OS 代码阅读报告

江灿 2019011325

清华大学 软件学院，北京 100084

【摘　要】 本次实验是

【关键词】 光栅衍射，波长，光栅常数，最小偏向角

代码报告基本要求

- 重要函数或语句的代码分析与注释
- 基本流程分析
- 实现概述
- 阅后心得
- 评分：知识的理解程度，主要技术分析程度，认真程度

Chapter 1 Operatings system interfaces

preface

The job of an operating system is to share a computer among multiple programs and to provide a more useful set of services than the hardware alone supports.And provides services to user programs through an interface

What is a good interface :

provides services to user programs through an interface;offer many powerful features to applications

some words

- xv6:mimicking Unix's internal design,the Os
- RISV - v : microprocessor (微处理器)
- QEMU : mechine simulator(under linux)
- knrnel:a special program that provides services to running programs
- program(called a process):has memory containing instructions, data, and a stack
- instructions:implement the program's computation
- data:variables on which the computation acts
- stack:organizes the program's procedure calls
- grep (global search regular expression(RE) and print out the line
- cat(concatenate)

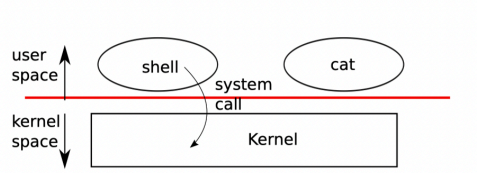

Figure 1.1: A kernel and two user processes.

图 1 A kernel and two user processes

shell:an ordinary program that reads commands from the user and executes them. The fact that the shell is a user program, and not part of the kernel

Processes and memory

System call	Description
int fork()	Create a process, return child's PID.
int exit(int status)	Terminate the current process; status reported to wait(). No return.
int wait(int *status)	Wait for a child to exit; exit status in *status; returns child PID.
int kill(int pid)	Terminate process PID. Returns 0, or -1 for error.
int getpid()	Return the current process's PID.
int sleep(int n)	Pause for n clock ticks.
int exec(char *file, char *argv[])	Load a file and execute it with arguments; only returns if error.
char *sbrk(int n)	Grow process's memory by n bytes. Returns start of new memory.
int open(char *file, int flags)	Open a file; flags indicate read/write; returns an fd (file descriptor).
int write(int fd, char *buf, int n)	Write n bytes from buf to file descriptor fd; returns n.
int read(int fd, char *buf, int n)	Read n bytes into buf; returns number read; or 0 if end of file.
int close(int fd)	Release open file fd.
int dup(int fd)	Return a new file descriptor referring to the same file as fd.
int pipe(int p[])	Create a pipe, put read/write file descriptors in p[0] and p[1].
int chdir(char *dir)	Change the current directory.
int mkdir(char *dir)	Create a new directory.
int mknod(char *file, int, int)	Create a device file.
int fstat(int fd, struct stat *st)	Place info about an open file into *st.
int stat(char *file, struct stat *st)	Place info about a named file into *st.
int link(char *file1, char *file2)	Create another name (file2) for the file file1.
int unlink(char *file)	Remove a file.

图 2 Xv6 system calls 0(no error)/-1(error)

PID: process identifier

$$int\ fork()$$

system call / give the new process exactly the same memory contents (both instructions and data) as the calling process

return original process(new process's PID);

new process(0)

The physical addresses in parent and child must be different. But C program accesses virtual addresses (if running on an OS that uses virtual memory). The child process gets an exact copy of parent process and virtual address don't change in child process.

```
    int pid = fork();
if(pid > 0){
  printf("parent: child = %d\n",pid);
  pid = wait((int *) 0);
  printf("child %d is done\n",pid);
}else if(pid == 0){
    printf("child : exiting \n");
    exit(0);
}
else{
    printf("fork error\n");
}
```

$$int\ exit(int\ status)$$

0:success 1:failure

$$int\ wait(int\ *status)$$

returns the PID of an exited (or killed) child of the current process

copies the exit status of the child to the address passed to wait no child:-1 pass 1 0 address : don't care about the exit status of a child

$$int\ exec(char\ *file,\ char\ *argv[])$$

replaces the calling program with an instance of the program

argv[0] conventionally the name of the program

$$main$$

```
144 int
145 main(void)
146 {
147   static char buf[100];
148   int fd;
149
150   // Ensure that three file descriptors are open.
151   while((fd = open("console", O_RDWR)) >= 0){
152     if(fd >= 3){
153       close(fd);
154       break;
155     }
156   }
157
158   // Read and run input commands.
159   while(getcmd(buf, sizeof(buf)) >= 0){
160     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
161       // Chdir must be called by the parent, not the child.
162       buf[strlen(buf)-1] = 0;  // chop \n
163       if(chdir(buf+3) < 0)
164         fprintf(2, "cannot cd %s\n", buf+3);
165       continue;
166     }
167     if(fork1() == 0)//在child 运行
168       runcmd(parsecmd(buf));
169     wait(0);
170   }
171   exit(0);
172 }
```

```
8 runcmd(struct cmd *cmd)// while parent run runcmd
78      exec(ecmd->argv[0], ecmd->argv);//child run exec
```

A process that needs more memory at run-time (perhaps for malloc) can call sbrk(n) to grow its data memory by n bytes; sbrk returns the location of the new memory.

I/O and File descriptors

```
150   // Ensure that three file descriptors are open.
151   while((fd = open("console", O_RDWR)) >= 0){
152     if(fd >= 3){//if is 3, break
153       close(fd);
154       break;
155     }
156   }
```

```
% 代码段

```