Project 4 Report
CS333 - Intro to Operating Systems
Winter 2018

Ryan Hoover

2/25/18

0.1 Description

The previous assignment, Project 3, had us rewrite the old `scheduler` routine to use the ready list when searching for a `RUNNABLE` process instead of iterating through the process array. The rewritten scheduler is still functionally equivalent to the old one: both are simple, round-robin schedulers. In this project, a new scheduler is implemented for xv6.

0.1.1 MLFQ, New State Lists & Priority

A variation of a Multi-Level Feedback Queue has been implemented, that uses a slightly different approach than the text for preventing process starvation. Our approach utilizes "demotion" based on a *budget*, and "periodic promotion" (starvation prevention), which is the primary difference between the approach of the book and our implementation.

Each process will have an associated priority that will dictate the ready list to which it belongs when in the `RUNNABLE` state. Upon allocation, each process will have the same initial (default) priority value, the highest priority. The process priority value may be changed during process execution via a new system call (`setpriority`).

0.1.2 New Process Budget, Promotion & Demotion

This implementation of an MLFQ will utilize a time *budget* instead of basing the decision on the fraction of a time slice used. Each process is assigned its own budget. When a process is removed from the CPU via a context switch, the budget is updated according to a formula: *budget = budget - (time_out - time_in)*. If the budget is less than or equal to `0`, then the process will be demoted and placed at the tail of the next lower queue and the budget value is reset. If the budget is not expired, the process will be placed at the tail of the appropriate queue when it again reaches the `RUNNABLE` state. Periodically a promotion timer will expire. The expiration of this timer will cause each process to be promoted one level. Promoted processes are placed at the tail of the new queue. The promoted process's budget will be kept the same.

0.1.3 Modified "ps" and Control Commands

The `ps` and `control-P` commands will now report the priority levels of each process alongside the rest of the information these commands already cover.
The `control-R` command will now will print data for each of the `ready` lists, in order from `0` to `MAX`. The `PID` (Process ID) and available *budget* of each process will be displayed in order on each list.

0.2 Deliverables

This section will provide descriptions and details of all features added to xv6 for this project.

0.2.1 New Ready Lists, MAX, Priority

The ready list for `RUNNABLE` processes will need to be updated to support multiple priority queues. The amount of queues will be determined by a new constant, `MAX`. The number of each queue will determine what *priority* it represents, with `0` being the highest priority.

New Ready Lists

Changes to the declaration of the `ready` list in `struct StateLists`:

```
struct proc* ready[MAX+1];
```

The `ready` list from Project 3 is now implemented as an array of lists, where each index of the `ready` list corresponds to a priority queue in the MLFQ. The invariant for the ready list will change as well: the ready list contains all of the `RUNNABLE` processes of the system with each process in `pLists.ready[`*i*`]` having a priority of *i* (`0 <=` *i* `<= MAX`).

MAX

We should not assume a fixed value for the number of priority queues. Instead, we will use a `#define` that is visible in both user and kernel mode. The scheduling algorithm is flexible enough to adapt to wide variations in the number of queues.

Priority

Each process will have an associated priority (as a `uint`) in the range of `[0...MAX]` that will dictate the ready list to which it belongs when in the `RUNNABLE` state. This means that there are `MAX + 1` possible priorities for each process. Upon allocation, each process will have the same initial (default) priority value, the highest priority. The process priority value may be changed during process execution via the `setpriority` system call. The highest priority in the system will be `0` with each value greater than zero being a successively lower priority, the lowest being `MAX`.

0.2.2 New "setpriority" System Call

The function prototype is:

```
int setpriority(int pid, int priority);
```

The `priority` argument value ranges from `[0...MAX]`. The system call will have the effect of setting the priority for an *active* process with PID `pid` to the value of the `priority` argument and resetting the *budget* of the default value. An error is returned if the values for `pid` or `value` are not correct. The system call enforces bounds checking on the priority value.

0.2.3 Process Promotion

To address the problem of **starvation**, we implemented a promotion strategy. The strategy is to periodically increase the priority of all active processes by one priority level; that is, the priority of all processes in the `RUNNABLE`, `SLEEPING`, and `RUNNING` states will be periodically adjusted. The following approach will be used:

1. Added a new field to the `ptable` structure, `uint promoteAtTime`. The value stored will be the *ticks* value at which promotion will occur. This value is the same for all processes, so it is set in the `ptable` structure, *not* each process.

2. Created a constant `#define TICKS_TO_PROMOTE XXX`, where `XXX` is the maximum number of ticks that the scheduler runs before all the priorities are adjusted. Each time that the routine `scheduler()` runs, the value is checked to see if it is time to adjust priorities.

3. When the value of `ticks` reaches `promoteAtTime`,

   (a) Adjust the priority value for all relevant processes to the next higher priority.

   (b) Change the priority queue for a process as appropriate, putting any adjusted process on the back of their new queue(s). Processes for which the priority was not adjusted are not moved, to avoid starvation.

   (c) Value of `promoteAtTicks` will be set to `ticks + TICKS_TO_PROMOTE`, to calculate when the next adjustment will be.

0.2.4 Process Demotion

Process demotion will be handled in accordance with the familiar MLFQ algorithm. When a process is removed from the CPU via a context switch, the budget is updated according to a formula:

```
budget = budget - (time_out - time_in)
```

If *budget* is less than or equal to `0`, then the process will be *demoted* and placed at the tail of the next lower queue and thr `budget` value is reset. If the *budget* is not expired, the process will be placed at the tail of the appropriate queue when it again reaches the `RUNNABLE` state.

