Memory management

陈建腾, 1120213582, 单人完成
jianteng.chen@bit.edu.cn

1. 实验要求

1.1. Objectives

• Modify memory layout to move stack to top of address space (70%)

• Implement stack growth (30%)

1.2. Part 1: Changing memory layout

1.2.1. Overview

In this part, you'll be making changes to the xv6 memory layout. Sound simple? Well, there are a few tricky details.

Details

In xv6, the VM system uses a simple two-level page table. You may find the description in Chapter 1 of the xv6 manual sufficient (and more relevant to the assignment).

The xv6 address space is currently set up like this:

```
stack (fixed-sized, one page)
heap (grows towards the high-end of the address space)
```

In this part of the xv6 project, you'll rearrange the address space to look more like Linux:

```
heap (grows towards the high-end of the address space)
... (gap)
stack (at end of address space; grows backwards)
```

You can see the general map of the kernel memory in `memlayout.h`; the user memory starts at 0 and goes up to KERNBASE. Note that we will not be changing the kernel memory layout at all, only the user memory layout

Right now, the program memory map is determined by how we load the program into memory and set up the page table (so that they are pointing to the right physical pages). This is all implemented in `exec.c` as part of the exec system call using the underlying support provided to implement virtual memory in `vm.c`. To change the memory layout, you have to change the exec code to load the program and allocate the stack in the new way that we want. Moving the stack up will give us space to allow it to grow, but it complicates a few things. For example, right now xv6 keeps track of the end of the virtual address space using one value (sz). Now you have to keep more information potentially e.g., the end of the bottom part of the user memory (i.e., the top of the heap, which is called brk in un*x), and bottom page of the stack.

Once you figure out in `exec.c` where xv6 allocates and initializes the user stack; then, you'll have to figure out how to change that to use a page at the high-end of the xv6 user address space, instead of one between the code and heap.

Some tricky parts: Let me re-emphasize: one thing you'll have to be very careful with is how xv6 currently tracks the size of a process's address space (currently with the sz field in the proc struct). There are a number of places in the code where this is used (e.g., to check whether an argument passed into the kernel is valid; to copy the address space). We recommend keeping this field to track the size of the

code and heap, but doing some other accounting to track the stack, and changing all relevant code (i.e., that used to deal with sz) to now work with your new accounting. Note that this potentially includes the shared memory code that you are writing for part 2.

1.3. Part 2: Growing the Stack

The final item, which is challenging: automatically growing the stack backwards when needed. Getting this to work will make you into a kernel boss, and also get you those last 10% of credit. Briefly, here is what you need to do. When the stack grows beyond its allocated page(s) it will cause a page fault because it is accessing an unmapped page. If you look in `traps.h`, this trap is T_PGFLT which is currently not handled in our trap handler in `trap.c`. This means that it goes to the default handling of unknown traps, and causes a kernel panic.

So, the first step is to add a case in trap to handle page faults. For now, your trap handler should simply check if the page fault was caused by an access to the page right under the current top of the stack. If this is the case, we allocate and map the page, and we are done. If the page fault is caused by a different address, we can go to the default handler and do a kernel panic like we did before.

Bonus (5%): Write code to try and get the stack to grow into the heap. Were you able to? If not explain why in detail showing the relevant code.

1.4. Hints

IMPORTANT Check the help file posted to walk you through this assignment.

Particularly useful for this project: Chapter 1 of xv6 + anything else about fork() and exec(), as well as virtual memory.

Take a look at the exec code in exec.c which loads a program into memory. It will be using VM functions from vm.c such as allocuvm (which uses mappages). These will be very instructive for implementing shm_open – we are allocating a new page the first time (similar to allocuvm) and adding it to the page table (similar to mappages). It may be helpful to try to answer these questions to yourself:

• Read chapter 2 in the xv6 book. Briefly explain the operation of allocuvm() and mappages() and Figure 1-2. Check how exec uses them for an idea. • Explain how you would given a virtual address figure out the physical address if the page is mapped otherwise return an error. In other words, how would you find the page table entry and check if its valid, and how would you use it to find the physical address. • Find where in the code we can figure out the location of the stack.

