Assignment-4

Tags	ASSIGNMENT
contributors	Rhythm , Khushi
Repo containing our working history	https://github.com/RhythmAgg/xv6-adv/tree/master

FCFS scheduler

The policy 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(scheduling is non-preemtive)

- ***Implementation***

▼ Edited the proc.h file for storing the creation time info of each created process in the *struct proc.*

```
struct proc {
  ...
  int intime;
  ...
};
```

▼ added the initialisation of all the new variables in the *allocproc()* function in kernel/proc.c file and similarly set these variables to zero in freeproc()

```
static struct proc* allocproc()
{
  ...
  found:
    p->intime = ticks;
  ...
}
```

▼ In the *scheduler()* in proc.c implemented the FCFS scheduling algorithm

```
void scheduler(){
  struct proc *p;
  struct cpu *c = mycpu();
```

```
    ...
#elif defined(FCFS)
  c->proc = 0;
struct proc* temp = proc;
for(;;){
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();
    uint64 max = 1000000000;

    for(p = proc; p < &proc[NPROC]; p++) {
      // struct proc* pro;
      // for(pro = proc; pro<&proc[NPROC]; pro++) {
      //   if(pro->state == RUNNABLE)
      //   {
      //     if(pro->start_time < max){
      //       max = pro->start_time;
      //       temp = pro;
      //     }
      //   }
      // }
      if(p->state != RUNNABLE) continue;
      acquire(&p->lock);
      if(p->intime <= max)
      {
        temp = p;
        max = p->intime;
      }
      release(&p->lock);
    }
    acquire(&temp->lock);
    if(temp->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.
      temp->state = RUNNING;
      c->proc = temp;
      swtch(&c->context, &temp->context);

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

▼ Since the algorithm is a non-preemtive algorithm, we disable the yield() function calls when there is a timer interrupt in trap.c (usertrap() and kerneltrap());

- ***Average runtime and wait time***

Average Runtime(rtime)	Average waittime(wtime)
27	124

LBS(Lottery Based Scheduling)

Implement a preemptive scheduler that assigns a time slice to the process randomly in proportion to the number of tickets it owns. That is the probability that the process runs in a given time slice is proportional to the number of tickets owned by it.

- ***Implementation***

▼ Edited the proc.h file for storing no. of tickets of each created process in the *struct proc.*

```
struct proc {
  ...
  int tickets;
  ...
};
```

▼ added the initialisation of all the new variables in the *allocproc()* function in kernel/proc.c file and similarly set these variables to zero in freeproc()

```
static struct proc* allocproc()
{
  ...
  found:
    p->tickets = 1; // each process will be given a ticket so that
  ...               // the process doent have zero probability of getting scheduled
}
```

▼ In the *scheduler()* in proc.c implemented the LBS scheduling algorithm

```
void scheduler(){
  struct proc *p;
  struct cpu *c = mycpu();
    ...
  struct proc* temp = proc;
  c->proc = 0;
  for(;;){
    int count_runnable=0;
    acquire(&lock_proc_table);
    for(p = proc; p < &proc[NPROC]; p++) {
      if(p->state == RUNNABLE) count_runnable += p->tickets;
    }
    release(&lock_proc_table);
    int x = (count_runnable > 0) ? (rand() % count_runnable) : 0;
    int prefix = 0;
    // Avoid deadlock by ensuring that devices can interrupt.
```

```
    intr_on();
    for(p = proc; p < &proc[NPROC]; p++) {
      if(p->state != RUNNABLE) continue;
      prefix += p->tickets;
      if(x <= prefix) temp = p;
      else{
        continue;
      }
      acquire(&temp->lock);
      if(temp->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.
        temp->state = RUNNING;
        c->proc = temp;
        swtch(&c->context, &temp->context);
        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;

      }
      release(&temp->lock);
    }
  }
#endif
  ...
}
```

▼ the scheduler first finds all the processes that are currently runnable.then it calls a *rand()* function to select a ticket among these runnable processes. We use prefix sum to calculate which process the ticket belongs to. On finding the process, we schedule the process.

▼ a system call int settickets(int number) , which sets the number of tickets of the calling process. By default, each process should get one ticket; calling this routine makes it such that a process can raise the number of tickets it receives, and thus receive a higher proportion of CPU cycles.

- ***Average runtime and wait time***

Average Runtime(rtime)	Average wait time(wtime)
14	145

PBS(Policy Based Scheduling)

Implement 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).

- ***Implementation***

▼ The Static Priority of a process (SP) can be in the range [0,100], the smaller value will represent higher priority. Set the default priority of a process as 60. The lower the value the higher the priority.

