Pdf 1

سوال 1)

سیستم عامل xv6 از دو بخش user space و kernel space تشکیل شده. در این سیستم عامل از kernel که برنامه ای برای سرویس دهی به برنامه های در حال اجرا است استفاده می شود که این برنامه های در حال اجرا پردازه یا process نام دارند.وقتی یک پردازه به سرویسی از kernel احتیاج دارد از طریق فراخوانی های سیستمی(system call) درخواست می کند که system call وارد kernel می شود kernel سرویس را اجرا می کند و خارج می شود. بنابراین هر پردازه بین فضای کاربر و فضای هسته در حال تناوب است.مجموعه ای از فراخوانی های سیستمی که هسته فراهم می کند رابطی است که برنامه های سمت کاربر میبینند. سیستم عامل xv6 یک سیستم عامل monolithic و multiprocessor , 32 بیتی است و یک سیستم عامل بر مبنی سیستم های یونیکسی است.

سوال 2)

هر پردازه در سیستم عامل xv6 شامل اطلاعات حافظه در فضای کاربر یعنی (دستورالعمل ها, داده و پشته) می باشد.سیستم عامل xv6 پردازه ها را اصطلاحا time-share می کند, به این معنی که پردازنده های قابل استفاده را بین مجموعه ای از پردازه ها که در انتظار اجرا هستند switch می کند. وقتی یک پردازه در حال اجرا نیست xv6 رجیستر های پردازنده را نگه می دارد و دوباره برای اجرای بعدی پردازه آن ها را بازیابی می کند.
هر پردازه با یک شناسه به نام pid برای هسته قابل تشخیص است.

سوال 3)

برای ایجاد یک پردازه از فراخوان سیستمی fork استفاده می شود.fork یک پردازه ی child با همان محتوای حافظه ای پردازه ی parent ایجاد میکند.در پردازه ی parent , آیدی پردازه ی child را بر میگرداند و در پردازه ی child صفر برمیگرداند.

فراخوان سیستمی exec حافظه ی پردازه ی فراخوان کننده را با یک memory image جدید که از یک فایل ذخیره شده در فایل سیستم load شده جایگزین می کند.این فایل باید فرمت به خصوصی داشته باشد, که مشخص می کند کدام قسمت از فایل دستورالعمل ها, کدام قسمت داده و از کدام دستور العمل شروع کند.

Shell xv6 از فراخوان های fork و exec برای اجرا کردن برنامه ها از طرف کاربر استفاده می کند.در حلقه ی اصلی با getcmd یک خط از ورودی خوانده می شود.سپس fork فراخوانده می شود که یک کپی از پردازه ی shell می سازد.پردازه ی parent صبر میکند در حالی که child دستور را اجرا می کند.در پردازه ی child , runcmd با استفاده از فراخوان exec دستور را اجرا می کند سپس خارج شده و به پردازه ی parent باز میگردد.به این دلیل است که پیاده سازی این دو فراخوانی به صورت مجزا هوشمندانه است.

سوال 4)

File descriptor یک عدد صحیح است که فایل باز پردازه را به طور منحصر به فرد مشخص می کند.
Pipe یک بافر هسته ی کوچک است که به عنوان یک جفت file descriptor در اختیار پردازه ها قرار داده می شود.یکی برای خواندن یکی برای نوشتن نوشتن در یک انتهای پایپ داده را برای خوانده شدن در طرف دیگر پایپ در دسترس قرار می دهد.پایپ یک راه برای ارتباط پردازه ها فراهم می کند.

