گزارشکار آزمایش اول سیستم عامل

پارسا احمدی ناو، آریا عازم، مجید صادقی نژاد

آشنایی با سیستم عامل xv6

۱- از آنجایی که تمام اجزای اصلی سیستم عامل به صورت یکپارچه در کرنل پیاده سازی شده اند می توان نتیجه گرفت معماری این سیستم عامل از نوع monolithic kernel است همچنین هیچ ماژولی که خارج از فضای کرنل پیاده سازی شده باشد وجود ندارد که بتوانیم نتیجه گیری کنیم سیستم عامل میکروکرنل است. همچنین توصیف متن داکیومنت xv6 در زیر موجود است.

> This chapter provides an overview of how operating systems are organized to achieve these 3 requirements. It turns out there are many ways to do so, but this text focuses on mainstream designs centered around a *monolithic kernel*, which is used by many Unix operating systems. This chapter introduces xv6's design by tracing the creation of the first process when xv6 starts running. In doing so, the text provides a

۲- یک پردازه در این سیستم عامل از فضای حافظه ی کاربر شامل دستورات، دیتا و فضای استک و همچنین یک استیت پیش پردازش شده که فقط مخصوص کرنل است تشکیل شده، این سیستم عامل از سیستم تایم time share برای اختصاص پردازه ها به پردازنده استفاده می کند به این معنی که پردازه ای که cpu در حال کار بر روی آن است در بازه های زمانی عوض می کند.

> An xv6 process consists of user-space memory (instructions, data, and stack) and per-process state private to the kernel. Xv6 can *time-share* processes: it transparently switches the available CPUs among the set of processes waiting to execute. When a process is not executing, xv6 saves its CPU registers, restoring them when it next runs the process. The kernel associates a process identifier, or `pid`, with each process.

۳- مفهوم file descriptor یک مقدار عددی کوچک است که توسط کرنل به هر فایل باز در سیستم های یونیکس اختصاص داده می شود در واقع می توان گفت این مفهوم همانند آیدی برای هر فایل باز است تا سیستم عامل بتواند فایل های باز را کنترل کند. همچنین مفهوم پایپ (که از نوع بافر است) همانند یک لوله بین دو فایل برای خواندن و نوشتن عمل می کند در واقع پایپ یک راه ارتباطی بین دو فایل است

> A *file descriptor* is a small integer representing a kernel-managed object that a process may read from or write to. A process may obtain a file descriptor by opening a file, directory, or device, or by creating a pipe, or by duplicating an existing descriptor. For simplicity we'll often refer to the object a file descriptor refers to as a "file"; the file descriptor interface abstracts away the differences between files, pipes, and devices, making them all look like streams of bytes.

> A *pipe* is a small kernel buffer exposed to processes as a pair of file descriptors, one for reading and one for writing. Writing data to one end of the pipe makes that data available for reading from the other end of the pipe. Pipes provide a way for processes to communicate.

۴- دستور fork یک کپی از فرآیند فعلی ایجاد می‌کند. یعنی یک پردازه جدید به وجود می آورد که دقیقا شبیه به پردازه ی اصلی است و دستور exec جایگزینی پردازه ی حال حاضر را با برنامه‌ای جدید جایگزین می کند. اگر این دو دستور ادغام می‌شدند، چند مشکل پیش می‌آمد: جدا بودن این دو دستور به برنامه‌نویس اجازه می‌دهد قبل از اجرای برنامه‌ی جدید، کارهای دیگری مانند تغییر مسیر ورودی و خروجی، بسته کردن فایل‌های غیرضروری را انجام دهد. اگر ادغام می‌شدند، این تنظیمات از دست می‌رفتند.همچنین می‌توانست باعث مصرف بی‌مورد حافظه و پردازش شود، زیرا به صورت یکجا باید تمام کارها انجام می شد.