```
struct proc {
  ...
  int static_prio;
  int current_trun;
  int current_tsun;
  int no_of_runs;
  int niceness;
  ...
};
```

▼ added the initialisation of all the new variables in the *allocproc()* function in kernel/proc.c file and similarly set these variables to zero in freeproc()

```
static struct proc* allocproc()
{
  ...
  found:
    p->static_prio = 60; // each process will be given a ticket so that
    p->niceness = 5;
    p->current_trun = 0;
    p->current_tsun = 0;
    p->no_of_runs = 0;
  ...                     // the process doent have zero probability of getting scheduled
}
```

▼ In the *scheduler()* in proc.c implemented the PBS scheduling algorithm

```
void scheduler(){
  struct proc *p;
  struct cpu *c = mycpu();
    ...
  c->proc = 0;
  struct proc* temp = proc;
  for(;;){
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();
    int minDP = 1000;
    for(p = proc; p < &proc[NPROC]; p++) {
      if(p->state != RUNNABLE) continue;
      acquire(&p->lock);
      int min_ = ((p->static_prio - p->niceness + 5) > 100) ? 100 : (p->static_prio - p->niceness + 5);
      int max_ = (0 > min_) ? 0 : min_;
      int temp_dp = max_;
      if(temp_dp < minDP)
      {
```

```
        minDP = temp_dp;
        temp = p;
      }
      else if(temp_dp == minDP)
      {
        if(p->no_of_runs < temp->no_of_runs) temp = p;
        else if(p->no_of_runs == temp->no_of_runs)
        {
          if(p->start_time < temp->start_time) temp = p;
        }
      }
      release(&p->lock);
    }
    acquire(&temp->lock);
    if(temp->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.
      temp->state = RUNNING;
      temp->no_of_runs++;
      c->proc = temp;
      swtch(&c->context, &temp->context);
      temp->niceness = (int)(10 * temp->current_tsun) / (temp->current_tsun + temp->current_trun);
      temp->current_tsun = 0;
      temp->current_trun = 0;
      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&temp->lock);
  }
#endif
  ...
}
```

▼ Dynamic Priority (DP) is calculated from static priority and niceness.The niceness is an integer in the range [0, 10] that measures what percentage of the time the process was sleeping

$$\text{niceness} = \text{Int}\left(\frac{\text{Ticks spent in (sleeping) state}}{\text{Tick spent in (running + sleeping) state}} * 10\right)$$

- Use Dynamic Priority to schedule processes which is given as:

$$DP = \max(0, \min(SP - \text{niceness} + 5, 100))$$

▼ we use variables *current_trun and current_tsun* to store the current running and sleeping times of the processes and use them to calculate/update the niceness of the processes after each scheduling round

▼ to change the static priority of the processes, we implement a system call *set_priority()* *int set_priority(int new_priority, int pid)*

▼ the call will return the old static priority of the process with pid specified, if In case the priority of the process increases(the value is lower than before), then rescheduling is done.

```
uint64
sys_set_priority(void)
{
  int ret = 0;
  struct proc* p;
  int pid;
  argint(1,&pid);
  int new_prio;
  argint(0,&new_prio);
  for(p = proc; p < &proc[NPROC]; p++)
  {
    if(p->pid == pid){
      ret = p->static_prio;
      p->static_prio = new_prio;
      p->niceness = 5;
      break;
    }
  }
  if(ret > p->static_prio) yield();
  return ret;
}
```

▼ to call this system call, we implement a user program *setpriority*

- ***Average runtime and wait time***

Average Runtime(rtime)	Average wait time(wtime)
14	145

MLFQ

Implement a simplified preemptive MLFQ scheduler that allows processes to move between different priority queues based on their behavior and CPU bursts.
If a process uses too much CPU time, it is pushed to a lower priority queue, leaving I/O bound and interactive processes in the higher priority queues.
To prevent starvation, we implement aging.

- ***Implementation***

▼ each process has a variable that stores the current priority of the process. (0→queue 1 , 1→queue 2, 2→queue 3, 3→queue 4, 4→queue 5)

```
struct proc {
  ...
  int qprio;
  int wait_time;
  ...
};
```

▼ added the initialisation of all the new variables in the *allocproc()* function in kernel/proc.c file and similarly set these variables to zero in freeproc(). We put each process in queue-1 initially, thus qprio = 0.