0.2.5 Updated "ps" Command, "ctrl-P" & "ctrl-R"

The following commands are updated in Project 4:

- `ps` Command. The `ps` command now reports the priority level of each process. The `uproc` structure has been updated with a new `priority` field, to be copied from an active process's `priority` field.

- `control-P` Console Command. The priority level of each active process will be displayed, read from a particular process's `priority` field.

- `control-R` Console Command. The ready list will now display the contents of each ready list, corresponding to different queues. The contents of each item in these lists will be altered to not only show the PID of each process, but the *budget* of each process as well.

0.2.6 How MLFQ Affects Scheduling Decisions

Our MLFQ algorithm utilizes a time budget instead of basing our decision on the fraction of a time slice used. Each time the MLFQ algorithm runs, the process at the front of the highest priority queue will be selected to run. Each time the algorithm looks for a new process, it must start by checking the highest priority queue and only checking a lower queue if no higher priority jobs are available. The approach:

1. Each process is assigned its own budget.

2. Each priority level has an associated FIFO queue, each of which is serviced in a round robin fashion.

3. A newly created process is inserted at the end of the highest priority queue and is assigned to a CPU. The system already records the time at which the process entered the CPU in the process structure.

4. At some stage the process reaches the head of the queue and is assigned to a CPU. The system already records the time at which the process entered the CPU in the process structure.

5. If the process exits before the time slice expires, it leaves the system.

6. When a process is removed from the CPU via a context switch, the budget is updated. If the budget value is ¡= 0, then the process is demoted and placed at the tail of the next lower queue and budget value is reset. If the budget is not expired, the process will be placed at the tail of the appropriate queue when it again reaches the `RUNNABLE` state.

7. Periodically, a promotion timer will expire, The expiration of this timer will cause each process to be promoted one level. Promoted processes are placed at the tail of the new queue.

The MLFQ tries to address a fundamental problem. First, it optimizes *turnaround time* by running shorter jobs first. Second, MLFQ makes a system responsive to interactive users and minimizes *response time*. The scheduler will "learn" that a process is interactive by the way promotions and demotions are handled, and what `ready` list the process tends to reside on, based on its behavior.

0.3 Implementation

This section identifies all files and line numbers modified for the implementation of Project 4.

0.3.1 Priority Queues

- *New ready lists in* `ptable`*; New* `MAX` *define; Initialization of* `ready` *lists.*
  - `proc.c` (Line 15). The `ready` list from Project 3 is modified to be an array of lists, set to size [`MAX + 1`].
  - `param.h` (Line 19). `MAX` is defined here, set to any size we want. The ready lists and all of the associated functionality scale alongside this constant.
  - `proc.c` (Lines 281 – 285). Here int `userinit()`, the ready lists are initialized. The highest priority list (index 0) is assigned to the newly allocated `RUNNABLE` process, that process's `next` pointer set to `NULL` (0), and the rest of the lists (if `MAX > 0`) are initialized to `NULL` (0).
- *New* `priority` *field in* `proc` *structure & initialization of priority.*
  - `proc.h` (Line 83). A process's priority is defined here as a `uint` (won't be below 0).
  - `proc.c` (Line 187). In `userinit`, the process's priority is set to the highest, `0`.
- *Changes to* `proc.c` *functions to properly place processes on the correct ready list.*
  In `proc.c`:
  - `userinit()` (Line 187). The first process in the system is assigned to priority 0.
  - `fork()` (Line 418). The process is added to a ready list at index `[p->priority]`; could also be 0, since every new process is created here (after the one in `userinit()`).
  - `yield()` (Line 872). The process is added to a `ready` list at the index corresponding to its priority (`[p->priority]`), which can be anything at the time of its removal from the `running` list. A demotion can take place after the removal from the `running` list, and before adding to a `ready` list.
  - `wakeup1()` (Line 994). The process is removed from the `sleep` list, and added to the `ready` list corresponding to its priority.
  - `kill()` (Line 1052). If a `SLEEPING` process is woken, it is made `RUNNABLE` and added to the ready list at `[p->priority]`.
- *Any modifications to helper functions or how to assert processes were on the correct priority queue on removal.*
  Helper functions were not modified from their implementation in Project 3. The dereferencing that occurs when using the [] brackets to specify an index of an array allows the array of ready lists to be used with the existing list manipulation helper functions. These functions take a double pointer (`struct proc** sList`) as an argument, and the newly created array of ready lists make `ptable.pLists.ready` a *triple* pointer. Adding or removing a process from a `ready` list simply involved changing the lines that added or removed a process from the `ready` list, to add or remove from a specific index of the `ready` list array, at whatever index corresponds to the process's current `priority` value.
  The line would look something like this, depending on the exact function name:
  `addToStateListEnd(&ptable.pLists.ready[p->priority], p);`

0.3.2 MLFQ Scheduler

Modifications to the scheduler so that it will potentially go through all `ready` *lists and pull the first process from the highest priority queue.*

In `proc.c`. The scheduler algorithm was not modified very much from Project 3. In (Line 761), the scheduler accesses the `ready` list array starting at index 0, and checks to see if the current index it is valid (Line 763). The first availible index will be chosen, and will not stop until the loop index has reached size `MAX` (not +1, because this is the actual index of the `ready` list array). If so, the process is removed from its `ready` list at whatever priority it is (Line 767) and added to the running list as usual (Line 777). A new variable, `ran`, is set from 0 to 1, to indicate that for this round, a process has been scheduled (Line 786). This is used as a loop condition for the `ready` list array traversal (so only one process is run each round), and reset in the scheduler's infinite outer loop to 0 (Line 752), so that it can search for a new process.

