Project 4
MLFQ Scheduler

Prerequisites

You **must** have all of Project 2 finished, except for the `time` command. You must also have completed the ready, running and sleep lists from project 3, including ensuring 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.*

Introduction

In the previous project, you rewrote the old `scheduler()` routine 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. In this project, you will implement a new 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.

For this project, you will use the existing Makefile flag, `-DCS333_P3P4`, 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_P3P4` to `#ifdef CS333_P3P4_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 `Implementation` section of the project report. Otherwise, we will grade your project assuming that you've implemented all of Project 3. Do *not* use the `ptable.proc[]` structure when implementing this project; use your lists instead.

Overview

In class, you learned about several approaches to scheduling processes. Each had 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*) and "periodic promotion" (starvation prevention). This latter feature is the primary difference between the approach of the book and your implementation.

Process Priority

Each process will have an associated priority (as a uint) in the range $[0 \ldots \text{MAX}]$ that will dictate the ready list to which it belongs when in the `RUNNABLE` state. This means that there are $\text{MAX} + 1$ 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. That is,

$$\forall P_i \in \{0, 1, 2, \ldots \text{MAX}\},$$
$$P_0 > P_1 > P_2 > \ldots > P_{\text{MAX}}$$

Modifying the Ready List

You will need to change your declaration of the ready list in `struct StateLists` to the following:

```
struct StateLists {
  struct proc* ready[MAX+1];
  struct proc* free;
  struct proc* sleep;
  struct proc* zombie;
  struct proc* running;
  struct proc* embryo;
};
```

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 list contains all of the RUNNABLE processes of the system with each process in `pLists.ready`[i] having a priority of i, $0 \leq i \leq$ MAX. Be careful when modifying the ready list insertion code in `kill()`, especially if you are using the old routine.

The `setpriority()` System Call

The function prototype is

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

where `value` ranges from [0...MAX]. The system call will have the effect of setting the priority for an *active* process with PID *pid* to *value* and resetting the *budget* to the default value. Return an error if the values for `pid` or `value` 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 valid values.

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

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

where `rc` is the return code from the system call and `value` is an integer.

Periodic Priority Adjustment

We want some policy about 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`[2] 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 it is set in the `ptable` structure, not each process.

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

[2] Since we have more than one CPU, it is possible for one CPU to be running the adjust code while the process we are adjusting is running on another CPU.

2. Create 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, check 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. Put any adjusted processes on the back of the queue. Do not move processes for which the priority was not adjusted or you risk introducing starvation.

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

MLFQ Algorithm

Your algorithm is a variation of this algorithm where you will utilize a time *budget* instead of basing your decision on the fraction of a time slice used[3].

Each time that the MLFQ algorithm runs, the process at the front of the highest priority queue will be selected to run. This means that each time the algorithm looks for a new process that 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*.

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 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 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 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.

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. You must decide how best to deal with the *budget* value on promotion. Be sure to document your reasoning in the project report.

[3] xv6 is not set up to capture timing information at granularity less than a time slice. This could be fixed with not too much difficulty, but our algorithm avoids this necessity,

Modified Commands

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       0       1.41    0.3     sleep   12288
2       sh           0          0       1       1       1.35    0.3     sleep   16384
3       ps           0          0       2       1       0.2     0.2     run     49152
```

ctrl – 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       0       267.28  0.3     sleep   12288   80105469 8
2       sh           0          0       1       0       267.22  0.3     sleep   16384   80105469 8
```

ctrl – r Console Command

You will modify the output to be formatted as follows:

Ready List Processes:
0: $(PID_{01}, B_{01}) \rightarrow (PID_{02}, B_{02}) \ldots \rightarrow (PID_{0n}, B_{0n})$

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

2: $(PID_{21}, B_{21}) \rightarrow (PID_{22}, B_{22}) \ldots \rightarrow (PID_{2p}, B_{2p})$
...
MAX: $(PID_{MAX1}, B_{MAX1}) \rightarrow (PID_{MAX2}, B_{MAX2}) \ldots \rightarrow (PID_{MAXq}, B_{MAXq})$

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

Discussion

You may not assume a fixed value for the number of priority queues. Instead, you will use a define that is visible in both user[4] 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[5]. Your testing must account for the number of queues being "1", "3", "7".

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.

[4] See "MAX" in `setprioity()` above.
[5] It would be quite inefficient though.

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 properly.

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 here. You are free to use this program as you see fit for testing.

Required Tests

You must have tests to ensure that the following is correctly achieved for this project.

1. Demonstrate that round robin scheduling is still enforced by the MLFQ for a single priority level.
2. The MLFQ scheduler always selects the first process on the highest non-empty priority queue.
3. All active processes in the system are moved up by one queue when process promotion occurs. For `RUNNABLE` processes, processes at priority level 0 remain unchanged, processes at priority level 1 are appended to the end of processes at priority level 0 while processes at priority level i move to priority level $i-1$ for $1 \leq i \leq \text{MAX}$ with the process *ordering* in each queue being maintained (i.e., that you do not break the round-robin in each level).
4. When an active process uses up its budget, it gets demoted by one level and put into the correct queue for a `RUNNABLE` process. Note that processes at level MAX remain unchanged.
5. The `setpriority()` system call correctly sets the process of a *non-RUNNABLE* process when a valid `pid` and `priority` are passed in.
6. The `setpriority()` system call sets the process of a `RUNNABLE` process correctly when a valid `pid` and `priority` are passed in *AND* moves it to the end of the queue for the new priority level. If the priority for a process is set to the existing priority for the process then no changes occur to either the process priority or its place in the queue.
7. The `setpriority()` system call returns a relevant error code if an invalid `priority` is passed in. Your test program(s) must correctly process the return code.
8. The `setpriority()` system call returns a relevant error code if an invalid `pid` is passed in, where an invalid `pid` is either a process identifier not associated with an active process.
9. The MLFQ scheduler works correctly for $\text{MAX} = 1, 3, 7$.
10. The ctrl$-$p command correctly displays the process priority.
11. The `ps` command correctly displays the process priority.
12. The ctrl$-$r command correctly displays the correct processes in each priority level, along with their budget as shown above.

Hints/Suggestions

Test each functionality of the scheduler independently. 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 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. This is more difficult than it may appear, however.

If you have the elapsed CPU time feature correct from Project 2, you can demonstrate Requirements 1 and 2 simultaneously by creating several infinitely looping processes at each priority level and then using ctrl – p. Think about the elapsed CPU time of a starved process.