CSE 460 Operating System (Winter 2020)

Homework #2 (Due on March 6th, 2020)

All assignments are to be submitted to Blackboard. Please note that the due time of each homework is at 11:55 pm (Blackboard time) on the due date. Please make sure to "submit" after uploading your files. Please do not attach unrelated files. You will not be able to change your files after clicking submit.

This team project are to be done in groups. Groups will be assigned based on mutual student preferences and then random assignment to groups. The group size should be between 4 to 6. A group will submit one project report. All team members should make at least five commits to the team project.

You are recommended to use Github to store your project repository. This repository will be accessible to all members of your team and all team members are expected to commit and push changes contributions to the repository equally.

1. The date() System Call

The first task is to add a new system call to xv6. Your new system call will get the current UTC time and return it to the invoking user program. You must use the function, `cmostime()`, defined in `lapic.c`, to read the real-time clock. The supplied include file `date.h` contains the definition of the `rtcdate` data structure, a pointer to which you will provide as an argument to `cmostime()`.

Adding a New System Call

Adding a new system call to xv6, while straightforward, requires that several files be modified. The following files will need to be modified in order to add the `date()` system call.

a. `syscall.h` contains the mappings of system call name to system call number. This is a critical part of adding a system call as system calls are invoked within the operating system by number, not name. Follow the existing pattern for adding new system call numbers.

b. `defs.h` is where function prototypes for kernel-wide function calls, that are not in `sysproc.c` or `sysfile.c`, are defined. This file is not modified in this task but will be modified in subsequent tasks.

c. `user.h` contains the function prototypes for the xv6 system calls and library functions. This file is required to compile user programs that invoke these routines. The function prototype is

```
int date(struct rtcdate*);
```

d. `sysproc.c` is where you will implement `sys_date()`. The implementation of your date() system call is very straightforward. All you will need to do is add the new routine to `sysproc.c`; get the argument off the stack; call the `cmostime()` routine correctly; and return a code indicating SUCCESS or FAILURE, which will most likely be 0 because this routine cannot actually fail. Note that since you passed into the kernel a pointer to the `rtcdate` structure, any changes made to the structure within the kernel will be reflected in user space. The implementation is provided for you

```
int
sys_date (void)
{
   struct rtcdate *d;
   if(argptr(0, (void*)&d, sizeof(struct rtcdate)) < 0)
         return -1;
   cmostime(d);
   return 0;
}
```

See the xv6 book for how to properly use the `argptr()` routine. Be sure to read and understand the comments for the `argptr()` routine in `syscall.c` as this will become critical later.

e. `usys.S` contains the list of system calls made available (exported) by the kernel. Add any new system calls at the end of this list.

f. `syscall.c` is where the system call entry point will be defined. There are several steps to defining the entry point in this file.

1. The entry point will point to the implementation. The implementation for our system call will be in another file. You indicate that with the "extern" keyword.

```
extern int sys_date(void);
```

which you will add to the end of the list of similar extern declarations. You must declare the routine as taking a void argument!

2. Add to the syscall function definition. The function dispatch table

```
int (*syscalls[])(void) = {
```

declares the mapping from symbol name (as used by usys.S and declared in syscall.h) to the function name, which is the name of the function that will be called for that symbol.

Add `[SYS_date] sys_date,` to the end of this structure.

The routine `syscall()`, at the bottom of `syscall.c`, invokes the correct system call based on these definitions. The data structure `syscalls[]` is an array of function pointers which is often called a function dispatch table.

The date Command

We cannot directly test a system call and instead have to write a program that we can invoke as a command at the shell prompt. An example program, `date.c` has been provided.

```
# include "types.h"
# include "user.h"
# include "date.h"

int
main (int argc, char *argv[])
{
      struct rtcdate r;
      if (date(&r)) {
            printf(2, "date failed\n");
            exit();
      }

      printf(1, "%d/%d/%d %d:%d:%d\n", r.day, r.month, r.year, r.hour,
r.minute, r.second);

      exit();
}
```

The `printf()` routine takes a file descriptor as its first argument: 1 is standard output (stdout) and 2 is error output (stderr).

The function `main()` must terminate with a call to `exit()` and not `return()` or simply fall off the end of the routine. This is a very common source of compilation errors.

