



به نام خدا



دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس آزمایشگاه سیستم عامل

پروژه پنجم

دانیال سعیدی(810198571)

سروش صادقیان(810898048)

محمد قره حسنلو(810198461)

نام و نام خانوادگی

تاریخ ارسال گزارش

ریپو گیتهاب

آخرین Commit ID

فهرست گزارش سوالات

2.....	سوال 1
3.....	سوال 2
3.....	سوال 3
3.....	سوال 4
3.....	سوال 5
3.....	سوال 7

سوال 1

در لینوکس، هسته از نواحی memory mapping پیگیری virtual memory های پردازه استفاده میکند؛ مثلا یک process یک VMA برای کدش، یک VMA برای هر نوع دیتا، یک VMA برای هر memory mapping در صورت وجود داشتن، داشته باشد. هر VMA شامل تعدادی page هست که هر کدام یک entry به page table دارد. اما در xv6، از آدرس های مجازی 32 بیتی استفاده میکند که فضای آدرسی مجازی 4 گیگابایتی را ایجاد میکند. همچنین xv6 از ساختار جدول دو سطحی استفاده میکند. xv6 مفهومی از جاگذاری مجازی ندارد.

سوال 2

چون که از صفحه پردازه ای استفاده میشود که میخواهیم از آن استفاده کنیم و بقیه را که لازم شان نداریم را لود نمیکنیم و اینکه فقط آدرس شروع صفحه را در اینجا نگه داری میکنیم و نیازی به نگه داری همه آدرس های یک صفحه نمیشود و میتوانیم بقیه آدرس ها را با offset دسترسی پیدا کنیم. همچنین در اینجا با mapping کدها به اشتراک گذاشته میشوند که حافظه کمتری را در نتیجه لازم خواهد داشت.

سوال 3

12 بیت برای entry که دو سطح (page table و page directory) داشته و در بیت D باهم متفاوت اند که در page directory یعنی تغییرات در زمان نوشتن صفحه در دیسک اعمال میشود و در page table مقدار آن تاثیری ندارد، 20 بیت برای پوینتر به سطح بعدی هم وجود دارد.

سوال 4

فضایی از حافظه فیزیکی را تخصیص میدهد kalloc

سوال 5

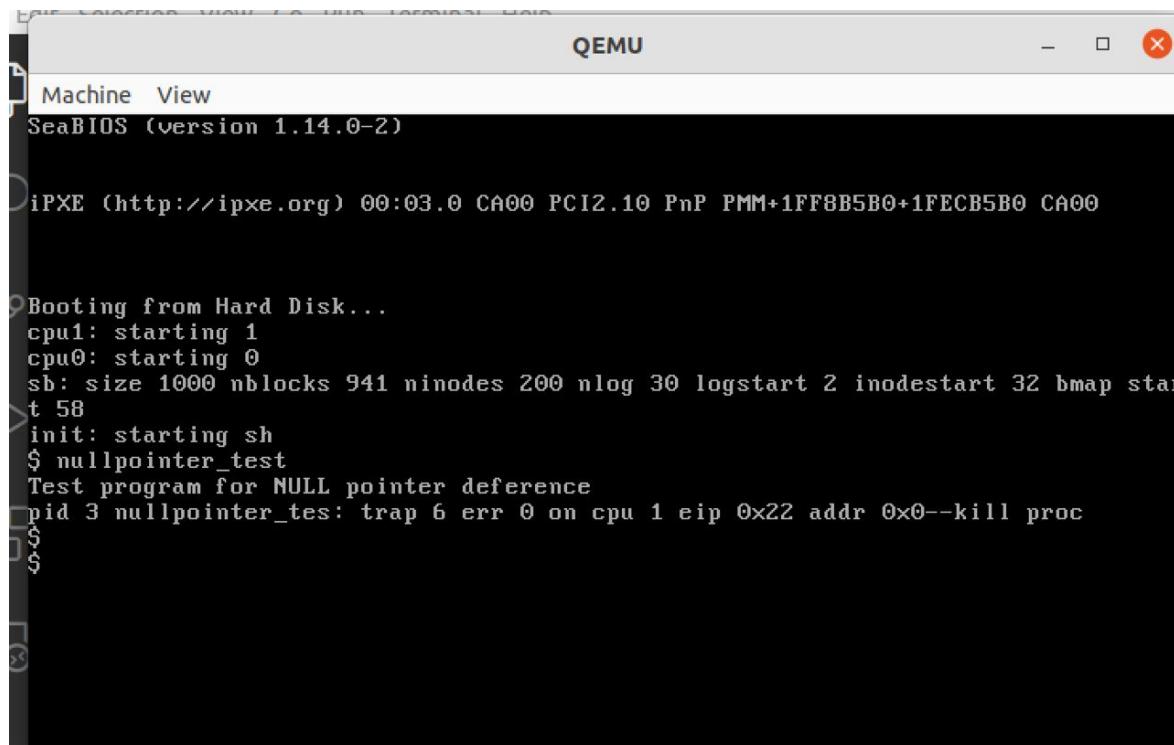
mappages آدرس یک خانه از حافظه فیزیکی و مجازی همراه با سایز را گرفته و صفحه موجود در را در آدرس حافظه مجازی بارگذاری میکند و این کار را با توجه به سایزی که دادیم

