PROJECT 4: BETTER PROCESS MANAGEMENT

Ricardo Valencia

PORTLAND STATE UNIVERSITY - CS333: Intro to Operating Systems

Project Description

This project focuses on adding a new scheduler that utilizes a new priority field which will avoid starvation with a timer. In order to do this, we add a ready and free list where the ready list improves the performance, and the free list improves efficiency during process creation. In order to accommodate the new priority field for the processes, we also add a new system call that changes the priority based on the pid passed in.

Project Deliverables

Set Priority System Call

- syscall.h
- defs.h
- user.h
- proc.h
    - Sets constants for the priorities you can have: HIGH (0), DEFAULTPRO(1), and LOW(2).
- usys.S
- sysproc.c
    - Has edge case checks so that the priority cant be below 0 or above 2, and so that the pid can't be less than 0.
- syscall.c
- uproc.h
- ps.c
- proc.c
- testPriority.c

New Scheduler

- proc.h
- proc.c
- testSched.c

Set Priority System Call

In order to implement this new system call, I add function prototypes and implementations for the new functions. In syscall.h, the system call is given a unique system call id, user.h is where the prototype is at, syscall.c declarations is what connects the shell to the system call and sysproc.c is where the actual implementation is held for the system call. In usys.S I added another declaration for the set priority system call that links the user to the system call. Finally, I made an additional helper function that iterated through the process list to locate the correct process with the pid in proc.c.

For the actual implementation of my set priority system call, I had to pop two integer values from the stack, the process pid and the priority. Once I had these available in my kernel code, I made the previously mentioned helper function setPriority() in proc.c. In here, I cycle through all the process using the ptable.proc field and look for a process with the matching pid that was passed in as an argument. Once it's found, it stores the old priority in a variable and reassigned the priority of the process to the new priority passed in as an argument. If the process is runnable, then it has to be put back into the correct queue, so I remove it from the previous queue using the old priority and then I add it back into the new queue using the new priority. If it is not running, then only the priority is changed. If the old priority matches the new priority, however, then I leave the process alone, release the lock, and return out of the function. I should note here that I had to acquire the lock during the iteration of my processes because I would be manipulating the data. It's released before I return out of the function. It returns 0 if the process with the matching pid was found, and -1 if it was not found.

Since this system call requires a new priority field for our processes, then I also had to add a new field in the proc structure in proc.h. Since these system calls have to be displayed when the new user program 'ps' or the 'ctrl + p' is ran, then I had to add a new field to the uproc structure that holds the priority to be displayed by ps, copy the priority from a process to the uproc process table in the getProcInfo() function in proc.c which is used by 'ps', and add a new print formatter to the procdump() function in proc.c as well to display the priority when 'ctrl + p' is ran.

New Scheduler

The new scheduler is a bit convoluted since there was a lot to keep track of. First I had to create new fields in the ptable all located in proc.c: a ready list, a free list, and the time to reset. The ready list stores only processes that are runnable while the free list holds on processes that are unused. In order to make these a list, I added a new proc pointer field, next, to the proc structure in proc.h that points to another process (the next item in the list). The time to reset field was set to 4500 which I found worked efficiently enough to prevent starvation of the processes in the lowest priority queue of the ready list. The ready list was implemented by being a static array of size 3 of pointers to a process. The highest priority was 0, default priority is 1, and the lowest priority is 2. Each of these integer values have constants created for them in proc.h. I also stored the time to reset, 4500, into the constant count in proc.h.

To start the actual implementation of the new scheduler, I began in userinit() in proc.c. I started by acquiring the lock and initializing my free list and timeToRest before releasing the lock. To initialize my free list, I first set it to 0, then I iterated through all the processes and searched for all the processes that were in the unused state. If they were, then I would add it to the front of the free list. Followed by this initialization, allocproc() is immediately called which initializes a processes. In allocproc() I get the first process in the free list, and if it is not null then I set the

head pointer of the free list to the next process in the list, and reassign the next pointer to 0 so it no longer points to the free list. If it does happen to be 0, then it releases the lock it acquired and then returns out. It also does a check to make sure that the kernel stack memory was properly allocated, if it wasn't then it puts the process back onto the free list and returns. This process is then returned from allocproc() and userinit() continues to initialize the first process. At the end of userinit() the first process is set to the runnable state, so I first initialize my ready list by setting each index to 0, then I add this process to the appropriate queue, which happens to be 1, the default priority. Allocproc() is used throughout the new scheduler to initialize any new unused processes from our free list.

Before continuing further into the implementation, I should note that I created multiple helper functions that helped me initialize my free list, add to my freelist, add to my ready list at the appropriate queue, remove from my ready list, and a function that resets the ready list, as well as processes in the running or sleeping state, back to their default priority, and to their appropriate queue afterwards.

In fork(), since this is where new processes are created, so this is where it will be put on the ready list given that it first copies the parent process correctly. If it doesn't copy the process state from the parent process p, then it is put into the free list, otherwise it is put onto the ready list and has the state set to runnable. The wait() function also does a check to see if the process is a zombie, if it is, then it sets the state to unused and puts it back onto the free list. Yield() sets a process to runnable, so it has to be put back onto the ready list at the appropriate queue. Then comes wakeup1() which needs to put a process back on the ready listen if it's a sleeping process. Unlike the previous functions, this function acquires the lock before the function is called so I didn't have to explicitly acquire the lock in wakeup1() and release it, since it's done outside the function. In kill(), if the process state is sleeping, then it sets it back to runnable and inserted into the ready list using the helper function, just as the other functions do.

