EXERCÍCIO-PROGRAMA 3: POLÍTICAS DE ALOCAÇÃO

SISTEMAS OPERACIONAIS — MAC0422

RENATO LUI GEH
NUSP: 8536030
GUILHERME FREIRE
NUSP: 7557373

1. Introdução

O EP foi feito em um Minix 3.1.2a simulado pela VM VirtualBox. Os arquivos fonte estão localizados em `/usr/local/`.

Os arquivos modificados foram:

- `/usr/local/include/unistd.h`
- `/usr/local/include/callnr.h`
- `/usr/local/src/include/unistd.h`
- `/usr/local/src/include/callnr.h`
- `/usr/local/src/lib/posix/Makefile.in`
- `/usr/local/src/servers/fs/misc.c`
- `/usr/local/src/servers/fs/proto.h`
- `/usr/local/src/servers/fs/table.c`
- `/usr/local/src/servers/pm/alloc.c`
- `/usr/local/src/servers/pm/proto.h`
- `/usr/local/src/servers/pm/table.c`

As versões modificadas estão em `/usr/local/`, assim como os arquivos não modificados. Deste jeito, pode-se rodar `/usr/local/src/tools/Makefile` sem alterar o código original. Um arquivo foi adicionado:

- `/usr/local/src/lib/posix/_alloc_algorithm.c`

Quando os blocos de código transcritos neste relatório não forem muito grandes, vamos indicar as modificações feitas. Um símbolo `-` no início da linha indica a linha original no Minix. Um símbolo `+` no início da linha indica a nova linha adaptada para o EP. Uma linha vazia com o símbolo `-` indica que no código original a linha não existia. Analogamente, `+` em uma linha vazia indica que deletamos a linha

Date: 3 de novembro de 2016.

original. Um # indica um comentário no código, ou seja, a linha indicada por este símbolo não existe no arquivo original.

2. Alterando a política

Foram adicionadas as macros

```
-
+ #define FIRST_FIT  0
+ #define WORST_FIT  1
+ #define BEST_FIT   2
+ #define RANDOM_FIT 3
```

e o protótipo de função.

```
-
+ _PROTOTYPE(void alloc_algorithm, (int _policy));
```

Adiciona-se a macro que define a chamada de sistema em `include/minix/callnr.h`:

```
-
+ #define ALLOC_ALGORITHM 58
```

No arquivo `lib/posix/_alloc_algorithm.c` implementamos a função que passa a mensagem para os servidores:

```c
#include <lib.h>
#define alloc_algorithm _alloc_algorithm
#include <unistd.h>
#include <stdio.h>

PUBLIC void alloc_algorithm(_policy)
int _policy;
{
  message m;
  m.m1_i1 = _policy;
  return _syscall(MM, ALLOC_ALGORITHM, &m);
}
```

Note como o argumento `_policy` é mandado como mensagem para `_syscall`. Usaremos esta mensagem quando formos implementar a função nos servidores. Em seguida, adicionamos a nova função à tabela dos servidores (`pm` e `fs`).

```
- no_sys, /* 58 = unused */
+ do_alloc_algorithm, /* 58 = ALLOC_ALGORITHM */
```

E adicionamos o protótipo da função em `proto.h`.

```
-
+ _PROTOTYPE(int do_alloc_algorithm, (int policy));
```

E em seguida implementamos a nova chamada de sistema em `pm/alloc.c`.

```
1  PUBLIC int do_alloc_algorithm(policy)
2  int policy;
3  {
4    /* Gets argument policy from message. */
5    policy = m_in.m1_i1;
6    if (policy != FIRST_FIT && policy != BEST_FIT
7          && policy != WORST_FIT && policy != RANDOM_FIT)
8        return EINVAL; /* invalid argument error as defined in errno.h */
9    alloc_policy = policy;
10   return OK;
11 }
```

Linha 5 refere-se a mensagem que mandamos por `lib/posix/_alloc_algorithm.c`. Verificamos se a política enviada não é uma política válida. Se tal erro ocorre, retornamos `EINVAL`, que é o sinal de argumento inválido. Senão, atualizamos uma variável global `alloc_policy` com o novo valor.