```
static struct proc* allocproc()
{
  ...
  found:
    int qprio = 0;
    int wait_time = 0;
  ...                    // the process doent have zero probability of getting scheduled
}
```

▼ In the *scheduler()* in proc.c implemented the MLFQ scheduling algorithm

```
void scheduler(){
  struct proc *p;
  struct cpu *c = mycpu();
    ...
  struct proc* temp = 0;
    uint64 min_intime = 1000000000;
    int found = 0;
    intr_on();
    for(int i=0;i<4;i++)
    {
      if(q_count[i]>0)
      {
        for(p = proc; p<&proc[NPROC]; p++){
          if(p->qprio != i) continue;
          if(p->state != RUNNABLE) continue;
          if(p->intime < min_intime){
            temp = p;
            min_intime = p->intime;
            found = 1;
          }
        }
        if(found==1) break;
      }
    }
    if(found == 0)
    {
      for(p = proc; p < &proc[NPROC]; p++)
      {
        if(p->state != RUNNABLE) continue;
        else{
```

```
          temp = p;
          found = 1;
          break;
        }
      }
    }
    if(temp!=0 && temp->state == RUNNABLE) {
      acquire(&temp->lock);
      temp->state = RUNNING;
      c->proc = temp;
      temp->wait_time = 0;
      swtch(&c->context, &temp->context);
      temp->current_tsun = 0;
      temp->current_trun = 0;
      temp->wait_time = 0;
      c->proc = 0;
      release(&temp->lock);
    }
  }
#endif
  ...
}
```

▼ to handle the usertraps and kerneltraps when a timer interrupt occurs, we first check if higher priority process is currently runnable or not, if we find such a process we yield() else we run the time slice check.

▼ for time slicing each priority queue is given a time slice after which the processes are preempted.

The time-slice are as follows:

a. For priority 0: 1 timer tick
b. For priority 1: 2 timer ticks
c. For priority 2: 4 timer ticks
d. For priority 3: 8 timer ticks
e. For priority 4: 16 timer ticks

```
trap(){
    int x=0;
    struct proc *p = myproc();
    int prio = p->qprio;
    for(int i=prio - 1;i>=0;i--){
      if(q_count[i] > 0){
          for(struct proc *p = proc; p < &proc[NPROC]; p++){
          if(p->qprio == i && p->state == RUNNABLE){
            yield();
            x=1;
```