انجام میدهد. دلیل آن به این صورت است که صفحه مربوط به پردازه در حال اجرا را به درستی نشان دهد.

سوال 7

در آدرس یک خانه حافظه مجازی سطح page directory را میگیرد و آدرس که در همان خانه حافظه مجازی است را برمیگرداند.

Read Only



The screenshot shows a terminal window titled "QEMU" running SeaBIOS (version 1.14.0-2). The terminal output includes:

```
Machine View
SeaBIOS (version 1.14.0-2)

ipXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B5B0+1FECB5B0 CA00

Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ nullpointer_test
Test program for NULL pointer deference
pid 3 nullpointer_tes: trap 6 err 0 on cpu 1 eip 0x22 addr 0x0--kill proc
$
```

A screenshot of a macOS desktop environment showing a VMware Fusion window running an Ubuntu 64-bit 21.10 virtual machine. Inside the VM, Visual Studio Code is open with a file named `usertests.c`. The code is a C program with assembly intermix. It includes a loop that tries to crash the kernel by passing a bad string pointer to `link("nosuchfile", (char*)p)`. The code is annotated with comments explaining its purpose.

```
1552     |     |     "ebx");
1553 }
1554
1555 void
1556 validateTest(void)
1557 {
1558     int hi, pid;
1559     uint p;
1560
1561     printf(stdout, "validate test\n");
1562     hi = 1100*1024;
1563     for(p = 4096; p <= (uint)hi; p += 4096){
1564         if((pid = fork()) == 0){
1565             // try to crash the kernel by passing in a badly placed integer
1566             validateInt((int*)p);
1567             exit();
1568         }
1569         sleep(0);
1570     }
1571     sleep(0);
1572     kill(pid);
1573     wait();
1574
1575     // try to crash the kernel by passing in a bad string pointer
1576     if(link("nosuchfile", (char*)p) != -1){
1577         printf(stdout, "link should not succeed\n");
1578         exit();
1579     }
1580
1581     printf(stdout, "validate ok\n");
1582 }
```

