```
1                                       /* Cabecalho mestre. Inclui
    alguns outros arquivos e
2                                        * define as constantes
    principais.
3                                        */
4    #define _POSIX_SOURCE 1             /* headers incluem POSIX */
5    #define _MINIX 1                    /* cabecalhos incluem MINIX*/
6    #define _SYSTEM 1                   /* cabecalhos incluem que
    este é o kernel */
7
8                                       /* Os seguintes são básicos,
    todos os *.c tem automaticamente */
9    #include <minix/config.h>          /* Deve ser a primeira */
10   #include <ansi.h>                  /* Deve ser a segunda */
11   #include <sys/types.h>
12   #include <minix/const.h>
13   #include <minix/type.h>
14
15   #include <fcntl.h>
16   #include <unistd.h>
17   #include <minix/syslib.h>
18
19   #include <limits.h>
20   #include <errno.h>
21
22   #include "const.h"
23   #include "type.h"
24   #include "proto.h"
25   #include "glo.h"
26
27
28
29   /* Esse arquivo contém o programa principal de gerência de
    memória e alguns procedimentos
30    * relacionados. Quando o MINIX inicia, o kernel roda por alguns
    instantes,
31    * inicializando a si mesmo e seus processos, e então roda o  MM
    e o  FS. Ambos MM
32    * e FS inicializam na maneira do possível. FS então chama o
33    * MM, porque MM tem que esperar por FS para adquirir um disco
    RAM. MM pede
34    * ao kernel por toda a memória livre e inicia requisições ao
    servidor.
35    *
36    * Os pontos de entrada para este arquivo são:
37    * main: inicia MM
38    * reply: responde a um processo fazendo um chamada de sistema ao
    MM
39    */
40
41   #include "mm.h"
42   #include <minix/callnr.h>
43   #include <minix/com.h>
```

```
44    #include <signal.h>
45    #include <fcntl.h>
46    #include <sys/ioctl.h>
47    #include "mproc.h"
48    #include "param.h"
49
50    FORWARD _PROTOTYPE( void get_work, (void) );
51    FORWARD _PROTOTYPE( void mm_init, (void) );
52
53
      /*===========================================================================*
54     * main *
55
      *===========================================================================*/
56    PUBLIC void main()
57    {
58    /* Rotina principal da gerência de memória. */
59
60    int error;
61
62    mm_init(); /* inicializa as tabelas de gerência de memória */
63
64    /*  Loop principal do MM, que roda para sempre  */
65    while (TRUE) {
66    /* espera mensagem. */
67    get_work(); /* espera uma chamada de sistema por gerência de
      memória */
68    mp = &mproc[who];
69
70    /* Seta algumas flags. */
71    error = OK;
72    dont_reply = FALSE;
73    err_code = -999;
74
75    /* Se o número de chamada é válido, realiza a tarefa. */
76    if (mm_call < 0 || mm_call >= NCALLS)
77    error = EBADCALL;
78    else
79    error = (*call_vec[mm_call])();
80
81    /* Envia os resultados de volta para o usuário, indicando que
      completou */
82    if (dont_reply) continue; /* Sem resposta para EXIT e WAIT */
83    if (mm_call == EXEC && error == OK) continue;
84    reply(who, error, result2, res_ptr);
85    }
86    }
87
88    /*===========================================================================*
      ===========*
89     * função get_work *
```

```
 90
    *===========================================================================*/
 91   PRIVATE void get_work()
 92   {
 93                                /* Espera uma próxima chamada e
    extrai alguma informação útil dela */
 94
 95   if (receive(ANY, &mm_in) != OK) panic("MM receive error",
    NO_NUM);
 96   who = mm_in.m_source;       /* quem enviou a mensagem */
 97   mm_call = mm_in.m_type;     /* número da chamada de sistema */
 98   }
 99
100
101
    /*===========================================================================*
102   * reply *
103
    *===========================================================================*/
104   PUBLIC void reply(proc_nr, result, res2, respt)
105   int proc_nr;        /* processo ao qual responder */
106   int result;         /* resultado da chamada (usualmente OK ou
    erro #)*/
107   int res2;           /* segundo resultado */
108   char *respt;        /* resultado do ponteiro */
109   {
110                       /* Envia uma resposta a um processo de um
    usuário. */
111
112   register struct mproc *proc_ptr;
113
114   proc_ptr = &mproc[proc_nr];
115               /*
116               * Essa verificação de validade deve ser ignorada se
    o chamador for uma tarefa.
