Universidade Federal de São Carlos
Departamento de Computação

Relatório - Projeto de Implementação 2
Grupo 2

Lucas Vinícius Domingues - 769699
Eduardo dos Santos Gualberto - 769726
Rafael Yoshio Yamawaki Murata - 769681
Victor Luís Aguilar Antunes - 769734

1. Proposta de Implementação

Este projeto consiste na implementação de tarefas no sistema operacional xv6 com o objetivo de entender como funciona o gerenciamento de memória e paginação de sistemas operacionais.

2. Tasks

A seguir, serão explicadas as resoluções elaboradas para as tarefas propostas do projeto de implementação 2.

2.1 Task 1: Enhancing process details viewer

A tarefa consiste na modificação da função executada quando se é pressionado ctrl + p na execução de xv6, de maneira que sejam exibidas mais informações de memória sobre o processo sendo executado.

Inicialmente, as únicas informações exibidas são: O Estado do processo e os endereços de pc para onde a função retorna.

Após a modificação da função *procdump* (em proc.c), além das informações iniciais, são exibidos: a localização na memória do diretório de páginas, a entrada na tabela de páginas e o número da página, a localização na memória da tabela de páginas e as entradas na tabela de páginas de outras páginas do processo

2.1.1 Implementação

Antes:

```
procdump(void)
{
  static char *states[] = {
  [UNUSED]    "unused",
  [EMBRYO]    "embryo",
  [SLEEPING]  "sleep ",
  [RUNNABLE]  "runble",
  [RUNNING]   "run   ",
  [ZOMBIE]    "zombie"
  };
  int i;
  struct proc *p;
  char *state;
  uint pc[10];

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == UNUSED)
      continue;
    if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
      state = states[p->state];
    else
      state = "???";
    cprintf("%d %s %s", p->pid, state, p->name);
    if(p->state == SLEEPING){
      getcallerpcs((uint*)p->context->ebp+2, pc);
      for(i=0; i<10 && pc[i] != 0; i++)
        cprintf(" %p", pc[i]);
    }
    cprintf("\n");
  }
}
```

Modificado:

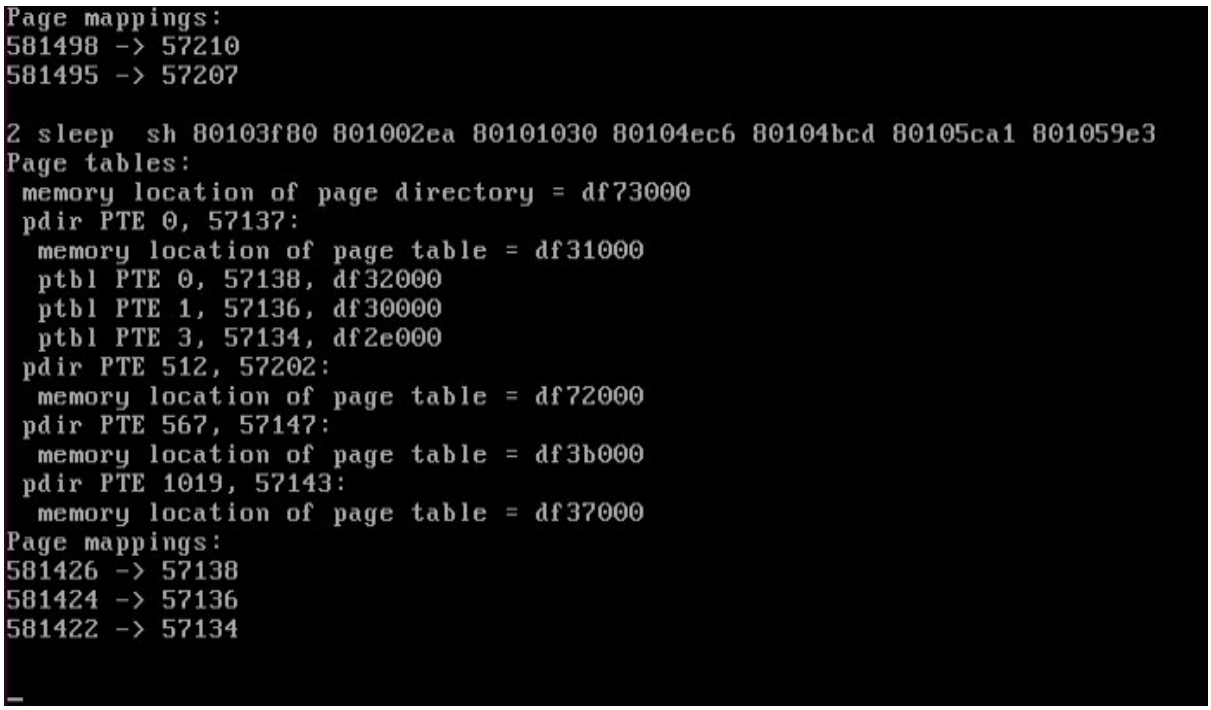
```
  uint tabela_pag[20];  //vetor q ira armazenar os endereços
  int cont = 0; //contador da tabela de paginas

  cprintf("\nPage tables:\n memory location of page directory = %x\n", V2P(p->pgdir));  //imprime endereço fisico
  for (int i = 0; i < NPDENTRIES; i++)  //ira percorrer todas as entradas de pag por diretorio
  {
    if ((PTE_FLAGS(p->pgdir[i]) & 0x020) && (PTE_FLAGS(p->pgdir[i]) & PTE_U))   //se pag for acessada (32 - dec / 0x020 - hex) e for de usuario
    {
      pte_t* ptable = (pte_t*) PTE_ADDR(p->pgdir[i]);
      cprintf(" pdir PTE %d, %d:\n  memory location of page table = %x\n", i, (uint)ptable >> 12, ptable);
      for (int j = 0; j < NPTENTRIES; j++)  //percorre tabela de paginas
      {
        pte_t* endereco = (pte_t*)((pte_t*)P2V(ptable))[j];   //endereco recebe endereço virtual
        uint flags = PTE_FLAGS(endereco);
        if ((flags & 0x020) && (flags & PTE_U))
        {
          cprintf("  ptbl PTE %d, %d, %x\n", j, (uint)endereco >> 12, PTE_ADDR(endereco));
          tabela_pag[cont++] = (uint) V2P(endereco);
        }
      }
    }
  }
  cprintf("Page mappings:\n");
  for(int i = 0; i < cont; i++)
    cprintf("%d -> %d\n",tabela_pag[i] >> 12,(V2P(tabela_pag[i])) >> 12); //imprime endereços armazenados
  cprintf("\n");
 }
}
```

2.1.2 Resultados

Antes:

```
$ 1 sleep  init 80103fd7 8010407f 80104a8d 80105ba1 801058ec
2 sleep  sh 80103fa0 801002ea 80101040 80104d86 80104a8d 80105ba1 801058ec
```

Modificado:



2.2 Task2: Null Pointer Protection

- A tarefa consiste em criar uma proteção que impeça o acesso ao endereço 0, que contém o segmento de texto do processo.

2.2.1 Implementação

- No makefile as mudanças foram para retirar a porção de texto do processo da posição 0x0. Pois dessa forma não ocorre o problema em buscar o valor do ponteiro nulo

- Em outros arquivos foram feitos ajustes para poder "começar a contar" o processo depois da primeira página de tabela e para que as variáveis não fiquem para trás do endereço 4096. (exec.c linha 42 / vm.c linha 329)

- Devido ao número considerável de modificações nos arquivos originais do xv6, será mostrado aqui apenas o

código de teste. Todos os códigos e modificações estão disponíveis no código enviado .

```c
#include "syscall.h"
#include "types.h"
#include "user.h"

#define NULL 0x0

int main()
{
    printf(1, "Rodando programa de teste para ponteiro nulo...\n");
    int *p = NULL;
    printf(1, "*p: %d \n",*p);
    exit();
}
```

