Assignment 5 : XV6 OS Modifications

T.H.Arjun

2019111012, CSD

How to Run

1. Run on terminal in this folder, `make clean; make qemu SCHEDULER= <FLAG>`
2. `<FLAG>` can be MLFQ , RR , PBS , FCFS. If the `SCHEDULER` Flag is not used the RR is taken as default.

Overview

All assignment requirements has been done whose implementations have been described below and **Bonus has been done** for which the python code used, data collected and graphs have been included. **An ellaborate report has been also included as Report.pdf** which has performance Evalutions and Plots. Kindly go through those for a deeper understanding. This README tries to deal with implementation and code and instructions, while Report.pdf is a detailed analysis including bonus.

Bonus, Report , Graphs, Performance Analysis

Kindly refer Report.pdf

Implemention

Task 1 :

`int waitx(int* wtime, int* rtime)` syscall and `time` user program

For implementing this I added the fields `int ctime; int rtime; int etime; int iotime;` to proc structure in proc.h .They store the values they mention.

- `ctime` is updated as `ticks` in `allocproc()` where a process is created since this where creation happens.
- `etime` is updated as `ticks` in `exit()` since this is where the process ends.
- `rtime`,`iotime` is updated in `updateRunsandOthers()` which is called from trap.c whenever a tick is increased according to state of process.
- waitx was added to proc.c and it is a modified version of wait and I am updating the following which are passed as arguments as

```
*rtime= p->rtime;
*wtime=p->etime- (p->ctime + p->rtime + p->iotime);
```

Other files were also modified for the same like all other syscalls like sysproc.c , user.h , usys.S , syscall.h, syscall.c , sysproc.c , defs.h , proc.c , proc.h.

`time` command which is a user program is also implemented in time.c which uses this syscall. `time` takes a command and executes it. After that it outputs the run time , waiting time and return value of `waitx`.

`ps` user program

The final modified `proc struct` looks like given below

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int ctime;                   //! for waitx - creation time
  int rtime;                   //! for waitx - totalcputime
  int etime;                   //! for waitx - endtime
  int iotime;                  //! for waitx - iotime
  int priority;                //! priority of process for PBS
  int queue;                   //! queue where the process belongs
  int q_ticks[5];              //! ticks at each q
  int curr_q_ticks;            //! ticks in current queue
  int n_run;                   //! number of runs
  int lastWorkingticks;        //! last tick the process worked at
};
```

The modifications were used to code up `int getprocstable(struct Info_req* InfoTable)` which resides in proc.c and a new structure was made `Info_req` which resides in Info_req.h. The userprogram ps resides in ps.c . The code is well commented and written and doesn't need much explanations. Some fields are only defined some algorithms and these have been handled.

Task 2:

`scheduler` function in proc.c is the home to all scheduler algorithms. As mentioned the `SCHEDULER` flag is used to switch between the modes during compile time. This has been implemented in makefile.

(a) First come - First Served (FCFS)

- Loop through the process table to find the process which is runnable and has least creation time.

```
    struct proc *nextone = NULL; //this is the next process to run
    acquire(&ptable.lock);       //lock table
    //loop through process table to find the process with min creation
time
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if (p->state != RUNNABLE)
        continue;
      if (nextone == NULL)
      {
        nextone = p;
      }
      else if (p->ctime < nextone->ctime)
      {
        nextone = p; //this has lower creation time, change nextone
      }
    }
```

- Take that process and execute.

```
    if(nextone!=NULL){
      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      p=nextone; //p is the next one to run, so change it to nextone
      //oldcode from default
      c->proc = p;
      switchuvm(p);
      p->n_run++;
      p->state = RUNNING;
      p->lastWorkingticks=ticks;

      swtch(&(c->scheduler), p->context);
      switchkvm();
      p->lastWorkingticks=ticks;

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
```

- Since this Algorithm is non-premptive we should not yield in trap.c in case of FCFS. Thus the following has been added to trap.c

```
if(myproc() && myproc()->state == RUNNING &&tf->trapno ==
T_IRQ0+IRQ_TIMER){
    #ifndef FCFS  //if it is not FCFS yield ...
    yield();
    #endif

}
```

(b) Priority Based Scheduler (PBS)

- Default Priority of 60 is assigned to all processes in `allocproc()` function in proc.c where new process is initialised.
- Loop through process table to find the minimum priority

```
struct proc *minone=NULL;//the current minimum
    acquire(&ptable.lock); //lock table
    //loop through process table to find the process with min priority
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      if(minone==NULL){
        minone=p;
      }
      else if(p->priority< minone->priority){
        minone=p; //this has lower priority , change minone
      }
    }
```

- Now we enter, Round Robin mode and execute all processes with this minimum priority, but at each yield we check if a new process with lower priority has entered the process table if so then we exit the Round Robin and do this all again..

