Sistemas Operacionais

Prof. Valeria Bastos

DCC/IM/UFRJ

Segundo Trabalho

18 de fevereiro de 2016

Componentes do grupo:

Eduardo Costa – DRE: 112226744
Patricia Ghiraldelli – DRE: 112190951

Índice

1. Introdução 3
2. Estratégias para solução 3
    2.1. Criação dos processos 3
    2.2. Solicitações de páginas virtuais 3
    2.3. Exclusão mútua e ordem 3
    2.4. Working set com algorítmo LRU (Last Recently Used) 4
    2.5. Alocando na memória principal 5
    2.6. Swap out de processos 5
    2.7. Swap in 6
3. Execução 7
    3.1. Computador utilizado 7
    3.2. Compilação e execução do programa 7
    3.3. Opções 7
    3.4. Modo básico 8
    3.5. Modo completo 9
4. Dificuldades 10
    4.1. Exclusão mútua 10
    4.2. Sincronização 10
    4.3. Exibição da informação 10
5. Conclusão 10
6. Bibliografia 11
    6.1. Solução do problema proposto 11
    6.2. Interface 11

1. Introdução

O trabalho propôs a criação de um simulador de gerenciamento de memória virtual, com a utilização do algorítmo de substituição de páginas *LRU* (*Last Recently Used*).

Para isso, foram utilizados conhecimentos adquiridos na disciplina ao longo do período, como gerenciamento de memória e concorrência entre threads, e também encontrados em pesquisa na Internet.

O simulador foi implementado na linguagem C, com a utilização da biblioteca pthreads para a criação de threads e a concorrência.

Ao longo deste relatório será detalhado cada passo para a construção do simulador.

2. Estratégias para solução

Para a elaboração deste trabalho, foi necessário a utilização de algumas técnicas para a implementação da parte concorrente, e para o correto funcionamento do gerenciador de memória. Estas serão apresentadas aqui.

2.1. Criação dos processos

Os processos, neste trabalho, são representados por threads. Cada thread é criada a cada `TEMPO_NOVOPROC` segundos, diretiva definida no header `comum.h`. Num loop, na `main`, a função `novo_processo()` é chamada, e esta cria a thread.

2.2. Solicitações de páginas virtuais

Após a thread que representa o processo ter sido criada, a função `thread_processo` é executada pela thread nova. Nesta função, é atualizado o contador de tempo, e a `solicita_pv`, responsável por atender uma solicitação de página virtual, é chamada. Porém, foi preciso o uso de execlusão mútua para as solicitações, para que uma seja atendida de cada vez.

2.3. Exclusão mútua e ordem

Para impedir que acessos concorrentes sejam feitos à memória principal, foi necessário o uso de exclusão mútua. Antes de chamar a função `solicita_pv`, a thread obtém o lock da variável de exclusão mútua `mutex`.

Porém, isso traria um problema: na exclusão mútua com as funções `pthread_mutex_lock` e `pthread_mutex_unlock`, não é garantido que as threads em espera serão liberadas na ordem que chegaram. Não é uma fila. De acordo com a documentação, a `pthread_mutex_unlock` apenas libera para alguma outra entrar na seção crítica, e quem decide qual será é o escalonador de threads do sistema operacional. Desta forma, não havia garantia de que os pedidos seriam atendidos na ordem em que foram feitos, e poderia ocorrer um caso de *starvation* - determinada thread nunca conseguir o mutex e ficar esperando para sempre.

Para solucionar este problema, utilizamos um sistema de “tickets”, similar ao sistema de senhas de atendimento de um banco, onde cada thread que vai entrar na região crítica pega uma “senha”. Ao conseguir entrar, é verificado se a próxima senha a ser atendida é a sua. Se não for, se bloqueia com `pthread_cond_wait`. Ao término de uma região crítica, as threads bloquedas são acordadas (`pthread_cond_broadcast`), e cada uma delas vai checar se é a sua vez, dentro de um loop. Se não for, é bloqueada outra vez.