2. Ctrl-P

The xv6 kernel supports a special control sequence, Ctrl-P, which displays process information on the console. It is intended as a debugging tool and you will expand the information reported as you add new features to the xv6 process structure. Note that the routine `procdump()`, at the end of `proc.c`, implements the bulk of the Ctrl-P functionality. You will

modify `struct proc` in `proc.h` to include a new field, `uint start_ticks`. You will modify the routine `allocproc()` in `proc.c` so that this field is initialized to the value for the existing global kernel variable `ticks`. This allows the process to "know" when it was created. This will allow us to later calculate how long the process has been active. Put the initialization code right before the return at the end of the routine. Our version of xv6 does not require a lock around accesses to the `ticks` global variable.

See `procdump()` to understand the context, and for an example of how to handle the existing fields. Note that this routine handles printing the program counters (more below) at the end of each row for you. You will then modify the `procdump()` stub routine in `proc.c` to print out:

1. The amount of time that has passed since the process was allocated (i.e. how long it has been active, or the elapsed time). This change is fairly straightforward, since each "tick" is a millisecond (10^{-3} second. `ticks` is incremented approximately every millisecond by the timer interrupt, see `trap()` in `trap.c` if you are curious about this). Report the elapsed time in seconds, accurate to the millisecond. The longest running process in xv6 will always be the `init` process, but the first shell will be close behind.
2. You will also modify the routine to print the size of the process. The process size is already part of the process structure, so you are just printing it out, not calculating it.

The first line of output is a set of labels for each column, see `procdump()` for the code that prints the headers. You will be adding to the Ctrl-P output in later tasks and the header will help with interpreting the output. The last set of information printed from `procdump()` for each row contains the program counters as returned by the routine `getcallerpcs()`. The header for this last section is labeled "PCs".

Example Output

PID	Name	Elapsed	State	Size	PCs
1	init	1.543	sleep	12288	80104bf3 80104968 80106560 80105767 801068f0
2	sh	1.499	sleep	16384	80104bf3 80100a37 80101f68 80101265 8010591e

3. UIDs and GIDs and PPIDs

At this point xv6 has no concept of users or groups. You will begin to add this feature to xv6 by adding a `uid` and `gid` field to the process structure, where you will track process ownership.

These will be of type `unsigned int` since negative UIDs and GIDs make no sense in this context.

Note that when these values are passed into the kernel, they will be taken off the stack as `ints`. There is no issue with this as you will convert them to `unsigned ints` immediately. It is, however, critical, that the function prototypes in `user.h` declare values as `unsigned` as you will see below.

The `ppid` is the "parent process identifier" or parent PID. The `proc` structure does not need a ppid field as the parent can, and should, be determined on-the-fly. Look carefully at the existing `proc` structure in `proc.h` and add the following code in structure.

```
uint uid;
uint gid;
```

Note that the `init` process is a special case as it has no parent. Your code must account for any process whose parent pointer is NULL. For any such pointer, you will display the PPID to be the same as the process PID. Do not modify a parent pointer that is set to NULL; leave it that way as it becomes important in a later task.

Add the following system calls:

```
uint getuid (void) // UID of the current process
uint getgid (void) // GID of the current process
uint getppid ( void) // process ID of the parent process
int setuid (uint) // set UID
int setgid (uint) // set GID
```

Your kernel code cannot assume that arguments passed into the kernel are valid and so your kernel code must check the values for the correct range. The `uid` and `gid` fields in the process structure may only take on values 0 <= value <= 32767. You are required to provide tests that show this bound being enforced by the kernel-side implementation of the system calls.

The following code is a starting point for writing a test program that demonstrates the correct functioning of your new system calls. This example is missing several important tests and fails to check return codes.

```
int
main(void)
{
    uint uid, gid, ppid;
```

```
    uid = getuid();
    printf(2, "Current UID is: %d\n", uid);
    printf(2, "Setting UID to 100\n");
    setuid(100);
    uid = getuid();
    printf(2, "Current UID is: %d\n", uid);

    gid = getgid();
    printf(2, "Current GID is: %d\n", gid);
    printf(2, "Setting GID to 100\n");
    setgid(100);
    gid = getgid();
    printf(2, "Current GID is: %d\n", gid);

    ppid = getppid();
    printf(2, "My parent process is: %d\n", ppid);
    printf(2, "Done!\n");

    exit();
}
```

Other Necessary Modifications

You have modified the process structure to include the `uid` and `gid` for the process, but that isn't all the work necessary to support these new features. The `fork()` system call allocates a new process structure and copies all the information from the original process structure to the new one, with the exception of the `pid`. But you modified the process structure! You will need to find the code for the `fork()` system call in `proc.c` and make sure to copy the `uid` and `gid` of the current process to the new child process.

```
np->uid = curproc->uid;
np->gid = curproc->gid;
```

Not all processes are created with `fork()`. The first process, which eventually becomes the `init` process, is created piece-by-piece at boot time. The routine `userinit()` in the file `proc.c` is where this initialization takes place. You should also be able to set the `uid` and `gid` of the currently executing shell with appropriate built-in commands.

