Assignment-4 : Enhancing XV-6

Gowlapalli Rohit 2021101113
Losetti Mourya 2021101121

Specification 4 : Report

The default scheduler of xv6 is round-robin-based.

The kernel shall only use one scheduling policy which will be declared at compile time, with default being round robin in case none is specified.

To support the SCHEDULER macro to compile the specified scheduling algorithm , CFLAG is introduced in the Makefile as shown

```
xv6-riscv/Makefile
  + ifndef SCHEDULER
  + SCHEDULER:=RR
  + endif
  + CFLAGS+="-D$(SCHEDULER)"
  + ifeq ($(SCHEDULER),MLFQ)
  + CPUS := 1
  + endif
```

- FCFS (First Come First Serve)

  $ make qemu SCHEDULER=FCFS

```
+ xv6-riscv/kernel/proc.h ::
  int timeofcreation;

+ xv6-riscv/kernel/proc.c ::

  void scheduler(void):

  #ifdef FCFS
      struct proc *p;
      struct proc *temp = 0;
      for (p = proc; p < &proc[NPROC]; p++)
      {
        acquire(&p->lock);
        uint64 diff = p->timeofcreation - temp->timeofcreation;
        if (p->state == RUNNABLE && (!temp || diff < 0))
```

```
          {
            if (temp)
            {
              release(&temp->lock);
            }
            temp = p;
          }
          else
          {
            release(&p->lock);
          }
        }
        if (temp)
        {
          temp->state = RUNNING;
          c->proc = temp;
          p->numscheduled++;
          swtch(&c->context, &temp->context);
          c->proc = 0;
          release(&temp->lock);
        }

    #endif

  void allcoproc(void):
    p->timeofcreation = ticks;
```

FCFS implements a policy that selects the process with the lowest creation time (creation time refers to the tick number when the process was created).

The process will run until it no longer needs CPU time.

yield() is disabled in usertrap() & kerneltrap() to prevent the preemption of a process incase of CPU timer interrupts

We iterate through all processes and pick the process that requests the CPU first and hold the resources by not releasing the lock for that process and then context switch to that process , thereby executing it , This implementation is implemented concurrently on 3 Different CPU's

Whenever a process is running , the scheduler always has a lock on that process.

The lock of an acquired process is released only when we find a process with lesser starttime than the current minimum.

After getting the process with the minimum starttime among all the processes, switch the CPU's context with the process's context and change its state to RUNNING and eventually release the lock of the process with minimum starttime.

- LBS (Lottery Based Scheduler)

```
$ make qemu SCHEDULER=LBS
```

```
+ xv6-riscv/kernel/proc.h ::
  int timeofcreation;

+ xv6-riscv/kernel/proc.c ::

  int do_rand(unsigned long *ctx)
  {
    /*
    * Compute x = (7^5 * x) mod (2^31 - 1)
    * without overflowing 31 bits:
    *      (2^31 - 1) = 127773 * (7^5) + 2836
    * From "Random number generators: good ones are hard to find",
    * Park and Miller, Communications of the ACM, vol. 31, no. 10,
    * October 1988, p. 1195.
    */
    long hi, lo, x;

    /* Transform to [1, 0x7ffffffe] range. */
    x = (*ctx % 0x7ffffffe) + 1;
    hi = x / 127773;
    lo = x % 127773;
    x = 16807 * lo - 2836 * hi;
    if (x < 0)
      x += 0x7fffffff;
    /* Transform to [0, 0x7ffffffd] range. */
    x--;
    *ctx = x;
    return (x);
  }

  unsigned long rand_next = 1;

  int rand(void)
  {
    return (do_rand(&rand_next));
  }

  void scheduler(void):

  #ifdef LBS
      struct proc *p;
      for (p = proc; p < &proc[NPROC]; p++)
      {
        acquire(&p->lock);
        if (p->state != RUNNABLE)
        {
          release(&p->lock);
        }
        else
```

```
        {
          int ticktotal = tickettotal();
          if (ticktotal)
          {
            r = (rand()%ticktotal)+1;
          }
          else if (r <= 0)
          {
            r = (rand()%ticktotal)+1;
          }
          r -= p->tickets;
          if (r > 0)
          {
            release(&p->lock);
          }
          else
          {
            if (p->state == RUNNABLE)
            {
              p->state = RUNNING;
              c->proc = p;
              swtch(&c->context, &p->context);
              c->proc = 0;
            }
            release(&p->lock);
          }
        }
      }
  #endif

  void allcoproc(void):
    p->timeofcreation = ticks;

  uint64 tickettotal(void):
  {
    struct proc *p;
    uint64 sum = 0;
    for (p = proc; p < &proc[NPROC]; p++)
    {
      if (p->state == RUNNABLE)
        sum += p->tickets;
    }
    return sum;
  }

  uint64 changetickets(int tickets):
  {
    int pid = myproc()->pid;
    struct proc *p = 0;
    for (p = proc; p < &proc[NPROC]; p++)
    {
      if (p->pid == pid)
      {
        p->tickets = tickets;
```

```
      break;
    }
  }
  return pid;
}

+ xv6-riscv/kernel/sysproc.c

  uint64
  sys_settickets(void):
  {
    int tickets;
    if (argint(0, &tickets) < 0)
      return -1;
    changetickets(tickets);
    return 0;
  }
```

LBS is a preemptive scheduler that assigns a time slice to the process randomly in proportion to the number of tickets it owns.

By default, each process gets one ticket

When the scheduler runs, it picks a random number between 0 and total tickets . It then loops over all the runnable processes and picks the one with the winning ticket.

The system call settickets allows a process to specify how many tickets it wants.

A method similar to sampling a Uniform a random variable via stochastic simulation is employed to operate scheduler()

To schedule a process we sample a number from the uniform random variable , and we compare FX(x) & p->tickets based on which we release locks of all processes that are not picked and schedule the picked process

To generate random numbers , do_rand() and rand() functions are used which are prewritten in kernel/grind.c

The scheduler is tested via lbstest and schedulertest.

- PBS (Priority Based Scheduler)

  $ `make qemu SCHEDULER=PBS`

```
+ xv6-riscv/kernel/proc.h ::
  uint64 numscheduled;
  uint64 staticpriority;
```

```
  uint64 runtime;
  uint64 starttime;
  uint64 sleeptime;
  uint64 totalruntime

+ xv6-riscv/kernel/proc.c ::

void scheduler(void):
  #ifdef PBS
      struct proc *p;
      int cnt = 101;
      struct proc *temp = 0;
      for (p = proc; p < &proc[NPROC]; p++)
      {
        acquire(&p->lock);
        int niceness = 5;
        if (p->numscheduled != 0)
        {
          uint64 var = p->runtime + p->sleeptime;
          if (!var)
          {
            niceness = 5;
          }
          else
          {
            niceness = (p->sleeptime / var);
            niceness = niceness * 10;
          }
        }
        int val = p->staticpriority;
        val = val - (niceness - 5);
        int tmp, processdp;
        if (val < 100)
          tmp = val;
        else
          tmp = 100;
        if (tmp < 0)
        {
          processdp = 0;
        }
        else
        {
          processdp = tmp;
        }
        if (p->state == RUNNABLE)
        {
          if (!temp || (processdp == cnt && p->numscheduled < temp->numscheduled)
|| (processdp == cnt && p->numscheduled == temp->numscheduled && p->timeofcreation
< temp->timeofcreation))
          {
            if (temp)
            {
              release(&temp->lock);
              temp = p;
```

```
              cnt = processdp;
              continue;
            }
            else
            {
              temp = p;
              cnt = processdp;
              continue;
            }
          }
        }
        release(&p->lock);
      }
      if (temp)
      {
        temp->numscheduled++;
        temp->starttime = ticks;
        temp->state = RUNNING;
        temp->runtime = 0;
        temp->sleeptime = 0;
        c->proc = temp;
        swtch(&c->context, &temp->context);
        c->proc = 0;
        release(&temp->lock);
      }
    #endif

    int set_priority(int priority, int pid):
    {
      for (struct proc *p = proc; p < (&proc[NPROC]); p++)
      {
        acquire(&p->lock);
        if (p->pid != pid)
        {
          release(&p->lock);
        }
        else
        {
          int val = p->staticpriority;
          p->staticpriority = priority;
          p->sleeptime = 0;
          p->runtime = 0;
          release(&p->lock);
          int diff = val - priority;
          if (diff > 0)
          {
            yield();
            return val;
          }
          else
          {
            return val;
          }
        }
```

```
    }
    return -1;
  }
void allcoproc(void):
  p->staticpriority = 60;

+ xv6-riscv/user/setpriority.c ::
  #include "kernel/param.h"
  #include "kernel/types.h"
  #include "kernel/stat.h"
  #include "user/user.h"

  int main(int argc, char **argv)
  {
      if (argc - 3)
      {
          fprintf(2, "Wrong no of arguments\n");
          exit(1);
      }
      int priority = atoi(argv[1]);
      if (priority >= 0 && priority <= 100)
      {
          int pid = atoi(argv[2]);
          set_priority(priority, pid);
          exit(1);
      }
      else
      {
          fprintf(2, "Priority isn't within the limit\n");
          exit(1);
      }
  }
```

PBS is a non-preemptive priority-based scheduler that selects the process with the highest priority for execution.

In case two or more processes have the same priority, we use the number of times the process has been scheduled to break the tie.

If the tie remains, use the start-time of the process to break the tie(processes with lower start times should be scheduled further).

To keep track of the total ticks ran and the total ticks for which it was sleeping, we used the variables pbsrtime and pbsstime in struct proc. These values will be updated in update_time().

To keep track of how many times a process was scheduled, we used a variable called numscheduled in struct proc. It is updated everytime a process is scheduled. It is given a default value of 0 in allocproc().

Once we have all the variables tracked, we just implement the logic for the given formulae in the scheduler and make it pick the processes based on the calculated DP value.

We implement a system call named set_priority() which changes the value of the variable static priority.

We can test this scheduler via schedulertest and procdump.

- MLFQ (Multi Level Feedback Queue)

  $ make qemu SCHEDULER=MLFQ

```
+ xv6-riscv/kernel/proc.h ::
  enum queued
  {
    QUEUED,
    DEQUED
  };
  struct proc{
    p->qlevel = 0;
    p->qstate = DEQUED;
    p->qentrytime = 0;
    for (int i = 0; i < QCOUNT; i++)
    {
      p->q[i] = 0;
    }
  }


+ xv6-riscv/kernel/proc.c ::

void initqueuetable(void):
  {
    for (int i = 0; i < QCOUNT; i++)
    {
      queuetable[i].front = 0;
      queuetable[i].back = 0;
    }
  }
void scheduler(void):
  #ifdef MLFQ
    struct proc *p;
    for (p = proc; p < &proc[NPROC]; p++)
    {
      if (p->state == RUNNABLE && ticks >= p->qentrytime + AGE)
      {
        remove(&queuetable[p->qlevel], p);
        p->qlevel = max(0, p->qlevel - 1);
        p->qentrytime = ticks;
      }
    }
      struct proc *p;
      for (p = proc; p < &proc[NPROC]; p++)
      {
        if (p->state == RUNNABLE)
```

```
        {
          if(p->qstate == DEQUED)
          push(&queuetable[p->qlevel], p);
        }
      }
      struct proc *temp = 0;
      for (int i=0;i<5;i++)
      {
        while (empty(queuetable[i]) == 0)
        {
          struct proc *p = pop(&queuetable[i]);
          if (p->state == RUNNABLE)
          {
            temp = p;
            break;
          }
        }
        if (temp)
        i=5;
      }
      if (temp == 0)
      continue;
      p=temp;
      acquire(&p->lock);
      p->scheduletick = ticks;
      p->qentrytime = ticks;
      p->qruntime = 0;
      p->state = RUNNING;
      p->starttime = ticks;
      p->pbsrtime = 0;
      p->pbsstime = 0;
      p->numscheduled++;
      c->proc = p;
      swtch(&c->context, &p->context);
      p->qentrytime = ticks;
      c->proc = 0;
      release(&p->lock);
  #endif

+ xv6-riscv/kernel/queue.c ::

  void push(struct PriorityQueue* q, struct proc* p) {
      q->queue[q->front++] = p;
      if(q->front == QSIZE)
      q->front = 0;
      if (q->front == q->back) {
        panic("Can't push in full queue");
      }
      p->qstate = QUEUED;
    }

  struct proc* pop(struct PriorityQueue* q)
    {
      if (q->back == q->front) {
```

```
      panic("Can't pop empty queue");
    }
    struct proc* p = q->queue[q->back];
    p->qstate = DEQUED;
    if(q->back == NPROC)
    q->back = 0;
    else
    q->back++;
    return p;
  }

void remove(struct PriorityQueue* q, struct proc* p) {
    if (p->qstate == DEQUED)
    {
        return;
    }
    int i = q->back;
    while (i != q->front)
    {
        if (p == q->queue[i])
        {
            p->qstate = DEQUED;
            int j = i + 1;
            while (j != q->front)
            {
                q->queue[(j + QSIZE - 1) % QSIZE] = q->queue[j];
                j = (j + 1) % QSIZE;
            }
            q->front--;
            if (q->front < 0)
                q->front += QSIZE;
            break;
        }
        i = (i + 1) % QSIZE;
    }
  }

int empty(struct PriorityQueue q) {
    if(q.front == q.back)
    {
      return 1;
    }
    else
    {
      return 0;
    }
  }

+ xv6-riscv/kernel/trap.c ::

void usertrap(void):
  if(which_dev==2)
+  {
#ifdef MLFQ
```

```
      struct proc *p = myproc();
      if (p->pbsrtime >= (1 << p->qlevel))
      {
        p->qlevel++;
        if (p->qlevel > 5)
        {
          p->qlevel = 5;
          yield();
        }
        else
        {
          yield();
        }
      }
      int flag = 0;
      for (int i = 0; i < p->qlevel; i++)
    {
      if (empty(queuetable[i])==0)
      {
        flag = 1;
        break;
      }
    }
    if(flag)
    yield();
#endif
 }

void kerneltrap(void):
{
 if (which_dev == 2 && myproc() != 0 && myproc()->state == RUNNING)
{
  +  #ifdef MLFQ
      struct proc *p = myproc();
      if (p->pbsrtime >= (1 << p->qlevel))
      {
        p->qlevel++;
        if (p->qlevel > 5)
        {
          p->qlevel = 5;
          yield();
        }
        else
        {
          yield();
        }
      }
      int flag = 0;
      for (int i = 0; i < p->qlevel; i++)
    {
      if (empty(queuetable[i])==0)
      {
        flag = 1;
        break;
```

```
        }
      }
      if(flag)
      yield();
    +  #endif
  }
```

Created a file queue.c to implement queue-related functions as follows :

- push -> pushes a process onto the specified queue and changes it's state to QUEUED.
- pop -> returns the next process in queue and pops it from the queue as well as changing the state to DEQUED.
- remove -> removes the process specified from the queue specified, and makes it DEQUED.
- empty -> returns whether the specified queue is empty or not.

Created a struct Queue which implements a circular queue of size 5 each, and uses front and back variables to store the front and back of the queue.

Created an array of queues of size 65 to store the queued processes in MLFQ.

initqueuetable() initializes all structs of the queuetable while booting xv6.

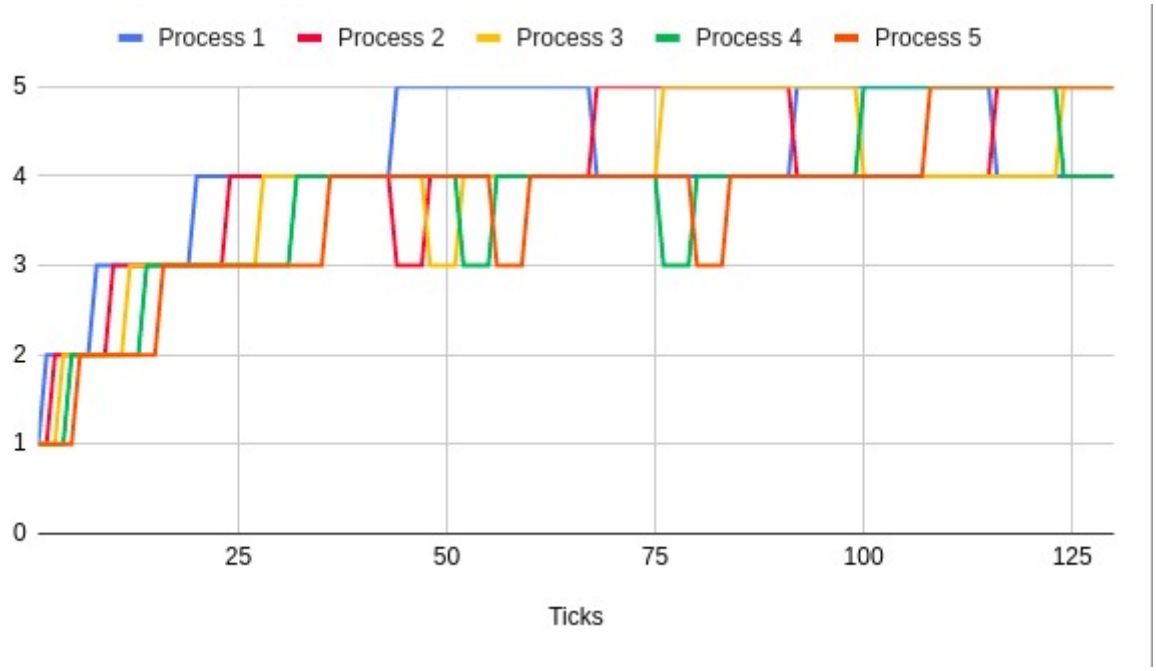
In scheduler() we iterate over processes in proc[NPROC] and insert them into respective queues if RUNNABLE & DEQUED

In trap.c, we preempt the running process if the running time has exceeded (1 << p->qlevel) .If we need to preempt it, we decrease the priority.

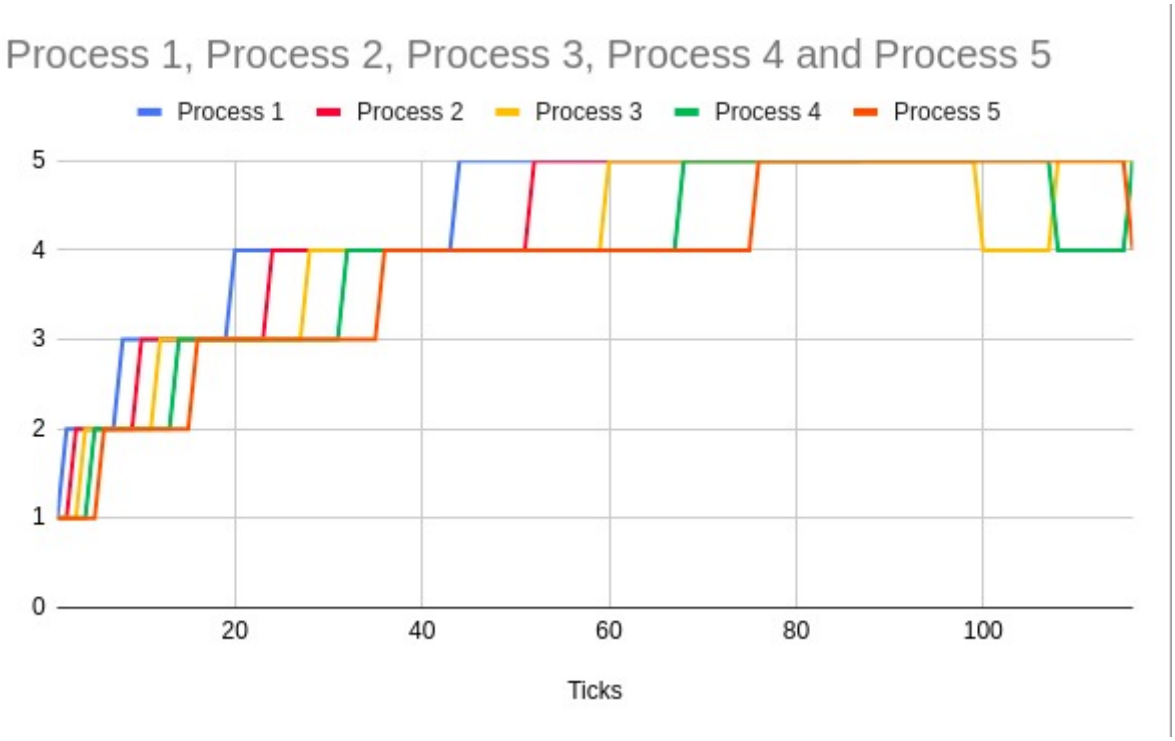
To prevent aging, we check the queueentrytime of every process every time scheduler() is called and we increase priority of the process if the process has been waiting for more than 20 ticks

To preempt processes when a higher priority process is added to the queue, a function getpreempted() checks if the levels above the current processes queuelevel are empty. If any one of them isn't, then we yield in the usertrap and kerneltrap functions on a timer interrupt.

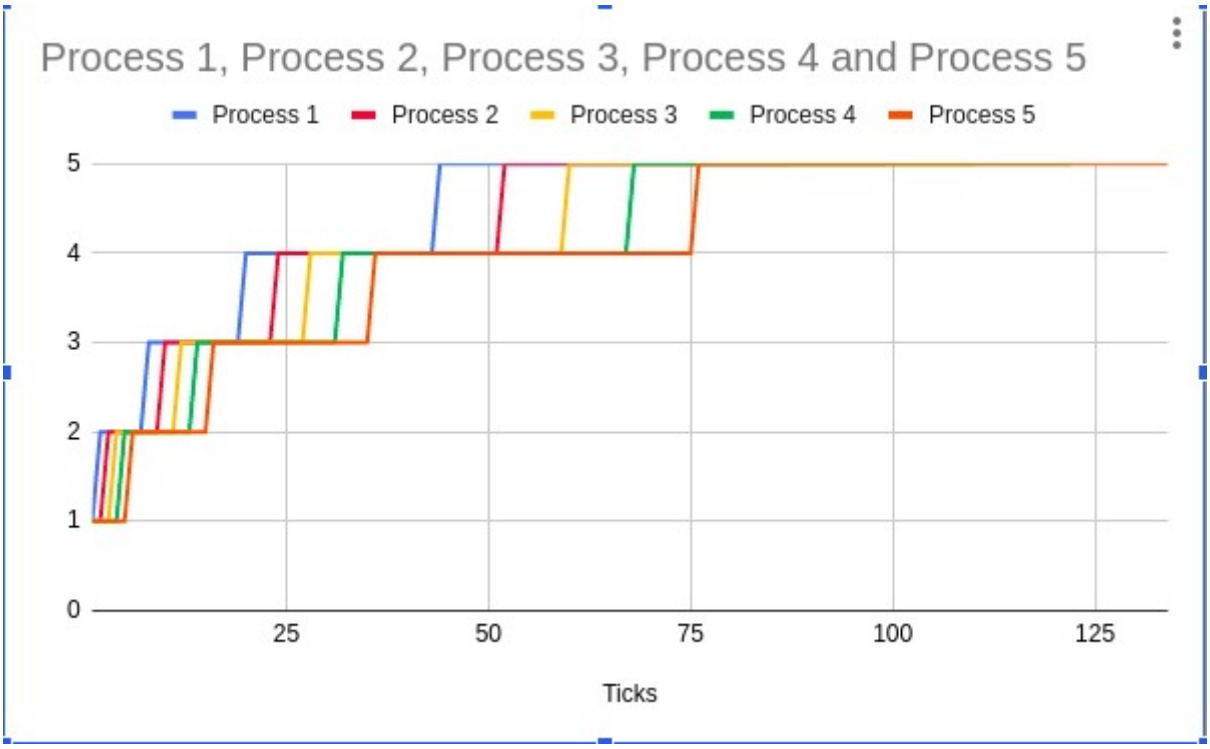
- AGING TIME = 20 Ticks



- AGING TIME = 40 Ticks



- AGING Time = 60 Ticks


Process 1, Process 2, Process 3, Process 4 and Process 5
Process 1
Process 2
Process 3
Process 4
Process 5
5
4
3
2
1
0
25
50
75
100
125
Ticks