ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO

Departamento de Engenharia de Computação e Sistemas Digitais

PCS2042 – Sistemas Operacionais

Projeto 4
Alocação de Memória a Processos

Grupo 1:

Albert Kuniyoshi
Bruno Umeda Grisi
Emílio Pietro Rosseto

Professor: Jorge Kinoshita

Índice

1. Objetivo

O objetivo principal desta atividade é compreender o gerenciamento de memória realizado pelo Minix 3, particularmente no que se refere à alocação e desalocação de memória a processos.

A seguir, segue o mapa das atividades sugeridas e realizadas neste projeto sobre gerenciamento de memória:

- Mostrar através de um log como a memória foi alocada e desalocada a processos
- Toda vez que um processo executa uma das chamadas de sistema fork, exec ou exit, é necessário alocar ou desalocar memória, o que deve ser registrado no log
- No log, as informações são apresentadas da seguinte maneira:
    - fork: processo /usr/bin/init + segmento 0x30 - 5 clicks.
    - exec: processo /usr/bin/firefox + segmento 0x40 - 7 clicks.
    - exit: processo /usr/bin/firefox - segmento 0x40 - 7 clicks.

      onde:
        - +: significa que memória está sendo alocada ao processo da posição 0x00230 até 0x00340
        - -: significa que memória está sendo desalocada e devolvida para a área livre
- Adicione ao log a informação de quais tipos de segmento (dados, pilha ou texto/código) foram alocados ou desalocados

2. Log sobre a Alocação da Memória

2.1. Contextualização ao Código-Fonte do Minix 3

As chamadas de sistema fork, exec e exit estão localizadas no diretório do Gerenciador de Processos (**pm**), mais especificamente nos seguintes arquivos: *forkexit.*c e *exec.c*.

O arquivo *forkexit.*c (path: **/usr/src/servers/pm**) trata da criação de processos (via FORK) e de sua exclusão (via EXIT/WAIT). Quando um processo realiza fork, uma nova entrada na tabela *mproc* é alocada para ele e uma cópia da imagem do núcleo do pai é feita para o filho. Então, o núcleo e o sistema de arquivos são informados. Um processo será removido da tabela *mproc* quando dois eventos tiverem ocorrido:

- Processo saiu ou foi eliminado por um sinal
- Processo-pai executou uma operação WAIT

Se o processo sai primeiro, ele continua a ocupar uma entrada até que o pai execute uma operação WAIT.

Os pontos de entrada para o arquivo *forkexit.c* são:

- do_fork: executa a chamada se sistema FORK

- do_pm_exit: executa a chamada se sistema EXIT (chamando a função *pm_exit()*)
- pm_exit: realiza a saída realmente
- do_wait: executa a chamada se sistema WAITPID ou WAIT

O arquivo *exec.c* (path: **/usr/src/servers/pm**) manipula a chamada de sistema EXEC e executa a seguinte seqüência de tarefas:

1) Verifica se as permissões deixam que o arquivo seja executado
2) Busca os argumentos iniciais e o ambiente do espaço de usuário
3) Aloca a memória para o novo processo
4) Copia a pilha inicial do pm no processo
5) Lê os segmentos de texto e de dados e os copia no processo
6) Cuida dos bits *setuid* e *setgid*
7) Informa o núcleo sobre EXEC
8) Salva o deslocamento no argumento inicial

Os pontos de entrada para o arquivo *exec.c* são:

- do_exec: executa a chamada de sistema EXEC
- rw_seg: lê ou escreve um segmento em um arquivo
- find_share: encontra um processo cujo segmento de texto pode ser compartilhado

Algumas estruturas de dados utilizadas nesses arquivos são importantes para uma melhor compreensão do código:

- *struct mproc* *rmp: ponteiro referente ao processo-pai (processo apontado por *mproc* ou processo corrente)
- *struct mproc* *rmc: ponteiro referente ao processo-filho

Todas as informações de gerenciamento de processo para cada processo ficam armazenadas na tabela *mproc.h* (path: **/usr/src/servers/pm**). Esta tabela define os segmentos de texto, dados e pilha, uids, gids e vários flags para cada processo. Da mesma forma, o núcleo e o sistema de arquivos têm tabelas que também são indexadas pelo processo com o conteúdo das entradas correspondentes se referindo ao mesmo processo em todos os três.

