Universidade Federal de São Carlos
Departamento de Computação

Segunda Prova de SO1

Professora Dra. Kelen Vivaldini

Bruna Zamith Santos (RA: 628093)
Henrique Cordeiro Frajacomo (RA: 726536)
João Victor Pacheco (RA: 594970)
Marcos Augusto Faglioni Junior (RA: 628301)

07/2018
São Carlos - SP, Brasil

Conteúdo

1 Introdução

O presente relatório visa detalhar os procedimentos realizados para a solução da segunda prova de Sistemas Operacionais 1 (SO1), disciplina ministrada pela professora Dra. Kelen Vivaldini, ao longo do primeiro semestre de 2018. A prova trata de *Memory Managment* (Gerenciamento de Memória) no xv6[1]. O xv6 é um sistema operacional baseado em Unix desenvolvido pelo MIT, com a finalidade de ser usado para aprendizado em cursos de SO.

O relatório está apresentado como segue: Na Seção 2, apresentamos alguns conceitos importantes para o entendimento do xv6; a Seção 3 introduz e responde perguntas introdutórias, relevantes para posterior solução das *tasks*, descritas e solucionadas na Seção 4; a Seção 5 serve como um apêndice e indica quais arquivos foram alterados e quais funções foram criadas, por *task*.

As imagens aqui apresentadas, assim como os conceitos e definições detalhados, foram obtidos das referências sobre xv6 do CSE IIT Bombay[2][3] e do MIT[4].

[1] `https://pdos.csail.mit.edu/6.828/2012/xv6.html`

[2] `http://www.cse.iitd.ernet.in/~sbansal/os/previous_years/2013/xv6-html/files.html`

[3] `https://www.cse.iitb.ac.in/~mythili/teaching/cs347_autumn2016/notes/08-xv6-memory.pdf`

[4] `https://pdos.csail.mit.edu/6.828/2012/xv6.html`

2 Conceitos importantes

Para podermos executar o trabalho, tivemos que buscar entender o funcionamento do xv6 no que tange o gerenciamento de sua memória. A seguir estão expostos os conceitos entendidos, que serão aplicados para respondermos as perguntadas introdutórias (Seção 3) e nas resoluções das *tasks* (Seção 4).

Cada processo do xv6 é determinado por um endereço virtual, o qual pode ser significantemente maior que a memória física. Para traduzirmos cada um desses endereços virtuais em um endereço físico, cada processo é organizado em *frames* e possui sua própria *Page Table*.

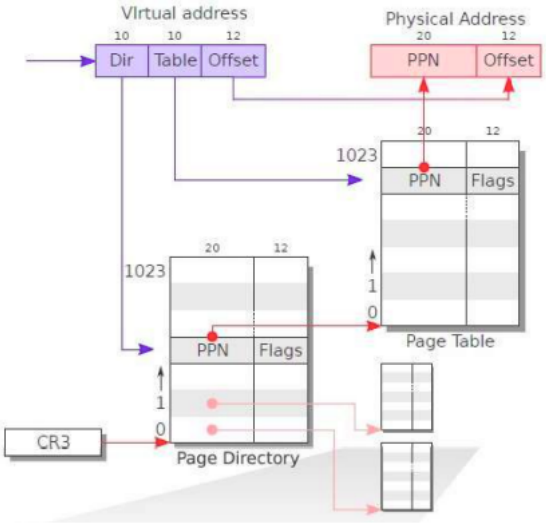


Figura 1: Esquema de gerenciamento de memória no xv6

Como pode ser visto na Figura 1, cada processo aponta para uma entrada do *Page Directory Table*. Chamamos essas entradas de *Page Tables*. Dado que um processo pode possuir vários segmentos na memória física, uma *Page Table* também contém várias entradas.

Tanto o *Page Directory Table* quanto o *Page Table* contêm um *Physical Page Number* (PPN) e algumas *flags*. As principais flags estão descritas na Figura 2.

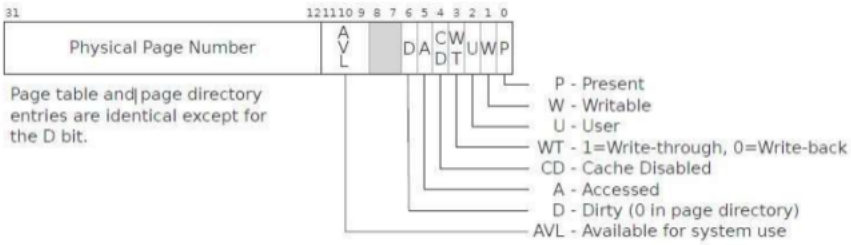


Figura 2: PPN e *flags*

3 Perguntas Introdutórias

- ***How does the kernel know which physical pages are used and unused?***

De acordo com as *flags* presentes no *Page Directory Table*.

- ***What data structures are used to answer this question?***

A estrutura de dados utilizada é *Segment Descriptor* (Descritor de Segmentos). Mais especificamente, o **struct segdesc**. Essa estrutura guarda informações como endereço de base, tipo de segmento (executável, leitura ou escrita), nível de privilégio, limite de segmento; e a variável **AVL**, que indica se a página está disponível ou não (sendo usada ou não).

- ***Where do these reside?***

A estrutura reside no *Page Directory Table* e, mais especificamente, sua declaração pode ser encontrada no **struct segdesc** do arquivo **mmu.h**.

- ***Does xv6 memory mechanism limit the number of user processes?***

Sim, ele limita para 64 processos. Isto está definido na constante **NPROC** do arquivo **param.h**.

- ***What is the lowest number of processes xv6 can 'have' at the same time?***

Existem dois processos que estão sempre sendo executados no xv6 - o **init** (inicialização do sistema) e o **sh** (a linha de comando *shell*, que recebe a entrada do usuário). Isso fica evidente quando pressionados as teclas Ctrl + P imediatamente após a inicialização do sistema:

```
1 $ 1 sleep init 80103ec7 80104879 80105835 8010564f
2 $ 2 sleep sh 80103dec 801002ca 80100f9c 80104b62 80104879 80105835 8010564f
```

4 Tasks

4.1 Task 1 - Enhancing process details viewer

O objetivo dessa *task* é aumentar a quantidade de informações a serem exibidas após o pressionamento das teclas Ctrl + P no *shell* do xv6. As informações mostradas originalmente são o PID, o nome do processo, seu estado atual e os endereços virtuais dos segmentos em hexadecimal:

```
1 sleep  init 80103e27 80103ec7 80104879 80105835 8010564f
2 sleep  sh 80103e27 80103ec7 80104879 80105835 8010564f
3 run    stressfs
4 sleep  stressfs 80103dec 80102bda 8010107b 80104bd2 80104879 80105835 8010564f
5 sleep  stressfs 80103dec 80102bda 8010107b 80104bd2 80104879 80105835 8010564f
6 sleep  stressfs 80103dec 80102bda 8010107b 80104bd2 80104879 80105835 8010564f
7 sleep  stressfs 80103dec 80102bda 8010107b 80104bd2 80104879 80105835 8010564f
```

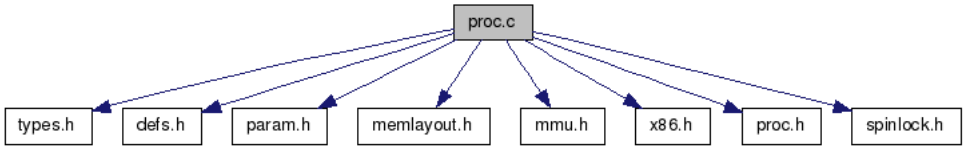
As informações a serem acrescentadas são, para cada processo:

- Local da memória no *Page Directory*;
- Local da memória no *Page Table*;
- *Pagging mapping*.

4.1.1 Arquivos Modificados

Para realizarmos as modificações anteriormente expostas, é preciso modificar apenas um arquivo, o **proc.c**. Este arquivo contém as definições das funções referentes aos procedimentos. Dentro dele, existe a função **procdump()**, a qual é responsável por determinar o que ocorre com o pressionamento das teclas Ctrl + P no *shell*. Ela é chamada pela função **consoleintr()** do arquivo **console.c**. Ou seja, quando pressionamos Ctrl + P no *shell*, a função **consoleintr()** identifica tal entrada e chama a função **procdump()**.

O arquivo **proc.c** importa algumas bibliotecas:



As bibliotecas de interesse para nossa modificação são a **mmu.h** e a **memorylayout.h**. Isso porque a **mmu.h** através das constantes **NPDENTRIES** e **NPTENTRIES** nos informa o número de elementos na *Page Directory Table* e o número de *Page Table Entries* em cada *Page Table*, respectivamente; e a **memorylayout.h** nos fornece as funções de conversão **V2P()** e **P2V()**. Também nesse arquivo estão definidas as *flags* do *Page Table* e do *Page Directory*.

4.1.2 Função antes da modificação

A seguir apresentamos a função **procdump()** original, antes de qualquer modificação:

Listing 1: Função **procdump()** original

```
1  void procdump(void)
2  {
3    static char *states[] = {
4    [UNUSED]    "unused",
5    [EMBRYO]    "embryo",
6    [SLEEPING]  "sleep ",
7    [RUNNABLE]  "runble",
8    [RUNNING]   "run   ",
9    [ZOMBIE]    "zombie"
10   };
11   int i;
12   struct proc *p;
13   char *state;
14   uint pc[10];
15
16   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
17     if(p->state == UNUSED)
18       continue;
19     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
20       state = states[p->state];
21     else
22       state = "???";
23     cprintf("%d %s %s", p->pid, state, p->name);
24     if(p->state == SLEEPING){
25       getcallerpcs((uint*)p->context->ebp+2, pc);
26       for(i=0; i<10 && pc[i] != 0; i++)
27         cprintf(" %p", pc[i]);
28     }
29     cprintf("\n");
30   }
31 }
```

4.1.3 Implementação

A seguir apresentamos a função **procdump()** editada, após as modificações para que a mesma exibisse mais informações sobre os processos.

Listing 2: Função **procdump()** editada

```
1  void procdump(void)
2  {
3    static char *states[] = {
4    [UNUSED]    "unused",
5    [EMBRYO]    "embryo",
6    [SLEEPING]  "sleep ",
7    [RUNNABLE]  "runble",
8    [RUNNING]   "run   ",
9    [ZOMBIE]    "zombie"
10   };
11   int i, j, count;
12   struct proc *p;
13   char *state;
14   uint pc[10];
15
16   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
17     if(p->state == UNUSED)
```

```
18        continue;
19      if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
20        state = states[p->state];
21      else
22        state = "???";
23      cprintf("PID = %d \nEstado = %s \nNome = %s\n", p->pid, state, p->name);
24      if(p->state == SLEEPING){
25        getcallerpcs((uint*)p->context->ebp+2, pc);
26        for(i=0; i<10 && pc[i] != 0; i++){
27          cprintf("%p ", pc[i]);
28         }
29      }
30
31    cprintf("\n\nInformacoes Adicionais sobre o Processo %d:\n", p->pid);
32      cprintf("Page Tables:\n");
33      cprintf("Physical Directory Address = %x\n", V2P(p->pgdir));
34      for(i=0; i<NPDENTRIES; i++){
35        if((p->pgdir[i] & PTE_P) && (p->pgdir[i] & PTE_U)){
36            pte_t *pageTabVir = P2V((pte_t*)PTE_ADDR(p->pgdir[i]));
37            count = 0;
38            for(j=0; j<NPTENTRIES; j++){
39              if((pageTabVir[j] & PTE_P) && (pageTabVir[j] & PTE_U)){
40                count++;
41              }
42            }
43            if(count > 0){
44              cprintf("Numero de Paginas Utilizadas = %p\n",count);
45              cprintf("Virtual Page Entry = %p\n", (PTE_ADDR(p->pgdir[i]) >> 12));
46              cprintf("Physical Page Entry = %p\n", PTE_ADDR(p->pgdir[i]));
47            }
48        }
49
50      }
51      cprintf("\n");
52      cprintf("Page Mappings:\n");
53      for(i=0; i<NPDENTRIES; i++){
54        if((p->pgdir[i] & PTE_P) && (p->pgdir[i] & PTE_U)){
55          pte_t *pageTabVir = P2V((pte_t*)PTE_ADDR(p->pgdir[i]));
56          count = 0;
57          for(j=0; j<NPTENTRIES; j++){
58            if((pageTabVir[j] & PTE_P) && (pageTabVir[j] & PTE_U)){
59              count++;
60            }
61          }
62          if(count > 0){
63            cprintf("<Virtual> %p -> <Physical> %p\n",
64              (PTE_ADDR(P2V(PTE_ADDR(p->pgdir[i])))) >> 12,
65              ((int)PTE_ADDR(p->pgdir[i]) >> 12));
66          }
67        }
68      }
69      cprintf("\n");
70      cprintf("\n");
71
72    }
73  }
```

Começamos por alterar o código para especificar quais o PID, estado e nome do processo. Isso já

era feito, mas não de maneira clara. Podemos ver essa alteração na linha 23 do código.

Então, aproveitando o *loop* já implementado, o qual percorre cada um dos processo ativos, teremos resultados para todos os processos. Começamos pela impressão do PID do processo, o que é feito através da chamada **p->pid**. Depois, imprimimos as informações referentes às *Page Tables*. Obtemos o endereço físico do diretório através da chamada da função **V2P()**, que é definida no arquivo **memlayout.h** para conversão do endereço virtual para físico, sobre **p->pgdir**. O **p->pgdir** retorna o endereço virtual do *Page Directory* do processo, e está implementada na estrutura do processo em **proc.h**.

Então percorremos, a partir de um *loop*, todas as *Page Tables* apontadas na *Page Directory*, usando um contador que vai de zero a **NPDENTRIES**, definida em **mmu.h**. É feita, na linha 35, a verificação sobre as *flags* "*Present*"e "*User*", de modo a verificar se estao acionadas ou não. Se sim, na linha 39 temos uma nova verificação, desta vez para verificar quais páginas de uma determinada *Page Table* estão sendo usadas, analisando-se as mesmas *flags*. Para tal, percorremos de zero a **NPTENTRIES**. Se mais de uma páginas está sendo utilizada, printamos a entrada do endereço físico e a entrada do endereço virtual. Para o endereço virtual, precisamos fazer uso do $>> 12$ porque temos interesse apenas nos bits a partir do décimo segundo, como foi visto no segmento do *virtual address* na Seção 2.

Por fim, fazemos o *Page Mapping*. Primeiro são percorridas todas as entradas do *Page Directory*. O endereço virtual é coletado e armazenado. Armazenamos os *virtual page number* e *physical page number* a partir dos últimos 12 *bits* dos endereços virtual e físico, respectivamente. As verificações referentes às *flags* são feitas de maneira análoga às informações do *Page Table* descritas anteriormente.

4.1.4 Teste

Após as modificações explicitadas e apertando-se Ctrl+P, obtemos o seguinte resultado:

```
$ PID = 1
Estado = sleep
Nome = init
80103e27 80103ec7 80104a19 801059d5 801057ef

Informacoes Adicionais sobre o Processo 1:
Page Tables:
Physical Directory Address = dfbb000
                    Numero de Paginas Utilizadas = 2
                    Virtual Page Entry = df79
                    Physical Page Entry = df79000

Page Mappings:
                    <Virtual> 8df79 -> <Physical> df79


PID = 2
Estado = sleep
Nome = sh
80103dec 801002ca 80100f9c 80104d02 80104a19 801059d5 801057ef

Informacoes Adicionais sobre o Processo 2:
Page Tables:
Physical Directory Address = df73000
                    Numero de Paginas Utilizadas = 3
                    Virtual Page Entry = df31
                    Physical Page Entry = df31000

Page Mappings:
                    <Virtual> 8df31 -> <Physical> df31
```

4.2 Task 2 - Null pointer protection

O xv6 não possui proteção em casos de tentativa de referenciar ponteiros nulos (que acessam memória na localização 0). No xv6, a memória no endereço zero contém o segmento de texto do programa.