2. 实验环境

1. Linux Environment

- OS: Ubuntu 22.04.2 LTS on Windows 10 x86_64
- Kernel: 5.15.90.1-microsoft-standard-WSL2
- Shell: zsh 5.8.1
- CPU: AMD Ryzen 7 5800H with Radeon Graphics (8) @ 3.194GHz
- GPU: af4e:00:00.0 Microsoft Corporation Device 008e

2. xv6 Environment

通过以下代码在Linux环境上安装xv6：

```
sudo apt install qemu-system-x86
cd ~
git clone https://github.com/mit-pdos/xv6-public.git xv6
cd xv6
```

```
make
echo "add-auto-load-safe-path $HOME/xv6/.gdbinit" > ~/.gdbinit
```

3. Start the xv6

分裂终端，在一个终端内输入以下命令：

```
make-nox-gdb
```

在另一终端内打开gdb即可。

3. 实验分析

程序的内存映射取决于我们如何加载程序到内存并设置页表，以便正确地映射到物理页。在操作系统中，这一过程通常作为 `exec` 系统调用的一部分实现，即使用 `vm.c` 中提供的底层支持来管理虚拟内存。因此，如果我们要改变内存布局，我们需要修改 `exec.c` 代码，以实现我们期望的新的程序加载和堆栈分配方式。

`exec.c` 中的 `exec` 函数执行以下操作：

1. 验证并解析用户提供的程序文件路径，确保文件存在且可执行。
2. 调用 `setupkvm()` 函数来初始化内核内存，`setupkvm()` 将内核页映射到进程的虚拟地址空间。
3. 使用 `loaduvm()` 将可执行文件中的段加载到内存中。`loaduvm()` 为每个段创建在内存中的页，并通过初始化页表指针将它们映射到进程的地址空间。xv6将上述段从 `VA0(PGROUNDDOWN(va))` 开始加载，然后向上扩展空间。每个新的段都会从新页面的开头开始。正如已经提到过的，这些段包括代码、全局/静态数据段等。由于xv6使用 `proc->sz` 来定义进程的大小，当我们想要对一个新页面进行映射时时，可以使用 `sz`来作为要映射的段的虚拟地址。
4. 创建栈。由于栈是从上向下扩展的，这意味着没有栈的空间没法被扩展，即栈超过初始的边界继续向下扩展时将会扩展到代码或数据段中(即缓冲区溢出)。为了防止栈溢出，xv6在添加了一个不可访问的页面作为缓冲区。当栈的增长超出预设的栈空间时，它会继续增长到不可访问页面中，这会导致内存错误并终止程序的执行。

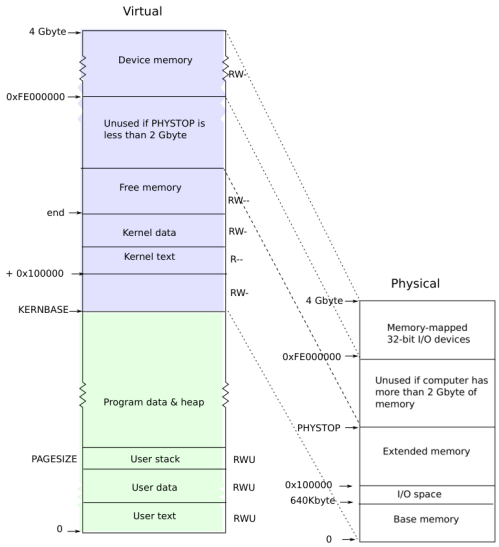

Figure 1: 进程的虚拟地址空间布局和物理地址空间布局。请注意，如果一台机器的物理内存超过2GB，xv6只能使用在KERNBASE和0xFE00000之间的内存。