Parte desse código é mostrado abaixo:

```
meu_ticket = ticket++;

pthread_mutex_lock(&mutex);

while(!ok || prox_ticket < meu_ticket)
     pthread_cond_wait(&cond, &mutex);
```

2.4. Working set com algorítmo LRU (Last Recently Used)

Quando uma solicitação de página virtual é atendida (função `solicita_pv`), a primeira coisa a ser feita é checar se a página já está no *working set*.

No caso em que a página já existe no *working set*, para indicar que ela foi usada recentemente, a mesma é movida para o final, com um mecanismo similar ao usado no *bubble sort* – troca repetidamente de lugar com a página à sua direita, até que se chegue no final. Após isso, o atendimento da solicitação é concluído.

```
// colocar no final (LRU)
for(j = i; j < WORKING_SET-1; j++) {
     if(processos[proc].working_set[j+1] != -1) { // se não já for o último
           int temp = processos[proc].working_set[j];
           processos[proc].working_set[j] = processos[proc].working_set[j+1];
           processos[proc].working_set[j+1] = temp;
     }
}
```

Caso a página não esteja no *working set*, ocorre *page fault*, e a página deve ser adicionada no *working set* e alocada na memória principal.

2.5. Alocando na memória principal

Quando ocorre *page fault*, é chamada a função `adiciona_pv`. Dessa vez, a página precisa de fato ser alocada na memória principal.

Primeiramente, é verificado se o *working set* está cheio, para sabermos se será uma nova página, ou se substituirá a mais antiga (conforme o algorítmo *LRU – Last Recently Used*, como visto acima).

Caso não esteja cheio, é simplesmente adicionado no final, e depois, adicionada na memória principal num novo *frame*, e na tabela de páginas.

Caso esteja cheio, a primeira (mais antiga) sai, e a nova página entra no final:

```
// o primeiro sai
for(i = 0; i < WORKING_SET-1; i++) {
     processos[proc].working_set[i] = processos[proc].working_set[i+1];
}

// nova PV entra no final
processos[proc].working_set[WORKING_SET-1] = num;
```

Após isso, a página é subsituída na memória principal, e a tabela de páginas é atualizada.

2.6. Swap out de processos

No caso do *working set* não cheio, a página deve ser adicionada a um novo *frame* da memória principal, como vimos acima. Porém, o que ocorre no caso da memória estar lotada?

Neste caso, algum processo deve ser removido da memória. Esta ação recebe o nome de *swap out* – o processo é movido para a área de *swap*, para dar lugar a outro processo. Neste trabalho, o critério para decidir qual processo sofre *swap* é o *FIFO - First In First Out* (o processo que entrou primeiro na memória, será o primeiro a sair).

Para que seja possível saber qual processo entrou primeiro, foi preciso adicionar um novo campo à estrutura `processo`. Foi uma variável `entrada`, do tipo `int`, que indica a ordem de entrada de cada processo na memória. Uma variável global `num_entrada` é inicializada com `0`, e, sempre que um processo entra na memória, seu campo `entrada` recebe `num_entrada`, e `num_entrada` é incrementado em 1 unidade:

```
// alterar o numero de entrada pra saber que ele entrou agora
processos[proc].entrada = num_entrada++;
```

O *swap out* em si ocorre na função `swap_out`, que percorre a memória e verifica, para cada *frame*, o campo `entrada` de seu processo correspondente, a fim de encontrar o menor valor (processo que entrou mais cedo), para, assim, saber qual processo será removido. Determinado o processo, ele é removido, e os frames que suas páginas virtuais ocupavam agora estão livres para serem utilizados pelas páginas de outro processo.

2.7. Swap in

Já no *swap in*, o que ocorre é o oposto: um processo que estava em *swap* volta à memória. Em nossa implementação, a lista do *working set* continua salva, mesmo quando o processo está na área de *swap*. Portanto, na hora de voltar, só é preciso percorrer o *working set* e adicionar as páginas virtuais de volta na memória.

Primeiramente, precisamos checar se há espaço livre na memória. Se não houver, será feito o *swap out* de outro processo, para que o processo quer voltar tenha espaço.

