Relatório do EP3

MAC0422 – Sistemas Operacionais – 2s2017

Bruno Ferrero (3690142), Rodrigo Souza (6800149), Taís Pinheiro (7580421)

1 Introdução

O simulador de gerência de memória foi implementado em Python 2.7 [1]. A escolha de Python foi consenso no grupo, devido ao conhecimento da linguagem por todos os membros do grupo e principalmente pelos recursos da linguagem em lidar com diferente tipos de dados/objetos. O livro de referência utilizado para as dúvidas com a linguagem foi Martelli *et al.* (2008).

Como estratégia de organização de projeto, buscamos modularizar o código. A seguir a apresentamos todos os arquivos entregues (`ep3.tar.gz`) e em seguida detalharemos a modularização do código:

```
$ tree -d
|--- src
- |--- ep3.py
- |--- execute.py
- |--- linkedlist.py
- |--- memory.py
- |--- paging.py
- |--- processo.py
- |--- run_tests.sh
|--- doc
- |--- relatorio.tex
- |--- relatorio.pdf
- |--- apresentacao.pdf
|--- LEIAME.txt
```

- **ep3.py**: Implementação do modo terminal. Ele é chamado de dentro da nossa função `main()`, que também poderá fazer uma chamada de simulação em *batch mode*. Aqui também está implementada a função que carrega um arquivo *trace*, que gerará uma lista de processos que será

[1] `https://docs.python.org/2/reference/index.html` [Acessado em Nov/2017]

posteriormente transformada em uma lista de execução ordenada pelo tempo. Um ponto a destacar é que já na leitura do trace são criadas as memórias física e virtual com a classe Memory. São criadas também as listas ligadas para controle de espaço livre das memórias.

- **execute.py**: Aqui que de fato ocorre a simulação. A simulação pode ser resumida com as seguintes etapas:

  1. A partir da lista gerada com a leitura do *trace*, cria-se uma nova lista a partir do *trace* com o seguinte formato:

     ```
     [ t, ACAO, tam/pos memoria, <processo>],
     ```

     em que `t` é o tempo de chegada da `ACAO`; a `ACAO` pode ser: `REMOVER` um processo, `ALOCAR` um espaço na memória virtual de tamanho `tam`, ACESSAR uma posição de memória `pos` ou `COMPACTAR` as memórias;

  2. Essa nova lista (lista de execução) é ordenada em ordem crescente de `t`, como mostrado na Figura 2;

  3. Inicia-se a simulação: Temos um *loop* atualizando um *clock* e iterando na lista de execução. Toda vez que o *clock* == *t*, verifica-se a ação desse item da lista (Figura 1);

```
26      while True:
27          if clock == execucao[0]:
28              if isinstance(execucao[1], basestring):
29                  #print execucao
30                  if execucao[1] == 'COMPACTAR':
31                      print 't: ' + str(execucao[0]) + ' COMPACTAR'
32                      compactar()
33                  elif execucao[1] == 'REMOVER':
34                      # Aqui vamos tratar o caso de remover um processo
35                      mem_virtual.remover_processo(execucao[3],mem_fisica)
36                  elif execucao[1] == 'ALOCAR':
37                      # Chegou um novo processo: manda pra memoria virtual
38                      print "t: " + str(execucao[0]) + ' ' + execucao[1] + ' para ' + execucao[3].nome
39                      # disparando o cronometro
40                      t_i = time.time()
41                      if espaco == 1:
42                          mem_virtual.best_fit(execucao[3])
43                      elif espaco == 2:
44                          mem_virtual.worst_fit(execucao[3])
45                      elif espaco == 3:
46                          mem_virtual.quick_fit(execucao[3])
47                      #carrega processo na tabela de pagina (mem virtual)
48                      mem_virtual.set_pagina_tabela (execucao[3])
49                      # para o cronometro e soma o tempo
50                      t_f = time.time()
51                      timetotal += (t_f - t_i)
52                  elif execucao[1] == 'ACESSO':
53                      # disparando o cronometro
54                      t_i = time.time()
55                      # executa ACESSO a memoria
56                      executa(execucao,substitui)
57                      # para o cronometro e soma o tempo
58                      t_f = time.time()
59                      timetotal += (t_f - t_i)
60                  count += 1
```

Figura 1: Trecho com o *loop* da simulação.

