University of Macau

SFTW330

OPERATING SYSTEM II

TEAM PROJECT: SCHEDULING

Supervisor:

Miss Zhuang yan

Team Members:

Li Chao Zheng da729032
Cao Han da728190
Cheng xi da728093

Part 1

Round Robin

(1) Description of round robin:

We use technique of time slicing to reduce the penalty that short jobs suffer with FCFS is to use preemption based on a clock. We called such method round robin.

(2) General flow of program and descriptions on used programming algorithms:

(i). First, we get the input queue from the input file.

And initialize dispatcher queues.

Fill input queue from dispatch list file.

```
    while ( fscanf ( fp , "%d , %d , %d" , &arrivalt, &prior, &cput) != 
EOF ) {
        tempPcb = createnullPcb();
        tempPcb->arrivaltime = arrivalt;
        tempPcb->remainingcputime = cput;
        tempPcb->priority = prior;
        pcbQ = enqPcb(pcbQ , tempPcb);
    }
```

(ii). Start dispatcher timer. Then unload any of the queues or there is a currently running process.

LOOP:Dequeue process from input queue and enqueue on RR queue while head of process arrival time is smaller than dispatcher timer.

```
    while ( rQ || pcbQ || activePcb ) {
    //Unload any pending processes from the input queue:
        while( pcbQ->arrivaltime <= dispatchertime ){

            rQ = enqPcb(rQ, deqPcb(&pcbQ));
        }
```

(iii).If a process is currently running: decrement process remainingcputime, if times up free up process structure memory, else if other processes are waiting in RR queue, enqueue it back on RR queue;

```
    if ( activePcb ) {
            //Decrement process remainingcputime and check it is time up 
or not
            if ( --(activePcb->remainingcputime) <= 0 ) {
                //Terminate it and free up memory
                terminatePcb(activePcb);
                free(activePcb);
                activePcb = NULL;
            }
            //else if other processes are waiting in RR queue:
            else if ( rQ /*&& activePcb->remainingcputime > 0*/ ){
                activePcb = suspendPcb( activePcb );
```

```
            rQ = enqPcb( rQ, activePcb );
            free(activePcb);
            activePcb = NULL;
        }

    }
```

(iv). If no process currently running and RR queue is not empty, then dequeue process from RR queue. If the process is suspended , restart it else start it. In the other hand, set it as currently running process. Sleep for one second then increment dispatcher timer.

```
if( !activePcb && rQ ){
    // if suspend
    activePcb = deqPcb(&rQ);
    if( activePcb->status == PCB_SUSPENDED ){

        startPcb(activePcb);
        joblist_id[j++] = activePcb->pid;
    }
    else{

        startPcb(activePcb);
        joblist_id[j++] = activePcb->pid;
    }
```

(v). Then go back to LOOP.

(3) Some explanations about important section of programming codes in achieve certain tasks.

(i).
```
if( activePcb && activePcb->remainingcputime > 0 && dispatchertime % 4 != 0 ){
 startPcb(activePcb);
 activePcb->remainingcputime--;
 sleep(1);
 dispatchertime++;
 continue;
 }
```

Meaning: The function of this part is to restart processes when the slice time is longer than 1 second.

(ii).
```
activePcb = deqPcb(&rQ);
        //if suspend
        if( activePcb->status == PCB_SUSPENDED ){
            startPcb(activePcb);


        }
        else{
            startPcb(activePcb);
            joblist_id[j++] = activePcb->pid;
```

```
        }
    }
```

Meaning: This part is used to choose which process to be the activePcb.

(4). **Techniques in handling exceptions.**

(i). file-open exception: avoid errors because of system.

```
scanf( "%s" , filename ) ;
fp = fopen( filename , "r" );
if (fp == NULL) {
    perror(" Input File ");
    exit(1);
}
```

(ii). if at the specific instant, no process can be dispatched to run, we print "Idle" instead.

```
if (job_per_s[i] == -10)
        printf("Idle ");
```

(5). **Parameter modification:**

Users can input the slice time themselves in the beginning of the program. Round robin is based on FCFS. The difference is that round robin uses a RR queue to store processes that is suspended when one piece of slice time is finished, but if we choose one sufficiently big as time quantum of Round Robin, then it can perform FCFS as well.

(6). **Achievement in program requirements.**

The output is correct and the quantum can be defined at the beginning of the program. Different processes have different colors in its background.

(7). **Further implementations and what you have learnt.**

I learnt how the round robin and fcfs algorithm work. Also, I know the difference between round robin and fcfs, and how to complete concrete realization.

(8). **Reference:**

A clock interrupt is generated at periodic intervals. When the interrupt occurs, the currently running process is placed in the ready queue, and the next ready job is selected on a FCFS basis. This technique is also known as time slicing, because each process is given a slice of time before being preempted.

Round robin is particularly effective in a general-purpose time-sharing system or transaction processing system.

(9). **Snapshots:**

(i). F**iles included for successful program execution:**

For Round Robin, we need pcb.h pcb.c round.c.

(ii). **Command typed in compiling your program:**

gcc –o process sigtrap.c

gcc –o round round.c

(iii). **Command typed in executing your program:**

```
[da72809@umacnx3 ~/aaaa]$ ./round
```

(iv) **Snapshots to illustrate how you achieve the project requirements.**

```
Input the silce time = 1
Input File = rr.txt
   pid arrive  prior    cpu  status
 11150      0      3      3  RUNNING
 11150: START
 11150: tick 1
 11150: tick 2
 11150: SIGTSTP
   pid arrive  prior    cpu  status
 11151      2      3      6  RUNNING
 11151: START
 11151: tick 1
 11151: SIGTSTP
 11150: SIGCONT
 11150: tick 3
 11150: SIGINT
 11151: SIGCONT
 11151: tick 2
 11151: SIGTSTP
   pid arrive  prior    cpu  status
 11154      4      3      4  RUNNING
 11154: START
 11154: tick 1
 11154: SIGTSTP
 11151: SIGCONT
 11151: tick 3
 11151: SIGTSTP
   pid arrive  prior    cpu  status
 11155      6      3      5  RUNNING
 11155: START
 11155: tick 1
 11155: SIGTSTP
 11154: SIGCONT
 11154: tick 2
 11154: SIGTSTP
 11151: SIGCONT
 11151: tick 4
 11151: SIGTSTP
   pid arrive  prior    cpu  status
 11162      8      3      2  RUNNING
 11162: START
 11162: tick 1
 11162: SIGTSTP
 11155: SIGCONT
 11155: tick 2
```

```
    pid arrive  prior    cpu  status
  11162      8      3      2  RUNNING
  11162: START
  11162: tick 1
  11162: SIGTSTP
  11155: SIGCONT
  11155: tick 2
  11155: SIGTSTP
  11154: SIGCONT
  11154: tick 3
  11154: SIGTSTP
  11151: SIGCONT
  11151: tick 5
  11151: SIGTSTP
  11162: SIGCONT
  11162: tick 2
  11162: SIGINT
  11155: SIGCONT
  11155: tick 3
  11155: SIGTSTP
  11154: SIGCONT
  11154: tick 4
  11154: SIGINT
  11151: SIGCONT
  11151: tick 6
  11151: SIGINT
  11155: SIGCONT
  11155: tick 4
  11155: tick 5
  11155: SIGINT
)ispatcher is free at DispatcherTime = 21.
[he finish time is 20 s.
[he job list is:1 1 2 1 2 3 2 4 3 2 5 4 3 2 5 4 3 2 4 4
```

The result of Round Robin based on SPN.

```
[da72809@umacnx3 ~/aaaa]$ ./round
 Input the silce time = 4
 Input File = rr.txt
    pid arrive  prior    cpu  status
  11189      0      3      3  RUNNING
  11189: START
  11189: tick 1
  11189: tick 2
  11189: tick 3
  11189: SIGINT
    pid arrive  prior    cpu  status
  11192      2      3      6  RUNNING
  11192: START
  11192: tick 1
  11192: tick 2
  11192: tick 3
  11192: tick 4
  11192: SIGTSTP
    pid arrive  prior    cpu  status
  11193      4      3      4  RUNNING
  11193: START
  11193: tick 1
  11193: tick 2
  11193: tick 3
  11193: tick 4
  11193: SIGINT
    pid arrive  prior    cpu  status
  11194      6      3      5  RUNNING
  11194: START
  11194: tick 1
  11194: tick 2
  11194: tick 3
  11194: tick 4
  11194: SIGTSTP
  11192: SIGCONT
  11192: tick 5
  11192: tick 6
  11192: SIGINT
    pid arrive  prior    cpu  status
  11196      8      3      2  RUNNING
  11196: START
  11196: tick 1
  11196: tick 2
  11196: SIGINT
  11194: SIGCONT
  11194: tick 5
  11194: SIGINT
Dispatcher is free at DispatcherTime = 21.
The finish time is 20 s.
The job list is:1 1 1 2 2 2 2 3 3 3 3 4 4 4 4 2 2 5 5 4
```

```
    4978: SIGCONT
    4978: tick [illegible]
    4978: SIGTSTP
    4978: SIGCONT
    4978: tick [illegible]
    4978: SIGTSTP
    4978: SIGCONT
    4978: tick [illegible]
    4978: SIGTSTP
    4978: SIGCONT
    4978: tick [illegible]
    4978: SIGINT
     pid arrive   prior    cpu  status
    4983      6       3      5  RUNNING
    4983: START
    4983: tick 1
    4983: tick 2
    4983: tick 3
    4983: tick 4
    4983: tick 5
    4983: SIGINT
Dispatcher is free at DispatcherTime = 21.
The finish time is 20 s.
The job list is:1 1 1 2 3 3 3 3 4 4 2 2 2 2 2 5 5 5 5 5
[da72809@umacnx19 ~/aaaa]$
```

```
Input the silce time = 1
Input File = rr.txt
   pid arrive  prior    cpu  status
  4977      0      3      3  RUNNING
  4977: START
  4977: tick 1
  4977: tick 2
  4977: SIGTSTP
  4977: SIGCONT
  4977: tick 3
  4977: SIGINT
   pid arrive  prior    cpu  status
  4978      2      3      6  RUNNING
  4978: START
  4978: tick 1
  4978: SIGTSTP
   pid arrive  prior    cpu  status
  4979      4      3      4  RUNNING
  4979: START
  4979: tick 1
  4979: SIGTSTP
  4979: SIGCONT
  4979: tick 2
  4979: SIGTSTP
  4979: SIGCONT
  4979: tick 3
  4979: SIGTSTP
  4979: SIGCONT
  4979: tick 4
  4979: SIGINT
   pid arrive  prior    cpu  status
  4982      8      3      2  RUNNING
  4982: START
  4982: tick 1
  4982: SIGTSTP
  4982: SIGCONT
  4982: tick 2
  4982: SIGINT
  4978: SIGCONT
  4978: tick 2
  4978: SIGTSTP
  4978: SIGCONT
  4978: tick 3
  4978: SIGTSTP
  4978: SIGCONT
  4978: tick 4
  4978: SIGTSTP
  4978: SIGCONT
  4978: tick 5
```

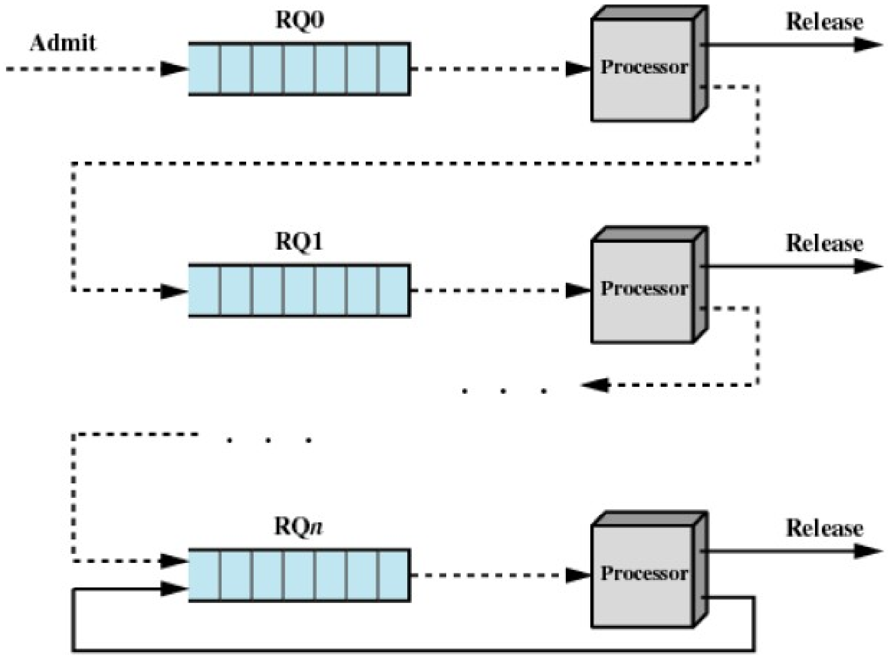
Feedback algorithm

(1). **Description of feedback algorithm:**

For processes, one very important goal of the processor is arrange the processes to be processed efficiently. However, for different intentions, many different processing methods, called scheduling algorithm, are applied. Feedback algorithm is one typical algorithm.

As mentioned, round robin has already solved problem long-running jobs inside bring in. here, feedback algorithm provides different priority queue to penalize long-running jobs focusing on time spent in execution so far. It is done on preemptive basis, and also, a dynamic priority mechanism is used.

The algorithm of feedback algorithm are as follows:



When a process first enters the system, it is placed in RQ0. After the first preemption, when it returns to the Ready state, it is placed in RQ1. Each subsequent time that it is preempted, it is demoted to the next lower-priority queue. Once in the lowest-priority queue, a process cannot go lower, but is returned to this queue repeatedly until it completes execution.

(2). **General flow of program and descriptions on used programming algorithms.**

1. Initialize dispatcher queues (input queue and feedback queues);
2. Fill input queue from dispatch list file;
3. Start dispatcher timer (dispatcher timer = 0);
4. While there's anything in any of the queues or there is a currently running process:
   i. Unload pending processes from the input queue:
      While (head-of-input-queue.arrival-time <= dispatcher timer)
      dequeue process from input queue and enqueue on highest priority feedback queue (assigning it the appropriate priority);
   ii. If a process is currently running:

a. Decrement process remainingcputime;
b. If times up:
   A. Send SIGINT to the process to terminate it;
   B. Free up process structure memory;
c. else if other processes are waiting in any of the feedback queues:
   A. Send SIGTSTP to suspend it;
   B. Reduce the priority of the process (if possible) and enqueue it on the appropriate feedback queue

iii. If no process currently running && feedback queues are not all empty:
   a. Dequeue a process from the highest priority feedback queue that is not empty
   b. If already started but suspended, restart it (send SIGCONT to it) else start it (fork & exec)
   c. Set it as currently running process;

iv. sleep for one second;

v. Increment dispatcher timer;

vi. Go back to 4.

5. Exit.

(3). **Some important sections explanations:**

(i)

```
if ( --(activePcb->remainingcputime) <= 0 ) {
              terminatePcb(activePcb);
              free(activePcb);
              activePcb = NULL;
              quantum_time=0;
          }
          else if (( pcbQ1 || pcbQ2 || pcbQ3 )&& quantum_time>=time )
{
              suspendPcb(activePcb);
              if ( activePcb->priority < 3 ) activePcb->priority++;
              if ( activePcb->priority == 2 )
                  pcbQ2 = enqPcb(pcbQ2, activePcb);
              else
                  pcbQ3 = enqPcb(pcbQ3, activePcb);
              activePcb = NULL;
              quantum_time=0;
          }
```

As the time goes by, when the dispatchertime increases one, there are two situations: first one is that current process is finished; second one is that the current process run out of time in one quantum but not finished; third one is that the current process does not run out of time and not finished. The code above is the code of the first and the second one.

From the segment of code, if we meet first situation, we need to terminate the process. If we meet the second situation, we need suspend current process and insert it into related queue.

(ii).

```
if ( ( pcbQ1 || pcbQ2 || pcbQ3 ) && !activePcb ) {
            //Dequeue process and start it
            if ( pcbQ1 )
               activePcb = deqPcb(&pcbQ1);
            else if ( pcbQ2 )
                activePcb = deqPcb(&pcbQ2);
            else
                activePcb = deqPcb(&pcbQ3);
            startPcb(activePcb);
            for ( i=0; i < 10 && activePcb->pid != joblist_id[i]; i++ )
{}
        if ( i == 10 )
                joblist_id[j++] = activePcb->pid;
        }
```

If any of feedback queues is not empty and no process is currently running, the code above describes the behavior. from the piece of code, it is obvious that when no current process is running, we will take the new active process from the highest feedback queue. At last, keep record of the new active process's pid.

(iii)

```
if ( !activePcb ) {
            printf( "Dispatcher is free at DispatcherTime = %d.\n" ,
dispatchertime ) ;
            job_per_s[dispatchertime] = -10;
        } else {
            for (i = 0; i < 10; i++)
                if (activePcb->pid == joblist_id[i])
                    job_per_s[dispatchertime] = i+1;
        }
```

If all feedback queues are empty and no current process is running, then it indicates all processes are dispatched or finished. The code mainly works for keeping record job.

(4). **Techniques in handling exceptions:**

(i). file-open exception: avoid errors because of system.

```
scanf( "%s" , filename ) ;
fp = fopen( filename , "r" );
if (fp == NULL) {
    perror(" Input File ");
    exit(1);
}
```

(ii). if at the specific instant, no process can be dispatched to run, we print “Idle” instead.

```
if (job_per_s[i] == -10)
          printf("Idle ");
```

(5). **Parameter modification:**

First, we can modify quantum parameter, which can determine how long a process can run at one time. We can observe how good if we choose one value as the parameter quantum, at the same time, we can also simulate other algorithm by changing parameter, for example, if we choose sufficiently big value as parameter quantum, feedback algorithm becomes FCFS algorithm:

The feedback algorithm with parameter quantum 20

(because no process cputime exceeds 20)

```
[da72903@umacnx25 ~/feedback]$ ./feedback4
 Input File = input.txt
 Time quantum: 20
    pid arrive  prior    cpu  status
   1753      0      1      3  RUNNING
   1753; START
   1753; tick 1
   1753; tick 2
   1753; tick 3
   1753; SIGINT
    pid arrive  prior    cpu  status
   1754      2      1      6  RUNNING
   1754; START
   1754; tick 1
   1754; tick 2
   1754; tick 3
   1754; tick 4
   1754; tick 5
   1754; tick 6
   1754; SIGINT
    pid arrive  prior    cpu  status
   1756      4      1      4  RUNNING
   1756; START
   1756; tick 1
   1756; tick 2
   1756; tick 3
   1756; tick 4
   1756; SIGINT
    pid arrive  prior    cpu  status
   1757      6      1      5  RUNNING
   1757; START
   1757; tick 1
   1757; tick 2
   1757; tick 3
   1757; tick 4
   1757; tick 5
   1757; SIGINT
    pid arrive  prior    cpu  status
   1758      8      2      2  RUNNING
   1758; START
   1758; tick 1
   1758; tick 2
   1758; SIGINT
Dispatcher is free at DispatcherTime = 21.
The finish time is 20 s.
The job list is:1 1 1 2 2 2 2 2 2 3 3 3 3 4 4 4 4 4 5 5
[da72903@umacnx25 ~/feedback]$
```

The FCFS algorithm：

```
[da72903@umacnx25 ~/feedback]$ ./fcfs
 Input File = input.txt
    pid arrive  prior    cpu  status
   1823      0      1      3  RUNNING
   1823; START
   1823; tick 1
   1823; tick 2
   1823; tick 3
   1823; SIGINT
    pid arrive  prior    cpu  status
   1827      2      1      6  RUNNING
   1827; START
   1827; tick 1
   1827; tick 2
   1827; tick 3
   1827; tick 4
   1827; tick 5
   1827; tick 6
   1827; SIGINT
    pid arrive  prior    cpu  status
   1828      4      1      4  RUNNING
   1828; START
   1828; tick 1
   1828; tick 2
   1828; tick 3
   1828; tick 4
   1828; SIGINT
    pid arrive  prior    cpu  status
   1829      6      1      5  RUNNING
   1829; START
   1829; tick 1
   1829; tick 2
   1829; tick 3
   1829; tick 4
   1829; tick 5
   1829; SIGINT
    pid arrive  prior    cpu  status
   1830      8      2      2  RUNNING
   1830; START
   1830; tick 1
   1830; tick 2
   1830; SIGINT
Dispatcher is free at DispatcherTime = 21.
The finish time is 20 s.
The job list is:1 1 1 2 2 2 2 2 2 3 3 3 3 4 4 4 4 4 5 5
```

From the two snapshots, we recognize that as long as time quantum is sufficient big, feedback algorithm will become FCFS algorithm.

If we choose a small value as parameter quantum, maybe feedback is different from FCFS:

The feedback algorithm with parameter quantum 1:

```
[da72903@umacnx25 ~/feedback]$ ./feedback4
 Input File = input.txt
 Time quantum: 1
    pid arrive  prior    cpu  status
   1775      0      1      3  RUNNING
   1775; START
   1775; tick 1
   1775; tick 2
   1775; SIGTSTP
    pid arrive  prior    cpu  status
   1776      2      1      6  RUNNING
   1776; START
   1776; tick 1
   1776; SIGTSTP
   1775; SIGCONT
   1775; tick 3
   1775; SIGINT
    pid arrive  prior    cpu  status
   1777      4      1      4  RUNNING
   1777; START
   1777; tick 1
   1777; SIGTSTP
   1776; SIGCONT
   1776; tick 2
   1776; SIGTSTP
    pid arrive  prior    cpu  status
   1778      6      1      5  RUNNING
   1778; START
   1778; tick 1
   1778; SIGTSTP
   1777; SIGCONT
   1777; tick 2
   1777; SIGTSTP
```

```
   pid arrive  prior    cpu  status
  1779      8      2      2  RUNNING
  1779; START
  1779; tick 1
  1779; SIGTSTP
  1778; SIGCONT
  1778; tick 2
  1778; SIGTSTP
  1776; SIGCONT
  1776; tick 3
  1776; SIGTSTP
  1777; SIGCONT
  1777; tick 3
  1777; SIGTSTP
  1779; SIGCONT
  1779; tick 2
  1779; SIGINT
  1778; SIGCONT
  1778; tick 3
  1778; SIGTSTP
  1776; SIGCONT
  1776; tick 4
  1776; SIGTSTP
  1777; SIGCONT
  1777; tick 4
  1777; SIGINT
  1778; SIGCONT
  1778; tick 4
  1778; SIGTSTP
  1776; SIGCONT
  1776; tick 5
  1776; SIGTSTP
  1778; SIGCONT
  1778; tick 5
  1778; SIGINT
  1776; SIGCONT
  1776; tick 6
  1776; SIGINT
Dispatcher is free at DispatcherTime = 21.
The finish time is 20 s.
The job list is:1 1 2 1 3 2 4 3 5 4 2 3 5 4 2 3 4 2 4 2
```

The FCFS:

```
[da72903@umacnx25 ~/feedback]$ ./fcfs
 Input File = input.txt
    pid arrive  prior    cpu  status
   1823      0      1      3  RUNNING
   1823; START
   1823; tick 1
   1823; tick 2
   1823; tick 3
   1823; SIGINT
    pid arrive  prior    cpu  status
   1827      2      1      6  RUNNING
   1827; START
   1827; tick 1
   1827; tick 2
   1827; tick 3
   1827; tick 4
   1827; tick 5
   1827; tick 6
   1827; SIGINT
    pid arrive  prior    cpu  status
   1828      4      1      4  RUNNING
   1828; START
   1828; tick 1
   1828; tick 2
   1828; tick 3
   1828; tick 4
   1828; SIGINT
    pid arrive  prior    cpu  status
   1829      6      1      5  RUNNING
   1829; START
   1829; tick 1
   1829; tick 2
   1829; tick 3
   1829; tick 4
   1829; tick 5
   1829; SIGINT
    pid arrive  prior    cpu  status
   1830      8      2      2  RUNNING
   1830; START
   1830; tick 1
   1830; tick 2
   1830; SIGINT
Dispatcher is free at DispatcherTime = 21.
The finish time is 20 s.
The job list is:1 1 1 2 2 2 2 2_2 3 3 3 3 4 4 4 4 4 5 5
```

That is different from the two pictures.

(6). Achievement in program requirements and unfinished tasks.

The output is correct and the quantum can be defined at the beginning of the program. Different processes have different colors in its background.

(7). Further implementations and what you have learnt.

From the feedback algorithm, firstly, we know how the operating system operates many processes using feedback. Feedback algorithm indeed overcomes many difficulties just like long-running process occupies much time resulting in starvation of short process. However, we see that feedback algorithm also has some problems such that continuous new processes come into the highest priority queue ,resulting in the starvation of processes in lower priority queue. We need to apply better algorithm to realizing the difficulties.

(8). **Reference of how dispatcher working:**

Firstly, all processes are put into input queue. Secondly, if arrivaltime<=dispatchertime, we can put head process of input queue into highest priority feedback queue. When system has no active process, the dispatcher picks up the head process from one highest feedback queue that is nonempty. Until all processes in all queues terminates, dispatcher stops working.

(9). **Snapshots:**

(i).**Files included for successful program execution:**

sigtrap.c , feedback2.c, input.txt will be included in the directory.

(ii).**Command typed in compiling your program?**

```
[da72903@umacnx25 ~/feedback]$ gcc -o process sigtrap.c
[da72903@umacnx25 ~/feedback]$ gcc -o feedback4 feedback4.c
```

(iii).**Command typed in executing your program?**

```
[da72903@umacnx25 ~/feedback]$ ./feedback4
```

(iv). **Snapshots to illustrate how you achieve the project requirements.**

```
[da72903@umacnx25 ~/feedback]$ ./feedback4
 Input File = input.txt
 Time quantum: 1
    pid arrive  prior    cpu  status
   1403      0      1      3  RUNNING
   1403; START
   1403; tick 1
   1403; tick 2
   1403; SIGTSTP
    pid arrive  prior    cpu  status
   1404      2      1      6  RUNNING
   1404; START
   1404; tick 1
   1404; SIGTSTP
   1403; SIGCONT
   1403; tick 3
   1403; SIGINT
    pid arrive  prior    cpu  status
   1405      4      1      4  RUNNING
   1405; START
   1405; tick 1
   1405; SIGTSTP
   1404; SIGCONT
   1404; tick 2
   1404; SIGTSTP
    pid arrive  prior    cpu  status
   1406      6      1      5  RUNNING
   1406; START
   1406; tick 1
   1406; SIGTSTP
   1405; SIGCONT
   1405; tick 2
   1405; SIGTSTP
    pid arrive  prior    cpu  status
   1407      8      2      2  RUNNING
   1407; START
   1407; tick 1
   1407; SIGTSTP
   1406; SIGCONT
   1406; tick 2
   1406; SIGTSTP
   1404; SIGCONT
   1404; tick 3
   1404; SIGTSTP
```

```
    1405; SIGCONT
    1405; tick 3
    1405; SIGTSTP
    1407; SIGCONT
    1407; tick 2
    1407; SIGINT
    1406; SIGCONT
    1406; tick 3
    1406; SIGTSTP
    1404; SIGCONT
    1404; tick 4
    1404; SIGTSTP
    1405; SIGCONT
    1405; tick 4
    1405; SIGINT
    1406; SIGCONT
    1406; tick 4
    1406; SIGTSTP
    1404; SIGCONT
    1404; tick 5
    1404; SIGTSTP
    1406; SIGCONT
    1406; tick 5
    1406; SIGINT
    1404; SIGCONT
    1404; tick 6
    1404; SIGINT
Dispatcher is free at DispatcherTime = 21.
The finish time is 20 s.
The job list is:1 1 2 1 3 2 4 3 5 4 2 3 5 4 2 3 4 2 4 2
```

SPN

Shortest Process Next

(1). Description of SPN:

First we read the case study which is FCFS, we think FCFS is similar to SPN, because they both are non-preemptive. And the difference between them is the way processor get the process from the queue, so we find the the corresponding part of the FCFS program, inset a subfunction

```
PcbPtr DesideShortest(PcbPtr *PcbQ, int dispatchertime)
```

Which we write to deside the appropriate process to be executed.

About RoundRobin, it is a little different to FCFS, because RR is preemtive. First we define a time slot to limit the time processor execution time. And use that queue to decide the order of all processes, and use deqPcb() and enqPcb() recuisively.

(2). General flow of program and descriptions on used programming algorithms.

1. Program start, read the processes information from the txt file and sive it in main queue PcbQ
2. Enter the main loop unless no process in processor and in queue, loop stop.

```
while ( pcbQ || activePcb ){}
```

3. In the main loop, classify 3 situation to deal with.
4. First situation: when the processor is doing a process if (activePcb){}. Then we should check wether the currently process should be terminated. If it should be terminated, then do what should we do.

```
if ( --(activePcb->remainingcputime) <= 0 ) {
    //Terminate it and free up memory
    terminatePcb(activePcb);
    free(activePcb);
    activePcb = NULL;
//  tempPcb = DesideShortest(pcbQ);
}
```

5. Second situation, when main queue is not empty and processor is free and there exist such a process which is ready to enter the processor. Then we should deque to get the apropriate processor and start it and sive the data into the result.

```
if ( pcbQ &&  !activePcb && DesideShortest(&pcbQ,dispatchertime)){
            activePcb = deqPcb(&pcbQ);
            startPcb(activePcb);
            joblist_id[j++] = activePcb->pid;
        }
```

6. Third situation, when the processor is free, that means the main queue is not empty and the processor is free and no process is already. Then print the related information and sive it in the result. If the processor is not free, then record the information, and sive it into result.

```
    if ( !activePcb ) {
            printf( "Dispatcher is free at DispatcherTime = %d.\n" ,
dispatchertime ) ;
            job_per_s[dispatchertime] = -1;
        } else {
            for (i = 0; i < 10; i++)
```

```
            if (activePcb->pid == joblist_id[i])
                job_per_s[dispatchertime] = i+1;
        }
    }
```

7. Output the result using int job_per_s[100], joblist_id[10];

(3). some explanations about important section of programming codes in achieve certain tasks.

(i).

```
for(; p != NULL; p = p->next)
    if(p->arrivaltime <= dispatchertime)
        if(p1->remainingcputime > p->remainingcputime && p1 != NULL)
            p1 = p;
```

This part decide wither to produce such a process fit properties of SPN. The return value is p1, which is initiallize to a ready process if exist either still is NULL. Then this loop is try to find such a process which is arrived and remaining time is less. if found assign the value to p1.

(ii).

```
if(p1 != NULL)
for(p = *PcbQ; p != p1;)
{
    p2 = p->next;
    p= deqPcb(PcbQ);
    *PcbQ = enqPcb(*PcbQ , p);
    p = p2;
}
```

This part decide what is that process and put it on the head of the queue. Because in the main loop, it use deque function to get the process from the queue, then we must put the process which should be exeuted next into the header of the queue. This loop is for this, it use deque function to pop the previous processes until the p1 pointed process is checked. Then let the header pointer PcbQ point to the header.

(iii).

```
if ( pcbQ &&  !activePcb && DesideShortest(&pcbQ, dispatchertime)) {
    //Dequeue process and start it
    activePcb = deqPcb(&pcbQ);
    startPcb(activePcb);
    joblist_id[j++] = activePcb->pid;
}
```

This is call function. It is inseted into the if structure. After check the main queue and processor, issure they are ready for a new process to execute. Then execute this function to decide which processs is apropriate, if none is ok then return NULL.

(4). Techniques in handling exceptions.

(i). file-open exception: avoid errors because of system.

```
scanf( "%s" , filename ) ;
fp = fopen( filename , "r" );
if (fp == NULL) {
    perror(" Input File ");
    exit(1);
```

```
}
```

(ii). if at the specific instant, no process can be dispatched to run, we print “Idle”instead.

```
if (job_per_s[i] == -10)
          printf("Idle ");
```

(5). Parameter modification:

The parameter of SPN is same to FCFS, which arival time, execution time. About RR, additional parameter is time slot, which define the limitation of CPU processing time.

RoundRobin is preemptive, it impartial dispatch equally time to every process every cycle. So this algorithm prefer to execute large process compare to SPN. About SPN it just preemptive short process, and it is non-preemptive.

(6). Achievement in program requirements and unfinished tasks.

About SPN, we design a sub function to achieve the program requirement. About RR, we modify the main loop of FCFS algorithm. No unfinished tasks.

(7). Further implementations and what you have learnt.

I learned that how to implement simple schedulings, and know what the data structure is in program to represent a process, and understand every scheduling polisy deeply. And I also learned some commends in operating system layer which is not familiar.

(8). Reference of dispatcher working:

First all process enter the main queue. Then the dispatchertime added until a process’ arrival time < dispatchertime, processor execute it, if more than one porcesses arrives the processor pick the process whose remaining time is smallest. Then loop by loop until the last process is finished then the program print the result.

(9). Snapshots:

(i). Files included for successful program execution:

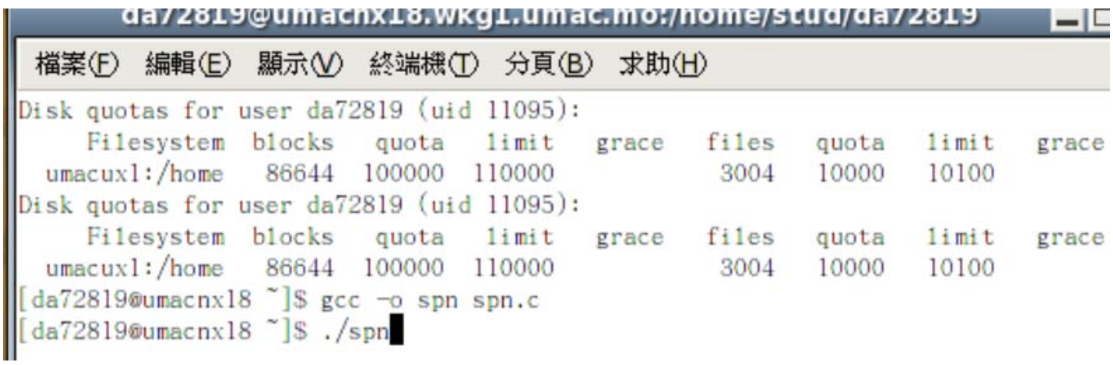
sigtrap.c, pcb.h, pcb.c, spn.c files are included in the project.

(ii). Command typed in compiling your program:

gcc –o process sigtrap.c

gcc –o spn spn.c

(iii) Command typed in executing your program:



(iv) Snapshots to illustrate how you achieve the project requirements.

```
[da72819@umacnx18 ~]$ ./spn
 Input File = input.txt
    pid arrive  prior    cpu  status
   9709      0      1      3  RUNNING
   9709: START
   9709: tick 1
   9709: tick 2
   9709: tick 3
   9709: SIGINT
    pid arrive  prior    cpu  status
   9710      2      1      6  RUNNING
   9710: START
   9710: tick 1
   9710: tick 2
   9710: tick 3
   9710: tick 4
   9710: tick 5
   9710: tick 6
   9710: SIGINT
    pid arrive  prior    cpu  status
   9711      8      2      2  RUNNING
   9711: START
   9711: tick 1
   9711: tick 2
   9711: SIGINT
    pid arrive  prior    cpu  status
   9712      4      1      4  RUNNING
   9712: START
   9712: tick 1
   9712: tick 2
   9712: tick 3
   9712: tick 4
   9712: SIGINT
```

```
    pid arrive  prior    cpu  status
   9714      6      1      5  RUNNING
   9714: START
   9714: tick 1
   9714: tick 2
   9714: tick 3
   9714: tick 4
   9714: tick 5
   9714: SIGINT
Dispatcher is free at DispatcherTime = 21.
The finish time is 20 s.
The job list is:1 1 1 2 2 2 2 2 2 3 3 4 4 4 4 5 5 5 5 5
[da72819@umacnx18 ~]$
```

Appendix

The code of round robin based on FCFS:

```
#include <stdio.h>

#include <stdlib.h>

#include "pcb.c"

#include <assert.h>


int main() {

      int dispatchertime = 0 ;

      int arrivalt , prior , cput;

      int job_per_s[100], joblist_id[10];

      int i = 0, j = 0, s = 0, q = 0;

      char FN[20] ;

      FILE *fp;

      //Initialize dispatcher queues

      PcbPtr pcbQ = NULL;

      PcbPtr rQ = NULL;  // RR queue

      PcbPtr tempPcb;

      PcbPtr activePcb = NULL;

      for (i = 0; i < 10; i++)

            joblist_id[i] = 999;

      for (i = 0; i < 100; i++)

            job_per_s[i] = -1;

      printf( " Input the silce time = " );

      scanf( "%d", &q );

      printf( " Input File = " );

      scanf( "%s" , FN ) ;

      fp = fopen( FN , "r" );

      if (fp == NULL) {

            perror(" Input File ");

            exit(1);

      }

      //Fill dispatcher queue from dispatch list file

      while ( fscanf ( fp , "%d , %d , %d" , &arrivalt, &prior, &cput) != EOF ) {

      tempPcb = createnullPcb();

      tempPcb->arrivaltime = arrivalt;

      tempPcb->remainingcputime = cput;

      tempPcb->priority = prior;

      pcbQ = enqPcb(pcbQ , tempPcb);

      }

      fclose( fp );

      //While there's anything in any of the queues or there is a currently running process:

      while ( rQ || pcbQ || activePcb ) {
```

```
        //Unload any pending processes from the input queue:
                while( pcbQ && pcbQ->arrivaltime <= dispatchertime  ){

                        rQ = enqPcb(rQ, deqPcb(&pcbQ));
                }

if( activePcb && activePcb->remainingcputime > 1 && s < q ){
        startPcb(activePcb);
        activePcb->remainingcputime--;
        s++;
        sleep(1);
        dispatchertime++;

                if ( !activePcb ) {
                        printf( "Dispatcher is free at DispatcherTime = %d.\n" , dispatchertime ) ;
                        job_per_s[dispatchertime] = -1;
                } else {
                        for (i = 0; i < 10; i++)
                                if (activePcb->pid == joblist_id[i])
                                        job_per_s[dispatchertime] = i+1;
                }

        continue;
}
        //If a process is currently running

        if ( activePcb ) {
                        //Decrement process remainingcputime and check it is time up or not

                        if ( --(activePcb->remainingcputime) <= 0 ) {
                                //Terminate it and free up memory
                                terminatePcb(activePcb);
                                free(activePcb);
                                activePcb = NULL;
                        }

                        //else if other processes are waiting in RR queue:
                        else if ( rQ && activePcb->remainingcputime > 0 ){
                                activePcb = suspendPcb( activePcb );
                                rQ = enqPcb( rQ, activePcb );
                                activePcb = NULL;
                        }
```

```
	}
//If no process currently running && RR queue is not empty:
if( !activePcb && rQ ){

	activePcb = deqPcb(&rQ);
	//if suspend
	if( activePcb->status == PCB_SUSPENDED ){
		startPcb(activePcb);
		s = 1;
	}
	else{
		startPcb(activePcb);
		joblist_id[j++] = activePcb->pid;
		s = 1;
	}

}




	//sleep for one second and increment dispatcher timer
	sleep(1);
	dispatchertime++;

	if ( !activePcb ) {
		printf( "Dispatcher is free at DispatcherTime = %d.\n" , dispatchertime ) ;
		job_per_s[dispatchertime] = -1;
	} else {
		for (i = 0; i < 10; i++)
			if (activePcb->pid == joblist_id[i])
				job_per_s[dispatchertime] = i+1;
	}



}



//print out the job list and finish time
printf("The finish time is %d s.\n", dispatchertime-1);
printf("The job list is:");
for (i = 1; i < dispatchertime; i++) {
	if (job_per_s[i] == -1)
		printf("Idle ");
	else
		printf("%d ", job_per_s[i]);
```

```
    }
    printf("\n");
    return 0 ;
}
```

===

The code of round robin based on SPN:

```
#include <stdio.h>
#include <stdlib.h>
#include "pcb.c"
#include <assert.h>

PcbPtr DesideShortest(PcbPtr *PcbQ, int dispatchertime)
{
    PcbPtr p = *PcbQ, p1 = *PcbQ, p2;
    PcbPtr temp;
    for(; p1 != NULL; p1 = p1->next)
        if(p1->arrivaltime <= dispatchertime) break;

    for(; p != NULL; p = p->next)
        if(p->arrivaltime <= dispatchertime)
            if(p1->remainingcputime > p->remainingcputime && p1 != NULL)
                p1 = p;

    if(p1 != NULL)
    for(p = *PcbQ; p != p1;)
    {
        p2 = p->next;
        p= deqPcb(PcbQ);
    *PcbQ = enqPcb(*PcbQ , p);
        p = p2;
    }

/*
    if(p1 != NULL && (*PcbQ)->next != NULL)
    {
        for(p = *PcbQ; p->next != p1; p = p->next);
        temp = (*PcbQ)->next;
        (*PcbQ)->next = p1->next;
        p->next = *PcbQ;
        p1->next = temp;
        *PcbQ = p1;
    }
*/
    return p1;
```

```
}



int main() {
      int dispatchertime = 0 ;
      int arrivalt , prior , cput;
      int job_per_s[100], joblist_id[10];
      int i = 0, j = 0, s = 0, q = 0;
      char FN[20] ;
      FILE *fp;
      //Initialize dispatcher queues
      PcbPtr pcbQ = NULL;
      PcbPtr rQ = NULL;   // RR queue
      PcbPtr tempPcb;
      PcbPtr activePcb = NULL;
      for (i = 0; i < 10; i++)
            joblist_id[i] = 999;
      for (i = 0; i < 100; i++)
            job_per_s[i] = -1;
      printf( " Input the silce time = " );
      scanf( "%d", &q );
      printf( " Input File = " );
      scanf( "%s" , FN ) ;
      fp = fopen( FN , "r" );
      if (fp == NULL) {
            perror(" Input File ");
            exit(1);
      }
      //Fill dispatcher queue from dispatch list file
      while ( fscanf ( fp , "%d , %d , %d" , &arrivalt, &prior, &cput) != EOF ) {
      tempPcb = createnullPcb();
      tempPcb->arrivaltime = arrivalt;
      tempPcb->remainingcputime = cput;
      tempPcb->priority = prior;
      pcbQ = enqPcb(pcbQ , tempPcb);
      }
      fclose( fp );
      //While there's anything in any of the queues or there is a currently running process:
      while ( rQ || pcbQ || activePcb ) {
      //Unload any pending processes from the input queue:
            while( pcbQ && pcbQ->arrivaltime <= dispatchertime  ){

                  rQ = enqPcb(rQ, deqPcb(&pcbQ));
            }
```

```
if( activePcb && activePcb->remainingcputime > 1 && s < q ){
      startPcb(activePcb);
      activePcb->remainingcputime--;
      s++;
      sleep(1);
      dispatchertime++;

            if ( !activePcb ) {
                  printf( "Dispatcher is free at DispatcherTime = %d.\n" , dispatchertime ) ;
                  job_per_s[dispatchertime] = -1;
            } else {
                  for (i = 0; i < 10; i++)
                        if (activePcb->pid == joblist_id[i])
                              job_per_s[dispatchertime] = i+1;
            }

      continue;
}
      //If a process is currently running

      if ( activePcb ) {
                  //Decrement process remainingcputime and check it is time up or not

                  if ( --(activePcb->remainingcputime) <= 0 ) {
                        //Terminate it and free up memory
                        terminatePcb(activePcb);
                        free(activePcb);
                        activePcb = NULL;
                  }

                  //else if other processes are waiting in RR queue:
                  else if ( rQ && activePcb->remainingcputime > 0 ){
                        activePcb = suspendPcb( activePcb );
                        rQ = enqPcb( rQ, activePcb );
                        activePcb = NULL;
                  }




            }
      //If no process currently running && RR queue is not empty:
      if( !activePcb && rQ){
            //activePcb = sort(rQ);
            activePcb  = DesideShortest(&rQ,dispatchertime);
            deqPcb( &rQ );
```

```
		//if suspend
		if( activePcb->status == PCB_SUSPENDED ){
			startPcb(activePcb);
			//s = 1;
		}
		else{
			startPcb(activePcb);
			joblist_id[j++] = activePcb->pid;
			//s = 1;
		}

	}
		//sleep for one second and increment dispatcher timer
		sleep(1);
		dispatchertime++;

		if ( !activePcb ) {
			printf( "Dispatcher is free at DispatcherTime = %d.\n" , dispatchertime ) ;
			job_per_s[dispatchertime] = -1;
		} else {
			for (i = 0; i < 10; i++)
				if (activePcb->pid == joblist_id[i])
					job_per_s[dispatchertime] = i+1;
		}



	}



	//print out the job list and finish time
	printf("The finish time is %d s.\n", dispatchertime-1);
	printf("The job list is:");
	for (i = 1; i < dispatchertime; i++) {
		if (job_per_s[i] == -1)
			printf("Idle ");
		else
			printf("%d ", job_per_s[i]);
	}
	printf("\n");
	return 0 ;
}
```

The code of feedback:

```
#include <stdio.h>
#include <stdlib.h>
#include "pcb.c"

int main() {
	int dispatchertime = 0 ;
	int arrivalt , prior , cput;
	int job_per_s[100], joblist_id[10];
	int i = 0, j = 0;
	char filename[20] ;
	FILE *fp;


	int time,quantum_time;


	//Initialize dispatcher queues
	PcbPtr pcbin = NULL;
	PcbPtr pcbQ1 = NULL;
	PcbPtr pcbQ2 = NULL;
	PcbPtr pcbQ3 = NULL;
	PcbPtr tempPcb;
	PcbPtr activePcb = NULL;
	for (i = 0; i < 10; i++)
		joblist_id[i] = 999;
	for (i = 0; i < 100; i++)
		job_per_s[i] = -1;
	printf( " Input File = " );
	scanf( "%s" , filename ) ;
	fp = fopen( filename , "r" );
	if (fp == NULL) {
		perror(" Input File ");
		exit(1);
	}

	//ask user to input quantum
	printf( " Time quantum: " );
	scanf( "%d" , &time ) ;
	quantum_time=0;



	//Fill input queue from dispatch list file
```

```
        while ( fscanf ( fp , "%d , %d , %d" , &arrivalt, &prior, &cput) != EOF ) {
        tempPcb = createnullPcb();
        tempPcb->arrivaltime = arrivalt;
        tempPcb->remainingcputime = cput;
        tempPcb->priority = prior;
        pcbin = enqPcb(pcbin , tempPcb);
        }
        fclose( fp );

        //While there's anything in the queue or there is a currently running process
        while ( pcbin || pcbQ1 || pcbQ2 || pcbQ3 || activePcb ) {
              while (pcbin && pcbin->arrivaltime <= dispatchertime)
                    pcbQ1 = enqPcb(pcbQ1, deqPcb (&pcbin));
              //If a process currently running
              if ( activePcb ) {

                    quantum_time++;



                    //Decrement process remainingcputime and check if time's up
                    //if time's up, terminate current process
                    if ( --(activePcb->remainingcputime) <= 0 ) {
                          terminatePcb(activePcb);
                          free(activePcb);
                          activePcb = NULL;
                          quantum_time=0;
                    }
                    else if (( pcbQ1 || pcbQ2 || pcbQ3 )&& quantum_time>=time ) {
                          suspendPcb(activePcb);
                          if ( activePcb->priority < 3 ) activePcb->priority++;
                          if ( activePcb->priority == 2 )
                                pcbQ2 = enqPcb(pcbQ2, activePcb);
                          else
                                pcbQ3 = enqPcb(pcbQ3, activePcb);
                          activePcb = NULL;
                          quantum_time=0;
                    }
              }
           // if there is no current process running &&
              // feedback queue nonempty
/*notice*/    if ( ( pcbQ1 || pcbQ2 || pcbQ3 ) && !activePcb ) {
                    //Dequeue process and start it
                    if ( pcbQ1 )
                      activePcb = deqPcb(&pcbQ1);
                    else if ( pcbQ2 )
```

```
                        activePcb = deqPcb(&pcbQ2);
                  else
                        activePcb = deqPcb(&pcbQ3);
                  startPcb(activePcb);
                  for ( i=0; i < 10 && activePcb->pid != joblist_id[i]; i++ ) {}
            if ( i == 10 )
                        joblist_id[j++] = activePcb->pid;
            }
            //sleep for one second and increment dispatcher timer
            sleep(1);
            dispatchertime++;
      // if there is still no active process after notice, mark the job_per_s[] as -1 to represent idle
            if ( !activePcb ) {
                  printf( "Dispatcher is free at DispatcherTime = %d.\n" , dispatchertime ) ;
                  job_per_s[dispatchertime] = -10;
            } else {
                  for (i = 0; i < 10; i++)
                        if (activePcb->pid == joblist_id[i])
                              job_per_s[dispatchertime] = i+1;
            }
      }
      //print out finish time and job list
      printf("The finish time is %d s.\n", dispatchertime-1);
      printf("The job list is:");
      for (i = 1; i < dispatchertime; i++) {
            if (job_per_s[i] == -10)
                  printf("Idle ");
            else
                  printf("%d ", job_per_s[i]);
      }
      printf("\n");
      return 0 ;
}
```

The code of SPN:

```
#include <stdio.h>
#include <stdlib.h>
#include "pcb.c"
PcbPtr DesideShortest(PcbPtr *PcbQ, int dispatchertime)
{
      PcbPtr p = *PcbQ, p1 = *PcbQ, p2;
      PcbPtr temp;
      for(; p1 != NULL; p1 = p1->next)
```

```
            if(p1->arrivaltime <= dispatchertime) break;

      for(; p != NULL; p = p->next)
            if(p->arrivaltime <= dispatchertime)
                  if(p1->remainingcputime > p->remainingcputime && p1 != NULL)
                        p1 = p;
      if(p1 != NULL)
      for(p = *PcbQ; p != p1;)
      {
            p2 = p->next;
            p= deqPcb(PcbQ);
      *PcbQ = enqPcb(*PcbQ , p);
            p = p2;
      }
      return p1;
}

int main() {
      int dispatchertime = 0 ;
      int arrivalt , prior , cput;
      int job_per_s[100], joblist_id[10];
      int i = 0, j = 0;
      char FN[20] ;
      FILE *fp;
      //Initialize dispatcher queues
      PcbPtr pcbQ = NULL;
      PcbPtr tempPcb;
      PcbPtr activePcb = NULL;
      for (i = 0; i < 10; i++)
            joblist_id[i] = 999;
      for (i = 0; i < 100; i++)
            job_per_s[i] = -1;
      printf( " Input File = " );
      scanf( "%s" , FN ) ;
      fp = fopen( FN , "r" );
      if (fp == NULL) {
            perror(" Input File ");
            exit(1);
      }
      //Fill dispatcher queue from dispatch list file
      while ( fscanf ( fp , "%d , %d , %d" , &arrivalt, &prior, &cput) != EOF ) {
      tempPcb = createnullPcb();
      tempPcb->arrivaltime = arrivalt;
      tempPcb->remainingcputime = cput;
      tempPcb->priority = prior;
```

```
pcbQ = enqPcb(pcbQ , tempPcb);
}
fclose( fp );
//While there's anything in the queue or there is a currently running process
while ( pcbQ || activePcb ) {
      //If a process is currently running
      if ( activePcb ) {
            //Decrement process remainingcputime and check it is time up or not
            if ( --(activePcb->remainingcputime) <= 0 ) {
                  //Terminate it and free up memory
                  terminatePcb(activePcb);
                  free(activePcb);
                  activePcb = NULL;
            //    tempPcb = DesideShortest(pcbQ);
            }
      }
      //If no process currently running && dispatcher queue is not empty &&
      //arrivaltime of process at head of queue is <= dispatcher timer
      if ( pcbQ &&  !activePcb && DesideShortest(&pcbQ,dispatchertime)) {
            //Dequeue process and start it
            activePcb = deqPcb(&pcbQ);
            startPcb(activePcb);
            joblist_id[j++] = activePcb->pid;
      }
      //sleep for one second and increment dispatcher timer
      sleep(1);
      dispatchertime++;

      if ( !activePcb ) {
            printf( "Dispatcher is free at DispatcherTime = %d.\n" , dispatchertime ) ;
            job_per_s[dispatchertime] = -1;
      } else {
            for (i = 0; i < 10; i++)
                  if (activePcb->pid == joblist_id[i])
                        job_per_s[dispatchertime] = i+1;
      }
}
//print out the job list and finish time
printf("The finish time is %d s.\n", dispatchertime-1);
printf("The job list is:");
for (i = 1; i < dispatchertime; i++) {
      if (job_per_s[i] == -1)
            printf("Idle ");
      else
            printf("%d ", job_per_s[i]);
```

```
    }
    printf("\n");
    return 0 ;
}
```

Part II

(1). Description of scheduling:

For-Level Priority Dispatcher:

The dispatcher operates at four priority levels: Real-time processes must be run immediately on a FCFS basis. Normal user processes are run on a three level feedback dispatcher. The basic timing quantum of the dispatcher is 1 second. Every process should be allocated enough memory before enter ready queue. And there are five memory allocation policies which will be described in following parts.

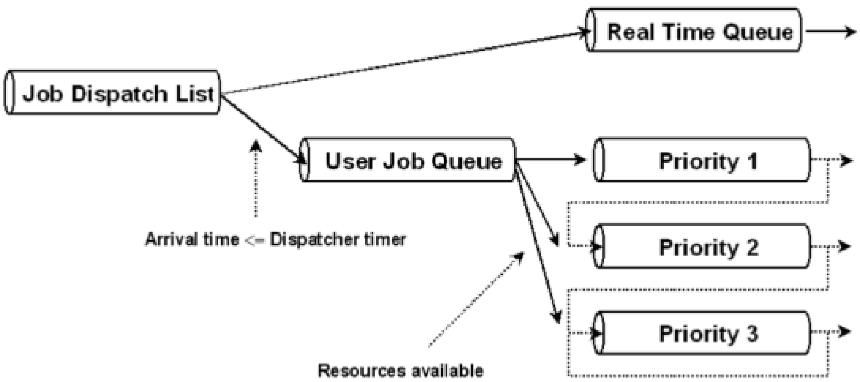


Figure 3. Dispatcher Logic Flow

Resource Constraints:

Low-priority processes can use any or all of these resources, but the Host dispatcher is notified of which resources the process will use when the process is submitted. The dispatcher ensures that each requested resource is solely available to that process throughout its lifetime in the "ready-to-run" dispatch queues: from the initial transfer from the job queue to the Priority 1-3 queues through to process completion, including intervening idle time quanta.

Memory Allocation:

For each process, a contiguous block of memory must be assigned. The memory block must remain assigned to the process for the lifetime of the process. In our host there five memory allocation policies witch are Best Fit, First Fit Next Fit, Worse.

Processes:

Processes on HOST are simulated by the dispatcher creating a new process for each dispatched process. This process is a generic process that can be used for any priority process.

Dispatch List:

The Dispatch List is the list of processes to be processed by the dispatcher. The list is contained in a text file that is specified on the command line. Each line of the list describes one process with the following data as a " comma-space" delimited list: <arrival time>. <priority>, <processor time>. <Mbytes>, <#printers>.<#scanners>, <#modems>, <#CDs>.

(2) General flow of program and descriptions on used programming algorithms:

This program is used to simulate a dispatcher, include memory arrangement and recourse management. First all processes are put in the Job dispatcher queue, then the arrived processes was put into User Job Queue for waiting for allocating enough memory and recourses. Then if the process is real time, put it into Real Time Queue, or put it into Priority queue, until the processor execute it. When all processes are finished then quit the main loop and print the result.

(3) Some explanations about important section of programming codes in achieve certain tasks:

Initialize the memory block

```
MabPtr startMab ()
{
   MabPtr m;
   if ( m = (MabPtr) malloc( sizeof(Mab) ) ) {
      m->offset = 0;
      m->size = 960;
      m->allocated = 0;
      m->next = NULL;
      m->prev = NULL;
   }
   return m;
}
```

check if memory available

```
MabPtr memChk(MabPtr m, int size) {  // check if memory available
   //printf("%d %d 1", m->size, m->allocated);
   MabPtr n = m;
   if (m == NULL) return NULL;
   while ( n != NULL && ( n->size < size || n->allocated != 0 )) {
      n = n->next; //printf("%d 2", n->size);
   }
   if ( n == NULL && m->offset != 0 ) {
      n = m;
      while ( n->offset != 0)
         n = n->prev;
      while ( n != m && ( n->size < size || n->allocated != 0 )) {
         n = n->next; //printf("%d 2", n->size);
      }
      if ( n == m )
         n = NULL;
   }
   //printf("%d 3", m->size);
   return n;
}
```

allocate memory block

```
MabPtr memAlloc(MabPtr m, int size) { // allocate memory block
   //m = memChk(m, size);
```

```
    //printf("3");
    if (m == NULL) return NULL;
    if (m->size != size)//printf("2");
        m = memSplit(m, size);
    m->allocated = 1;
    return m;
}
```

free memory block

```
MabPtr memFree(MabPtr m) {        // free memory block
    if (m == NULL) return NULL;
    m->allocated = 0;
    m = memMerge(m);
    return m;
}
```

merge two memory blocks

```
MabPtr memMerge(MabPtr m)
split a memory block
MabPtr memSplit(MabPtr m, int size)
print the memory arena
void printMab(MabPtr m) {
    printf("offset      size      allocated   \n");
    MabPtr temp = m;
    for( ; temp != NULL; temp = temp->next ) {
    //while ( temp ) {
        printf("%3d       %3d       ", temp->offset, temp->size);
        if (temp->allocated)
            printf("TRUE\n");
        else
            printf("FALSE\n");
      //temp = temp->next;
    }
}
```

initialize the memory block for the buddy system

```
Mab1Ptr startMab1 ()
{
    Mab1Ptr m;
    if ( m = (Mab1Ptr) malloc( sizeof(Mab1) ) ) {
        m->offset = 0;
        m->size = 960;
        m->allocated = 0;
        m->next = NULL;
        m->prev = NULL;
        m->parent = NULL;
    }
```

```
    return m;
}
```
check if memory available for the buddy system
```
Mab1Ptr memChk1(Mab1Ptr m, int size)
allocate memory block for the buddy system
Mab1Ptr memAlloc1(Mab1Ptr m, int size) { // allocate memory block
    //m = memChk(m, size);
    //printf("3");
    if (m == NULL) return NULL;
    while ((m->size)/2 >= size)//printf("2");
        m = memSplit1(m, size);
    m->allocated = 1;
    return m;
}
```
free memory block for the bubby system
```
Mab1Ptr memFree1(Mab1Ptr m) {        // free memory block
    if (m == NULL) return NULL;
    m->allocated = 0;
    while ( m->parent !=NULL && m->parent->prev->allocated == 0 &&
m->parent->next->allocated == 0 ) //{
        m = memMerge1(m);//printf("6");}
    //printf("TESTING\n");
    return m;
}
```

merge two memory blocks for the buddy system
```
Mab1Ptr memMerge1(Mab1Ptr m)
split a memory block for the buddy system
Mab1Ptr memSplit1(Mab1Ptr m, int size)
print the memory arena for the buddy system
void printMab1(Mab1Ptr m)
```

Initialize the resource constraints
```
RsrcPtr
startRsrc () {
    RsrcPtr r;

    if ( r = (RsrcPtr) malloc( sizeof(Rsrc) ) ) {
        r->printers = 2;
        r->scanners = 1;
        r->modems = 1;
        r->cds = 2;
    }
    return r;
```

```
}
```

check if the resources that the process required is available

int

```
rsrcChk(RsrcPtr r, PcbPtr p) {
  return ( (r->printers >= p->printers) && (r->scanners >= p->scanners)
&& (r->modems >= p->modems) && (r->cds >= p->cds) );
}
```

allocate the resources that the process requires

```
RsrcPtr
rsrcAlloc(RsrcPtr r, PcbPtr p) {
  r->printers -= p->printers;
  r->scanners -= p->scanners;
  r->modems -= p->modems;
  r->cds -= p->cds;
  return r;
}
```

release the resources that the process held back to the dispatcher

```
RsrcPtr
rsrcFree(RsrcPtr r, PcbPtr p) {
  r->printers += p->printers;
  r->scanners += p->scanners;
  r->modems += p->modems;
  r->cds += p->cds;
  return r;
}
```

print the current number of resources that is available in the dispatcher

```
void
printRsrc(RsrcPtr r) {
  printf("printers  #scanners  #modems  #CDs");
  printf("    %d          %d          %d        %d\n",  r->printers,
r->scanners, r->modems, r->cds);
}
```

In host_new.c. there is main policy of dispatching processes and manage memory and recourses.

Variable description:

```
PcbPtr pcbQ = NULL; // hold all processes to be executed
PcbPtr realQ = NULL;    // queue for real time process
PcbPtr jobQ = NULL; // queue for allocating memory
PcbPtr fb1Q = NULL; // R1 in feedback
```

```
PcbPtr fb2Q = NULL; //R2 in feedback
PcbPtr fb3Q = NULL;     //R3 in feedback
PcbPtr activePcb = NULL;    //the process is executing
MabPtr m = startMab ();     //use for memory allocation
Mab1Ptr mm = startMab1 ();  //use for memory allocation in buddy system
int choice      //decide the kind of memory allocation policy
RsrcPtr r = startRsrc ();   //use for recourse allocation
```

Function Description

`while ( realQ || pcbQ || jobQ || fb1Q || fb2Q || fb3Q || activePcb ) { }`
this is main algorithm part which if no process is to be executed then end. Then as project part 1, we classify several situations to discuss.

```
        while (pcbQ && pcbQ->arrivaltime <= dispatchertime)
            if (pcbQ->priority == 0)
              realQ = enqPcb(realQ, deqPcb(&pcbQ));
            else
              jobQ = enqPcb(jobQ, deqPcb(&pcbQ));
```

First situation, when pcbQ is not empty and the first process have arrival then we should decide whether it is real time.

```
    if ( activePcb ) {
            if ( --(activePcb->remainingcputime) <= 0 ) {
    for ( i=0; i < 10 && activePcb->pid != joblist_id[i]; i++ ) {}
                if (activePcb->priority != 0)
                {
                   memFree(jobmem_offset[i]);
                     rsrcFree(r, activePcb);
                }
                terminatePcb(activePcb);
                free(activePcb);
                activePcb = NULL;
            }}
```

This is second situation, when processor is executing, then check whether the current process is finished.

Then we will deal with the situation which there are real time process arrived. When the processor is executing, check the currently process is real time or not, if so, go on execute, or suspend it.

```
    if( realQ )
     {
        if( activePcb )
        {
        if( activePcb->priority > 0 )             {
            .
```

```
        .
        .
        .
    }
    else
    {
      .
        .
        .
     }

}
```

After that, if no real time arrived, then check the user queue to see whether a appropriate process should be put into processor. We use four policies of memory allocations ,which are:

```
if(userQ){
            n_normal = memChk(mem_normal, userQ->mbytes);
         if (n_normal != NULL) {
             switch ( selection ) {
                  case 1:
                      .
                      .
                      .
                      //1). best fit

                  break;
               case 2:
                      .
                      .
                      .
                      // first fit
                case 3:
                      .
                      .
                      .
                      //(3). next fit
                  break;
                case 4:
                      .
                      .
                      .
                      //(4). worse fit
                    }
               }
```

After that we check the priority queues to find the highest priority queue and suspend the currently executing process, just like

```
  if (fbQ1 || fbQ2 || fbQ3)
        {
            if(activePcb)
            {
                suspendPcb(activePcb);
                        .
                        .
                        .
            }
            if ( fbQ1 )
              activePcb = deqPcb(&fbQ1);
            else if ( fbQ2 )
                activePcb = deqPcb(&fbQ2);
            else if (fbQ3)
                activePcb = deqPcb(&fbQ3);
            startPcb(activePcb);
    for ( i=0; i < 10 && activePcb->pid != joblist_id[i]; i++ ) {}
        if ( i == 10 )
                joblist_id[j++] = activePcb->pid;
        }
    }
```

Then times up:

```
        sleep(1);
        dispatchertime++;
```

At last we discuss the last situation, when the processor is free, then print and save result, or go on executing.

(4) Techniques in handling exceptions:

(i). file-open exception: avoid errors because of system.

```
scanf( "%s" , filename ) ;
fp = fopen( filename , "r" );
if (fp == NULL) {
    perror(" Input File ");
    exit(1);
}
```

(ii). if at the specific instant, no process can be dispatched to run, we print “Idle”instead.

```
if (job_per_s[i] == -10)
          printf("Idle ");
```

(5) Parameter modification:

Since in part 2, we involved kinds of structures to represent process, memory and recourse, so the parameters should become complexly. Base on data structure, to finish a dispatch, we need a structure to represent a block of memory, a structure to represent a group of recourses, and 3 priority queue for feedback algorithm. With these parameter completely, the dispatcher will be ok.

(6) Achievement in program requirements and unfinished tasks:

In this part, we simulate processes, blocks of memory, recourses successfully, and discuss the memory allocation in 4 policies, and combine with feedback scheduling. About unfinished tasks, we design the feedback only with time slot is 1 second, that's a pity

(7) Further implementations and what you have learnt:

In this part, I learn the policies about the memory allocation and recourses arrangement, and learn some skills for simulate blocks of memory and recourses and processes, and know how to implement different kink of memory allocation and processes dispatch policies.

(8) Reference of dispatcher working:

It is easy to say that, the arrived processes will be put in the particular queue for waiting for enough memory and recourse, then real time processes will be executed preferred, then the last processes will be put in the priority queue to waiting for their execution according to feedback (q = 1)algorithm

(9). Snapshots:

(i) Files included for successful program execution:

Sigtrap.c, mab.c, mab.h, pcb.h, pcb.c, host_new1.c files and other input files are included.

(ii) Command typed in compiling your program:

gcc –o process sigtrap.c

gcc –o host_new1 host_new1.c

(iii) Command typed in executing your program:

./host_new1

(iv) Snapshots to illustrate how you achieve the project requirements.

The best fit:

```
[da72903@umacnx25 ~/part2]$ ./host_new1
 Input File = feedback1.txt
select the number of the memory allocation method:
(1). best fit
(2). first fit
(3). next fit
(4). worse fit
1
    pid arrive  prior    cpu Mbytes     prn    scn    modem    cd  status
  10335      0      1      4     64       0      0       0      0  RUNNING
  10335; START
  10335; tick 1
  10335; SIGTSTP
    pid arrive  prior    cpu Mbytes     prn    scn    modem    cd  status
  10336      0      1      4     64       0      0       0      0  RUNNING
  10336; START
  10336; tick 1
  10336; SIGTSTP
  10335; SIGCONT
  10335; tick 2
  10335; SIGTSTP
  10336; SIGCONT
  10336; tick 2
  10336; SIGTSTP
  10335; SIGCONT
  10335; tick 3
  10335; SIGTSTP
  10336; SIGCONT
  10336; tick 3
  10336; SIGTSTP
    pid arrive  prior    cpu Mbytes     prn    scn    modem    cd  status
  10337      6      1      3     64       0      0       0      0  RUNNING
  10337; START
  10337; tick 1
  10337; SIGTSTP
  10337; SIGCONT
  10337; tick 2
  10337; SIGTSTP
  10335; SIGCONT
  10335; tick 4
  10335; SIGINT
  10336; SIGCONT
  10336; tick 4
  10336; SIGINT
  10337; SIGCONT
  10337; tick 3
  10337; SIGINT
Dispatcher is free at DispatcherTime = 12.
The finish time is 11 s.
The job list is:1 2 1 2 1 2 3 3 1 2 3
```

The first fit:

```
[da72903@umacnx25 ~/part2]$ ./host_new1
 Input File = feedback1.txt
select the number of the memory allocation method:
(1). best fit
(2). first fit
(3). next fit
(4). worse fit
2
    pid arrive  prior     cpu Mbytes      prn     scn     modem    cd  status
  10362      0      1       4     64        0       0        0      0  RUNNING
  10362; START
  10362; tick 1
  10362; SIGTSTP
    pid arrive  prior     cpu Mbytes      prn     scn     modem    cd  status
  10363      0      1       4     64        0       0        0      0  RUNNING
  10363; START
  10363; tick 1
  10363; SIGTSTP
  10362; SIGCONT
  10362; tick 2
  10362; SIGTSTP
  10363; SIGCONT
  10363; tick 2
  10363; SIGTSTP
  10362; SIGCONT
  10362; tick 3
  10362; SIGTSTP
  10363; SIGCONT
  10363; tick 3
  10363; SIGTSTP
    pid arrive  prior     cpu Mbytes      prn     scn     modem    cd  status
  10364      6      1       3     64        0       0        0      0  RUNNING
  10364; START
  10364; tick 1
  10364; SIGTSTP
  10364; SIGCONT
  10364; tick 2
  10364; SIGTSTP
  10362; SIGCONT
  10362; tick 4
  10362; SIGINT
  10363; SIGCONT
  10363; tick 4
  10363; SIGINT
  10364; SIGCONT
  10364; tick 3
  10364; SIGINT
Dispatcher is free at DispatcherTime = 12.
The finish time is 11 s.
The job list is:1 2 1 2 1 2 3 3 1 2 3
```

The next fit:

```
[da72903@umacnx25 ~/part2]$ ./host_new1
 Input File = feedback1.txt
select the number of the memory allocation method:
(1). best fit
(2). first fit
(3). next fit
(4). worse fit
3
    pid arrive  prior    cpu Mbytes     prn    scn   modem    cd  status
  10373      0      1      4     64       0      0       0     0  RUNNING
  10373; START
  10373; tick 1
  10373; SIGTSTP
    pid arrive  prior    cpu Mbytes     prn    scn   modem    cd  status
  10374      0      1      4     64       0      0       0     0  RUNNING
  10374; START
  10374; tick 1
  10374; SIGTSTP
  10373; SIGCONT
  10373; tick 2
  10373; SIGTSTP
  10374; SIGCONT
  10374; tick 2
  10374; SIGTSTP
  10373; SIGCONT
  10373; tick 3
  10373; SIGTSTP
  10374; SIGCONT
  10374; tick 3
  10374; SIGTSTP
    pid arrive  prior    cpu Mbytes     prn    scn   modem    cd  status
  10375      6      1      3     64       0      0       0     0  RUNNING
  10375; START
  10375; tick 1
  10375; SIGTSTP
  10375; SIGCONT
  10375; tick 2
  10375; SIGTSTP
  10373; SIGCONT
  10373; tick 4
  10373; SIGINT
  10374; SIGCONT
  10374; tick 4
  10374; SIGINT
  10375; SIGCONT
  10375; tick 3
  10375; SIGINT
Dispatcher is free at DispatcherTime = 12.
The finish time is 11 s.
The job list is:1 2 1 2 1 2 3 3 1 2 3
```

The worse fit:

```
[da72903@umacnx25 ~/part2]$ ./host_new1
 Input File = feedback1.txt
select the number of the memory allocation method:
(1). best fit
(2). first fit
(3). next fit
(4). worse fit
4
    pid arrive  prior    cpu Mbytes     prn    scn    modem    cd  status
  10390      0      1      4     64      0      0       0     0  RUNNING
  10390; START
  10390; tick 1
  10390; SIGTSTP
    pid arrive  prior    cpu Mbytes     prn    scn    modem    cd  status
  10391      0      1      4     64      0      0       0     0  RUNNING
  10391; START
  10391; tick 1
  10391; SIGTSTP
  10390; SIGCONT
  10390; tick 2
  10390; SIGTSTP
  10391; SIGCONT
  10391; tick 2
  10391; SIGTSTP
  10390; SIGCONT
  10390; tick 3
  10390; SIGTSTP
  10391; SIGCONT
  10391; tick 3
  10391; SIGTSTP
    pid arrive  prior    cpu Mbytes     prn    scn    modem    cd  status
  10392      6      1      3     64      0      0       0     0  RUNNING
  10392; START
  10392; tick 1
  10392; SIGTSTP
  10392; SIGCONT
  10392; tick 2
  10392; SIGTSTP
  10390; SIGCONT
  10390; tick 4
  10390; SIGINT
  10391; SIGCONT
  10391; tick 4
  10391; SIGINT
  10392; SIGCONT
  10392; tick 3
  10392; SIGINT
Dispatcher is free at DispatcherTime = 12.
The finish time is 11 s.
The job list is:1 2 1 2 1 2 3 3 1 2 3
```

Appendix

The code of main program:

```
#include <stdio.h>
#include <stdlib.h>
#include "pcb.c"

int main() {
    int dispatchertime = 0 ;
    int arrivalt , prior , cput, mbytes, printers, scanners, modems, cds;
    int selection, diff;
    int job_per_s[1000], joblist_id[1000];
    MabPtr jobmem_offset[10];
    Mab1Ptr jobmem_offset1[10];
    int i = 0, j = 0, k = 0;
    char filename[20] ;
    FILE *fp;
    //Initialize dispatcher queues
    PcbPtr pcbQ = NULL;
    PcbPtr realQ = NULL;
    PcbPtr userQ = NULL;
    PcbPtr fbQ1 = NULL;
    PcbPtr fbQ2 = NULL;
    PcbPtr fbQ3 = NULL;
    PcbPtr tempPcb;
    PcbPtr activePcb = NULL;
    MabPtr mem_normal = startMab ();
    MabPtr n_normal, o , p = mem_normal;
    RsrcPtr r = startRsrc ();
    for (i = 0; i < 10; i++)
        joblist_id[i] = 999;
    for (i = 0; i < 100; i++)
        job_per_s[i] = -1;
    printf( " Input File = " );
    scanf( "%s" , filename ) ;
    printf("select the number of the memory allocation method:\n");
    printf("(1). best fit\n(2). first fit\n(3). next fit\n(4). worse
fit\n");
    scanf("%d", &selection);
    fp = fopen( filename , "r" );
    if (fp == NULL) {
        perror(" Input File ");
        exit(1);
    }
```

```
    //Fill dispatcher queue from dispatch list file
    while ( fscanf ( fp , "%d , %d , %d , %d , %d , %d , %d , %d" , &arrivalt,
&prior, &cput, &mbytes, &printers, &scanners, &modems, &cds) != EOF ) {
        tempPcb = createnullPcb();
        tempPcb->arrivaltime = arrivalt;
        tempPcb->remainingcputime = cput;
        tempPcb->priority = prior;
    tempPcb->mbytes = mbytes;
       tempPcb->printers = printers;
       tempPcb->scanners = scanners;
       tempPcb->modems = modems;
       tempPcb->cds = cds;
        pcbQ = enqPcb(pcbQ , tempPcb);
    }
    fclose( fp );

    //While there's anything in the queue or there is a currently running
process
    while ( realQ || pcbQ || userQ || fbQ1 || fbQ2 || fbQ3 || activePcb )
{
        while (pcbQ && pcbQ->arrivaltime <= dispatchertime)
            if (pcbQ->priority == 0)
              realQ = enqPcb(realQ, deqPcb(&pcbQ));
            else
              userQ = enqPcb(userQ, deqPcb(&pcbQ));

     if ( activePcb ) {
            //Decrement process remainingcputime and check it is time up
or not
            if ( --(activePcb->remainingcputime) <= 0 ) {
                //Terminate it and free up memory
                for ( i=0; i < 10 && activePcb->pid != joblist_id[i]; i++ )
{}
                if (activePcb->priority != 0)
                {
                  memFree(jobmem_offset[i]);
                    rsrcFree(r, activePcb);
                }
                terminatePcb(activePcb);
                free(activePcb);
                activePcb = NULL;
            }
            }
     if( realQ )
```

```
    {
        if( activePcb )
        {
        if( activePcb->priority > 0 ) //preempt the user process when real
time process reach and decrese the priority of the preempted process
            {
                suspendPcb(activePcb);
                if ( activePcb->priority < 3 ) activePcb->priority++;
                if ( activePcb->priority == 2 )
                    fbQ2 = enqPcb(fbQ2, activePcb);
                else
                    fbQ3 = enqPcb(fbQ3, activePcb);
                activePcb = deqPcb(&realQ);
                if(activePcb->pid == 0)
                {
                  startPcb(activePcb);

                  joblist_id[j++] = activePcb->pid;
                }
                else
                    startPcb(activePcb);
            }
        }
        else
        {
            //Dequeue process and start it
            activePcb = deqPcb(&realQ);
            startPcb(activePcb);
            joblist_id[j++] = activePcb->pid;
        }

   }
    else if((userQ || fbQ1 || fbQ2 || fbQ3) && (!activePcb || (activePcb
&& activePcb->priority != 0)))
   {   //while there are enough resources and memory for the process,
put the process into the feedback queue and hold the resource until the
prcess is terminated
        if(userQ){
            n_normal = memChk(mem_normal, userQ->mbytes);
         if (n_normal != NULL) {
             switch ( selection ) {
                  case 1:
                      diff = n_normal->size - userQ->mbytes;
                      for ( o = memChk(n_normal->next, userQ->mbytes);
```

```
o != NULL && o != n_normal; o = memChk(o->next, userQ->mbytes) ) {
                        if ( ( o->size - userQ->mbytes ) < diff ) {
                          diff = o->size - userQ->mbytes;
                          p = o;
                       }
                    }
                    n_normal = p;
                    break;
                 case 2:
                     break;
                  case 3:
                     n_normal = memChk(p, userQ->mbytes);
                    break;
                  case 4:
                     diff = n_normal->size - userQ->mbytes;
                       for ( o = memChk(n_normal->next, userQ->mbytes); o !=
NULL && o != n_normal; o = memChk(o->next, userQ->mbytes) ) {
                          if ( ( o->size - userQ->mbytes ) > diff ) {
                            diff = o->size - userQ->mbytes;
                            p = o;
                         }
                      }
                      n_normal = p;
                 }  }   }
                   //allocate memory to the process

          while(userQ && n_normal!=NULL && rsrcChk(r, userQ))
        {
                   r = rsrcAlloc(r, userQ);
                        jobmem_offset[k++]  =  (p  =  memAlloc(n_normal,
userQ->mbytes));
                   if(userQ->priority == 1)
                        fbQ1 = enqPcb(fbQ1, deqPcb(&userQ));
                   else if(userQ->priority == 2)
                        fbQ2 = enqPcb(fbQ2, deqPcb(&userQ));
                   else if(userQ->priority == 3)
                        fbQ3 = enqPcb(fbQ3, deqPcb(&userQ));

    if(userQ) {
             n_normal = memChk(mem_normal, userQ->mbytes);

          if (n_normal != NULL) {
              switch ( selection ) {
                   case 1:
```

```
                    diff = n_normal->size - userQ->mbytes;
                    for ( o = memChk(n_normal->next, userQ->mbytes);
o != NULL && o != n_normal; o = memChk(o->next, userQ->mbytes) ) {
                   if ( ( o->size - userQ->mbytes ) < diff ) {
                     diff = o->size - userQ->mbytes;
                     p = o;
                  }
               }
               n_normal = p;
               break;
            case 2:
                break;
             case 3:
                n_normal = memChk(p, userQ->mbytes);
               break;
             case 4:
                diff = n_normal->size - userQ->mbytes;
                  for ( o = memChk(n_normal->next, userQ->mbytes); o !=
NULL && o != n_normal; o = memChk(o->next, userQ->mbytes) ) {
                     if ( ( o->size - userQ->mbytes ) > diff ) {
                       diff = o->size - userQ->mbytes;
                       p = o;
                    }
                 }
                 n_normal = p;
            }  } }
        }

        //find the non empty queue with highest priority
       if (fbQ1 || fbQ2 || fbQ3)
        {
            if(activePcb)
            {
                suspendPcb(activePcb);
                if ( activePcb->priority < 3 ) activePcb->priority++;
                if ( activePcb->priority == 2 )
                    fbQ2 = enqPcb(fbQ2, activePcb);
                else
                    fbQ3 = enqPcb(fbQ3, activePcb);
                activePcb = NULL;
            }
            if ( fbQ1 )
              activePcb = deqPcb(&fbQ1);
            else if ( fbQ2 )
```

```
                activePcb = deqPcb(&fbQ2);
            else if (fbQ3)
                activePcb = deqPcb(&fbQ3);
            startPcb(activePcb);
            for ( i=0; i < 10 && activePcb->pid != joblist_id[i]; i++ )
{}
        if ( i == 10 )
                joblist_id[j++] = activePcb->pid;
        }
    }

        //sleep for one second and increment dispatcher timer
        sleep(1);
        dispatchertime++;

        if ( !activePcb ) {
            printf( "Dispatcher is free at DispatcherTime = %d.\n" ,
dispatchertime ) ;
            job_per_s[dispatchertime] = -1;
        } else
            for (i = 0; i < 10; i++)
                if (activePcb->pid == joblist_id[i])
                    job_per_s[dispatchertime] = i+1;
        }
    //print out the job list and finish time
    printf("The finish time is %d s.\n", dispatchertime-1);
    printf("The job list is:");
    for (i = 1; i < dispatchertime; i++) {
        if (job_per_s[i] == -1)
            printf("Idle ");
        else
            printf("%d ", job_per_s[i]);
    }
    printf("\n");
    return 0 ;
}
```

The code of mab.c file:

```
#include "mab.h"

// initialize the memory block
MabPtr startMab ()
```

```
{
    MabPtr m;
    if ( m = (MabPtr) malloc( sizeof(Mab) ) ) {
        m->offset = 0;
        m->size = 960;
        m->allocated = 0;
        m->next = NULL;
        m->prev = NULL;
    }
    return m;
}

// check if memory available
MabPtr memChk(MabPtr m, int size) {  // check if memory available
    //printf("%d %d 1", m->size, m->allocated);
    MabPtr n = m;
    if (m == NULL) return NULL;
    while ( n != NULL && ( n->size < size || n->allocated != 0 )) {
        n = n->next; //printf("%d 2", n->size);
    }
// if not find free memory from m to bottom, look for free memory from
m to top
    if ( n == NULL && m->offset != 0 ) {
        n = m;
        while ( n->offset != 0)
            n = n->prev;
        while ( n != m && ( n->size < size || n->allocated != 0 )) {
            n = n->next; //printf("%d 2", n->size);
        }
        if ( n == m )
            n = NULL;
    }
    //printf("%d 3", m->size);
    return n;
}

// allocate memory block
MabPtr memAlloc(MabPtr m, int size) { // allocate memory block
    //m = memChk(m, size);
    //printf("3");
    if (m == NULL) return NULL;
    if (m->size != size)//printf("2");
        m = memSplit(m, size);
    m->allocated = 1;
```

```
  return m;
}

// free memory block
MabPtr memFree(MabPtr m) {        // free memory block
  if (m == NULL) return NULL;
  m->allocated = 0;
  m = memMerge(m);
  return m;
}

// merge two memory blocks
MabPtr memMerge(MabPtr m) {
      //printf("1");        // merge two memory blocks
  if (m->next != NULL && m->next->allocated == 0) {
    m->size += m->next->size;
    //printf("2");
    if(m->next->next != NULL) {
      m->next = m->next->next;
      //printf("3");
      free(m->next->prev);
      //free(m->next);
      //printf("4");
      m->next->prev = m;
      //printf("5");
    }
    else {
      //printf("a");
      free(m->next);
      //printf("b");
      m->next = NULL;
    }
  }
  //printf("9");
  if (m->prev != NULL && m->prev->allocated == 0) {
    MabPtr n;
    m->prev->size += m->size;
    //printf("q");
    //if (m->prev->prev != NULL) {
      //printf("w");
    /*n = m->prev;
    m->prev = n->prev;
    m->offset = n->offset;
    free(n);
```

```
     m->prev->next = m;*/
        m->prev->next = m->next;
        //printf("e");
     if(m->next != NULL)
           m->next->prev = m->prev;
        //printf("r");
        //n = m->next;
        //printf("t");
        free(m);
     //}
     //else {
        //printf("y");
    // m->offset = m->prev->offset;
        //free(m->prev);
        //printf("u");
        //m->prev = NULL;
     //}
  }
  return m;
}

// split a memory block
MabPtr memSplit(MabPtr m, int size) { // split a memory block
  MabPtr o = (MabPtr) malloc( sizeof(Mab) );
  MabPtr p; //printf("1");
  o->allocated = 0;
  o->size = m->size - size;
  m->size = size;
  //printf("1");
  if (m->next != NULL) {
  p = m->next;
  o->next = p;
  //printf("2");
  o->prev = m;
  //printf("3");
  m->next = o;
  //printf("4");
  p->prev = o;
  //printf("5");
  }
  else {
     m->next = o;
     o->prev = m;
     o->next = NULL;
```

```
        //printf("6");
    }
    o->offset = m->size + m->offset;
    return m;
}

//print the memory arena
void printMab(MabPtr m) {
    printf("offset      size      allocated   \n");
    MabPtr temp = m;
    for( ; temp != NULL; temp = temp->next ) {
    //while ( temp ) {
        printf("%3d       %3d       ", temp->offset, temp->size);
        if (temp->allocated)
            printf("TRUE\n");
        else
            printf("FALSE\n");
      //temp = temp->next;
    }
}

// initialize the memory block for the buddy system
Mab1Ptr startMab1 ()
{
    Mab1Ptr m;
    if ( m = (Mab1Ptr) malloc( sizeof(Mab1) ) ) {
        m->offset = 0;
        m->size = 960;
        m->allocated = 0;
        m->next = NULL;
        m->prev = NULL;
        m->parent = NULL;
    }
    return m;
}

// check if memory available for the buddy system
Mab1Ptr memChk1(Mab1Ptr m, int size) {  // check if memory available
    //printf("%d %d 1", m->size, m->allocated);
    Mab1Ptr n = m;
    if ( n != NULL && ( n->size >= size && n->allocated == 0 )) {
        if ( n->prev == NULL && n->next == NULL )
            return n;
        else if ( n->prev != NULL && n->next != NULL ) {
```

```
        Mab1Ptr o = n;
        n = memChk1(n->prev, size); //printf("%d 2", n->size);
        if ( n != NULL )
          return n;
        n = memChk1(o->next, size);
      }
      else if ( n->prev != NULL ) {
        n = memChk1(n->prev, size); //printf("%d 2", n->size);
          return n;
      }
      else if ( n->next != NULL ) {
        n = memChk1(n->next, size); //printf("%d 2", n->size);
          return n;
      }
    }
    else
      return NULL;
    //printf("%d 3", m->size);
}

// allocate memory block for the buddy system
Mab1Ptr memAlloc1(Mab1Ptr m, int size) { // allocate memory block
    //m = memChk(m, size);
    //printf("3");
    if (m == NULL) return NULL;
    while ((m->size)/2 >= size)//printf("2");
      m = memSplit1(m, size);
    m->allocated = 1;
    return m;
}

// free memory block for the bubby system
Mab1Ptr memFree1(Mab1Ptr m) {        // free memory block
    if (m == NULL) return NULL;
    m->allocated = 0;
    while  (  m->parent  !=NULL  &&  m->parent->prev->allocated  ==  0  &&
m->parent->next->allocated == 0 ) //{
      m = memMerge1(m);//printf("6");}
    //printf("TESTING\n");
    return m;
}

// merge two memory blocks for the buddy system
Mab1Ptr memMerge1(Mab1Ptr m) {
```

```
    //printf("1");       // merge two memory blocks
  if (m->parent->prev == m) {//printf("2");
    if (m->parent->next->allocated == 0 && m->parent->next->prev == NULL
&& m->parent->next->next == NULL) {//printf("3");
      m = m->parent;
      free(m->prev);
      free(m->next);
      m->prev = NULL;
      m->next = NULL;
      //printf("4");
    }
  }
  else {//printf("3");
    if (m->parent->prev->allocated == 0 && m->parent->prev->prev == NULL
&& m->parent->prev->next == NULL) {
      m = m->parent;
      free(m->prev);
      free(m->next);
      m->prev = NULL;
      m->next = NULL;
    }
  }
  //printf("5");
  return m;
}

// split a memory block for the buddy system
Mab1Ptr memSplit1(Mab1Ptr m, int size) { // split a memory block
  Mab1Ptr n = (Mab1Ptr) malloc( sizeof(Mab1) );
  Mab1Ptr o = (Mab1Ptr) malloc( sizeof(Mab1) );
  Mab1Ptr p; //printf("1");
  o->allocated = 0;
  n->allocated = 0;
  o->size = m->size/2;
  n->size = o->size;
  o->prev = NULL;
  o->next = NULL;
  n->prev = NULL;
  n->next = NULL;
  o->parent = m;
  n->parent = m;
  //printf("1");
  m->prev = n;
  m->next = o;
```

```
    o->offset = m->offset + o->size;
    n->offset = m->offset;
    return n;
}

//print the memory arena for the buddy system
void printMab1(Mab1Ptr m) {
    //printf("offset      size      allocated   \n");
    Mab1Ptr temp = m;

    if ( m != NULL ) {
        printMab1(m->prev);
        //printf("TESTING1\n");
        if( temp->prev == NULL && temp->next == NULL ) {
        //while ( temp ) {
            printf("%3d       %3d       ", temp->offset, temp->size);
            if (temp->allocated)
                printf("TRUE\n");
            else
                printf("FALSE\n");
           //temp = temp->next;
        }//printf("TESTING\n");
        printMab1(m->next);
        //printf("TESTING2\n");
    }
}

//Initialize the resource constraints
RsrcPtr
startRsrc () {
    RsrcPtr r;

    if ( r = (RsrcPtr) malloc( sizeof(Rsrc) ) ) {
        r->printers = 2;
        r->scanners = 1;
        r->modems = 1;
        r->cds = 2;
    }
    return r;
}

//check if the resources that the process required is available
int
rsrcChk(RsrcPtr r, PcbPtr p) {
```

```
    return ( (r->printers >= p->printers) && (r->scanners >= p->scanners)
&& (r->modems >= p->modems) && (r->cds >= p->cds) );
}

//allocate the resources that the process requires
RsrcPtr
rsrcAlloc(RsrcPtr r, PcbPtr p) {
    r->printers -= p->printers;
    r->scanners -= p->scanners;
    r->modems -= p->modems;
    r->cds -= p->cds;
    return r;
}

//release the resources that the process held back to the dispatcher
RsrcPtr
rsrcFree(RsrcPtr r, PcbPtr p) {
    r->printers += p->printers;
    r->scanners += p->scanners;
    r->modems += p->modems;
    r->cds += p->cds;
    return r;
}

//print the current number of resources that is available in the dispatcher
void
printRsrc(RsrcPtr r) {
    printf("printers  #scanners  #modems  #CDs");
    printf("    %d          %d          %d        %d\n",  r->printers,
r->scanners, r->modems, r->cds);
}
```

The code of mab.h file:

```
#ifndef MAB_H
#define MAB_H
#include <stdio.h>
#include <stdlib.h>

struct mab {
     int offset;
     int size;
     int allocated;
```

```
    struct mab * next;
    struct mab * prev;
};

typedef struct mab Mab;
typedef Mab * MabPtr;

MabPtr startMab ();               // initialize the memory block
MabPtr memChk(MabPtr m, int size);   // check if memory available
MabPtr memAlloc(MabPtr m, int size); // allocate memory block
MabPtr memFree(MabPtr m);          // free memory block
MabPtr memMerge(MabPtr m);         // merge two memory blocks
MabPtr memSplit(MabPtr m, int size); // split a memory block
void printMab(MabPtr m);           //print the memory arena

//#endif

//for buddy system
struct mab1 {
    int offset;
    int size;
    int allocated;
    struct mab1 * next;
    struct mab1 * prev;
    struct mab1 * parent;
};

typedef struct mab1 Mab1;
typedef Mab1 * Mab1Ptr;

Mab1Ptr startMab1 ();               // initialize the memory block
Mab1Ptr memChk1(Mab1Ptr m, int size);   // check if memory available
Mab1Ptr memAlloc1(Mab1Ptr m, int size); // allocate memory block
Mab1Ptr memFree1(Mab1Ptr m);          // free memory block
Mab1Ptr memMerge1(Mab1Ptr m);         // merge two memory blocks
Mab1Ptr memSplit1(Mab1Ptr m, int size); // split a memory block
void printMab1(Mab1Ptr m);           //print the memory arena

struct rsrc {
    int printers;
    int scanners;
    int modems;
    int cds;
};
```

```
typedef struct rsrc Rsrc;
typedef Rsrc * RsrcPtr;

RsrcPtr startRsrc ();                    // initialize the resource
constraints
int rsrcChk(RsrcPtr r, PcbPtr p);      // check if resource available
RsrcPtr rsrcAlloc(RsrcPtr r, PcbPtr p); // allocate resource
RsrcPtr rsrcFree(RsrcPtr r, PcbPtr p);  // free resource
void printRsrc(RsrcPtr r);            // print the resource arena

#endif
```