Project 4
MLFQ Scheduler

1 Prerequisites

You **must** have all of Project 2 finished, except for the `time` command. You must also have completed the ready, running and sleep list implementation from project 3, this includes making sure that the relevant invariants of these lists are enforced.

Let the course staff know if this is not the case or if you have significant errors in your implementation *before you begin this project.*

2 Introduction

In the previous project, you rewrote the old `scheduler()` to use the ready list when searching for a `RUNNABLE` process instead of iterating through the process array. Your rewritten scheduler is still functionally equivalent to the old one: both are simple, round-robin schedulers, but the new one changes the complexity from $\mathcal{O}(n)$ to $\mathcal{O}(1)$. However, fairness is not addressed. In this project, you will implement a new "fairer" scheduler for xv6. You will become familiar with:

1. Implementing an MLFQ scheduler for xv6.
2. Implementing a new system call for setting process priority.
3. Implementing a new system call for getting process priority.

For this project, you will be using a new Makefile flag, `-DCS333_P4`, for conditional compilation. If you have errors anywhere outside of the Project 3 prerequisites, you can turn the relevant features off by changing `#ifdef CS333_P3` to `#ifdef CS333_P3_OFF` for any instances that you wish to disable. Please make a note of the exact Project 3 features that you turned off (if any) in the project report. Otherwise, we will grade your project assuming that you have implemented all of Project 3.

Do *not* use the `ptable.proc[]` structure when implementing this project except in `getprocs()`, `procdump()`, and free list initialization; use your lists instead.

3 Overview

In class, you learned about several approaches to scheduling processes. Each has its strengths and weaknesses. One approach was called "Muli-Level Feedback Queue" or MLFQ. The text explained the MLFQ algorithm using a periodic priority reset. You will implement a variation that uses a slightly different approach for preventing process starvation.

The approach that you will implement utilizes both "demotion", based on a *budget*, for fairness and "periodic promotion" for starvation prevention. **Warning:** the promotion strategy used here is not the same as described in the OSTEP text.

4 Process Priority

You will `#define` a constant `MAXPRIO` in `pdx.h` to be a number ≥ 0. Add to each process an associated `priority` in the range $[0 \ldots \text{MAXPRIO}]$ that will dictate the ready list to which the process belongs when in the `RUNNABLE` state. This means that there are $\text{MAXPRIO} + 1$ possible priorities for each process. We need to track a process' priority while it is not `RUNNABLE` so that we know where to put it if it later becomes `RUNNABLE` again.

Upon allocation, each process will have the same initial (default) priority value, the highest priority. The process priority value may be changed programatically during process execution via the `setpriority()` system call.

The highest priority in the system will be `MAXPRIO` with each value less than `MAXPRIO` being a successively lower priority, the lowest being 0. That is,

$$\forall P_i \in \{0, 1, 2, \ldots \text{MAXPRIO}\},$$
$$P_{MAXPRIO} > P_{MAXPRIO-1} > \ldots > P_0$$

4.1 Modifying the Ready List

You will need to add a new declaration for an array of ready lists in `struct ptable`:

```
static struct {
  struct spinlock lock;
  struct proc proc[NPROC];
#ifdef CS333_P3
  struct ptrs list[statecount];
#endif
#ifdef CS333_P4
  struct ptrs ready[MAXPRIO+1];
  uint PromoteAtTime;
#endif
} ptable;
```

where each index of the ready list corresponds to a priority queue in the MLFQ. You will need to modify your ready list code to now work with process priority. Observe that your invariant for the ready list will now change to the following: the ready lists contain all of the `RUNNABLE` processes in the system with each process in `ptable.ready`$[i]$ having a priority of i, $0 \leq i \leq \text{MAXPRIO}$.

Be careful when modifying the ready list insertion code in `kill()`, especially if you are using the old routine.

4.2 The `setpriority()` and `getpriority()` System Calls

The function prototypes are

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

where `priority` ranges from [0 . . . MAXPRIO]. The `setpriority()` system call will have the effect of setting the priority for an *active* process with PID `pid` to `priority` and resetting the *budget* to a default value (which could have a name like `DEFAULT_BUDGET`), which you must `#define` in `pdx.h`. Return an error if the values for `pid` or `priority` are not correct. This means that the system call is required to enforce bounds checking on the priority value; you *may not* assume that a user program only passes correct values.

Here is a simple way for a process to *set* its own priority:

```
rc = setpriority(getpid(), newPriority);
```

where `rc` is the return code from the system call and `newPriority` is an integer. If the value for the priority is invalid, leave the original priority for the process unchanged and return an error. Likewise, the budget for a process should not be changed unless the priority is correct.

