PROJECT 3: SYSTEM CALLS PT. 3

Ricardo Valencia

PORTLAND STATE UNIVERSITY - CS333: Intro to Operating Systems

Project Description

This project focuses on using locks for kernel-level data structures, implementing additional system calls that pass data structures in and out of the kernel, implementing a new user-level command that displays process states, and modifying the xv6 console to display additional process information.

Project Deliverables

UID, GID, and PPID System Calls

- syscall.h
- user.h
- proc.h
    - Set constant values for gid and uid, which were 10 and 20 respectively.
- usys.S
- sysproc.c
- syscall.c
- testuidgid.c

User-Level ps Command

- syscall.h
- defs.h
- uproc.h
- user.h
- usys.S
- sysproc.c
- syscall.c
- proc.c
    - Made the assumption that if if the process had no parent value, then the ppid for that process would just be the pid when it was being copied into the uproc table.
- ps.c

- A default value for the uproc table was set to 20 in uproc.h. This makes the assumption that there will never be more than 20 running, runnable, or sleeping processes. It also only copies up to a maximum of 10 process.

Console Modification

- proc.c

UID, GID, and PPID System Calls

To start implementing the uid, gid, and ppid system calls, I started off exactly as how I my date system call from project 2 was implemented. First, I had to add function prototypes and declarations for each of these new system calls functions, 3 getters for each of them and a setter function for uid and gid. These matched the same locations as the date system call. I added the declarations in syscall.h which is what connected my system call to my implementation, to sysproc.c where the actual implementation is located, user.h where the function prototype is held, syscall.c where the connection of the shell and the system call defined in syscall.h is, and usys.S which is where the link between the user and the system call is created.

I had to then add new variables to my process structure located in proc.h so that it could hold the uid and gid of a process. These values were stores as uint because the value could end up being drastically big, and then they were initialized to constant values in proc.h using #define at the top of the file. When implementing the setters for uid and gid, I wasn't able to pass in an argument from the bash to the kernel, so I had to pop the integer value off the stack using the argint(..) command (sysproc.c:124 – 144). Then I used this value to store it into the process's uid or gid respectively. As for the getters, I was only returning the value held in the process structure, so there was nothing that needed to be popped from the stack. After this was implemented, I could set the gid an uid from the new bash and then display the new value.

User-Level ps Command

Much like my previous system calls, I added the declarations and function prototypes to the files listed in the project deliverables. These linked the bash with the kernel so that it could be used properly. This differs from the previous system calls in that it passes in a data structure into the kernel, an array of uproc processes. A uproc structure was made in uproc.h where it contains

information about a process to be displayed by the ps command. The uproc data structure holds information such as the pid, gid, uid, ppid, the state the process is in, the size of the process in bytes, and the process name. In sysproc.c where the implementation of the system call sys_getprocs(void) is at, I needed to pop off an integer and a uproc pointer form the stack since that is the two parameters for the functions (sysproc.c:148). After doing so successfully, I called a new function, getProcInfo(..) within the system call in sysproc.c to populate the uproc table from the stack with all the necessary processes up to max, the integer from the stack.

In proc.c, where getProcInfo(int, uproc*) is implemented, I cycle acquire the lock to access a process and then I cycle through the processes. If I find a process that is either running, runnable, or sleeping, then I copy that process into the my uproc table. As soon as I run out of process, or if I reach my max limit, then I release the lock and return the number of processes copied in.

Finally, I implemented my ps command that utilizes the get getproc() system call. This user command is implemented in ps.c, and added into the Makefile under UPROGS in order to be executed from the bash in xv6. In proc.c, I call getprocs with a statically allocated uproc table, and then display the results from each process.

Console Modification

This was not necessarily difficult to display. In xv6 you can display basic process information by typing ctrl+p, but this information is missing the gid and uid. In order to add in the gid and uid, I went to proc.c, which is what the console.c file calls when displaying the information, and in the procdump() method, I added the uid and gid to the cprintf() function (proc.:466). Then, after pressing ctrl+p, the process displays properly with the gid and uid.

Testing Methodology

UID, GID, and PPID System Calls

For testing the uid, pid, and ppid, I created a simple test file testuidgid.c that can be executed to testing the functionality of the setters and getters. Running the command gives the following

output:

```
$ testuidgid
Current UID is 20
Setting UID to 42...
Current UID is 42
Current GID is 10
Setting GID to 77...
Current GID is 77
My parent process id is 2
Before creating a new process...
The current process gid: 77, uid: 42, ppid: 2
Creating a new process...
The outer-child process pid: 0, gid: 77, uid: 42, ppid: 3
Creating a new process...
The inner-child process pid: 0, gid: 77, uid: 42, ppid: 4
The inner-parent process pid: 5, gid: 77, uid: 42, ppid: 3
The outer-parent process pid: 4, gid: 77, uid: 42, ppid: 2
Testing Complete
```

As you can see, the setters and getters works fine. When it comes to using fork() to create a new process, you'll notice that the ppid of the outer-child process does not match the pid of the outer-parent, but the extra fork called in the outer-child. In this new process, the ppid of the inner-parent should be the pid of the outer-child process since that is the process that forked, which doesn't match up either. I checked my implementation of the sys_getppid() system call, and it is returning the current process's parent's pid, so the values should not be incorrect. From testing, I concluded that the processes must be using the pid of the sh process instead of the the actual current process. As you can see from the output, the inner-child's ppid is the outer-parents process.

Although unusual outputs, I've reviewed my implementations and they seem logically correct, so there must be something happening with the actual forking process.

User-Level ps Command

For my ps command, I simply used the output from the shell to make sure it was working correctly. We know that there are a few processes running when we start xv6. First, the init process, which is the very first process created, followed by the actual bash shell. When we run

the ps command, then that is another process, so that is included in the output too. If you look at the output, you'll see that the init and sh process are sleeping because the ps command is currently running.

```
$ ps
pid  uid  gid  ppid  state  sz  nm
1  10  20  1  SLEEPING  12288  init
2  10  20  1  SLEEPING  16384  sh
12  10  20  2  RUNNING  12288  ps
```

Additionally, we can change the uid and gid of the current process by typing '_setuid 70' and '_setgid 90' and then doing the ps command to see that the ps command's process now has the new gid and uid values. The shell process also takes on the new values because we are changing the values for the shell with the previous commands, but since ps command's parent process is the shell. As you can see, in relation to the previous testing for the uid, gid, and ppid commands, the parent process id for the ps command matches with the pid of the shell process. Then the shell command matches with the init process.

```
$ _setuid 70
$ _setgid 90
$ ps
pid  uid  gid  ppid  state  sz  nm
1  10  20  1  SLEEPING  12288  init
2  90  70  1  SLEEPING  16384  sh
13  90  70  2  RUNNING  12288  ps
```

Console Modification

This new command is rather simple to test. After adding the additional features to display the uid and gid (in that order in the display following the pid), then I can just run the command and analyze the output:

```
$ 1 20 10 sleep  init 80104c7e 80104a4d 801064e2 801056f9 801069f4 801067e5
2 70 90 sleep  sh 80104c7e 801009b6 80101e55 8010118f 801058d0 801056f9 801069f4
 801067e5
```

From this output, you can see that the init process matches the pid, uid, and gid from our ps command, and it shows that it is in the sleep state. Then the shell process matches the values for the pid, uid, and gid from the ps command as well. This too is in the sleeping state, so it follows the results from ps.