Implementations

Prerequisite
Ensure qemu is installed on the system.
To install qemu:

```
     $ sudo apt-get install qemu
```

For running the program run the command

```
  > make qemu [SCHEDULER = POLICY]
```

The POLICY used can be either FCFS, RROBIN , MLFQ or PBS. However, please note that, before switching between policies, do "make clean" once

Task 1

Waitx sys call

The primary aim is to append to the existing structure of proc, a set of time keeping variables, namely:

```
int priority;                     //0-100;
 int st_t;
 int end_t;
 int io_t;
 int run_t;
 int run_num;
 int e_t;
 int cq;
 int ticks[5];
```

Consequently, in order to extract these time variables to use them while implementing the scheduling algorithms mentioned below, we define a new syscall, *waitx.*
Note that the moment a new process is created, the creation-time is set. As time progresses, after each time-tick, the run-time is incremented. And finally, end-time is updated when the process ceases to exist.

```
for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        release(&ptable.lock);

        *run_t = p->run_t;
        *wait_t = p->end_t - (p->st_t + p->run_t + p->io_t);
        return pid;
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids || curproc->killed){
      release(&ptable.lock);
      return -1;
    }
```

The function accepts two arguments: integer pointers to wait-time and creation-time of the process and returns the pid corresponding to the process

Getpinfo

The objective of this function is to return a basic information about a process. Provided a PID, the function should be able to retrieve the process name, total run time of the process, the total number of times the process has executed, the current queue of the process (for MLFQ), and the number of times the process has executed in each queue. For this we declared a new struct, called proc_stat,

```
struct proc_stat{
    int pid;            //PID of the process
    int runtime;        //total run time of the process
    int num_run;        //number of times the process has run in total
    int ticks[5];       //no of ticks in each queue level
    int ql;             //current queue level
};
```

The function itself takes a struct proc_stat* argument to store the values, and a int pid to find the correct process. It then loops through the process table to find the required process, and returns 1 if corresponding process is found, 0 otherwise.

```
int
getpinfo(struct proc_stat* p,int pid)
{
  struct proc* checker;
  for(checker=ptable.proc;checker<&ptable.proc[NPROC];checker++)
//search
  {
    if(checker->pid==pid)    //if found
    {
      p->pid=checker->pid;
      p->num_run=checker->run_num;
      p->runtime=checker->run_t;
      return 1;     }
  }
  return 0;
}
```

FCFS

For this, the basic concept is that we want to choose a runnable process that has the lowest start time, ie it was initiated first and hence needs to be dealt with first as per first come first serve to schedule.

We first pick a runnable process from the process table (ptable) and make sure that that process has the lowest start time if there is a process that has a lower start time we pick that process
(we have to make sure we acquire the lock for the ptable , since it is a critical section)

This also requires us to add the notion of start time , so we add that to the `struct proc` in `proc.h`

```
acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
      continue;
    fcfs = p;
    c->proc = p;
    // highest priority choosing
    for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++) { // choose
a runnable process with the lowest start time ie , it started first
      if(p1->state != RUNNABLE){
        continue;
      }
      if( p1->startTime < fcfs->startTime)
      {
        fcfs = p1;
      }
    }
```

We then make a context switch to that process

```
    p = fcfs;
    c->proc = p;
    p->timesrun++;
    switchuvm(p);
    p->state = RUNNING;
    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;
```

Priority Based Scheduler

The basic idea of a priority-based scheduler is to select the process with the highest priority for execution. We add a new parameter called `priority [int]` to `struct proc` in `proc.h`

We also create a new function called `set_priority` which takes pid and priority as arguments and sets the pid with that priority.

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

 acquire(&ptable.lock); //acquire lock for ptable
 for(p=ptable.proc;p<&ptable.proc[NPROC]; p++){ //iterate over the
processes
   if(p->pid==pid) // find the one with the pid we are looking for
   {
     p->priority=priority; // set its priority to the priority we want
     break;
   }
 }
 release(&ptable.lock); // release lock
 return pid;
}
```

Also, in function `allocproc(void)` we set the default priority as 60 (`p->priority = 60;`)

Priority scheduler is similar to FCFS in the sense that there we checked for the lowest start time in FCFS , here we check for the highest priority in the list of runnable process and then context switch to that process.

```
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
     if(p->state != RUNNABLE)
       continue;

     highP = p;

     for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
       if(p1->state != RUNNABLE)
         continue;
       if(highP->priority > p1->priority)
         highP = p1;
     }
     p = highP;

     // Switch to chosen process.  It is the process's job
     // to release ptable.lock and then reacquire it
     // before jumping back to us.
    //preparing for context switch
      c->proc = p;
     switchuvm(p);
     p->state = RUNNING;

     swtch(&(c->scheduler), p->context); //saves the current registers
in cpu scheduler
     switchkvm();

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

 }
```

Round Robin Scheduling

The basic concept behind round-robin scheduling is that a particular time slice is allocated to each process called the time quantum. If the execution of the process gets completed in that time quantum, the process will terminate otherwise the process will go to the ready queue again.

Multi-Level Feedback Queue Scheduler

Multilevel Feedback Queue Scheduling (MLFQ) keep analyzing the behavior (time of execution) of processes and according to which it changes its priority.
A process in the lower priority queue can suffer from starvation due to some short processes taking all the CPU time.

Comparison

Through manually trying out cases I have seen the following observations on comparing the implemented scheduling algorithms :

- MLFQ and default scheduling process are faster than rest of algorithms.
- MLFQ algorithm, though difficult to implement, is faster than rest in almost all cases and gives **best response time** and throughput.
- FCFS is the **slowest** of all implementations.
- PBS algorithm is optimised for **real time** applications.