UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
CIÊNCIA DA COMPUTAÇÃO

Artur Luiz Rizzato Toru Soda
Davi Menegaz Junkes
Thiago Kenzo Takahashi

Relatório - Trabalho 2 - Simulação de Alocação e Gerenciamento de Memória Livre

Florianópolis
2024

1. INTRODUÇÃO

Este trabalho tem como objetivo aplicar os conhecimentos aprendidos na disciplina Sistemas Operacionais I, simulando alocação e gerenciamento de memória livre, sendo eles: Bitmap, e Lista Duplamente Encadeada, recebendo como entrada o método de gerência, a quantidade memória, o tamanho de bloco mínimo e o algoritmo a ser utilizado.

Em seguida, o programa recebe pedidos de alocação e desalocação, e então deve simular o comportamento de gerenciamento de memória livre em sistemas operacionais modernos.

2. SIMULADOR

No começo de sua execução, o simulador carrega as informações lidas do arquivo de entrada e inicia as variáveis mediadoras e indicadoras, além do gerenciador de memória, devidamente:

```
4   Simulator::Simulator() {
5       FileReader fr = FileReader();
6       fr.readFile();
7       parameters* p = fr.getParameters();
8       loadParameters(p);
9
10      bytes_allocated = 0;
11      bytes_deleted = 0;
12      qty_allocation = 0;
13      qty_deletion = 0;
14
15      start = new int[REQUEST_SIZE];
16      size = new int[REQUEST_SIZE];
17
18      int qty_block = mem_size/block_size;
19      qty_block += (mem_size % block_size == 0) ? 0 : 1;
20
21      if (memory_manager == 1) manager = new BitmapManager(qty_block, block_size);
22      else                     manager = new DllManager(qty_block, block_size);
23  }
```

Importante citar, que os dois gerenciadores de memória *BitmapManager* e *DllManager* herdam de uma classe abstrata *Manager*, e possuem os mesmos métodos da classe mãe:

```
class Manager {
public:
    Manager();
    ~Manager();

    virtual int alloc(int size, int alloc_alg) = 0;
    virtual void del(int seg_start, int seg_size) {};

    virtual int firstFit(int size) = 0;
    virtual int nextFit(int size) = 0;
    virtual int searchSeg(int  seg_ptr, int size) = 0;
    virtual int getQtyAllocatedBlocks() = 0;

    virtual void printState() {};

    int block_size;
};
```

Agora voltando ao simulador, no método *run()* do simulador (onde está o looping principal da simulação), é identificado cada pedido feito, e é chamado a função *alloc()* ou *del()* dependendo da solicitação:

```
65  void Simulator::run() {
66      cout << endl;
67      for (int i = 0; i < qty_requests; i++) {
68          string request = requests[i];
69          stringstream ss(request);
70
71          string type;
72          int size_, id, qty_block;
73
74          ss >> type;
75          if (type == "A") {
76              ss >> size_ >> id;
77              qty_block = size_/block_size;
78              qty_block += (size_ % block_size == 0) ? 0 : 1;
79              alloc(qty_block, id);
80
81          } else {
82              ss >> id;
83              del(id);
84          }
85          cout << " -> " << type << " " << (type == "A" ? to_string(size_)+" " : "") << id << endl;
86          manager->printState();
87          cout << endl;
88      }
```

Por fim, ainda no método *run()*, é impresso o estado final do gerenciador, e as

medições feitas:

```
90          cout << endl;
91          cout << "-=-=-=-=-=-=-=-=-=-=- Impressoes finais -=-=-=-=-=-=-=-=-=-=- " << endl;
92          cout << endl;
93
94          cout << "Estado Final do Gerenciador de Memoria:" << endl;
95          manager->printState();
96          cout << endl;
97          printFinalState();
98          cout << "-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=- " << endl;
99      }
100
```