A estrutura de dados *mproc* (path: **/usr/src/servers/pm/mproc.h**) está apresentada na Tabela 1 que destaca os atributos utilizados no arquivo de log a ser criado.

```
EXTERN struct mproc {
  struct mem_map mp_seg[NR_LOCAL_SEGS]; /* aponta para texto, dados,
pilha */
  char mp_exitstatus;            /* storage for status when process exits */
  char mp_sigstatus;             /* storage for signal # for killed procs */
  pid_t mp_pid;                  /* process id */
  int mp_endpoint;          /* kernel endpoint id */
  pid_t mp_procgrp;         /* pid of process group (used for signals) */
  pid_t mp_wpid;            /* pid this process is waiting for */
```

```
  int mp_parent;          /* index of parent process */

  /* Child user and system times. Accounting done on child exit. */
  clock_t mp_child_utime; /* cumulative user time of children */
  clock_t mp_child_stime; /* cumulative sys time of children */

  /* Real and effective uids and gids. */
  uid_t mp_realuid;       /* process' real uid */
  uid_t mp_effuid;        /* process' effective uid */
  gid_t mp_realgid;       /* process' real gid */
  gid_t mp_effgid;        /* process' effective gid */

  /* File identification for sharing. */
  ino_t mp_ino;                 /* inode number of file */
  dev_t mp_dev;                 /* device number of file system */
  time_t mp_ctime;        /* inode changed time */

  /* Signal handling information. */
  sigset_t mp_ignore;           /* 1 means ignore the signal, 0 means don't
*/
  sigset_t mp_catch;            /* 1 means catch the signal, 0 means don't */
  sigset_t mp_sig2mess;         /* 1 means transform into notify message */
  sigset_t mp_sigmask;          /* signals to be blocked */
  sigset_t mp_sigmask2;         /* saved copy of mp_sigmask */
  sigset_t mp_sigpending; /* pending signals to be handled */
  struct sigaction mp_sigact[_NSIG + 1]; /* as in sigaction(2) */
  vir_bytes mp_sigreturn; /* address of C library __sigreturn function */
  struct timer mp_timer;  /* watchdog timer for alarm(2) */

  /* Backwards compatibility for signals. */
  sighandler_t mp_func;         /* all sigs vectored to a single user fcn */

  unsigned mp_flags;            /* flag bits */
  vir_bytes mp_procargs;       /* ptr to proc's initial stack arguments */
  struct mproc *mp_swapq; /* queue of procs waiting to be swapped in */
  message mp_reply;       /* reply message to be sent to one */

  /* Scheduling priority. */
  signed int mp_nice;           /* nice is PRIO_MIN..PRIO_MAX, standard 0. */

  char mp_name[PROC_NAME_LEN];      /* nome do processo */
}
```

Tabela 1 – Código-Fonte da Estrutura de Dados mproc

A estrutura de dados *mem_map* (path: **/usr/src/include/minix/type.h**), apresentada na Tabela 2, define o mapa de memória para os segmentos do sistema. Nesta Tabela, foram evidenciados os atributos da estrutura *mp_seg* (definida como *mem_map*) destacada na Tabela anterior. Note que estes atributos foram declarados como **clicks**. No Minix 3, cada click corresponde a 1024 bytes.

```
/* Memory map for local text, stack, data segments. */
struct mem_map {
  vir_clicks mem_vir;         /* virtual address */
  phys_clicks mem_phys;       /* physical address */
  vir_clicks mem_len;         /* length */
};
```

Tabela 2 – Código-Fonte da Estrutura de Dados mem_map

2.2. Criação do Log

A idéia para a construção de um log que apresenta informações sobre a alocação e a desalocação de memória realizadas pelos processos que executaram uma das chamadas de sistema FORK, EXEC e EXIT é modificar o código referente às funções que, de fato, executam estas chamadas de sistema de modo a escrever em um arquivo de log apropriado aquelas informações.

