

Project Four Report  
Introduction to Operating Systems  
Spring 2017

Andy Keene

## Description

For this assignment I learned about improved process scheduling by implementing a Multilevel Feedback Queue (MLFQ) as the scheduling algorithm; this included hand-tuning constants which dictate the promotion/demotion intervals of individual processes and making consequential OS design decisions.

## Deliverables

The following features were added to xv6:

- A new system call `setpriority()` was added that sets the priority the process belonging to the given `pid` to the new, given, `priority` if it is in the range in the range of `[0...MAX]`.

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

- Console control sequences were updated to display information about the processes priority and/or budget.

- `ctrl-r`, was updated to display process PIDs and their associated budgets for each priority queue in order of head to tail. The new output displays as:

Ready List Processes:

0:  $(PID_{01}, B_{01}) \rightarrow (PID_{02}, B_{02}) \dots \rightarrow (PID_{0n}, B_{0n})$

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

2:  $(PID_{21}, B_{21}) \rightarrow (PID_{22}, B_{22}) \dots \rightarrow (PID_{2p}, B_{2p})$

...

MAX:  $(PID_{MAX1}, B_{MAX1}) \rightarrow (PID_{MAX2}, B_{MAX2}) \dots \rightarrow (PID_{MAXq}, B_{MAXq})$

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.

- `ctrl-p` was updated to display the priority of each process under the column *Prio*. The new output displays as:

```
$
PID      Name      UID      GID      PPID      Prio      Elapsed CPU      State      Size      P
1         init         0         0         1         0        267.28  0.3      sleep    12288    8
2         sh          0         0         1         0        267.22  0.3      sleep    16384    8
```

- The user command `ps` was updated to display information about the priority of each processes under the column *Prio*. To support this the `uproc struct` was given a new `priority` field, and the system call `getprocs()` was updated to fill in the `uproc` priority. The new output displays as:

```
$ 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
```

- Process scheduling is now determined by a Multilevel Feedback Queue (MLFQ) using priority queues;  
\*see MLFQ in the *Implementation* for algorithms details.

## Implementation

### Setpriority() system call

Using the process outlined in project one, the `setpriority()` system call was implemented by adding: a user-side header in `user.h` (line 39); creating a system call number in `syscall.h` (line 32); updating the system call jump table, the system call name table, and adding a kernel side header in `syscall.c` (lines 109, 142, 177); a user-side stub in `usys.S` (line 40); and implementation in `sysproc.c` (lines 176-185).

The implementation in `sysproc.c` pulls the pid and priority arguments off the stack, returning -1 on failure, and on success calls the kernel side `setpriority` function in `proc.c` (lines 1219-1242) whose prototype is defined in `defs.h` (line 127). `setpriority` is implemented in `proc.c` since process state lists must be accessed in order to update a processes priority. `setpriority` verifies for the given priority  $i$ ,  $0 \leq i \leq MAX$  before looking for the process on the EMBRYO, RUNNABLE, RUNNING, and ZOMBIE state lists (line 1230) while holding the lock where if the priority is not in a valid range -1 is returned (lines 1224-1225) - the helper function `findprocess()` in `proc.c` lines 113-143 was modified to traverse the ready lists when looking for a given PID. It is believed that since processes *not* in the UNUSED state have a valid PID and priority, and that `kill` allows such calls to kill processes in these states, that `setpriority()` should also support this - thus all lists with the exception of free are searched (it is also argued that this increases code reusability and simplicity). If the process is found its new priority is set and it is placed back onto the appropriate list, meaning if it *was* on one of the priority queues, it is transitioned to the queue of its new priority. If the process is not found `setpriority` returns -1 (lines 1223, 1240), which is passed back to the calling process by the system call. The system call prototype is defined as:

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

### User command ps Modifications

To support displaying process priority information in the `ps` user command, a priority field was added to the `uproc struct` (`uproc.h` line 11). The system call `getprocs()` now fills in the `uproc struct` priority field (`proc.c` line 1201), which was chosen to be of type `uint` to match the type of priority in the process structure (see MLFQ implementation). Additionally, the user command was edited to display *priority* in the field header (`ps.c` line 30) and each processes priority value in-line (line 32). \*See *Deliverables* section for the implemented display format.

### Console control sequence modifications

- `ctrl-p` modifications. The console control sequence `ctrl-p` kernel side function `procdump()` was modified to print *priority* as a field header (`proc.c` line 1158) and each processes priority value in-line (`proc.c` line 1116). \*See *Deliverables* section for the implemented display format.
- `ctrl-r` modifications. The kernel side function `dofreelist()` invoked by `ctrl-r` was modified to iterate over each priority queue, or ready list, 0, 1, 2, ...,  $MAX$  and from head to tail display each processes PID and *budget* as an ordered pair (PID, *budget*) (`proc.c` lines 1320-1339). \*See *Deliverables* section for the implemented display format.

## MLFQ

The algorithm used by the MLFQ to determine process scheduling and manage the priority queues. If  $P_i$  is the priority of queue  $i$ , then the priority of the RUNNABLE queues is ordered as  $P_0 > P_1 > \dots > P_{MAX}$ . The algorithm works as follows:

1. The head of the highest non-empty priority queue is always the next process to be run.
2. Each priority level has an associated FIFO queue which is serviced in a round robin fashion.
3. A newly created process is inserted at the end (tail) of the highest priority FIFO queue ( $P_0$ ) when it is moved from the EMBRYO to the RUNNABLE state.
4. If the process exits before the time slice expires, it leaves the system.
5. Each process is assigned its own budget, where 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, not exceeding the maximum queue, and the budget value is reset.

If the *budget* is not expired, the process will be placed at the tail of its current priority queue when it again reaches the RUNNABLE state.

6. Periodically a *promotion timer* will expire. The expiration of this timer will cause each process to be *promoted* one level and for its budget to be *reset*. If the process is currently at the maximum priority, it will retain its ordering in regards to the original queue at the time of promotion. If a promoted process is not at the highest priority queue it is placed at the tail of the new, higher priority queue, such that  $P_{new} = P_{original}-1$ .

All transitions to and from the ready list are now updated to use their corresponding priority queue (i.e. the ready *lists*) instead.

To support the implementation of a MLFQ based scheduler the ready list contained within `StateLists` struct was modified to be an *array* of lists of size MAX+1 (proc.c line 16). The index  $i$  of the ready list represents the priority of a priority queue,  $P_i$ . Here-forth, priority queue is used interchangeably with  $a$  index of the ready list. The following modifications were made to core components of the process structure, state list transitions and scheduler to implement the MLFQ algorithm (all line numbers refer to proc.c unless noted otherwise):

- Added constants used by the MLFQ algorithm:
  - TPS, defined as 100, represents the number of ticks-per-second in the xv6 system (types.h line 6) so that other constants may be defined based on it.
  - The promotion timer interval uses TICKS\_TO\_PROMOTE defined as 11\*TPS or 11 seconds (proc.h line 7).
  - The maximum, or default, budget used is DEAULT\_BUDGET defined as 3\*TPS or 3 seconds (proc.h line 8).
  - MAX is defined as 5 and represents the lowest priority queue, or alternatively the maximum index of the ready list (types.h line 5) resulting in a number of MAX+1 queues. MAX is defined in types.h so that it is visible in both user and kernel space.

