Sam Salin

5/15/16

Project 4

1. Description
    a. Modify the process table
        i. Add a free list
        ii. Add 3 ready lists, one for each priority
    b. Add a system call to set process priorities
    c. Implement a priority queue scheduler with three different priority levels
2. Deliverables
    a. Modified Process table
        i. Add a free list
            1. All unused processes are put on the free list
        ii. Implement priority queues
            1. If a process is runnable, it will be in the ready list for runnable processes of that priority, I.E if a process had a priority of one it would be stored in pReadyList[1]
    b. System call to set process priorities
        i. System call that takes a PID and priority, will change the priority of the process with that PID, will return errors if there is no process or the priority is an invalid value.
    c. New scheduler
        i. Processes with higher priority will always be run before processes with lower priority
        ii. Every so often
3. Implementation
    a. Modified Process table
        i. Free list
            1. Added a proc* pFreeList, a linked list of unused processes
            2. Tweaked userinit() in proc.c to add all processes to free list before the first process is created
            3. Added helper function addtofree() to stick processes back on the free list after they are killed. Adds unused processes to front of the list.
        ii. Ready Lists
            1. Pretty simple, just an array of proc * linked lists
            2. Added helper functions to proc.c
                a. addtoready
                    i. Will add a process to the head of the list if the list is empty, or to the tail of the list if it is not.
                b. removefromready

i. Either removes a process from the head of a list, or traverses through it with a previous pointer to remove the process from the list correctly

c. Both the helper functions look at the priority of the process being added or removed to make sure it is added or removed from the correct priority queue

b. System call to set process priorities
   i. Modified
      1. Proc.c
         a. Added dspri/dspri2 functions. dspri is can be called by userland via setpriority() and locks everything, dspri2 is called by stuff that already is holding a lock.
      2. Sysproc.c
         a. Added sys_setpriority system call. calls dspri since sysproc doesn't know about locks or the ptable.
            i. Takes two arguments, a PID and a priority.
      3. Testpriority.c
         a. Program changes the priority of itself, then reads it back to make sure it was set correctly.
   ii. Also added a syscall to get a process's priority to check if set worked.
      1. dgpri in proc.c
      2. sys_getpriority in sysproc.c
         a. take a PID, returns the priority for that process I it can be found or -1 if not

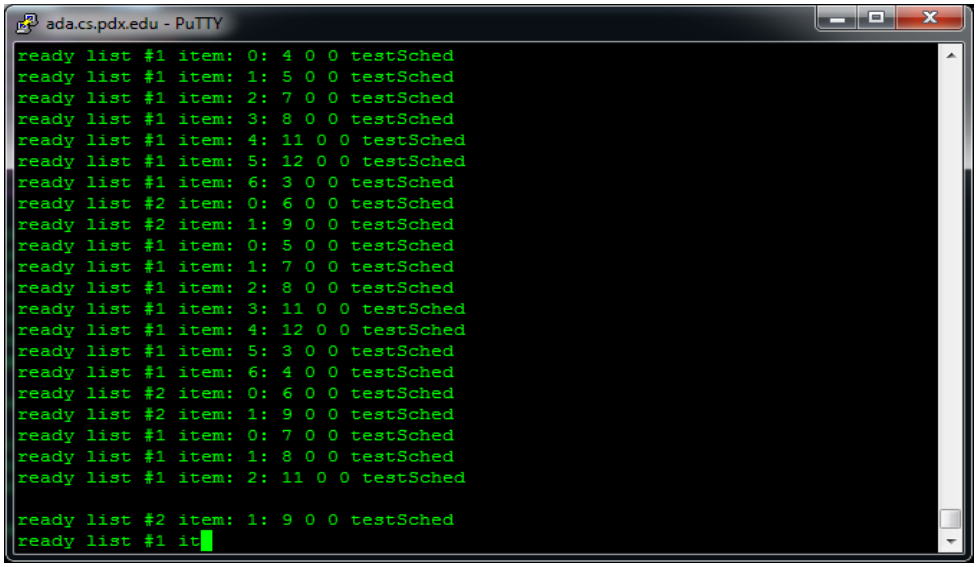
c. New Scheduler
   i. Modified Files
      1. Param.h
         a. Added MaxTimeToReset, the value that determines how often process priorities are reset to the default.
      2. Proc.h
         a. Added priority to process structure
         b. Added a proc * next field to process structure for linked lists
      3. Proc.c
         a. Modified the ptable to have a timetoreset member
         b. Modified the scheduler() routine
            i. Used a few if statements to make sure a higher priority job(0) will be selected over lower priority ones(1 and 2)
            ii. Modified to use ready lists instead of scanning the whole process table for runnable processes
            iii. Every time a process is about the be chosen the timetoreset counter is decremented, when it hits 0 all processes will be reset to the default priority (1) by the resetpriority() function