117               * Para tornar o MM robusto, verifica se o destino
    ainda está vivo
118               * Essa verificação checa se o chamador é uma tarefa.
119               */
120   if ((who >=0) && ((proc_ptr->mp_flags&IN_USE) == 0 ||
121   (proc_ptr->mp_flags&HANGING))) return;
122
123   reply_type = result;
124   reply_i1 = res2;
125   reply_p1 = respt;
126   if (send(proc_nr, &mm_out) != OK) panic("MM can't reply",
    NO_NUM);
127   }
128
129
```

```
130
    /*=========================================================================
    ==========*
131  * mm_init *
132
    *=========================================================================
    =========*/
133  PRIVATE void mm_init()
134  {
135  /* Inicializa o gerênciador de memória */
136
137  static char core_sigs[] = {
138  SIGQUIT, SIGILL, SIGTRAP, SIGABRT,
139  SIGEMT, SIGFPE, SIGUSR1, SIGSEGV,
140  SIGUSR2, 0 };
141  register int proc_nr;
142  register struct mproc *rmp;
143  register char *sig_ptr;
144  phys_clicks ram_clicks, total_clicks, minix_clicks,
    free_clicks, dummy;
145  message mess;
146  struct mem_map kernel_map[NR_SEGS];
147  int mem;
148
149             /*
150             * Constrói um conjunto de sinais que causam
    duplicações de núcleo.
151             * Faz isso da maneira do POSIX, então nenhum
    conhecimento de posições
152             * bit é necessário
153             */
154
155  sigemptyset(&core_sset);
156  for (sig_ptr = core_sigs; *sig_ptr != 0; sig_ptr++)
157  sigaddset(&core_sset, *sig_ptr);
158
159              /*
160              * Obtém o mapa de memória do kernel para ver quanta
    memória isso usa,
161              * incluindo o gap entre o endereco 0 e o início do
    kernel
162              */
163  sys_getmap(SYSTASK, kernel_map);
164  minix_clicks = kernel_map[S].mem_phys + kernel_map[S].mem_len;
165
166             /* inicializa as tabelas de gerênciamento de memória */
167
168  for (proc_nr = 0; proc_nr <= INIT_PROC_NR; proc_nr++) {
169  rmp = &mproc[proc_nr];
170  rmp->mp_flags |= IN_USE;
171  sys_getmap(proc_nr, rmp->mp_seg);
172  if (rmp->mp_seg[T].mem_len != 0) rmp->mp_flags |= SEPARATE;
173  minix_clicks += (rmp->mp_seg[S].mem_phys +
```

```
    rmp->mp_seg[S].mem_len)
174  - rmp->mp_seg[T].mem_phys;
175 }
176  mproc[INIT_PROC_NR].mp_pid = INIT_PID;
177  sigemptyset(&mproc[INIT_PROC_NR].mp_ignore);
178  sigemptyset(&mproc[INIT_PROC_NR].mp_catch);
179  procs_in_use = LOW_USER + 1;
180
181              /*
182              * Espera por FS para enviar uma mensagem dizendo ao
    disco RAM o tamanho que vai on-line
183              */
184  if (receive(FS_PROC_NR, &mess) != OK)
185  panic("MM can't obtain RAM disk size from FS", NO_NUM);
186
187  ram_clicks = mess.m1_i1;
188
189            /* Inicializa as tabelas para toda a memória física. */
190
191  mem_init(&total_clicks, &free_clicks);
192
193         /* Mostra as informações de memória */
194
195  printf("\nMemory size =%5dK ", click_to_round_k(total_clicks));
196  printf("MINIX =%4dK ", click_to_round_k(minix_clicks));
197  printf("RAM disk =%5dK ", click_to_round_k(ram_clicks));
198  printf("Available =%5dK\n\n", click_to_round_k(free_clicks));
199
200         /* Diz ao FS para continuar. */
201
202  if (send(FS_PROC_NR, &mess) != OK)
