Universidade de São Paulo
Instituto de Matemática e Estatistica
IME

EP3 - Sistemas Operacionais

Patrícia da Silva Rodrigues (nºUSP 11315590)
Gabriel Ferreira de Souza Araujo (nºUSP 12718100)

Junho
2023

1 Implementação da chamada de sistemas compact_mem

se trata de um algoritmo de complexidade quadrática ($O(n^2)$) que percorre a lista de buracos n $n * (n-1) * (n-1) * ...2 * 1$ vezes, buscando, para cada bloco hp com fragmentação, qual dos outros n - hp blocos possuem a melhor quantidade de clicks que melhor compactaria o bloco em questão (para que ele fique compactado da melhor maneira possível, o ideal é encontrar, dentre os que cabem dentro do espaço disponível no bloco hp (hp->h_len > hp2->h_base), o maior dentre os menores. O bloco que possuir a maior quantidade de clicks dentre os que cabem em hp, será o bloco que melhor compactará o bloco hp, deixando assim a menor quantidade de buraco possível (hp->h_len), de modo que fique bem compacto.

para executa-lo, basta fazer: no diretório root: ./compact_mem

Infelizmente, apesar do codigo compilar sem erros, o algoritmo de compactação gera uma pani no PM, acreditamos que isso tem haver com o fato de or processos ordenados nao poderem ter prioridade maior do que a priordade do PM, mas não soubemos como arrumar isso. No entanto, escrevemos aqui toda a logica de funcionamento do programa:

no diretório **/usr/src/servers/pm**

O algoritmo opera dentro do alloc_mem da seguinte forma: Quando queremos compactar a memória, fazemos uma chamda de sistema que dispara uma mudança de variável na variável global alloc_policy. Essa variável fica responsável por direcional o que a função alloc_policy fará.

caso alloc_policy == 0: executa o first_fit (algoritmo padrão de allocação)

caso alloc_policy == 1: executa o compact_mem (nosso algoritmo de compactação (passamos para o alloc_mem um parametro qualquer de clicks que não iremos usar para nada, apenas para poder fazer esse versionamento)

caso alloc_policy == 2: executa o best_fit que, apesar de não ser um algoritmo de realocação e sim de alocação, consegue gerar uma melhor compactação na memória, visto que opera de uma maneira a alocar os clicks passados em um lugar onde gere menos ragmentação.

Implementação da syscall

compact_memory: Esse programa se encontra na /root. Ele é responsável pela compctação da memória e fazer que o usuário escolha a maneira de fazer isso. Para executa-lo, basta fazer:
./compact_memory first_fit , ou
./compact_memory compact_bestfit , ou
./compact_memory compact_mem

misc.c: do_challocpolicy recebe a mensagem que contém o algoritmo de alocação de- sejado e chama a função ch alloc policy para o sistema usar esse algoritmo.

table.c: Colocamos a chamada do challocpolicy na entrada 66 da tabela.

proto.h: Definimos o protótipo da função ch alloc policy e da função do challocpolicy aqui.

usr/include/minix/call.nr e usr/src/include/minix/call.nr

Definimos a constante CHALLOCPOLICY como 66. usr/src/lib/posix/challocpolicy.c

Monta a mensagem com o argumento que representa o algoritmo de alocação desejado e faz a syscall.

Segue abaixo o algoritmo de compactaçao

Code Listing 1: Código de compactação de memória

```
1
2
3     /*compacta memoria*/
4     prev_ptr = NIL_HOLE;
5     prev_ptr2 = NIL_HOLE;
6     hp = hole_head;
7     while (hp != NIL_HOLE && hp->h_base < swap_base) {
8       vazio = 1;
9       maiorDosMenores = NIL_HOLE;
10      hp2 = hole_head;
11
12      while (hp2 != NIL_HOLE && hp2->h_base < swap_base) {
13        if (hp->h_len >= hp2->h_base) {
14          if (vazio == 1) {
15            maiorDosMenores = hp2;
16            vazio = 0;
17          } else {
18            if (maiorDosMenores->h_base < hp2->h_base) {
19              maiorDosMenores = hp2;
20            }
21          }
22          prev_ptr2 = hp2;
23        }
24        prev_ptr = hp;
25        hp2 = hp2->h_next;
26      }
27
28        if (vazio == 0 ) {
29
30
31          hp->h_base += maiorDosMenores->h_base;
32          hp->h_len -= maiorDosMenores->h_base;
33
34          maiorDosMenores->h_base -= maiorDosMenores->h_base
                ;
35          maiorDosMenores->h_len += maiorDosMenores->h_base;
36
37          if(maiorDosMenores->h_base > high_watermark)
38            high_watermark = maiorDosMenores->h_base;
39
40          if (hp->h_len == 0) {
41            del_slot(prev_ptr,hp);
42          }
43        }
44
45        hp2 = hp->h_next;
46        prev_ptr = hp;
47        hp = hp->h_next;
```