c. Modified fork() and allocproc() to set new processes to default priority(1)
d. Changed procdump to print the process's priority after its name when ctl-p is pressed.
e. Modified the sleep routing
    i. Clear the next pointer on sleeping processes since they aren't on any lists.
f. Modified the yield routine
    i. Calls addtoready() after the process state is changed from running to runnable
g. Modified the wait routine
    i. Clears priority and next pointer, calls addtofree()
h. Modified wakeup1
    i. When a process is woken up it now gets placed back on a ready list via addtoready()
i. Added resetpriority() function
    i. Uses removefromready to remove a process from whatever ready list it is in, changes the priority, then calls addtoready() to stick it back on the right list. Also checks if a process is currently running and does not add/remove from lists if this is the case.
j. The new scheduler relies very heavily on the addtoready and removefromready functions

ii. Added files
    1. Test programs
        a. testSched.c
            i. creates a bunch of child processes and changes their priority. Code for this program was provided via the class email list
        b. testpriority.c
            i. basic little test program that gets and sets its own priority.

iii. misc. changes
    1. modified PS program to print process priority after its name
    2. modified procdump to show a process's priority after its name
    3. It should be noted that the laptop I was doing a lot of work on died near the end of this project. Since all the work was being done on a school server this was inconsequential from a functionality standpoint. The changelog I was making, however, was lost so there may be a couple small changes that were not documented properly as a result.

4. Testing
    a. Building the new scheduler was an arduous process that took a lot of steps, not all of which were documented well. There was a lot of incremental development, an attempt to get everything working before adding layers of complexity. Test data is all from the finished 3-level priority queue
    b. A lot of testing was done using cprintf statements, these have all been commented out/deleted from the submitted code because removing them also removed some deadlocks or other spots where the system would hang.
    c. Test Programs
        i. The function printlists() was added with the intention of using it to show the priority lists, sadly when it was called the only processes running were it and init/sh, and init/sh were sleeping at the time so the only process it would print was the one that had just been fired off. I put a call in scheduler() then called the testsSched program and got this, which shows that multiple lists are being created and managed correctly. Call has since been commented out, but can be re-implemented on line 504 of proc.c



        ii. testpriority
            1. Output from the program is as follows:

```
process 3 priority is 1
process 3 priority is now 2
```

            From this it can be seen that the get/set priority system calls work since for the second line the process priority was read via the get call.

        iii. testSched

```
$ testSched
8: start prio 1
7: start prio 0
10: start prio 0
10: new prio 1
```

```
7: new prio 1
11: start prio 1
12: start prio 2
13: star1t prio 0
6: start prio 0
11463: new prio 1
: start prio 1
: start prio 1
11: new prio 2
16: new prio 1
16 0 0 runble testSched 1
15 0 0 runble testSched 2
14 0 0 run    testSched 1
13 0 0 run    testSched 1
12 0 0 runble testSched 2
11 0 0 runble testSched 2
10 0 0 runble testSched 2
9 0 0 runble testSched 2
8 0 0 runble testSched 2
7 0 0 runble testSched 2
6 0 0 runble testSched 1
2 0 0 sleep  sh 1 80105236 80104da4 80106d16 80105f08 801072fa 801070eb
1 0 0 sleep  init 1 80105236 80104da4 80106d16 80105f08 801072fa 801070eb
```

From this it can be seen that process can be created, ctl-p is used to check that priorities are assigned correctly. There aren't any high-priority processes in that particular log, but it can be seen that items with a priority of 1 are being run over items with a priority of two, which is what should be happening.