2.2.2 Resultado do teste de proteção

```
$ null_test
Rodando programa de teste para ponteiro nulo...
pid 4 null_test: trap 14 err 4 on cpu 0 eip 0x1021 addr 0x0--kill proc
```

2.3 Task 3: Protection of Read-Only segments

- A tarefa a ser implementada é proteger segmentos do programa que deveriam apenas ter permissão para leitura, visto que o Makefile original do xv6 faz com que todos os

segmentos sejam executados com permissão de escrita e leitura.

2.3.1 Implementação

- Na tentativa de retirar a flag -N em certas seções do Makefile, ocorreram erros na compilação qemu, forçando o grupo a procurar por outras alternativas para a realização da task;

- Dessa forma, foram implementadas system calls para “proteger” e “desproteger” segmentos do processo.

- Na função de proteção (em vm.c linha 400), ao verificar que a PTE encontra-se numa área de usuário e não pode realizar escrita, lhe é negada a flag de escrita permitindo então que a função seja executada somente no modo de leitura.

- Já na função de retirar a proteção (em vm.c linha 423), verifica-se se de acordo com a PTE, deve ser permitido escrever no segmento do processo, ativando a flag de escrita em caso positivo.

```
int
proteger(void *addr, int tam){
  struct proc *curproc = myproc();
  pte_t *pte;
  int endereco = (int)addr;

  for (int i = endereco; i < (endereco + tam*PGSIZE); i += PGSIZE){
    pte = walkpgdir(curproc->pgdir,(void*) i, 0);
    if(pte && ((*pte & PTE_U)) && ((*pte & PTE_P)))     //PTE deve ser de
      *pte = (*pte) & (~PTE_W) ;                         //Muda flag para
    else
      return -1;
  }
  //Carrega reg3 com endereco da pag de diretorio
  lcr3(V2P(curproc->pgdir));
  return 0;
}
```

```
int
desproteger(void *addr, int tam){
  struct proc *curproc = myproc();
  pte_t *pte;
  int endereco = (int)addr;

  for (int i = endereco; i < (endereco + tam*PGSIZE); i += PGSIZE){
    pte = walkpgdir(curproc->pgdir,(void*) i, 0);
    if(pte && ((*pte & PTE_U)) && ((*pte & PTE_P)))   //PTE deve ser de usu
      *pte = (*pte) | (PTE_W);                         //Coloca flag para
    else
      return -1;
  }
  //Carrega reg3 com endereco da pag de diretorio
  lcr3(V2P(curproc->pgdir));
  return 0;
}
```

2.3.2 Teste da implementação

```
$ sanitytest
Valor de *p comeca com = 0


Protegendo para escrita...


Lendo. *p = 0


Desprotegendo para escrita...


Tentando escrever em main (eh para funcionar)...Lendo. *p = 1


Protegendo para escrita...


Tentando escrever em main (eh para dar erro)...pid 4 sanitytest: trap 14 err 7 o
n cpu 0 eip 0x10e2 addr 0x1000--kill proc
```

2.4 Task 4: Copy-on-Write

- A tarefa consiste na construção da funcionalidade de copy-on-write no sistema xv6. Nessa funcionalidade, uma página de memória é copiada somente quando tiver que ser alterada durante a realização de um fork. Dessa forma, ao invés

de copiar todas as páginas da memória do processo pai para o processo filho, fazemos com que ambos os processos compartilhem as mesmas páginas. Assim, evita-se acessos de memória desnecessários

2.4.1 Implementação

- Para a realização da tarefa foi necessário:

(A) Adicionar uma nova syscall chamada ***cowfork***

(B) Modificar ***kalloc.c*** para conseguirmos contar quantos processos estão compartilhando determinada página (reference Count)

(C) Criar um função para lidar com as “page faults”