No Minix 3, os logs criados em tempo de execução estão localizados no diretório **log** (path: `/var/log`). Nele, foi criado o arquivo **memory.log** que corresponde ao log proposto neste projeto. O acesso a este arquivo é realizado pela função **open** que utiliza um caminho e flags indicadas na biblioteca *fcntl.h*.

Para escrever em *memory.log*, foram utilizadas as funções **sprintf** e **write**. Primeiramente, a função *sprintf* armazena em um buffer os dados passados como parâmetros de escrita. Em seguida, utiliza-se a função *write* que escreve uma string em um arquivo a partir do endereço do buffer de bytes e o tamanho de bytes que se deseja escrever.

Assim, incluímos as seguintes bibliotecas:

- stdio.h: controle de entrada e saída
- fcntl.h: função *open* e suas flags
- sys/types.h: declaração dos tipos utilizados no sistema

As informações que devem ser apresentadas no arquivo de log são:

- identificação da chamada de sistema
- nome do processo corrente
- símbolo que identifica se memória foi alocada (+) ou desalocada (-)
- endereço físico do segmento alocado
- comprimento em *clicks* do segmento
- comentário que identifica quais tipos do segmento (dados, pilha ou texto/código) foram alocados ou desalocados

Os tipos de segmento são encontrados no código-fonte do Minix 3 sob a forma de constantes no arquivo **const.h** (path: `/usr/src/include/minix`) cujo trecho correspondente está evidenciado na Tabela 3.

```
/* Memory related constants. */
...

#define LOCAL_SEG     0x0000     /* flags indicating local memory segment */
#define NR_LOCAL_SEGS      3     /* # local segments per process (fixed) */

/* constantes que identificam o tipo de segmento */
#define T                   0  /* proc[i].mem_map[T] is for text */
#define D                   1  /* proc[i].mem_map[D] is for data */
#define S                   2  /* proc[i].mem_map[S] is for stack */
```

Tabela 3 – Constantes dos Tipos de Segmento

No log proposto pelo projeto (*memory.log*), a informação sobre o tipo do segmento está na forma de comentário para identificar que tipo de segmento sofreu alocação ou desalocação. No código-fonte do Minix 3, o mapa de memória de um determinado tipo de segmento é obtido da seguinte maneira:

- m_proc -> m_segment[TYPE_SEG]

  onde:

  - m_proc: estrutura de dados *mproc* (vide Tabela 1)
  - m_seg[NR_LOCAL_SEGS]: estrutura de dados *mem_map* (vide Tabela 2) definida dentro da estrutura de dados *mproc*
  - TYPE_SEG: tipo do segmento
    - D: dados (data)
    - S: pilha (stack)
    - T: texto (text)

O bloco básico, descrito na Tabela 4, resume os comandos explicados anteriormente para a criação do arquivo de log, sendo utilizado em cada uma das funções referentes às chamadas de sistema especificadas no projeto.

```
/* criação e abertura do arquivo .log  */
/* log é uma variável que identificará o arquivo .log */
log = open("/var/log/memory.log", O_APPEND|O_CREAT|O_WRONLY);

/* tam é o identificador de onde será guardada a string */
tam = sprintf(buf, "<nome da syscall> <símbolo de inclusão ou retirada
da memória (+ ou -)> processo %s + segmento 0x%lx - %d clicks\n",
<processo>->mp_name, <processo>->mp_seg[D].mem_phys,
<processo>->mp_seg[D].mem_len);

/* escreve no arquivo .log */
write(log, buf, tam);
```

Tabela 4 – Código para a Criação do Log

2.3. Alteração do Código

2.3.1. Chamada de Sistema FORK (*forkexit.c*)

O padrão do Minix 3 para a alocação de memória é utilizar o conceito de **texto compartilhado**, no qual os segmentos de dados e pilha são separados do segmento de texto (ou código) de forma a possibilitar o compartilhamento deste texto com diversos processos que o utilizem, sem a necessidade de alocar mais memória para uma informação que já existe.

Convém explicitar que é possível configurar um processo para que ele não utilize este padrão de **texto compartilhado**, tratando os segmentos de pilha, dados e texto juntos. Essa configuração pode ser efetuada, alterando o valor da **flag SEPARATE** que está definida na tabela *mproc*.