Porém, pode ocorrer o caso em que mais de um processo precisa ser tirado da memória. Considere a seguinte situação: a memória está lotada, e o processo mais antigo (que, portanto, será removido no *swap out*) só está alocando, no momento, 2 *frames* da memória principal. O que vai voltar tem 4 páginas virtuais em seu *working set*. Quando esse mais antigo sair, somente 2 *frames* serão liberados, e, portanto, não será suficiente para o que quer voltar. Neste caso, será necessário o *swap out* de outro processo. Por isso, é necessário fazer *swap out* em um loop, até que o espaço livre seja suficiente.

```
// enquanto o WS que for entrar não couber, vai tirando outros processos
while(tamanho_ws > livres) {

     // swap out de alguém
     swap_out(-1);

     // atualiza quantidade de frames livres
     livres = 0;
     for(i = 0; i < FRAMES; i++) {
           if(frames[i].processo == -1) livres++;
     }
}
```

Dessa forma, conseguimos liberar o espaço necessário, e o processo pode voltar à memória.

3. Execução

3.1. Computador utilizado

Para os exemplos presentes neste relatório e testes internos, o programa foi compilado com o gcc 4.8.1, com o Makefile incluído nesta pasta, e foi utilizado o openSUSE 13.1 de 64 bits.

3.2. Compilação e execução do programa

O programa conta com um Makefile. Para a compilação, basta usar o make:

```
make
```

Na execução, está disponível o seguinte argumento:

```
./trabalho2 MODO
```

1. **MODO**: **basico** ou **completo**, indicando a forma de exibição da saída.

O modo **básico** exibe apenas texto, e não requer interação do usuário.

O modo **completo** exibe as tabelas em tempo real, e permite que seja acompanhada cada ação de forma muito mais fácil.

Mais detalhes nas seções a seguir.

3.3. Opções

Opções como o número de *frames*, processos, páginas virtuais, e o tamanho do *working set*, devem ser configuradas no header `comum.h`, antes da compilação, através das seguintes diretivas:

```
#define FRAMES           64
#define PROCESSOS        20
#define PAGINAS_VIRTUAIS 50
#define WORKING_SET      4

#define TEMPO_NOVAREQ    3
#define TEMPO_NOVOPROC   3
```

Também são configurados aí o tempo entre as requisições, e o tempo entre a criação de cada processo; os dois últimos itens, respectivamente.

3.4. Modo básico

O modo básico conta com uma interface com apenas texto, e não requer interação do usuário durante a execução.

Para executar no modo básico, utilize o argumento `basico` ao rodar o programa:

```
$ ./trabalho2 basico
```

No exemplo abaixo, são utilizados 50 frames e 20 páginas virtuais.

```
$ ./trabalho2 basico
*****
Por favor, maximize sua janela.
*****

[00:00] P00: iniciei

[00:00] P00: solicito PV12
    └ Page fault!
    └ PV12 alocada no frame 00 da MP

    **********************
    * Memória Principal *
    **********************
    Frame 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
    Proc  00 -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
    PV    12 -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -

    Frame 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
    Proc  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
    PV    -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -

    ****************************
    * Tabela de páginas de P00 *
    ****************************
    PV   00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19
    End* -  -  -  -  -  -  -  -  -  -  -  -  00 -  -  -  -  -  -  -
    *(número do frame onde a PV está)

    **********************
    * Working Set de P00 *
    **********************
    12 -  -  -


```

3.5. Modo completo

No modo completo, você tem um acompanhamento interativo, e é exibido o conteúdo da memória e tabela de página em "tempo real", permitindo que o gerenciamento de memória seja acompanhado com muito mais facilidade.

Para rodar no modo completo, utilize o argumento `completo`:

```
$ ./trabalho2 completo
```

Neste modo, para que os eventos sejam analisados de forma mais detalhada, os eventos não acontecem de forma automática, e é necessário pressionar a tecla **Espaço** para exibir o próximo evento.

Além disso, as tabelas são exibidas verticalmente, e é possível rolar utilizando certas teclas. Mais detalhes na tabela abaixo.

Teclas para interação no modo completo

Q	Sair
Espaço	Avançar para próximo evento
A e **Z**	Scroll na tabela de páginas
↑ e ↓ (setas direcionais)	Scroll na memória principal

4. Dificuldades

4.1. Exclusão mútua

No nosso programa, foi necessário o uso de exclusão mútua em duas ocasiões: primeiro, para garantir que o acesso à memória principal não fosse feito por mais de uma thread ao mesmo tempo. Depois, para impedir que mais de uma thread exibisse (imprimisse) informações ao mesmo tempo. Foi necessária a utilização de uma variável de exclusão mútua exclusiva para os `printf`s que precisavam estar juntos. Caso contrário, `printf`s de threads separadas acabavam ficando intercalados.