0.3.3 Promotion

- *New* `promoteAtTime` *field in* `ptable`*,* `TIME_TO_PROMOTE` *define and* `promoteAtTime`*'s initialization.*
  - `proc.c` (Line 37). Field `promoteAtTime` added to `struct ptable`.
  - `param.h` (Line 18). Constant `TIME_TO_PROMOTE` can be set to any value before compiling, this will power our promotion timer.
  - `proc.c` (Line 235). In `userinit()`, `promoteAtTime` is initialized to the value of `TIME_TO_PROMOTE`: `ptable.promoteAtTime = TIME_TO_PROMOTE;`.
- *How the scheduler determines it is time to promote.*
  In `proc.c` (Lines 756 – 759), after acquiring the `ptable` lock, the scheduler checks if the value `ptable.promoteAtTime` is equal to the global variable `ticks`. If so, the new `promoteAll()` function is run, increasing the priority of every process on the `running`, `sleeping`, and every index of the `ready` list array. The `promoteAtTime` field is then updated to incorporate the *current* number of `ticks` with `TIME_TO_PROMOTE`, setting a future time when the system will determine it is next time to promote:
  `ptable.promoteAtTime = (ticks + TIME_TO_PROMOTE);`
- *Which processes are promoted and how the budget of each process is handled during promotion.*
  A new function was added to handle promoting all of the active (`RUNNING`, `RUNNABLE`, and `SLEEPING`) processes: `static void promoteAll(void);` (Lines 1470 – 1516).
  This function traverses the `ready` list array, traverses each item in each list, and decrements the `priority` value of each process if it is greater than 0. While updating the `ready` lists, a process must be removed from the `ready` list at the index corresponding to its old priority, then priority is updated, then the process is moved to the new ready list (Lines 1474 – 1494). The `running` and `sleep` lists are traversed also, and their priorities adjusted if greater than 0 (Lines 1496 – 1514). The budgets are kept the same.
- *How promotion avoids starvation.*
  The promotion algorithm avoids starvation by setting `TIME_TO_PROMOTE` to a reasonable value. The text has referred to this as a "black magic number", or one that needs some kind of magic to determine the correct value. It can't be too high, or processes are never promoted in time, and it can't be too low or processes will only live on the higher priority `ready` lists, defeating the purpose of MLFQ.

0.3.4 Demotion

- *New* `budget` *field in* `proc` *structure and initialization.*
  - `proc.h` (Line 84). `int budget;` declared to store whatever budget value is given.
  - `proc.c` (Line 186). In `allocproc`, the budget field is set to the constant `BUDGET`, set in `param.h`.
  - `param.h` (Line 20). A constant is defined, set to whatever value we want. Like `TIME_TO_PROMOTE`, this is another "black magic number".

- *When demotion occurs and how it affects priority and the budget.*
  Demotion occurs in two places within `proc.c`, where a process would exit the CPU:
  - `sleep()` (Lines 936 – 943). Upon exiting the CPU, the process budget is updated according to a formula:
    `proc->budget -= (ticks - proc->cpu_ticks_in);` The budget subtracts and assigns the value of the time it spent in the CPU subtracted from the total `ticks` the system has been running. The budget is then checked if it is less than or equal to 0, and if so the process's `priority` value is incremented (demotion to lesser priority list), but only if the priority is already less than the largest `ready` list array index value. Regardless of whether or not the demotion occurs, the process's budget is reset to the constant value.
  - `yield()` (Lines 864 – 870). This function performs the exact set of steps described above. The demotion is performed before adding the process to a `ready` list, so that it will join the correct list depending on its updated priority.

0.3.5 `setpriority` System Call

- *All files updated to add the system call, and implementation in* `proc.c`*.*
  - `proc.c` (Lines 1520 – 1569).
  - `sysproc.c` (Lines 177 – 197).
  - `setpriority.c` Entire file added.
  - `Makefile` (Line 26). Added `_setpriority` label.
  - `runoff.list` (Line 99).

- *Finding the process, changing priority and budget.*
  `proc.c`. Similar to `promoteAll()`, the user-side function in `proc.c` traverses the `ready` list array and the list at each index, looking to match the `pid` argument to a process PID (Line 1528). It will look in the `sleep` and `running` lists if no match is found in a previous list (Lines 1546 & 1557). Once found, the process priority is set to the argument, and the budget reset.

- *Changing the state list when appropriate.*
  `proc.c`. If found on a `ready` list, the process is removed from one list, its priority set to the `priority` argument value, and its budget reset Lines 1528 – 1536).

- *Bounds checking on user input and error conditions.*
  `sysproc.c` The kernel-side function does the bounds checking, and returns `-1` for any errors. First, it checks to see if the `pid` argument is between `0` and `32767` to avoid an overflow (Line 189). The `priority` argument is checked to see if it is between 0 and `MAX`, to denote a valid priority to be set (Line 193).

0.3.6 Updated Commands

- *Modifications to* `control-P` *to include priority and* `control-R` *to display each processes budget.*
  In `proc.c`:
  - (Lines 1364 – 1388). `ctrl-R` calls the `printReadyList()` function, which was updated in two ways. It now traverses and displays all of the `ready` lists, and displays the budget of each process in addition to the PID.
  - (Lines 1182 – 1214). `ctrl-P` calls `procdump()`, which was rewritten and cleaned up, separated from the Project 2 and earlier version. This now displays the process's `priority` value alongside the rest of the process info.
- *Modifications to* `ps` *command to display priority, and new* `priority` *field in* `struct uproc`*.*
  - `ps.c` (Line 58). The table now prints a given process's `priority` value.
  - `uproc.h` (Line 16). Added a field for process `priority`.