No caso do FORK, o Minix 3 considera que é necessária a alocação apenas para dados e pilha, já que o segmento de texto é compartilhado (padrão). No caso em que foi modificado o padrão, o endereço de memória físico do segmento de texto aponta para o endereço do processo pai, mas nenhuma alocação adicional é realizada!

Portanto, toda a chamada de sistema FORK aloca memória apenas para os segmentos de dados e de pilha no Minix 3.

```
/* Determine how much memory to allocate.  Only the data and
stack need to be copied, because the text segment is either
shared or of zero length */
prog_clicks = (phys_clicks) rmp->mp_seg[S].mem_len;
prog_clicks += (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
prog_bytes = (phys_bytes) prog_clicks << CLICK_SHIFT;
if ( (child_base = alloc_mem(prog_clicks)) == NO_MEM)
    return(ENOMEM);

/* A separate I&D child keeps the parents text segment */
/* The data and stack segments must refer to the new copy */
if (!(rmc->mp_flags & SEPARATE))
    rmc->mp_seg[T].mem_phys = child_base;
rmc->mp_seg[D].mem_phys = child_base;
rmc->mp_seg[S].mem_phys = rmc->mp_seg[D].mem_phys +
                (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
rmc->mp_exitstatus = 0;
rmc->mp_sigstatus = 0;
```

Tabela 5 – Padrão do Minix 3 para Alocação de Memória em FORK

A escrita das informações relativas ao gerenciamento de memória realizado pela chamada de sistema FORK no arquivo de log foi feita pela adição do código, representado pela Figura 1, dentro da função **do_fork** no arquivo *forkexit.c*.