```
            break;
          }
        }
      }
      if(x == 1) break;
    }
    if(x == 0){
    if(prio == 0){
      if(p->current_trun % 1 == 0){
        p->current_trun = 0;
        q_count[prio]--;
        p->qprio = 1;
        q_count[1]++;
        p->intime = ticks;
        yield();
      }
    }
    else if(prio == 1){
      if(p->current_trun % 2 == 0){
        p->current_trun = 0;
        q_count[prio]--;
        p->qprio = 2;
        q_count[2]++;
        p->intime = ticks;
        yield();
      }
    }
    else if(prio == 2){
      if(p->current_trun % 4 == 0){
        p->current_trun = 0;
        q_count[prio]--;
        p->qprio = 3;
        q_count[3]++;
        p->intime = ticks;
        yield();
      }
    }
    else if(prio == 3){
      if(p->current_trun % 8 == 0){
        p->current_trun = 0;
        q_count[prio]--;
        p->qprio = 4;
        q_count[4]++;
        p->intime = ticks;
        yield();
      }
    }
    else{
      if(p->current_trun % 16 == 0){
        p->current_trun = 0;
        yield();
      }
    }
    }
}
```

▼ If the process uses the complete time slice assigned for its current priority queue, it is preempted and inserted at the end of the next lower level queue.

▼ If a process voluntarily relinquishes control of the CPU(eg. For doing I/O), it leaves the queuing network, and when the process becomes ready again after the I/O, it is inserted at the tail of the same queue, from which it is relinquished earlier. Inserting at the end means the process has the highest intime in the queue.

▼ A round-robin scheduler should be used for processes at the lowest priority queue.

▼ To prevent starvation, we implement aging by keeping track of the waiting time of the process(time a process spends in the queue waiting to be scheduled)

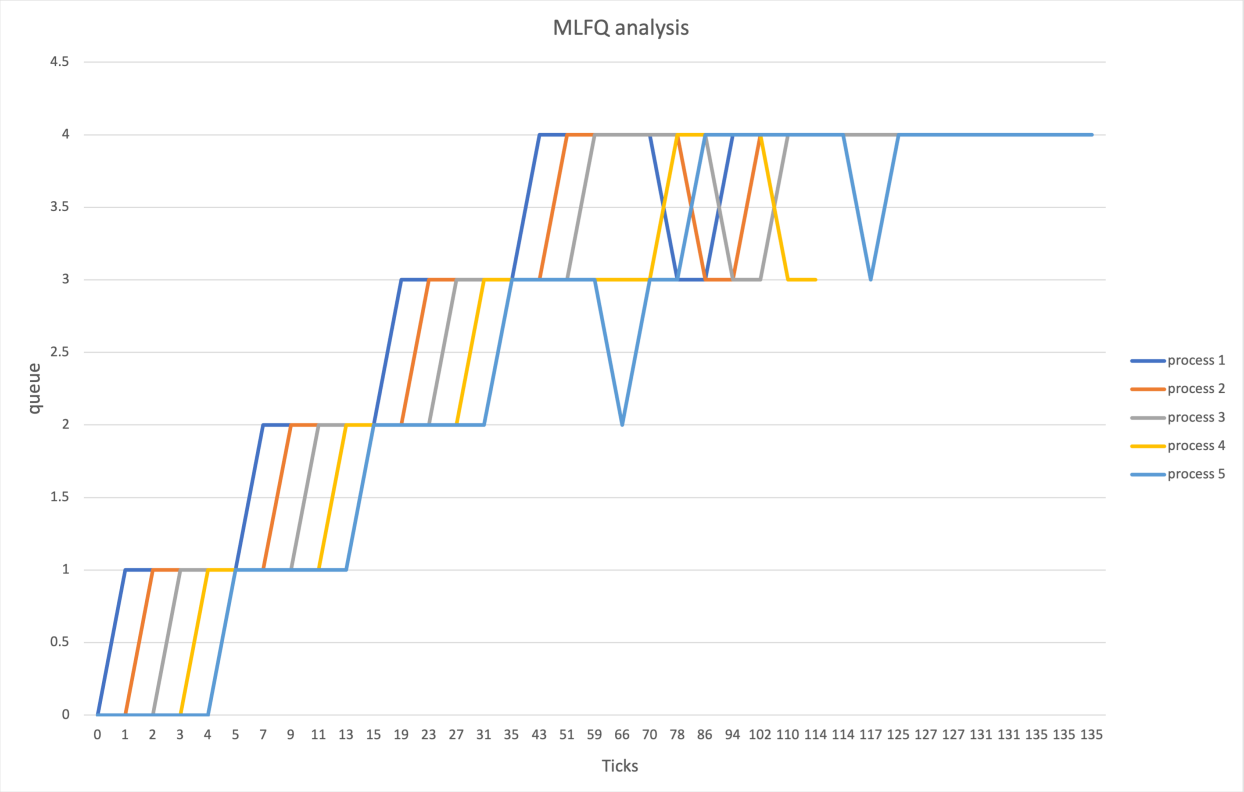
```
update_time(){
...
if(p->state == RUNNABLE)
    {
      if(p != myproc()){
        p->wait_time++;
        if(p->qprio != 0 && p->wait_time > 40)
        {
          q_count[p->qprio]--;
          p->qprio--;
          q_count[p->qprio]++;
          p->intime = ticks;
          p->wait_time = 0;
        }
      }
    }
...
}
```

- ***Average runtime and wait time***

Average Runtime(rtime)	Average wait time(wtime)
13	151

MLFQ ANALYSIS

we use excel to create the graph:



Aging Time - 30

https://iiitaphyd-my.sharepoint.com/:x:/g/personal/rhythm_aggarwal_students_iiit_ac_in/EePyfelDu31HuoWBt638OooBRExEHzJ3W-GjhULOPcWjKQ?e=NaKG9O