(A) Syscall ***cowfork:***

- A implementação da syscall cowfork é semelhante à syscall fork

- A diferença é que os processos irão compartilhar das mesmas páginas, ao invés de copiarmos todas elas

- Tal mudança será feita na função ***copyuvm,*** agora chamada: ***cow_copyuvm***

```
//!MODIFICADO
pde_t*
cow_copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;

  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);
    *pte |= PTE_COW;                    // Marcando com página compartilhada
    *pte &= ~PTE_W;                     // Passa a permitir somente leitura
    flags = PTE_FLAGS(*pte);

    if(mappages(d, (void*)i, PGSIZE, pa, flags) < 0) {
      goto bad;
    }

    // Sempre que um filho aponta para a mesma página, incrementar a referencia em 1
    incrementReferenceCount(pa);

  }
  // Realizando flush no TLB devido às alteracões feitas
  lcr3(V2P(pgdir));
  return d;

bad:
  freevm(d);

  lcr3(V2P(pgdir));
  return 0;
}
```

- - No código acima, adicionamos a macro PTE_COW, que indica a página está sendo compartilhada.

- Além disso, negamos a permissão de escrita nas páginas para os processos pais e filhos.

- Adicionamos também, uma função ***incrementReferenceCount*** que incrementa o contador de processos que estão compartilhando determinada página.

- Usamos da função do xv6 ***lcr3*** para atualizarmos o TLB sempre que uma alteração for feita.

(B) Alterações Feitas em ***kalloc.c:***

- Adicionando à estrutura de dados:

uint free_pages;

uint pg_refcount[PHYSTOP >> PGSHIFT]

```
struct {
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
  uint free_pages;                      // <----
  uint pg_refcount[PHYSTOP >> PGSHIFT]; // <----
} kmem;
```

- Método para determinarmos quantas páginas estão livres (usada para testes, posteriormente):

```
uint numFreePages(void)
{
  acquire(&kmem.lock);
  uint free_pages = kmem.free_pages;
  release(&kmem.lock);
  return free_pages;
}
```

- Métodos para manipularmos o contador de processos que compartilham de uma mesma página:

```
void decrementReferenceCount(uint pa)
{
  if(pa < (uint)V2P(end) || pa >= PHYSTOP)
    panic("decrementReferenceCount");

  acquire(&kmem.lock);
  --kmem.pg_refcount[pa >> PGSHIFT];
  release(&kmem.lock);
}

void incrementReferenceCount(uint pa)
{
  if(pa < (uint)V2P(end) || pa >= PHYSTOP)
    panic("incrementReferenceCount");

  acquire(&kmem.lock);
  ++kmem.pg_refcount[pa >> PGSHIFT];
  release(&kmem.lock);
}
uint getReferenceCount(uint pa)
{
  if(pa < (uint)V2P(end) || pa >= PHYSTOP)
    panic("getReferenceCount");
  uint count;

  acquire(&kmem.lock);
  count = kmem.pg_refcount[pa >> PGSHIFT];
  release(&kmem.lock);

  return count;
}
```

- Mudanças na função ***kalloc,*** que passa a ter um contador de processos por página alocada e passa a manipular o total de páginas livres**.** Quando aloca-se uma página, decrementa-se o valor de páginas livres.

```
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  if(r) {
    kmem.freelist = r->next;
    kmem.free_pages--;                              // Ao  alocar 1 pg, diminuir o número de páginas livres
    kmem.pg_refcount[V2P((char *) r) >> PGSHIFT] = 1; // contador de referencia para a página passa a ser 1
  }

  if(kmem.use_lock)
    release(&kmem.lock);
  return (char*)r;
}
```