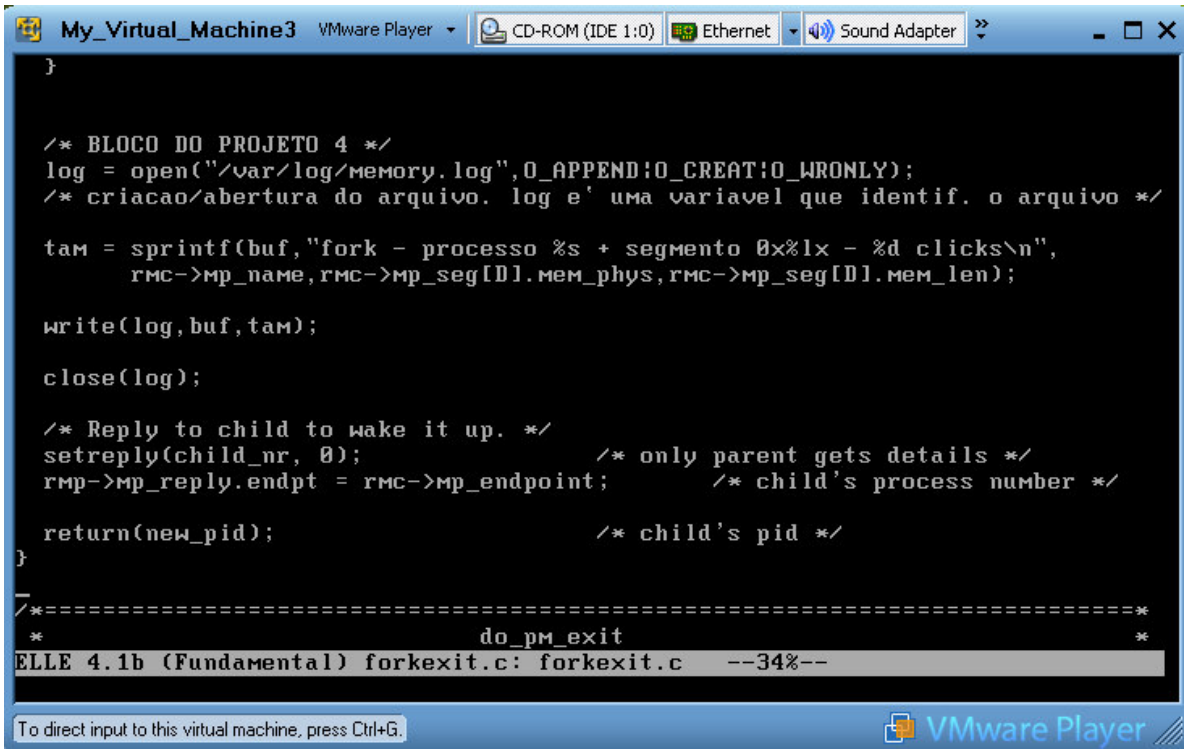

Figura 1 – Modificação do Código em do_fork

2.3.2. Chamada de Sistema EXEC (exec.*c*)

A escrita das informações relativas ao gerenciamento de memória realizado pela chamada de sistema EXEC no arquivo de log foi feita pela adição do código, representado pela Figura 2, dentro da função **do_exec** no arquivo *exec.c*.

Note que a informação quanto ao tipo de segmento alocado durante a execução da chamada de sistema não está definida neste código. Aqui, são definidos o nome, o endereço de memória físico e o comprimento do processo criado.

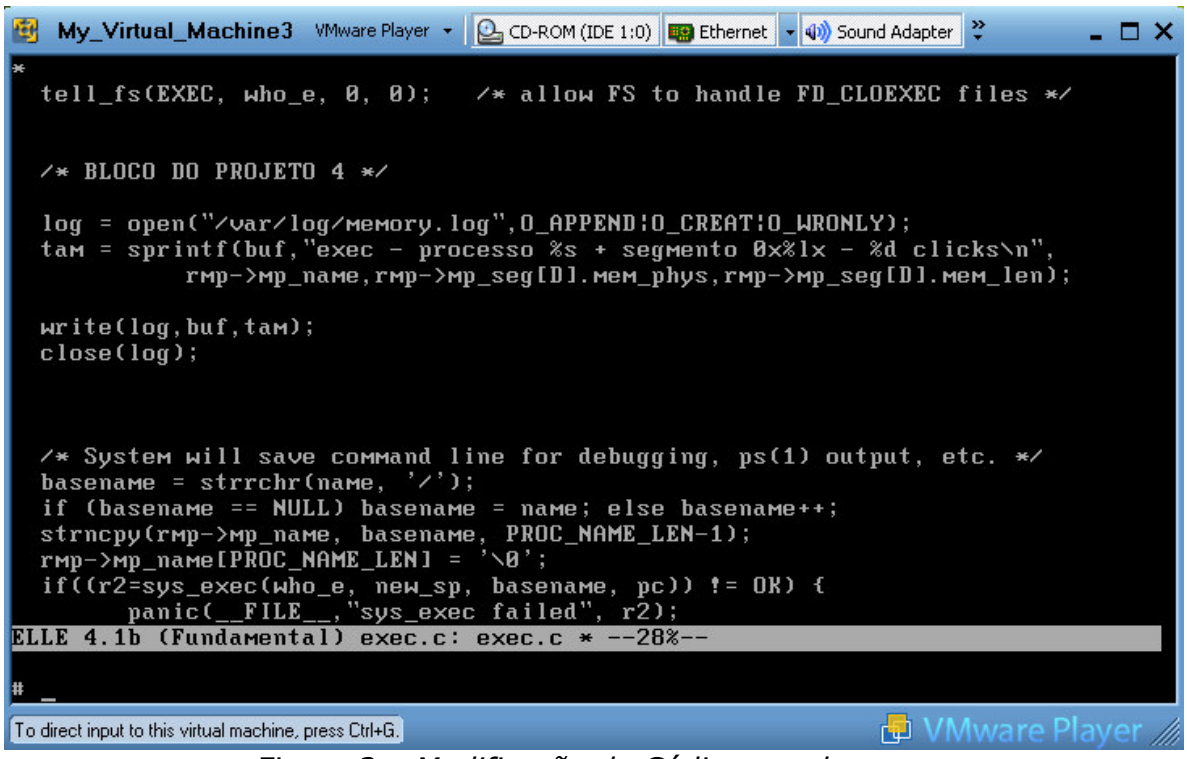

Figura 2 – Modificação do Código em do_exec

A informação do tipo de segmento que foi alocado durante a execução da EXEC é obtida na função **new_mem** do arquivo *exec.c*. Esta função fica responsável por verificar se existe texto para ser compartilhado e por alocar memória para o processo de acordo com a estrutura de segmentação do Minix 3 para o gerenciamento de memória.