- **linkedlist.py**: Nesse modulo está implementado a classe que vai criar e manipular a lista encadeada. Os nós da lista são da seguinte forma:

```
class Node(object):
    def __init__(self, data, inicio, tamanho, next_node):
        self.data = data                # L se eh um noh livre e P se tem processo
        self.inicio = int(inicio)       # Em que posicao comeca
        self.tamanho = int(tamanho)     # Tamanho ocupado
        self.next_node = next_node      # Proximo noh
        self.previous_node = None       # No anterior
```

O objetivo dessa classe é guardar os espaços livres e ocupados a memoria.

```
t: 1 ALOCAR para P0
t: 1 ACESSO de P0 em: 1
t: 2 ACESSO de P0 em: 2
t: 2 ALOCAR para P1
t: 2 ACESSO de P1 em: 1
t: 3 ACESSO de P0 em: 3
t: 3 ACESSO de P1 em: 2
t: 3 COMPACTAR
t: 4 ACESSO de P1 em: 4
t: 4 ALOCAR para P2
t: 4 ACESSO de P2 em: 1
t: 5 REMOVER P1 com inicio em p: 8
t: 5 ACESSO de P2 em: 2
t: 6 ACESSO de P2 em: 3
t: 7 ACESSO de P2 em: 4
t: 7 REMOVER P2 com inicio em p: 16
t: 10 REMOVER P0 com inicio em p: 0
```

Figura 2: Exemplo de uma lista de execução ordenada em ordem crescente de tempo `t`.

- **memory.py**: Nesse modulo estão todas as implementações referentes à memoria. Temos uma classe Memory:

```
class Memory:
    def __init__ (self, total, s, p, filename):

        self.tamanho = total
        self.s = s
        self.p = p
        self.arquivo = filename
        self.tabela = []
        #monta a tabela de paginas para esta memoria
        self.set_tabela()
        #lista que vai espelhar a situacao da memoria. Vamos escrever essa lista no
            arquivo
        self.vetor = [-1] * self.tamanho
        #lista ligada para controle do espaco livre da memoria
        self.lista = LinkedList('L',0,self.tamanho,None)
        #Abre o arquivo para a memoria
        self.memfile = open (filename,'wb')
        #define tamanho do arquivo binario em bytes
        # b signed char => 1 byte
        self.memfile.write(pack(str(self.tamanho)+'b',*self.vetor))
        self.memfile.flush()
```

É aqui também que estão implementados os algoritmos de gerência de memórias, que serão detalhados mais a frente.

- **paging.py**: Aqui estão os algoritmos de substituição de páginas e também temos uma classe Page:

```
#classe page: representa uma pagina na tabela de paginas
class Page:
    def __init__ (self, inicio, p):
        self.tamanho = p                #define o tamanho da pagina
        self.inicio = int(inicio)       #onde a pagina comeca
        self.fim = int(inicio) + p - 1  #onde a pagina termina
        self.presente = 0               #presente = 1 se a pagina esta mapeada na
            memoria fisica e 0 caso contrario
        self.mapeada = -1               #link recebe o indice da pagina que esta
            linkada na memoria fisica
        self.procId = -1                #O id do processo dono desta pagina
        self.tAcesso = -1
        self.m = 0
        self.r = 0
        self.countLRUv4 = 0
```

- **processo.py**: Nesse modulo temos a classe processo e seus métodos para lidar com suas variáveis.

```
class Processo:
    current_pid = 0

    def __init__(self, t0, tf, b, nome, acessos):
        self.t0 = int(t0)         # tempo de chegada do processo
        self.tf = int(tf)         # tempo de termino do processo
        self.b = int(b)           # tamanho de memoria requisitado pelo processo
        self.nome = nome          # string com nome do processo
        self.ocupa = 0            # tamanho ocupado pelo processo devido a uniadade de
            alocacao (tamanho marcado com seu pid)
        self.reserva = 0          # tamanho reservado pelo processo devida ao tamanho
            da pagina
        self.base = -1            # endereco base do processo
        self.limite = -1          # endereco limite do processo
        self.setpid()             # define pid do processo
        self.setAcessos(acessos)  # pares de acesso a memoria (pn,tn)
```

- **run_tests.sh**: Shell-script que gera os resultados para todas as combinações de algoritmos de espaço e substituição.

A seguir são descritas as implementações dos algoritmos de gerência de espaço livre e dos algoritmos de substituição de páginas. Todas as implementações foram feitas com base nas notas de aulas e com referências de Tanenbaum (2009).

Algumas considerações gerais sobre a implementação são comentadas a seguir:

- Quando duas ações (alocação de memoria na chegada de um processo, acesso a uma posição, etc.) acontecem no mesmo instante de tempo, é mantido a ordem em que a ação aparece no trace;
- o `clock` (que conta o tempo na simulação) é um contador que incrementa em um *loop*;
- os tamanhos das memórias são em *bytes*, cada *byte* representa uma posição da memória;
- para a conversão dos bytes foi utilizados as funções `pack` e `unpack` da biblioteca `struct` do Python.
- os tempos de busca de espaço livre foram feitos com a função `time()` da biblioteca `time` em trechos do código em que eram aplicados os algoritmos gerência de memória.

