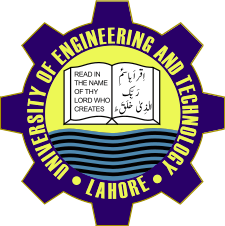
Operating Systems EE431L

Lab 4 - Report



*Implementing Lottery & MLFQ Scheduling in xv6*

*Submitted by:*

2016-EE-189 Muhammad Kamil

*Submitted to:*

Mr. M. Usama Zubair

Section I

*Changes made to different files in xv6*

**proc.h**

Added the following variables to ‘proc’ struct:

struct **proc** {

**. . . . . . .**

  int ticket;*// number of tickets assigned to this proc*

  int hticks;*// no. of ticks consumed by this proc in high-priority queue*

  int lticks;*// no. of ticks consumed by this proc in low-priority queue*

  int queue;*// identifies proc priority [0 => Low, 1 => High]*

  int inuse;*// tells whether this process table entry is in use*

};

Added the following definitions in this header file:

#define **BOOST\_PERIOD\_TICKS**  20 *// no. of ticks between bulk priority-boosts*

#define **\_\_MLFQ\_BOOST\_\_** *// to enable MLFQ PERIODIC BOOST*

#define **PR\_HIGH** 1 *// high-priority queue/proc identifier*

#define **PR\_LOW**  0 *// low-priority queue/proc identifier*

**trap.c**

Initialized the following variables in this file:

*// mlfq boost params (originally in defs.h)*

**uint** ticks\_at\_last\_boost = 0;

**uint** mlfq\_boost\_pending = 0;

Updated the trap handler function (timer-interrupt) to set the ‘*mlfq\_boost\_pending’* variable:

  case **T\_IRQ0** + **IRQ\_TIMER**:

**. . . . . . .**

      if (ticks - ticks\_at\_last\_boost >= **BOOST\_PERIOD\_TICKS**)

      {

        ticks\_at\_last\_boost = ticks;

        mlfq\_boost\_pending = 1;

      }

**. . . . . . .**

    break;

**System Calls**

Added the “*int* ***setticket****(int num)*” system call to xv6. This function assigns ticket to the calling process, equal to the number passed as its argument. It returns 0 upon successful assignment of ticket.

int **setticket**(int num)

{

**argint**(0, &num);

  if (num < 0) return -1;

  proc->ticket = num;

  return 0;

}

Added the “*int* ***getpinfo****(struct pstat\* stat)*” system call to xv6. This function takes a (pointer to) ‘pstat’ struct from the user, passes it to the kernel space, and fills in useful information into the structure’s arrays, such as all process’ pid, inuse, lticks, and hticks, using the ‘ptable’ array of procs in proc.c.

int **getpinfo**(struct **pstat**\* stat)

{

**argptr**(0, (void\*)&stat, sizeof(struct **pstat**));

  struct **proc** \*p1 = ptable.proc;

  for(struct **proc** \*p2 = ptable.proc; p2 < &ptable.proc[**NPROC**]; p2++)

  {

*// set inuse=0 if process entry (in ptable) not in use*

    if (p2->pid==0) p2->inuse = 0;

  }

*// fill pstat vars for each process in ptable*

  for(int i=0; i<**NPROC** && p1 < &ptable.proc[**NPROC**]; i++, p1++)

  {

    stat->pid[i] = p1->pid;

    stat->inuse[i] = p1->inuse;

    stat->lticks[i] = p1->lticks;

    stat->hticks[i] = p1->hticks;

  }

  return 26;

}

**proc.c**

Initialized ‘proc’ struct custom-variables in **allocproc()** function:

*// init custom vars*

  p->ticket = 1;

  p->queue = 1;

  p->hticks = 0;

  p->lticks = 0;

  p->inuse = 1;

**proc.c** *(cont.)*

To covert xv6’s RR-scheduling to Lottery + MLFQ scheduling, the **scheduler()** function in proc.c was modified, along with addition of separate functions – e.g. to generate random numbers, and check-for and boost priorities (based on notification variable from trap.c).

void **scheduler**(void)

{

*// 1 => HIGH PRIORITY QUEUE // PR\_HIGH*

*// 0 => LOW PRIORITY QUEUE // PR\_LOW*

*// 'ticks' declared in defs.c gives the current total ticks (not used here)*

*// 'ticks\_at\_last\_boost' declared in defs.c gives total ticks at last boost*

*// 'mlfq\_boost\_pending' tells scheduler when to boost all process' priority*

  struct **proc** \*p;

  for(;;){

*// Enable interrupts on this processor.*

**sti**();

*// Loop over process table looking for process to run.*

**acquire**(&ptable.lock);

*// calculate ticket sum and get current priority queue*

*// starting at higher priority - for a queue to be chosen, its ticket sum*

*// must be non-zero i.e. it has runnable process(es)*

*// if not, switch to lower priority queue*

    int ticket\_sum = 0, curr\_priority = **PR\_HIGH**;

    for (; curr\_priority >= **PR\_LOW**; curr\_priority--)