O código que escreve no arquivo de log esta informação está representado pela Figura 3.

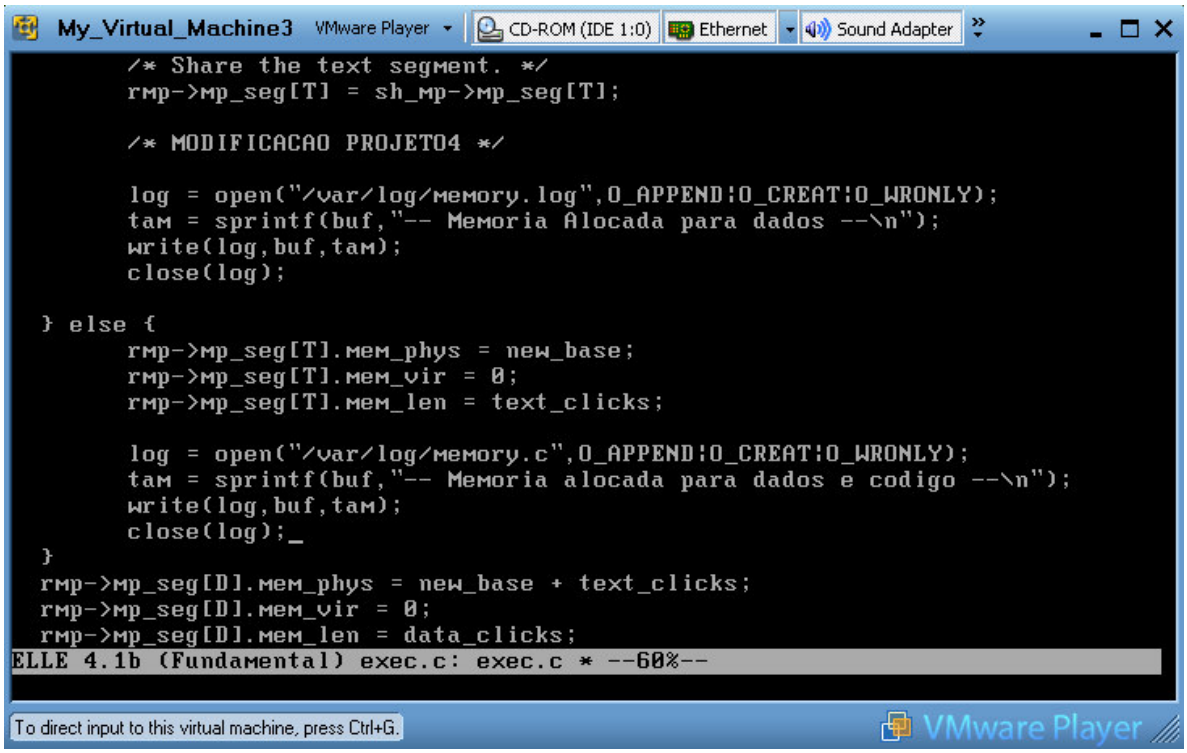

Figura 3 – Modificação do Código em new_mem

2.3.3. Chamada de Sistema EXIT (forkexit.c)

A chamada de sistema EXIT é executada pela função **pm_exit** do arquivo *forkexit.c* (o mesmo arquivo em que é tratada a FORK). Esta função fica responsável por verificar se existe algum processo que compartilha o texto do processo corrente e por desalocar memória dos segmentos de pilha, dados e texto não-compartilhado.

A parte do código de *pm*_exit que trata a desalocação de memória do processo corrente está representada pela Tabela 6.

```
/* Release the memory occupied by the child */
if (find_share(rmp, rmp->mp_ino, rmp->mp_dev, rmp->mp_ctime) == NULL)
{
      /* No other process shares the text segment, so free it */
        free_mem(rmp->mp_seg[T].mem_phys, rmp->mp_seg[T].mem_len);
}
/* Free the data and stack segments */
free_mem(rmp->mp_seg[D].mem_phys, rmp->mp_seg[S].mem_vir
                + rmp->mp_seg[S].mem_len - rmp->mp_seg[D].mem_vir);
```

Tabela 6 – Desalocação de Memória em EXIT

A escrita das informações relativas ao gerenciamento de memória realizado pela chamada de sistema EXIT no arquivo de log foi feita pela adição do código, representado pela Figura 4, dentro da função **pm_exit** no arquivo *forkexit.c*.