4. The “ps” command

Xv6 does not have the `ps` command like Linux, we have added `ps` command in the lab exercises. This command is used to find out information regarding active processes in the system. We define “active" here to be a process in the RUNNABLE, SLEEPING, RUNNING, or

ZOMBIE state. Processes in the UNUSED or EMBRYO states are not considered active. When write your `ps` program, you add system calls, such as `cps()` and `nps()`.

You will find `struct proc`, "the process structure", in the file `proc.h`. When xv6 is running, there is an array of `proc structs` in the data structure named `ptable` in `proc.c`. All information for each process is in a `struct proc` in the `proc[]` array.

The `ptable` also contains a field lock, that you use when you need to access the other contents of the `ptable`, such as the `proc[]` array, in an atomic transaction. Whenever you need to read or modify multiple values (words of memory) in the `ptable` indivisibly, you must follow the pattern below. This locking discipline ensures that these accesses happen in an atomic transaction, i.e. all at once with respect to other atomic transactions that use the `ptable` lock:

1. Acquire the ptable lock
2. Access the contents of the ptable
3. Release the ptable lock

This pattern assumes that you do not already hold the ptable lock, and that you've accessed everything you intend to indivisibly access before releasing the lock.

Modify your `ps` command to print the following information for each active process:

1. process id (as decimal integer)
2. name (as string)
3. process uid (as decimal integer)
4. process gid (as decimal integer)
5. parent process id (as decimal integer)
6. process elapsed time (as a floating point number)
7. process total CPU time (as a floating point number)
8. state (as string)
9. size (as decimal integer)

You'll print one line for each process, with a header indicating the contents for each column. You'll have to load multiple values to get all of these fields for each process. If we don't ensure that these values are accessed in an atomic transaction, we could observe a state of the `ptable` that never existed at any one point in time, because some values could be updated by another CPU while your CPU is in the process of loading them.

Note that the xv6 `printf` routine does not support floating-point numbers. This is fine since, for the values we will be calculating, the integer portion before the decimal point can be calculated with integer division and the integer portion after the decimal can be calculated using modulo arithmetic.

The system call that you use is called `cps()`,the `ptable` data structure is statically declared in `proc.c`. This means that even though you'll implement `sys_getprocs` in `sysproc.c` as usual, it will need to call a helper in `proc.c` in order to actually access the `ptable`.

Modifying the Console

The Ctrl-P command should have the same headings as the `ps` command with the addition of the existing output for PC (program counter) information. Here is an example output. Note that because of the resolution of the ticks variable that the various times are represented to three digits past the decimal. This will require some special checks in your updated `procdump()` routine!

$ Ctrl-P
PID Name UID GID PPID Elapsed CPU State Size PCs
1 init 0 0 1 1.249 0.033 sleep 12288 801039c6 80103adb 80104fd5
2 sh 0 0 1 1.213 0.012 sleep 16384 801039c6 801002c4 80101814
$ ps
PID Name UID GID PPID Elapsed CPU State Size
1 init 0 0 1 2.549 0.033 sleep 12288
2 sh 0 0 1 2.513 0.018 sleep 16384
3 ps 0 0 2 0.022 0.011 run 49152

5. File System Protection

The task requires you to implement a completely new abstraction in xv6: file system protection. This is a very large area, so you will focus on a small subset of protection that will nevertheless give you insights into how to implement additional protection and security concepts. You will implement file system protection, user programs for manipulating those protections, and add protection checking to the `exec()` system call.

New System Calls

You are required to implement three new system calls.

1. `int chmod(char *pathname, int mode);`

The `chmod`() system call sets the mode, or permission, bits for the target specified by pathname. The return value is '0' on success and '-1' on failure. See the description for the `chmod` command below for details.

2. `int chown(char *pathname, int owner);`

The `chown`() system call sets the user UID for the target specified by pathname. The return value is '0' on success and '-1' on failure. See the description for the `chown` command below for details.

