waitx sys call:

I implemented this using waitx sys call and I created time.c which uses this syestem call. I added few more attributes in the proc structure. Few of them are as follows:

```
  int ctime;                  // Creation time
  int etime;                  // end time
  int rtime;                  // total time
  int wtime;                  // wait time
```

I aslo added the statements when required in the files where syscall are declared. I did this by searching for a syscall and added accordingly the rest of the places I added the new syscall. 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.

Files changed:

```
syscall.h
Added part---------->#define SYS_waitx  22

defs.h
Added part---------->int   waitx(int* , int*);

user.h
Added part---------->int waitx(int*, int*);

sysproc.h
Added part---------->
int
sys_waitx(void)
{
  int* wtime;
  int* rtime;
  if(argptr(0,(void*)&wtime,sizeof(int))<0||argptr(1,
(void*)&rtime,sizeof(int))<0)
  {
    return -1;
  }
  // cprintf("%p *** %p\n",wtime,rtime);
  return waitx(wtime,rtime);
}

usys.S
Added part---------->SYSCALL(waitx)

syscall.c
Added part---------->
extern int sys_waitx(void);
[SYS_waitx]   sys_waitx,
```

```
proc.c
I added the functionality.
```

In proc.c I used the given wait function in the proc.c I extended it by returing the rtime and wtime to the calling funtion by equlaing it to the passed addresses. I have implemented time.c in which I used this waitx syscall. These rtime and wtime are getting updated at the time ticks are getting updated so that we use the same clock for everything.

ps:

I implemented ps syscall and created a user program ps.c to test it. Similar to the previous syscall I added the necessary statments for this syscall as well. This will feature all the required information as shown in the pdf. I extracted the required values of the field from the structure of the proc. In case of staets other than RUNNING and RUNNABLE all others will have current queue to be -1 since it is unwise of talking about the same. Some times the n_run and rtime maynot always hold the consition rtime >= nrum becuase the time for which the process runs may be less than tick and it may be the case that updation of nrun happens first than the rtime because we kept lock so this relation may not hold always.

```
void
ps()
{
  struct proc* p;
  acquire(&ptable.lock);
    cprintf("PID \t Priority \t State \t \t r_time \t w_time \t n_run \t
cur_q \t q0 \t q1 \t q2 \t q3 \t q4 \n");
  for(p=ptable.proc;p< &ptable.proc[NPROC]; p++)
  {
     if(p->state==UNUSED)
    continue;
    // p->agingtime=ticks - p->lastvisit;
    if(p->state==RUNNABLE)
    cprintf("%d \t %d \t \t RUNNABLE \t %d \t \t %d \t \t %d \t %d \t %d \t
%d \t %d \t %d \t %d\n", p->pid, p->priority, p->rtime, p->agingtime, p-
>n_run,p->cur_q,p->q[0],p->q[1],p->q[2],p->q[3],p->q[4]);
    if(p->state==RUNNING)
    cprintf("%d \t %d \t \t RUNNING \t %d \t \t %d \t \t %d \t %d \t %d \t
%d \t %d \t %d \t %d\n", p->pid, p->priority, p->rtime, p->agingtime, p-
>n_run,p->cur_q,p->q[0],p->q[1],p->q[2],p->q[3],p->q[4]);
    if(p->state==SLEEPING)
    cprintf("%d \t %d \t \t SLEEPING \t %d \t \t %d \t \t %d \t -1 \t %d \t
%d \t %d \t %d \t %d\n", p->pid, p->priority, p->rtime, p->agingtime, p-
>n_run,p->q[0],p->q[1],p->q[2],p->q[3],p->q[4]);
    if(p->state==EMBRYO)
    cprintf("%d \t %d \t \t EMBRYO \t %d \t \t %d \t \t %d \t -1 \t %d \t
%d \t %d \t %d \t %d\n", p->pid, p->priority, p->rtime, p->agingtime, p-
>n_run,p->q[0],p->q[1],p->q[2],p->q[3],p->q[4]);
    if(p->state==ZOMBIE)
    cprintf("%d \t %d \t \t ZOMBIE \t %d \t \t %d \t \t %d \t -1 \t %d \t
%d \t %d \t %d \t %d\n", p->pid, p->priority, p->rtime, p->agingtime, p-
>n_run,p->q[0],p->q[1],p->q[2],p->q[3],p->q[4]);
```

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

FCFS:

This is the simplest algorithm we need to schedule the process with least arrival time to kernel. For this I maintained a varianble which tells me the arrival time ctime. This is made equal to tick in allocproc and then I will choose the process with minimum ctime among all RUNNABLE processes in the ptable. Difference beteween rest of the algorithms and this is that we will not yield() it after every colock tick this will be run as long as it wants. It will relinquish only in case of its completion or for for I/O operation.

```
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    int current=0;                // This stores the min ctime values.
    struct proc * curr=0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)// checking if the process is RUNNABLE.
        continue;
      curr=p;                 // This loop is for finding some base current
and curr value.
      current=p->ctime;
      break;
    }
    for(p=ptable.proc ; p<&ptable.proc[NPROC];p++)
    {
      if(p->state!=RUNNABLE)
        continue;
      if(current>p->ctime)      // Checking if we find a process with
lesser ctime.
      {
        current = p->ctime;     // in that case we will update the current
min value.
        curr=p;
      }
    }
    if(curr==0)
    {
      release(&ptable.lock);     // If we didnt find any process then we
will release the lock
      continue;
    }
    p=curr;                      // else curr will the process to be run.
    // 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 = p->n_run +1;      // Updating the number of times the
process was run.
```

```
    swtch(&(c->scheduler), p->context);
    switchkvm();
    p->lastvisit=ticks;          // updating the last visit time.
    // Process is done running for now.
    // It should have changed its p->state before coming back.
    c->proc = 0;
    release(&ptable.lock);
```

```
int current=0;
    struct proc * curr=0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      curr=p;
      current=p->lastvisit;
      break;
    }
    for(p=ptable.proc ; p<&ptable.proc[NPROC];p++)
    {
      if(p->state!=RUNNABLE)
        continue;
      if(current>p->lastvisit)
      {
        current = p->lastvisit;
        curr=p;
      }
    }
```

The above code represents the code for finding the minimum arrived time process.

Priority Based Scheduler:

In priority based scheduler we need to find the process with least priority number (highest priority) if there are more than one then we scheduling based on RR for this I found the min of priority of all RUNNABLE processes using this priority I will be able to schedule the required process.

```
// Loop over process table looking for process to run.
    acquire(&ptable.lock);
    int current=101;                          // This stores the highest
priority number(leastnumber)
    for(p = ptable.proc; p<&ptable.proc[NPROC];p++){
      if(p->state != RUNNABLE)                // Checking if the process
is RUNNABLE.
        continue;
      current=p->priority;                    // If we found one then I
set that to be the starting val.
      break;
    }
    for(p=ptable.proc ; p<&ptable.proc[NPROC];p++)
```

```
    {
      if(p->state!=RUNNABLE)
        continue;
      if(current>p->priority)
      {
        current = p->priority;                  // changing the min if we
see a better one.
      }
    }
```

Then I will loop over the ptable processes and serch for the process with min priority once I find it I will schedule that process. After that I will look for the process with higher priority process than this If I find one I will leave it else I will loop over the process for some process with the same priority as before. This ensures that the Round Robin is taken care if the priorities are same since wqe are looping in circular fasion over the ptable. Please see the commented code snippet for better understandong.

```
for(p=ptable.proc ; p<&ptable.proc[NPROC];p++)
    {
      if(p->state!=RUNNABLE)                    // checking if it is
RUNNABLE.
      continue;
      int tester=0;
      if(p->priority==current)
      {
        // 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 = p->n_run +1;                 // Updating the number of
times it ran in the CPU.
        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;
        for(p=ptable.proc;p<&ptable.proc[NPROC];p++)// looping through the
ptable processes.
        {
            if(p->state!=RUNNABLE)continue;
            if(p->priority<current)             // Checking if there is any
better process with better priority.
            {
              tester=-1;                        // If we find such a
process then we no longer continue.
              break;
            }
        }
        if(tester==-1)                          // If we didnt find any
then we will schdule based on RR
```

```
        break;                                      // Which inturn depends on
the ctime.
      }                                             // so we continue to look
for the process as current in the ptable and this will continue untill we
find a better process or we complted looping in the ptable once.
      if(tester==-1)
      break;
    }
```

MLFQ:

In MLFQ we have queues and the processes in it may starve so for this purpose I have implemented aging time after which the process upgrade to their higher priority queue. We will update these operations when we are scheduling, we will check if there is any process whose waiting time for the cpu slice is greater than a limit if it is the case then we will change its queue and will reset the aging time. We will aslo change the last visit time to the instant when there is a queue change this is to make sure that we track the time when the process entered the current queue.

```
  for(p=ptable.proc;p<&ptable.proc[NPROC];p++) // looping through all the
process in the ptable.
  {
    if(p->state == RUNNABLE && p->cur_q!=-1&&p->cur_q!=0)// making sure
that the process si Runnable.
    {// We need to check the aging time only when the process is RUNNABLE
in all other cases the aging time would be 0.
      if(ticks-p->lastvisit >= 20)          // Agingtime is current time -
time at which it entered the queue
      {
        p->currentslice=0;                  //  Time spent in the execution
is set to 0.
        p->lastvisit=ticks;                 //  Time at which the process
entered the queue is updated.
        p->agingtime=0;                     //  Time for which it was
waiting is 0.
        p->cur_q=p->cur_q-1;                //  We upgraded the process to
its higher priority queue.
        #ifdef YES                          //  This is used to print graph
        cprintf("%d %d %d\n",ticks,p->pid,p->cur_q);
        #endif
      }

    }
  }
```