TICKS\_TO\_PROMOTE, DEAULT\_BUDGET, and MAX are all defined in proc.h since these should only be visible by the kernel and are inherently only needed by functions in proc.h.

The reasoning supporting setting the promotion timer to 11 seconds and the budget to 3 seconds is that the promotion timer should be greater than an average number of CPU-bound running processes, otherwise, no processes will be demoted in-between intervals, resulting in a MLFQ that acts as a single ready list. Testing results agreed with these hand picked values. It should be noted with limited granularity and heuristics that these constants will not optimally support *all* scenarios.

- Structure changes and field additions:
  - The fields `int budget` and `uint priority` were added to the process structure `proc struct` to enable a priority and budget per process (lines 83-84). `budget` was chosen as type integer so that a negative value would indicate that the budget has been used up, and `priority` was chosen as type unsigned integer since priority in our case is always non-negative - unsigned supports a higher theoretical maximum.
  - The ready state list was altered to be an array of size `MAX+1` supporting a maximum, or lowest priority of `MAX` (`proc.c` line 16).
  - The field `uint PromoteAtTime` was added to the process table structure to track the next time a promotion or priority boost must occur in terms of system ticks (lines 30). This field is initialized to `ticks + TICKS_TO_PROMOTE` during `userinit()` (line 473).
- The following functions were modified to support state transitions and initialization with priority:
  - `userinit()` (line 505) and `fork()` (line 583) now transition a process from the embryo state to its current priority. `allocproc()` now sets a process's budget to `DEFAULT_BUDGET` and its priority to 0 (lines 420-421). Thus, each process begins on priority queue 0, or  $P_0$ .
  - `exit()` now calls the helper function `abandonchildren()` for each priority queue of the ready list (lines 668-670). This helper function was described previously in Project Report 3.
  - `sched()` now updates both the CPU time and the budget of a process. The budget is updated according to
 
$$budget = budget - (time\_out - time\_in)$$
 (lines 924-926) where  $time\_out - time\_in = cpu\_time\_used$ . If the budget of the process has been used (i.e.  $budget \leq 0$ ) the budget is reset and its priority is demoted *iff* its current priority is less than `MAX` (933-934); otherwise it remains at priority `MAX`. Upon demotion if the process is currently in the `RUNNABLE` state it is transitioned to the new corresponding priority queue (lines 929-932).
  - `yield()` now appends the running process to its current priority queue before calling `sched()` (line 954).
  - `wakeup1()` now appends each process from the sleep list to its corresponding priority queue (line 1053).
  - `kill()` now appends any killed process found on the sleep list to the priority list corresponding to its held priority value (line 1111). Not changing the process's priority introduces a worst case time delay of  $MAX * TICKS\_TO\_PROMOTE$  in the freeing of killed process memory - found acceptable by the OS author on grounds that it instead introduces no delay into the natural scheduled ordering of higher priority processes.
  - The helper function `hasChildren()` used by `kill()` was updated to check all priority queues (i.e. the ready lists) when looking for children (lines 182-189).
  - `findprocess()` used by `kill()` (and now `setpriority()`) was modified to look for the process of the given PID on each of the ready lists, or priority queues (lines 113-143). \*Previously noted.

- With the preceding implementations support, the scheduling algorithm in `scheduler()` was modified to check for promotion time expiration and update the priority queues accordingly, and to select the process on the head of the highest non-empty priority queue to be run.

- Prior to making any scheduling decision, `scheduler()` now checks to see if the promotion timer, `PromoteAtTime`, has expired and calls a new helper function `priorityPromotion()` to handle the promotion routine (lines 842-845) before updating the promotion timer to `ticks + TICKS_TO_PROMOTE`. `priorityPromotion()` (lines 354-375) iterates over the priority queues from  $i, 0, \dots, MAX - 1$  and appends the head of  $P_{i+1}$  to  $P_i$  using the new helper function `appendToQueue()` (lines 321-336) - `appendToQueue()` is implemented the same as `appendToStateList()` but does not nullify the last pointer or set the process state. The budget of each process on the merged  $P_i + P_{i+1}$  is then reset with each priority set to  $i$  by calling the helper function `setPrioBudget()` (lines 343-352). It should be noted that for each process on the given state list `setPrioBudget()` moves the priority up one if it is not currently 0 (i.e. the highest priority) and sets each budget to `DEFAULT_BUDGET`.  $P_{i+1}$  is then nullified before repeating. Before exiting, `priorityPromotion()` calls `setPrioBudget()` on the sleep and running lists since these too must be updated.

This means that with the exception of `queue0` all list transitions occur in  $O(1)$  time, where the entirety of budget and priority updates occur in  $O(n)$  time where  $n$  is number of processes on ready, sleep, and running lists. It should also be noted that by the algorithm budgets of  $P_0$  will be reset; this is based on the reasoning that not resetting their priority would increase the changes of their demotion in between subsequent priority promotions. Simply changing `.ready[i]` on line 364 to `.ready[i+1]` in a future modification could easily maintain the original budgets of  $P_0$  during priority promotions.

- After handling the promotion logic, the scheduler now iterates over the heads of the priority queues from  $P_0$  to  $P_{MAX}$ , attempting to find a process to run (lines 847-851). If a process is found, the flag `found_proc` is set to communicate the process must be run (lines 838, 849, 854). Since the priority queues are searched starting from  $P_0$ , only the head of the first non-empty queue is run, and all other transitions `append` a process to its corresponding priority queue we are ensured that round robin is maintained for a single priority queue, and that the front process on the highest priority queue will always be scheduled.

## Testing

`findProc()` used by the function `checkProcs()` (proc.c 1119-1149) which verifies the list invariant that each process is on one and only one list when `DEBUG` is defined, was modified to look for, and count, the given process on the new priority queues. `DEBUG` was enabled during the duration of some tests to help ensure the invariant still holds.

### Required Tests

For reference, the required tests are listed as follows.

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 `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.

### Test 1 - Demotion, `ctrl-p`, `ctrl-r`, `ps` (Requirements 4, 10, 11, 12)

For this test we will verify `ctrl-p`, `ctrl-r` and `ps` display the correct priority for each *active* process and that when an *active* process uses up its allotted budget it is demoted by one level and put into the correct queue for a *runnable* process. To support this test `TICKS_TO_PROMOTE` will be changed to 40 seconds), `DEFAULT_BUDGET` to 3 seconds, and `MAX` to 3 so that we have time to see outputs before any process is demoted, and so that process demotions are not interrupted by a priority boost. The steps of this test are as follows:



- Immediately after xv6 boots, we will begin by executing `ps`, pressing `ctrl-p` and pressing `ctrl-r`. Since we know `sh` and `user init` could not have run for 5 seconds, and they are our only *active* processes on boot, they must have a priority of 0 (defined as highest, and given at allocation). Thus, with the exception of `ps` displaying itself and the PCs displayed by `ctrl-p`, we expect all output processes and corresponding priorities to match between `ctrl-p` and `ps`; where `ctrl-r` will be empty since no processes are running/runnable.
- Next we will execute the user program `test` with `test &` to regain shell control. `test` will create 6 infinitely looping processes by calling `inf_loop()`. We create this many so that when two are running we may still see others on the ready list. Since each process has a budget of 4 seconds, we expect that when running in round robin *if* demotion occurs correctly, every 6\*5 seconds or so the all RUNNABLE/RUNNING processes will have used up their budget and be demoted to the next level. We will press `ctrl-r`, `ctrl-p` and run `ps` periodically to see the processes on each priority as they make their way down the MLFQ. We have demonstrated other the other features of `ctrl-p` and `ps` in past reports (1-3) and consequently are only concerned with the priority information matching; we will ignore `ps` on it's own output since it will `exit()` before any other control feature is able to display information. Additionally, we expect that since CPU time determines a processes allotted time at each priority that at level  $i$ ,  $(i) * \text{DEFAULT\_BUDGET} \leq \text{CPU} < (i + 1) * \text{DEFAULT\_BUDGET}$  (here CPU refers to CPU time in the `ctrl-p` and `ps` outputs).
- Lastly, once the processes reach queue MAX, or 3, we will continually press `ctrl-r`. We expect that since no promotion timer has yet to occur, due to its large value, that the budgets for these processes will be reset and they will remain on queue MAX.

```

sb: size 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$
$
PID   Name   UID    GID    PPID   Prio   Elapsed CPU   State   Size   PCs
2     sh     0      0      1      0      3.36   0.84   sleep  16384  80105929 80100aaf 80101fe9 801012af 80106af5 8010693b 80107ca6 80107aa1
1     init    0      0      1      0      3.41   0.83   sleep  12288  80105929 801054a7 80107736 8010693b 80107ca6 80107aa1

$ Ready List Processes:
0:
1:
2:
3:

$ ps
PID   Name   UID    GID    PPID   Prio   Elapsed CPU   State   Size
6     ps     0      0      2      0      0.03   0.82   run    45056
2     sh     0      0      1      0      7.09   0.11   sleep  16384
1     init    0      0      1      0      7.14   0.83   sleep  12288

```

Figure 1: Initial output

```

$ test &
$ Parent of Infiite Loops: 9

$ Ready List Processes:
0: (13,259) -> (14,264) -> (10,247) -> (9,244)
1:
2:
3:

$ ps
PID   Name   UID    GID    PPID   Prio   Elapsed CPU   State   Size
17    ps     0      0      2      0      0.06   0.01   run    45056
14    test   0      0      9      0      2.64   0.87   runble  16384
13    test   0      0      9      0      2.78   0.52   runble  16384
12    test   0      0      9      0      2.84   0.94   runble  16384
11    test   0      0      9      0      2.87   0.96   runble  16384
9     test   0      0      1      0      2.99   1.05   run    16384
10    test   0      0      9      0      2.94   1.03   runble  16384
2     sh     0      0      1      0      15.81   0.26   sleep  16384
1     init    0      0      1      0      15.86   0.83   sleep  12288

$
PID   Name   UID    GID    PPID   Prio   Elapsed CPU   State   Size   PCs
14    test   0      0      9      0      3.98   1.31   runble  16384
13    test   0      0      9      0      4.12   1.34   runble  16384
12    test   0      0      9      0      4.18   1.37   run    16384
11    test   0      0      9      0      4.21   1.39   run    16384
9     test   0      0      1      0      4.33   1.49   runble  16384
10    test   0      0      9      0      4.28   1.45   runble  16384
2     sh     0      0      1      0      17.15   0.30   sleep  16384  80105929 80100aaf 80101fe9 801012af 80106af5 8010693b 80107ca6 80107aa1
1     init    0      0      1      0      17.20   0.83   sleep  12288  80105929 801054a7 80107736 8010693b 80107ca6 80107aa1

```

Figure 2: Test start, Queue 0

```

Ready List Processes:
0:
1: (13,271) -> (10,269) -> (11,270) -> (14,271)
2:
3:

$ ps
  PID   Name   UID    GID   PPID   Prio  Elapsed CPU   State  Size
  21    ps      0      0      2      0    0.01  0.01    run   45056
  14   test     0      0      9      1   11.28  3.81   runble 16384
  13   test     0      0      9      1   11.42  3.79   runble 16384
  12   test     0      0      9      1   11.48  3.80   runble 16384
  11   test     0      0      9      1   11.51  3.80   runble 16384
  9    test     0      0      1      1   11.63  3.79    run   16384
  10   test     0      0      9      1   11.58  3.82   runble 16384
  2    sh      0      0      1      0   24.45  0.39   sleep 16384
  1    init     0      0      1      0   24.50  0.03   sleep 12288

$
  PID   Name   UID    GID   PPID   Prio  Elapsed CPU   State  Size  PCs
  14   test     0      0      9      1   12.01  4.31    run   16384
  13   test     0      0      9      1   12.95  4.30   runble 16384
  12   test     0      0      9      1   13.01  4.31   runble 16384
  11   test     0      0      9      1   13.04  4.31   runble 16384
  9    test     0      0      1      1   13.16  4.30   runble 16384
  10   test     0      0      9      1   13.11  4.32    run   16384
  2    sh      0      0      1      0   25.98  0.40   sleep 16384
  1    init     0      0      1      0   26.03  0.03   sleep 12288
                                80105929 80100aaf 80101fe9 801012af 80106af5 8010693b 80107ca6 80107aa1
                                80105929 801054a7 80107736 8010693b 80107ca6 80107aa1

```

Figure 3: Queue 1

```

Ready List Processes:
0:
1:
2: (10,216) -> (12,216) -> (11,216) -> (14,216)
3:

$
  PID   Name   UID    GID   PPID   Prio  Elapsed CPU   State  Size  PCs
  14   test     0      0      9      2   24.01  8.04   runble 16384
  13   test     0      0      9      2   24.15  8.03   runble 16384
  12   test     0      0      9      2   24.21  8.04    run   16384
  11   test     0      0      9      2   24.24  8.04   runble 16384
  9    test     0      0      1      2   24.36  8.03    run   16384
  10   test     0      0      9      2   24.31  8.05   runble 16384
  2    sh      0      0      1      0   37.18  0.42   sleep 16384
  1    init     0      0      1      0   37.23  0.03   sleep 12288
                                80105929 80100aaf 80101fe9 801012af 80106af5 8010693b 80107ca6 80107aa1
                                80105929 801054a7 80107736 8010693b 80107ca6 80107aa1

$ ps
  PID   Name   UID    GID   PPID   Prio  Elapsed CPU   State  Size
  26    ps      0      0      2      0    0.06  0.06    run   45056
  14   test     0      0      9      2   26.20  8.76   runble 16384
  13   test     0      0      9      2   26.34  8.75   runble 16384
  12   test     0      0      9      2   26.40  8.75   runble 16384
  11   test     0      0      9      2   26.43  8.76   runble 16384
  9    test     0      0      1      2   26.55  8.75   runble 16384
  10   test     0      0      9      2   26.50  8.76    run   16384
  2    sh      0      0      1      0   39.37  0.43   sleep 16384
  1    init     0      0      1      0   39.42  0.03   sleep 12288

