Paul Kummer

Dr. Hanku Lee

Operating Systems CSIS 430-01

10/07/2021

Lab 05: XV6 Copy & CPS

Purpose of Lab

The purpose of this lab is to show the students the difference between system calls and a regular command. The System calls require editing all the system files to update them with a new capability, whereas a command is simple and only requiring the file of the command and editing the makefile. This lab demonstrates specifically how to make a copy command and a current program status system call.

User Command

Copy Command

The copy command allows the user to make a duplicate of a file.

cp.c

The cp commands executes the cp.c code to copy a source file as a destination file. The copy command uses command line arguments. The arguments that are used are the source file to be copied and what and the destination file should be. If there are not enough arguments, an error message will be displayed. Otherwise, the first file will have its characters read and written to the destination file one by one.

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h" //file control options O_RDONLY

char buf[512];

int main(int argc, char *argv[])
{
    int fd0, fd1, n;

    if(argc <= 2)
    {
        printf(1, "Usage: There must be two arguments!\n");
        exit();
    }

    if((fd0 = open(argv[1], O_RDONLY)) < 0)
    {
        printf(1, "cp: Cannot Open %s\n", argv[1]);
        exit();
    }

    if((fd1 = open(argv[2], O_CREATE|O_RDWR)) < 0)
    {
        printf(1, "cp: Cannot Open %s\n", argv[2]);
        exit();
    }

    while((n = read(fd0, buf, sizeof(buf))) > 0)
    {
        write(fd1, buf, n);
    }

    close(fd0);
    close(fd1);

    return 0;
}
```

Makefile

To add the new command to the XV6, the Makefile must be modified to compile the new file. In the Makefile, the user command must be added to “UPROGS” and the file name must be added to “EXTRA”. After this is done, the OS will compile with the new capability for the user.

```
EXTRA=\
	mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
	ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c hello.c cp.c\
	printf.c umalloc.c\
	README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
	.gdbinit.tmpl gdbutil\
```

```
UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_usertests\
	_wc\
	_zombie\
	_hello\
	_cp\
```

System Call

Current Processes Status

The Current Processes Status (CPS) will show the status of all the current processes. The processes will typically be in three different states of RUNNING, RUNNABLE, or SLEEPING.

ps.c

The ps.c file contains the user command that will result in the system call to cps.

C: > Users > pakum > AppData > Local > Temp > scp0

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

int main(int args, char *argv[])
{
    cps();
    exit();
}
```

syscall.h

This file contains the declarations for the system call to cps and gives it the number 23.

C: > Users > pakum > AppData > Local >

```c
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_hello  22
#define SYS_cps    22
```

This has been changed to 23 instead of being 22 *syscall.c*

This file contains the available system calls, and states that the sys_cps system call will be defined externally.

```c
extern int sys_chdir(void);
extern int sys_close(void);
extern int sys_dup(void);
extern int sys_exec(void);
extern int sys_exit(void);
extern int sys_fork(void);
extern int sys_fstat(void);
extern int sys_getpid(void);
extern int sys_kill(void);
extern int sys_link(void);
extern int sys_mkdir(void);
extern int sys_mknod(void);
extern int sys_open(void);
extern int sys_pipe(void);
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_hello(void);
extern int sys_cps(void);


static int (*syscalls[])(void) =
{
  [SYS_fork]    sys_fork,
  [SYS_exit]    sys_exit,
  [SYS_wait]    sys_wait,
  [SYS_pipe]    sys_pipe,
  [SYS_read]    sys_read,
  [SYS_kill]    sys_kill,
  [SYS_exec]    sys_exec,
  [SYS_fstat]   sys_fstat,
  [SYS_chdir]   sys_chdir,
  [SYS_dup]     sys_dup,
  [SYS_getpid]  sys_getpid,
  [SYS_sbrk]    sys_sbrk,
  [SYS_sleep]   sys_sleep,
  [SYS_uptime]  sys_uptime,
  [SYS_open]    sys_open,
  [SYS_write]   sys_write,
  [SYS_mknod]   sys_mknod,
  [SYS_unlink]  sys_unlink,
  [SYS_link]    sys_link,
  [SYS_mkdir]   sys_mkdir,
  [SYS_close]   sys_close,
  [SYS_hello]   sys_hello,
  [SYS_cps]     sys_cps,
};
```

defs.h

The cps function is declared, but not yet defined. The system call sys_cps will utilize this cps

```
//PAGEBREAK: 16
// proc.c
int             cpuid(void);
void            exit(void);
int             fork(void);
int             growproc(int);
int             kill(int);
struct cpu*     mycpu(void);
struct proc*    myproc();
void            pinit(void);
void            procdump(void);
void            scheduler(void) __attribute__((noreturn));
void            sched(void);
void            setproc(struct proc*);
void            sleep(void*, struct spinlock*);
void            userinit(void);
int             wait(void);
void            wakeup(void*);
void            yield(void);
int             cps(void);
```

