Operating Systems - EECE.5730

Instructor: *Prof. Dalila Megherbi*

Final Project-Part2

Due by 12-05-16

By,

- *Naga Ganesh Kurapati*
- *Vishal Sundarrajan*
- *-Swapnil Chaghule*

1) Objective:

In this project, we are analyzing the different round robin scheduling algorithms Standard Round robin, adaptive RR, efficient RR based on average waiting time, average turn-around time, number of context switches. The process arrival time is random so as the waiting time. By analyzing the rankings of turn around and waiting times of different RR algorithms, we are going to present the performance of the RR algorithms based on the resulting parameters.

2) Background:

Standard Round Robin algorithm is used many used in time sharing and real time operating systems to schedule the processes. It gives equal time share to all the processes in the ready queue and response time is also very low. Despite of this it experiences high turnaround time, high waiting time and high number of context switches. But there are different other improved Round Robin algorithms that have improved performance the standard RR.

In adaptive RR, first processes are sorted by their burst times, shorter process at the front of the ready queue. If Even number of processes in queue, then time quantum is average of the all burst times. If odd, equal to burst time of the middle process in the ready queue. Any process comes in middle of the execution, then it is not scheduled in current round.

In efficient RR, Combines Shortest remaining algorithm and Standard RR. Process with shortest remaining burst time is selected first. When new processes come in, at end of time slice process with shortest remaining burst time is selected first. Long process will suffer starvation.

3) Algorithms/Functions used:

pthread_mutex_init() - to initialize mutex
sem_init() – to initialize semaphore
malloc() – to allocate memory
pthread_create() – to create pthreads
pthread_join() – to wait for threads to complete and join
pthread_mutex_destroy() – destroy mutex
sem_destroy() – destroy semaphore
sem_wait() – wait on semaphore
sem_post () – post the semaphore
pthread_mutex_lock() – mutex lock
pthread_mutex_unlock() -mutex unlock

User defined functions:

void enq(int data_1,int data_2) – To put data on to the queue
int empty() – Find whether queue is empty or not
void display() – Display the elements of a queue
void create() – To create a queue
int queuesize() – Return queue size
int addall() – To add all elements of a queue
int peekprocessID() – to return process ID of a process in the ready queue
int peekburst() - to return peekburst time of a process in the ready queue
int findndel_min() – find and delete minimum element in a queue
void swap() – To swap to elements in a queue
void SelectionSort() – TO sort elements in queue
void deq() – To delete elements in queue
void *processor() – Processor function need to be scheduled
void *scheduler() – Put the processes on to the ready queue

4) Results:

This section shows the screenshots of the standard RR, adaptive RR and efficient RR algorithm execution on a Linux. Pthreads are used for creating the real scenario of concurrency between the scheduler, new processes coming in to the ready queue and processes that are executing on the CPU. User need to supply the arrival time, burst time and priority of the process the coming on to the ready queue before starting the simulation of the RR algorithms.

```
Enter Process 1 Arrivaltime,Bursttime,Priority:
0,7,2
Enter Process 2 Arrivaltime,Bursttime,Priority:
0,5,1
Enter Process 3 Arrivaltime,Bursttime,Priority:
3,4,6
Enter Process 4 Arrivaltime,Bursttime,Priority:
5,4,4
Enter Process 5 Arrivaltime,Bursttime,Priority:
10,8,3
Enter Process 6 Arrivaltime,Bursttime,Priority:
13,8,5
Pushed process 1 in to the queue
Pushed process 2 in to the queue
Process 1 executed b/w the time 0 and 4
Pushed process 3 in to the queue
Process 2 executed b/w the time 4 and 8
Pushed process 4 in to the queue
Process 1 executed b/w time 8 and 11
Pushed process 5 in to the queue
Process 3 executed b/w time 11 and 15
Pushed process 6 in to the queue
Process 2 executed b/w time 15 and 16
Process 4 executed b/w time 16 and 20
Process 5 executed b/w the time 20 and 24
Process 6 executed b/w the time 24 and 28
Process 5 executed b/w time 28 and 32
Process 6 executed b/w time 32 and 36
```