1.1 Algoritmos de gerência de espaço livre

Os algoritmos se baseiam na lista ligada para gerência de espaço livre da memória virtual. Assumimos para estes algoritmos que sempre existirá espaço na memória virtual para quaisquer processos que cheguem para execução. Cada nó da lista ligada para a gerência de espaço livre tem a configuração como descrita acima em *class Node(object)*

1.1.1 Best-Fit

```
def best_fit(self, p):
```

Este algoritmo é implementado como procedimento da classe Memory. Ele recebe um processo *p* e varre toda a lista de controle de espaço livre da memória virtual em busca do espaço livre que tenha o menor tamanho suficiente para acomodar todo o processo, levando em consideração o número de páginas que o processo em questão deve ocupar. Quando este processo é alocado, definem-se as posições *base* e *limite* para o mesmo.

1.1.2 Worst-Fit

```
def worst_fit(self, p):
```

Análogo *Best-Fit*este algoritmo é também implementado como procedimento da classe Memory. Ele recebe um processo *p* e varre toda a lista ligada de controle de espaço livre da memória virtual em busca do espaço livre que tenha o maior tamanho para acomodar o processo. Com base na posição alocada, defini-se a *base* e *limite* para o processo.

1.1.3 Quick-Fit

```
def quick_fit(self, p):
```

A ideia deste algoritmo é criar listas que mapeiem os 3 espaços de alocação mais requisitados pelo processos. Não conseguimos terminar a implementação deste algoritmo.

1.2 Algoritmos de substituição de páginas

A nossa implementação da parte de substituição de paginas ocorre sempre depois olhar toda a tabela de paginas da memoria física e verificar que não há um espaço para alocar uma nova página. Se houver um lugar, a nova página ocupa esse espaço. Caso contrário, entram os algoritmos de substituição de páginas para escolher qual página será retirada da memória física. Quando o código usa algoritmo de substituição além de escolher a página que vai ser removida é preciso fazer todo o remapeamento das tabelas de páginas e marcar (de acordo com as especificidades de cada algoritmo) a nova pagina que entrou na memória. O ideia do código de remapeamento de tabelas é comum a todos os algoritmos de substituição.

A seguir são apresentados alguns detalhes das implementações dos algoritmos de substituição de página.

1.2.1 Optimal

Não conseguimos terminar a implementação desse algoritmo a tempo.

1.2.2 First in, First Out

Toda vez que uma página é mapeada na memória é guardado na página o `clock`, no qual ela entrou na memória. Usa-se esse valor para descobrir qual é a pagina mais velha e com isso remove-se página que foi mapeada a mais tempo.

1.2.3 Least Recently Used (Segunda Versão)

A implementação procurou ser fiel ao algoritmo apresentado em sala de aula, no qual utiliza-se uma matriz quadrada de tamanho igual ao numero de paginas que cabe na memória. Toda vez que uma pagina é mapeada ou acessada na memória física marca-se a linha e em seguida a coluna matriz correspondente a posição da pagina. Quando ocorre um Pagefault, busca-se nessa matriz a pagina que foi menos referenciada.

```
# devolve a pagina menos acessada de acordo com a matriz de acesso matriz_LRUv2
def LRUv2_pagina (npaginas):
    global matriz_LRUv2
```

```
        maior = int(''.join(npaginas*['1']),2)
        pagina = 0
        for i in range(npaginas):
            x = int(''.join(matriz_LRUv2[i]),2) # converte a linha binaria para int
            if x < maior:
                pagina = i
                maior = x
        return pagina
```

1.2.4 Least Recently Used (Quarta Versão)

Na implementação do LRUv4 nós usamos uma variável extra da nossa classe Pagina :

```
self.countLRUv4 = 0
```

que é atualizada a cada acesso ou mapeamento com o seguinte método:

```
def set_countLRUv4(self,k):
        self.countLRUv4 += 1
        self.countLRUv4 =  self.countLRUv4%k
```

Basicamente, a variável k vai definir a "idade"máxima da página, equivalente ao número de bits, da implementação apresentada em classe. Quando ocorre um Pagefault, retira-se a página com o menor `countLRUv4`. O valor k foi fixado em $k = 6$ em todo nosso código.

