An Upgrade to the exisiting XV6

Original Readme can be found here

To run

```
 make clean
make SCHEDULER=<FCFS/PBS/MLFQ>
make qemu
#or
make qemu-nox # To run in the same terminal
```

> If SCHEDULER argument is not provided or any wrong value is passed, The default Round Robin Scheduling is done.

Achieved through the upgrade

- Addition of Schedulers
  - FCFS Scheduler
  - PBS Scheduler (Priority Based)
  - MLFQ Scheduler (**M**ulti **L**evel **F**eedback **Q**ueue)
- Extending the process structure with
  - wait time
  - run time
  - creation time
  - end time
- Custom waitx system call mimicking wait but assigning values of runtime and wait time to argument passed.
- ps, a user function which displays details of all processes in proc table.

Let's look at all changes in detail

waitx system call

It is exactly similar to the wait system call already used, but in addition it takes in two integer arguments `wtime` and `rtime`
to which the values of the total wait time and total run time of process are assigned respectively.
To do this proc structure has been extended with:
`wtime, ctime, rtime, twtime, iotime`
In every call of trap.c at the moment tick is incremented updateRuntime Function is invoked

```
void updateRuntime(void){
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state==RUNNING){
      p->rtime++;
      #ifdef MLFQ
      p->mlfq_time--;
      p->queue[p->cur_q]++;
      #endif
    }
    if(p->state==RUNNABLE){
      p->wtime++;
      p->twtime++;
    }
    if(p->state==SLEEPING)
      p->iotime++;

  }
  release(&ptable.lock);
}
```

If process is * sleeping iotime is incremented * running runtime is incremented * runnable waittime and totatl waitime is incremented * If MLFQ the the time stayed in mlfq queue is decremented and num ticks in that queue for process is incremented.

> We assume only a process in runnable state to be waiting, a sleeping process does not count for wait as it is a voluntary exit. wtime is reset to 0 everytime the scheduler picks it up or every time it changes queue

ps

A system call was created getPinfo which was called through ps.c.
It prints all details of processes as:

```
 PID     Priority        State   r_time  w_time  n_run   cur_q   q0      q1      q2      q3      q4
1       60              sleep   2       0       14      1       1       1       0       0       0
2       60              sleep   3       0       24      2       1       2       0       0       0
15      60              run     1       0       2       1       1       0       0       0       0
5       60              sleep   1       0       12      1       1       0       0       0       0
10      76              runble  0       0       954     0       0       0       0       0       0
11      75              sleep   0       0       955     0       0       0       0       0       0
12      74              sleep   0       0       955     0       0       0       0       0       0
```

It extracts the details from the extended proc structure and prints it.

> wtime is majorly shown 0 as the temporary wait time is too fast to be noticed and printed.

Scheduler

Code:

```
 struct proc *p;
struct cpu *c = mycpu();
c->proc = 0;

for(;;){
  // Enable interrupts on this processor.
  sti();
  /**
   * The code here would be provided for each respective scheduler below
   * Excluding the default Round Robin
   */
}
```

FCFS Scheduler

The code:

```
 struct proc *fp=0;
// Loop over process table looking for process to run.
acquire(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE)
    continue;
  if(!fp)
    fp=p;
  else if(p->ctime<fp->ctime){
    fp=p;
  }
}
if(fp){
  p=fp;
  // 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->state = RUNNING;
  p->n_run+=1;
  p->wtime=0;
  swtch(&(c->scheduler), p->context);
  switchkvm();

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

A small change done in trap.c

```
 #ifndef FCFS
#ifndef MLFQ
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
if(myproc() && myproc()->state == RUNNING &&
  tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
#endif
#endif
```

If FCFS scheduling is happening then the process won't be yielded as we are aiming at non-pre emptive FCFS Scheduling.
Once a process completes execution the next process with smallest creation tim is executed

> The same has been done for MLFQ as the time after which it yields is dependednt on the queue it is in hence a slightly different version has been implemented.

Priority Scheduler

The code:

```
struct proc *hp=0;
  // Loop over process table looking for process to run.
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
      continue;
    if(!hp)
      hp=p;
    else if(p->priority<hp->priority){
      hp=p;
    }
  }
  if(hp){
    p=hp;
    // 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->state = RUNNING;
    p->n_run+=1;
    p->wtime=0;
    swtch(&(c->scheduler), p->context);
    switchkvm();

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

As each process is yielded after every tick we check for the highest priority process exisiting and start running it.
The Priority Scheduler implemented is pre-emptive in nature.