3. `int chgrp(char *pathname, int group);`

The `chgrp`() system call sets the group GID for the target specified by pathname. The return value is '0' on success and '-1' on failure. See the description for the `chgrp` command below for details.

New Commands

Three new commands will allow the user to set the UID, GID, or mode for a file or directory. In order to simplify the testing of the new features in xv6, set the permissions and ownership for these commands as: owner = 0; group = 0; and mode = 0755.

1. `chown`: sets the owner (UID) for a file or directory.

Usage: chown OWNER TARGET

where OWNER is the numeric UID to set as the owner of TARGET and TARGET is the name of a file or directory.

2. `chgrp`: sets the group (GID) for a file or directory.

Usage: chgrp GROUP TARGET

where GROUP is the numeric GID to set as the group of TARGET and TARGET is the name of a file or directory.

3. `chmod`: sets the mode bits (permissions) for a file (or directory).

Usage: chmod MODE TARGET

where MODE is a string of octal values specifying the mode bits to set for TARGET and TARGET is the name of a file or directory.

The numeric MODE is four octal digits (0-7), derived by adding up the bits with values 4, 2, and 1. No digits may be omitted, and any digit can be the value 0. The first digit selects the set UID (1) attribute. The second digit selects permissions for the user who owns the file: read (4), write (2), and execute (1); the third selects permissions for other users in the file's group; and the fourth for other users not in the file's group, with the same values. It is permissible for one of user / group / other to be 0, which indicates no access to the file, e.g., chmod 0000 TARGET is valid.

Mode bits are evaluated in this order:

(a) User. Check against the user permissions. If the UID of the invoking process and the UID of the file are the same, use these permissions.

(b) Group: Check against the group permission. If the GID of the invoking process and the GID of the file are the same, use these permissions.

(c) Other: If neither the user or group permissions are used, apply these permissions.

(d) Setuid: If the permissions allow the invoking process to execute the file, set the UID of the process executing the file to be the same as the UID for the file.

chmod Examples

1. chmod 0755 File

File is not setuid, user has all permissions, group and other have read and execute permission.

2. chmod 1550 File

File is setuid, when the le is executed the process takes on the UID of File. Only the user and group may read or execute File. Others cannot access File.

3. chmod 0055 File

File is not setuid and the user may not access the le. File may be read and executed by group or others.

Modied Commands

1. `ls`: The `ls` command must be modified to display the new fields. A new routine, `print_mode` is provided in the file `print_mode.c` to handle the formatting. Since the output is getting crowded, you are to provide a header that labels each column in the output. The source code file `print_mode.c` can be included directly into the `ls.c` source code file.

mode. The first column will display the mode bits for the file/directory/device. The first character indicates if the item is a regular file ('-'), directory ('d'), or device le ('c'). If the file is setuid, then the 'x' in the user permissions will be displayed as 'S'.

name. Name of the file or directory.

uid. User identifier (owner) of the file or directory.

gid. Group identifier (owning group) of the file or directory.

inode. Inode number of the file or directory.

size. Size in bytes of the file or directory.

Sample Output

$ ls

mode name uid gid inode size

drwxr-xr-x . 0 0 1 512

drwxr-xr-x . . 0 0 1 512

-rwxr-xr-x README 0 0 2 1973

-rwxr-xr-x cat 0 0 3 14884

-rwxr-xr-x echo 0 0 4 13849

2. `mkfs`. File `mkfs.c`. This program is used to create the xv6 file system. It is used when the file system is built, which must be done before you can boot the kernel. It is invoked from the `Makefile`.

This program will set the default ownership (UID and GID) and default permissions (mode bits) for items in the file system.

Modied System Calls

1. exec(). File `exec.c`. The `exec()` system call will require two changes. The first change is that the file should not be read unless the process has execute permission for the file. The second change is when the new image is committed; if the file is setuid then the new uid will need to be set.

2. fstat. Deep into the implementation of the `fstat()` system call is the code where information from the inode is copied into the buffer provided to the `fstat()` system call. You will need to ensure that the new information is copied as well.

Deliverable:

The project report should include the following information:

1. Group member information (name, coyote id, responsibility of each member)
2. Table of content
3. The code added or modified when implementing each task
4. The screenshot of result of each task, and a brief description of the result.

Grading Rubric:

Homework 2	
1. Teamwork	30
2. Project	
a. date system call and command	10
b. Ctrl+P	10
c. uid, gid, ppid	10
d. ps command and Ctrl+P	10
e. file protecion	20
f. report	10
Total:	**100**