Continuando, a função e *alloc()* e *del()*, apenas chama a função *alloc()* do gerenciador, e faz as medições necessárias.

```
46  void Simulator::alloc(int size_, int id) {
47          int seg_start = manager->alloc(size_, alloc_alg);
48
49          if (seg_start != -1) {
50              bytes_allocated += size_;
51              qty_allocation += 1;
52              size[id] = size_;
53              start[id] = seg_start;
54          }
55      }
56
57      // Desaloca segmento do pedido
58  void Simulator::del(int id) {
59          bytes_deleted += size[id];
60          qty_deletion += 1;
61          manager->del(start[id], size[id]);
62      }
```

2.1. Lista Duplamente Encadeada

No método de alocação do gerenciador por Lista Duplamente Encadeada, primeiro é pesquisado qual o bloco de início do segmento livre que deve ser utilizado, utilizando o algoritmo de alocação firstFit ou nextFit indicados nos parâmetros da função, caso não ache segmento que caiba o pedido, nada é alocado.

Em seguida, é deletado o segmento livre escolhido da lista, e alocamos o novo segmento. Por fim, adicionamos a lista, o espaço que tiver sobrado do segmento livre deletado.

```
16  int DllManager::alloc(int size, int alloc_alg) {
17      int seg_start;
18      element old_seg, allocated_seg, free_seg;
19
20      // Busca bloco de inicio do segmento livre a ser alocado
21      // Caso nao ache segmento com tamanho o suficiente, nao aloca
22      seg_start = (alloc_alg == 1) ? firstFit(size) : nextFit(size);
23      if (seg_start == -1) return seg_start;
24
25      // Marca bloco de inicio do ultimo bloco alocado
26      last_allocation_start = seg_start;
27
28      // Busca e remove seguimento livre encontrado
29      for (size_t i = 0; i < mem_list_dll.size(); i++) {
30          old_seg = mem_list_dll.at(i);
31          if (old_seg.start == seg_start) break;
32      }
33      mem_list_dll.remove(old_seg);
34
35      // Aloca seguimento do pedido feito
36      allocated_seg = {seg_start, size, 1};
37      mem_list_dll.insert_sorted(allocated_seg);
38
39      // Aloca o restante do seguimento livre removido, se houver
40      if (old_seg.size - allocated_seg.size) {
41          free_seg = {seg_start+size, old_seg.size-allocated_seg.size, 0};
42          mem_list_dll.insert_sorted(free_seg);
43      }
44
45      return seg_start;
46  }
```

Ademais, tanto na estratégia *first-fit* quanto na *next-fit* é pesquisado o bloco de início do primeiro segmento encontrado que caiba o pedido feito. Entretanto, enquanto no *first-fit* a pesquisa é sempre feita a partir do início da estrutura utilizada pelo gerenciador, no *next-fit* a pesquisa é sempre feita a partir do último bloco alocado.

Dessa forma, as funções *firtsFit()* e *nextFit()* apenas indicam por onde a pesquisa deve iniciar.

```
90      int DllManager::firstFit(int size) {
91          // Busca segmento a partir do inicio do lista
92          return searchSeg(0, size);
93      }
94
95
96      // Retorna bloco de inicio do primeiro segmento livre que caiba o pedido
97      //     a partir do inicio do ultimo segmento alocado
98      int DllManager::nextFit(int size) {
99          int seg_ptr;
100         // Busca segmento a partir do ultimo segmento alocado
101         for (size_t i = 0; i < mem_list_dll.size(); i++) {
102             if (mem_list_dll.at(i).start == last_allocation_start) {
103                 seg_ptr = i;
104                 break;
105             }
106         }
107         return searchSeg(seg_ptr, size);
108     }
```