(C) Função ***pagefault:***

- Lida com os pagefaults que ocorrem

- Quando ocorre um “pagefault”, o endereço virtual em que ocorreu é passado para o registrador CR2.

```
void pagefault(uint error_code) {

  // cprintf("Ocorreu Page Fault!!!!\n");
  // Registrado CR2 "pega" o endereco virtual em que ocorreu a falha
  uint virtual_addr = rcr2();
  pte_t *pte;

  // Verificando se o endereco virtual é ilegal (não está entre os mapeados na tabela de pgs do processo)
  if(virtual_addr >= KERNBASE || (pte = walkpgdir(myproc()->pgdir, (void*) virtual_addr, 0)) == 0 ||
    !(*pte & PTE_P) || !(*pte & PTE_U)) {

      cprintf("Endereco Virtual Ilegal! cpu: %d\t endereco: 0x%x\n", mycpu()->apicid, virtual_addr);
      cprintf("Eliminando o processo %s com pid = %d\n", myproc()->name, myproc()->pid);

      myproc()->killed = 1;
      return;
  }

  // Falha de Página causada por escrita
  if(*pte & PTE_W) {
    cprintf("Erro: %x, endereco 0x%x\n", error_code, virtual_addr);
    panic("Já está permitido escrever\n");
  }
```

```
// Checando se a página NÃO É compartilhada, ou seja, se PTE_COW = 0;
// Se NÃO for: elimina o processo
if(!(*pte & PTE_COW)) {

  myproc()->killed = 1;
  return;

} else {
  uint physical_addr = PTE_ADDR(*pte);
  uint refCount = getReferenceCount(physical_addr);
  char *mem;

  if (refCount > 1) {

    // Alocando uma NOVA página para o processo
    if ((mem = kalloc()) == 0) {
      cprintf("Não foi possível alocar memória\n");
      cprintf("Eliminando proc %s com pid = %d\n", myproc()->name, myproc()->pid);
      myproc()->killed = 1;
      return;
    }

    // Copiando o conteúdo da página original apontada pelo endereco virtual
    memmove(mem, (char *)P2V(physical_addr), PGSIZE);

    // Apontando o apontador pte para a nova página alocada
    *pte = V2P(mem) | PTE_P | PTE_U | PTE_W;

    // Como mudamos a direcão do apontador pte para a nova página
    // Devemos decrementar o refCount em 1
    decrementReferenceCount(physical_addr);

  }
```

- Após a copiarmos a página, atualizamos o TLB com a função ***lcr3***

```
    else if(refCount == 1) {
      // Removendo a restricão de leitura, pois a página só possui uma referência
      // Alterando o status da página para NÃO compartilhada
      *pte |= PTE_W;
      *pte &= ~PTE_COW;

    } else {
      panic("Contagem de referencias está errada (nunca deve ser 0)\n");
    }

    // Realizando o flush no TLB para atualizar as informacões
    lcr3(V2P(myproc()->pgdir));
  }

}
```

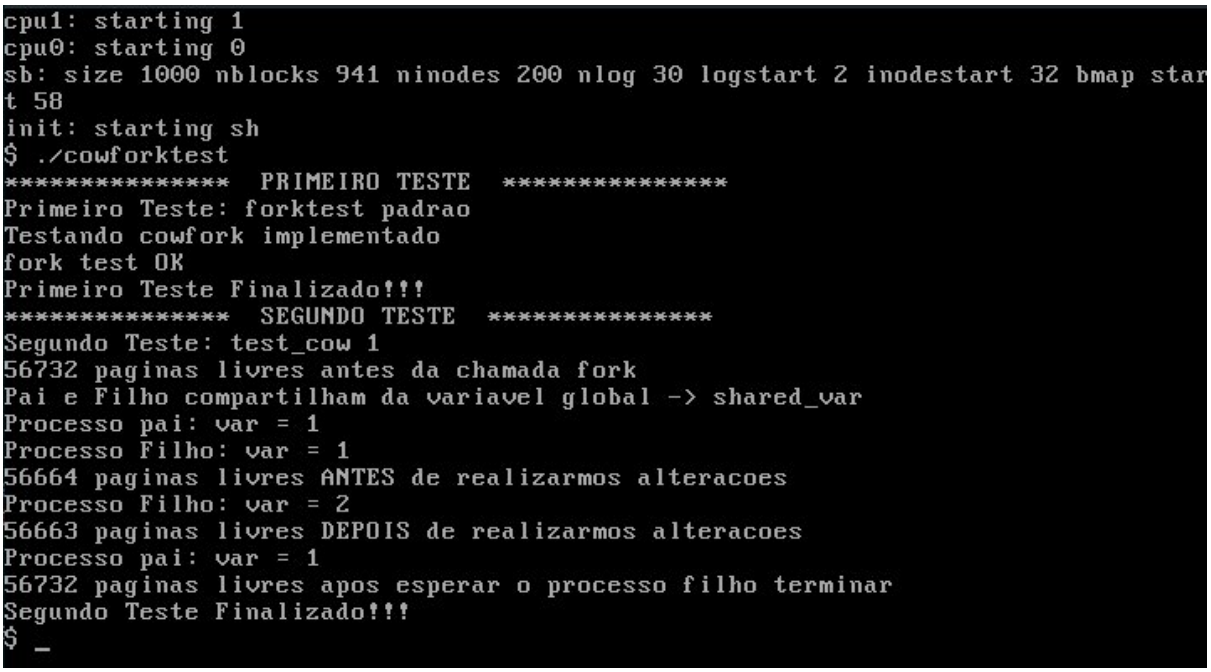
2.4.2 Teste de Implementação:

- cowforktest.c

```
// Variável Global compartilhada pelos pai e filhos
int shared_var = 1;

void
cowforktest(void)
{
  int n, pid;

  printf(1, "Testando cowfork implementado\n");

  for(n=0; n<N; n++){
    pid = cowfork();
    if(pid < 0)
      break;
    if(pid == 0)
      exit();
  }

  if(n == N){
    printf(1, "fork claimed to work N times!\n", N);
    exit();
  }

  for(; n > 0; n--){
    if(wait() < 0){
      printf(1, "wait stopped early\n");
      exit();
    }
  }

  if(wait() != -1){
    printf(1, "wait got too many\n");
    exit();
  }

  printf(1, "fork test OK\n");
}
```

```c
void test_cow1() {

  printf(1, "%d paginas livres antes da chamada fork\n", getNumFreePages());
  printf(1, "Pai e Filho compartilham da variavel global -> shared_var\n");

  // chamando fork
  int pid = cowfork();

  if (pid < 0) {
    printf(1, "Falha no teste_cow 1\n");
  }

  // Filho
  if (pid == 0) {

    // Antes
    printf(1, "Processo Filho: var = %d\n", shared_var);
    printf(1, "%d paginas livres ANTES de realizarmos alteracoes\n", getNumFreePages());

    shared_var = 2;

    // Depois
    printf(1, "Processo Filho: var = %d\n", shared_var);
    printf(1, "%d paginas livres DEPOIS de realizarmos alteracoes\n", getNumFreePages());
    exit();
  }

    // Pai
    printf(1, "Processo pai: var = %d\n", shared_var);

    wait(); // Espera o processo filho terminar

    printf(1, "Processo pai: var = %d\n", shared_var);
    printf(1, "%d paginas livres apos esperar o processo filho terminar\n", getNumFreePages());
    return;

}
```

2.4.3 Resultados:



- A partir do resultado obtido concluimos que:

1. Após os processos serem criados, páginas são alocadas para eles (Pai e Filho) de maneira compartilhada
2. Quando há a necessidade de escrita na memória, cria-se uma cópia da página e o **apontador pte** passa a apontar para essa nova página alocada (paginas livres - 1). E, nela será possível escrever a informação desejada (No caso a mudança do valor da variável ***shared_var***), pois, agora, possui permissão para escrita.
3. Ao final, após ambos os processos terminarem, as páginas são desalocadas e voltam a serem livres.