```

Figure 4: Queue 2

```

$ Ready List Processes:
0:
1:
2:
3: (10,262) -> (12,261) -> (9,262) -> (13,262)

$
PID Name UID GID PPID Prio Elapsed CPU State Size PCs
14 test 0 0 9 3 29.97 10.00 runble 16384
13 test 0 0 9 3 30.11 10.00 runble 16384
12 test 0 0 9 3 30.17 10.01 run 16384
11 test 0 0 9 3 30.20 10.00 runble 16384
9 test 0 0 1 3 30.22 10.00 runble 16384
10 test 0 0 9 3 30.27 10.01 run 16384
2 sh 0 0 1 0 43.14 0.45 sleep 16384 00105929 00106aaf 00101fe9 001012af 00106af5 0010693b 00107cae 00107aa1
1 init 0 0 1 0 43.19 0.03 sleep 12288 00105929 00106aaf 00101fe9 001012af 0010693b 00107cae 00107aa1

$ ps
PID Name UID GID PPID Prio Elapsed CPU State Size
30 ps 0 0 2 0 0.02 0.02 run 45856
14 test 0 0 9 3 32.52 10.04 runble 16384
13 test 0 0 9 3 32.66 10.04 runble 16384
12 test 0 0 9 3 32.72 10.06 runble 16384
11 test 0 0 9 3 32.75 10.04 runble 16384
9 test 0 0 1 3 32.87 10.04 run 16384
10 test 0 0 9 3 32.82 10.06 runble 16384
2 sh 0 0 1 0 45.69 0.46 sleep 16384
1 init 0 0 1 0 45.74 0.03 sleep 12288

```

Figure 5: Queue 3

```

$ Ready List Processes:
0:
1:
2:
3: (12,28) -> (9,30) -> (13,30) -> (10,28)

Ready List Processes:
0:
1:
2:
3: (11,294) -> (14,294) -> (12,292) -> (9,294)

```

Figure 6: Remaining on queue 3 after budget expires

In figure 1, we see that initial outputs of `ctrl-p` and `ps`, and process priorities match with the exception of `ps`; we also see that `ctrl-r` is empty. Thus for the first step our expectations are met.

In figures 2-4 we see that the test begins, that all processes are demoted one queue at a time, and that our expectations for the output of `ctrl-r`, `ctrl-p`, and `ps` are met. Further, at each priority level,  $(i) * \text{DEFAULT\_BUDGET} \leq \text{CPU} < (i+1) * \text{DEFAULT\_BUDGET}$  indeed holds with a `DEFAULT\_BUDGET` of approx. 3 seconds. Thus the expectation for the second stage of our test have been met.

Lastly, figure 5 shows that when a process uses up it's budget at the MAX priority, 3, that the budget is reset and the process is placed back on the MAX priority queue; this is evident by an increased budget while remaining on queue MAX.

Since all of our expectations were met at each stage of the test, this test **PASSES**.

## Test 2 - Round Robin and Process Selection (Requirements 1, 2)

This test will demonstrate that round robin scheduling is still enforced for a single priority level and that the MLFQ scheduler selects the first process from the highest non-empty priority queue. To do this we will change the number of children `inf_loop()` creates to 20. We will also change the `DEFAULT\_BUDGET` to 2 seconds. The steps will then proceed as:

- Execute `text &` to regain shell access. We will then hold down `ctrl-r` to get ready list info as quickly as possible; with round robin scheduling on a single priority queue we expect that (similar to the Project 3 Report) with two CPUs unless the scheduler has made it through the entire list which is evident by a different budget value, if some line is A, D, E, J, H, K, P, Q, W and the next line starts with a K, then K must at least be followed by P, Q, W (since K didn't run P, Q and W must not have either, thus the ordering and budget for these remains) and P, Q, W must be followed by A, D, E and the two processes that *were* running, in some order since these ran and must have been inserted into the back of the list.
- We will then wait some time for all of the processes to settle down to the MAX (or lowest) priority queue. At this point we will execute `text &` again to create a new batch of processes to run. Since each process starts at the highest priority, these will begin at queue 0. We expect as we press `ctrl-r`

repeatedly, that only process from queue 0 will be selected since  $P_0 > P_{MAX}$ . This will be evident by a changing order of queue 0 with diminishing budgets over time, and the budgets on queue MAX remaining constant.

```

$ Parent of infinite loops is done creating children (4)
Ready List Processes:
0: (6,51) -> (7,54) -> (5,49) -> (9,60) -> (10,61) -> (11,62) -> (13,67) -> (12,65) -> (14,67) -> (15,70) -> (16,72) -> (17,73) -> (19,77) -> (18,75)
1:
2:
3:

Ready List Processes:
0: (13,61) -> (12,59) -> (14,61) -> (15,64) -> (16,66) -> (17,67) -> (19,71) -> (18,69) -> (8,51) -> (4,39) -> (6,44) -> (7,47) -> (5,42) -> (9,53)
1:
2:
3:

Ready List Processes:
0: (10,54) -> (11,55) -> (13,60) -> (12,58) -> (14,60) -> (15,63) -> (16,65) -> (17,66) -> (19,70) -> (18,68) -> (8,50) -> (4,38) -> (6,43) -> (7,46)
1:
2:
3:

Ready List Processes:
0: (8,50) -> (4,38) -> (6,43) -> (7,46) -> (5,41) -> (9,52) -> (10,53) -> (11,54) -> (13,59) -> (12,57) -> (14,59) -> (15,62) -> (16,64) -> (17,65)
1:
2:
3:

```

Figure 7: Initial process batch

```

Ready List Processes:
0:
1:
2:
3: (19,82) -> (13,82) -> (9,81) -> (8,81) -> (4,81) -> (5,81) -> (6,81) -> (7,81) -> (10,81) -> (11,81) -> (12,81) -> (14,81) -> (15,81) -> (16,81)

$ test &
$ Parent of infinite loops is done creating children (22)
Ready List Processes:
0: (32,89) -> (34,93) -> (35,94) -> (36,96) -> (33,91) -> (37,99) -> (22,66) -> (24,74) -> (25,76) -> (23,73) -> (27,82) -> (28,83) -> (29,83) -> (30,85)
1:
2:
3: (10,100) -> (11,100) -> (12,100) -> (14,100) -> (15,100) -> (17,1) -> (16,100) -> (18,100) -> (19,100) -> (13,100) -> (9,99) -> (8,99) -> (4,99) -> (5,99) -> (6,99) -> (7,99)

Ready List Processes:
0: (22,60) -> (24,60) -> (25,70) -> (23,67) -> (27,76) -> (28,77) -> (29,77) -> (30,79) -> (31,80) -> (26,73) -> (32,82) -> (34,86) -> (35,87) -> (36,89)
1:
2:
3: (10,100) -> (11,100) -> (12,100) -> (14,100) -> (15,100) -> (17,1) -> (16,100) -> (18,100) -> (19,100) -> (13,100) -> (9,99) -> (8,99) -> (4,99) -> (5,99) -> (6,99) -> (7,99)

Ready List Processes:
0: (25,70) -> (23,67) -> (27,76) -> (28,77) -> (29,77) -> (30,79) -> (31,80) -> (26,73) -> (32,82) -> (34,86) -> (35,87) -> (36,89) -> (37,93) -> (22,60)
1:
2:
3: (10,100) -> (11,100) -> (12,100) -> (14,100) -> (15,100) -> (17,1) -> (16,100) -> (18,100) -> (19,100) -> (13,100) -> (9,99) -> (8,99) -> (4,99) -> (5,99) -> (6,99) -> (7,99)

Ready List Processes:
0: (27,75) -> (28,76) -> (29,76) -> (30,78) -> (31,79) -> (26,72) -> (32,81) -> (34,85) -> (35,86) -> (36,88) -> (37,92) -> (22,59) -> (33,83) -> (24,66)
1:
2:
3: (10,100) -> (11,100) -> (12,100) -> (14,100) -> (15,100) -> (17,1) -> (16,100) -> (18,100) -> (19,100) -> (13,100) -> (9,99) -> (8,99) -> (4,99) -> (5,99) -> (6,99) -> (7,99)