203  panic("MM can't sync up with FS", NO_NUM);
204
205
206         /* Diz ao processo de memória onde minha tabela de
    processo está para o ps(1) */
207
208  if ((mem = open("/dev/mem", O_RDWR)) != -1) {
209  ioctl(mem, MIOCSPSINFO, (void *) mproc);
210  close(mem);
211
212  }
213  }
214
215              /*
216              * Este arquivo está preocupado com a alocação de
    memória e liberação
217              * de blocos de tamanho arbitrário de blocos de
    memória em nome das chamadas
218              * de sistema FORK e EXEC. Os dados principais da
    estrutura usada é a tabela
219              * de furos, que mantém uma lista de furos na
    memórria. Ele também é mantido
```

```
220               * classificado em ordem crescente de endereco de
    memória. Os enderecos contido
221               * refere-se a memória física, começando no endereco 0.
222               * (ou seja, eles não são relacionados ao início da
    gerencia de memória)
223               * Durante a inicialização do sistema, a parte de
    memória que contém os
224               * vetores de interrupção, kernel e gerência de
    memória são alocados para marcá-los
225               * como disponíveis e removidos da lista de buracos
226               * Os pontos de entrada para este arquivo são:
227               * The entry points into this file are:
228               * alloc_mem: aloca um dado pedaco de memória
229               * free_mem: libera um pedaco previamente alocado de
    memória
230               * mem_init: inicializa as tabelas quando a gerência
    de memória inicia
231               * max_hole: retorna o maior burado atualmente
    disponível
232               */
233
234    #include "mm.h"
235    #include <minix/com.h>
236
237    #define NR_HOLES 128       /* max # de entradas na tabela de
    buracos */
238    #define NIL_HOLE (struct hole *) 0
239
240    PRIVATE struct hole {
241    phys_clicks h_base;                        /* onde inicia o
    buraco? */
242    phys_clicks h_len;                         /* qual o tamanho do
    buraco? */
243    struct hole *h_next;                       /* ponteiro da
    próxima entrada na lista */
244    } hole[NR_HOLES];
245
246
247    PRIVATE struct hole *hole_head;            /* ponteiro para o
    primeiro buraco */
248    PRIVATE struct hole *free_slots;           /* ponteiro apra a
    lista de buracos não utilizadas na tabela */
249
250    FORWARD _PROTOTYPE( void del_slot, (struct hole *prev_ptr,
    struct hole *hp) );
251    FORWARD _PROTOTYPE( void merge, (struct hole *hp) );
252
253
254
    /*============================================================
    ==========*
255    * alloc_mem *
256
```

```
*===========================================================================*/
257  PUBLIC phys_clicks alloc_mem(clicks)
258  phys_clicks clicks; /* amount of memory requested */
259  {
260                /*
261                * Aloca um bloco de memória da lista livre usando first fit.
262                * O bloco consite na sequência de bytes contíguos, cujo comprimento
263                * em cliques é dado por cliques. Um ponteiro para o bloco é retornado.
264                * O bloco está sempre em um limite de cliques.