user.h

The cps function is also declared to be callable by the user.

```
C: > Users > pakum > AppData > Local > Temp > scp43770 > hor
struct stat;
struct rtcdate;

// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int hello(void);
int cps(void);
```

proc.c

The actual code to show programs status is defined here as a process. It will allow interrupts so other processes can run while this process is executing. To prevent any of the data from being modified on the process table as it is being read; A lock is added on the ptable and all the process in the table are iterated over, and their statuses are printed. After they are iterated over, the process table is unlocked, allowing modification.

```c
// Current Process Status
int cps()
{
  struct proc *p;

  // Enable interrupts on this processor
  sti();

  // Loop over process table looking for process to run.
  acquire(&ptable.lock);

  // Critical Region
  cprintf("name \t pid \t state \t \t");

  // Print the status of each process
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if(p->state == SLEEPING)
    {
      cprintf("%s \t %d \t SLEEPING \t \t", p->name, p->id);
    }
    else if(p->state == RUNNING)
    {
      cprintf("%s \t %d \t RUNNING \t \t", p->name, p->id);
    }
    else if(p->state == RUNNABLE)
    {
      cprintf("%s \t %d \t RUNNABLE \t \t", p->name, p->id);
    }
  }

  release(&ptable.lock);

  return 23;
}
```

sysproc.c

This file will define the function sys_cps, which is used as the system call. The function simply calls the cps function that is defined in the proc.c. The screenshot has been modified to what is in the textbox.

```c
//cps
int cps(void)
{
    return cps();
}
```

```c
int sys_cps(void)
{
        return cps();
}
```

usys.S

Since the cps is a system call, the process must be managed at the kernel level. This is why the SYSCALL(cps) is added.

C: > Users > pakum > AppData > Local > Temp > scp

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(hello)
SYSCALL(cps)
```

Makefile

This file is used for compiling the OS. The new C files have to be added in order for them to compile correctly. The ps user command must be added to UPROGS and EXTRA.

```
UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_usertests\
	_wc\
	_zombie\
	_hello\
	_cp\
	_ps\
```

```
EXTRA=\
	mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
	ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c hello.c cp.c ps.c\
	printf.c umalloc.c\
	README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
	.gdbinit.tmpl gdbutil\
```

Execution

Commands

cp

The command cp is strictly a user command, and the kernel has no knowledge of its existence.

```
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ls
.              1 1 512
..             1 1 512
README         2 2 2327
cat            2 3 13688
echo           2 4 12692
forktest       2 5 8128
grep           2 6 15564
init           2 7 13280
kill           2 8 12748
ln             2 9 12648
ls             2 10 14832
mkdir          2 11 12828
rm             2 12 12808
sh             2 13 23296
stressfs       2 14 13476
usertests      2 15 56408
wc             2 16 14224
zombie         2 17 12472
hello          2 18 12420
cp             2 19 13296
console        3 20 0
$ cp README README.txt
pid 4 cp: trap 14 err 5 on cpu 1 eip 0xffffffff addr 0xffffffff--kill proc
$ ls
.              1 1 512
..             1 1 512
README         2 2 2327
cat            2 3 13688
echo           2 4 12692
forktest       2 5 8128
grep           2 6 15564
init           2 7 13280
kill           2 8 12748
ln             2 9 12648
ls             2 10 14832
mkdir          2 11 12828
rm             2 12 12808
sh             2 13 23296
stressfs       2 14 13476
usertests      2 15 56408
wc             2 16 14224
zombie         2 17 12472
hello          2 18 12420
cp             2 19 13296
console        3 20 0
README.txt     2 21 2327
```

ps

Despite being a user called command. The ps command calls the system command of sys_cps that eventually will call the cps function in the proc.c

```
init: starting sh
$ ps
name     pid     state
init     1       SLEEPING
sh       2       SLEEPING
ps       3       RUNNING
$ QEMU: Terminated
```

Conclusion

This lab helped show me the difference between a user called process and a system called process. With system calls, many of the system files must be modified to allow the kernel to know about the command and how to handle it. The user commands are simple and easy to implement. Also, I learned more about what how the system files interact with each other. I inadvertently learned more about the zombie command too, which I initially thought was like hello world, but it is entirely different and is used as a demonstration of a process. I also gained some experience in debugging the files that had typos that were hard to detect because they were very close to what they should have been.