Documentação do lrusimul

INF01142 – Sistemas Operacionais I – Turma B – Prof. Alexandre Caríssimi

versão: 14 de junho de 2012

1. Identificação do Grupo

- Luiz Gustavo Frozi de Castro e Souza - Cartão 96957
- Pedro Henrique Frozi de Castro e Souza - Cartão 161502

2. Descrição da Plataforma de Desenvolvimento

Foram usados dois computadores com configurações diferentes para realizar o desenvolvimento, o lrusimul foi compilado e testado e dois ambientes linux virtualizados. Foi também utilizado para desenvolvimento o versionador SVN, utilizando um repositório do Google Code, disponível em: http://code.google.com/p/lrusimul2012-1/.
A tabela abaixo faz um detalhamento dos ambientes de desenvolvimento e execução:

	Computador 1	**Computador 2**
Tipo de Processador	Intel Core 2 Duo T5450 1,67GHz	AMD Athlon(tm) X2 Dual Core Processor L310 1.2 GHz
Número de Cores	2	2
Suporte a HT?	Sim	Sim
Memória RAM	4GB	3GB
Sistema Operacional (Host)	Windows 7 Professional de 64bits	Windows 7 Professional de 64bits
Foi usado Ambiente Virtualizado?	Sim	Sim
Máquina Virtual Utilizada	Oracle VM VirtualBox 4.1.6 r74713	Oracle VM VirtualBox 4.1.12 r77245
Sistema Operacional (Guest, VM)	Linux de 32bits	Linux 64bits
Distribuição Utilizada	Debian 6.0.3 com LXDE	Ubuntu 12.04 com Xubuntu
Versão do Kernel	2.6.32-5-686	3.0.0-17
Versão o GCC	4.4.5	4.6.1

3. Descrever a estrutura de dados empregada para encadear as páginas no LRU para permitir a análise delas como se fosse o movimento de um ponteiro de relógio (algoritmo segunda chance).

Primeiro criamos uma estrutura que representa uma célula de memória (quadros),

conforme podemos ver abaixo:

```
typedef struct tmemoria {
    int  pid;     // PID do Processo cuja página ocupa a célula da memória
    int  pagina;  // Página do processo que está ocupando a célula
      int   bitRef;   // Bit de Referência, indica se a página foi ou não acessada
recentemente
    int  bitSujo; // Bit de Sujo, indica se a página foi alterada recentemente
} tmemoria;
```

Essa é a estrutura que define uma única célula de memória, para criar toda a listagem de memória, de acordo com *MEMSIZE n*, é alocado um vetor com *n* posições de elementos do tipo *tmemoria*. Como se pode ver abaixo na função *criaMemoria()* e *memSize()*:

```
tmemoria* criaMemoria(int n) {
    ...
    tmemoria* aux;
    aux = (tmemoria*)malloc(sizeof(tmemoria) * n);
    ...
    return aux;
}
…
Memoria = criaMemoria(size);
```

A partir de então a memória é acessada como um vetor comum. Para fazer o movimento de relógio basta incrementar o índice da posição atual da memória, representado pela variável *clockPointer*.
Quando há um *PageFault* o algoritmo tratador primeiro tenta alocar as páginas em algum quadro livre, se não houver quadros livres, o algoritmo de substituição do relógio é acionado.

4. Descrever a estrutura da tabela de páginas.

Os processos são armazenados em uma fila encadeada, cada processo possui um vetor que representa a sua estrutura de página, com a localização (swap ou memória) e as estatísticas solicitadas (acessos, número de pagefaults e número de substituições, conforme podemos ver na estrutura abaixo:

```
typedef struct page {
    int  pagina;
    int  acessos;
    int  nroPageFault;   // será no mínimo 1, quando a página é lida pela primeira vez
    int  nroSubst;       // Quantas vezes esta página foi escolhida como "vítima",
                         // caso não existam mais páginas disponíveis na memória RAM
     char local;            // Indica se a página está na memória 'M' ou no swap 'S',
quando o
                         // processo é criado todas as páginas estão no swap
} page;
```

Cada processo é representado com uma estrutura do tipo *process* conforme pode-se ver abaixo:

```
typedef struct process {
    int             pid;
    int             size; // Quantidade de páginas
    int             estado;
    page*           paginas;
    struct process* prox;
} process;
```