E é no método *searchSeg()* que é feito a busca do segmento livre em si, possuindo o seguinte funcionamento: visita todos os segmentos da estrutura a partir do que foi indicado em seu parâmetro, até achar o primeiro que caiba o pedido feito, caso visite todos os segmentos e não ache um apropriado, é retornado -1 que indica que não foi encontrado um segmento adequado.

```cpp
int DllManager::searchSeg(int seg_ptr, int size) {
    element elem;
    int seg_start;
    bool flag;

    // Array para controle de seguimentos ja visitados
    bool visited_seg[mem_list_dll.size()];
    for (size_t i = 0; i < mem_list_dll.size(); i++)
        visited_seg[i] = false;

    seg_start = -1;
    flag = true;
    // Loop enquanto ainda tiver segmentos a serem visitados
    while (true) {
        // Verifica se ha segmentos para ser visitados
        for (auto seg: visited_seg) {
            if (seg == false) flag = false;
        }
        if (flag) break;

        // Se ponteiro passar do ultimo segmento, volta a apontar ao primeiro segmento
        if (seg_ptr >= static_cast<int>(mem_list_dll.size()))
            seg_ptr = 0;

        // Indica como segmento visitado
        visited_seg[seg_ptr] = true;

        // Verifica se segmento apontado, possui tamanho desejado e se esta livre
        elem = mem_list_dll.at(seg_ptr);
        if ((elem.size >= size) and (elem.status == 0)) {
            seg_start = elem.start;
            break;
        }

        // Aponta para o proximo segmento
        seg_ptr++;
    }

    return seg_start;
}
```

Por fim, no método de desalocação da Lista Duplamente Encadeada, buscamos o segmento a ser desalocado, e a partir dele e dos seus segmentos vizinhos (se estiverem livres), é montado o segmento livre a ser colocado no lugar. Assim, deletamos o segmento indicado pelo pedido da lista, e adicionamos o segmento livre criado.

```
49  void DllManager::del(int seg_start, int seg_size) {
50      element cur_seg, back_seg, front_seg, free_seg;
51
52      for (size_t i = 0; i < mem_list_dll.size(); i++) {
53          cur_seg = mem_list_dll.at(i);
54
55          // Busca seguimento para desalocar
56          if (seg_start == cur_seg.start) {
57              // Guarda seguimentos vizinhos, se houverem
58              if (i != 0) back_seg = mem_list_dll.at(i-1);
59              if (i != (mem_list_dll.size()-1)) front_seg = mem_list_dll.at(i+1);
60
61              // Define inicio, tamanho e status do
62              //     do seguimento livre a ser alocado no lugar
63              //     unindo com os segmentos vizinhos, se existirem e estiverem livres
64              free_seg.start = ((i != 0) && (back_seg.status == 0)) ? back_seg.start : cur_seg.start;
65
66              free_seg.size = seg_size;
67              free_seg.size += ((i != 0) && (back_seg.status == 0)) ? back_seg.size : 0;
68              free_seg.size += ((i != (mem_list_dll.size()-1)) && (front_seg.status == 0)) ? front_seg.size : 0;
69
70              free_seg.status = 0;
71
72              // Desaloca seguimento do pedido feito
73              mem_list_dll.remove(cur_seg);
74
75              // Une deleta vizinhos unidos com o segmento livre
76              if ((i != 0) && (back_seg.status == 0)) mem_list_dll.remove(back_seg);
77              if ((i != (mem_list_dll.size()-1)) && (front_seg.status == 0)) mem_list_dll.remove(front_seg);
78
79              // Insere na lista o segmento livre
80              mem_list_dll.insert_sorted(free_seg);
81
82              return;
83          }
84      }
85  }
```

2.2. Bitmap

No método de alocação do gerenciador por Bitmap, primeiro é pesquisado qual o bloco de início do segmento livre que deve ser utilizado, utilizando o algoritmo de alocação firstFit ou nextFit indicados nos parâmetros da função, caso não ache segmento que caiba o pedido, nada é alocado. Em seguida fazemos a alocação.

```
18  int BitmapManager::alloc(int size, int alloc_alg) {
19      // Busca bloco de inicio do segmento livre a ser alocado
20      // Caso nao ache segmento com tamanho o suficiente, nao aloca
21      int seg_start = (alloc_alg == 1) ? firstFit(size) : nextFit(size);
22      if (seg_start == -1) return seg_start;
23
24      // Marca bloco de inicio do ultimo bloco alocado
25      last_allocation_start = seg_start + size;
26
27      // Aloca
28      for (int i = 0; i < size; i++)
29          mem_list_bit->fix(seg_start + i);
30
31      return seg_start;
32  }
```