0.3.7 New Test Programs

These programs were added to test new functionality:

- `p4test.c`. Bad edits to zFree.c.
- `p4test2.c`. The "countForever" program provided by Professor Morrissey.
- `tPrio.c`. Tests setting priority.
- `tspin.c`. Forks and endless loops.

```
Ready List Processes:

0: (7, 99170)-> (4, 99131)-> (5, 99130)-> (3, 99160)

Ready List Processes:

0: (5, 99080)-> (3, 99110)-> (8, 99110)-> (6, 99100)

Ready List Processes:

0: (8, 99060)-> (6, 99050)-> (7, 99060)-> (4, 99021)

Ready List Processes:

0: (7, 99010)-> (4, 98971)-> (5, 98970)-> (3, 99000)
```

Figure 1: Round robin enforced for a single priority level.

0.4 Testing

0.4.1 Round Robin

Demonstrate that round robin scheduling is still enforced by the MLFQ for a single priority level.

The screenshot in (Figure 1) shows that round robin scheduling is enforced by tracking how the process PIDs cycle through the ready list. This was done while running `p4test2` and repeatedly hitting `ctrl-R`. `MAX` was set to `0`, and the `Priocount` of the test program was set to `1`.

Result: Test 1 `PASSES`.

0.4.2 Promotion

Show that promotion properly changes the priority level for sleeping and running processes.

This test was performed by running the test program `tspin` to simply fork a few processes, set the priority of each one to `MAX` and spin forever. Also, the promotion timer is set to one second, and the budget time set to a high value so that demotion will not occur during the test, and hitting `ctrl-P` during the test to watch the promotion occur (see Figure 2). The sleeping processes are shown (see Figure 3) to be similarly updated by the promotion.

Result: Test 2 `PASSES`.

0.4.3 Demotion

Show that demotion properly resets the budget of the demoted process.

This test was performed by first adjusting the define `TIME_TO_PROMOTE` to an extremely high number, over 5 minutes, to ensure that no automatic promotions would take place during testing. The `budget` define for each process was changed to `2` seconds (`2000 ticks`), so that we could have enough time to track the demotion and see the budget being reset (See Figure 4).
This test was performed by running `p4test2` and hitting `ctrl-P` and `ctrl-R` repeatedly to see the relevant

```
$ p4test

PID     Name    UID     GID     PPID    Prio    Elapsed CPU     State   Size    PCs
1       init    0       0       1       0       3.619   0.015   sleep   12288   801056e5        80105187        80107cd2        80106ed2        8010827c        80108077
2       sh      0       0       1       0       3.603   0.011   sleep   16384   801056e5        80105187        80107cd2        80106ed2        8010827c        80108077
3       p4test  0       0       2       0       0.508   0.039   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
4       p4test  0       0       3       2       0.496   0.023   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
5       p4test  0       0       3       2       0.491   0.031   run     12288
6       p4test  0       0       3       2       0.486   0.041   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
7       p4test  0       0       3       2       0.481   0.025   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
8       p4test  0       0       3       2       0.475   0.023   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
9       p4test  0       0       3       2       0.470   0.047   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077

PID     Name    UID     GID     PPID    Prio    Elapsed CPU     State   Size    PCs
1       init    0       0       1       0       4.000   0.015   sleep   12288   801056e5        80105187        80107cd2        80106ed2        8010827c        80108077
2       sh      0       0       1       0       3.984   0.011   sleep   16384   801056e5        80105187        80107cd2        80106ed2        8010827c        80108077
3       p4test  0       0       2       0       0.889   0.039   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
4       p4test  0       0       3       1       0.877   0.073   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
5       p4test  0       0       3       1       0.872   0.047   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
6       p4test  0       0       3       1       0.867   0.073   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
7       p4test  0       0       3       1       0.862   0.051   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
8       p4test  0       0       3       1       0.856   0.035   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
9       p4test  0       0       3       1       0.851   0.081   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077

PID     Name    UID     GID     PPID    Prio    Elapsed CPU     State   Size    PCs
1       init    0       0       1       0       4.307   0.015   sleep   12288   801056e5        80105187        80107cd2        80106ed2        8010827c        80108077
2       sh      0       0       1       0       4.291   0.011   sleep   16384   801056e5        80105187        80107cd2        80106ed2        8010827c        80108077
3       p4test  0       0       2       0       1.196   0.039   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
4       p4test  0       0       3       1       1.184   0.097   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
5       p4test  0       0       3       1       1.179   0.071   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
6       p4test  0       0       3       1       1.174   0.093   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
7       p4test  0       0       3       1       1.169   0.087   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
8       p4test  0       0       3       1       1.163   0.048   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
9       p4test  0       0       3       1       1.158   0.089   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077

PID     Name    UID     GID     PPID    Prio    Elapsed CPU     State   Size    PCs
1       init    0       0       1       0       4.781   0.015   sleep   12288   801056e5        80105187        80107cd2        80106ed2        8010827c        80108077
2       sh      0       0       1       0       4.765   0.011   sleep   16384   801056e5        80105187        80107cd2        80106ed2        8010827c        80108077
3       p4test  0       0       2       0       1.670   0.039   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
4       p4test  0       0       3       1       1.658   0.121   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
5       p4test  0       0       3       1       1.653   0.090   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
6       p4test  0       0       3       1       1.648   0.112   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
7       p4test  0       0       3       1       1.643   0.121   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
8       p4test  0       0       3       1       1.637   0.073   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
9       p4test  0       0       3       1       1.632   0.134   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077

PID     Name    UID     GID     PPID    Prio    Elapsed CPU     State   Size    PCs
1       init    0       0       1       0       5.051   0.015   sleep   12288   801056e5        80105187        80107cd2        80106ed2        8010827c        80108077
2       sh      0       0       1       0       5.035   0.011   sleep   16384   801056e5        80105187        80107cd2        80106ed2        8010827c        80108077
3       p4test  0       0       2       0       1.940   0.051   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
4       p4test  0       0       3       0       1.928   0.142   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
5       p4test  0       0       3       0       1.923   0.096   run     12288
6       p4test  0       0       3       0       1.918   0.138   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
7       p4test  0       0       3       0       1.913   0.130   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
8       p4test  0       0       3       0       1.907   0.086   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
9       p4test  0       0       3       0       1.902   0.152   sleep   12288   801056e5        80107db0        80106ed2        8010827c        80108077
```