Para cada processo o número total de páginas é alocado dinamicamente como uma vetor no campo *página* na função *insere()*, conforme podemos ver abaixo:

```
process* insere(process* l, int pid, int size) {
    ...
    /* Gera um array que é um conjunto de páginas para o processo */
    paginas = (page*) malloc((sizeof(page) * size));
    ...
    /* Gera um novo elemento processo na lista */
    novo = (process*) malloc(sizeof(process));
    novo->pid  = pid;
    novo->size = size;
    novo->estado = INICIALIZADO;
    novo->paginas = paginas;
    ...
}
```

5. Explicitar, em função dos bits de referência e modificação, qual foi a ordem de preferência para escolher uma página vítima (modificada e acessada; acessada e não modificada; não acessada e modificada; não acessada e não modificada). Explique o porque, as vantagens e as desvantagens dessa escolha. Utilize, também, os seus casos de teste para justificar a resposta.

De acordo com Silberchatz[1] temos as seguintes combinações possíveis para os bits de Referência e Sujo, bem como suas prioridades para a substituição:

Classe Prioridade	Bit Ref.	Bit Sujo	Descrição
1	0	0	Não Acessado, Não Modificado (melhor caso para substituir)
2	0	1	Não Acessado, Modificado
3	1	0	Acessado, Não Modificado
4	1	1	Acessado, Modificado (melhor caso para substituir)

Segundo o autor a melhor forma é substituir a primeira página encontrada da classe de menor prioridade não vazia, ou seja, o melhor caso para substituir é uma página com *Bit Ref. = 0* e *Bit Sujo = 0*. O relógio pode dar várias voltas até achar uma página disponível para substituir.
No caso implementado, o algoritmo do relógio pode dar até três voltas até encontrar uma página possível de substituição, pois na primeira volta ele procura pelo par *Bit Ref. = 0* e *Bit Sujo = 0*, e vai zerando os bits de referência das outras classes (dando uma segunda chance para as páginas recentemente acessadas), na segunda volta ele também busca par *Bit Ref. = 0* e *Bit Sujo = 0* (podendo pegar um processo que na primeira volta estava na classe de prioridade 3), na terceira volta ele irá buscar a primeira página com *Bit Ref. = 0*, ignorando o *Bit Sujo*. Isso foi necessário para que o algoritmo não ficasse tentando buscar indefinidamente uma página na primeira classe de referência.
As vantagens desse algoritmo são:

- Simplicidade de implementação
- Não gera *overhead* de processamento para ordenar a lista a cada acesso de

[1] SILBERCHATZ, Abraham. Operating Systems Concepts. 7ª Ed. John Wiley & Sons. Inc - Hoboken, NJ, 2005. Pg. 337-338.

página
- Usa pouca quantidade de memória, pois, no máximo, dois bits precisam ser alterados a cada acesso de página
- Reduz o número de requisições de E/S, pois prioriza para substituição páginas não alteradas, esse recursos pretende aumentar a velocidade.

As desvantagens são:

- Pode ter que varrer várias vezes a lista de páginas até encontrar uma adequada para a substituição
- A página escolhida pode não ser, necessariamente, a mais antiga.

6. Descrição sobre o que está funcionando e o que não está.

O código está compilando e gerando o binário sem erros de compilação.
Durante algumas execuções, principalmente no ambiente do **Computador 1** obtivemos alguns erros de *Segmentation Fault*, fato não observado no ambiente do **Computador 2**, algo semelhante com o que ocorreu no primeiro trabalho.
Com os arquivos de testes que criamos pudemos observar o comportamento do algoritmo do relógio com bit Sujo (LRU 2ª Chance Modificado) condiz com o esperado, conforme descrito no item anterior.
O arquivo de teste **não** pode ter linhas em branco.

7. Metodologia de Teste Utlizada

Utilizamos a metodologia de teste com valores inválidos, que consiste em executar as funções com os valores limites (valores de entrada e quantidade de processos). Não foi utilizado nenhum “debugger” específico, verificamos o retorno das funções no próprio terminal.
Também foram utilizados os arquivos de teste e verificados os logs de saída dos arquivos para testar cada função.

8. Dificuldades encontradas e soluções

A maior dificuldade foi depurar erros de *Segmentation Fault* resultantes de manipulações de variáveis do tipo ponteiro usadas para gerenciar a lista de processos. Como não utilizamos uma IDE com debugger integrado, esses erros aumentaram a dificuldade na hora de corrigir o código.

Uma coisa que notamos é que nas máquinas virtuais obtivemos constantemente erros de *Segmentation Fault*, principalmente quando o estado era salvo e restaurado, ao invés de reiniciado.