Thomas Asuncion
OS Concepts
Tutorial 1

How to Create User Progs & Syscalls in xv6

Objective: Creating a custom user program "grc" that returns the total number of times the *read()* syscall was called by using a custom *getreadcount()* syscall in xv6.

Creating a user program: Firstly, you will need to create a user program in your xv6 operating system. Assuming you have untarred your xv6.tar.gz file with:

```
tar – zxvf xv6. tar. gz
```

You will need to change directory into your /xv6 directory:

```
cd xv6
```

Once you have done that you will need to change directory in xv6's /user directory, this is where all the user programs will be stored, additionally you can list all the contents in that directory if you want to see all of the available user progs:

```
cd user
```

```
ls
```

Note: that *ls* is in fact a user program, it has its own code that you can view from *ls.c*. Once you have done that you will need to create a *.c* user program, in our case it will be *grc.c* which will print out the total number of times the *read()* syscall was invoked:

```
emacs grc. c
```

```
// The following is grc.c:

#include "types.h"
#include "stat.h"
#include "user.h"

int main(void)
{
      printf(1, "The total number of times 'read()' syscall has been
      invoked is: %d.\n", getreadcount());

      exit();
}
```

Now that we have created our user program we need to add it to your Makefile so that it can be detected. Look for *makefile.mk* and add the following adjustments:

```
emacs makefile. mk
```

```
// The following is makefile.mk:

# user programs
USER_PROGS := \
        cat\
        echo\
        forktest\
        grc\
        grep\
        init\
        kill\
        ln\
        ls\
        mkdir\
        rm\
        sh\
        stressfs\
        tester\
        usertests\
        wc\
        zombie

USER_PROGS := $(addprefix user/, $(USER_PROGS))

# user library files
USER_LIBS := \
        ulib.o\
        usys.o\
        printf.o\
        umalloc.o

USER_LIBS := $(addprefix user/, $(USER_LIBS))

USER_OBJECTS = $(USER_PROGS:%=%.o) $(USER_LIBS)

USER_DEPS := $(USER_OBJECTS:.o=.d)

USER_CLEAN := user/bin $(USER_PROGS) $(USER_OBJECTS) $(USER_DEPS)

--More--(22%) ...
```

All we should've done to the makefile.mk is add one line of code under our USER_PROGS list: *grc*\. It is because of that I am only showing a little bit of ***makefile.mk*** and not the entire file as it is a lot of text. Please note: if you see --More--(n%) … this means that only SOME of the file is shown.

Once we've done that, we have successfully added a user program. Now we need to add our *getreadcount()* syscall since our user program uses it. If our user program didn't use any syscalls we could *make qemu* (build our OS) and run it!

Creating our syscall: In order to create our syscall we need to add our syscall to some xv6 files so that it gets recognized during the building process. Assuming you're out of the xv6 /user directory and just in /xv6, we need to change directory into /include

```
cd include
```

List out your contents and look for ***syscall.h***. Once you've found it, open it up in a text editor such as emacs and make the following adjustments:

```
emacs syscall.h
```

```
// The following is syscall.h:

#ifndef _SYSCALL_H_
#define _SYSCALL_H_

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

#endif // _SYSCALL_H_
```

Once again, we've only added one line of code, that is our *getreadcount syscall #22*. From there we need to go back to our /xv6 main directory and change into xv6's kernel directory:

```
cd ..
```

```
cd kernel
```

Look for the file ***syscall.c*** and once you've found it make the following changes to it:

emacs syscall. c

```
// The following is syscall.c:

// Declare for getreadcount
extern int
sys_getreadcount(void);

// array of function pointers to handlers for all the syscalls
static int (*syscalls[])(void) = {
[SYS_chdir]   sys_chdir,
[SYS_close]   sys_close,
[SYS_dup]     sys_dup,
[SYS_exec]    sys_exec,
[SYS_exit]    sys_exit,
[SYS_fork]    sys_fork,
[SYS_fstat]   sys_fstat,
[SYS_getpid]  sys_getpid,
[SYS_kill]    sys_kill,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_mknod]   sys_mknod,
[SYS_open]    sys_open,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_unlink]  sys_unlink,
[SYS_wait]    sys_wait,
[SYS_write]   sys_write,
[SYS_uptime]  sys_uptime,
[SYS_getreadcount] sys_getreadcount,
};

// Called on a syscall trap. Checks that the syscall number (passed
via eax)
// is valid and then calls the appropriate handler for the syscall.
void
syscall(void)
{
  int num;

--More--(92%) ...
```

Due to the large amount of text, I've only displayed a portion of ***syscall.c***.

Notice that there should only be 2 things done to this file: the *extern int* syscall declaration for getreadcount and adding it to the array of function pointers to handlers for all syscall.

From there we need to implement our syscall function. That is, we need to code up how our *getreadcount()* syscall will return the total number of times *read()* gets called.

In the same directory look for the file ***sysfile.c***. That file contains the implementation for the *read()* syscall and more importantly that's where our implementation for *getreadcount()* should belong. Open it up and make the following adjustments:

emacs sysfile. c

```
// The following is sysfile.c:

// Counter for read count
int readcounter = 0;

int
sys_read(void)
{
  struct file *f;
  int n;
  char *p;
  readcounter++;

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

int
sys_getreadcount(void)
{
  return readcounter;
}

int
sys_write(void)
{
  struct file *f;
  int n;
  char *p;

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

--More--(20%) ...
```

Notice that we've created a variable *int readcounter* and increment it every time *read()* is called.

Additionally, notice that our implementation for *getreadcount()* simply returns the readcounter that we increment in *read()*. This is how we will measure how many times *read()* is called. We're almost done!

Lastly, we need to go back into our /user directory and adjust the ***usys.S*** file and ***user.h*** file. Let's change back into the /user directory.

```
cd ..
```

```
cd user
```

Edit the ***usys.S*** file and ***user.h*** file like so:

```
emacs usys. S
```

```
// The following is usys.S:

#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(getreadcount)
```

```
// The following is user.h:

#ifndef _USER_H_
#define _USER_H_

struct stat;

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

// user library functions (ulib.c)
int stat(char*, struct stat*);
char* strcpy(char*, char*);
void *memmove(void*, void*, int);
char* strchr(const char*, char c);
int strcmp(const char*, const char*);
void printf(int, char*, ...);
char* gets(char*, int max);
uint strlen(char*);
void* memset(void*, int, uint);
void* malloc(uint);
void free(void*);

--More--(95%) ...
```

Once we've added our system call in those files, this will make our system call visible to the user program. Now we can make and run!

To make our xv6 OS run the following command:

```
make qemu
```

Once we've ran *make qemu* you will now be in our xv6 shell. We can run commands or user programs such as *ls, grep,* etc. You will notice you are very limited to what you can run. More importantly try running your custom user program that you just made, *grc*!

```
grc
```

Since we haven't called *read()* at all we will get a count of 0. Once you've called *read()* you will notice that the readcounter value will increase.

Please note: It is important that if you want to add more user programs to your xv6 OS (maybe you want to add a user program that calls *read()*) you need to adjust the ***makefile.mk*** accordingly!