FIFO, Random e Second Chance seguem o padrão esperado como aparece no OStep, referência do trabalho, mas como o eixo Y está invertido no gráfico, o número de hits aqui decresce a medida quanto mais alto o valor no eixo, o desenho ficou invertido. Por outro lado os algoritmos Aging e NRU ficaram com estatísticas bem diferentes dos demais. Uma possível causa é o tratamento com o ciclo de clock. Como o OStep não trata tais casos (Aging e NRU), não podemos validar somente a partir da comparação.

- **Pasta Looping:**

Já para as entradas da pasta Looping o resultado foi bem diferente. O Second chance, como sempre da uma segunda chance, obteve um comportamento diferente dos demais. O mesmo ocorreu com o FIFO.

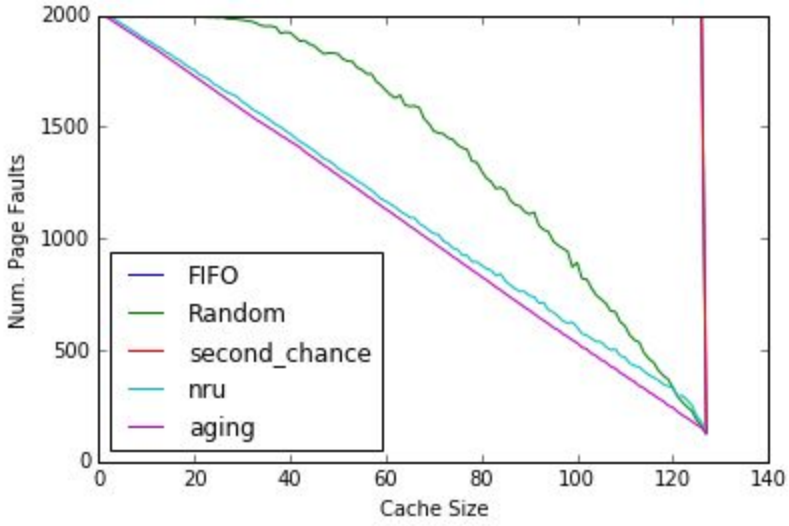
O Random (implementado pelo professor no esqueleto) possui um pior desempenho e os algoritmos Aging, FIFO e Second Chance se comportam de maneira diferente.

algorithm	faults	cachesize	clockfreq
aging	1503.000000	64.0	1.0
fifo	1985.251969	64.0	1.0
nru	1096.204724	64.0	1.0
random	1416.858268	64.0	1.0
second_chance	1985.251969	64.0	1.0



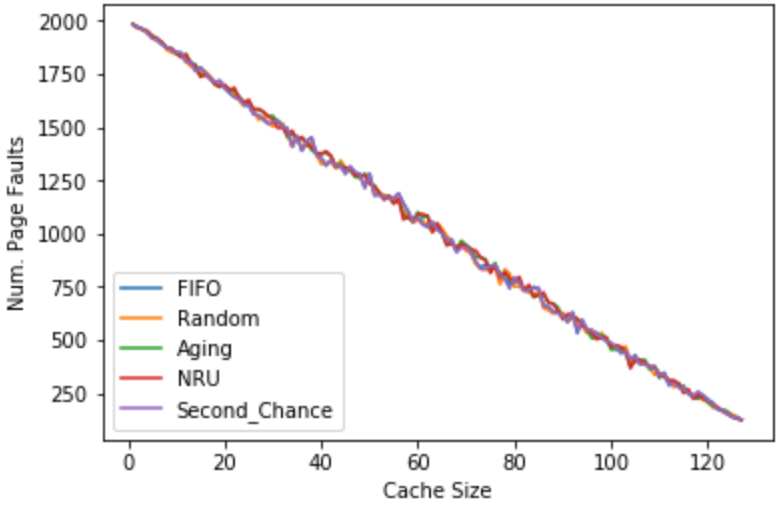
Quando o ciclo de clock aumentou (para 3), os resultados se mantiveram bem parecidos, com exceção do Aging:

algorithm	faults	cachesize	clockfreq
aging	1070.000000	64.0	3.0
fifo	1985.251969	64.0	3.0
nru	1114.763780	64.0	3.0
random	1418.582677	64.0	3.0
second_chance	1985.251969	64.0	3.0