Fig.1 – Standard RR

```
Enter Process 1 Arrivaltime,Bursttime,Priority:
0,7,2
Enter Process 2 Arrivaltime,Bursttime,Priority:
0,5,1
Enter Process 3 Arrivaltime,Bursttime,Priority:
3,4,6
Enter Process 4 Arrivaltime,Bursttime,Priority:
5,4,4
Enter Process 5 Arrivaltime,Bursttime,Priority:
10,8,3
Enter Process 6 Arrivaltime,Bursttime,Priority:
13,8,5
Pushed process 1 in to the queue
Pushed process 2 in to the queue
Process 2 executed b/w time 0 and 5
Process 1 executed b/w the time 5 and 11
Pushed process 3 in to the queue
Pushed process 4 in to the queue
Pushed process 5 in to the queue
Process 1 executed b/w time 11 and 12
Process 3 executed b/w time 12 and 16
Process 4 executed b/w time 16 and 20
Process 5 executed b/w the time 20 and 24
Pushed process 6 in to the queue
Process 5 executed b/w time 24 and 28
Process 6 executed b/w the time 28 and 34
Process 6 executed b/w time 34 and 36
```

Fig.2 – Adaptive RR

```
Enter Process 1 Arrivaltime,Bursttime,Priority:
0,7,2
Enter Process 2 Arrivaltime,Bursttime,Priority:
0,5,1
Enter Process 3 Arrivaltime,Bursttime,Priority:
3,4,6
Enter Process 4 Arrivaltime,Bursttime,Priority:
5,4,4
Enter Process 5 Arrivaltime,Bursttime,Priority:
10,8,3
Enter Process 6 Arrivaltime,Bursttime,Priority:
13,8,5
Pushed process 1 in to the queue
Pushed process 2 in to the queue
Time_quantum: 4
Process 2 executed b/w the time 0 and 4
Pushed process 3 in to the queue
Process 2 executed b/w time 4 and 5
Pushed process 4 in to the queue
Process 3 executed b/w time 5 and 9
Process 4 executed b/w time 9 and 13
Pushed process 5 in to the queue
Pushed process 6 in to the queue
Process 1 executed b/w the time 13 and 17
Process 1 executed b/w time 17 and 20
Process 5 executed b/w the time 20 and 24
Process 5 executed b/w time 24 and 28
Process 6 executed b/w the time 28 and 32
Process 6 executed b/w time 32 and 36
vishal@ubuntu:~/Documents/C_TestProgs$
```

Fig.3 – Efficient RR

5) Observations:

The processes are simulated for the given scenarios as shown in the below table

Process	Arrival Time	Burst Time	Priority
P1	0	7	2
P2	0	5	1
P3	3	4	6
P4	5	4	4
P5	10	8	3
P6	13	8	5

Type	Average Turnaround Time	Average Waiting Time	Number of Context Switches
Standard Round Robin	17.17	11.17	9
Adaptive RR	16.67	10.67	7
Efficient RR	13.33	7.33	8

Above table showing the comparison of average turnaround time, average waiting time, and number of context switches of different RR algorithms.

Efficient RR seems to be performing better than standard RR and Adaptive RR with low average turnaround time and average waiting time. But in real scenario, process with longer burst time will experience starvation. While the Adaptive RR is performing better than the standard RR. It has less number context switches than others.

6) Conclusions:

Even though standard round robin algorithm has equal time sharing and low response time but it experiencing high turnaround time, high waiting and number of context switches throughout the different scenarios of the processes scheduling. Adaptive RR and Efficient RR are performing better with low turnaround time, waiting and number of context switches when compared to Standard RR. So, they can be preferred for process scheduling. Even though Efficient is performing better than Adaptive RR, but it makes the longer burst time process to starve.

7) Source Code:

See the attachment to find the source code of standard RR, adaptive RR, efficient RR and ready queue algorithms respectively.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
#include <signal.h>
#include "ready_queue.h"

struct inp_parameters *param;
int updatedtimequantum=0,finish=0;
sem_t sema,sema1;

struct inp_parameters
{
  int processID,arrivaltime,bursttime,priority;
};

void *processor()
{
  int bursttime,processID,time_quanta=4,updatedburst,previoustime,count;
   while(1)
    {
      if(finish==0)
       sem_wait(&sema);
            if(empty()==1 && finish==1)//queue is empty and finish is true, so break
             {
               break;
             }
            else if(empty()==1) // queue is empty, increment time
             {
               updatedtimequantum++;
             }
            else //queue not empty,find min and execute for time quantum,write back if not
            finished
             {
               count=queuesize();
               bursttime=peekburst();
               processID=peekprocessID();
               deq();
               //printf("Time_quantum: %d\n",time_quanta);
               //printf("ProcessID: %d\n",processID);
               //printf("burst time: %d\n",bursttime);
               updatedburst=bursttime-time_quanta;
               //printf("updated burst: %d\n",updatedburst);
                 if(updatedburst>0)
                   {
                     enq(processID,updatedburst);
                     previoustime=updatedtimequantum;
                     updatedtimequantum=updatedtimequantum+time_quanta;
                     printf("Process %d executed b/w the time %d and
```

