ENHANCING XV6

ADDED FEATURES

For the assignment I have modified the process structure to include the essential features.

```
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)

  // adding required fieds
  int rtime;                   // total time
  int runtime;                   // running time
  int wtime;                   // total wait time
  int waittime;                // waiting time
  int ctime;                   // creation time
  int etime;                   // end time
  int priority;                // Priority
  int iowaittime;              // iowaittime
  int q[6];                    // Time in seconds
  int cur_q;                   // Current queue
  int n_run;                   // Number of times picked by scheduler
  int qentertime[6];            // for finding the front process of a queue
  int lastentertime;            // Time when it last got the CPU
  int currentslice;             // Time for which the process is executing
};
```

WAITX SYSCALL

We have added a new system call '''C int waitx(int *wtime, int *rtime) . The waitx syscall is similar to wait system call, it makes the parent wait for the child to finish its execution and the returns the total run time and the total wait time(RUNNABLE state). For the implementation I have added a attrbute ctime, rtime, wtime to the process struct and also added a few other such as iotimetime, cpuwaittime. I have updated the dynamic values of the struct by calling function "updateruntime" from the file trap.c. I also added the statements and declarations where they are required. When the waitx syscall is called I wait until I find a child process state as "ZOMBIE" then by using the values wtime , rtime i return required values.For using

and testing this I used time.c file (user program) so that the rtime and wtime of the given process will be displayed.

--> 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 updateruntime() 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

```
    *waittime = (p->wtime);
    p->rtime =  p->runtime + p->waittime + p->iowaittime;
    *runtime = p->runtime;
```

PS SYSCALL

It returns a detailed list of processes present in the system at that particular instant. It loops through the ptable to obtain the details of the processes. It make use of the process struct to obtain the required information.

TASK-2 : SCHEDULING

First come First served(FCFS)

The FCFS serves the process that arrives firstThe CPU doesnot get relinquised until it encounters I/O or exits. We are following non-premptive policy. Hence if a process with a longer CPU burst time arrives first and one with a shorter CPU burst time arrives next, the second process will have to wait till the longer process is done (in case of 1 CPU). Processes are picked based on least creation time. Priority is assigned based upon the arrival time.

We choose the process with minimum create time(ctime) and execute it till it either goes to sleep or exits. For preventing the call to yield (which premptes the process) we are using ifdef condiitons.

Priority Based Scheduling

Every process is assigned a priority of 60. We loop through the ptable and find the minimum of the process priorities. If more the one processes of equal priority exists then we execute round robin robin.The processes with higher priority are chosen and given CPU attention before the ones with lower priority. (A lower value of priority indicates a higher priority). The set_priority syscall is used to change the priorities of the process.

The priority of the can be in the range[1,100] .set_priority() calls yield() when the priority of a process becomes lowerthan its old priority as per assignment requirements

```
#ifdef PBS

  int find = 0;
  //struct proc *  pbsp =0;
  int minpriority = 1000;
```

```
  //int minnpid = 100000000;
  acquire(&ptable.lock);
  // Finding the process with minimum priority

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE) // The process has to be runnable to
schedule
          continue;
        else
        {
            if (p->priority < minpriority)
            {
            find = 1;
    //              pbsp = p;
                minpriority=p->priority;
  //          minnpid = p->pid;
            }
        }
  }
  if(find == 1) // If there is any process
  {
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if(p->state !=RUNNABLE)
    {
        continue;
    }
    // we schedule them only if they have minimum priority and they
    // are in runnable state.
    if(minpriority == p->priority && p->state == RUNNABLE)
    {
        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;
        p->n_run++;
        p->wtime += p->waittime;
        p->waittime = 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;
        struct proc * isitmin = 0;
        int ff = 0;
        // To include premption
        // We check if any higher priority process has entered
        for(isitmin = ptable.proc; isitmin < &ptable.proc[NPROC];
isitmin++)
    {
     if(isitmin->state != RUNNABLE)
             continue;
          else
          {
```

```
            if (isitmin->priority < minpriority)
            {
    //      pbsp = p;
              ff =1;
              break;
    //      minnpid = p->pid;
            }
          }
      }
        if(ff == 1)
        {
          ff = 0;
          break;
        }
    }

  }
  }
  else
  {
      release(&ptable.lock);
      continue;
  }

  release(&ptable.lock);

#endif
```

MULTI-LEVEL FEED BACK QUEUE

We are maintaining 5 queues with the highest priority being number as 0 and bottom queue with the lowest priority as 4.. Each queue support different silces i.e 1,2,4,6,8 ticks for the queues 1 to 5 respectively. I am implementing in a NON_PREMPTIVE fashion. Each queue has a demotion techmique if the time spent by a process in a queue crosses a permisable ticks of the queue we denote the process to a lower queue.