```
// now RR for minone priority -let's go!
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      if(p->state==RUNNABLE && p->priority==minone->priority){ //if p is
runnable and it has the minimum priority run in RR style
            // Switch to chosen process.  It is the process's job
            // to release ptable.lock and then reacquire it
            // before jumping back to us.
            c->proc = p;
            switchuvm(p);
            p->n_run++; //update
            p->state = RUNNING;
            p->lastWorkingticks=ticks;

            swtch(&(c->scheduler), p->context);
            switchkvm();
            p->lastWorkingticks=ticks;

            // Process is done running for now.
            // It should have changed its p->state before coming back.
            c->proc = 0;
            // here the process got yielded - 1 tick over
            int flag =0; //flag to check if better process is there.. if
it is there break from RR
            for(struct proc *nextone = ptable.proc; nextone <
&ptable.proc[NPROC]; nextone++){
              if(nextone->state != RUNNABLE){
                continue;
              }
              if(nextone->state==RUNNABLE && nextone->priority<minone-
>priority){
                flag=1; //better is one is here... stop RR
                break;

              }

            }
            if(flag){
              break; //hey break RR go back to normal mode
            }

      }

}
```

- The priority of a process can be in the range [0,100], the smaller value will represent higher priority.
- To change priority of a process a new system call has been added called `int set_priority(int new_priority, int pid)` as per assignment requirements in proc.c
- set_priority() calls yield() when the priority of a process becomes lower than its old priority as per assignment requirements.

Following is the code snippet for the same.

```
int set_priority(int new_priority, int pid){

  if (new_priority <= -1 || new_priority > 100)  //error handling
        return -1;
  int returnValue=-1;   //error code
  int shouldYield=0;    //variable which tells if it should yield or not
  struct proc *p;
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if(p->pid == pid)
        {
            acquire(&ptable.lock);
            returnValue = p->priority;
        p->priority = new_priority;
      if (returnValue > new_priority) //in this case yield
              shouldYield= 1; //may be that process has a chance at this
CPU so yield...
            release(&ptable.lock);
            break;
        }
    }
  if(shouldYield){
    yield(); //yield
  }
  return returnValue; //return value of set_priority

}
```

- A new user program `setPriority` was created according to the requirements of Assignment which resides in setPriority.c . The code for this is self explanatory.

(c) Multi-level Feedback Queue Scheduling (MLFQ)

- There are 5 queues with different priorities based on time slices, i.e. 1, 2, 4, 8, 16 timer ticks. queue 0 is highest priority and queue 4 is lowest priority
- Each queue has an aging criteria.

These are shown below:

```
  int max_tick_of_q[5]={1,2,4,8,16};  //! Max time slices allowed in queues
  int q_max_time[5]={10,10,15,20,30}; //! Max time  at each queue
```

- In my implementation I am mainting a `queue` variable field in `proc struct` to simulate queue. It holds the value for the queue to which the process belongs to.
- We all age processes as shown below first to avoid starvation. For aging we do `ticks-(p->lastWorkingticks` in which `lastWorkingticks` is the last ticks at which the process was running or sleeping. We check this with the aging factor of each queue. If it aged we do `queue--` if it is not in queue 0 , and we update `lastWorkingticks`.

```
    struct proc * nextone=NULL; //this is the nextone to execute
    acquire(&ptable.lock);
    struct proc *queues[5] = {0};  //this will hold process chosen from each
  queue
      //next we age all procesess to avoid starvation
    for(p = ptable.proc;p < &ptable.proc[NPROC];p++){
        if(p->state != RUNNABLE)
        continue;

        if(p->queue > 0 && (ticks-(p->lastWorkingticks)) > q_max_time[p-
  >queue] ){
          p->queue--;
            if(PLOT){
              cprintf("%d %d %d\n", ticks, p->pid, p->queue);
            }
          p->lastWorkingticks = ticks;
        }
      }
```

- We loop through the whole process table and find possible candidates for each queue as follows.