```
                    %d\n",processID,previoustime,updatedtimequantum);
                  }
                else
                  {
                    previoustime=updatedtimequantum;
                    updatedtimequantum=updatedtimequantum+bursttime;
                    printf("Process %d executed b/w time %d and
                    %d\n",processID,previoustime,updatedtimequantum);
                  }
                count--;
              }//else
      sem_post(&sema1);
      }
}

void *scheduler()
{
  int i=0;
    while(1)
     {
       //sleep(1);
        if(updatedtimequantum>=param[i].arrivaltime)  //push the process into the ready queue
          {
            if(i==6) //all processes moved to the ready queue
              {
               finish=1;
               break;
              }
            enq(param[i].processID,param[i].bursttime);
            printf("Pushed process %d in to the queue\n",param[i].processID);
            i++;
          }
        else
          {
            sem_post(&sema);
            sem_wait(&sema1);
          }
      }//while
    sem_post(&sema);
}

int main()
{
   param=malloc(8*sizeof(struct inp_parameters));
   create();
   int i;
    for(i=0;i<6;i++)
     {
      param[i].processID=i+1;
      printf("Enter Process %d Arrivaltime,Bursttime,Priority: \n",i+1);
      scanf("%d,%d,%d",&param[i].arrivaltime,&param[i].bursttime,&param[i].priority);
     }
   int state=sem_init(&sema,0,0);
```

```
    int state1=sem_init(&sema1,0,0);
      if(state||state1!=0)
        puts("Error in Mutex or semaphore initialization!!");
    pthread_t thread[2];
    int t;
    for (t=0;t<2;t++)
     {
     if(t==0)
      state=pthread_create(&thread[t],NULL,processor,NULL);
     else
      state=pthread_create(&thread[t],NULL,scheduler,NULL);
       if(state)
        {
         printf("\ncannot create thread\n");
         exit(-1);
        }
    }
    for(t=0;t<2;t++)
     {
       state=pthread_join(thread[t],NULL);
         if(state)
         {
          printf("\nError");
          exit(-1);
         }
     }
    sem_destroy(&sema);
    sem_destroy(&sema1);
    free(param);
    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
#include <signal.h>
#include "ready_queue.h"

struct inp_parameters *param;
int updatedtimequantum=0,totalprocesses,finish=0;
sem_t sema,sema1;

struct inp_parameters
{
  int processID,arrivaltime,bursttime,priority;
};

void *processor()
{
  int bursttime,processID,time_quanta,updatedburst,previoustime,count;
   while(1)
    {
      if(finish==0)
       sem_wait(&sema);
            if(empty()==1 && finish==1)//queue is empty and finish is true, so break
             {
               break;
             }
            else if(empty()==1) // queue is empty, increment time
             {
               updatedtimequantum++;
             }
            else //queue not empty,find min and execute for time quantum,write back if not
            finished
             {
               count=queuesize();
               time_quanta=addall()/count;
               SelectionSort();
              while(count>0)
              {
               //sleep(3);
               bursttime=peekburst();
               processID=peekprocessID();
               deq();
               //printf("count: %d\n",count);
               //printf("Time_quantum: %d\n",time_quanta);
               //printf("ProcessID: %d\n",processID);
               //printf("burst time: %d\n",bursttime);
               updatedburst=bursttime-time_quanta;
               //printf("updated burst: %d\n",updatedburst);
                 if(updatedburst>0)
```

```
                    {
                      enq(processID,updatedburst);
                      previoustime=updatedtimequantum;
                      updatedtimequantum=updatedtimequantum+time_quanta;
                      printf("Process %d executed b/w the time %d and
                      %d\n",processID,previoustime,updatedtimequantum);
                    }
                  else
                    {
                      previoustime=updatedtimequantum;
                      updatedtimequantum=updatedtimequantum+bursttime;
                      printf("Process %d executed b/w time %d and
                      %d\n",processID,previoustime,updatedtimequantum);
                    }
                  count--;
                }//while
              }//else
      sem_post(&sema1);