```

Figure 8: Second process batch

In figure 1 we see that in the first three outputs that the scheduler did in fact run every process on the list before `ctrl-r` was able to print again (they both use the lock); this is evident by the decreased budget on all processes between the first, second and third prints. However, the general ordering described does in fact remain. In the last output, print 4, we see (8, 50) as the head of queue 0 where on the list immediately prior, print 3, we see (8, 50) towards the tail with (4, 38), (6, 43) and (7, 46) following. Since process 8 did not run between print 3 and 4, and is followed by the processes 4, 6, and 7 with unchanged budgets, we can conclude that our expectations for round robin scheduling are in fact met.

In figure 2, we see that after the first batch had reached queue 3 (MAX) before we executed `test &` for the second time. The new batch at queue 0 is then prioritized over the last batch at queue 3 during scheduling. Again we see round robin scheduling at queue 0 but more importantly we see that all processes on queue 3 have unchanged budgets (see PIDs to match them to batch one). Thus, figure 2 demonstrates the second stage of our test.

Because all expectations were met for each stage of this test, this test **PASSES**.

### Test 3 - `setpriority()` for RUNNABLE and non-RUNNABLE processes (Requirements 5, 6)

Here we will test that `setpriority()` correctly sets the priority number for both RUNNABLE and non-RUNNABLE processes and that when the priority is set for a RUNNABLE process that the process is

moved to the end of its new priority queue. To do this we will set `DEAULT_BUDGET` to 100 seconds and `PROMOTE_AT_TICKS` to 100 seconds so that processes stay at their set priority for the duration of the test. The stages of this test are as follows:

- Execute `test &`. This will be set to call `valid_setpriority()` (test.c lines 198-236) which creates a batch of 15 children that will spin at priority queue 0 since queue 0 is the inherited priority of any created processes and the `DEAULT_BUDGET` is greater than the time we will allow them to spin.
- The parent will then notify us it completed the first stage and create 5 more processes, where it will set the priority of these 5 children priority to MAX. Because the first stage filled queue 0 with spinning processes, the call to `valid_setpriority()` with a valid PID and priority will likely occur before the child first runs. `valid_setpriority()` will then take the child off queue 0 and place it upon queue MAX. The parent will notify us of it's success (through the return code of `valid_setpriority()`) and sleep for a moment. At this point we will press `ctrl-r` which was demonstrated to be correct in Test 1. Here we expect since `valid_setpriority()` was successful, and the child likely did not run, that it will be at the *end* of queue MAX with a full budget of 10,000.
- The parent will then kill all children it created thus far.
- Lastly the parent will create 3 more children who will immediately sleep. The parent will print the PID of the created child, sleep for a few seconds to allow the children to get to the SLEEPING state and let us press `ctrl-p`. At this point we expect to see it in the SLEEPING state with priority 0. The parent will then call `valid_setpriority(childPID, MAX)`; after being notified we will press `ctrl-p` again and now expect to the the priority of this child set to MAX - or 3 in this case.

```
sb: size 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ test &
$
Initial batch done

Set priority of child: 20 to 3. Return code: 0
Ready List Processes:
0: (17,9979) -> (18,9982) -> (19,9984) -> (5,9958) -> (6,9958) -> (7,9959) -> (4,9967) -> (8,9968) -> (9,9964) -> (10,9965) -> (11,9966) -> (12,9969) -> (13,9970) -> (15,9976)
1:
2:
3: (20,10000)

Set priority of child: 21 to 3. Return code: 0
Ready List Processes:
0: (18,9955) -> (19,9957) -> (5,9931) -> (7,9932) -> (6,9931) -> (8,9933) -> (9,9937) -> (10,9938) -> (11,9939) -> (4,9966) -> (12,9942) -> (13,9943) -> (15,9949) -> (14,9947)
1:
2:
3: (20,10000) -> (21,10000)

Set priority of child: 22 to 3. Return code: 0
Ready List Processes:
0: (7,9900) -> (6,9909) -> (8,9901) -> (9,9905) -> (10,9906) -> (11,9907) -> (12,9910) -> (4,9960) -> (13,9911) -> (15,9917) -> (14,9915) -> (16,9918) -> (17,9919) -> (18,9922)
1:
2:
3: (20,10000) -> (21,10000) -> (22,10000)

Killing children
```

Figure 9: Runnable priorityset()

```

$ $ killed first batch
Created sleeping child: 25
Ready List Processes:
0:
1:
2:
3:

PID   Name   UID   GID   PPID   Prio   Elapsed CPU   State   Size   PCs
25    test    0     0     4     0     2.47  0.14   sleep  16384  80105929 80107814 8010693b 80107ca6 80107aa1
4     test    0     0     1     0     11.39 0.50   sleep  16384  80105929 80107814 8010693b 80107ca6 80107aa1
2     sh      0     0     1     0     12.97 0.08   sleep  16384  80105929 80100aaf 80101fe9 801012af 80106af5 8010693b 80107ca6 80107aa1
1     init    0     0     1     0     13.03 0.03   sleep  12288  80105929 801054a7 80107736 8010693b 80107ca6 80107aa1

Set priority of child: 25 to 3. Return code: 0

PID   Name   UID   GID   PPID   Prio   Elapsed CPU   State   Size   PCs
25    test    0     0     4     3     7.05  0.40   run    16384  80105929 80107814 8010693b 80107ca6 80107aa1
4     test    0     0     1     0     16.77 0.67   sleep  16384  80105929 80107814 8010693b 80107ca6 80107aa1
2     sh      0     0     1     0     18.35 0.08   sleep  16384  80105929 80100aaf 80101fe9 801012af 80106af5 8010693b 80107ca6 80107aa1
1     init    0     0     1     0     18.41 0.03   sleep  12288  80105929 801054a7 80107736 8010693b 80107ca6 80107aa1

Created sleeping child: 26

PID   Name   UID   GID   PPID   Prio   Elapsed CPU   State   Size   PCs
25    test    0     0     4     3     19.91 0.61   sleep  16384  80105929 80107814 8010693b 80107ca6 80107aa1
26    test    0     0     4     0     0.90  0.00   sleep  16384  80105929 80107814 8010693b 80107ca6 80107aa1
4     test    0     0     1     0     19.83 0.81   sleep  16384  80105929 80107814 8010693b 80107ca6 80107aa1
2     sh      0     0     1     0     21.41 0.08   sleep  16384  80105929 80100aaf 80101fe9 801012af 80106af5 8010693b 80107ca6 80107aa1
1     init    0     0     1     0     21.47 0.03   sleep  12288  80105929 801054a7 80107736 8010693b 80107ca6 80107aa1

Set priority of child: 26 to 3. Return code: 0

PID   Name   UID   GID   PPID   Prio   Elapsed CPU   State   Size   PCs
25    test    0     0     4     3     16.77 0.82   sleep  16384  80105929 80107814 8010693b 80107ca6 80107aa1
26    test    0     0     4     3     6.76  0.48   sleep  16384  80105929 80107814 8010693b 80107ca6 80107aa1
4     test    0     0     1     0     25.69 1.24   sleep  16384  80105929 80107814 8010693b 80107ca6 80107aa1
2     sh      0     0     1     0     27.27 0.08   sleep  16384  80105929 80100aaf 80101fe9 801012af 80106af5 8010693b 80107ca6 80107aa1
1     init    0     0     1     0     27.33 0.03   sleep  12288  80105929 801054a7 80107736 8010693b 80107ca6 80107aa1

Created sleeping child: 27

PID   Name   UID   GID   PPID   Prio   Elapsed CPU   State   Size   PCs
25    test    0     0     4     3     21.39 1.19   run    16384  80105929 80107814 8010693b 80107ca6 80107aa1
26    test    0     0     4     3     11.38 0.88   sleep  16384  80105929 80107814 8010693b 80107ca6 80107aa1
27    test    0     0     4     0     1.37  0.04   sleep  16384  80105929 80107814 8010693b 80107ca6 80107aa1
4     test    0     0     1     0     30.31 1.47   sleep  16384  80105929 80107814 8010693b 80107ca6 80107aa1
2     sh      0     0     1     0     31.89 0.08   sleep  16384  80105929 80100aaf 80101fe9 801012af 80106af5 8010693b 80107ca6 80107aa1
1     init    0     0     1     0     31.95 0.03   sleep  12288  80105929 801054a7 80107736 8010693b 80107ca6 80107aa1

Set priority of child: 27 to 3. Return code: 0

PID   Name   UID   GID   PPID   Prio   Elapsed CPU   State   Size   PCs
25    test    0     0     4     3     25.84 1.48   sleep  16384  80105929 80107814 8010693b 80107ca6 80107aa1
26    test    0     0     4     3     15.83 1.11   sleep  16384  80105929 80107814 8010693b 80107ca6 80107aa1
27    test    0     0     4     3     5.82  0.27   sleep  16384  80105929 80107814 8010693b 80107ca6 80107aa1
4     test    0     0     1     0     34.76 1.73   sleep  16384  80105929 80107814 8010693b 80107ca6 80107aa1