A screenshot of a macOS desktop environment showing a VMware Fusion window running an Ubuntu 64-bit 21.10 virtual machine. Inside the VM, Visual Studio Code is open with a file named `exec.c`. The code is a C program that handles program execution. It includes logic for reading ELF files and loading them into memory. The code is annotated with comments explaining its purpose.

```
38     if((pgdir = setupKvm()) == 0)
39         goto bad;
40
41     // Load program into memory.
42     sz = PGSIZE;
43
44     if(read1(ip, (char*)ph, off, sizeof(ph)) != sizeof(ph))
45         goto bad;
46     if(ph.type != ELF_PROG_LOAD)
47         continue;
48     if(ph.memsz < ph.filesz)
49         goto bad;
50     if(ph.vaddr + ph.memsz < ph.vaddr)
51         goto bad;
52     if((sz = allocUvM(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
53         goto bad;
54     if(ph.vaddr % PGSIZE != 0)
55         goto bad;
56     if(loadUvM(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
57         goto bad;
58
59     iunlockput(ip);
60     end_op();
61     ip = 0;
62
63     // Allocate two pages at the next page boundary.
64     // Make the first inaccessible. Use the second as the user stack.
65     sz = PGROUNDUP(sz);
66     if((sz = allocUvM(pgdir, sz, sz + 2*PGSIZE)) == 0)
67         goto bad;
68     clearpteU(pgdir, (char*)(sz - 2*PGSIZE));
```

The screenshot shows a Visual Studio Code window running on an Ubuntu 64-bit 21.10 system. The title bar indicates the file is 'syscall.c - xv6-public - Visual Studio Code'. The code editor displays the 'syscall.c' file, which contains C code for handling system calls. The code includes functions like 'argptr' and 'argstr' that check pointers and string arguments against memory boundaries and kernel usage. The left sidebar shows the project structure under 'EXPLORER', including files like 'usertests.c', 'exec.c', 'vm.c', and 'testnull.c'. The bottom status bar shows the current line (Ln 71, Col 2), spaces (Spaces: 2), encoding (UTF-8), and file type (Linux).

```
C syscall.c > argptr(int, char **, int)
56 // to a BLOCK of memory of size bytes. Check that the pointer
57 // lies within the process address space.
58 int
59 argptr(int n, char **pp, int size)
60 {
61     int i;
62     struct proc *curproc = myproc();
63
64     if(argint(n, &i) < 0)
65         return -1;
66     // when i == 0 , means the point to the first part of address space
67     if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz || i == 0)
68         return -1;
69     *pp = (char*)i;
70     return 0;
71 }
72
73 // Fetch the nth word-sized system call argument as a string pointer.
74 // Check that the pointer is valid and the string is nul-terminated.
75 // (There is no shared writable memory, so the string can't change
76 // between this check and being used by the kernel.)
77 int
78 argstr(int n, char **pp)
79 {
80     int addr;
81     if(argint(n, &addr) < 0)
82         return -1;
83     return fetchstr(addr, pp);
84 }
85
86 extern int sys_chdir(void);
87 extern int sys_close(void);
```

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

    if(d = setupkvm()) == 0)
        return 0;
    for(; 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;
        }
    }
    return d;
}
bad:
```

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

#define NULL 0
#define stdout 1

int main()
{
    printf(stdout, "Test program for null ptr\n");
    int *p = NULL;
    printf(1, "*p: %d\n", *p);
    exit();
}
```

```
grep init 2 6 10500
init 2 7 15176
kill 2 8 15030
ls 2 9 14532
2 10 17104
mkdir 2 11 14660
rm 2 12 14649
rmdir 2 13 14629
sh 2 14 15568
stressfs 2 15 62788
usertests 2 16 16088
wc 2 17 14208
zombie 2 18 16200
strdiff 2 19 14184
ps 2 20 14580
foo 2 21 14428
set_queue 2 22 14504
set_bjf 2 23 14484
set_all_bjf 2 24 14248
console 3 25 0
$ testnull
Test program for null ptr
pid 4 testnull: trap 14 err 4 on cpu 0 eip 0x101d addr 0x0--kill proc
```