For the `getpriority()` system call, the return value is the priority of the process or -1 on error. `getpriority()` should return the priority for the process that matches the PID provided and is not in the `UNUSED` state. If there is no process with the provided PID, or the process with the provided PID is in the `UNUSED` state, it is an error.

Here is a simple way for a process to *get* its own priority:

```
rc = getpriority(getpid());
```

where `rc` is the return code from the system call.

4.3 Periodic Priority Adjustment

We want a policy regarding adjusting priorities periodically. Consider this scenario:

> There are many processes in the system. One of the processes has the lowest priority while the remaining have the default priority. The process execution mix is such that the process with the lowest priority never gets to run.

The result is *starvation*, which the scheduler needs to avoid.

To address the problem of starvation, you will implement a promotion strategy. The strategy is to periodically increase the priority of all active processes by one priority level[1]; that is, the priority of all processes in the `RUNNABLE`, `SLEEPING`, and `RUNNING` states will periodically be adjusted. You will use the following approach:

1. Add a new field to the `ptable` structure. Make it an unsigned integer (`uint`) and call it `PromoteAtTime`. The value stored will be the *ticks* value at which promotion will occur. This value is the same for all processes so put it in the `ptable` structure not each process. While there is an overflow issue here, you will ignore it for this project. You'll need to initialize `PromoteAtTime` to `ticks + TICKS_TO_PROMOTE` in `userinit()`, because the scheduler will expect `PromoteAtTime` to be set to some sane value as soon as it starts running.
2. Create a constant `#define TICKS_TO_PROMOTE XXX`, where `XXX` is the number of ticks that will elapse before all the priorities are adjusted. Each time that the routine `scheduler()` runs, check to see if it is time to adjust priorities.
3. When the value of `ticks` reaches, or exceeds[2], `PromoteAtTime`:
   (a) Adjust the priority value for active[3] non-zombie processes to the next higher priority if not already at the highest priority.

[1]A process may not have a priority value outside of the range [0 . . . MAXPRIO]

[2]The scheduler does not run on every tick so it is possible for the ticks value to exceed the `PromoteAtTime` value. Be careful with your algorithm!

[3]See Project 2 for how "active" is defined.

(b) Change the priority queue for a process as appropriate. Put any adjusted processes on the back of the new queue. Do not move processes for which the priority was not adjusted.

(c) Set the value for `PromoteAtTime` to `ticks + TICKS_TO_PROMOTE`[4].

5 MLFQ Algorithm

Your algorithm is a variation of this algorithm where you will utilize a time *budget*, rather than basing priority assignment on the fraction of a time slice used.

Each time that the MLFQ algorithm runs, the process at the front of the highest priority non-empty queue will be selected to run. This means that each time the algorithm looks for a new process, the algorithm must start by checking the highest priority queue and only checking a lower queue if no higher priority jobs are available.

Approach:

1. Each process is assigned its own *budget*. This budget must be initialized to the default value (see Section 4.2) whenever a new process is allocated.
2. Each priority level has an associated FIFO queue. Each queue is serviced in a round robin fashion.
3. A newly created process is inserted at the end (tail) of the highest priority FIFO queue when it is moved from the `EMBRYO` to the `RUNNABLE` state.
4. At some point the process reaches the head of its 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 (i.e. transitions out of the `RUNNING` state), the budget is updated according to this formula:

   $$budget = budget - (time\_out - time\_in)$$

   If $budget \leq 0$ then the process will be *demoted* and placed at the tail of the next lower priority queue when it again reaches the `RUNNABLE` state. The budget value will be reset to the default.

   If the *budget* is not expired, the process will not be demoted and 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 priority level. Promoted processes are placed at the tail of the new queue. On promotion, set the process budget to the default value.

[4]The course staff find that initially setting the budget to 300 and `TICKS_TO_PROMOTE` to 3000 produces reasonable behavior your development. You may need to change these values to test all functionality.