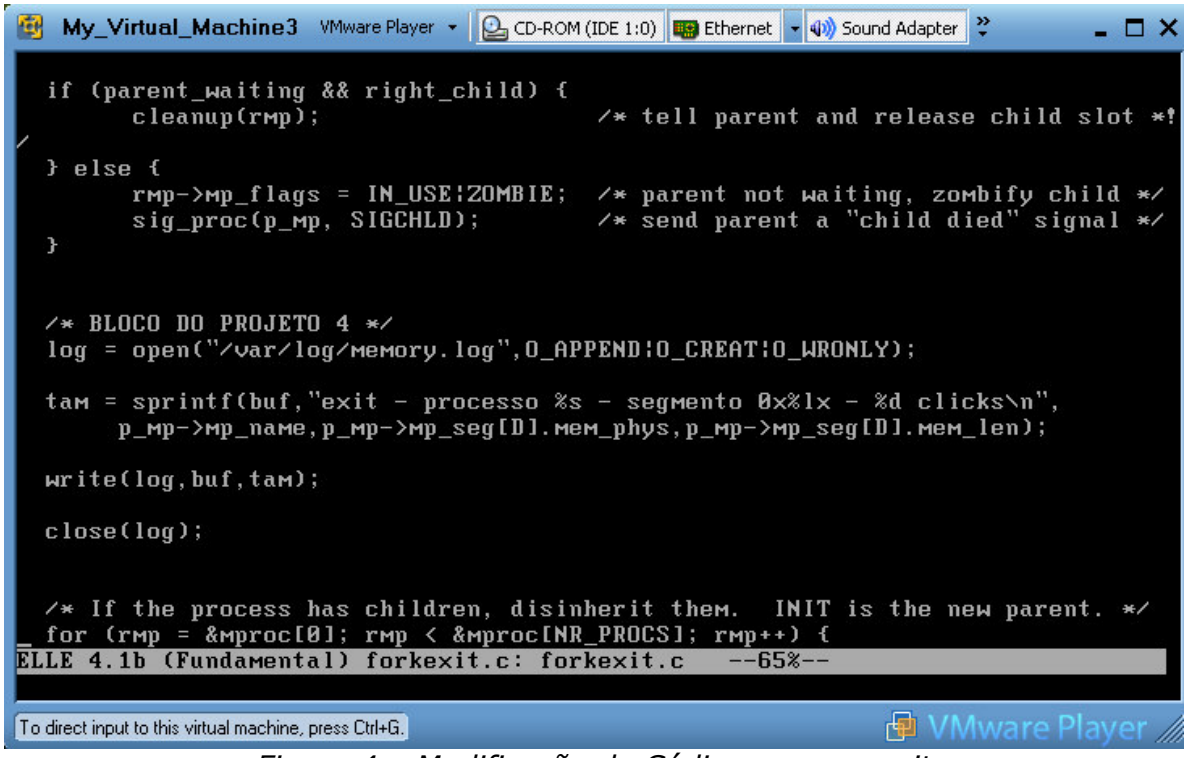

Figura 4 – Modificação do Código em pm_exit

2.4. Resultados

Após a alteração do código das chamadas de sistema que se relacionam ao gerenciamento de memória, foram realizados alguns testes para verificar se, de fato, foi criado o arquivo de log proposto.

Executou-se um boot do sistema para a subida da nova imagem criada e, logo em seguida, abriu-se o *memory.log* em seu respectivo diretório (*/var/log*) a fim de se checar todas as chamadas de sistema que foram executadas durante o processo de boot. Assim, foi encontrado um arquivo de log bem extenso com todas as informações relativas à alocação de memória. Na Figura 5, é possível visualizar parte do conteúdo do log gerado.

Note que foram identificados casos diversos de alocação de memória quanto ao tipo de segmento alocado ou desalocado para cada chamada de sistema executada.