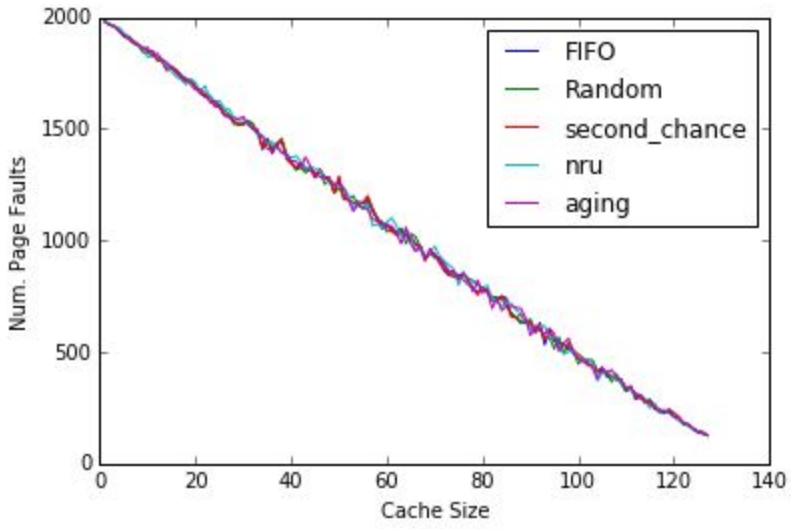
- **Pasta Locality:**

algorithm	faults	cachesize	clockfreq
aging	1026.307087	64.0	1.0
fifo	1022.330709	64.0	1.0
nru	1026.842520	64.0	1.0
random	1019.370079	64.0	1.0
second_chance	1021.755906	64.0	1.0



Quando aumentamos o ciclo do clock para 3 os resultados se mostraram bem parecidos:

algorithm	faults	cachesize	clockfreq
aging	1023.921260	64.0	3.0
fifo	1022.330709	64.0	3.0
nru	1026.440945	64.0	3.0
random	1022.834646	64.0	3.0
second_chance	1022.188976	64.0	3.0



Como mostrado no gráfico acima, para os arquivos de teste da pasta Locality todos os algoritmos possuem um comportamento semelhante.

Parte 2: AhLoka!

Na segunda parte do trabalho prático a tarefa foi implementar um procedimento de gerenciamento de alocações usando as técnicas aprendidas em sala, basicamente deveríamos simular um malloc e um free usando as seguintes funções: Best Fit, Next Fit, Worst Fit e First Fit.

Fomos providos de um esqueleto base que tinha uma lista encadeada representando a memória e um nó com tamanho da memória ocupado por ele. A memória tinha um tamanho total de 4mb e o cabeçalho contendo os dados de cada nó tinha um tamanho de 24bytes. A partir disso nos foram fornecidas entradas de alocação e desalocação para o teste das funções.

Implementação dos algoritmos

- **Next Fit:** A função nextFit foi implementada muito similar à função passada como exemplo alwaysGrow, onde a alocação sempre acontecia no final da lista encadeada e o cálculo do espaço livre a sua frente era feito com o tamanho total da memória subtraindo disso o espaço total já ocupado por todos os nós até o momento.

- **First Fit:** A função firstFit seguiu apenas a função de exemplo como referência, já que o seu funcionamento é levemente diferente. A função anterior sempre pegava o final da fila para a alocação, mas a firstFit sempre pega o primeiro espaço livre de memória para alocar, caminhando do início da lista.

  Para realizar esse procedimento foi implementado um loop que caminhava pela lista e procurava o primeiro espaço que comportava o bloco a ser alocado. Quando esse espaço era encontrado o loop acabava e a memória era alocada usando dos cálculos necessários.

- **Best Fit:** A função bestFit funcionava similarmente à função firstFit, já que era necessário percorrer a lista para encontrar a posição em que a memória seria alocada, a diferença é que mesmo que encontrasse um espaço, ela continuaria procurando para conseguir encontrar um espaço que fosse menor e ainda coubesse o bloco. Para que isso acontecesse, o loop deveria ser executado em sua inteiridade verificando todos os

espaços livres da lista, comparando-os e identificando o menor que coubesse o bloco.

- **Worst Fit:** A função worstFit segue a mesma lógica da bestFit acima, a única diferença é que a comparação do loop era para achar o maior espaço livre que coubesse a alocação.

Resultados

Os algoritmos tiveram saídas com resultados esperados.

Parte 3: Garbage Collection

Não foi implementado.

Conclusão

Este trabalho não foi concluído com sucesso devido a dificuldade com a compreensão do esqueleto e com a linguagem solicitada, C, de maneira geral. Consequentemente, não tivemos muito tempo para poder ter certeza dos resultados e de realizar a parte 3.

No entanto, tivemos a oportunidade de aprender melhor conceitos de memória de um Sistema Operacional conforme especificado na introdução e ao longo deste trabalho.

Referências

GitHub do professor.