Figure 2: This shows promotions changing the priority level of sleeping processes.

```
PID     Name    UID     GID     PPID    Prio    Elapsed CPU     State   Size    PCs
1       init    0       0       1       0       4.799   0.016   sleep   12288   801056e5
2       sh      0       0       1       0       4.789   0.020   sleep   16384   801056e5
3       tspin   0       0       2       1       0.990   0.347   run     12288
4       tspin   0       0       3       1       0.966   0.327   run     12288
5       tspin   0       0       3       1       0.958   0.329   runble  12288
6       tspin   0       0       3       1       0.951   0.310   runble  12288
7       tspin   0       0       3       1       0.924   0.300   runble  12288
8       tspin   0       0       3       1       0.907   0.301   runble  12288

PID     Name    UID     GID     PPID    Prio    Elapsed CPU     State   Size    PCs
1       init    0       0       1       0       5.180   0.016   sleep   12288   801056e5
2       sh      0       0       1       0       5.170   0.020   sleep   16384   801056e5
3       tspin   0       0       2       0       1.371   0.477   runble  12288
4       tspin   0       0       3       0       1.347   0.457   run     12288
5       tspin   0       0       3       0       1.339   0.459   runble  12288
6       tspin   0       0       3       0       1.332   0.440   runble  12288
7       tspin   0       0       3       0       1.305   0.430   run     12288
8       tspin   0       0       3       0       1.288   0.431   runble  12288
```

Figure 3: This shows promotion changing the priority level at a 1 second interval.

```
Ready List Processes:

0: Empty.

1: (7, 270)-> (4, 250)-> (8, 230)

2: (6, 1990)

PID     Name    UID     GID     PPID    Prio    Elapsed CPU     State   Size    PCs
1       init    0       0       1       0       10.809  0.015   sleep   12288   801056e5        80105187        80107cd2        80106ed2        8010827c        80108077
2       sh      0       0       1       0       10.792  0.018   sleep   16384   801056e5        80105187        80107cd2        80106ed2        8010827c        80108077
3       p4test2 0       0       2       1       6.820   1.987   runble  12288
4       p4test2 0       0       3       1       6.798   3.892   run     12288
5       p4test2 0       0       3       1       6.792   1.909   runble  12288
6       p4test2 0       0       3       2       6.735   0.010   runble  12288
7       p4test2 0       0       3       1       6.730   3.870   run     12288
8       p4test2 0       0       3       1       6.715   1.910   runble  12288

Ready List Processes:

0: Empty.

1: Empty.

2: (7, 2000)-> (8, 1990)-> (3, 1990)-> (5, 1990)
4: new prio 1

PID     Name    UID     GID     PPID    Prio    Elapsed CPU     State   Size    PCs
1       init    0       0       1       0       11.357  0.015   sleep   12288   801056e5        80105187        80107cd2        80106ed2        8010827c        80108077
2       sh      0       0       1       0       11.340  0.018   sleep   16384   801056e5        80105187        80107cd2        80106ed2        8010827c        80108077
3       p4test2 0       0       2       2       7.368   2.157   runble  12288
4       p4test2 0       0       3       1       7.346   4.172   run     12288
5       p4test2 0       0       3       2       7.340   2.089   runble  12288
6       p4test2 0       0       3       2       7.283   0.100   runble  12288
7       p4test2 0       0       3       2       7.278   4.080   runble  12288
8       p4test2 0       0       3       2       7.263   2.080   run     12288

Ready List Processes:

0: Empty.

1: Empty.

2: (6, 1860)-> (7, 1880)-> (8, 1870)-> (3, 1870)

PID     Name    UID     GID     PPID    Prio    Elapsed CPU     State   Size    PCs
1       init    0       0       1       0       11.869  0.015   sleep   12288   801056e5        80105187        80107cd2        80106ed2        8010827c        80108077
2       sh      0       0       1       0       11.852  0.018   sleep   16384   801056e5        80105187        80107cd2        80106ed2        8010827c        80108077
3       p4test2 0       0       2       2       7.880   2.257   run     12288
4       p4test2 0       0       3       1       7.858   4.682   run     12288
5       p4test2 0       0       3       2       7.852   2.189   runble  12288
6       p4test2 0       0       3       2       7.795   0.200   runble  12288
7       p4test2 0       0       3       2       7.790   4.180   runble  12288
8       p4test2 0       0       3       2       7.775   2.190   runble  12288

Ready List Processes:

0: Empty.

1: Empty.

2: (5, 1770)-> (6, 1750)-> (7, 1770)-> (8, 1760)
```

Figure 4: The processes at the top have a low budget, and are dropped to the next lowest list. We can see their new budgets.

info. The active process table shows that `RUNNING` and `SLEEPING` processes are being demoted as well as the `RUNNABALE` processes on the `ready` lists.