5. Bugs
    a. Exec sometimes fails
    b. Time doesn't pass things to exec quite right

Project 3

1. Description
    a. Add foundation for users and groups by adding user and group ids to processes. Also add a spot for parent process ids.
    b. add the following system calls:

i. getuid
ii. getgid
iii. getppid
iv. setuid
v. setgid

c. Implement a PS user program
    i. Add a user data structure uproc for the process information we need
    ii. Add a getprocs() syscall that can fill in the uproc structures
    iii. Getprocs returns number of processes on success and -1 on any error
d. Add uid and gid of processes to the ctl-p function

2. Deliverables
    a. System Calls
        i. The ability to set a processes
            1. User ID (UID)
            2. Group ID (GID)
        ii. The ability to get a processes
            1. UID
            2. GID
            3. Parent Process ID (PPID)
        iii. Make sure that the first process has everything squared away correctly
        iv. Make sure fork makes new processes inherit their parents user and group IDs
    b. PS user program
        i. When the user enters "ps" at the command line the following information needs to be displayed, separated by two spaces each
            1. Process ID (decimal)
            2. Process UID (decimal)
            3. Process GID (decimal)
            4. PPID (decimal)
            5. State (string, all caps)
            6. Size (decimal)
            7. Name (string)
        ii. There should be a variable MAX inside the program that tells it how many processes to try and grab
        iii.
    c. Modified version of the ctl-p function with the following additions
        i. Prints process UIDs
        ii. Prints process GIDs
3. Implementation
    a. UID, GID, PPID, and System Calls
        i. Modified the following files
            1. sysproc.c
                a. Implementation for all the get/set system calls is in this file between lines 110 and 141.

2. user.h
    a. Added all the function prototypes to allow user programs access to the system calls
3. proc.h
    a. Modified the proc structure to include signed integers for UID/GID. I chose to use signed since a negative value isn't necessarily bad.
4. sh.c
    a. Changed the shell to allow system calls to be called with _getxxx/_setxxx commands. Code was provided by instructor.
5. proc.c
    a. Tweaked the fork function to make new processes inherit their parents UID/GID as well as setting the PPID
    b. Modified allocproc to set the GID, UID, and PPID of new processes to 0 by default

b. PS user program
    i. Modified the following files
        1. user.h
            a. Added a declaration for getprocs() so user programs can call it
        2. sysproc.c
            a. Added the getprocs() system call, which is basically a wrapper for dogetprocs since letting the user directly access the process table is a terrible idea.
        3. proc.c
            a. Added the dogetprocs() function which is called by the getprocs() system call. It allows information to be stored in the uproc data structure without having to make proc.h visible to the user. It should be noted that the dogetprocs will fill in the uproc structure for processes in any state other than unused. Basically the function locks the process table and runs through it until either max iterations have been run or it sees an unused process, at which point it unlocks the process table and returns. It also should be noted that if max is larger than NPROC an error will be returned.
        4. defs.h
            a. Added dogetprocs() so that it would be available in sysproc.c
    ii. Added the following files
        1. ps.h
            a. Contains a data structure uproc that , with one exception, is in aligment with the datatypes in the ps program requirements. I chose to use an integer to contain the state since it uses less memory than the char[max_length]

alternatives and could not find a copy_to_user() type function in xv6.

2. ps.c
    a. contains implementation for ps program, the following things are of note
        i. states are printed with a switch() control block instead
        ii. make an array of uproc structures of size max
        iii. if there are less running processes than max only the number of running processes will be displayed
        iv. there is a print loop loop that will iterate either max or the return value of getprocs number of times if the return value is smaller.

c. Modified Ctl-p
    i. Modified the following file
        1. Proc.c
            a. procdump function was tweaked to print the uid/gid for all running processes, all that was needed was to add a couple %d places to the printf command. Currently it goes: PID, GID, UID

4. Testing
    a. System Calls
        i. Tried setting and getting from the command line

```
$ _setuid 50
$ _getuid
50
$_setgid 5
$ _getgid
5
```

        ii. Used provided code for test program called testids (testids.c) to check if getppid works.

```
$ testids
Current UID is : 0
Setting UID to 100
Current UID is : 100
Current GID is : 0
Setting GID to 100
Current GID is:100
My parent process is : 2
Done!
$ 1 0 0 sleep  init 80104cac 80104a7b 80106761 80105953 80106cb3 80106aa4
2 0 0 sleep  sh 80104cac 801009b6 80101e55 8010118f 80105b2d 80105953 80106cb3 80106aa4
```

        The last two lines are the output of the ctl-p program, and as can be seen the shell is process 2, which make the test program having a parent pid of 2 seem valid.