Continuando, os métodos *firstFit()*, *nextFit()* e *searchSeg()* possuem a mesma lógica de funcionamento já explicados anteriormente no tópico de Lista Duplamente Encadeada, apenas é adaptado para bitset:

```
44  int BitmapManager::firstFit(int size) {
45      // Busca segmento a partir do inicio do bitset
46      return searchSeg(0, size);
47  }
48
49  // Retorna bloco de inicio do primeiro segmento livre que caiba o pedido
50  //     a partir do inicio do ultimo segmento alocado
51  int BitmapManager::nextFit(int size) {
52      int block_ptr;
53      // Busca segmento a partir do ultimo segmento alocado
54      for (int i = 0; i < mem_list_bit->size(); i++) {
55          if (i == last_allocation_start) {
56              block_ptr = i;
57              break;
58          }
59      }
60      return searchSeg(block_ptr, size);
61  }
```

```
65  int BitmapManager::searchSeg(int seg_ptr, int size) {
68
69      // bitset para controle de blocos ja visitados
70      Bitset visited_block = Bitset(mem_list_bit->size());
71
72      seg_start = -1;
73      flag = false;
74
75      // Loop enquanto ainda tiver blocos a serem visitados
76      while (visited_block.get_qtd_zero()) {
77          // Se ponteiro passar do ultimo bloco, volta a apontar ao primeiro bloco
78          if (seg_ptr >= mem_list_bit->size())
79              seg_ptr = 0;
80
81          seg_size = 0;               // Indica tamanho do segmento livre atual
82          bitset_ptr = seg_ptr;       // Ponteiro para verificar segmento livre
83          visited_block.fix(seg_ptr); // Indica bloco visitado
84
85          // Verifica bloco livre
86          while (mem_list_bit->value()[bitset_ptr] == '0') {
87              seg_size++;
88
89              // Verifica se segmento livre alcacar o tamanho desejado
90              if (seg_size >= size) {
91                  flag = true;
92                  break;
93              }
94
95              // Aponta para o segmento livre que estamos
96              bitset_ptr++;
97
98              // se passarmos do ultimo bloco da lista, paramos de verificar
99              if (bitset_ptr >= static_cast<int>(mem_list_bit->size()))
100                 break;
101
102             // Indica bloco visitado
103             visited_block.fix(bitset_ptr);
104         }
105
106         // Verifica se achou segmento livre de tamanho desejado
107         if (flag) {
108             seg_start = seg_ptr;
109             break;
110         }
111
112         // Aponta para proximo bloco nao livre
113         seg_ptr += (seg_size) ? seg_size : 1;
114     }
115     return seg_start;
```

Por fim, no método de desalocação do Bitmap, apenas desalocamos o segmento indicado.

```
36      void BitmapManager::del(int seg_start, int seg_size) {
37          for (int i = 0; i < seg_size; i++)
38              mem_list_bit->unfix(seg_start+i);
39      }
```

3. EXECUÇÃO

Para o exemplo a seguir, foi utilizado o seguinte arquivo de entrada, lembrando que cada linha corresponde as seguintes informações, respectivamente: gerenciador de memória, tamanho da memória, tamanha mínimo do bloco, algoritmo de alocação, e por fim, os pedidos de alocação e desalocação:

```
1       1
2       100
3       1
4       1
5       A 10 1
6       A 20 2
7       A 5 3
8       A 30 4
9       A 5 5
10      D 2
11      D 4
12      A 10 6
```

Assim, para essa entrada temos um gerenciador por Bitmap, com 100 bytes de memória, e 1 byte por bloco, e utilizando first-fit como algoritmo de alocação.