Result: Test 3 `PASSES`.

0.4.4 For (`MAX == 2`)

1. *Show that the Scheduler always selects the first process on the highest non-empty priority queue.*
   Figure 2 shows that process 3 is always at priority 0, and consistently has a higher CPU time than the other processes. Figures 5 & 6 also demonstrate the first process taken off of a ready list, no matter what level it is at.
   This sub-test `PASSES`.

2. *Show that promotion correctly moves the processes on the ready lists (except highest priority list) up to the next highest priority list, and maintains queue status.*
   This test was performed by running `tspin` to make some new processes that just run forever, and hitting `ctrl-R` to track changes to the ready lists. The promotion timer was set to a second, and the budget was set too high to demote any processes during the test (see Figures 5 & 6).
   This sub-test `PASSES`.

3. *Show that demotion correctly moves processes to the next lower priority list (if one exists) when the processes budget is used up.*
   Test 3 was performed while `MAX` was set to 2, and [Figure 4] properly demonstrates how the process is demoted after its budget is used, and the budget is reset.
   This sub-test `PASSES`.

Result: All sub-tests pass; Test 4 `PASSES`.

```
Ready List Processes:
0: Empty.
1: Empty.
2: (4, 1999834)-> (3, 1999870)-> (8, 1999870)-> (5, 1999840)
Ready List Processes:
0: Empty.
1: (8, 1999760)-> (5, 1999731)-> (7, 1999750)-> (6, 1999740)
2: Empty.
Ready List Processes:
0: Empty.
1: (4, 1999674)-> (3, 1999709)-> (8, 1999710)-> (5, 1999681)
2: Empty.
```

Figure 5: We can follow PIDs to see when promotion occurs and the order of the queues

```
Ready List Processes:
0: Empty.
1: Empty.
2: (5, 1999870)-> (4, 1999860)-> (3, 1999890)-> (6, 1999880)
Ready List Processes:
0: Empty.
1: (8, 1999640)-> (7, 1999630)-> (5, 1999610)-> (4, 1999600)
2: Empty.
Ready List Processes:
0: Empty.
1: (5, 1999510)-> (4, 1999500)-> (3, 1999530)-> (6, 1999520)
2: Empty.
Ready List Processes:
0: (5, 1999340)-> (4, 1999330)-> (3, 1999360)-> (6, 1999350)
1: Empty.
2: Empty.
```

Figure 6: The queue remains in the same order upon promotion switching them to another list.

0.4.5 For (`MAX == 6`)

1. *Show that the scheduler always selects the first process on the highest non-empty priority queue.*
   This test was done in mostly the same way as Test 4, just with the `MAX` define set to 6, thus making 7 lists. The promotion timer was set to a low number, the budget timer to a high one to prevent demotion, and `tspin` was ran. `ctrl-R` is hit repeatedly to watch the scheduler select the first available process (see Figure 7).
   This sub-test `PASSES`.

2. *Show that promotion correctly moves the processes on the ready lists (except highest priority list) up to the next highest priority list, and maintains queue status.*
   This test was done exactly like Test4-2, by running `tspin` to loop forever, and track the changes to the ready list as they are promoted by a quick timer (see Figure 8).
   This sub-test `PASSES`.

3. *Show that demotion correctly moves processes to the next lower priority list (if one exists) when the processes budget is used up.*
   Thus test was performed in almost an identical way to test 4-3, just with more ready lists. `p4test2` was run to fork multiple processes and run them, with the promotion timer set extremely high and the budget set to `500 ticks` so we can watch demotion happen quickly (see Figure 9).
   This sub-test `PASSES`.

Result: All sub-tests pass; Test 5 `PASSES`.

0.4.6 For (`MAX == 0`)

1. *Show that the scheduler operates as a single round robin queue, as it did in Project 3.*
   This test is nearly identical to Test 1. This test involved setting `MAX` to 0, so there will be one ready list. I then hit `ctrl-R` repeatedly to watch the processes move through the queue (see Figures 1 & 10).
   This sub-test `PASSES`.

2. *Show that promotion and demotion do not change round robin queue.* This test was performed by setting the promotion timer and the budget time to half-second values (`500 ticks`) so that they would execute quickly while we ran the test, and we could see that nothing changed, even though it can be seen that process budgets are being reset, so we know that promotions are also supposed to be occurring as well, every half-second (see Figure 11).
   This sub-test `PASSES`.

Result: All sub-tests pass; Test 6 `PASSES`.

0.4.7 The `setpriority()` system call

1. *Show that the priority is changed and budget reset when given a valid* `pid` *and* `priority`*.*
   For this test, I ran `tspin` in the background and used the `setpriority` user command to set a process to a new priority, and watched the budget reset along with it. I gave the promotion timer define a value of 50 million and the budget a similar value, so that no automatic promotions or demotions would occur during the testing (see Figure 12).
   This sub-test `PASSES`.