A process may create a new process using the `fork` system call. `Fork` creates a new process, called the *child process*, with exactly the same memory contents as the calling process, called the *parent process*. `Fork` returns in both the parent and the child. In the parent, `fork` returns the child's pid; in the child, it returns zero. For example, consider the following program fragment:

The `exec` system call replaces the calling process's memory with a new memory image loaded from a file stored in the file system. The file must have a particular format, which specifies which part of the file holds instructions, which part is data, at which instruction to start, etc. xv6 uses the ELF format, which Chapter 2 discusses in more detail. When `exec` succeeds, it does not return to the calling program; instead, the instructions loaded from the file start executing at the entry point declared in the ELF header. `Exec` takes two arguments: the name of the file containing the executable and an array of string arguments. For example:

مقدمه ای درباره سیستم عامل و xv6

پاسخ ۱:

یک) مدیریت و تقسیم منابع سخت‌افزاری

دو) مدیریت نرم‌افزارهای کاربر به وسیله interface

سه) ایجاد ارتباط بین سخت‌افزار و نرم‌افزار

پاسخ ۲:

Basic Headers -> Define of data types, constants, and function declarations

Locks -> synchronization mechanisms, like spinlocks, safe access to shared resources during multiple processes.

Processes -> manage process activities in xv6: creating and scheduling, switching between processes, loading and executing programs.

System calls -> including the system call handler, implementing kernel functions for calls.

File Systems -> implements the file system layer in xv6, managing files, directories, and disk I/O operations

Pipes -> pipe mechanism for inter-process communication, allowing processes to share a common buffer so one can write data for another one to read

String Operations -> string related operations, such as copying and comparing two string

Low-level HW -> directly interacts to HW and manages operations in this level

BootLoader -> booting xv6, loads kernel into memory and starts its execution

Memory Layout -> how object files should be linked, defining kernels memory structure

پاسخ ۳:

خط زیر فایل برنامه‌ها را به یکدیگر لینک کرده و فایل نهایی هسته را می‌سازد.

$(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode entryother

پاسخ ۴:

متغیر UPROGS نشان‌دهنده برنامه‌های سطح کاربر است(user space program). و ULIB متدهای پرکاربرد برای برنامه‌های سطح کاربر هستند.(user libraries) شامل printf.o و ...

پاسخ ۵:

هنگام اجرای make qemu -n دو دیسک اصلی به qemu متصل می‌شوند: kernel.img که شامل کرنل کامپایل شده است و fs.img که شامل File System و برنامه‌های سطح کاربر است.

همانطور که گفته شد، این دو دیسک شامل سه خروجی اصلی فرایند بیلد هستند که شامل کرنل، file system و برنامه‌های سطح کاربر است که در fs قرار دارند. به کمک این دو دیسک، بوت xv6 و اجرای برنامه‌های کاربری انجام می‌شود.

پاسخ ۸:

در فرآیند build xv6، پس از کامپایل کرنل، خروجی به‌صورت یک فایل ELF تولید می‌شود. اما برای اینکه این فایل بتواند توسط بوت‌لودر و شبیه‌ساز QEMU به‌عنوان یک تصویر باینری (binary image) بارگذاری شود، نیاز است که قالب آن تغییر کند.

استفاده از objcopy خروجی فایل ELF را به یک فایل باینری ساده تبدیل می‌کند. این تبدیل باعث می‌شود که فایل خروجی را بتوانیم به qemu به عنوان کرنل قابل بوت بدهیم. همچنین با objcopy اطلاعات غیرضروری از فایل حذف می‌شود و حجم آن کاهش می‌یابد.

پاسخ ۱۳:

با قرار دادن کرنل در آدرس بالاتر از 1MB، از تداخل با بخش‌های دیگر حافظه که توسط بوت‌لودر یا سخت‌افزارهای دیگر اشغال شده‌اند جلوگیری می‌شود. این باعث می‌شود که بوت‌لودر، جداول وقفه و سایر بخش‌های مهم حافظه از بازنویسی محافظت شوند. این توضیح در شکل حافظه که در صورت آزمایش آمده هم قابل مشاهده است. با این کار کرنل بالاتر از bootblock نوشته شده و روی آن نوشته نمی‌شود.

پاسخ ۱۸:

SEG_USER فضای کاربر را از کرنل جدا می‌کند تا امنیت و پایداری سیستم حفظ شود. GDT سطح دسترسی را مشخص کرده و از دسترسی غیرمجاز به کرنل جلوگیری می‌کند. پردازنده در حالت کاربر فقط به SEG_USER

دسترسی دارد و هر تلاش برای دسترسی به کرنل باعث segmentation fault می‌شود. این تفکیک یک اصل مهم در طراحی سیستم‌عامل‌هاست.