A process always starts in Q0. If the process used the complete time slice assigned for its current priority queue, it is pre-empted and inserted at the end of the next lower level queue. For the last level queue Q4 we are using Round Robin scheduling. To simulate this we are using a field 'qentertime' and iowaittime these fields give wait when the process last executed. Based on this field we go through the whole queue on and on.The process executes until it completes it time slice. After the completion of time slice we check for demotion and go -on to schedule another process.

After 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.We put it at the end of queue . To simulate this i am using the lastenter which denotes the last executable time. Then we restart the whole scheduling. If itrelinquished CPU before it finished its time slice we don't demote just update the values and reschedule

```
// CODE FOR DEMOTING A PROCESS
 if(bestprocess->currentslice >= maximumof[minqueue] && bestprocess->state
!= ZOMBIE && bestprocess->cur_q > -1 )
      {
        bestprocess->currentslice = 0;
        if(minqueue >= 4)
        {
          bestprocess->qentertime[minqueue] = ticks;
          bestprocess->waittime = 0;
            p->wtime += p->waittime;
        }
        else
        {
          bestprocess->cur_q = bestprocess->cur_q + 1;
            #ifdef YES
              cprintf("%d %d %d\n",ticks, bestprocess->pid,bestprocess-
>cur_q);
            #endif
          bestprocess->qentertime[minqueue] = ticks;
            bestprocess->lastentertime = ticks;
          bestprocess->waittime = 0;
            p->wtime += p->waittime;
        }
        break;
      }
      if(bestprocess->state == SLEEPING)
      {
          bestprocess->currentslice = 0;
        bestprocess->qentertime[minqueue] = ticks;
        bestprocess->waittime = 0;
          p->wtime += p->waittime;
        break;
      }
      aging();
    }
    // bestprocess->lastentertime = ticks;
    break;
    }
```

There is function aging which helps in prevention of starvation. We maintain a array agefactor[5] . agefactor[i] denotes the maximum waittime before it gets promoted to queue i-1.

```
void aging()
{
  int agefactor[6];
  agefactor[0] = 90;
  agefactor[1] = 90;
  agefactor[2] = 90;
  agefactor[3] = 90;
  agefactor[4] = 80;
  struct proc * pp;
```

```
  //  acquire(&ptable.lock);
   for(pp = ptable.proc; pp< &ptable.proc[NPROC]; pp++)
   {
     if(pp->state!= RUNNABLE || pp->cur_q < 0)
       continue;
     if(pp->waittime >= agefactor[pp->cur_q])
     {
       if(pp->cur_q >= 1)
       {
         pp->cur_q = pp->cur_q - 1;
           #ifdef YES
            cprintf("%d %d %d\n",ticks, pp->pid,pp->cur_q);
           #endif
         pp->currentslice = 0;
           pp->lastentertime = ticks;
           pp->wtime += pp->waittime;
         pp->waittime = 0;
         pp->iowaittime = 0;
         pp->qentertime[pp->cur_q] = ticks;
       }
     }
   }
  // release(&ptable.lock);
 }
```

TESTING

I have made benchmark2 program which is used for testing of algorithms.

for benchmark2 program we have to give 2 parameters one for determine which type of processes should be taken. O for the processes which suits the MLFQ better and 1 for the processes which are CPU bound and 2 for the process which are I/O bound and 3 for the process which are mixed process.

The second argument will be the number of processes which are to be run.

E.g : benchmark2 <test_number> <number_of_processes>

BONUS

1. I have used PLOT to select whether we want to do plotting or not. if PLOT = YES we will print the pid, ticks and the queue number for every queue change so that we get the values to plot. e.g: make qemu SCHEDULER=MLFQ PLOT=YES > graphfile.txt
2. We output the values to graphfile.txt file and make changes to the file.
3. I am using matplotlib (python) to get the graphs.

I have included the analysis of various scheduling algorithms and graphs in report.pdf.