2. *Show that changing the priority of a process on a ready list, correctly moves the process to the list corresponding to the new priority.*
   This test is demonstrated by also Figure 12, process 6 is given a new priority of 0, and we can see in

```
$ tspin
Ready List Processes:
0: Empty.
1: Empty.
2: Empty.
3: Empty.
4: Empty.
5: (7, 1999960)-> (8, 1999961)-> (6, 1999950)-> (5, 1999930)
6: Empty.
Ready List Processes:
0: Empty.
1: Empty.
2: Empty.
3: Empty.
4: (6, 1999870)-> (5, 1999850)-> (4, 1999783)-> (3, 1999869)
5: Empty.
6: Empty.
Ready List Processes:
0: Empty.
1: Empty.
2: Empty.
3: (7, 1999820)-> (8, 1999821)-> (6, 1999810)-> (5, 1999790)
4: Empty.
5: Empty.
6: Empty.
Ready List Processes:
0: Empty.
1: (6, 1999700)-> (5, 1999680)-> (4, 1999613)-> (3, 1999699)
2: Empty.
3: Empty.
4: Empty.
5: Empty.
```

Figure 7: The scheduler running the first available process, while priority is updated quickly

```
Ready List Processes:
0: Empty.
1: Empty.
2: Empty.
3: Empty.
4: (4, 1999778)-> (5, 1999780)
5: (3, 1999990)-> (8, 1999990)
6: Empty.
Ready List Processes:
0: Empty.
1: Empty.
2: Empty.
3: (4, 1999708)-> (5, 1999710)
4: (3, 1999990)-> (8, 1999990)
5: Empty.
6: Empty.
Ready List Processes:
0: Empty.
1: Empty.
2: (6, 1999590)-> (7, 1999590)
3: (3, 1999990)-> (8, 1999990)
4: Empty.
5: Empty.
6: Empty.
```

Figure 8: This shows queue status being maintained as promotion occurs, and proper ready lists.

```
Ready List Processes:
0: Empty.
1: Empty.
2: Empty.
3: Empty.
4: Empty.
5: (5, 20)-> (3, 20)-> (4, 20)-> (8, 10)
6: Empty.
Ready List Processes:
0: Empty.
1: Empty.
2: Empty.
3: Empty.
4: Empty.
5: Empty.
6: (5, 430)-> (3, 430)-> (4, 430)-> (8, 420)
```

Figure 9: Processes being demoted after budget runs out, budgets reset.

```
Ready List Processes:
0: (8, 300)-> (5, 280)-> (7, 280)-> (4, 280)
Ready List Processes:
0: (7, 250)-> (4, 250)-> (9, 240)-> (6, 240)
Ready List Processes:
0: (4, 200)-> (9, 190)-> (6, 190)-> (5, 200)
Ready List Processes:
0: (4, 160)-> (9, 150)-> (6, 150)-> (5, 160)
Ready List Processes:
```

Figure 10: The processes are cycled through the single priority list, round-robin style.

```
Ready List Processes:

0: (5, 360)-> (6, 390)-> (7, 400)-> (8, 410)

Ready List Processes:

0: (7, 180)-> (8, 190)-> (4, 120)-> (3, 180)

Ready List Processes:

0: (7, 50)-> (8, 60)-> (4, 490)-> (3, 50)

Ready List Processes:

0: (7, 340)-> (8, 350)-> (4, 280)-> (3, 340)

Ready List Processes:

0: (4, 10)-> (3, 70)-> (5, 20)-> (6, 50)

Ready List Processes:

0: (4, 330)-> (3, 390)-> (5, 340)-> (6, 370)
```

Figure 11: Round robin queue unchanged, with budget being reset, this is when demotion would occur.

```
Ready List Processes:

0: Empty.

1: Empty.

2: (8, 499983681)-> (6, 499983676)-> (7, 499983671)-> (9, 499983694)

PID     Name    UID     GID     PPID    Prio    Elapsed CPU     State   Size    PCs
1       init    0       0       1       0       88.279  0.016   sleep   12288   801056e5
2       sh      0       0       1       0       88.262  0.050   sleep   16384   801056e5
5       tspin   0       0       4       0       79.788  61.379  run     12288
4       tspin   0       0       1       0       79.827  32.870  run     12288
6       tspin   0       0       4       2       79.776  16.324  runble  12288
7       tspin   0       0       4       2       79.769  16.329  runble  12288
8       tspin   0       0       4       2       79.762  16.319  runble  12288
9       tspin   0       0       4       2       79.746  16.306  runble  12288
setpriority 6 0
int i: 6.
int n: 0.
$
Ready List Processes:

0: (6, 499999020)

1: Empty.

2: (8, 499983681)-> (7, 499983671)-> (9, 499983694)
```

Figure 12: Process 6 is given a new priority of 0, and its budget is reset as well.

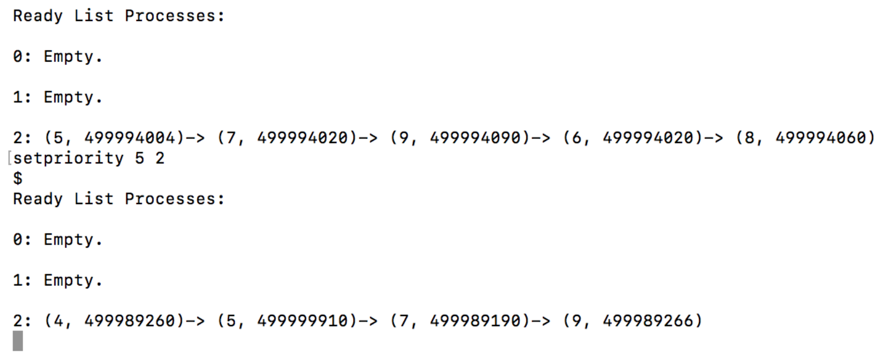

Figure 13: Process 5 is given the same priority, and we can see its budget has been reset, while maintaining its place on the queue.

the picture that it existed on priority list 2 before the command, and was moved to list 0 afterwards. This sub-test `PASSES`.

3. *Show that setting the priority of a process on a ready list to the same priority it already has, does not change it's position in the queue.*
   This test was performed by running `tspin` in the background (with `&`) and watching the queues, and then setting a process to the same priority, and watching the queues again. We can see that the budget is reset, but the process remains in the same position in the queue it was before (see Figure 13).
   This sub-test `PASSES`.