    b. ps program
        i. Tried running the ps program from the shell, everything looked good
        ii. Tried timing the ps program, the time process showed up as expected

```
$ _setuid 10
$ _setgid 9
$ time ps
1  0  0  0  SLEEPING  12288  init
```

```
2  10  9  1  SLEEPING  16384  sh
9  10  9  2  SLEEPING  12288  time
10  10  9  9  RUNNING  12288  ps
the program ran in 0 minutes and 0 seconds
```

Since the first column is the PID and the 4th is PPId, we can see that processes set their PPID correctly. We can also see that UIDs and GIDs are properly inherited based on the second and third column. Time has such a high PID because I ran ps a few times before thinking to include showing uid/gid inheritance.

iii.

c. Modified ctl-p
    i. Tried pressing ctl-p at the shell, everything showed up in a manner which was as expected. The first column is the PID, the second column is the group id, and the third column is the UID. Otherwise this function preforms the same as it did before being modified.

```
$_setuid 10
$ _setgid 4
$ 1 0 0 sleep  init 80104cac 80104a7b 80106761 80105953 80106cb3 80106aa4
2 4 10 sleep  sh 80104cac 801009b6 80101e55 8010118f 80105b2d 80105953 80106cb3 80106aa4
```

5. Bugs
    a. Calling _getpid or _getppid from the shell doesn't work

```
$ _getppid
Invalid _get parameter
$ _getpid
Invalid _get parameter
```

    b. The time program is not passing argv into exec() correctly
    c. exec fails sometimes.

Project 2

1. Description
    a. System call tracing
        i. Add code to the kernel that prints the name and return value of every system call that happens.
    b. Date system call
        i. create a date() system call that uses cmostime() to read the system clock.
    c. Time program
        i. use the date system call to implement a time program that times how long a program takes to run.
2. Deliverables
    a. System call tracing
        i. a modified version of syscall.c that has the ability to print the name/return value whenever a system call is made.
    b. Date system call
        i. date system call that reads the clock and user program that prints the current date and time.
    c. Time program

i. working time program time program that correctly measures the runtime of a program using the date system call.

3. Implementation
    a. System call tracing
        i. Modified syscall.c
            1. Added a function that took the syscall number and printed out its name called get_name. not super proud of it since it was basically a giant case statement and I feel like there is a cleaner way to do it. Has been commented out of the syscall() function because I don't really understand how to use IFDEFs. See lines 211/213 of syscall.c to see where called. It should be noted that this was not modified to accommodate the date system call.
    b. Date system call
        i. Modified the following files
            1. user.h
                a. added prototype for date()
            2. usys.S
                a. added date to the list of system calls
            3. syscall.c
                a. added date to syscall table
                b. added a different date() prototype so the call would be available outside the kernel
            4. sysproc.c
                a. added sys_date() function at the bottom

        ii. Added the following file
            1. date.c
                a. contains the user program which calls sys_date as well as a function to display what the date is. rtcdate structs were handed to cmostime() in order to hold the date information.
    c. Time program
        i. No pre-existing files were modified
        ii. Added the following file
            1. time.c
                a. time.c is pretty simple, there are three rtcdate structs that are used to grab the start time and end time and hold the final time. A little bit of math was done to make sure the displayed time values weren't negative owing to rollover effects.

4. Testing
    a. The system call trace
        i. Nothing fancy, just watched the boot sequence and made sure the values being were as expected. Went so far as to step through in GDB and watch every syscall fire off, it was interesting but maybe not very productive. Also tried running a few programs from the shell and got the expected results.

b. Date system call
    i. Again nothing fancy, just called date from the command line a few times. So far as I know it is impossible for cmostime() to fail, so not super worried about things going wrong.
    ii. Tried giving date() some input but nothing happens, which makes sense because it doesn't look at stdin.
c. Time program
    i. Tried timing a few programs, for the ones that took less than a minute it was kinda hard to do any error checking, but when compared to a wall clock time usertests seemed like it was right even though the times I was getting were about half what people on the mailing lists were getting.
    ii. I timed a few things and tried several times, especially near hour/minute changes to make sure the effects of rolling over into a new hour or minute were properly accounted for. Wasn't super exhaustive owing to the need to wait an hour between testing to make sure rollover was dealt with properly, appeared to work fine though.
    iii. Also tried feeding the program bad input, all that happens is exec() fails and the program thinks it runs in 0 time.

5. Bugs
    a. Sometimes when things are run at the shell they will fail for unknown reason, it's possible exec might fail for the timer or date.
    b. The system call tracing has some concurrency issues where console output will get mixed up and isn't super legible.