The screenshot shows a Visual Studio Code window running on an Ubuntu 64-bit 21.10 virtual machine via VMware Fusion. The title bar indicates the file is named 'Makefile' and is located in the 'xv6-public' folder under 'Visual Studio Code'. The code editor displays a Makefile with several rules and definitions. A search term 'syscall' is highlighted in the code. The left sidebar shows the project structure with files like 'In.sym', 'log.c', 'log.d', 'log.o', 'ls.asm', 'ls.c', 'ls.d', 'ls.o', 'ls.sym', 'main.c', 'main.d', 'main.o', 'Makefile', 'memide.c', 'memlayout.h', 'mkdir.asm', 'mkdir.c', 'mkdir.d', 'mkdir.o', 'mkdir.sym', 'mkfs', 'mkfs.c', 'mmu.h', 'mp.c', 'mp.d', and 'UPROGS'. The status bar at the bottom shows 'Ln 147, Col 1' and other terminal-related information.

```
tags: $(OBJS) entry.o other.o _INIT_
      etags *.S *.c

vectors.S: vectors.pl
        ./vectors.pl > vectors.S

ULIB = ulib.o usys.o printf.o umalloc.o

%.o: %.S $(ULIB)
    $(LD) $(LDFLAGS) -N -e main -Ttext 0x1000 -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

mkfs: mkfs.c fs.h
      gcc -Werror -Wall -o mkfs mkfs.c

# 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\

Ln 147, Col 1  Tab Size: 4  UTF-8  LF  Makefile
```

Next Part

A screenshot of a macOS desktop environment showing a VMware Fusion window running an Ubuntu 64-bit 21.10 virtual machine. The VM's taskbar at the top shows icons for various applications like Finder, Mail, and Safari. The main window is Visual Studio Code, displaying the file `sysproc.c` from a project named `xv6-public`. The code implements system calls for memory protection. The interface includes a sidebar with file navigation, a central code editor with syntax highlighting, and a status bar at the bottom.

```
117 }
118 void sys_set_all_bjf_params(void)
119 {
120     int priority_ratio, arrival_time_ratio, executed_cycle_ratio;
121     argint(0, &priority_ratio);
122     argint(1, &arrival_time_ratio);
123     argint(2, &executed_cycle_ratio);
124     set_all_bjf_params(priority_ratio, arrival_time_ratio, executed_cycle_ratio);
125 }
126
127 int
128 sys_mprotect(void){
129     int d;
130     int n = 0;
131     if(argint(0, &d)<0 || argint(1, &n)<0)
132         return -1;
133     return mprotect((void *)d,n);
134 }
135
136 int
137 sys_munprotect(void){
138     int d;
139     int n = 0;
140     if(argint(0, &d)<0 || argint(1, &n)<0)
141         return -1;
142     return munprotect((void *)d,n);
143 }
144
145 }
146 }
```

A screenshot of the same VMware Fusion setup, showing the file `vm.c` in Visual Studio Code. This file contains C code for a system call handler. It includes comments explaining the purpose of the code, such as "mprotect system call makes page table entries both readable and writable". The code implements the `mprotect` and `munprotect` functions by looping through page table entries (pte_t) to update their permissions. The interface and file structure are similar to the previous screenshot.

```
426 //mprotect system call makes page table entries both readable and writable
427 int
428 mprotect(void *addr, int len){
429     struct proc *curproc = myproc();
430
431     //Check if addr points to a region that is not currently a part of the address space
432     if(len <= 0 || (int)addr+len>PGSIZE>curproc->sz){
433         sprintf("\nwrong len\n");
434         return -1;
435     }
436
437     //Check if addr is not page aligned
438     if((int)((long)addr) % PGSIZE != 0){
439         sprintf("\nwrong addr %p\n", addr);
440         return -1;
441     }
442
443     //Loop for each page
444     pte_t *pte;
445     int i;
446     for (i = (int)addr; i < ((int)addr + (len) * PGSIZE); i+= PGSIZE){
447         // Getting the address of the PTE in the current process's page table (pgdir)
448         // that corresponds to virtual address (i)
449         pte = walkpgdir(curproc->pgdir, (void*) i, 0);
450         if(pte && ((pte & PTE_U) != 0) && ((pte & PTE_P) != 0) ){
451             *pte = (*pte) | (PTE_W) ; //Setting the write bit
452             sprintf("\nPTE : 0x%p\n", pte);
453         } else {
454             return -1;
455         }
456     }
457     //Reloading the Control register 3 with the address of page directory
458     //to flush TLB
459     lcr3(V2P(curproc->pgdir));
460
461     return 0;
462 }
463 }
```