    }
}

void *scheduler()
{
  int i=0;
    while(1)
     {
       //sleep(1);
        if(updatedtimequantum>=param[i].arrivaltime)  //push the process into the ready queue
          {
            if(i==6) //all processes moved to the ready queue
              {
               finish=1;
               break;
              }
            enq(param[i].processID,param[i].bursttime);
            printf("Pushed process %d in to the queue\n",param[i].processID);
            i++;
          }
        else
          {
            sem_post(&sema);
            sem_wait(&sema1);
          }
      }//while
    sem_post(&sema);
}

int main()
{
   param=malloc(8*sizeof(struct inp_parameters));
   create();
   int i;
```

```
    for(i=0;i<6;i++)
     {
      param[i].processID=i+1;
      printf("Enter Process %d Arrivaltime,Bursttime,Priority: \n",i+1);
      scanf("%d,%d,%d",&param[i].arrivaltime,&param[i].bursttime,&param[i].priority);
     }
   int state=sem_init(&sema,0,0);
   int state1=sem_init(&sema1,0,0);
     if(state||state1!=0)
       puts("Error in Mutex or semaphore initialization!!");
   pthread_t thread[2];
   int t;
   for (t=0;t<2;t++)
    {
    if(t==0)
     state=pthread_create(&thread[t],NULL,processor,NULL);
    else
     state=pthread_create(&thread[t],NULL,scheduler,NULL);
      if(state)
       {
        printf("\ncannot create thread\n");
        exit(-1);
       }
   }
   for(t=0;t<2;t++)
    {
      state=pthread_join(thread[t],NULL);
        if(state)
        {
         printf("\nError");
         exit(-1);
        }
    }
   sem_destroy(&sema);
   sem_destroy(&sema1);
   free(param);
   return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
#include <signal.h>
#include "ready_queue.h"

struct inp_parameters *param;
int updatedtimequantum=0,finish=0;
sem_t sema,sema1;

struct inp_parameters
{
  int processID,arrivaltime,bursttime,priority;
};

void *processor()
{
  int bursttime,processID,time_quanta=4,updatedburst,previoustime,count;
  printf("Time_quantum: %d\n",time_quanta);
   while(1)
    {
      if(finish==0)
       sem_wait(&sema);
            if(empty()==1 && finish==1)//queue is empty and finish is true, so break
             {
               break;
             }
            else if(empty()==1) // queue is empty, increment time
             {
               updatedtimequantum++;
             }
            else //queue not empty,find min and execute for time quantum,write back if not
            finished
             {
               count=queuesize();
               bursttime=findndel_min();
               processID=getprocessID();
               //printf("ProcessID: %d\n",processID);
               //printf("burst time: %d\n",bursttime);
               updatedburst=bursttime-time_quanta;
               //printf("updated burst: %d\n",updatedburst);
                 if(updatedburst>0)
                   {
                     enq(processID,updatedburst);
                     previoustime=updatedtimequantum;
                     updatedtimequantum=updatedtimequantum+time_quanta;
                     printf("Process %d executed b/w the time %d and
                     %d\n",processID,previoustime,updatedtimequantum);
                   }
```

```
                    else
                      {
                        previoustime=updatedtimequantum;
                        updatedtimequantum=updatedtimequantum+bursttime;
                        printf("Process %d executed b/w time %d and
                        %d\n",processID,previoustime,updatedtimequantum);
                      }
                    count--;
                }//else
       sem_post(&sema1);
       }
}

void *scheduler()
{
  int i=0;
    while(1)
     {
       //sleep(1);
        if(updatedtimequantum>=param[i].arrivaltime)  //push the process into the ready queue
          {
            if(i==6) //all processes moved to the ready queue
              {
               finish=1;
               break;
              }
            enq(param[i].processID,param[i].bursttime);
            printf("Pushed process %d in to the queue\n",param[i].processID);
            i++;
          }
        else
          {
            sem_post(&sema);
            sem_wait(&sema1);
          }
      }//while
    sem_post(&sema);
}

int main()
{
   param=malloc(8*sizeof(struct inp_parameters));
   create();
   int i;
    for(i=0;i<6;i++)
     {
      param[i].processID=i+1;
      printf("Enter Process %d Arrivaltime,Bursttime,Priority: \n",i+1);
      scanf("%d,%d,%d",&param[i].arrivaltime,&param[i].bursttime,&param[i].priority);
     }
   int state=sem_init(&sema,0,0);
   int state1=sem_init(&sema1,0,0);
    if(state||state1!=0)
```