    {

*// start with sum=0 for this queue/priority*

ticket\_sum = 0;

*// accumulate ticket sum for this priority level (queue),*

*// if that process is runnable*

      for(p = ptable.proc; p < &ptable.proc[**NPROC**]; p++)

      {

        if (p->queue == curr\_priority && p->state == RUNNABLE)

        {

          ticket\_sum += p->ticket;

        }

      }

*// if there are runnable processes in this queue, ticket sum must be*

*// non-zero, hence break*

      if (ticket\_sum != 0) break;

*// if no runnable processes on this queue, check lower priority one*

    }

*// generate random number within ticket sum*

    int ran\_num = **random\_at\_most**(ticket\_sum);

*// for each process in selected priority, accumulate tickets uptill it*

*// (considering only runnable processes)*

*// schedule it if its ticket is greater than sum so far*

    int curr\_ticket\_sum = 0;

    for(p = ptable.proc; p < &ptable.proc[**NPROC**]; p++)

    {

*// skip process if not runnable or not in current priority queue*

      if(p->state != RUNNABLE || p->queue != curr\_priority) continue;

*// accumulate ticket sum so far*

      curr\_ticket\_sum += p->ticket;

*// if sum so far is less than generated random val, move to next process*

      if (curr\_ticket\_sum < ran\_num) continue;

*// else, this is the winning process, schedule it, update ticks*

      if (bDebug) **cprintf**("<< [PR: %d] [SM: %d] [RND: %d] [W-TKT: %d] \

[W-PID: %d] >>\n", curr\_priority, ticket\_sum,

ran\_num, p->ticket, p->pid);

*// Switch to chosen process. It is the process's job to release*

*// ptable.lock and then reacquire it before jumping back to us.*

      proc = p;

**switchuvm**(p);

      p->state = RUNNING;

**swtch**(&cpu->scheduler, proc->context);

**switchkvm**();

*// if it is a high-priority process, update process high-ticks,*

*// and reduce process priority*

      if (p->queue == **PR\_HIGH**)

      {

        p->hticks++;

        p->queue = **PR\_LOW**;

      }

*// else if it is a low-priority process, update process low-ticks,*

*// and schedule it for a second time-slice (if runnable)*

      else if (p->queue == **PR\_LOW**)

      {

        p->lticks++;

*// before rescheduling, check if boost is pending*

        #ifdef **\_\_MLFQ\_BOOST\_\_**

        if (**check\_do\_boost**()) { proc = 0; break; };

        #endif

        if (p->state == RUNNABLE)

        {

**switchuvm**(p);

          p->state = RUNNING;

**swtch**(&cpu->scheduler, proc->context);

**switchkvm**();

          p->lticks++;

        }

      }

*// Process is done running for now.*

*// It should have changed its p->state before coming back.*

      proc = 0;

*// priority boost*

*// (only done if defined, and some process was just scheduled)*

      #ifdef **\_\_MLFQ\_BOOST\_\_**

**check\_do\_boost**();

      #endif

      break;

    }

**release**(&ptable.lock);

  }

}

*// checks if the ‘mlfq\_boost\_pending’ variable (updated in trap handler) has*

*// been set, and if so, boosts all processes to the high-priority queue*

int **check\_do\_boost**()

{

  if (mlfq\_boost\_pending)

  {

    mlfq\_boost\_pending = 0;

    for(struct **proc** \*p = ptable.proc; p < &ptable.proc[**NPROC**]; p++)

    {

      p->queue = **PR\_HIGH**;

    }

**cprintf**("[TICKS: %d]\n", ticks);

    return 1;

  }

  return 0;

}

**Note on modified scheduler**

We have two main queues, one of high-priority, or priority 1, and one of low-priority, or priority 0. In the higher-priority queue, a process runs for one time-slice, and in lower-priority queue, it runs for two time-slices.

Initially, all processes have a priority of 1 (high), and after getting scheduled once, their priority is reduced to 0 (low). Then after a specific number of ticks (period of MLFQ), all processes are pushed to highest priority.

As for the lottery scheduling part, before scheduling any process, we first check their ticket (plus sum of previous tickets) against a random-number generated from a max-value equal to the sum of tickets of all runnable processes in currently-selected queue. This procedure is done first for the higher-priority queue, and if no process exists on this queue, then we choose a winning process from the lower-priority queue.

Section II

*Scheduling Tests and Graphs*

A test file (program) was added to xv6 (for use in user-mode) that spawns several child processes; the child processes then set for themselves different ticket values using the newly-added **setticket()** system call, and then each proceeds to run a long for-loop.