Finally, since all cases where a state change occurs I managed my ready list and free list appropriately, then I was able to implement my new scheduler. As a rule, only processes with the unused state can be in the free list, and only processes with a runnable state can be in the ready list, so the previous section went through every block of code where a state change happened and managed the process appropriately. In my scheduler I then only have to cycle through my ready list to find a process that was ready to be ran. If the highest queue is empty, then it continues to the next queue. Once it finds a process that it can run, it sets the head of the ready list – which is where I pop from since I add to the tail end of the list – to the next process in the list after assigning the process to a pointer, p. I then decrement my time to reset counter, which again is set to 4500, and I do a quick check to see if it is 0. If it's 0, then it's time to reset the

priority of my runnable, running, and sleeping processes in order to prevent starvation of the processes in the lowest queue. After finding a priority, I always start back at the highest priority instead of starting off at the same or next queue, so it always starts at the highest priority when looking for the next process to run.

All of the new features are include in ifdef statements that I then add to the CFLAGs of my Makefile so that I can still run my old code. If the CFLAG is present, then my new scheduler code runs, otherwise the old scheduler runs.

Testing Methodology

Set Priority System Call

In order to test my set priority system call, I created a testing file testPriority.c than I run in xv6 that takes two processes, init and sh and sets their priority (integer right after the state). I chose these two processes because I could re-use the code from ps.c to print a process's information and init and sh are always showing up with a pid of 1 and 2 respectively. Running my program, I get the following output:

```
$ testPriority
Before setting priority for pid 1 (init)...
pid  uid  gid  ppid  state prio sz  nm
1  10  20  1  SLEEPING 1  12288  init
Setting priority of pid 1 (init) to 0...
pid  uid  gid  ppid  state prio sz  nm
1  10  20  1  SLEEPING 0  12288  init
Setting priority of pid 1 (init) to 2...
pid  uid  gid  ppid  state prio sz  nm
1  10  20  1  SLEEPING 2  12288  init
Setting priority of pid 2 (init) to -10...
pid  uid  gid  ppid  state prio sz  nm
1  10  20  1  SLEEPING 2  12288  init
Setting priority of pid 2 (init) to 100...
pid  uid  gid  ppid  state prio sz  nm
1  10  20  1  SLEEPING 2  12288  init
Setting priority of pid 2 (init) to 1...
pid  uid  gid  ppid  state prio sz  nm
1  10  20  1  SLEEPING 1  12288  init

Before setting priority for pid 2 (sh)...
pid  uid  gid  ppid  state prio sz  nm
2  10  20  1  SLEEPING 1  16384  sh
Setting priority of pid 2 (sh) to 0...
pid  uid  gid  ppid  state prio sz  nm
2  10  20  1  SLEEPING 0  16384  sh
Setting priority of pid 2 (sh) to 2...
pid  uid  gid  ppid  state prio sz  nm
2  10  20  1  SLEEPING 2  16384  sh
Setting priority of pid 2 (sh) to -10...
pid  uid  gid  ppid  state prio sz  nm
2  10  20  1  SLEEPING 2  16384  sh
Setting priority of pid 2 (sh) to 100...
pid  uid  gid  ppid  state prio sz  nm
2  10  20  1  SLEEPING 2  16384  sh
Setting priority of pid 2 (sh) to 1...
pid  uid  gid  ppid  state prio sz  nm
2  10  20  1  SLEEPING 1  16384  sh
Testing Complete
```

As you can see, I cycle through changing the priority form 0-2 and the changes reflect in the process, and then I test the edge cases of attempting to change the priority to a negative number or a number larger than 2, and in both cases the priority does not change. I also set a check to in my code that tests to see if the pid is less than 0, and if it is then it returns -1 just like an invalid priority integer. In my setPriority() helper function in proc.c, I make sure to change

the priority of any process, not just one that is in the runnable state which is why these sleeping processes can get their priority changed.

New Scheduler

To test the new scheduler, I utilized the provided testSched.c test file that creates various new processes at different priorities. As each new process is created, it is inserted into the correct queue based on my fork() implementation. During my testing, I inserted a print statement that would display a message before the reset ready list code was executed. This let me configure my count constant, or the time to reset value: 4500. This is a snippet of how the testSched.c started and set priorities:

```
$ testSched
4: start prio 0
5: start prio 1
6: start prio 2
7: start prio 0
4: new prio 1
8: start prio 1
9: start prio 2
10: start prio 0
7: new prio 1
11: start prio 1
12: start prio 2
3: start prio 1
13: start prio 0
10: new prio 1
```

From these changes, I used Ethan Grinnell's small script in GDB that lets you print all the items in your ready list and it gave me these results:

```
(gdb) plist
Queue1 testSched PID 7
Queue1 testSched PID 11
Queue1 testSched PID 10
Queue1 testSched PID 3
Queue1 testSched PID 5
Queue1 testSched PID 4
Queue2 testSched PID 6
Queue2 testSched PID 9
Queue2 testSched PID 12
```

As you can see, process's 6, 9, and 12 are all in the lowest queue, 2, and everything else is in queue 1. You'll notice that some priorities have been set to the highest queue, 0, but they aren't showing up in the ready list from the plist command. After stepping through with gdb, I found that generally the processes that are inserted in the highest queue are ran immediately and there is never really enough time in between each process to have more than a couple process's in these queues without being ran almost immediately. In summation, they are being inserted,

except they are being taken off to be ran as soon as they are put on since they are in the highest queue. In order to actually check to make sure that they were being inserted into the highest queue, I had to set a breakpoint in my scheduler() function and wait until I hit a process with a priority of 0, then I displayed the results and it showed up in the highest queue.

As for the reset, I waited until the time to reset counter hit 0, then my print statement came out and I set a breakpoint at my ready list reset function allowing me to display my ready list after the reset which you can see from the following results: