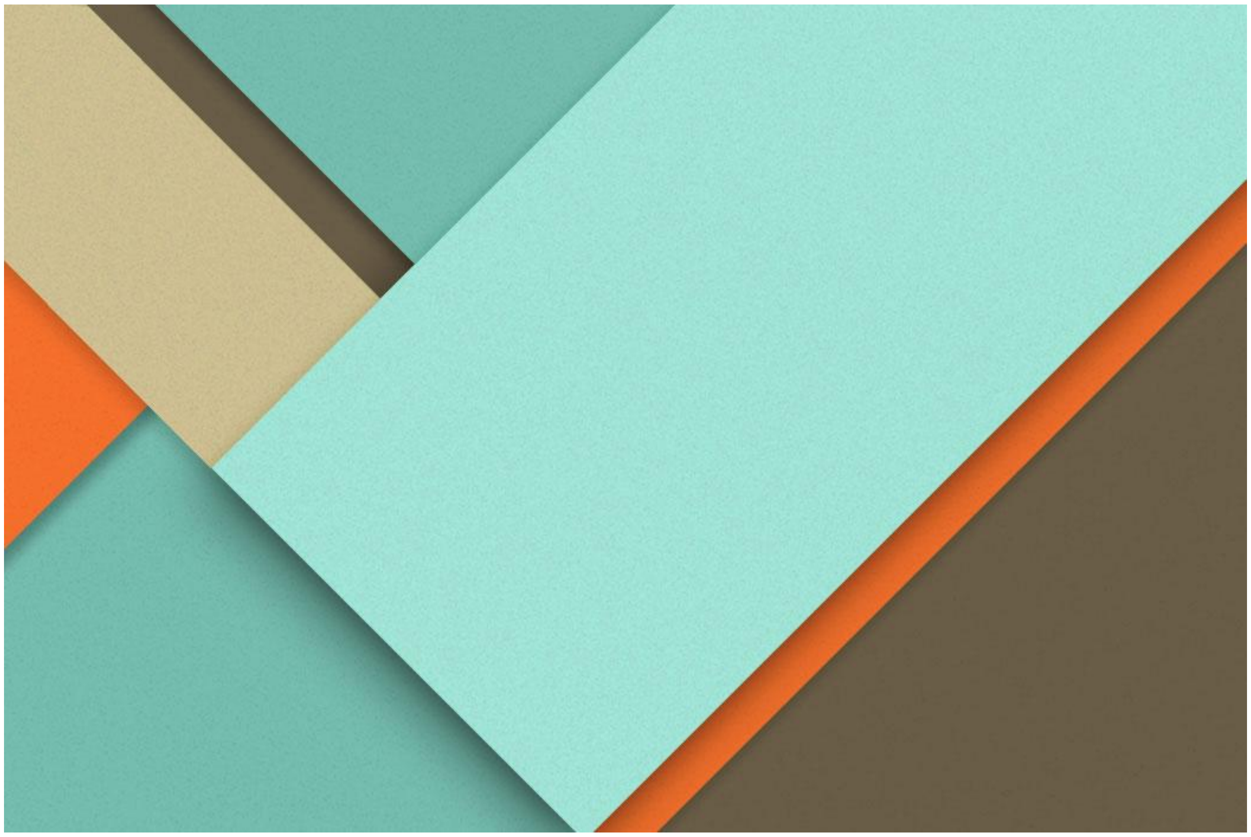

# Operating Systems and Networks - Enhancing xv6-riscv

—

Sasanka GRS - 2019112017

Viswanadh KS - 2019112011

## Run the operating system

Run the command -

make clean && make qemu SCHEDULER=<scheduler>

<scheduler> = RoundRobin

(For Round Robin scheduling)

<scheduler> = FCFS (For First-come first-serve scheduling)

<scheduler> = LOT (for Lottery Based)

<scheduler> = PBS (For Priority Based Scheduling)

<scheduler> = MLFQ (For Multi-level Feedback Queue Scheduling)

## Files Modified/Added

- defs.h
- Makefile
- proc.c
- proc.h
- setpriority.c
- syscall.c
- syscall.h
- sysproc.c
- trap.c
- user.h
- usys.S
- kalloc.c
- vm.c
- alarmtest.c
- schedulertest.c
- strace.c
- time.c
- usys.pl

## Explanation of Scheduling Algorithms:

### Round Robin (Default):

Iterate through all the processes and find the process that is RUNNABLE. If found, allocate CPU to it. (File: proc.c)

```
#ifdef RoundRobin
 for (;;)
 {
   // Avoid deadlock by ensuring that devices can interrupt.
   intr_on();
   for (p = proc; p < &proc[NPROC]; p++)
   {
     acquire(&p->lock);
     if (p->state == RUNNABLE)
     {
       // Switch to chosen process.  It is the process's job
       // to release its lock and then reacquire it
       // before jumping back to us.
       p->state = RUNNING;
       c->proc = p;
       swtch(&c->context, &p->context);

       // Process is done running for now.
       // It should have changed its p->state before coming back.
       c->proc = 0;
     }
     release(&p->lock);
   }
 }
#endif
```

## FCFS Scheduler:

Iterate through all the processes to find the one with minimum start time. Then,

allocate the CPU to it. (File: proc.c)

```
#ifdef FCFS
 // --------------------------------------------- FCFS
----------------------------------------------
 for (;;)
 {
   // Avoid deadlock by ensuring that devices can interrupt.
   intr_on();

   struct proc *min;
   min = 0;

   for (p = proc; p < &proc[NPROC]; p++)
   {
     acquire(&p->lock);
     if (p->state == RUNNABLE)
     {
       // Switch to chosen process.  It is the process's job
       // to release its lock and then reacquire it
       // before jumping back to us.
```

```
        if (min && p->start < min->start)
        {
          release(&min->lock);
          min = p;
          continue;
        }
        if (!min)
        {
          min = p;
          continue;
        }
      }
      release(&p->lock);
    }

    if (min)
    {
      min->state = RUNNING;
      c->proc = min;
      swtch(&c->context, &min->context);
      c->proc = 0;
      release(&min->lock);
    }
  }
#endif
```

## Lottery Based Scheduling:

- Declare tickets for each process in proc structure. (File: proc.h)

```
int tickets;                  // Tickets
------------------------------------------------
```

- Initialise the tickets to 1 in allocproc (File: proc.c)

```
p->tickets = 1;
```

- Add a system call sys_settickets. (File: syscall.c)

```
extern uint64 sys_sigreturn(void);


[SYS_settickets] sys_settickets,
```

- Define the system call sys_settickets. (File: sysproc.c)

```
uint64
sys_settickets(void)
{
 int n;
 argint(0, &n);
 struct proc *p = myproc();
 p->tickets = n;
 return n;
}
```

- Make a function to get a random number less than a maximum value

```
long random_at_most(long max) {
 unsigned long
   // max <= RAND_MAX < ULONG_MAX, so this is okay.
   num_bins = (unsigned long) max + 1,
   num_rand = (unsigned long) RAND_MAX + 1,
   bin_size = num_rand / num_bins,
   defect   = num_rand % num_bins;

 long x;

 do {

  x = genrand();

 }

 // This is carefully written not to overflow

 while (num_rand - defect <= (unsigned long)x);


 // Truncated division is intentional

 return x/bin_size;

}
```

- Iterate through all the processes. When looping through the processes keep the **counter of the total number of tickets passed**. Just when the counter becomes greater the random value we got, run the process. Put a break at the end of for loop so that we don't execute the processes following the process we just run.

```
#ifdef LOT

 // -------------------------------------------- Lottery
---------------------------------------------

 for (;;)

 {

   // Enable interrupts on this processor.

   intr_on();


   int tickets_passed = 0;

   int totalTickets = 0;


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

   {

     if (p->state != RUNNABLE)

     {
       continue;
     }
     totalTickets = totalTickets + p->tickets;
   }

   int winner = random_at_most(totalTickets);

   foundproc = 0;

   // Loop over process table looking for process to run.
   for (p = proc; p < &proc[NPROC]; p++)
   {
     acquire(&p.lock);
     if (p->state != RUNNABLE)
     {
       release(&ptable.lock);
       continue;
     }
     tickets_passed += p->tickets;
     if (tickets_passed < winner)
     {
       continue;
     }

     // Switch to chosen process.  It is the process's job
     // to release ptable.lock and then reacquire it
     // before jumping back to us.
```

```
     p->state = RUNNING;

     c->proc = p;

     swtch(&c->context, &p->context);


     // Process is done running for now.

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

     c->proc = 0;

     release(&ptable.lock);

     break;

   }

 }

}

#endif
```

## Priority Based Scheduling:

- Declare priority for each process in proc structure. (File: proc.h)

```
int sp;                   // Static priority
-----------------------------------------------
```

- Initialize the priority to 60 in allocproc (File: proc.c)

```
p->sp = 60;
```

- Make a function to update times

```
void update_time()

{

 struct proc *p;

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

 {

   acquire(&p->lock);
```

```
    if (p->state == RUNNING)
    {
      p->rtime++;
      p->totTime++;
    }

    if (p->state == SLEEPING)
      p->slep++;

    release(&p->lock);
  }
}
```

- Make the function setpriority

```
int set_priority(int priority, int pid)
{
  struct proc *p;

  for (p = proc; p < &proc[NPROC]; p++)
  {
    acquire(&p->lock);

    if (p->pid == pid)
    {
      int val = p->sp;
      p->sp = priority;
```

```
     p->rtime = 0;
     p->slep = 0;

     release(&p->lock);

     if (val > priority)
       yield();
     return val;
   }
   release(&p->lock);
 }
 return -1;
}
```

- Iterate through all the processes to check for the one with least priority (value). If found, allocate CPU to it. (File: proc.c)

```
#ifdef PBS
// --------------------------------------------- Priority Based
---------------------------------------------

for (;;)
{
 // Avoid deadlock by ensuring that devices can interrupt.
 intr_on();

 struct proc *top = 0;
 int dp = 101;
 for (p = proc; p < &proc[NPROC]; p++)
 {
```

```
acquire(&p->lock);


int nice = 5; // Niceness


if (p->rep) // Repeats at least once
{
  if (p->slep + p->rtime != 0)
    nice = (int)((p->slep / (p->slep + p->rtime)) * 10);
  else
    nice = 5;
}


int dp1 = p->sp - nice + 5; // Dynamic Priority
if (dp1 < 0)
{
  dp1 = 0;
}
if (dp1 > 100)
{
  dp1 = 100;
}


if (p->state == RUNNABLE)
{
  if (!top)
  {
    top = p;
```

```
          dp = dp1;
          continue;
        }
        if (dp > dp1)
        {
          release(&top->lock);
          top = p;
          dp = dp1;
          continue;
        }
        if ((dp == dp1 && p->rep < top->rep) || (dp == dp1 && p->rep == 
top->rep && p->start < top->start))
        {
          release(&top->lock);
          top = p;
          dp = dp1;
          continue;
        }
      }
      release(&p->lock);
   }

   if (top)
   {
      top->rep++;
      top->startTime = ticks;
      top->state = RUNNING;
      top->rtime = 0;
      top->slep = 0;

      c->proc = top;

      swtch(&c->context, &top->context);

      c->proc = 0;

      release(&top->lock);

   }

  }

#endif
```

## MLFQ Scheduling:

- Add timeInQ[5], q, cpuTime, q_in to the proc struct (File: proc.h)
    - timeInQ[i] represents the number of ticks the process ran in queue i.
    - q is the current queue in which the process is present
    - cpuTime is the number of ticks spent by the process in a CPU before it relinquishes.
    - q_in is the time (in ticks) when the process enters the queue in which it is currently present.

```
 uint cpuTime;    // Amount of time process got cpu


 int q;                  // Current queue where process is

 int q_in;               // Time at which process entered queue

 int timeInQ[5];         // Time spent in each queue
```

- Create queues. (File: proc.c)
    - qProc stores the number of processes in the queue.
    - tickMax stores the maximum number of ticks that are allowed for the queue.

```
int qProc[5] = {0};

int tickMax[5] = {1, 2, 4, 8, 16};

struct proc *queue[5][NPROC];
```

- Initialize the arrays and variables. (File: proc.c)

```
p->cpuTime = 0;

 p->q = 0;

 p->q_in = 0;

 for(int i=0; i<5; i++)

 {

   p->timeInQ[i] = 0;

 }
```

- Add the process in the queue 0 when they are created. (File: proc.c)

```
// in userinit()

#ifdef MLFQ

   p->q_in = ticks;

   p->q = 0;

   queue[0][qProc[0]] = p;

   qProc[0]++;

 #endif


// in fork()

#ifdef MLFQ

   np->q_in = ticks;

   np->q = 0;

   queue[0][qProc[0]] = np;

   qProc[0]++;

 #endif
```

- Schedule the processes (File: proc.c)
    - Check for aging. If a process has not been allocated CPU for 100 ticks since it entered the queue, promote it to the upper queue (incase the current queue is non zero)

- Next, iterate over all the queues. If the length of the queue is greater than zero, schedule the process which is at the head of the queue.
- After the process has exited or finished its time slice, add it to the queue.
- The change of queue takes place in the trap function.

```
intr_on();

 acquire(&p.lock);

 for (int i = 1; i < 5; i++)
 {
   for (int j = 0; j < qProc[i]; j++)
   {
     p = queue[i][j];
     acquire(&p.lock);
     if (ticks - p->q_in > 150)
     {
       for (int k = j; k < qProc[i] - 1; k++)
       {
         queue[i][k] = queue[i][k + 1];
       }
       qProc[i]--;
       p->q_in = ticks;
       p->q = i - 1;
       queue[i - 1][qProc[i - 1]] = p;
       qProc[i - 1]++;
     }
     release(&p.lock);
   }
 }
```

```
struct proc *q = 0;

for (int i = 0; i <= 4; i++)

{

  if (qProc[i] > 0 && queue[i][0] != 0)

  {

    q = queue[i][0];

    acquire(&q.lock);

    for (int j = 0; j < qProc[i] - 1; j++)

    {

      queue[i][j] = queue[i][j + 1];

    }

    qProc[i]--;

    release(&p.lock);

    break;

  }

}
```

- Once the process is scheduled, check if it has exhausted its time slice. If yes, demote its queue and yield the process. Else, increment the cpuTime and continue its operation. (File: trap.c)

```
#ifdef MLFQ

 if ((which_dev == 2 && myproc() != 0 && myproc()->state == RUNNING))

 {

   if (myproc()->cpuTime >= tickMax[myproc()->q])

   {

     myproc()->cpuTime = 0;

     if (myproc()->q != 4)

     {
       myproc()->q++;
     }
     yield();
   }
   else
   {
     myproc()->cpuTime++;
     myproc()->timeInQ[myproc()->q]++;
   }
 }
#endif
```

- If the process has slept during its execution and the time slice has not expired, reschedule the process to the same queue once its state changes to RUNNABLE.

```
static void
wakeup1(void* chan)
{
 struct proc *p;
 for(p = proc; p<&proc[NPROC]; p++)
 {
   if(p->state == SLEEPING && p->chan == chan)
   {
     p->state = RUNNABLE;
     #ifdef MLFQ
       p->cpuTime = 0;
       p->q_in = 0;
       queue[p->q][qProc[p->q]] = p;
       qProc[p->q]++;
     #endif
   }
 }
}
```

```
// Kill the process with the given pid.
// The victim won't exit until it tries to return
// to user space (see usertrap() in trap.c).
int kill(int pid)
{
 struct proc *p;

 for (p = proc; p < &proc[NPROC]; p++)
 {
   acquire(&p->lock);
   if (p->pid == pid)
   {
     p->killed = 1;
     if (p->state == SLEEPING)
     {
       // Wake process from sleep().
       p->state = RUNNABLE;
       #ifdef MLFQ
         p->q_in = ticks;
         queue[p->q][qProc[p->q]] = p;
         qProc[p->q]++;
       #endif
     }
     release(&p->lock);
     return 0;
   }
   release(&p->lock);
 }
 return -1;
}
```