The parent process stores each of its child’s **pid**s in an array, and sleeps for some amount of time to let the children run, after which it uses the added **getpinfo()** system call to fill a ‘pstat’ struct with all the xv6’s processes’ info – it stores the following data about all 64 entries of the process-table:

struct **pstat** {

    int inuse[**NPROC**];*// whether this slot of the process table is in use (1 or 0)*

    int pid[**NPROC**]; *// the PID of each process*

    int hticks[**NPROC**]; *// the number of ticks each process has accumulated at HIGH priority*

    int lticks[**NPROC**]; *// the number of ticks each process has accumulated at LOW priority*

};

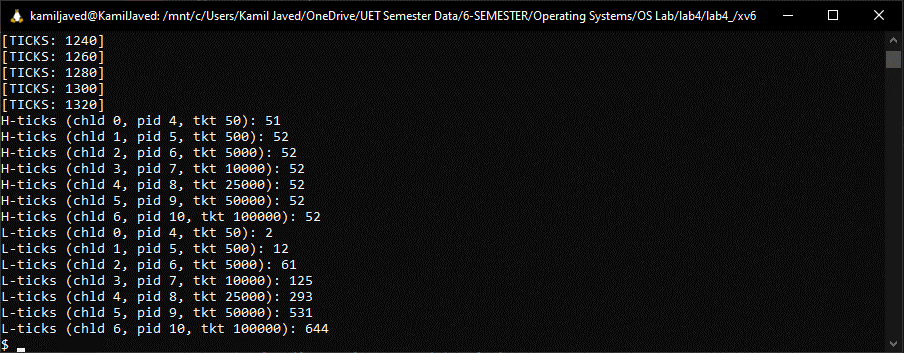
After collecting this data, the parent process kills off all its child processes, and prints the number of High and Low ticks consumed by each of the different-ticket-value children over their lifetime.

For the current test, 7 child processes were created, with the following ticket values:

#define **szArrTicks** 7

int arrTicks[**szArrTicks**] = {50, 500, 5000, 10000, 25000, 50000, 100000};

Running the test.c program in xv6 gave the following output:



In the console, ticks (“[TCIKS: ####]”) are printed whenever MLFQ priority boost (to HIGH or queue-1) is done, having period for which has been set to 20 time-slices (or ticks) – defined by BOOST\_PERIOD\_TICKS in proc.h.

**Graphs of Tickets vs Number of Ticks of Process**

For the 7 child processes (child 0 to 6), the ticks spent in high and low priority queues are shown in the console screenshot on the previous page, and is also shown below in tabular form:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Process No. | PID | Process Tickets | High Ticks | Low Ticks | Total Ticks |
| 0 | 4 | 50 | 51 | 2 | 53 |
| 1 | 5 | 500 | 52 | 12 | 64 |
| 2 | 6 | 5000 | 52 | 61 | 113 |
| 3 | 7 | 10000 | 52 | 125 | 177 |
| 4 | 8 | 25000 | 52 | 293 | 345 |
| 5 | 9 | 50000 | 52 | 531 | 583 |
| 6 | 10 | 100000 | 52 | 644 | 696 |

The bar-chart graphs for Process Tickets vs High-Ticks, Low-Ticks, and Total-Ticks are as follows:

**Ticks** for each Child Process in **High-Priority (1) Queue**

**Ticks** for each Child Process in **Low-Priority (0) Queue**

**Total CPU Ticks** for each Child Process

**Note on the Graph Trends**

The 7 processes run only for a defined period of time, before being killed by the parent process, so they all have roughly the same lifetime. Their number of ticks, during that lifetime, will vary based on the tickets assigned to them by the user.

Initially, when all processes are in the high-priority queue, each process will run at least once and then move on to the low-priority queue. This is because the process with the next highest ticket is still in the high-priority queue (which is checked first), and the previous highest ticket process was moved to the lower-priority queue after being de-scheduled. So, all processes get a chance to run on the high-priority queue, and they run (roughly) in order of decreasing ticket value.

According to the High-Ticks graph, all processes spend almost an equal amount of (overall) time in the high-priority queue, based on the fact that they are all boosted to the high-priority queue after a fixed period – after which they each get a tick one-by-one in the high-priority queue (the probability of the scheduling order being dependent on process’ ticket value, and somewhat on their index in the process-table).

Then, according to the Low-Ticks graph, in the low-priority queue, the process with the highest ticket value gets a greater number of Low-Ticks. This is because, once all processes reach the low-priority queue, they remain their until the next boost, and hence in the meantime, the likelihood of them being scheduled is determined mainly by their ticket value (and is also be affected by their index in the process-table).

The overall trend is visible in the above graph (Process Tickets vs Total Ticks), where, as the number of tickets assigned to a process increases, it is chosen as the winning process more often, and proceeds towards completion more rapidly. Hence, generally, the total number of ticks given to a process, within a specified timeframe, is directly proportional to the Number of Tickets that it’s been assigned.

These results indicate that the implemented combination of Multilevel Feedback Queue (of 2 priorities) and Lottery-Ticket Scheduling works as intended.