```

Figure 10: Sleeping (non-runnable) priorityset()

In figure 8 we see that for each child created, after the parent sets its priority that the child is placed at the end of the MAX queue *and* that the return code is 0 - indicating success. Thus our expectations for this stage of the test are met.

In figure 9, we see that all the children are in fact killed - reflected by ctrl-p which was demonstrated as correct in Test 1. We also see that the child makes it to the SLEEPING state with priority 0 before `setpriority()` is called and that after `setpriority()` is called it has the correct MAX priority and it is still in sleeping. The return code is 0 in all cases. Thus, our expectations for this stage of the test are met.

Because our expectations were met at each stage of the test, this test **PASSES**

## Test 4 - Priority Promotion (Requirements 3)

Here we will test that the priority promotion component of the MLFQ works as expected. To do this we will set `DEAULT_BUDGET` to 100 seconds and `PROMOTE_AT_TICKS` to 5 seconds then simply run `prioritypromotion()` (test.c lines 276-304) through executing `test`.

- The parent will create 10 spinning children that will remain on queue 0.
- The function will then create 10 spinning children and place them at queue MAX, or 3 by calling `setpriority()` - demonstrated in Test 3. After being notified that the low priority children are all created we will press ctrl-r approximately every 5 seconds and expect to see all processes at queue 0, remain at queue 0 but have their budget reset to 10000. Because we likely cannot press ctrl-r before a process runs again after having its budget reset, we expect each budget to remain between 9000-10000 even though each process has run for more than one second. We also expect the processes at queue MAX to increase one priority queue after each promotion so we will see them on queue 3, 2, 1, then 0. Additionally we expect to see that the exact order of the queue remains - this will be our best demonstration that one queue is appended to the end of the next highest queue. Their budgets wont change since they do not run; this is in accordance with the MLFQ algorithm outlined in the

requirements section and as demonstrated by Test 2. When the lower priority children are promoted to priority queue 0, we expect they will be appended to the end. However, the short-coming of this test is that we will not be able to perfectly identify this moment in time. Thus, we expect that if the originally lower priority processes are not at the end, that their budget will not be 10000 since they must have run. Due to the round robin scheduling demonstrated in test 2, the general ordering should remain.

```

Ready List Processes:
0: (9,9927) -> (11,9930) -> (10,9928) -> (12,9932) -> (13,9933) -> (14,9935) -> (8,9927) -> (6,9927)
1:
2:
3: (15,10000) -> (16,10000) -> (17,10000) -> (18,10000) -> (19,10000) -> (20,10000) -> (21,10000) -> (22,10000) -> (23,10000) -> (24,10000)

Ready List Processes:
0: (13,9974) -> (14,9974) -> (8,9974) -> (6,9974) -> (7,9974) -> (5,9974) -> (9,9974) -> (11,9973)
1:
2: (15,10000) -> (16,10000) -> (17,10000) -> (18,10000) -> (19,10000) -> (20,10000) -> (21,10000) -> (22,10000) -> (23,10000) -> (24,10000)
3:

Ready List Processes:
0: (8,9926) -> (6,9926) -> (7,9926) -> (5,9926) -> (9,9926) -> (11,9925) -> (10,9925) -> (12,9925)
1:
2: (15,10000) -> (16,10000) -> (17,10000) -> (18,10000) -> (19,10000) -> (20,10000) -> (21,10000) -> (22,10000) -> (23,10000) -> (24,10000)
3:

Ready List Processes:
0: (5,9986) -> (9,9986) -> (11,9986) -> (10,9985) -> (12,9985) -> (14,9985) -> (13,9985) -> (8,9985)
1: (15,10000) -> (16,10000) -> (17,10000) -> (18,10000) -> (19,10000) -> (20,10000) -> (21,10000) -> (22,10000) -> (23,10000) -> (24,10000)
2:
3:

Ready List Processes:
0: (7,9917) -> (5,9917) -> (9,9917) -> (11,9917) -> (12,9916) -> (10,9915) -> (14,9916) -> (13,9916)
1: (15,10000) -> (16,10000) -> (17,10000) -> (18,10000) -> (19,10000) -> (20,10000) -> (21,10000) -> (22,10000) -> (23,10000) -> (24,10000)
2:
3:

Ready List Processes:
0: (10,9995) -> (19,9995) -> (20,9995) -> (21,9995) -> (22,9995) -> (23,9995) -> (24,9995) -> (11,9994) -> (12,9994) -> (10,9994) -> (14,9994) -> (13,9994) -> (8,9994) -> (6,9994) -> (7,9994) -> (5,9994) -> (9,9994) -> (15,9994)
1:
2:
3:

```

Figure 11: Priority Promotion

In figure 11 we see that we were not precise in our pressing of `ctrl-r` every 5 seconds, so there are sometimes two displays between promotions. However we do see that the budget of all processes on queue 0 are always between 9000 and 10000, and that the budgets on the lower priority queue do not change. Demonstrating the budget reset of the priority boost, between `ctrl-r` print 3 and 4 we see process 8 on queue 0 has an increased budget! Thus this expectation of stage 2 of the test is met. We also see that the lower priority queue moves from 3 to 2 to 1 to 0 with the exact ordering retained, as expected and that once it reaches queue 0 its general ordering remains. Although these processes made it to the head of the queue, their budget is less than 10000 and thus have since run, meaning we could not see them at the end of queue 0 immediately following the priority reset. The limitations of this have been acknowledged.

Since our expectations for each stage of this test were met, this test **PASSES**.

## Test 5 - Invalid `setpriority()` calls (Requirements 7, 8)

This test will demonstrate that `setpriority()` calls with invalid arguments will return an error code, and not update the priority of the PID given. To do this we will add `invalid_priorityset()` to the user program `test` (`test.c` lines 256-274). When will then:

- Begin by executing `test &` and pressing `ctrl-p`, while it initially sleeps, to see all active process information.
- When `test` wakes after 2 seconds, `invalid_priorityset()` will then call `priorityset(PID, PRIORITY)` with PIDs -1, 1000, and 9999 and a priority of MAX. Since we know by virtue of the code, that PIDs are assigned in increasing order and that only `sh` and `initproc` are running at boot, all of these PIDs are invalid. This is confirmed by the `ctrl-p` output. We also know that MAX is a valid priority so that the failure of this call is strictly due to the PID being invalid. Thus when the process prints the values attempted to be assigned and the return code, we expect a return code of -1.
- Next, the `test` will attempt to assign invalid priorities for the process `init` which has a PID of 1, confirmed by `ctrl-p`, printing its attempts and the return codes. The invalid priorities given will be



MAX+1 and MAX+2. After which, ctrl-p will be pressed. We expect that `test` will have printed -1 for all return codes, and that the priority for init will remain unchanged.

```
cpu1: starting
cpu0: starting
sb: size 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ test &
$
PID      Name   UID    GID    PPID    Prio    Elapsed CPU    State  Size    PCs
4        test    0       0       1       0       0.52    0.01    sleep  16384   80105929 80107814 8010693b 80107ca6 80107aa1
2        sh      0       0       1       0       3.39    0.04    sleep  16384   80105929 80100aa7 80101fe9 801012af 80106af5 8010693b 80107ca6 80107aa1
1        init    0       0       1       0       3.47    0.03    sleep  12288   80105929 801054a7 80107736 8010693b 80107ca6 80107aa1

$ Priorityset() for PID -1 and priority MAX returned -1
$ Priorityset() for PID 1000 and priority MAX returned -1
$ Priorityset() for PID 9999 and priority MAX returned -1
$ Priorityset() for PID 1 and priority MAX+1 (4) returned -1
$ Priorityset() for PID 1 and priority MAX+2 (5) returned -1
zombie!

$
PID      Name   UID    GID    PPID    Prio    Elapsed CPU    State  Size    PCs
2        sh      0       0       1       0       6.44    0.07    sleep  16384   80105929 80100aa7 80101fe9 801012af 80106af5 8010693b 80107ca6 80107aa1
1        init    0       0       1       0       6.52    0.03    sleep  12288   80105929 801054a7 80107736 8010693b 80107ca6 80107aa1

$ Ready List Processes:
0:
1:
2:
3:
=
```

Figure 12: Invalid `setpriority()` calls

As expected, all calls to `setpriority()` returned -1 indicating failure. PIDs -1, 1000, and 9999 did not exist on ctrl-p. Init indeed had a PID of 1 but its priority did not change. Note the ready list output indicates MAX was 3 and the calls to change the priority of init were 4 and 5 respectively.

Since all our expectations for this test were met, this test **PASSES**.

## Test 6 - MLFQ with generic MAX (Requirement 9)

This test will demonstrate that the new MLFQ will operate as expected with a generic MAX value. To do this we will change MAX to 1, 3, and 7 with `DEFAULT_BUDGET` set to 100 seconds and `TICKS_TO_PROMOTE` set to 5 seconds- recompiling each time.

We will then invoke `test &` to regain shell control, where `test` will execute the function `prioritypromotion()` described in Test 4. Since `prioritypromotion()` relied on MAX to set the lower priority values, our expectations for the process promotion with a MAX of 1, 3, and 7 will be the same as in Test 4 (please reference Test 4 for outline of expectations). An additional expectation will be that the MAX queue is numbered corresponding to what MAX is defined as in the compilation (i.e. 1, 3, and 7 respectively).

```
init: starting sh
$ Ready List Processes:
0:
1:

$ test &
$ parent done creating creating prio 0 children

$ parent done creating low prio creating children
Ready List Processes:
0: (13,9960) -> (14,9963) -> (15,9965) -> (6,9943) -> (7,9945) -> (8,9947) -> (9,9958) -> (10,9952)
1: (16,10000) -> (17,10000) -> (18,10000) -> (19,10000) -> (20,10000) -> (22,10000) -> (23,10000) -> (24,10000) -> (25,10000) -> (26,10000)

Ready List Processes:
0: (19,9954) -> (20,9954) -> (22,9954) -> (23,9954) -> (24,9954) -> (25,9954) -> (26,9954) -> (10,9953) -> (11,9953) -> (12,9953) -> (13,9953) -> (14,9953) -> (15,9953) -> (6,9953) -> (7,9953) -> (8,9953) -> (9,9953) -> (16,9953)
1:

$ QEMU 2.5.0 monitor - type 'help' for more information
```