Para testar como o xv6 lida com ponteiros apontando para o endereço 0, criamos um arquivo **nullpointer.c**. Simplesmente tentamos acessar o valor apontado pelo ponteiro cujo endereço é 0. Seu código está descrito a seguir:

```
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4
5  int main(int argc, char *argv[]) {
6          char* ponteiro_nulo;
7          ponteiro_nulo = 0;
8          printf(1,"%x\n", *ponteiro_nulo);
9          exit();
10 }
```

E incluímos o procedimento **nullpointer()** no Makefile:

```
1  # Prevent deletion of intermediate files, e.g. cat.o, after first build, so
2  # that disk image changes after first build are persistent until clean.  More
3  # details:
4  # http://www.gnu.org/software/make/manual/html_node/Chained-Rules.html
5  .PRECIOUS: %.o
6
7  UPROGS=\
8          _cat\
9          _echo\
10         _forktest\
11         _grep\
12         _init\
13         _kill\
14         _ln\
15         _ls\
16         _mkdir\
17         _rm\
18         _sh\
19         _stressfs\
20         _usertests\
21         _nullpointer\
22         _wc\
23         _zombie\
```

Quando tentamos executar esse procedimento, o resultado é um erro, como exibido na Figura 3

```
$ nullpointer
pid 4 nullpointer: trap 6 err 0 on cpu 1 eip 0xa addr 0x0--kill proc
```

Figura 3: Reprodução do erro de acesso a um *null pointer*

O objetivo dessa *task* é bloquear o acesso ao endereço 0 e exibir uma mensagem de erro apropriada quando tal tentativa é feita. Para tal, precisamos fazer duas coisas: 1. Fazer com que a primeira página

não seja usada em nenhum momento pelo xv6; 2. Exibir uma mensagem de erro apropriada quando é feita a tentativa de acesso ao endereço de memória 0.

4.2.1 Arquivos modificados

No **Makefile**, encontramos o parâmetro **-Ttext** em algumas situações. Ele é responsável por determinar qual o menor endereço possível de segmento para cada situação. Como havíamos exposto anteriormente, o endereço zero originalmente contém o segmento de texto do programa. Queremos alterar o menor endereço possível para os arquivos .o. O código a seguir exibe o **-Ttext** antes das modificações.

```
%: %.o $(ULIB)
        $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^
        $(OBJDUMP) -S $@ > $*.asm
        $(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $*.sym
```

Precisamos alterar o valor de 0 (indicando a primeira página) para o valor 0x1000 (hexadecimal de 4096, indicando a segunda página):

```
%: %.o $(ULIB)
        $(LD) $(LDFLAGS) -N -e main -Ttext 0x1000 -o $@ $^
        $(OBJDUMP) -S $@ > $*.asm
        $(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $*.sym
```

Outro arquivo que precisa ser alterado é o arquivo **exec.c**. Isso porque ele é o arquivo responsável por carregar o processo na memória. Neste arquivo podemos encontrar o seguinte trecho de código dentro da função **exec()**:

```
  // Load program into memory.
  sz = 0;
  for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
      goto bad;
    if(ph.type != ELF_PROG_LOAD)
      continue;
    if(ph.memsz < ph.filesz)
      goto bad;
    if(ph.vaddr + ph.memsz < ph.vaddr)
      goto bad;
    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
      goto bad;
    if(ph.vaddr % PGSIZE != 0)
      goto bad;
    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
      goto bad;
  }
  iunlockput(ip);
  end_op();
  ip = 0;
```

A variável **sz** indica o tamanho (do inglês, sz - *size*) atual do espaço de memória alocado para o processo. Se ela começa em 0, então significa que a partir do endereço 0 já está sendo armazenado o programa na memória. Como queremos que comece apenas a partir da segunda página, vamos alterar

o **sz** para o valor correspondente, **PGSIZE - 1**. Sendo **PGSIZE** o tamanho da página definido no arquivo **mmuh.h**:

```
  // Load program into memory.
  sz = PGSIZE-1;
  for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
      goto bad;
    if(ph.type != ELF_PROG_LOAD)
      continue;
    if(ph.memsz < ph.filesz)
      goto bad;
    if(ph.vaddr + ph.memsz < ph.vaddr)
      goto bad;
    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
      goto bad;
    if(ph.vaddr % PGSIZE != 0)
      goto bad;
    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
      goto bad;
  }
  iunlockput(ip);
  end_op();
  ip = 0;
```

Ainda, precisamos editar o arquivo **vm.c**, cujo propósito é trabalhar com a *virtual memory*. Nele existe a função **copyuvm()**, a qual copia o conteúdo do processo pai (*parent*) para o processo filho (*child*), criado na execução do *fork*. O problema está na cópia do conteúdo do processo pai, que também copia a primeira página (vazia). O ponto inicial de cópia precisa ser alterado no *loop* que percorre o conteúdo do pai a partir da página 0, como pode ser visto a seguir:

```
pde_t* copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;
  char *mem;

  if((d = setupkvm()) == 0)
    return 0;
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
      panic("copyuvm: page not present");
    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto bad;
    memmove(mem, (char*)P2V(pa), PGSIZE);
    if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0)
      goto bad;
  }
  return d;

bad:
  freevm(d);
```

```
27    return 0;
28  }
```

Alteramos para que o *loop* começe a partir do **PGSIZE** (representa a segunda página):

```
1  pde_t* copyuvm(pde_t *pgdir, uint sz)
2  {
3    pde_t *d;
4    pte_t *pte;
5    uint pa, i, flags;
6    char *mem;
7
8    if((d = setupkvm()) == 0)
9      return 0;
10   //Alteracao do inicio do loop
11   //que realiza a copia, esta
12   //deve comecar na segunda pagina
13   //(PGSIZE) e entao criar a copia.
14   for(i = PGSIZE; i < sz; i += PGSIZE){
15     if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
16       panic("copyuvm: pte should exist");
17     if(!(*pte & PTE_P))
18       panic("copyuvm: page not present");
19     pa = PTE_ADDR(*pte);
20     flags = PTE_FLAGS(*pte);
21     if((mem = kalloc()) == 0)
22       goto bad;
23     memmove(mem, (char*)P2V(pa), PGSIZE);
24     if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0)
25       goto bad;
26   }
27   return d;
28
29 bad:
30   freevm(d);
31   return 0;
32 }
```

Percebemos, então que uma outra alteração precisava ser feita no **Makefile**. O parâmetro **-Ttext** do **fork_test** também precisava ser alterado de 0 para 0x1000, analogamente ao feito para o **-Ttext** dos arquivos .o. O código a seguir mostra antes da alteração:

```
1  _forktest: forktest.o $(ULIB)
2          # forktest has less library code linked in - needs to be small
3          # in order to be able to max out the proc table.
4          $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o _forktest forktest.o ulib.o usys.o
5          $(OBJDUMP) -S _forktest > forktest.asm
```

E o código a seguir, depois da alteração:

```
1  _forktest: forktest.o $(ULIB)
2          # forktest has less library code linked in - needs to be small
3          # in order to be able to max out the proc table.
4          $(LD) $(LDFLAGS) -N -e main -Ttext 0x1000 -o _forktest forktest.o ulib.o usys.o
5          $(OBJDUMP) -S _forktest > forktest.asm
```

