Things included by me in xv6 operating system.

- **Waitx** system call

It takes wtime and rtime returns same as wait
Test: Run time <wait time> <run time>
Its implemented in proc.c and the variables involved are
ctime (creation time)
iotime(I/O time)
etime (end time)
these were added into proc.h and updated in the proc.c

When the process is created ctime set to ticks and iotime in sleep state of the process in trap.c
etime while the process exits is stored and rtime when the process is in RUNNING state in trap.c
wtime is updated by subtracting iotime,ctime,rtime from etime.

Prints wait time and run time of the executed process in the test.

- **Ps** system call

Here we print
pid
priority which we set to 60 by default when the process born.
p->priority=60;
Run time
Wait time in the begining of the process I fixed a number around 10 and this sets to zero when the process enter to a respective Schedule or when the queue is changed.
n_run it is number of times the process is picked in the schedule , its the updated in every process.
When the process is running state regardless of schedule .
p->num_run++;
Current queue in the case of MLFQ only says the currently assogned queue.
Five Qi's gives number of times the process received ticks by each queue.

Task 2

In the Scheduler function of proc.c
Each schedule flag is implemented
Deafult one is RR Round Robin
Here the process is assigned when the time slice expires thus every process in the ready queue gets executed.

FCFS
First Come First Serve
Here the process with least creation time is fixed.So, shorter burst time need to wait if longer one comes first.

```
#ifdef FCFS
    struct proc *min_process = p;

    if (p->state != RUNNABLE)
      continue;
    p->waittime =0;
    for (cp_p = ptable.proc; cp_p < &ptable.proc[NPROC]; cp_p++)
    {
      if (cp_p->state != RUNNABLE)continue;
      if (cp_p->ctime < p->ctime) min_process = cp_p;
    }
    p = min_process;

#endif
```

PBS

Priority Based Scheduler

So , on the priority base the process is picked up and given tthe CPU .

Also the system call set_priority is implemented which needs to arguments to be passed the new priority and pid of the process .

Test for set_priority: run the setPriority <priority number> <pid>

```
#ifdef PBS

    struct proc *highest_priority = 0;
    struct proc *p1 = 0;
    if (p->state != RUNNABLE)continue;
    p->waittime=0;highest_priority = p;
    for (p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++)
    {
      if ((p1->state == RUNNABLE) && (highest_priority->priority > p1->priority))
        highest_priority = p1;
    }

    p = highest_priority;
```

```
int set_priority(int priority,int pid)
{
  struct proc *p;
  int to_yield = 0, old_priority = 0;
  if(priority<0 || priority>100)return -3;

  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if(p->pid == pid)
    {
      to_yield = 0;
      acquire(&ptable.lock);
      old_priority = p->priority;p->priority = priority;
      cprintf("\033[0;34mChanged priority of process %d from %d to %d\n", p->pid, old_priority, p->priority);
      if (old_priority > p->priority)to_yield = 1;
      release(&ptable.lock);
      break;
    }
  }
    if (to_yield == 1)yield();
    return old_priority;
}
```

MLFQ

Multi-Level Feedback Queue

Here are the 5 queueswith different time slices (1,2,4,8,16)

Among them highest priority is for 0 and lowest is 4.

So the new will be started in Queue[0] only.
if the receiving number of ticks are more than the queue then the few ticks (permissible number) are downgraded to lower queue. Thus starvation is avoided in ageing function for all the queue.

```
void ageing(void)
{
  //checking queue
  for(int index=1;index<=4;index++)
  {
    for(int i=0;i<=queue_popu[index];i++)
    {
      if(queue[index][i]->state!=RUNNABLE)continue;
      if((ticks-queue[index][i]->enter)>queue[index][i]->waittime)
      {
          cprintf("%s %d switching to queue %d  as waittime %ds exceeded\n\033[0m","\033[1;32m",queue[1][i]->pid,index-1,(
          queue_popu[index-1]++;
          queue[index][i]->cur_queue--;
          queue[index][i]->ticks[queue[index][i]->cur_queue]=0;
          queue[index][i]->enter=ticks;
          queue[index-1][queue_popu[index-1]]=queue[index][i];
          queue[index-1][i]-> waittime =0;
          remove_proc_from_queue(queue[index][i]->pid,queue[index][i]->cur_queue);
      }
    }
  }
}
```

If ticks are fine in number then iterate and increment ticks and number of runs.

Also the ticks for updated in trap.c file

```
  case T_IRQ0 + IRQ_TIMER:
    if (cpuid() == 0)
    {
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      // int o = ps(); //for mlfq graphs
      release(&tickslock);
      if (proc)
      {
        if (myproc()->state == RUNNING)
        {
          myproc()->rtime++;
          myproc()->ticks[proc->cur_queue]++;
        }
        else if (proc->state == SLEEPING)
          proc->iotime++;
      }
    }
```