Figure 13: MAX of 1

```

init: starting sh
$
$ Ready List Processes:
0:
1:
2:
3:

$ test &
$ parent done creating creating prio 0 children
parent done creating low prio creating children
Ready List Processes:
0: (7,9925) -> (8,9927) -> (9,9930) -> (10,9932) -> (11,9934) -> (12,9936) -> (13,9939) -> (14,9942)
1:
2:
3: (17,10000) -> (18,10000) -> (19,10000) -> (20,10000) -> (21,10000) -> (22,10000) -> (23,10000) -> (24,10000) -> (25,10000) -> (26,10000)

$ Ready List Processes:
0: (12,9927) -> (13,9927) -> (14,9927) -> (15,9927) -> (16,9927) -> (17,9927) -> (18,9927) -> (19,9927)
1:
2: (17,10000) -> (18,10000) -> (19,10000) -> (20,10000) -> (21,10000) -> (22,10000) -> (23,10000) -> (24,10000) -> (25,10000) -> (26,10000)
3:

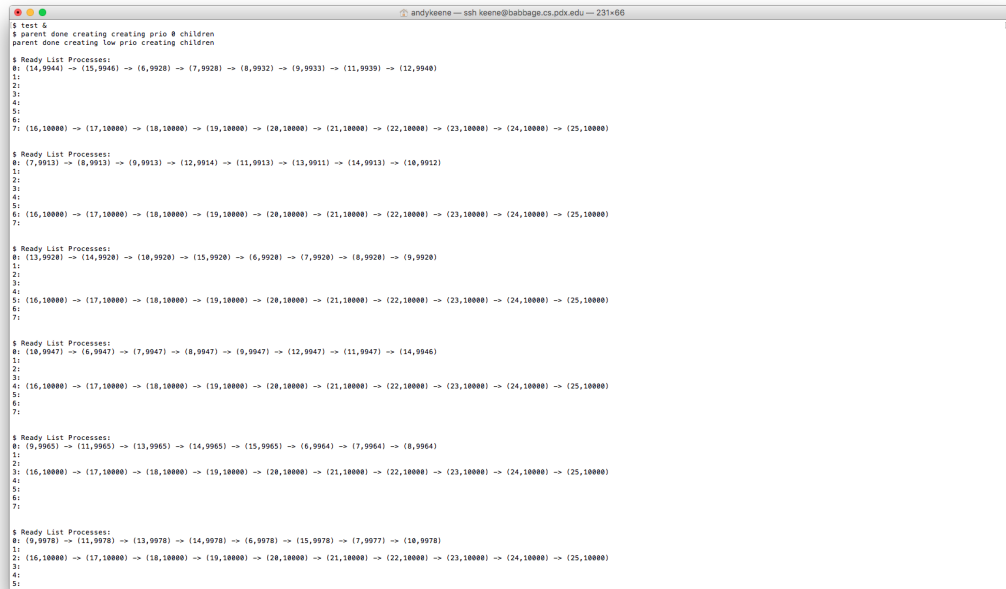
$ Ready List Processes:
0: (12,9940) -> (13,9940) -> (14,9940) -> (15,9940) -> (16,9940) -> (17,9940) -> (18,9940) -> (19,9939)
1: (17,10000) -> (18,10000) -> (19,10000) -> (20,10000) -> (21,10000) -> (22,10000) -> (23,10000) -> (24,10000) -> (25,10000) -> (26,10000)
2:
3:

$ Ready List Processes:
0: (20,9970) -> (21,9970) -> (22,9970) -> (23,9970) -> (24,9970) -> (25,9970) -> (26,9970) -> (16,9969) -> (7,9969) -> (8,9969) -> (9,9969) -> (11,9969) -> (10,9969) -> (12,9969) -> (13,9969) -> (14,9969) -> (15,9969) -> (17,9969)
1:
2:
3:

$ halt

```

Figure 14: MAX of 3



```

$ test &
$ parent done creating creating prio 0 children
parent done creating low prio creating children
$ Ready List Processes:
0: (14,9944) -> (15,9946) -> (6,9928) -> (7,9928) -> (8,9932) -> (9,9933) -> (11,9939) -> (12,9940)
1:
2:
3:
4:
5:
6:
7: (16,10000) -> (17,10000) -> (18,10000) -> (19,10000) -> (20,10000) -> (21,10000) -> (22,10000) -> (23,10000) -> (24,10000) -> (25,10000)

$ Ready List Processes:
0: (7,9913) -> (8,9913) -> (9,9913) -> (12,9914) -> (11,9913) -> (13,9911) -> (14,9913) -> (10,9912)
1:
2:
3:
4:
5:
6: (16,10000) -> (17,10000) -> (18,10000) -> (19,10000) -> (20,10000) -> (21,10000) -> (22,10000) -> (23,10000) -> (24,10000) -> (25,10000)
7:

$ Ready List Processes:
0: (13,9928) -> (14,9928) -> (10,9928) -> (15,9928) -> (6,9928) -> (7,9928) -> (8,9928) -> (9,9928)
1:
2:
3:
4:
5: (16,10000) -> (17,10000) -> (18,10000) -> (19,10000) -> (20,10000) -> (21,10000) -> (22,10000) -> (23,10000) -> (24,10000) -> (25,10000)
6:
7:

$ Ready List Processes:
0: (10,9947) -> (6,9947) -> (7,9947) -> (8,9947) -> (9,9947) -> (12,9947) -> (11,9947) -> (14,9946)
1:
2:
3:
4: (16,10000) -> (17,10000) -> (18,10000) -> (19,10000) -> (20,10000) -> (21,10000) -> (22,10000) -> (23,10000) -> (24,10000) -> (25,10000)
5:
6:
7:

$ Ready List Processes:
0: (8,9963) -> (11,9963) -> (13,9963) -> (14,9963) -> (15,9963) -> (6,9964) -> (7,9964) -> (8,9964)
1:
2:
3: (16,10000) -> (17,10000) -> (18,10000) -> (19,10000) -> (20,10000) -> (21,10000) -> (22,10000) -> (23,10000) -> (24,10000) -> (25,10000)
4:
5:
6:
7:

$ Ready List Processes:
0: (9,9978) -> (11,9978) -> (12,9978) -> (14,9978) -> (6,9978) -> (15,9978) -> (7,9977) -> (10,9978)
1:
2: (16,10000) -> (17,10000) -> (18,10000) -> (19,10000) -> (20,10000) -> (21,10000) -> (22,10000) -> (23,10000) -> (24,10000) -> (25,10000)
3:
4:
5:

```

Figure 15: MAX of 7, part 1

```

$ Ready List Processes:
0: (8,9978) -> (13,9978) -> (14,9978) -> (6,9978) -> (15,9978) -> (7,9977) -> (10,9978)
1:
2: (16,10000) -> (17,10000) -> (18,10000) -> (19,10000) -> (20,10000) -> (21,10000) -> (22,10000) -> (23,10000) -> (24,10000) -> (25,10000)
3:
4:
5:
6:
7:

$ Ready List Processes:
0: (12,9903) -> (8,9902) -> (6,9903) -> (9,9902) -> (11,9902) -> (13,9902) -> (14,9902) -> (15,9903)
1:
2: (16,10000) -> (17,10000) -> (18,10000) -> (19,10000) -> (20,10000) -> (21,10000) -> (22,10000) -> (23,10000) -> (24,10000) -> (25,10000)
3:
4:
5:
6:
7:

Ready List Processes:
0: (7,9977) -> (12,9977) -> (8,9977) -> (6,9977) -> (9,9977) -> (11,9976) -> (13,9976) -> (14,9976)
1: (16,10000) -> (17,10000) -> (18,10000) -> (19,10000) -> (20,10000) -> (21,10000) -> (22,10000) -> (23,10000) -> (24,10000) -> (25,10000)
2:
3:
4:
5:
6:
7:

$ Ready List Processes:
0: (29,9999) -> (8,9990) -> (9,9990) -> (11,9990) -> (13,9990) -> (14,9990) -> (15,9990) -> (18,9990) -> (12,9990) -> (7,9990) -> (6,9990) -> (16,9990) -> (17,9990) -> (18,9990) -> (19,9990) -> (20,9990) -> (21,9990) -> (22,9990)
1:
2:
3:
4:
5:
6:
7:

$ QEMU 2.5.0 monitor - type 'help' for more information

```

Figure 16: Max of 7, part 2

In figures 13-15 demonstrating priority promotion for MAXes of 1, 3, and 7 we see that each `ctrl-r` display at approximately 5 second intervals shows: a budget reset evident on the top-most priority queue; a promotion for the lower priority processes from queue MAX, MAX-1,..., 0; and that the lower priority processes are appended to queue 0 evident by the retention of general ordering with a non-full budget (acknowledged limitation of Test 4). Thus all expectations for priority promotion with MAX set to 1, 3, and 7, are met. Because all expectations for this test were met, this test **PASSES**

Further, because all tests passed we can conclude that the corresponding requirements 1-12 have been fulfilled.