Com essas alterações, a primeira página já não é mais utilizada. Agora, só precisamos apresentar uma mensagem de erro apropriada. Essa mensagem de erro pode ser alterada no arquivo **trap.c**, mais

especificamente no seguinte trecho de código da função **trap()**:

```
default:
    if(myproc() == 0 || (tf->cs&3) == 0){
      // In kernel, it must be our mistake.
      cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
              tf->trapno, cpuid(), tf->eip, rcr2());
      panic("trap");
    }
    // In user space, assume process misbehaved.
    cprintf("pid %d %s: trap %d err %d on cpu %d "
            "eip 0x%x addr 0x%x--kill proc\n",
            myproc()->pid, myproc()->name, tf->trapno,
            tf->err, cpuid(), tf->eip, rcr2());
    myproc()->killed = 1;
  }
```

Incluímos uma condicional para imprimir a mensagem de erro ”Acesso negado. Tentativa de acesso a pagina 0!”. A função **rcr2()** está definida no arquivo **x86.h** e retorna o conteúdo do registrador cr2. Esse registrador guarda o endereço da última tentativa de acesso no caso de um *Page Fault*. Assim, se ele armazenar o valor 0, significa que o usuário está tentando acessar a página 0. O **tf** é passado como parâmetro para a função de tratamento de erros e é do tipo *trapframe*, também definido no arquivo **x86.h**. O atributo **trapno** de **tf** armazena o código que indica qual o erro (*trap*) ocorrido, neste caso, 14.

```
default:
    if(myproc() == 0 || (tf->cs&3) == 0){
      // In kernel, it must be our mistake.
      cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
              tf->trapno, cpuid(), tf->eip, rcr2());
      panic("trap");
    }
    // In user space, assume process misbehaved.
    cprintf("pid %d %s: trap %d err %d on cpu %d "
            "eip 0x%x addr 0x%x--kill proc\n",
            myproc()->pid, myproc()->name, tf->trapno,
            tf->err, cpuid(), tf->eip, rcr2());

    //Inserindo a condicional para o caso de apresentar segment fault:
    //rcr2 retorna o ultimo endereco que se tentou acessar
    //tf->trapno retorna o erro ocasionado, no caso, erro 14
    if(rcr2() == 0 && tf->trapno == 14){
      cprintf("Acesso negado. Tentativa de acesso ao endereco 0!");
    }
    myproc()->killed = 1;
  }
```

4.2.2 Teste

Agora, quando executamos o comando **nullpointer**, o resultado é:

```
$ nullpointer
pid 3 nullpointer: trap 14 err 4 on cpu 1 eip 0x1001 addr 0x0--kill proc
Acesso negado. Tentativa de acesso ao endereco 0!$
```

4.3 Task 3 - Protection of read-only segments

A maioria dos sistemas operacionais trata segmentos de texto como *read-only* (apenas leitura). O arquivo **elf.h** especifica quais segmentos deveriam ser tratados como *read-only*. Contudo, o **Makefile** original faz o *link* do programa de tal modo que todos os segmentos possuem permissão *read-write-execute* (leitura, escrita e execução).

De modo a testar isso, criamos o arquivo **permissionexploit.c** que tenta escrever o valor hexadecimal B no segmento que deveria ser *read-only*. Esse segmento é o próprio ponteiro para a função **main()**. Como trata-se de uma função, ela é com certeza apenas *read-only* e a escrita nela não deveria ser permitida:

```
#include "types.h"
#include "stat.h"
#include "user.h"

int main(int argc, char *argv[]){
        int *ponteiro;
        ponteiro = (int*)main;
        ponteiro[0] = 0xB;
        printf(1,"%x\n",*ponteiro);
        exit();
}
```

Editamos o arquivo **Makefile** para reconhecer e executar o processo **permissionexploit**:

```
# Prevent deletion of intermediate files, e.g. cat.o, after first build, so
# that disk image changes after first build are persistent until clean.  More
# details:
# http://www.gnu.org/software/make/manual/html_node/Chained-Rules.html
.PRECIOUS: %.o

UPROGS=\
        _cat\
        _echo\
        _forktest\
        _grep\
        _init\
        _kill\
        _ln\
        _ls\
        _mkdir\
        _rm\
        _sh\
        _stressfs\
        _usertests\
        _wc\
        _zombie\
        _permissionexploit\
```

O resultado da execução do comando é:

```
$ permissionexploit
B
```

Portanto, há a escrita do valor B e a proteção *read-only* é ignorada.

4.3.1 Arquivos modificados

Observou-se que a *flag* **-N** do *linker* no arquivo **Makefile** é o responsável por ignorar as permissões definidas nos cabeçalhos de segmentos dos processos e por configurá-los como *read-write-execute* para todos os usuários.

```
_%: %.o $(ULIB)
	$(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^
	$(OBJDUMP) -S $@ > $*.asm
	$(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $*.sym

_forktest: forktest.o $(ULIB)
	# forktest has less library code linked in - needs to be small
	# in order to be able to max out the proc table.
	$(LD) $(LDFLAGS) -N -e main -Ttext 0 -o _forktest forktest.o ulib.o usys.o
	$(OBJDUMP) -S _forktest > forktest.asm
```

Se alterarmos para **-n**, o segmento de texto se torna *read-only*. Essa alteração pode ser vista no código abaixo:

```
_%: %.o $(ULIB)
	$(LD) $(LDFLAGS) -n -e main -Ttext 0 -o $@ $^
	$(OBJDUMP) -S $@ > $*.asm
	$(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $*.sym

_forktest: forktest.o $(ULIB)
	# forktest has less library code linked in - needs to be small
	# in order to be able to max out the proc table.
	$(LD) $(LDFLAGS) -n -e main -Ttext 0 -o _forktest forktest.o ulib.o usys.o
	$(OBJDUMP) -S _forktest > forktest.asm
```

Com a modificação no **Makefile**, todos os arquivos viraram *read-only*. Agora, precisamos restaurar aqueles que precisam ser, de fato, *read-write*. Podemos fazer isso modificando a função **loaduvm()** do arquivo **vm.c**. Antes da modificação, a função **loaduvm()** era:

```
int loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
  uint i, pa, n;
  pte_t *pte;

  if((uint) addr % PGSIZE != 0)
    panic("loaduvm: addr must be page aligned");
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
      panic("loaduvm: address should exist");
    pa = PTE_ADDR(*pte);
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    if(readi(ip, P2V(pa), offset+i, n) != n)
      return -1;

  }
  return 0;
}
```

Fizemos com que a função recebesse um novo argumento do tipo **uint**, chamada **permissao** e que representará as *flags* do cabeçalho de cada seção. Trabalhamos então com 3 valores binários: **permissao**, que recebe a *flag* do cabeçalho da seção, **PTE_W**, que indica a posição da *flag* correspondente ao *write*, e o ponteiro **pte**, que é o estado atual da *flag*.

Queremos que o estado do ponteiro **pte** vire 1 caso o *and* da **permissao** com o **PTE_W** dê 0, o que significará que a *flag* deveria estar ativada, mas está desativada. Assim, alteramos o valor do ponteiro **pte**, de modo a ativá-lo:

```
int loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz, uint permissao)
{
  uint i, pa, n;
  pte_t *pte;

  if((uint) addr % PGSIZE != 0)
    panic("loaduvm: addr must be page aligned");
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
      panic("loaduvm: address should exist");
    pa = PTE_ADDR(*pte);
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    if(readi(ip, P2V(pa), offset+i, n) != n)
      return -1;

    uint ativado = permissao & PTE_W; //faz um AND da flag recebida com o PTE_W (flag de write)

    cprintf("Flag de Permissao Antes = %d\n",permissao);
    cprintf("PTE_W = %d\n",PTE_W);
    cprintf("Ativado = %d\n",ativado);

    if(!ativado){ //se for 0, significa que a flag estah desativada mas deveria estar ativada
      *pte = *pte & ~PTE_W;
      cprintf("Flag de Permissao Depois = %d\n",*pte);
    }

  }
  return 0;
}
```