پاسخ ۱۹:

```
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
```

sz: اندازه فرآیند در حافظه

pgdir: اشاره‌گر به جدول صفحات.

kstack: اشاره‌گر به پشته کرنل

state: وضعیت فعلی فرآیند

pid: شناسه یکتای فرآیند

parent: اشاره‌گر به فرایند parent

tf: اشاره‌گر به ترپ

context: اشاره‌گر به ساختار context شامل رجیسترهای فرآیند برای تغییر زمینه

chan: کانالی که فرآیند در حال خواب بر اساس آن منتظر یک رویداد است

killed: نشان‌دهنده این است که آیا فرآیند برای خاتمه دادن killed شده است یا خیر

ofile: آرایه‌ای از اشاره‌گرهای به فایل‌های باز که توسط فرآیند استفاده می‌شوند

cwd: دایرکتوری جاری

name: نام فرآیند برای دیباگ

در لینوکس ساختار مشابهی به نام task_struct وجود دارد. (اطلاعات مدیریتی مربوط به هر فرایند را نگه می‌دارد.)

پاسخ ۲۳:

در مجموع، در تابع main هجده تابع فراخوانی شده است که ۴ مورد بین تمامی پردازنده‌ها مشترک هستند، در حالی که 14 تابع دیگر مخصوص پردازنده‌ای هستند که سیستم‌عامل را بوت کرده است.

توابع مشترک بین تمامی هسته‌ها:

`switchkvm, seginit, lapicinit, mpmain`

توابع اختصاصی برای پردازنده‌ای که سیستم‌عامل را بوت کرده است:

`kinitl,kvmalloc(setupkvm),mpinit, picinit, ioapicinit, consoleinit, uartinit, pinit, tvinit, binit, fileinit, ideinit, startothers, kinit2, userinit`

بخش‌های مشترک: کرنل، مدیریت حافظه، جدول فرآیندها، سیستم فایل و I/O

بخش‌های اختصاصی: راه‌اندازی هر هسته، پشته کرنل، رجیسترها، تغییر زمینه

زمان‌بند در mpmain صدا زده می‌شود پس بین همه هسته‌ها مشترک است. هر پردازنده هم زمان‌بند مختص خودش را دارد.

اشکال زدایی

۱- با استفاده از دستور info breakpoint در ترمینال GDB می توانیم بریک پوینت های مشخص شده را مشاهده کنیم:

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000000011f8 in cat at cat.c:12
```

۲- برای پاک کردن یک بریک پوینت می توان از دستور:

del <breakpoint Num>

در ترمینال GDB استفاده کرد برای مثال برای پاک کردن بریک پوینت بالا:

```
(gdb) del 1
```

۳- دستور bt , استک مخصوص به برنامه ای که هم اکنون در حال اجراست را نمایش می دهد. استک تابع کنونی به همراه متغییر های آن به شکل زیر نمایش داده می شوند همچنین این دستور مخفف backtrace است:

```
(gdb) bt
#0  cat (fd=0) at cat.c:12
#1  0x00005555555552a2 in main (argc=1, argv=0x7fffffffdd58) at cat.c:30
```

۴- دستور print برای نمایش مقدار یک متغیر (مانند n، به عنوان آرگومان دستور وارد می شود) به کار می رود در حالی که دستور x برای نمایش دادن مقدار موجود در یک خانه ی حافظه (آدرس یا متغیر آن به عنوان آرگومان دستور وارد می شود) استفاده می شود:

```
(gdb) print n
$1 = 0
```

```
(gdb) x buf
0x555555558040 <buf>:    0x00000000
```

۵- دستور info registers وضعیت و مقدار و نام رجیستر ها را نمایش می دهد:

```
(gdb) info registers
rax            0x55555555527f      93824992236159
rbx            0x0                 0
rcx            0x555555557d98      93824992247192
rdx            0x7fffffffdd68      140737488346472
rsi            0x7fffffffdd58      140737488346456
rdi            0x0                 0
rbp            0x7fffffffdc10      0x7fffffffdc10
rsp            0x7fffffffdbf0      0x7fffffffdbf0
r8             0x7ffff7f97f10      140737353711376
r9             0x7ffff7fc9040      140737353912384
r10            0x7ffff7fc3908      140737353890056
r11            0x7ffff7fde660      140737353999968
r12            0x7fffffffdd58      140737488346456
r13            0x55555555527f      93824992236159
r14            0x555555557d98      93824992247192
r15            0x7ffff7ffd040      140737354125376
rip            0x5555555551f8      0x5555555551f8 <cat+15>
eflags         0x206               [ PF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
```

همچنین با دستور info locals می توان متغیر های محلی و مقدار آن ها را مشاهده کرد:

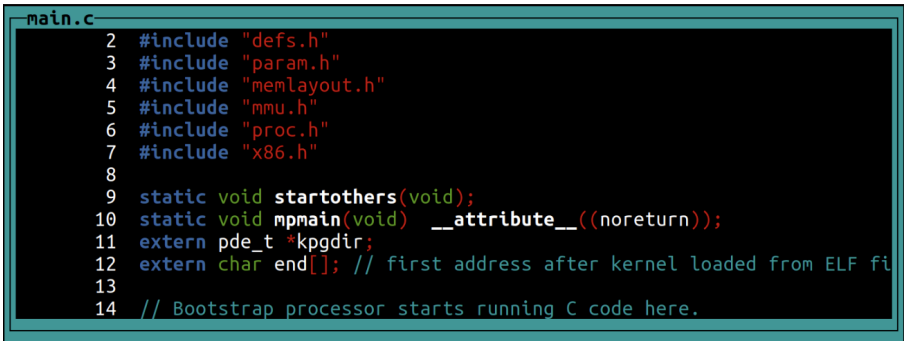
```
(gdb) info locals
n = 0
```

رجیستر edi که مخفف (Extended Destination Index) می باشد به عنوان رجیستر نگه دارنده ی آدرس مقصد برای عملیات های اجرایی بر روی رشته ها استفاده می شود. همچنین رجیستر esi که مخفف (Extended Source Index) می باشد به عنوان نگهدارنده ی آدرس مبدا برای عملیات های اجرایی بر روی رشته ها استفاده می شود.

۶- استراکت input مربوط به بافر کنسول سیستم عامل است که مقادیری که کاربر با استفاده از کیبورد در کنسول سیستم عامل وارد می کند در این بافر ذخیره می شود تا در ادامه بر روی آن ها پردازش صورت گیرد یا پاس داده شوند این استراکت داری چهار متغیر است ۱. یک آرایه ی از کاراکتر ها با سایز ۱۲۸ که مربوط به خود بافر است و مقادیر در آن جا ذخیره می شوند ۲. یک متغیر e که نشان دهنده ی آن است که در حال حاضر بر روی ایندکس کدام خانه ی آرایه هستیم (در واقع انگار cursor در آن خانه ی آرایه ی بافر قرار گرفته) که مخفف edit می باشد در این حالت هنوز کلید اینتر توسط کاربر زده نشده و به ازای فشرده شدن هر یک کلید یکی اضافه می شود ۳. متغیر w که باز هم یک مقدار عددی و نشان دهنده ی ایندکس یک خانه ی آرایه ی بافر است به محض این که کلید اینتر زده شود این مقدار با مقدار e برابر می شود و در واقع انتهای

رشته ی نوشته شده را نمایش می دهد و مخفف write است ۴. متغیر r که همانند دو متغیر قبل است با این تفاوت که نشان می دهد تا کدام ایندکس بافر پیش از زده شدن کلید اینتر خوانده شده در واقع ابتدای رشته ی کنونی را نشان می دهد و مخفف read است.

۷- با استفاده از دستور layout src می توان کد c مربوط به فایل در حال دیباگ را در محیط TUI دید همچنین با دستور layout acm می توان کد اسمبلی مربوط به فایل در حال دیباگ را در محیط TUI مشاهده کرد.



۸- با وارد کردن دستور های down, up می توان بین استک های توابع زنجیره ی فراخوانی جاری جا به جا شد کافی است یکی از این دستور ها در ترمینال gdb نوشته شود