Pdf 2

سوال 2)

پوشه ی اصلی که فایل های هسته ی سیستم عامل, فایل های header و فایل سیستم در آن قرار دارند دایرکتوری root است که با / نشان داده می شود.دایرکتوری root شامل همه ی دایرکتوری ها و فایل ها در سیستم است.وقتی در دایرکتوری root دستور ls می زنیم شاهد محتوایی هستیم که چند دایرکتوری و فایل های مهم را نشان می دهد.مثلا boot/ شامل فایل های است که کامپیوتر برای بوت شدن به آن ها نیاز دارد که bootloader و هسته ی لینوکس درآن قرار دارند و kernel یک فایل به نام vmlinuz است.یا sys/ یک دایرکتوری است که نظیر یک فایل سیستم مجازی است.از ان جایی که در سیستم های مبتنی بر unix همه چیز به عنوان فایل در نظر گرفته می شود برای سخت افزار های کامپیوتر نیز فایل هایی در نظر گرفته می شود که در دایرکتوری dev/ ذخیره می شوند.

فایل سیستم نحوه ی ذخیره و بازیابی داده های ذخیره شده را کنترل می کند.هر گروهی از داده ها به عنوان یک فایل در نظر گرفته می شود که فایل سیستم قواعدی برای مدیریت این داده ها و شناسه های ان ها ایجاد می کند.پس فایل هایی که در زیر گروه فایل سیستم هستند به این منظور طراحی شده و این کار را برای ما انجام می دهند.

سوال 3)

پس از اجرای دستور make -n آخرین دستور مشاهده شده به صورت زیر است :

```
objdump -S kernel > kernel.asm
objdump -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel.sym
dd if=/dev/zero of=xv6.img count=10000
dd if=bootblock of=xv6.img conv=notrunc
dd if=kernel of=xv6.img seek=1 conv=notrunc
```

سوال 8)

دستور objdump به طور کلی اطلاعاتی در مورد یک یا چند object file به ما می دهد که بر حسب نیاز می توان از گزینه های این دستور استفاده کرد که به طور خاص اطلاعاتی را نشان دهد.dwarf[=info,=decodeline— این دستور ها فهرست بخش های debug را در صورت وجود نشان می دهند.

(6

Bootmain.c

(9

بوت لودر XV6 به نزدیک به 470 bytes کد ماشین کامپایل می شود. برای آنکه در آن فضای محدود این حجم جا شود، بوت لودر XV6 یک فرض ساده کننده را در نظر می گیرد و آن اینست که kernel در دیسک بوت بطور پیوسته با شروع از سکتور 1 نوشته شده است. این در حالی است که معمولا کرنل ها در فایل های سیستم معمولی ذخیره شده اند که ممکن است پیوسته نباشند یا از روی نتوورک بارگیری می شوند. این پیچیدگی بوت لودر را ملزم می کند که بتواند دیسک ها و نتوورک کنترلر های متنوعی را درایو کند و فایل های سیستم و پروتکل های نتوورک متنوعی را بفهمد. به بیان دیگر بوت لودر باید خود یک سیستم عامل کوچک باشد. از آن جا که بوت لودر های اینچنین پیچیده قطعا نمی توانند در 512 bytes جا شوند، اکثر سیستم عامل ها باید از یک پروسه ی دو مرحله ای استفاده کنند. ابتدا بوت لودر معمولا با تکیه بر BIOS برای دسترسی به دیسک، بجای تلاش برای درایو کردن دیسک توسط خودش، یک بوت لودر کامل را از یک مکان مشخص در دیسک لود میکند. فول لودر که از بند محدودیت 512 bytes ای رها است، می تواند پیاده سازی پیچیده موردنیاز برای پیدا کردن و بارگیری و اجرای کرنل مورد نظر را انجام دهد.

(11

قبل از عرضه ی 80286 که مد 80286 را معرفی کرد. real mode تنها مد موجودبرای پردازنده های x86 بود. برای پشتیبانی نسل های قبل، همه ی پردازنده های x86 وقتی ریست می شوند، در مد حقیقی بالا میایند. ایراد های وارد به این مد را می توان عدم پشتیبانی از حفاظت حافظه، چندوظیفگی و سطوح امتیاز کد دانست. و اینکه در این مد تنها 1 MB حافظه ی قاب آدرس دهی در اختیار پردازنده است.