Como fizemos com que a função **loaduvm()** recebesse mais um parâmetro, precisamos incluir essa informação na definição da função. Originalmente, a função descrita no arquivo **defs.h** estava:

```
// vm.c
void            seginit(void);
void            kvmalloc(void);
pde_t*          setupkvm(void);
char*           uva2ka(pde_t*, char*);
int             allocuvm(pde_t*, uint, uint);
int             deallocuvm(pde_t*, uint, uint);
void            freevm(pde_t*);
void            inituvm(pde_t*, char*, uint);
int             loaduvm(pde_t*, char*, struct inode*, uint, uint);
pde_t*          copyuvm(pde_t*, uint);
void            switchuvm(struct proc*);
```

```
13 void            switchkvm(void);
14 int             copyout(pde_t*, uint, void*, uint);
15 void            clearpteu(pde_t *pgdir, char *uva);
```

Então incluímos mais uma entrada:

```
1  // vm.c
2  void            seginit(void);
3  void            kvmalloc(void);
4  pde_t*          setupkvm(void);
5  char*           uva2ka(pde_t*, char*);
6  int             allocuvm(pde_t*, uint, uint);
7  int             deallocuvm(pde_t*, uint, uint);
8  void            freevm(pde_t*);
9  void            inituvm(pde_t*, char*, uint);
10 int             loaduvm(pde_t*, char*, struct inode*, uint, uint, uint);
11 pde_t*          copyuvm(pde_t*, uint);
12 void            switchuvm(struct proc*);
13 void            switchkvm(void);
14 int             copyout(pde_t*, uint, void*, uint);
15 void            clearpteu(pde_t *pgdir, char *uva);
```

Por último, precisamos de fato enviar o valor da variável **permissao**. Isso é feito no arquivo **exec.c**, na função **exec()**. Passamos o argumento **ph.flags** que são justamente as permissões necessárias.

```
1   // Load program into memory.
2   sz = 0;
3   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
4     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
5       goto bad;
6     if(ph.type != ELF_PROG_LOAD)
7       continue;
8     if(ph.memsz < ph.filesz)
9       goto bad;
10    if(ph.vaddr + ph.memsz < ph.vaddr)
11      goto bad;
12    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
13      goto bad;
14    if(ph.vaddr % PGSIZE != 0)
15      goto bad;
16    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz, ph.flags) < 0)
17      goto bad;
18  }
19  iunlockput(ip);
20  end_op();
21  ip = 0;
```

4.3.2 Teste

Agora podemos tentar rodar novamente o comando **permissionexploit** e verificar se ainda é permitida a escrita. Como esperado, não foi permitida a escrita (vide Figura 4.3.2).

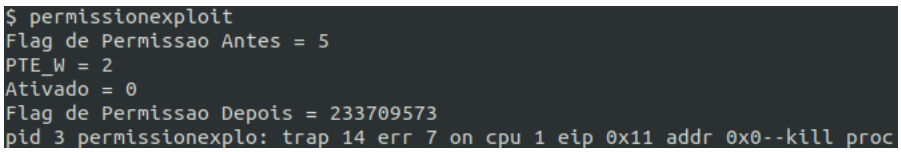


Figura 4: Resultado da execução do comando **permissionexploit** após as modificações

4.4 Task 4 - Copy-on-write (COW)

Quando executamos um comando *fork*, o proceso é duplicado junto com sua memória. No xv6, quando executamos um *fork*, todas as *memory pages* sáo copiadas ao processo filho, o que requer muito processamento e se torna um processo demorado. Em muitos casos, não é necessária essa cópia, pois *memory pages* serão inutilizados ou ficarão redundantes. É então socilitada a implementação do *copy-on-write* (COW), comum em muitos sistemas operacionais modernos. Nele, uma *memory page* é copiada apenas quando precisa ser modificada.

4.4.1 Discussão prévia

A implementação do *Copy-on-write* exige que os *forks* não criem cópias da memória virtual, pois muitas vezes elas não são modificadas durante o tempo de vida do *fork*. No COW, as páginas ocupadas pelo processo pai recebem um novo bit SH (”Shared”) que dita se esta página é compartilhada por um *fork* filho ou não. Além disso, o bit W (”Write”) é zerado. As páginas se tornam *read-only*. É possível que vários *forks* ocorram dentro de um único processo, fazendo com que o mesmo compartilhe suas páginas com vários *forks* filho. Para isso, devemos adicionar contadores de *forks* que estão presentes na mesma paginação do processo pai na memória física. A cada pedido de escrita, o contador deve ser reduzido, enquanto a cada pedido de *fork*, o contador deve aumentar - podemos fazer isso através de uma *Shared Table*. Assim que um pedido de escrita venha para uma página no qual o bit W é zero, e o bit SH é 1, o sistema operacional deve fazer a cópia das páginas ocupadas pelo processo pai, apontar o *fork* requisitante à nova região da memória virtual e realizar o pedido de escrita. Caso um pedido de escrita for feito e o contador de *forks* na memória compartilhada for 1, o sistema deve restaurar a permissão de escrita de tanto as páginas originais quanto das cópias e, só após este processo, realizar o pedido de escrita.

4.4.2 Criação de uma nova *system call*

O primeiro passo para a implementação do COW é criação de uma nova *system call*, que chamaremos de **cowfork**. Para tal, devemos seguir alguns passos. O primeiro deles é a adicionar uma entrada para a nova *syscall* no arquivo **syscall.h**. Ela é adicionada na linha 23 do código a seguir.

```
1  // System call numbers
2  #define SYS_fork    1
3  #define SYS_exit    2
4  #define SYS_wait    3
5  #define SYS_pipe    4
6  #define SYS_read    5
7  #define SYS_kill    6
8  #define SYS_exec    7
9  #define SYS_fstat   8
10 #define SYS_chdir   9
11 #define SYS_dup    10
12 #define SYS_getpid 11
13 #define SYS_sbrk   12
14 #define SYS_sleep  13
```

```
15 #define SYS_uptime 14
16 #define SYS_open   15
17 #define SYS_write  16
18 #define SYS_mknod  17
19 #define SYS_unlink 18
20 #define SYS_link   19
21 #define SYS_mkdir  20
22 #define SYS_close  21
23 #define SYS_cowfork 22
```

Precisamos então incluir nossa **cowfork** definida em **syscall.c**. Seguimos os padrões já definidos nesse arquivo, como pode ser visto no código a seguir (incluídas as linhas 22 e 25):

```
1  extern int sys_chdir(void);
2  extern int sys_close(void);
3  extern int sys_dup(void);
4  extern int sys_exec(void);
5  extern int sys_exit(void);
6  extern int sys_fork(void);
7  extern int sys_fstat(void);
8  extern int sys_getpid(void);
9  extern int sys_kill(void);
10 extern int sys_link(void);
11 extern int sys_mkdir(void);
12 extern int sys_mknod(void);
13 extern int sys_open(void);
14 extern int sys_pipe(void);
15 extern int sys_read(void);
16 extern int sys_sbrk(void);
17 extern int sys_sleep(void);
18 extern int sys_unlink(void);
19 extern int sys_wait(void);
20 extern int sys_write(void);
21 extern int sys_uptime(void);
22 extern int sys_cowfork(void);
23
24 static int (*syscalls[])(void) = {
25 [SYS_cowfork] sys_cowfork,
26 [SYS_fork]    sys_fork,
27 [SYS_exit]    sys_exit,
28 [SYS_wait]    sys_wait,
29 [SYS_pipe]    sys_pipe,
30 [SYS_read]    sys_read,
31 [SYS_kill]    sys_kill,
32 [SYS_exec]    sys_exec,
33 [SYS_fstat]   sys_fstat,
34 [SYS_chdir]   sys_chdir,
35 [SYS_dup]     sys_dup,
36 [SYS_getpid]  sys_getpid,
37 [SYS_sbrk]    sys_sbrk,
38 [SYS_sleep]   sys_sleep,
39 [SYS_uptime]  sys_uptime,
40 [SYS_open]    sys_open,
41 [SYS_write]   sys_write,
42 [SYS_mknod]   sys_mknod,
43 [SYS_unlink]  sys_unlink,
44 [SYS_link]    sys_link,
45 [SYS_mkdir]   sys_mkdir,
46 [SYS_close]   sys_close,
```

```
47 };
```