The screenshot shows a VMware Fusion window with an Ubuntu 64-bit 21.10 desktop environment. A Visual Studio Code window is open, displaying a C program named `protection_test.c`. The code demonstrates memory protection using `sbrk()`, `mprotect()`, and `fork()`. It prints the value at a memory location before and after protection, and shows that a statement is not printed if it crosses a page boundary. The code includes imports for `types.h`, `stat.h`, `user.h`, and `mmu.h`.

```
1 #include <types.h>
2 #include <stat.h>
3 #include <user.h>
4 #include <mmu.h>
5 int
6 main(int argc, char *argv[])
{
7     char *start = sbrk(0);
8     sbrk(PGSIZE);
9     *start=100;
10    mprotect(start, 1) ;
11    int child=fork();
12    if(child==0){
13        if(child==0){
14            printf(1, "protected value = %d\n",*start);
15            munprotect(start, 1) ;
16            *start=5;
17            printf(1, "After unprotecting the value became = %d\n",*start);
18            exit();
19        }
20    } else if(child>0){
21        wait();
22        printf(1, "\nTrap now\n");
23        *start=5;
24        printf(1, "\nThis statement will not be printed\n");
25        exit();
26    }
27 }
28
29 }
```

Booting from Hard Disk...

cpu0: starting

sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58

t 58

init: starting sh

\$ protection_test

Trap now

pid 3 protection_test: trap 14 err 7 on cpu 0 eip 0x1055 addr 0x4000--kill proc

\$

exec: fail

exec failed

\$

443 //loop for each page

444 pte_t *pte;

445 int i;

446 for (i = (int) addr; i < ((int) addr + (len) *PGSIZE); i+= PGSIZE){

447 }

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58

init: starting sh

\$ protection_test

PTE : 0x8dee1010

protected value = 100

PTE : 0x8df26010

After unprotecting the value became = 5

Trap now

pid 3 protection_test: trap 14 err 7 on cpu 0 eip 0x1055 addr 0x4000--kill proc

\$

exec: fail

exec failed

\$

```

C producer_consumer.c      C proc.c      X
C proc.c > ⚡ sem_release(int)
550 }
551
552 #define MAXPROC 100
553
554 typedef struct
555 {
556     int value;
557     struct proc* list[MAXPROC];
558     int last;
559 }semaphore;
560
561 semaphore sems[6];
562
563 void sem_sleep(struct proc *p1)
564 {
565     acquire(&ptable.lock);
566     p1->state = SLEEPING;
567     sched();
568     release(&ptable.lock);
569 }
570
571 void sem_wakeup(struct proc *p1)
572 {
573     acquire(&ptable.lock);
574     p1->state = RUNNABLE;
575     release(&ptable.lock);
576 }
577
578 int sem_init(int i , int v)
579 {
580     sems[i].value = v;
581     sems[i].last = 0;
582     return 0;
583 }
584

```

```
C producer_consumer.c    C proc.c    X
C proc.c > ⚭ sem_acquire(int)
582     return 0;
583 }
584
585     int sem_acquire(int i)
586     {
587         if(sems[i].value <= 0)
588         {
589             struct proc* p = myproc();
590             sems[i].list[sems[i].last] = p;
591             sems[i].last++;
592             sem_sleep(p);
593         }
594         else
595             sems[i].value--;
596
597     return 0;
598 }
599
600     int sem_release(int i)
601 {
602     if(sems[i].last)
603     {
604         sems[i].last--;
605         struct proc* p = sems[i].list[sems[i].last];
606         sem_wakeup(p);
607     }
608     else
609         sems[i].value++;
610
611     return 0;
612 }
```