I added the following to the proc structure. This code is well commented please go through the code snippet.

```
  int q[5];                    // Number of ticks it got in queue i
throughout its execution period.
```

```
  int agingtime;               // This is used for aging purpose, to
maintain the time for which the
                               // process has been waiting in the current
queue.
  int lastvisit;               // This is used for calculating the time at
which the process
                               // process entered the queue. Used for
calculating aging time.
  // int agecal;                  // This will represent the last time the
process is in queue.
  int currentslice;            // Time spent in the current queue while the
process is RUNNING.
```

Once we know all the meanings of the variables we are good to go for logic. After shuffling the processes correctly as per the aging time now we need to select a process. For selecting a process among many available, the process is to check if there is any process in the zeroth queue if it does we will schedule a process in queue0 but if there are no processes then we will go to the next queue ie queeu1 in this way we need to go through the the processes queue wise to schedule them.

```
  for(int i=0;i<5;i++)                        // looping from higher
priority queues to lower.
  {
    int current=-1,flag=-1;                   // current is used to store
the least visit time.
    for(p=ptable.proc;p<&ptable.proc[NPROC];p++)  // among all the
processes with cur_q i.
    {                                         // looping through all the
processes in the ptable.
      if(p->state != RUNNABLE)                // We will scheule the process
only if it is RUNNABLE.
          continue;
      if(p->cur_q!=i)                         // Checking if the process
belongs to the same queue.
          continue;
      flag=1;
      if(current==-1)                         // If this is the first
process then the min will be its
      {                                       // lastvisit.
        current = p->lastvisit;
      }
      if(current>p->lastvisit)                // If there is a process withe
last visittime before this
      {                                       // Then we will change the
current.
        current=p->lastvisit;
      }
    }
```

Once we get the process with the minimum last visit time it is the one which should be executed. For this purpose I went through all the processes again and checked for this process and updated its states correctly as shown below.

```
      if(p->state!=RUNNABLE)                // Checking if the process is
RUNNABLE.
        continue;
      if(p->cur_q!=i)                       // Checking if it belongs to
current queue of interest.
        continue;
      if(current!=p->lastvisit)             // Checking if the prcoess has
the lastvisit as minimum in all
        continue;
      p->currentslice=0;                    // If we find one such process
p then we will reset the time
                                            // spent in execution
environment tp 0.
      c->proc = p;                          // We are merking that the CPU
is executijng this process.
      p->n_run = p->n_run +1;               // Chenging the number of times
the process ran on CPU.
```

Now we need to check if the process consumes its maximum alloted time in each queue ie 1 ticks in queue 0 and 2 ticks in queue 1 and so on. For this I checked it using the current slice varaible which maintains the number of ticks that the process took for the process in the current queue slice. If it exceeds then I demoted the queue and then changed the currentslice and rest of the variables accordingly. If that is not the case then I continued running the same process untill it completes its slice or it relinquishes the CPU on its own.

```
while(p->currentslice<(int)pow(2,p->cur_q)&&p->state==RUNNABLE)
      {
        p->agingtime=0;                     // While the process is RUNNONG
we agingtime is 0.
        switchuvm(p);
        p->state = RUNNING;
        swtch(&(c->scheduler), p->context);
        switchkvm();
        p->lastvisit=ticks;                 // lastvisit of the process is
aslo the current time.
        p->agingtime=0;
        if(p->currentslice>=(int)pow(2,p->cur_q))
        {                                   // If the time spent in current
execution time is greater
          if(p->cur_q!=4&&p->cur_q>=0)
          {
                                            // than the maximum alloted
then the proces's queue is
                                            // decremented unless it is in
last queue.
```

```
            p->cur_q=p->cur_q+1;            // Inthat case we incresed the
queue.
            #ifdef YES                      // For graph plotting purpose.
            cprintf("%d %d %d\n",ticks,p->pid,p->cur_q);
            #endif
          }
          p->currentslice=0;                // Reseting the current slice.
          p->lastvisit=ticks;               // lastvisit time of the
process is updated.
          p->agingtime=0;                   // aging time of the process is
updated.
          // flag=2;
          break;
        }
        // break;
      }
```

Assumptions: cur_q incase of states other than RUNNABLE and RUNNING are -1.