6 Modified Commands

6.1 ps Command

The `ps` command must now report the priority level of each process. This will require that a new `priority` field be added to the `uproc` struct.

```
$ ps

PID     Name        UID        GID     PPID     Prio    Elapsed  CPU       State   Size
1       init         0          0       1       7       1.041    0.030     sleep   12288
2       sh           0          0       1       6       1.035    0.031     sleep   16384
3       ps           0          0       2       7       0.020    0.002     run     49152
```

6.2 control − p Console Command

The priority level of each active process will be displayed.

```
$
PID     Name        UID        GID     PPID     Prio    Elapsed CPU       State   Size    PCs
1       init        0          0       1        7       267.28  0.030     sleep   12288   8010546
2       sh          0          0       1        6       267.22  0.031     sleep   16384   8010546
```

6.3 control − r Console Command

You will modify the output to include priority and budget as follows:

Ready List Processes:

MAXPRIO: $(PID_{MAXPRIO1}, B_{MAXPRIO1}) \rightarrow (PID_{MAXPRIO2}, B_{MAXPRIO2}) \ldots \rightarrow (PID_{MAXPRIOn}, B_{MAXPRIOn})$

MAXPRIO-1: $(PID_{MAXPRIO-11}, B_{MAXPRIO-11}) \rightarrow (PID_{MAXPRIO-12}, B_{MAXPRIO-12}) \ldots \rightarrow (PID_{MAXPRIO-1m}, B_{MAXPRIO-1m})$

MAXPRIO-2: $(PID_{MAXPRIO-21}, B_{MAXPRIO-21}) \rightarrow (PID_{MAXPRIO-22}, B_{MAXPRIO-22}) \ldots \rightarrow (PID_{MAXPRIO-2p}, B_{MAXPRIO-2p})$

...

0: $(PID_{01}, B_{01}) \rightarrow (PID_{02}, B_{02}) \ldots \rightarrow (PID_{0q}, B_{0q})$

where PID_{ij} is the PID of the j^{th} process in the i^{th} ready list and where B_{ij} is the budget of the j^{th} process in the i^{th} ready list.

7 Discussion

You may not assume a fixed value for the number of priority queues. Instead, you will use a `#define` in `pdx.h` that is visible in both user[5] and kernel mode. Your scheduling algorithm is **required** to be flexible enough to adapt to wide variations in the number of queues. For example, if the number of queues is "1" then the algorithm will behave as a simple round robin scheduler. Your testing must account for the number of queues being "1", "3", "7", i.e. `MAXPRIO` values of "0", "2", and "6".

Determining the length of the promotion timer is tricky. There is no one, obvious value that works best in all cases. You will need to experiment with different values to see which one works best for you. This process is called *hand tuning*. In particular, your timer must be long enough so that you can properly test that your MLFQ algorithm and queue handling works properly. Your testing *must* show that your demotion and promotion strategies are working correctly. Note that you may need different timer values for different tests.

You will need at least one test that shows the correct priority level information for both the `ps` and `control-p` commands.

An example program that creates many children and causes them to periodically reset their priority is located on the course website. You are free to use or modify this program as you see fit for testing.

You will have to give some thought as to where to put the logic that updates the process budget and, when the budget expires, demotes the process and resets its budget. In order to accurately determine how many ticks of budget time have been used up, you'll need to do this bookkeeping when the process is transitioning out of the `RUNNING` state[6]. Check out the xv6 state transition diagram to see what functions correspond to these transitions. Additionally, you'll need different list management logic for different transitions, since a running process can transition to multiple states, each of which corresponds to a different destination list. Because of this, it is not appropriate to put all of the logic for budget management and demotion into one function (e.g., `sched()`). Some of this code will be the same for all transitions, so you are expected to place the common code in a function; but some of it will be different and **realizing this is key in getting this project right**.

8 Required Tests

All tests are defined in the accompanying rubric.

8.1 Hints/Suggestions

Test promotion, demotion, and scheduler functions separately. For example, you would not want processes to be demoted and/or promoted when you're trying to make sure that your scheduler selects the highest priority process. You can turn "off" promotion or demotion by setting the

[5]See `MAXPRIO` in `setpriority()` above.

[6]A process transitioning to the `ZOMBIE` state need not be demoted or promoted as a zombie can never run again.

relevant constants to a really high value. An alternate approach would be to develop new system calls for setting the default budget and promotion timer.

Using the elapsed CPU time feature from Project 2, you can demonstrate some features simultaneously by creating several infinitely looping processes at each priority level and then using control−p. Think about the elapsed CPU time of a starved process.

9 Before You Submit

As described in the *Survival Guide*, you are required to test that your submission works properly before submission. There is a `Makefile` target to help you. You are **required** to run "`make dist-test`" to verify that your kernel compiles and runs correctly, including all tests. A common error is to not put the name of new files into `runoff.list`. Once you have verified that the submission is correct, use "`make tar`" to build the archive file that you will submit. Submit your project report as a separate PDF document. **Failure to submit your report in PDF format will result in a grade of zero for the report. No exceptions.**