(13

دلیل اینکه بوت لودر کرنل را در 0x80100000، جایی که کرنل انتظار دارد دستورالعمل ها و داده هایش را پیدا کند، لود نمی کند این است که ممکن است هیچ حافظه ی فیزیکی ای در یکچنین ادرس بالایی روی یک ماشین کوچک وجود نداشته باشد. دلیل آنکه کرنل را در 0x100000 به جای 0x0 قرار می دهد اینست که رنج آدرس 0xa0000 تا 0x100000 دستگاه های I/O را در بر دارد.

(14

arch/x86/boot/header.S

(15

این جدول نگاشت هر آدرس مجازی را به یک آدرس فیزیکی روی حافظه می برد. برای دسترسی به این جدول از آن جا نمی توان از خود آن برای نگاشت آدرس ها استفاده کردۀ بنابراین مجبور هستیم به آدرس فیزیکی آن بطور مستقیم دسترسی پیدا کنیم.

(18

دستوراتی که برای segmentation استفاده می شوند به سطح دسترسی بالا نیاز دارند. دستوراتی که در حال اجرا در سگمنت هستند نیازمند سطح دسترسی کاربر هستند و نباید سطح دسترسی بالاتر داشته باشند.

1.

#0 sys_read () at sysfile.c:71

#1 0x80104837 in syscall () at syscall.c:139

#2 0x801058b9 in trap (tf=0x8dffefb4) at trap.c:43

#3 0x8010561f in alltraps () at trapasm.S:20

#4 0x8dffefb4 in ?? ()

2.

کد های داخل sys_read را در یک حلقه ی نامتناهی قرار می دهیم((for(;;). یک برک پوینت روی sys_read میگذاریم و با دستورات next، step و finish شروع به دیباگ میکنیم. متوجه خواهیم شد که در :

```
if(argfd(0,
0, &f) < 0
||
argint(2,
&n) < 0 ||
argptr(1,
&p, n) < 0)
                  return -1;
                return fileread(f, p, n);
```

هیچگاه در صورت براورده نشدن شرط if خارج از حلقه نمیتوانیم برویم.

4.

s[tep]i

Execute a single instruction and then return to the command line interpreter.

n[ext]i

Like stepi, except that if the instruction is a subroutine call, the entire subrou-tine is executed before control returns to the interpreter.

5.

(gdb) bt

#0 sys_write () at sysfile.c:83

#1 0x80104837 in syscall () at syscall.c:139

#2 0x801058b9 in trap (tf=0x8dffffb4) at trap.c:43

#3 0x8010561f in alltraps () at trapasm.S:20

#4 0x8dffffb4 in ?? ()

6.

layout asm:

```
B+>|0x80104b50 <sys_write>        push   %ebp                          |
   |0x80104b51 <sys_write+1>      xor    %eax,%eax                     |
   |0x80104b53 <sys_write+3>      mov    %esp,%ebp                     |
   |0x80104b55 <sys_write+5>      sub    $0x18,%esp                    |
   |0x80104b58 <sys_write+8>      lea    -0x14(%ebp),%edx              |
   |0x80104b5b <sys_write+11>     call   0x80104a10 <argfd>            |
   |0x80104b60 <sys_write+16>     test   %eax,%eax                     |
   |0x80104b62 <sys_write+18>     js     0x80104bb0 <sys_write+96>     |
   |0x80104b64 <sys_write+20>     lea    -0x10(%ebp),%eax              |
   |0x80104b67 <sys_write+23>     sub    $0x8,%esp                     |
   |0x80104b6a <sys_write+26>     push   %eax                          |
   |0x80104b6b <sys_write+27>     push   $0x2                          |
   |0x80104b6d <sys_write+29>     call   0x80104720 <argint>
```

layout src

```
┌──────────────────────────────────────────────────────────────────────┐
B+>|83    {                                                           |
   |84      struct file *f;                                           |
   |85      int n;                                                    |
   |86      char *p;                                                  |
   |87                                                               |
   |88      if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) <|
```

```
89        return -1;
90      return filewrite(f, p, n);
91    }
92
93    int
94    sys_close(void)
95    {
```