265                * Este procedimento é chamado quando memória é necessária pelo FORK ou EXEC
266                */
267
268   register struct hole *hp, *prev_ptr;
269   phys_clicks old_base;
270
271   hp = hole_head;
272   while (hp != NIL_HOLE) {
273   if (hp->h_len >= clicks) {
274                                 /* Encontrado um buraco grande o suficiente. Use-o. */
275   old_base = hp->h_base;      /* Relembre onde inicia */
276   hp->h_base += clicks;       /* pega um pedaco */
277   hp->h_len -= clicks;        /* idem */
278
279                                 /* Se um buraco está parcialmente usado, reduz o seu tamanho e retorna. */
280   if (hp->h_len != 0) return(old_base);
281
282                                 /* O buraco inteiro foi usado. Manipula a lista livre. */
283   del_slot(prev_ptr, hp);
284   return(old_base);
285   }
286
287   prev_ptr = hp;
288   hp = hp->h_next;
289   }
290   return(NO_MEM);
291   }
292
293
294
/*===========================================================================*
295    * free_mem *
296
*===========================================================================*/
```

```
297    PUBLIC void free_mem(base, clicks)
298    phys_clicks base;               /* endereco base do bloco a ser
       liberado */
299    phys_clicks clicks;             /* número de cliques para
       liberar */
300    {
301                /*
302                * Retorna um bloco de memória livre para lista de
       livres. Os parâmetros dizem
303                * onde o bloco inicia na memória física e qual o
       tamanho dele. O bloco é
304                * adicionado a lista de buracos. Se ele é contíguo
       com um buraco existente
305                * em ambas as extremidades, ele é fundido com o
       buraco ou buracos
306                */
307
308    register struct hole *hp, *new_ptr, *prev_ptr;
309
310    if (clicks == 0) return;
311    if ( (new_ptr = free_slots) == NIL_HOLE) panic("Hole table
       full", NO_NUM);
312    new_ptr->h_base = base;
313    new_ptr->h_len = clicks;
314    free_slots = new_ptr->h_next;
315    hp = hole_head;
316
317                /*
318                * Se este endereco de bloco é numericamente menor
       que o menor buraco atualmente
319                * disponível, ou se nenhum buraco está atualmente
       disponível, coloca este buraco
320                * na frente da lista de buracos.
321                */
322    if (hp == NIL_HOLE || base <= hp->h_base) {
323
324                /*  Bloco a ser liberado na frente da lista de
       buracos  */
325
326    new_ptr->h_next = hp;
327    hole_head = new_ptr;
328    merge(new_ptr);
329    return;
330    }
331
332                /* Bloco a ser retornado não vai a frente da lista de
       buracos. */
333
334    while (hp != NIL_HOLE && base > hp->h_base) {
335    prev_ptr = hp;
336    hp = hp->h_next;
337    }
338
```

```
339     /* Onde vai. Insere o bloco depois de 'prev_ptr'. */
340
341     new_ptr->h_next = prev_ptr->h_next;
342     prev_ptr->h_next = new_ptr;
343     merge(prev_ptr);    /* A sequência é 'prev_ptr', 'new_ptr', 'hp' */
344     }
345
346
347
        /*===========================================================
        ==========*
348     * del_slot *
349
        *===========================================================
        =========*/
350     PRIVATE void del_slot(prev_ptr, hp)
351     register struct hole *prev_ptr;         /* pointer to hole entry
        just ahead of 'hp' */
352     register struct hole *hp;               /* pointer to hole entry
        to be removed */
353     {
354                                 /*
355                                 * Remove uma entrada da lista de
        buracos. Este procedimento é chamado quando um
356                                 * chamada para alocar memória remove
        um buraco na sua totalidade, reduzindo
357                                 * assim, o número de buracos na
        memória e a eliminação de uma entrada na lista de buracos.
358                                 */
359
360     if (hp == hole_head)
361     hole_head = hp->h_next;
362     else
363     prev_ptr->h_next = hp->h_next;
364
365     hp->h_next = free_slots;
366     free_slots = hp;
367     }
368
369
370
        /*===========================================================
        ==========*
371     * merge *
372
        *===========================================================
        =========*/
373     PRIVATE void merge(hp)
374     register struct hole *hp; /* ponteiro para o buraco para juntar
        com seus sucessores */
375     {
376
377                 /*
```

```
378             * Verifica se há buracos e junta os encontrados.
    Buracos contíguos
379             * podem ocorer quando um bloco de memória está
    livre, e isso acontece
380             * para encostar em outro buraco uma ou ambas as
    extremidades.