Por fim, também incluímos a nova *system call* no arquivo **usys.S** (linha 12):

```
1  #include "syscall.h"
2  #include "traps.h"
3
4  #define SYSCALL(name) \
5    .globl name; \
6    name: \
7      movl $SYS_ ## name, %eax; \
8      int $T_SYSCALL; \
9      ret
10
11 SYSCALL(fork)
12 SYSCALL(cowfork)
13 SYSCALL(exit)
14 SYSCALL(wait)
15 SYSCALL(pipe)
16 SYSCALL(read)
17 SYSCALL(write)
18 SYSCALL(close)
19 SYSCALL(kill)
20 SYSCALL(exec)
21 SYSCALL(open)
22 SYSCALL(mknod)
23 SYSCALL(unlink)
24 SYSCALL(fstat)
25 SYSCALL(link)
26 SYSCALL(mkdir)
27 SYSCALL(chdir)
28 SYSCALL(dup)
29 SYSCALL(getpid)
30 SYSCALL(sbrk)
31 SYSCALL(sleep)
32 SYSCALL(uptime)
```

4.4.3 Implementação

Feitas essas definições, precisamos criar a função que trata do **cowfork** propriamente dito. Isso será feito dentro do arquivo **proc.c**. Como havíamos dito já no *task 1*, esse arquivo contém as funções referentes aos procedimentos. Para sua implementação, nos baseamos na função já definida do *fork*:

```
1  // Create a new process copying p as the parent.
2  // Sets up stack to return as if from system call.
3  // Caller must set state of returned proc to RUNNABLE.
4  int
5  fork(void)
6  {
7    int i, pid;
8    struct proc *np;
9    struct proc *curproc = myproc();
10
11   // Allocate process.
12   if((np = allocproc()) == 0){
13     return -1;
14   }
```

```
15
16    // Copy process state from proc.
17    if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
18      kfree(np->kstack);
19      np->kstack = 0;
20      np->state = UNUSED;
21      return -1;
22    }
23    np->sz = curproc->sz;
24    np->parent = curproc;
25    *np->tf = *curproc->tf;
26
27    // Clear %eax so that fork returns 0 in the child.
28    np->tf->eax = 0;
29
30    for(i = 0; i < NOFILE; i++)
31      if(curproc->ofile[i])
32        np->ofile[i] = filedup(curproc->ofile[i]);
33    np->cwd = idup(curproc->cwd);
34
35    safestrcpy(np->name, curproc->name, sizeof(curproc->name));
36
37    pid = np->pid;
38
39    acquire(&ptable.lock);
40
41    np->state = RUNNABLE;
42
43    release(&ptable.lock);
44
45    return pid;
46  }
```

Em relação à função **fork**, são feitas duas pequenas modificações: A primeira delas é, ao invés de usarmos a função **copyuvm()** que indica a cópia da memória do pai para o filho, usamos uma função chamada **share_cowfork()**, a qual iremos definir mais para frente. Além disso, ativamos as *flags* "*shared*"do processo pai e do processo filho.

```
1  int cowfork(void){
2    int i, pid;
3    struct proc *np;
4    struct proc* curproc = myproc();
5
6    // Allocate process.
7    if((np = allocproc()) == 0){
8      return -1;
9    }
10
11   //Aqui usamos a funcao share_cowfork ao inves da copyuvm
12   if((np->pgdir = share_cowfork(curproc->pgdir, curproc->sz)) == 0){
13     kfree(np->kstack);
14     np->kstack = 0;
15     np->state = UNUSED;
16     return -1;
17   }
18   np->sz = curproc->sz;
19   np->parent = curproc;
20   *np->tf = *curproc->tf;
```

```
21
22    // Clear %eax so that fork returns 0 in the child.
23    np->tf->eax = 0;
24
25    // Ativamos as flags que indicam que ha compartilhamento de memoria
26    np->shared = 1;
27    curproc->shared = 1;
28
29    for(i = 0; i < NOFILE; i++)
30      if(curproc->ofile[i])
31        np->ofile[i] = filedup(curproc->ofile[i]);
32    np->cwd = idup(curproc->cwd);
33
34    safestrcpy(np->name, curproc->name, sizeof(curproc->name));
35
36    pid = np->pid;
37
38    acquire(&ptable.lock);
39
40    np->state = RUNNABLE;
41
42    release(&ptable.lock);
43
44    return pid;
45  }
```

Essa *flag* "*shared*"foi criada no arquivo **proc.h**, dentro da estrutura do **proc**:

```
1  // Per-process state
2  struct proc {
3    uint sz;                     // Size of process memory (bytes)
4    pde_t* pgdir;                // Page table
5    char *kstack;                // Bottom of kernel stack for this process
6    enum procstate state;        // Process state
7    int pid;                     // Process ID
8    struct proc *parent;         // Parent process
9    struct trapframe *tf;        // Trap frame for current syscall
10   struct context *context;     // swtch() here to run process
11   void *chan;                  // If non-zero, sleeping on chan
12   int killed;                  // If non-zero, have been killed
13   struct file *ofile[NOFILE];  // Open files
14   struct inode *cwd;           // Current directory
15   char name[16];               // Process name (debugging)
16   int shared;
17 };
```

E também como flag de uma *page table entry* no **mmu.h**:

```
1  // Page table/directory entry flags.
2  #define PTE_P           0x001   // Present
3  #define PTE_W           0x002   // Writeable
4  #define PTE_U           0x004   // User
5  #define PTE_PWT         0x008   // Write-Through
6  #define PTE_PCD         0x010   // Cache-Disable
7  #define PTE_A           0x020   // Accessed
8  #define PTE_D           0x040   // Dirty
9  #define PTE_PS          0x080   // Page Size
10 #define PTE_MBZ         0x180   // Bits must be zero
11 #define PTE_SHARE       0x100
```

A ideia da função **share_cowfork()** é semelhante à **copyuvm()** já implementada no arquivo **vm.c**. No entanto, não faz a cópia de fato e sim lida com a chamada *Shared Table*, exposta na Seção 4.4.1. A seguir apresentamos o que seriam as funções do *Shared Table* e o **share_cowfork()**. É importante ressaltar que o *Shared Table* depende de um *mutex* para garantir que não haja concorrência de alteração. Na linha 70 do código, usamos a função **lcr3()** de modo a resolver o problema exposto na própria definição da *task*: "*When a page fault occurs, the faulty address is written to the cr2 register. After copying a page, the virtual to physical mapping for that page has changed, therefore, be sure to refresh the TLB*". Perceba que na linha 58 caso o contador de determinado endereço da *Shared Table* já seja maior que 0, há um único incremento. Se for igual a 0, há 2 incrementos: um corresponde ao processo pai, e outro ao processo filho criado.

```
1  #define MAXSHAREDTABLE PHYSTOP >> 12 // Maximo de memoria enderecavel shared table
2  static int sharedTable[MAXSHAREDTABLE]; // Shared table => contadores
3  struct spinlock mutex;// Bloqueia a shared table para que nao haja concorrencia
4
5  // cria e inicializa a shared table com 0
6  void init_sharedtable(void)
7  {
8    initlock(&mutex, "sharedtable");
9    acquire(&mutex);
10   int i;
11   for(i=0; i< MAXSHAREDTABLE; i++){
12     sharedTable[i]=0;
13   }
14   release(&mutex);
15 }
16
17 int countAddress(uint addr){
18   return sharedTable[((addr >> 12) & 0xFFFFF)];
19 }
20
21 void incCountAddress(uint addr){
22   sharedTable[((addr >> 12) & 0xFFFFF)]++;
23 }
24
25 void decCountAddress(uint addr){
26   sharedTable[((addr >> 12) & 0xFFFFF)]--;
27 }
28
29 pde_t* share_cowfork(pde_t *pgdir, uint sz)
30 {
31   pde_t *d;
32   pte_t *pte;
33   uint pa, i, flags;
34   struct proc *curproc = myproc();
35
36   if((d = setupkvm()) == 0)
37   return 0;
38
39   acquire(&mutex);
40
41   for(i = PGSIZE; i < sz; i += PGSIZE){
42     if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
43     panic("shared_cowfork: pte should exist");
44     if(!(*pte & PTE_P))
```

```
45      panic("shared_cowfork: page not present");
46
47      //seta 1 no shared e 0 no write
48      *pte |= PTE_SHARE;
49      *pte &= ~PTE_W;
50
51      pa = PTE_ADDR(*pte);
52      flags = PTE_FLAGS(*pte);
53
54      if(mappages(d, (void*)i, PGSIZE, pa, flags) < 0)
55      goto bad;
56
57      //ajusta sharedtable
58      if(countAddress(pa) != 0){
59        incCountAddress(pa);
60      }
61      else{
62        incCountAddress(pa);
63        incCountAddress(pa);
64      }
65    }
66
67    release(&mutex);
68    lcr3(V2P(curproc->pgdir));
69    return d;
70
71    bad:
72    freevm(d);
73    return 0;
74  }
```