4. 实验步骤

1. 修改栈空间，在 `proc.h` 中维护一个变量用于记录栈顶指针的位置。

```
// exec.c

int
exec(char *path, char **argv)
{
  char *s, *last;
  int i, off;
  uint argc, sz, stack_pos, sp, ustack[3+MAXARG+1];
  struct elfhdr elf;
  struct inode *ip;
  struct proghdr ph;
  pde_t *pgdir, *oldpgdir;
  struct proc *curproc = myproc();

  begin_op();

  if((ip = namei(path)) == 0){
    end_op();
    cprintf("exec: fail\n");
    return -1;
  }
  ilock(ip);
  pgdir = 0;

  // Check ELF header
  if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
    goto bad;
  if(elf.magic != ELF_MAGIC)
    goto bad;

  if((pgdir = setupkvm()) == 0)
    goto bad;

  // 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;

  // Allocate two pages at the next page boundary.
```

```
  // Make the first inaccessible.  Use the second as the user stack.
  sz = PGROUNDUP(sz);
  // if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
  //   goto bad;
  if ((stack_pos = allocuvm(pgdir, KERNBASE-2*PGSIZE, KERNBASE-PGSIZE)) == 0)
    goto bad;
  // clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
  sp = stack_pos;
  stack_pos -= PGSIZE;

  // Push argument strings, prepare rest of stack in ustack.
  for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
      goto bad;
    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
      goto bad;
    ustack[3+argc] = sp;
  }
  ustack[3+argc] = 0;

  ustack[0] = 0xffffffff;  // fake return PC
  ustack[1] = argc;
  ustack[2] = sp - (argc+1)*4;  // argv pointer

  sp -= (3+argc+1) * 4;
  if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
    goto bad;

  // Save program name for debugging.
  for(last=s=path; *s; s++)
    if(*s == '/')
      last = s+1;
  safestrcpy(curproc->name, last, sizeof(curproc->name));

  // Commit to the user image.
  oldpgdir = curproc->pgdir;
  curproc->pgdir = pgdir;
  curproc->sz = sz;
  curproc->tf->eip = elf.entry;  // main
  curproc->tf->esp = sp;
  curproc->stack_pos = stack_pos;
  switchuvm(curproc);
  freevm(oldpgdir);
  return 0;

 bad:
  if(pgdir)
    freevm(pgdir);
  if(ip){
    iunlockput(ip);
    end_op();
  }
  return -1;
}
```

```
// proc.h
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  uint stack_pos;              // Stack position
};
```

2. 后续工作：由于栈空间的修改，我们需要修改系统调用中访问用户空间的函数。

```
// Fetch the int at addr from the current process.
int
fetchint(uint addr, int *ip)
{
  // struct proc *curproc = myproc();

  // if(addr >= curproc->sz || addr+4 > curproc->sz)
  //   return -1;
  // *ip = *(int*)(addr);
  // return 0;
  *ip = *(int *) (addr);
  return 0;
}

// Fetch the nul-terminated string at addr from the current process.
// Doesn't actually copy the string - just sets *pp to point at it.
// Returns length of string, not including nul.
int
fetchstr(uint addr, char **pp)
{
  char *s, *ep;
  struct proc *curproc = myproc();
  *pp = (char *) addr;
  ep = (char *) curproc->sz;
  for (s = *pp; s < ep; s++){
    if (*s == 0)
      return s - *pp;
  }
  return -1;
  }

// Fetch the nth 32-bit system call argument.
int
argint(int n, int *ip)
{
  return fetchint((myproc()->tf->esp) + 4 + 4*n, ip);
```

```
}

// Fetch the nth word-sized system call argument as a pointer
// to a block of memory of size bytes.  Check that the pointer
// lies within the process address space.
int
argptr(int n, char **pp, int size)
{
  // int i;
  // struct proc *curproc = myproc();

  // if(argint(n, &i) < 0)
  //   return -1;
  // if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz)
  //   return -1;
  // *pp = (char*)i;
  // return 0;
  int i;

  if (argint(n, &i) < 0)
      return -1;
  *pp = (char *) i;
  return 0;
}

// Fetch the nth word-sized system call argument as a string pointer.
// Check that the pointer is valid and the string is nul-terminated.
// (There is no shared writable memory, so the string can't change
// between this check and being used by the kernel.)
int
argstr(int n, char **pp)
{
  // int addr;
  // if(argint(n, &addr) < 0)
  //   return -1;
  // return fetchstr(addr, pp);
  int addr;
  if(argint(n, &addr) < 0)
    return -1;
  return fetchstr(addr, pp);
}
```

3. 修改 `copyuvm()`，这个函数在 `fork()` 调用以创建子进程。

```
// vm.c
pde_t*
copyuvm(pde_t *pgdir, uint sz, uint stack_pos)
{
  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) {
      kfree(mem);
      goto bad;
    }
  }

  for (i = stack_pos; i < KERNBASE - PGSIZE; 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)
    {
        kfree(mem);
        goto bad;
    }
  }

  return d;

bad:
  freevm(d);
  return 0;
}
```