Por fim, para cada pedido feito é impresso a situação da estrutura do gerenciador. E no final da execução é impresso o estado final do gerenciador e as medições feitas.

```
artur@desktop-artur ~/SO1/Trabalho-2  <main*>
➤ ./simulador < Entradas/entradaFirstFitBitmap.txt
-=-=-=-=-=-=-=-=-=-=-=-=- Simulacao -=-=-=-=-=-=-=-=-=-=-=-=-
 -> Estado inicial:
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

 -> A 10 1
11111111110000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

 -> A 20 2
11111111111111111111111111111000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

 -> A 5 3
11111111111111111111111111111110000000000000000000000000000000000000000000000000000000000000000000000000000000000000

 -> A 30 4
11111111111111111111111111111111111111111111111111111111111110000000000000000000000000000000000000000000000000000000

 -> A 5 5
11111111111111111111111111111111111111111111111111111111111111100000000000000000000000000000000000000000000000000000

 -> D 2
11111111110000000000000000000111111111111111111111111111111111111100000000000000000000000000000000000000000000000000

 -> D 4
11111111110000000000000000000111111000000000000000000000000000011111000000000000000000000000000000000000000000000000

 -> A 10 6
11111111111111111110000000000111110000000000000000000000000000011111000000000000000000000000000000000000000000000000

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

-=-=-=-=-=-=-=-=-=-=- Impressoes finais -=-=-=-=-=-=-=-=-=-=-=-

Estado Final do Gerenciador de Memoria:
11111111111111111110000000000111110000000000000000000000000000011111000000000000000000000000000000000000000000000000

Quantidade de bytes em uso (ocupados):
 -> 30 Bytes.

Quantidade de bytes alocados:
 -> 80 Bytes.

Quantidade de bytes desalocado:
 -> 50 Bytes.

Numero de alocacoes:
 -> 6 Alocacoes.

Numero de desalocacoes:
 -> 2 Desalocacoes.
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
```

Agora um exemplo de gerenciador por lista duplamente encadeada, onde foi utilizado a entrada disponibilizado no enunciado do trabalho.

```
1    1
2    8388608
3    4
4    2
5    A 1024 1
6    A 100 2
7    A 50 3
8    D 2
```

Assim, para essa entrada temos 8 MegaBytes de memória, e 4 byte por bloco, e utilizando next-fit como algoritmo de alocação.

```
┌─artur@desktop-artur ~/SO1/Trabalho-2  ‹main*›
└─➤ ./simulador < Entradas/entrada.txt
-=-=-=-=-=-=-=-=-=-=-=-=-=- Simulacao -=-=-=-=-=-=-=-=-=-=-=-=-=-
 -> Estado inicial:
8388608 0

 -> A 1024 1
1024 1
8387584 0

 -> A 100 2
1024 1
100 1
8387484 0

 -> A 50 3
1024 1
100 1
52 1
8387432 0

 -> D 2
1024 1
100 0
52 1
8387432 0

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

-=-=-=-=-=-=-=-=-=-=-=- Impressoes finais -=-=-=-=-=-=-=-=-=-=-=-=-

Estado Final do Gerenciador de Memoria:
1024 1
100 0
52 1
8387432 0

Quantidade de bytes em uso (ocupados):
 -> 1076 Bytes.

Quantidade de bytes alocados:
 -> 294 Bytes.

Quantidade de bytes desalocado:
 -> 25 Bytes.

Numero de alocacoes:
 -> 3 Alocacoes.

Numero de desalocacoes:
 -> 1 Desalocacoes.
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
```

4. CONCLUSÃO

Considerando os tópicos supracitados, este trabalho nos permitiu simular alocações e gerenciamento em memória livre. Dessa forma, pudemos solidificar melhor o conhecimento sobre o comportamento de gerenciadores de memória em sistemas operacionais modernos.