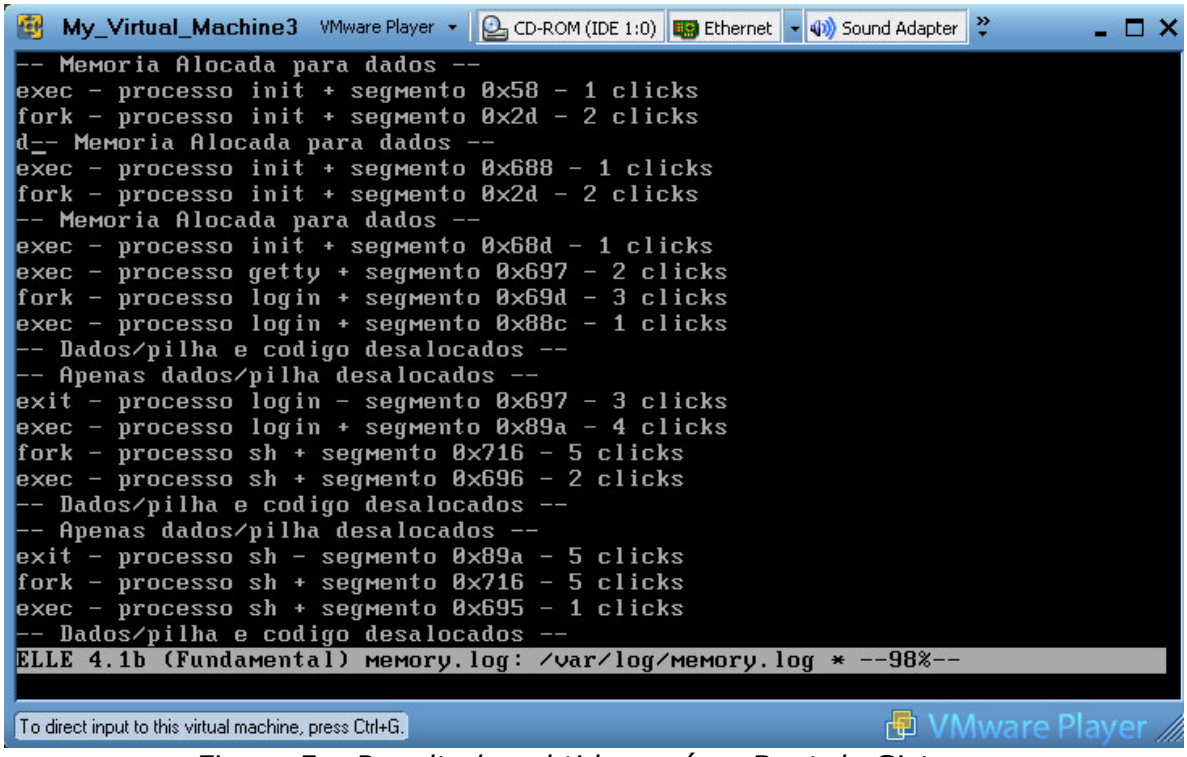

Figura 5 – Resultados obtidos após o Boot do Sistema

3. Conclusão

Este trabalho consolidou o aprendizado sobre o gerenciamento de memória realizado pelo Minix 3, particularmente no que se refere à alocação e desalocação de memória a processos.

Objetivando a realização das tarefas sugeridas, estudamos mais detalhadamente o código-fonte do Minix 3, o que proporcionou um melhor entendimento acerca de sua estrutura e funcionamento, notadamente o gerenciador de processos (pm).

Conforme foi descrito no objetivo deste relatório, criamos um arquivo de log que armazena as informações relativas aos processos quanto à memória que foi alocada ou desalocada. A escrita no log ocorria no momento em que as chamadas de sistema FORK, EXEC e EXIT eram executadas.

4. Bibliografia

1. TANENBAUM, Andrew S.; WOODHULL, Albert S. Sistemas Operacionais: Projeto e Implementação, 3ª edição
2. Código-Fonte do Minix 3
3. http://www.raspberryginger.com/jbailey/minix/html/mproc_8h-source.html
4. http://en.wikipedia.org/wiki/RAM_disk