381             * O ponteiro 'hp' aponta para o primeiro de uma
    série de buraco que
382             * potencialmente podem estar todos juntos
383             */
384
385  register struct hole *next_ptr;
386
387             /* Se 'hp' aponta para o último buraco, não é
    possível juntá-lo.
388             * Se não, tenta absorver seu sucessor e libera a
    entrada da tabela do sucessor.
389             */
390
391  if ( (next_ptr = hp->h_next) == NIL_HOLE) return;
392  if (hp->h_base + hp->h_len == next_ptr->h_base) {
393  hp->h_len += next_ptr->h_len; /* primeiro recebe o segundo */
394  del_slot(hp, next_ptr);
395  } else {
396  hp = next_ptr;
397  }
398
399             /* Se 'hp' agora aponta para o último buraco,
    retorna; De outra maneira,
400             * tenta absorver seu sucessor.
401             */
402  if ( (next_ptr = hp->h_next) == NIL_HOLE) return;
403  if (hp->h_base + hp->h_len == next_ptr->h_base) {
404  hp->h_len += next_ptr->h_len;
405  del_slot(hp, next_ptr);
406  }
407 }
408
409
410
    /*=============================================================
    ==========*
411  * max_hole *
412
    *=============================================================
    =========*/
413  PUBLIC phys_clicks max_hole()
414  {
415             /* Verifica a lista de buracos e retorna o maior
    deles. */
416
417  register struct hole *hp;
418  register phys_clicks max;
```

```
419
420     hp = hole_head;
421     max = 0;
422     while (hp != NIL_HOLE) {
423     if (hp->h_len > max) max = hp->h_len;
424     hp = hp->h_next;
425     }
426     return(max);
427     }
428
429
430
    /*==========================================================================*
431     * mem_init *
432
    *==========================================================================*/
433     PUBLIC void mem_init(total, free)
434     phys_clicks *total, *free; /* resumos de tamanho de memória */
435     {
436                 /* Inicializa a lista de buracos. São duas listas:
    'hole_head' aponta
437                 * para uma lista encadeada de buracos (memória não
    usada) no sistema;
438                 * 'free_slots' aponta para uma lista encadeada de
    entradas de tabela que
439                 * não estão em uso.
440                 * Inicialmente, a antiga lista tem uma entrada para
    cada pedaco de memória física
441                 * e a segunda lista encadeada junta o restante da
    tabela de slots.
442                 * Como a memória se torna mais fragmentado no
    decorrer do tempo (ou seja, os grandes
443                 * buracos iniciais se dividem em buracos menores),
    novos slots de tabela são
444                 * necessários para representá-los. Esses slots são
    retirados da lista
445                 * liderada por 'free_slots'
446                 */
447
448     register struct hole *hp;
449     phys_clicks base;   /* endereco base */
450     phys_clicks size;   /* tamnaho */
451     message mess;
452
453                     /* Coloca todos os buracos na lista */
454
455     for (hp = &hole[0]; hp < &hole[NR_HOLES]; hp++) hp->h_next = hp
    + 1;
456     hole[NR_HOLES-1].h_next = NIL_HOLE;
457     hole_head = NIL_HOLE;
458     free_slots = &hole[0];
```

```
459
460                     /*
461                     * Pergunte ao kernel por pedacoes de memória
    física e alocar um buraco para
462                     * cada um deles. A chamada SYS_MEM responde
    com a base e tamanho do próximo
463                     * pecaco e a quantidade total de memória.
464                     */
465     *free = 0;
466     for (;;) {
467     mess.m_type = SYS_MEM;
468     if (sendrec(SYSTASK, &mess) != OK) panic("bad SYS_MEM?", NO_NUM);
469     base = mess.m1_i1;
470     size = mess.m1_i2;
471     if (size == 0) break; /* sem mais? */
472
473     free_mem(base, size);
474     *total = mess.m1_i3;
475     *free += size;
476     }
477     }
478
```