```
    //get runnable processess for each queue
    for(p = ptable.proc;p < &ptable.proc[NPROC];p++){

      if(p->state != RUNNABLE){
        continue;
      }

        if(p->queue != 4){
          if(queues[p->queue] == 0){
            queues[p->queue] = p;
          }
          else if(p->ctime < queues[p->queue]->ctime && queues[p->queue]-
>state == RUNNABLE){ //Priority to creation time
            queues[p->queue] = p;
          }
        }

        else{
          if(queues[p->queue] == 0){
            queues[p->queue] = p;
          }

          else if(queues[p->queue]->lastScheduleTime > p-
>lastScheduleTime){  //RR for 4th queue as mentioned in assignment PDF
            queues[p->queue] = p;
          }
        }
    }
```

- In the first 4 queues creation time is given priority for selection.
- The 5th queue/ last queue uses a RR style. To simulate this I am using a variable field called `lastScheduleTime` which is the last clock tick at which this process got hold of the CPU.
- After we have chose the canditate we go through the queues and execute the best canditate untill it moves out of RUNNABLE state or it finished it's time slice as this algorithm is non-preemptive at each clock till. If its time slice got over we stop executing this process and update all values and demote it. Then we restart the whole scheduling. If it relinquished CPU before it finished its time slice we don't demote just update the values and reschedule. The following code snippet explains this part.

```
for (int i = 0; i < 5; i++)
    {
      if (queues[i] != 0)
      {
        queues[i]->curr_q_ticks = 0;          //we are gonna get some ticks
in this queue
        queues[i]->n_run++;                   //increase runs
        while (queues[i]->state == RUNNABLE) //run till its RUNNABLE and
hasn't reached its time slice
```

```
        {
          nextone = queues[i];
          c->proc = nextone;
          nextone->lastWorkingticks = ticks;
          switchuvm(nextone);
          nextone->state = RUNNING;
          swtch(&(c->scheduler), nextone->context);
          switchkvm();
          c->proc = 0;

          if (nextone->curr_q_ticks >= max_tick_of_q[i]) //if it finished
its time slice break from loop...it got preempted
            break;
        }

        if (queues[i]->curr_q_ticks >= max_tick_of_q[i] && i != 4) //it
used up its time slice.. move to lower priority queue
        {
          queues[i]->queue++;
          if (PLOT)
          {
            cprintf("%d %d %d\n", ticks, queues[i]->pid, queues[i]-
>queue);
          }
        }
        queues[i]->lastWorkingticks = ticks;
        break;
      }
    }
```

Testing

For Testing code, for Scheduling Algorithms I have created a new user program called load implemented in load.c which has various test cases.

Test Cases Included are:

1. Normal Benchmark that is given code for n procs
2. Benchmark for n CPU Bound Processes
3. Benchmark for n IO Bound Processes
4. The intelligent Process test. A process which gives up CPU during its calculations to stay in higher Queue.. Only useful for MLFQ Testing.
5. Mixture of IO Bound and CPU Bound Process
6. A modified version of given Benchmark
7. Here we show why FCFS fails if many Process IO Heavy and they come first. First n/2 are IO bound and n-n/2 is CPU bound but IO is made first

Usage : `load <testnumber> <number_of_procs>`

Please refer the code of the test cases for more details.

Instructions for Bonus - Plotting Graphs

The bonus has been also implemented. Instructions on how to plot is here:

1. Put the macro `PLOT` defined in `types.h` to 1 to collect data to plot.
2. Redirect the output of the VM in qemu to a file by doing `make clean; make qemu SCHEDULER=MLFQ > out.txt`
3. Run the test as `load <testnumber> <number_of_procs>`
4. It will print the necessary data which will be redirected to out.txt
5. Make the necessary changes in plot.py . It is well commented with instructions. It is also easily understandable code.
6. The file uses matplotlib python library which needs to installed on the system.

The Plotted graphs are included in the graphs folder. Also the graphs are included in Report.pdf

Performance Tests

For the performance tests you can use the same macro plot. We will use the last print by the benchmark program usually pid 3 if ran first. We subtract the ticks at the time of creation of this PID and from its exit tick. That is the amount of ticks for which the test took. Now we can compare the performances of these algorithms. **The comparisons have been done and has been included in Report.pdf**