4.2. Sincronização

Outra dificuldade encontrada, foi a necessidade de haver uma sincronização entre threads. No **modo completo** (ver 3.5), para a implementação do modelo de "esperar até que uma tecla seja pressionada para prosseguir com a simulação", foi necessário o uso de uma variável de condição, e das funções `pthread_cond_wait` e `pthread_cond_broadcast`. Além disso, foi usada uma variável que indicava se a simulação pode continuar. Caso seu valor fosse 0, a thread em questão era bloqueada, e no momento que a tecla for pressionada, este valor é alterado e as threads são sinalizadas (acordadas).

4.3. Exibição da informação

Sem dúvida, a maior dificuldade deste trabalho foi bolar uma solução para mostrar uma grande quantidade de informações de forma simples e fácil de ver. Devido à grande quantidade de passos que o programa deve dar, alterando informações em cada um deles, a saída acaba ficando muito grande e pode se tornar confusa. De nada adiantaria um simulador funcionando perfeitamente se o resultado disso não pudesse ser exibido de forma clara e eficiente.

Após muito pesquisar e refletir, chegamos à conclusão de que apenas uma sequência de texto não seria suficiente. Decidimos, então, elaborar uma interface com informações que são alteradas durante a execução, e que permite uma interação do usuário: o que chamamos de **modo completo** - ver 3.5. (porém, caso seja necessário, também está disponível um modo com apenas texto: ver 3.4)

5. Conclusão

A partir destse trabalho, foi possível vivenciar a funcionalidade de um gerenciador de memória virtual implementado pelo sistema operacional. Ao implementarmos um simulador por conta própria, fomos dirigidos a pensar em todas as possibilidades de execução, fazendo com que nos atentássêmos a detalhes importantes que antes haviam passado despercebidos.

Sem dúvidas, foi indispensável um conhecimento de concorrência, para ser feita sincronização e evitar problemas como a condição de corrida, que apareceu mais de uma vez durante a solução do problema.

Por fim, conseguimos entender a importância e as dificuldades do gerenciamento de memória virtual em um sistema operacional. Desde os passos mais simples, como uma básica inserção em uma lista, até os problemas de concorrência originados pela necessidade de atender a todos os processos "simultaneamente".

6. Bibliografia

6.1. Solução do problema proposto

- Funções da biblioteca pthreads:

  Páginas de manual oficiais. Podem ser acessadas com o comando `man` (`man <nome>`).

- Gerenciamento de memória:

  Material do curso, disponível em http://www.dcc.ufrj.br/~valeriab/mab366.php

6.2. Interface

Apresentamos, aqui, as fontes consultadas para a implementação do *modo completo* (ver 3.5), que, apesar de não terem relação com a solução do problema proposto em si, foram cruciais no conhecimento adquirido para criação da interface interativa do nosso programa.

- Cores e formatação de texto no terminal:
  http://misc.flogisoft.com/bash/tip_colors_and_formatting
  https://wiki.archlinux.org/index.php/Bash/Prompt_customization

- Posicionamento de texto no terminal:
  https://rosettacode.org/wiki/Terminal_control/Cursor_positioning#C.2FC.2B.2B

- Caracteres em Unicode para desenho das tabelas e setas:
  https://en.wikipedia.org/wiki/Box-drawing_character
  https://en.wikipedia.org/wiki/Arrow_%28symbol%29#Arrows_in_Unicode
  https://en.wikipedia.org/wiki/Geometric_Shapes

- Capturar tecla pressionada sem a necessidade de Enter, e sem exibir o caractere:
  http://stackoverflow.com/a/7469410

- Determinar o tamanho da janela do terminal:
  http://stackoverflow.com/a/263900

- Capturar SIGINT para finalizar a interface corretamente:
  http://www.thegeekstuff.com/2012/03/catch-signals-sample-c-code/