4. 后续工作：修改 `fork` 中调用 `copyuvm` 函数的地方与 `defs.h`。

```
int
fork(void)
{
  ...
  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz, curproc->stack_pos)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  ...
}

pde_t*          copyuvm(pde_t*, uint, uint);
```

5. 在 `trap.c` 中增加系统中断并实现栈的增长，这里我们选择 `traps.h` 中定义的陷阱并添加相应 case。当页错误出现时，我们检查引发页错误的地址,使用 `rcr2()` 函数去获得 CR2寄存器 中存

放的地址。一旦我们获得了这个地址，接下来我们就可以检查该地址是否位于栈底，并相应进行栈增长操作，调用`allocuvm()`函数来分配一个新的页面。

```
//traps.h

// #define T_COPROC      9      // reserved (not used since 486)
#define T_TSS           10      // invalid task switch segment
#define T_SEGNP         11      // segment not present
#define T_STACK         12      // stack exception
#define T_GPFLT         13      // general protection fault
#define T_PGFLT         14      // page fault
```

```
// trap.c
void
trap(struct trapframe *tf)
{
  ...
  switch(tf->trapno){
    ...
    case T_PGFLT:
      if (rcr2() < 0x7FFFF000){
        cprintf("page error %x ",rcr2());
        cprintf("stack pos : %x\n", myproc()->stack_pos);
        if ((myproc()->stack_pos = allocuvm(myproc()->pgdir, myproc()->stack_pos - 1 *
PGSIZE, myproc()->stack_pos)) == 0)
          myproc()->killed = 1;
        myproc()->stack_pos-=PGSIZE;
        cprintf("create a new page %x\n", myproc()->stack_pos);
        return;
      }
      else{
          myproc()->killed = 1;
          break;
      }
    ...
    }
  ...
}
```

6. 创建用户态程序 `stack_test.c` 并相应修改 `Makefile`。

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

int
main(int argc, char *argv[])
{
    printf(0, "\nargument num : %d\n", argc - 1);
    int n = 100;
    if (argc > 1)
    {
        for (int i = 1; i < argc; i++)
        {
            printf(0, "%s\n", argv[i]);
        }
        n = atoi(argv[1]);
        printf(0, "\ncreate %d int array\n", n);
```

```
    }
    else
    {
        printf(0, "\ncreate 100 int array\n");
    }

    int num[n];
    memset(num, 0, sizeof (num));
    printf(0,"the origination address of address num is: %x\n",num);

    int pid = fork();
    if (pid < 0)
    {
        printf(0, "fork error!\n");
    }
    else if (pid == 0)
    {
        printf(0, "\nchild %d fork\n", getpid());

        printf(0, "***child***\n");
    }
    else
    {
        wait();
        printf(0, "parent %d kill\n\n", getpid());
    }
    return 0;
}
```

5. 运行结果

```
$ stack_test

argument num : 0

create 100 int array
the origination address of address num is: 7FFFEE30

child 4 fork
***child***
parent 3 kill
```

```
$ stack_test 2000

argument num : 1
2000

create 2000 int array
page error 7fffdfac stack pos : 7fffe000
create a new page 7fffd000
the origination address of address num is: page error 7fffcffc stack pos : 7fffd000
create a new page 7fffc000
7FFFD070

child 6 fork
***child***
parent 5 kill
```

```
$ stack_test 10000

argument num : 1
10000

create 10000 int array
page error 7fffdfac stack pos : 7fffe000
create a new page 7fffd000
page error 7fffcfac stack pos : 7fffd000
create a new page 7fffc000
page error 7fffbfac stack pos : 7fffc000
create a new page 7fffb000
page error 7fffafac stack pos : 7fffb000
create a new page 7fffa000
page error 7fff9fac stack pos : 7fffa000
create a new page 7fff9000
page error 7fff8fac stack pos : 7fff9000
create a new page 7fff8000
page error 7fff7fac stack pos : 7fff8000
create a new page 7fff7000
page error 7fff6fac stack pos : 7fff7000
create a new page 7fff6000
page error 7fff5fac stack pos : 7fff6000
create a new page 7fff5000
the origination address of address num is: 7FFF5370

child 8 fork
***child***
parent 7 kill

$
```