4. *Show that calling* `setpriority` *with an invalid* `pid` *or* `priority`*, returns a relevant error code.*
   For this test, I ran `tspin` in the background, and set the priority of a process to 100, which is well out of bounds. This returned an error message that the command had failed (see Figure 14). I then repeated the same step for the PID field, which returned an error message that the command had failed (see Figure 15).
   This sub-test `PASSES`.

Result: All sub-tests pass; Test 7 `PASSES`.

0.4.8 Updated Commands

1. *Show that* `ctrl-P` *correctly displays the process priority.*
   This test has been demonstrated across all of the tests, for proof any one of the provided figures up to this point can prove it works. I provided an extra image of `ctrl-P`, `ps`, and `ctrl-R` being used back-to-back-to-back to show that they can be cross-referenced to show the correct information (see Figure 16).
   This sub-test `PASSES`.

2. *Show that* `ps` *correctly displays the process priority.*
   This test is shown in Figure 16 to present the process priority information.
   This sub-test `PASSES`.

3. *Show that* `ctrl-R` *correctly displays all the ready lists, and the budget for each process on the list.*
   This test is also shown in almost all of the provided screenshots so far, and again for emphasis at the top of Figure 16.
   This sub-test `PASSES`.

```
$ tspin &
$
Ready List Processes:

0: Empty.

1: Empty.

2: (8, 499999620)-> (7, 499999610)-> (5, 499999565)-> (4, 499999620)
setpriority 5 100
Setpriority failed.
$
Ready List Processes:

0: Empty.

1: Empty.

2: (5, 499997345)-> (4, 499997410)-> (9, 499997400)-> (6, 499997351)

PID     Name    UID     GID     PPID    Prio    Elapsed CPU     State   Size    PCs
1       init    0       0       1       0       12.144  0.013   sleep   12288   801056e5
2       sh      0       0       1       0       12.132  0.041   sleep   16384   801056e5
5       tspin   0       0       4       2       8.846   2.965   run     12288
4       tspin   0       0       1       2       8.888   2.990   run     12288
6       tspin   0       0       4       2       8.831   2.959   runble  12288
7       tspin   0       0       4       2       8.816   2.924   runble  12288
8       tspin   0       0       4       2       8.769   2.920   runble  12288
9       tspin   0       0       4       2       8.749   2.910   runble  12288
QEMU: Terminated
hoover4@synchrotron:~/cs333/xv6-pdx$
```

Figure 14: setpriority fails to set process 5 to a priority of 100, because 100 is well above `MAX == 2`

```
$ tspin &
$
Ready List Processes:

0: Empty.

1: Empty.

2: (7, 499999750)-> (5, 499999710)-> (4, 499999750)-> (8, 499999750)
setpriority 100 1
Setpriority failed.
$
Ready List Processes:

0: Empty.

1: Empty.

2: (7, 499995090)-> (4, 499995090)-> (9, 499995092)-> (8, 499995095)

PID     Name    UID     GID     PPID    Prio    Elapsed CPU     State   Size    PCs
1       init    0       0       1       0       23.712  0.014   sleep   12288   801056e5
2       sh      0       0       1       0       23.696  0.038   sleep   16384   801056e5
5       tspin   0       0       4       2       17.603  5.884   runble  12288
4       tspin   0       0       1       2       17.646  5.907   run     12288
6       tspin   0       0       4       2       17.593  5.890   runble  12288
7       tspin   0       0       4       2       17.584  5.840   run     12288
8       tspin   0       0       4       2       17.540  5.835   runble  12288
9       tspin   0       0       4       2       17.522  5.838   runble  12288
QEMU: Terminated
hoover4@synchrotron:~/cs333/xv6-pdx$
```

Figure 15: setpriority fails to set process 100 to anything, because it doesn't exist.

```
Ready List Processes:

0: Empty.

1: Empty.

2: (6, 499998320)-> (7, 499998320)-> (9, 499998360)-> (5, 499998296)

PID     Name    UID     GID     PPID    Prio    Elapsed CPU     State   Size    PCs
1       init    0       0       1       0       14.687  0.014   sleep   12288   801056e5
2       sh      0       0       1       0       14.674  0.020   sleep   16384   801056e5
5       tspin   0       0       4       2       7.459   2.504   run     12288
4       tspin   0       0       1       2       7.479   2.521   runble  12288
6       tspin   0       0       4       2       7.452   2.490   runble  12288
7       tspin   0       0       4       2       7.446   2.490   runble  12288
8       tspin   0       0       4       2       7.441   2.460   runble  12288
9       tspin   0       0       4       2       7.383   2.440   run     12288
ps

PID     Name    UID     GID     PPID    Prio    Elapsed CPU     State   Size
1       init    0       0       1       0       21.470  0.014   sleep   12288
2       sh      0       0       1       0       21.457  0.046   sleep   16384
5       tspin   0       0       4       2       14.242  4.764   runble  12288
4       tspin   0       0       1       2       14.262  4.781   runble  12288
6       tspin   0       0       4       2       14.235  4.750   runble  12288
7       tspin   0       0       4       2       14.229  4.735   run     12288
8       tspin   0       0       4       2       14.224  4.713   runble  12288
9       tspin   0       0       4       2       14.166  4.700   runble  12288
10      ps      0       0       2       0       0.070   0.006   run     45056
$
```

Figure 16: In order: `ctrl-R`, `ctrl-P`, and `ps` command.

Result: All sub-tests pass; Test 8 `PASSES`.

All Tests PASS