MLFQ Scheduler

The code:

```
int mlfqTime[]={1, 2, 4, 8, 16};
    struct proc *hp=0;
    acquire(&ptable.lock);

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      if(!hp)
        hp=p;
      else if(p->cur_q<hp->cur_q){
        hp=p;
      }else if(p->cur_q==hp->cur_q){
        if(p->mlfqprior<hp->mlfqprior)
          hp=p;
      }
    }
    if(hp){
      p=hp;
      // Swi3.0tch 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->state = RUNNING;
      p->n_run+=1;
      p->wtime=0;
      swtch(&(c->scheduler), p->context);
      switchkvm();

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

The MLFQ Scheduling has been implemented without actually having to use the queues.
With Inspiration from nextPid, we store a custom proc struture array of 5 that stores the highest position in that respective virtual queue.
Whenever a new process enters a queue, the process gets a queue number and the position alloted to it in queue, and the value in structure array is incremented.
The same is donr when a process exits from a queue.
If a process decides to sleep it is just readded back into queue by provided next largest index hence being the last in the queue to be executed.
while scheduling, First the lowest queue no process is selected if their are multiple processes in the lowest queue, the process having the smallest queue position is executed first.

Code for Aging and Exceeded timeslice

```
// Check if any process has crossed it's time in queue
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state!=RUNNABLE)
        continue;
      if(p->mlfq_time<=0){
        if(p->cur_q!=4){
          //cprintf("\nprocess %d exceeded time slice moving to lower queue %d, wtime so far=%d\n", p->pid, p->cur_q+1, p->wtime);
          p->cur_q++;
          p->mlfq_time=mlfqTime[p->cur_q];
          p->mlfqprior=queue[p->cur_q].curSize++;
          p->wtime=0;
          //cprintf("%d,%d,%d\n", p->pid, ticks, p->cur_q);
        }
      }
    }

    // Aging
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
      if(ticks - p->inqueuefrom > 1500){
        if(p->cur_q!=0){
          //cprintf("\nprocess %d Aged moving to higher queue %d, wtime so far=%d\n", p->pid, p->cur_q-1, p->wtime);
          p->cur_q--;
          p->mlfq_time=mlfqTime[p->cur_q];
          p->mlfqprior=queue[p->cur_q].curSize++;
          p->wtime=0;
          //cprintf("%d,%d,%d\n", p->pid, ticks, p->cur_q);
        }
      }
```

The above code has been implemented inside MLFQ Scheduler code.
It has been separated here for the sake of convinience.

For Aging A constant time equivalent to 1500 ticks, as soon as a process is added to queue the current ticks is stored. and at every tick it is checked if it has exceeded, if it did then it's priority will be increased by shifting it to a higher priority queue.
Nothing is done if it is already in queue 0.
For Exceeded Time we use 1, 2, 4, 8 as time limits for each queue 0, 1, 2, 3 if a process exceeds that time limit for the queue it'll be demoted to next lowest queue.
Nothing is done if it is already in queue 4.

Exploit in MLFQ

One exploit that I can think from out of my mind is, if a child of a process is able to figure out the time in the highet priority queue, every child in the process could voluntarily relinquish just a minuscle time before that and by doing so can continue to exist in the highest priorirty queue always executing first.
And in case of a Malicious program can also be used to crdate an infinite loop and crash.

Comparision

This has been done using `time benchmark` in each instance with different scheduler. The results are as Follows:

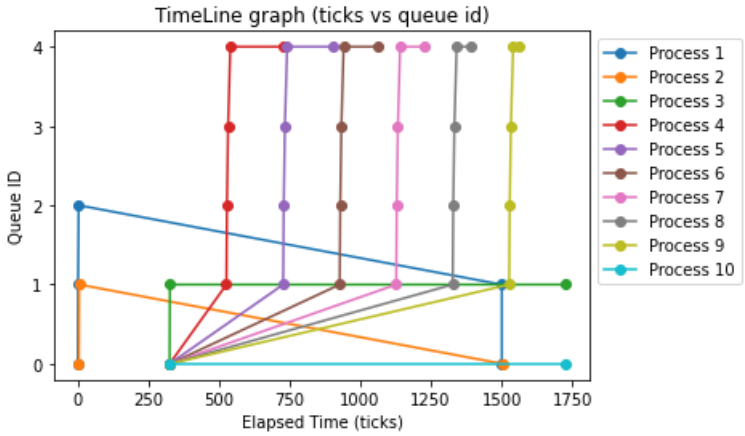
Scheduler	Total Time	Run Time
RR	2005	3
PBS	2004	2
FCFS	2243	3
MLFQ	2005	3

From our testing we conclude that PBS is the fastest where as FCFS is the slowest.
Though if we have a process where prioririte are being set wrong, PBS won't be of much help
For us:

```
PBS < (RR=MLFQ) < FCFS
```

The above is likely to chnage depending on the benchmarking program used for testing.

Bonus



The graph has been created using matplotlib in python

```
 #!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Nov  2 16:10:04 2020

@author: vjspranav
"""

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import time

dataset = pd.read_csv('test.csv')
li=dataset.pid.unique()
ds=dataset.groupby(dataset.pid)
fig, ax = plt.subplots()
plt.title('TimeLine graph (ticks vs queue id)')
for pid in li:
    xi=ds.get_group(pid).sort_values(['x']).x.to_numpy()
    yi=ds.get_group(pid).sort_values(['x']).y.to_numpy()
    print("Plotting ", xi, yi, 'For ', pid)
    ax.plot(xi, yi, '-o', label='Process '+str(pid))
ax.legend(loc="upper left", bbox_to_anchor=(1,1))
plt.yticks(range(0,5))
plt.xlabel('Elapsed Time (ticks)')
plt.ylabel('Queue ID')
plt.savefig('Timegraph.png', bbox_inches='tight')
plt.show()
```

The benchmark was run with print statements printing out the pid queueid and ticks
Using those values graph has been plotted.