Além disso, criamos mais quatro funções, **deallocuvm_cowfork()**, **freevm_cowfork()**, **trap_pagefault()** e **copyuvm_cowfork()**, também baseadas nas já implementadas **deallocuvm()** e **freevm()**. A **deallocuvm_cowfork()** desaloca a memória virtual fazendo as alterações necessárias na *Shared Table*. Já a **freevm_cowfork()**, libera a memória virtual de um processo que utilizou o **share_cowfork()**. As duas últimas funções, **trap_pagefault()** e **copyuvm_cowfork()**, se fazem necessárias para tratar casos onde ocorrem *Page Faults* e precisamos copiar a memória.

```
1  int deallocuvm_cowfork(pde_t *pgdir, uint oldsz, uint newsz)
2  {
3    pte_t *pte;
4    uint a, pa;
5
6    if(newsz >= oldsz)
7      return oldsz;
8
9    a = PGROUNDUP(newsz);
10
11   acquire(&mutex);
12
13   for(; a < oldsz; a += PGSIZE){
14     pte = walkpgdir(pgdir, (char*)a, 0);
15     if(!pte)
16     a += (NPTENTRIES - 1) * PGSIZE;
17     else if((*pte & PTE_P) != 0){
18       pa = PTE_ADDR(*pte);
19       if(pa == 0)
```

```
20        panic("kfree");
21
22        // Se a memoria esta sendo compartilhada, decrementa
23        if (countAddress(pa) > 1) {
24          decCountAddress(pa);
25        }
26        // Senao, eh liberada completamente
27        else {
28          char *v = P2V(pa);
29          kfree(v);
30          decCountAddress(pa);
31        }
32        *pte = 0;
33      }
34    }
35    release(&mutex);
36    return newsz;
37  }
38
39  void freevm_cowfork(pde_t *pgdir)
40  {
41    uint i;
42
43    if(pgdir == 0)
44    panic("freevm_cowfork: no pgdir");
45
46    deallocuvm_cowfork(pgdir, KERNBASE, 0);
47
48    for(i = 0; i < NPDENTRIES; i++){
49      if(pgdir[i] & PTE_P){
50        char *v = P2V(PTE_ADDR(pgdir[i]));
51        kfree(v);
52      }
53    }
54    kfree((char*)pgdir);
55  }
56
57
58  void trap_pagefault(struct proc* curproc){
59    uint addr = rcr2(); // Recebe o valor da pagina que deu pagefault
60    //struct proc *curproc = myproc();
61
62    if (addr == 0) {
63      cprintf("Segmentation Fault - Null Pointer Dereference\n");
64      kill(curproc->pid);
65    }
66    // Se o processo possui paginas compartilhadas, copia memoria
67    else{
68      pte_t* pte = walkpgdir(curproc->pgdir, (void *) addr, 0);
69
70      if(PTE_FLAGS(*pte)&PTE_SHARE){
71          copyuvm_cowfork(addr, curproc);
72          cprintf("Page Fault: cowfork \n");
73      }
74      else{
75        cprintf("Segmentation Fault - Writing to Read-only Memory\n");
76        kill(curproc->pid);
77      }
78    }
```

```
79 }
80
81 int copyuvm_cowfork(uint addr, struct proc* curproc)
82 {
83   uint pa;
84   pte_t *pte;
85   char *mem;
86   //struct proc *curproc = myproc();
87
88   pte = walkpgdir(curproc->pgdir, (void *) addr, 0);
89   pa = PTE_ADDR(*pte);
90
91   acquire(&mutex);
92   // Se pagina esta sendo compartilhada
93   if (countAddress(pa) > 1) {
94     if((mem = kalloc()) == 0) // aloca uma nova pagina de memoria
95       goto bad;
96     memmove(mem, (char*)P2V(pa), PGSIZE);
97     *pte &= 0xFFF;
98     *pte &= ~PTE_SHARE; // flag share = 0
99     *pte |= V2P(mem) | PTE_W; // permite a escrita
100    decCountAddress(pa);
101  }
102  //Senao
103  else {
104    *pte |= PTE_W;
105    *pte &= ~PTE_SHARE;
106  }
107
108  release(&mutex);
109
110  lcr3(V2P(curproc->pgdir));
111
112  return 1;
113
114 bad:
115   return 0;
116 }
```

Há mais uma alteração no arquivo **proc.c** que precisamos fazer: Na função **wait()**, devemos tratar corretamente os casos de liberação de memória, isto é, se o processo não possui paginas compartilhadas, pode-se liberar a memória; senão, é necessário liberar apenas as páginas que não estão sendo compartilhadas, com a função **freevm_cowfork()** criada. Para verificarmos isso, avaliamos a *flag* "*shared*".

```
1  int
2  wait(void) {
3    struct proc *p;
4    int havekids, pid;
5    struct proc *curproc = myproc();
6
7    acquire(&ptable.lock);
8    for(;;){
9      // Scan through table looking for exited children.
10     havekids = 0;
11     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
12       if(p->parent != curproc)
```

```
13          continue;
14        havekids = 1;
15        if(p->state == ZOMBIE){
16          // Found one.
17          pid = p->pid;
18          kfree(p->kstack);
19          p->kstack = 0;
20
21          //cow
22          if (p->shared == 0) {
23            freevm(p->pgdir);
24          }
25          else {
26            freevm_cowfork(p->pgdir);
27            p->shared = 0;
28          }
29
30          freevm(p->pgdir);
31          p->pid = 0;
32          p->parent = 0;
33          p->name[0] = 0;
34          p->killed = 0;
35          p->state = UNUSED;
36          release(&ptable.lock);
37          return pid;
38        }
39      }
40
41      // No point waiting if we don't have any children.
42      if(!havekids || curproc->killed){
43        release(&ptable.lock);
44        return -1;
45      }
46
47      // Wait for children to exit.  (See wakeup1 call in proc_exit.)
48      sleep(curproc, &ptable.lock);  //DOC: wait-sleep
49    }
50  }
```

Agora faz-se necessária a adição do cabeçalho dessas funções criadas em **defs.h**:

```
1  // cowfork
2  int             cowfork(void);
3  pde_t*          share_cowfork(pde_t*, uint);
4  void            init_sharedtable(void);
5  int             countAddress(uint);
6  void            incCountAddress(uint);
7  void            decCountAddress(uint);
8  void            freevm_cowfork(pde_t*);
9  int             deallocuvm_cowfork(pde_t*, uint, uint);
10 int             copyuvm_cowfork(uint, struct proc*);
11 void            trap_pagefault(struct proc*);
```

Precisamos fazer com que o arquivo **trap.c** chame a função **trap_pagefault()** sempre que ocorrer um *Page Fault*. Isso é feito dentro da função **trap()** no arquivo **trap.c**:

```
1   if (tf->trapno == T_PGFLT) {
2     myproc()->tf = tf;
3     struct proc *curproc = myproc();
```

```
4       trap_pagefault(curproc);
5       return;
6     }
```

