UFRJ

Universidade Federal do Rio de Janeiro

MAB366

Sistemas Operacionais I

# Gerência de memória com paginação


| *Aluno:* | *DRE:* |
|---|---|
| João Guio | 114094151 |
| Renato Pontes | 113131049 |
| Tiago Montalvão | 114043443 |


12 de junho de 2017

# Sumário

# 1 Introdução

Este documento propõe-se a explicar em detalhes a implementação de um sistema de gerência de memória com a técnica de paginação. Como não há a utilização de memória virtual, um processo deve estar todo na memória para ser executado.

Foi construído um simulador na linguagem C que permite explorar com detalhes o funcionamento de um sistema real de gerência de memória com paginação.

# 2 Compilação do simulador

O simulador pode ser compilado utilizando-se o *Makefile* fornecido junto ao código-fonte. Para compilar, basta executar:

```
$ make
```

Será gerado o executável `memory-manager`.

# 3 Utilização do simulador

Ao executar a aplicação, serão exibidas algumas informações sobre o ambiente simulado e em seguida exibida uma linha de comando, como a seguir:

```
============================================================
 Informações do sistema
------------------------------------------------------------
  Modo DENY
  Tamanho da memória principal: 65536 bytes
  Tamanho do frame: 1024 bytes
  Número de frames: 64
  Processos na memória: 0
  Total de processos: 0
============================================================

[00:00]>
```

O comando `h` ou `help` exibirá os comandos válidos.

Note que o simulador indica estar no modo padrão *DENY*. Neste modo, o simulador se recusará a criar processos se não estiver disponível memória suficiente. No modo *WAIT*, os processos esperam em uma fila até que sejam liberados frames suficientes para alocar as páginas do processo que se encontra na frente da fila. Para inciar o simulador no modo *WAIT*, forneça a *flag* **wait** na linha de comando:

```
$ ./memory-manager wait
```

A simulação fornece um alto grau de controle sobre a passagem do tempo. Para continuar a simulação por *t* segundos, utilize o comando `step` (ou apenas `s`):

```
[00:00]> step 10
```

Durante a simulação, notificações serão impressas para indicar que ocorreu a criação de um processo, alocação de frames na memória principal e outros eventos relevantes.

Para alocar um processo na memória de tamanho *b* bytes, utilize o comando `load` (ou `l`). Se *b* não for fornecido, o tamanho do novo processo é aleatório, mas é garantido que ele não ocupará mais espaço do que o disponível:

```
[00:11]> load 3000
      Novo processo P000 criado (3000 B, 3 páginas, 7 seg)
      P000 está carregado na memória.
      41 frames livres
```

Note que quando um processo é criado, ele recebe um PID (sequencial a partir de 0) e também um tempo de execução pseudo-aleatório. A medida que a simulação prossegue (utilizando-se o comando `step`), os processos terminam e são removidos da memória.

Quando programas estão carregados na memória e a simulação está em curso, eles fazem vários acessos a memória utilizando endereços relativos. Os endereços serão traduzidos para endereços físicos por meio da tabela de páginas do processo e impressos na tela. Se um programa tentar acessar um endereço de memória inválido, ocorre uma falha de segmentação. O comando `ref` pode ser usado para provocar uma referência a memória manualmente, fornecendo-se um *pid* válido e um endereço relativo:

```
[00:11]> ref 6 2059
      P006 referencia endereço 2059
      Endereço físico 60427 (frame: 59, offset: 11)
```

Por fim, os comandos `mem` e `paget` permitem que o usuário verifique o estado da memória principal e da tabela de páginas de um processo, respectivamente.

```
[00:11]> ref 6 2059
      P006 referencia endereço 2059
      Endereço físico 60427 (frame: 59, offset: 11)
```

Para sair do simulador, basta executar o comando `quit` (ou apenas `q`).

# 4 Implementação do simulador

A implementação está dividida em diversos arquivos, *headers* e *sources*. A seguir descrevemos alguns elementos importantes do código-fonte.

## 4.1 Parâmetros globais

Em `global.h` são definidos diversos parâmetros chave para a execução do simulador, como o tamanho da arquitetura, tamanho da memória, tamanho de cada frame, etc. Por padrão, a arquitetura é de 16 bits, tendo a memória 64KB, e um frame pode ser endereçado internamente por 10 bits, consequentemente cada frame tendo tamanho de 1KB.

## 4.2 Memória

A memória é implementada como uma struct que possui três membros:

- `pid_t *used_frames`: uma lista de todos os frames da memória, com o pid de qual processo tem sua página naquele frame. Se não houver tal processo, a posição contém o valor -1.
- `size_t free_frames`: a quantidade de frames livres.
- `size_t processes`: a quantidade de processos carregados na memória.

## 4.3 Tabela de Processos

A tabela de processos é uma estrutura global utilizada para gerenciar o estado dos processos criados. Cada entrada na tabela referencia um processo existente. É também do índice de um processo na tabela de processos que é gerado o PID correspondente.

## 4.4 Tabela de Páginas

A tabela de páginas é implementada como uma struct que possui dois membros:

- `frame_t *table`: uma lista de frames associados a cada página.
- `size_t size`: o tamanho da tabela.

Para cada processo é associada uma tabela de páginas, que indica em que frames da memória suas páginas estão alocadas. No modo *wait*, é possível que a tabela de páginas de um processo esteja vazia (quando o processo está esperando na fila).

## 4.5 Processo

Cada processo é implementado com uma struct com diversos membros:

- `pid_t pid`: um PID associado ao processo.
- `size_t proc_size`: o tamanho, em bytes, do processo.
- `int32_t exec_time`: o tempo de execução total do processo (usado para a simulação).
- `int32_t start_time`: o tempo em que o processo começa a ser executado.
- `Page_table *page_table`: a tabela de páginas do processo.

## 4.6 Estruturas auxiliares

O simulador utiliza a implementação de fila no modo *wait*, colocando os novos processos que não cabem na memória em uma fila de espera. Para este fim, foram implementadas uma lista duplamente encadeada e uma fila.

# 5 Conclusão

Este trabalho nos permitiu ter uma noção mais ampla do quão complexo é o gerenciamento de memória, mesmo em um nível mais simples.

Como a paginação simples é um passo intermediário entre o particionamento de tamanho fixo e a memória virtual, é interessante notar quais problemas a técnica é capaz de resolver, e que problemas são deixados em aberto.

Criando alguns processos e avançando na simulação, podemos notar que em determinados momentos as páginas de um processo serão alocadas de forma não contígua. Notamos também que qualquer *frame* vago na memória principal pode ser alocado para qualquer processo, e portanto não existe fragmentação externa. Como o tamanho total de um processo não precisa ser um múltiplo da página, ainda existe fragmentação interna na última página que é alocada.

Um problema grave que não é solucionado com a paginação é a necessidade de ter um programa completamente carregado na memória. Uma implementação de gerência de processos, por exemplo, seria extramente cara, pois uma troca de contexto consistiria em remover um programa (possivelmente muito grande) completamente da memória e em seguida carregar um outro programa inteiro. Como se não bastasse, um programa pode ser maior do que toda a memória disponível no sistema.

# Referências

[1] William Stallings. *Operating Systems: Internals and Design Principles*