amarelo: Inicialmente, são definidas duas variáveis auxiliares: prev_ptr e prev_ptr2. A variável prev_ptr será usada para manter o ponteiro para o elemento anterior da lista hole_head, enquanto prev_ptr2 será usada para manter o ponteiro para o elemento anterior durante o segundo loop interno.

verde:O ponteiro hp é inicializado com hole_head, que representa o primeiro elemento da lista de buracos de memória.

azul: O algoritmo entra em um loop principal que continua enquanto hp não for nulo e o endereço base (h_base) de hp for menor que swap_base (uma constante que representa o limite superior da área de swap). Dentro do loop principal, duas variáveis são inicializadas: vazio é definida como 1 (indicando que não foram encontrados buracos que possam ser combinados) e maiorDosMenores é definida como nula (apontando para nenhum buraco).

rosa: Um segundo loop interno é iniciado, percorrendo todos os buracos de memória novamente. Este loop tem o objetivo de encontrar o maior buraco (maiorDosMenores) que seja menor ou igual ao tamanho (h_len) do buraco atual (hp). Durante o segundo loop interno, se o tamanho do buraco atual (hp->h_len) for maior ou igual ao endereço base do buraco atual no segundo loop interno (hp2->h_base), são realizadas as seguintes verificações:

a) Se vazio for igual a 1 (ou seja, ainda não foi encontrado nenhum buraco para combinar com o buraco atual), maiorDosMenores é atualizado para o buraco atual do segundo loop interno e vazio é definido como 0.

b) Caso contrário, se o endereço base de maiorDosMenores for menor que o endereço base de hp2, maiorDosMenores é atualizado para hp2.

c) O ponteiro prev_ptr2 é atualizado para o buraco atual do segundo loop interno.

em azul claro: Após a conclusão do segundo loop interno, é verificado se vazio ainda é igual a 0. Se isso for verdade, significa há um buraco menor ou igual ao tamanho do buraco atual foi encontrado. Nesse caso, os seguintes passos são executados:

larana: a) O endereço base (h_base) do buraco atual (hp) é incrementado pelo tamanho do maiorDosMenores->h_base. b) O tamanho (h_len) do buraco atual é decrementado pelo tamanho do maiorDosMenores->h_base. c) O maiorDosMenores->h_base é definido como zero (pois o buraco foi completamente usado). d) O maiorDosMenores->h_len é incrementado pelo tamanho do maiorDosMenores->h_base. e) Se o maiorDosMenores->h_base for maior que

high_watermark (uma variável que representa o valor máximo alcançado pelo endereço base), high_watermark é atualizado com maiorDosMenores->h_base.

vermelho: Se o tamanho (h_len) do buraco atual (hp) se tornar zero após a compactação, ele é removido da lista de buracos de memória.

azul:: Por fim, os ponteiros hp2, prev_ptr, prev_ptr2 são atualizados para a próxima iteração do loop principal.

Infelizmente, apesar do codigo compilar sem erros, o algoritmo de compactação gera uma pani no PM, acreditamos que isso tem haver com o fato de or processos ordenados nao poderem ter prioridade maior do que a priordade do PM, mas não soubemos como arrumar isso.

2 Implementação do programa de usuário memstat

A o programa de usuário aciona a função memstat() coleta estatísticas sobre os buracos de memória no sistema Minix. Ela itera sobre os buracos de memória existentes, contabiliza o número de buracos, calcula a média e desvio padrão dos tamanhos dos buracos, determina a mediana dos tamanhos e imprime as estatísticas. Além disso, imprime a lista de buracos de memória, exibindo a base e o tamanho de cada buraco.

Para executa-lo, basta fazer: no diretório /root
./memstat (segundo que quer coletar as estatisticas)
ex: ./memstat 20 ./memstat 60