2 Resultados

Para gerar os resultados aqui apresentados foi utilizado um *Shell-script* (`run_tests.sh`) para gerar 30 execuções em modo *batch* de um trace. Em toda execução o `ep3.py` sempre vai imprimir o tempo total que levou para fazer todas as buscas por um espaço livre na memória durante uma simulação e o número de ocorrências de *PageFaults*. Esse script pode ser executado para qualquer trace, com a seguinte chamada:

```
$ ./run_tests.sh trace.txt
```

Um exemplo de saída do script para um arquivo *trace* qualquer é mostrado na Figura 3. A média e o desvio padrão foram calculados usando o awk.

```
media_std=$(
    for i in  ${results}; do echo $i;done |
        awk '{sum+=$1; sumsq+=$1*$1}END{print sum/NR " " sqrt(sumsq/NR - (sum/NR)**2)}'
)
```

A Figura 4 resume todos os testes feitos com os algoritmos implementados. Nessa figura são apresentados a média e o desvio padrão de 30 simulações de um arquivo trace teste. Esse *trace* foi produzido com intuito de tentar evidenciar algumas características dos algoritmos. Cada barra da

```
> chuva :ep3 $ ./run_tests.sh trace1.txt
best-fit & First In, First Out
media | std
0.00112181 0.000139746
Pagefault:  5
best-fit & LRUv2
media | std
0.00126387 0.000276424
Pagefault:  5
best-fit & LRUv4
media | std
0.00121386 0.000253529
Pagefault:  5
worst-fit & First In, First Out
media | std
0.00109519 0.000136653
Pagefault:  5
worst-fit & LRUv2
media | std
0.00115078 0.000177338
Pagefault:  5
worst-fit & LRUv4
media | std
0.00112385 0.000161002
Pagefault:  5
```

Figura 3: Média e desvio padrão de 30 execuções de um arquivo *trace* para a combinação de todos os algoritmos de gerência de espaço com os algoritmos de substituição de paginas implementados.

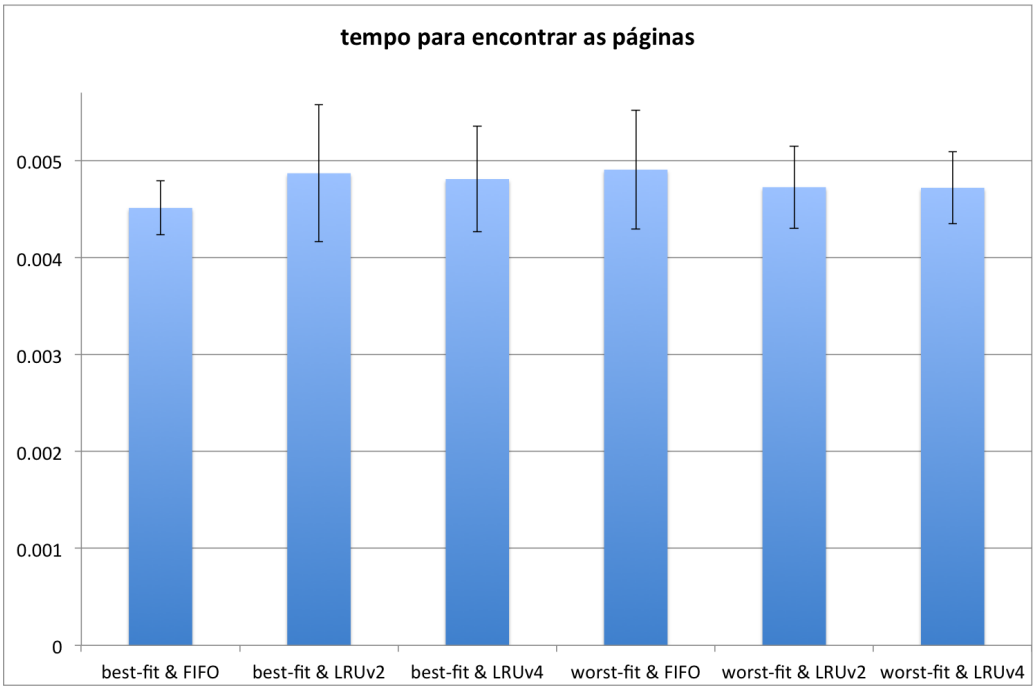

Figura 4: Média e desvio padrão de 30 execuções de um arquivo *trace* para a combinação de todos os algoritmos de gerência de espaço com os algoritmos de substituição de paginas implementados.

Figura 4 representa uma combinação de um algoritmo de gerência de espaço com um algoritmo de substituição de página.