No arquivo **main.c**, adicionamos a função que inicializa a *Shared Table* na função principal (linha 21):

```
1  int main(void)
2  {
3    kinit1(end, P2V(4*1024*1024)); // phys page allocator
4    kvmalloc();      // kernel page table
5    mpinit();        // detect other processors
6    lapicinit();     // interrupt controller
7    seginit();       // segment descriptors
8    picinit();       // disable pic
9    ioapicinit();    // another interrupt controller
10   consoleinit();   // console hardware
11   uartinit();      // serial port
12   pinit();         // process table
13   tvinit();        // trap vectors
14   binit();         // buffer cache
15   fileinit();      // file table
16   ideinit();       // disk
17   startothers();   // start other processors
18   kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
19   userinit();      // first user process
20   mpmain();        // finish this processor's setup
21   init_sharedtable();
22 }
```

4.4.4 Implementando a *system call*

Devemos então fazer modificações de tal forma que a **cowfork** funcione de fato como uma *system call* e pode ser inclusive chamada pelos programadores no xv6. O primeiro passo é modificar o arquivo **sysproc.h** e incluir a função **sys_cowfork()** que faz uma chamada para a função que criamos na seção anterior, a **cowfork()**:

```
1  int sys_cowfork(void)
2  {
3    return cowfork();
4  }
```

O passo seguinte é incluir a *system call* no arquivo **user.h**:

```
1  // system calls
2  int fork(void);
3  int exit(void) __attribute__((noreturn));
4  int wait(void);
5  int pipe(int*);
6  int write(int, void*, int);
7  int read(int, void*, int);
8  int close(int);
9  int kill(int);
10 int exec(char*, char**);
11 int open(char*, int);
12 int mknod(char*, short, short);
```

```
13 int unlink(char*);
14 int fstat(int fd, struct stat*);
15 int link(char*, char*);
16 int mkdir(char*);
17 int chdir(char*);
18 int dup(int);
19 int getpid(void);
20 char* sbrk(int);
21 int sleep(int);
22 int uptime(void);
23 int cowfork(void);
```

4.4.5 Teste

Para testarmos nossa implementação do *task 4*, criamos o arquivo **runcowfork**. O que ele faz é simplesmente chamar a função **cowfork()**:

```
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4
5  int main(int argc, char const *argv[]) {
6    int pid = cowfork();
7    if(pid==0){
8      printf(1,"Processo filho rodando\n");
9    }
10   else{
11     printf(1,"Processo pai rodando\n");
12   }
13   while (1);
14   exit();
15 }
```

Por último, fizemos as inclusões necessárias no **Makefile**:

```
1  UPROGS=\
2          _cat\
3          _echo\
4          _forktest\
5          _grep\
6          _init\
7          _kill\
8          _ln\
9          _ls\
10         _mkdir\
11         _rm\
12         _sh\
13         _stressfs\
14         _usertests\
15         _wc\
16         _zombie\
17         _runcowfork\
18
19 EXTRA=\
20         mkfs.c ulib.c user.h cat.c echo.c forktest.c runcowfork.c grep.c kill.c\
21         ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
22         printf.c umalloc.c\
```

```
23          README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
24          .gdbinit.tmpl gdbutil\
```

Recebemos um resultado indesejado:

```
$ runcowfork
Page Fault: cowfork
PID = Segmentation Fault - Writing to Read-only Memory
Segmentation Fault - Writing to Read-only Memory
Segmentation Fault - Writing to Read-only Memory
Segmentation Fault - Writing to Read-only Memory
Segmentation Fault - Writing to Read-only Memory
Segmentation Fault - Writing to Read-only Memory
4
Processo pai rodando
```

Figura 5: Resultado da execução do comando **runcowfork** com erros

Por algum motivo que não conseguimos descobrir, a nossa função **trap_pagefault()** não está funcionando da maneira adequada e não está passando na verificação da *flag* **PTE_SHARE** para o processo filho. Fizemos diversos testes e tentamos diversas soluções, mas nenhuma funcionou adequadamente. Na Figura 4.4.5 é possível ver como seria o resultado esperado, caso a função **trap_pagefault()** estivesse correta. Fizemos isso adaptando-a:

```
1  void trap_pagefault(struct proc* curproc){
2    uint addr = rcr2(); // Recebe o valor da pagina que deu pagefault
3    //struct proc *curproc = myproc();
4
5    if (addr == 0) {
6      cprintf("Segmentation Fault - Null Pointer Dereference\n");
7      kill(curproc->pid);
8    }
9    // Se o processo possui paginas compartilhadas, copia memoria
10   else{
11     pte_t* pte = walkpgdir(curproc->pgdir, (void *) addr, 0);
12
13     if(PTE_FLAGS(*pte)&PTE_SHARE){
14         copyuvm_cowfork(addr, curproc);
15         cprintf("Page Fault: cowfork \n");
16     }
17     else{
18       copyuvm_cowfork(addr, curproc);
19       cprintf("Page Fault: cowfork \n");
20       cprintf("PID = 0\n");
21       cprintf("Processo filho rodando\n");
22       kill(curproc->pid);
23     }
24   }
25 }
```

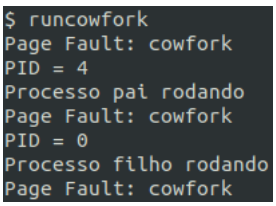


Figura 6: Resultado da execução do comando **runcowfork** com sucesso

5 Lista de Arquivos e Funções Citadas

5.1 Utilizados

- **CONSOLE.C**
  - **CONSOLEINTR**() = *task1*
- **DEFS.H** = *task3*, *task4*
- **ELF.H** = *task3*
- **EXEC.C**
  - **EXEC**() = *task2*, *task3*
- **MAIN.C**
  - **MAIN**() = *task4*
- **MAKEFILE** = *task2*, *task3*, *task4*
- **MEMORYLAYOUT.H**
  - **V2P**() = *task1*
  - **P2V**() = *task1*
- **MMU.H**
  - **NPDENTRIES** = *task1*
  - **NPTENTRIES** = *task1*
  - **PGSIZE** = *task2*
- **PROC.C**
  - **PROCDUMP**() = *task1*
  - **WAIT**() = *task4*
- **PROC.H**
  - **PGDIR** = *task1*
  - **PROC**() = *task4*
- **SPINLOCK.H** = *task4*
- **SYSPROC.C** = *task4*

- **SYSTEMCALL.C** = *task4*
- **SYSTEMCALL.H** = *task4*
- **TRAP.C**
  - **TRAP()** = *task2*, *task4*
  - **TF** = *task2*
- **USER.H** = *task4*
- **USYS.S** = *task4*
- **VM.C**
  - **COPYUVM()** = *task2*, *task4*
  - **DEALLOCUVM()** = *task4*
  - **FREEVM()** = *task4*
  - **LOADUVM()** = *task3*
- **X86.H**
  - **RCR2()** = *task2*
  - **LCR3()** = *task4*

5.2 Criados

- **NULLPOINTER.C** = *task2*
- **PROC.C**
  - **COWFORK()** = *task4*
- **PERMISSIONEXPLOIT.C** = *task3*
- **RUNCOWFORK.C** = *task4*
- **MMU.H**
  - **PTE_SHARE** = *task4*
- **VM.C**
  - **INIT_SHAREDTABLE()** = *task4*
  - **SHARE_COWFORK()** = *task4*

- **DEALLOCUVM_COWFORK()** = *task4*
- **FREEVM_COWFORK()** = *task4*
- **COUNTADDRESS()** = *task4*
- **DECCOUNTADDRESS()** = *task4*
- **INCCOUNTADDRESS()** = *task4*
- **TRAP_PAGEFAULT()** = *task4*
- **COPYUVM_COWFORK()** = *task4*