A função de usuário para mudar a política é dado por `change_allocation_policy.c` em `/root`.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <string.h>
4
5  int main(int argc, char *args[]) {
6    int pol;
7    char *str;
8
9    if (argc != 2) {
10     printf(``Usage:\n  %s policy\nArguments:\n''
11         ``  policy - Indicates which policy to use:\n''
12         ``    first_fit, worst_fit, best_fit, random_fit\n'', args[0]);
13     return 1;
14   }
15
16   str = args[1];
17   if (!strcmp(str, ``first_fit'')) {
18     pol = FIRST_FIT;
19   } else if (!strcmp(str, ``worst_fit'')) {
20     pol = WORST_FIT;
```

```
  } else if (!strcmp(str, ``best_fit'')) {
    pol = BEST_FIT;
  } else if (!strcmp(str, ``random_fit'')) {
    pol = RANDOM_FIT;
  } else {
    puts(``Argument invalid.'');
    return 2;
  }

  alloc_algorithm(pol);

  return 0;
}
```

Esta função apenas faz a comparação de strings para descobrir qual política o usuário deseja atualizar o sistema com. Em seguida, chama a função `alloc_algorithm` que acabamos de descrever.

3. Políticas de alocação

Nesta seção descreveremos como foram feitas as diferentes políticas. O Minix já implementa `FIRST_FIT` em `servers/pm/alloc.c` na função `alloc_mem`. Esta implementação serviu de base para as outras implementações.

3.1. Worst fit

```
prev_ptr = hole_head;
hp = hole_head->h_next;

cand_prev_ptr = NIL_HOLE;
candidate = hole_head;
candidate_flag = 0;

if (candidate->h_len >= clicks)
  candidate_flag = 1;

while (hp != NIL_HOLE && hp->h_base < swap_base) {
  if (hp->h_len > candidate->h_len && hp->h_len >= clicks) {
    cand_prev_ptr = prev_ptr;
    candidate = hp;
    candidate_flag = 1;
  }
  prev_ptr = hp;
  hp = hp->h_next;
}

```

```
21  if (candidate_flag) {
22    old_base = candidate->h_base;
23    candidate->h_base += clicks;
24    candidate->h_len -= clicks;
25
26    if(candidate->h_base > high_watermark)
27      high_watermark = candidate->h_base;
28
29    if (candidate->h_len == 0)
30      del_slot(cand_prev_ptr, candidate);
31
32    return(old_base);
33  }
```

Nesta política, escolhemos o buraco de memória que seja maior para adicionarmos o novo processo. Na linha 4-6 definimos o ponteiro para o buraco anterior ao candidato, o ponteiro para o candidato e uma *flag* para definir se achamos um possível candidato. As linhas 8-9 apenas verificam se a própria cabeça da lista é um possível candidato. Em seguida, iteramos pela lista. Se o tamanho do buraco for o suficiente para a memória do processo (`clicks`) e o tamanho de tal buraco excede o do nosso atual candidato, então atualizamos o candidato. Linha 21 ocorre quando já escolhemos um candidato. Se tal candidato for exatamente o tamanho da memória necessária (linha 29), então deletamos o buraco. Atualizamos o buraco candidato e retornamos o local de memória a ser usado. Caso a linha 21 retorne falso, então tentaremos fazer o `swap` da memória. Se mesmo assim não for possível, então retornaremos `NO_MEM`, que indica que não há memória suficiente.

3.2. Best fit

```
1   prev_ptr = hole_head;
2   hp = hole_head->h_next;
3
4   cand_prev_ptr = NIL_HOLE;
5   candidate = hole_head;
6   candidate_flag = 0;
7
8   if (candidate->h_len >= clicks)
9     candidate_flag = 1;
10
11  while (hp != NIL_HOLE && hp->h_base < swap_base) {
12    if (!candidate_flag && hp->h_len >= clicks) {
13      cand_prev_ptr = prev_ptr;
14      candidate = hp;
15      candidate_flag = 1;
16    }
17    if (hp->h_len < candidate->h_len && hp->h_len >= clicks) {
18      cand_prev_ptr = prev_ptr;
```

```
19      candidate = hp;
20      candidate_flag = 1;
21    }
22    prev_ptr = hp;
23    hp = hp->h_next;
24  }
25
26  if (candidate_flag) {
27    old_base = candidate->h_base;
28    candidate->h_base += clicks;
29    candidate->h_len -= clicks;
30
31    if(candidate->h_base > high_watermark)
32      high_watermark = candidate->h_base;
33
34    if (candidate->h_len == 0)
35      del_slot(cand_prev_ptr, candidate);
36
37    return(old_base);
38  }
```

Na política *best fit*, queremos achar o buraco que tenha o menor possível tamanho e ainda seja suficiente para manter a memória do processo. Assim como em *worst fit*, temos um candidato que indica qual o buraco a ser modificado (ou potencialmente removido). A única diferença entre worst fit e best fit é a condição na qual escolhemos o candidato. Note que a linha 17 escolhe um buraco que tenha o menor possível tamanho mas que ainda seja maior ou igual a `clicks`. Assim como em worst fit, após acharmos o melhor candidato, atualizamos o buraco (potencialmente removendo se o tamanho foi exatamente igual ao do requisitado) e em seguida retornamos o local da memória.

3.3. Random fit

```
1   hp = hole_head;
2
3   possible_candidates = 0;
4
5   while (hp != NIL_HOLE && hp->h_base < swap_base) {
6     if (hp->h_len >= clicks) {
7       possible_candidates++;
8     }
9     hp = hp->h_next;
10  }
11
12  prev_ptr = NIL_HOLE;
13  hp = hole_head;
14
```

```
15  if (possible_candidates > 0) {
16    selected = (random() \% possible_candidates) + 1;
17    i = 0;
18    while (hp != NIL_HOLE && hp->h_base < swap_base) {
19      if (hp->h_len >= clicks) {
20        i++;
21        if (i == selected) {
22          old_base = hp->h_base;
23          hp->h_base += clicks;
24          hp->h_len -= clicks;
25
26          if(hp->h_base > high_watermark)
27            high_watermark = hp->h_base;
28
29          if (hp->h_len == 0) del_slot(prev_ptr, hp);
30
31          return(old_base);
32        }
33      }
34      prev_ptr = hp;
35      hp = hp->h_next;
36    }
37  }
```

Em *random fit*, queremos um buraco aleatória que possa conter a memória do processo. Para isso, contamos o número de candidatos possíveis (`possible_candidates`) e em seguida escolhemos algum que esteja neste intervalo. Caso não hajam candidatos possíveis, retornamos sem memória.

4. MEMSTAT

O arquivo `memstat.c` em `/root` imprime a média, mediana e desvio padrão dos buracos a cada segundo.

```
1  /*getsysinfo(MM, SI_MEM_ALLOC, &store);*/
2  void getsysinfo(int who, int what, void *where) {
3    message m;
4    m.m1_i1 = what;
5    m.m1_p1 = where;
6    _syscall(who, GETSYSINFO, &m);
7  }
```

Para recuperar o estado da lista de buracos, precisamos chamar uma chamada de sistema chamada `getsysinfo`. Como esta chamada de sistema não tem função de usuário equivalente, temos de chama-la por meio de uma `syscall`. Para tanto,

enviamos as duas mensagens `what` e `where`, que equivalem a o quê desejamos procurar e onde guardar a mensagem. No caso de `memstat`, desejamos descobrir a situação de alocação de memória (`SI_MEM_ALLOC`) e iremos guardar tal informação em um `struct pm_mem_info`, que representa uma lista de buracos.

```
int mem_data(struct hole *holes, double *mean, double *median, double *stddev) {
  int n, i, t;
  struct hole *it = holes;
  int *ord_chks;
  *mean = *median = *stddev = 0;
  n = 0;
  for (i = 0; i < _NR_HOLES; ++i)
    if (holes[i].h_base && holes[i].h_len) {
      int bytes;
      bytes = holes[i].h_len << CLICK_SHIFT;
      *mean += bytes/1024.;
      ++n;
    }
  if (n == 0) {
    *median = *stddev = 0;
    return n;
  }
  *mean /= (double) n;
  ord_chks = (int*) malloc(n*sizeof(int));
  for (t = i = 0; i < _NR_HOLES; ++i)
    if (holes[i].h_base && holes[i].h_len) {
      int bytes;
      bytes = holes[i].h_len << CLICK_SHIFT;
      ord_chks[t++] = bytes;
    }
  t = n/2;
  qsort(ord_chks, n, sizeof(int), cmp_func);
  *median = ord_chks[t]/1024.;
  free(ord_chks);
  for (i = 0; i < _NR_HOLES; ++i)
    if (holes[i].h_base && holes[i].h_len) {
      int bytes;
      double k;
      bytes = holes[i].h_len << CLICK_SHIFT;
      k = (double) bytes/1024. - *mean;
      *stddev += k*k;
    }
  *stddev = sqrt(*stddev/n);
  return n;
}
```

Como os tamanhos dos buracos estão em `clicks`, precisamos transforma-los em bytes antes. Para isso, usamos `CLICK_SHIFT`, que é simplesmente uma constante definida pelo real tamanho do click `CLICK_SIZE`. Assim, podemos fazer o *shift-left* para multiplicar o número real em bytes em cada click pelo número de clicks. Durante a instalação do Minix, foi decidido durante a configuração que `CLICK_SIZE` fosse `4096`. Assim que fazemos o *shift* temos o tamanho em bytes. No entanto, como o tamanho em bytes será muito grande, decidimos representar os kilobytes ao invés. Em `mem_data`, primeiro computamos a média, contando o número real de buracos na memória. Em seguida, copiamos os tamanhos em um vetor, ordenamos de acordo com uma função crescente `cmp_func` e selecionamos a mediana. Ao final, computamos o desvio padrão. Retornamos o número de elementos e guardamos os valores da média, mediana e desvio padrão nos endereços dados.

5. Testes

Para testar os algoritmos, nós utilizamos o programa `forkmem` (fornecido de material) para gerar vários processos que consomem diferentes porções de memória, e para visualizar a mudança nos buracos de memória, usamos o `memstat`, já descrito anteriormente.

Criamos dois scripts simples para facilitar a execução de testes, são eles:

testmem01

```
1 /bin/sh
2
3 1 3 6 12 16 24 32 40
4
5 memstat 6 > $1 &
6 forkmem 32+0+2 32+0+3 32+0+4 12+2+3 12+2+4 12+2+5 3+3+4 3+3+4 3+3+4 3+3+5 3+3+5 3+3+5
```

Esse script executa o `memstat`, enviando a sua saída para um arquivo passado como argumento, e, em paralelo, executa o `forkmem` com uma série de processos.

No instante 0: são criados três processos de 32 MB, que acabam nos instantes 2, 3 e 4.

No instante 2: um processo de 32 MB é encerrado, liberando memória. Logo em seguida, outros três processos de 12 MB são criados, para se encerrarem nos instantes 3, 4 e 5. (Note que dois deles cabem no buraco de 32 MB liberado, mas o terceiro precisa de outro buraco)

No instante 3: um processo de 32 MB e um de 12 MB são encerrados. Seis processos de 3 MB são criados, três para se encerrarem no instante 4 e os outros três, no instante 5.

No instante 4: um processo de 32 MB, um de 12 MB, e três de 3 MB são encerrados.

No instante 5: um processo de 12 MB, e três de 3 MB são encerrados.

No instante 6: nada acontece, esse segundo a mais que o `memstat` observa é somente para dar uma pausa entre execuções.

e

fulltest01.sh

```
/bin/sh

change_allocation_policy first_fit
testmem01 output_first_fit_01

change_allocation_policy worst_fit
testmem01 output_worst_fit_01

change_allocation_policy best_fit
testmem01 output_best_fit_01

change_allocation_policy random_fit
testmem01 output_random_fit_01
```

Este script somente troca a política e executa o script anterior, passando o nome do arquivo no qual os resultados serão salvos.

Resultados:

First Fit:

```
6  173123.333 kB  60.000  kB  387026.100 kB
8  117498.500 kB  100.000 kB  310702.486 kB
9  108098.667 kB  100.000 kB  294137.151 kB
9  103956.444 kB  100.000 kB  291042.895 kB
10  94926.400 kB  116.000 kB  277469.849 kB
10  98218.400 kB  116.000 kB  276770.808 kB
11  90418.545 kB  116.000 kB  265040.812 kB
```

Worst Fit:

```
10  100578.800 kB  116.000 kB  301471.074 kB
12   78332.333 kB  116.000 kB  259479.364 kB
13   74837.538 kB  116.000 kB  249593.485 kB
```

```
15   64567.200 kB  132.000 kB  223890.411 kB
21   45203.048 kB  132.000 kB  187186.505 kB
19   51693.895 kB  132.000 kB  196179.738 kB
19   52347.579 kB  132.000 kB  196027.474 kB
```

Best Fit:

```
9   111754.222 kB  100.000 kB  315889.273 kB
9   104443.111 kB  100.000 kB  295210.328 kB
10   97288.800 kB  116.000 kB  280882.327 kB
11   88046.182 kB  116.000 kB  265468.727 kB
11   86296.727 kB  116.000 kB  265915.924 kB
11   89289.455 kB  116.000 kB  265338.156 kB
12   82883.667 kB  116.000 kB  254928.783 kB
```

Random Fit:

```
8   125723.500 kB  100.000 kB  332419.951 kB
8   117498.500 kB  100.000 kB  310658.647 kB
9   108098.667 kB  100.000 kB  294095.988 kB
11   88046.182 kB  132.000 kB  265426.831 kB
11   86296.727 kB  132.000 kB  263846.872 kB
10   98218.400 kB  152.000 kB  274524.072 kB
11   90418.545 kB  152.000 kB  262908.005 kB
```

Nesse teste, os resultados não mostram muitas variações entre cada política. O que mais se destaca é o Worst Fit. Ele gera uma quantidade muito maior de buracos ao longo das execuções, com a média de cada um muito menor (aproximadamente a metade dos outros). Vemos que o First Fit tem uma quantidade menor de buracos no começo da execução, mas, pela construção do exemplo, termina com valores similares ao Best e Random Fit. Por fim, o Best Fit termina com um buraco a mais, por isso tem uma média de tamanho de buracos um pouco menor do que os outros (além do Worst).

6. Observações importantes quanto a execução do EP

Como usamos um Floppy Controller na nossa VM, a tela inicial irá indicar que não foi encontrado um local de boot. Para resolver isto, pressione `F12` e em seguida pressione `1`. Isto selecionará o controlador principal (a que possue a imagem do Minix) como local de boot.

Como não mudamos o caminho de boot padrão do Minix, quando a VM for rodada, deve-se dar boot na imagem correta. Por padrão, a VM irá dar boot na imagem padrão original.

É recomendável que se recompile o Minix novamente para garantir que tudo esteja o mais recente possível. Caso não se recompile o Minix, a imagem em `/boot/image` mais recente é:

```
/boot/image/3.1.2ar44
```

Para rodar a imagem escolhida, basta indicar o caminho. Por exemplo, caso a imagem desejada seja `/boot/image/3.1.2ar44`, então:

```
# Garanta que esteja na tela de boot.
shutdown
# Indique qual imagem deve ser escolhida.
image=/boot/image/3.1.2ar44
# Faca o boot.
boot
```