A Figura 5 mostra as ocorrências de Pagefaults para o arquivo trace testado. A quantidade de Pagefault não vai variar pra diferentes rodadas de um mesmo trace. Dessa forma, os resultados desse análise não possuem o desvio padrão.

Os resultados mostrados nas Figuras 4 e 5 não conseguiram evidenciar forma muito clara as diferenças de comportamento entre as combinações de algoritmos. Acredita-se que um dos motivos principais para tal comportamento tenha sido a presença de alguma operação "gargalo"que tenha sido feita no trecho de codigo que tentamos cronometrar o tempo de busca de página. Como as implementações dos algoritmos `quick-fit` e *Optimal* não conseguiram ficar prontas a tempo de confeccionar os resultados, não mostramos os resultados com essas combinações. E muito provavelmente seria possível ver ganhos de tempo com o uso desses algoritmos.

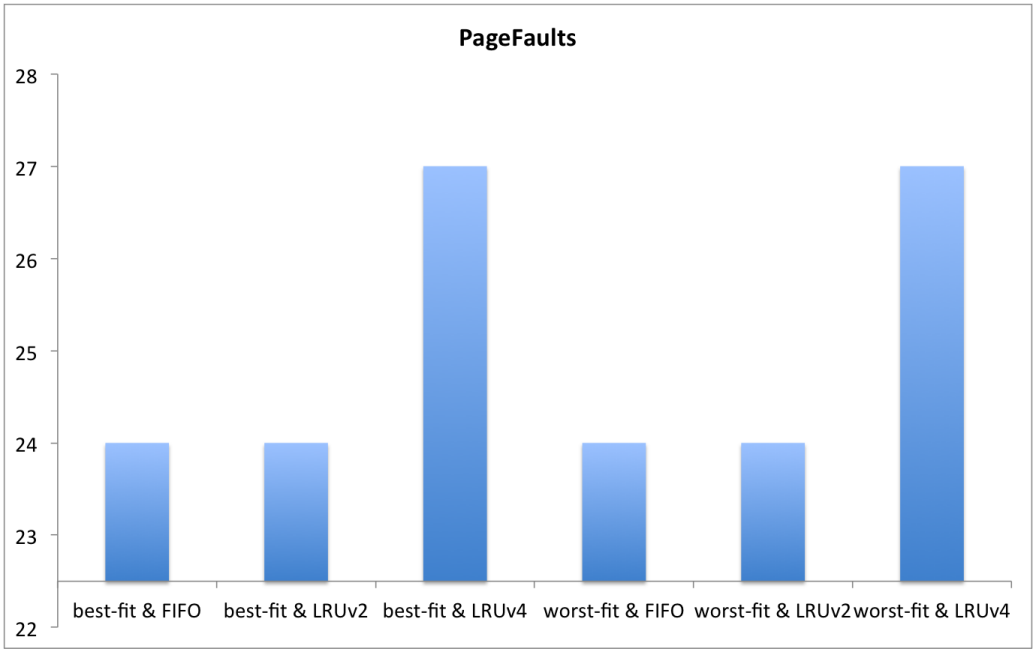

Figura 5: Quantidade de Pagefaults para todas as combinações dos algoritmos de gerência de espaço com os algoritmos de substituição de paginas implementados.

3 Divisão das tarefas de implementação

O repositório no qual foi desenvolvido o projeto encontra-se em `https://github.com/rasouza/ep3-so/`. Algumas informações dos *logs* do repositório (`commits` e linhas adicionadas) são mostradas a seguir:

```
$ git shortlog -sn
```

```
    41  brferrero
    30  taispin
     3  Rodrigo Alves
     3  Tais
$ git log --author="<USER>" --pretty=tformat: --numstat | awk '{inserted+=$1} END {printf
    "Commit stats:\n- Lines added (total)....  %s\n", inserted}'
USER=brferrero (Bruno Ferrero)
Commit stats:
- Lines added (total)....  802
USER=Rodrigo Alves
Commit stats:
- Lines added (total)....  167
USER=taispin (Tais Pinheiro)
Commit stats:
- Lines added (total)....  739
```

A participação no início do projeto foi bastante intensa por parte de todos os membros do grupo. Houve bastante discussão para podermos entender o que deveria ser implementado e como. Ao final do projeto foi possível notar que grande parte das decisões de projeto tomadas de início foram respeitadas e acertadas.

Referências

Martelli, A., Ravenscroft, A., Ascher, D., 2008. Python Cookbook, 2nd Edition. "O'Reilly Media, Inc.".

Tanenbaum, A. S., 2009. Modern Operating Systems, 4Ed. Pearson Education, Inc.