```
       puts("Error in Mutex or semaphore initialization!!");
   pthread_t thread[2];
   int t;
   for (t=0;t<2;t++)
    {
    if(t==0)
     state=pthread_create(&thread[t],NULL,processor,NULL);
    else
     state=pthread_create(&thread[t],NULL,scheduler,NULL);
      if(state)
       {
        printf("\ncannot create thread\n");
        exit(-1);
       }
   }
   for(t=0;t<2;t++)
    {
      state=pthread_join(thread[t],NULL);
        if(state)
        {
         printf("\nError");
         exit(-1);
        }
    }
   sem_destroy(&sema);
   sem_destroy(&sema1);
   free(param);
   return 0;
}
```

```
#ifndef _Queue_H
#define _Queue_H

#include <stdio.h>
#include <stdlib.h>

struct node
{
    int processID,remainingbursttime;
    struct node *ptr;
}*front,*rear,*temp,*front1,*beforemin,*replacebeforemin;

void enq(int data_1,int data_2);
int empty();
void display();
void create();
int queuesize();
int addall();
int peekprocessID();
int peekburst();
int findndel_min();
void swap();
void SelectionSort();
void deq();
int count = 0;
int previousprocessID;

void create()
{
    front = rear = NULL;
}

int queuesize()
{
    return count;
}

/* Enqueing the queue */
void enq(int data_1,int data_2)
{
    if (rear == NULL)
    {
        rear = (struct node *)malloc(1*sizeof(struct node));
        rear->ptr = NULL;
        rear->processID = data_1;
        rear->remainingbursttime = data_2;
        front = rear;
    }
    else
    {
        temp=(struct node *)malloc(1*sizeof(struct node));
        rear->ptr = temp;
        temp->processID = data_1;
```

```
        temp->remainingbursttime = data_2;
        temp->ptr = NULL;
        rear = temp;
    }
    count++;
}

void deq()
{
    front1 = front;

    if (front1 == NULL)
    {
        return;
    }
    else
        if (front1->ptr != NULL)
      {
            front1 = front1->ptr;
            //printf("\n Dequed value : %d", front->info);
            free(front);
            front = front1;
        }
        else
        {
            //printf("\n Dequed value : %d", front->info);
            free(front);
            front = NULL;
            rear = NULL;
        }
        count--;
}

void SelectionSort()
{
    struct node *start = front;
    struct node *traverse;
    struct node *min;

    while(start->ptr)
    {
        min = start;
        traverse = start->ptr;

        while(traverse)
        {
            /* Find minimum element from array */
            if(min->remainingbursttime > traverse->remainingbursttime )
            {
                min = traverse;
            }
                    traverse = traverse->ptr;
        }
```

```
        swap(start,min);            // Put minimum element on starting location
                //temp=start;
        start = start->ptr;
    }
}

/* swap data field of linked list */
void swap(struct node *p1, struct node *p2)
{
    int temp = p1->remainingbursttime;
        int temp2= p1->processID;
    p1->remainingbursttime = p2->remainingbursttime;
        p1->processID = p2->processID;
    p2->remainingbursttime = temp;
        p2->processID = temp2;
}

void display()
{
    front1 = front;

    if ((front1 == NULL) && (rear == NULL))
    {
        printf("Queue is empty\n");
        return;
    }
    while (front1 != rear)
    {
        printf("(%d,%d)\n", front1->processID,front1->remainingbursttime);
        front1 = front1->ptr;
    }
    if (front1 == rear)
        printf("(%d,%d)\n", front1->processID,front1->remainingbursttime);
}

int addall()
{
  int cummulativeburst=0;
   if ((front == NULL) && (rear == NULL))
    {
        return 0;
    }
   for(front1=front;front1!=rear;front1=front1->ptr)
    {
      cummulativeburst = cummulativeburst + front1->remainingbursttime;
    }
   cummulativeburst = cummulativeburst + front1->remainingbursttime;
  return cummulativeburst;
}

int findndel_min()
{
   if ((front == NULL) && (rear == NULL))
```

```
 {
     return 0;
 }
int min=front->remainingbursttime;
replacebeforemin=front;
temp=front;
previousprocessID=front->processID;
for(front1=front;front1!=rear;front1=front1->ptr)
 {
    if(front1->remainingbursttime<min)
      {
       min=front1->remainingbursttime;
       temp=front1;
       previousprocessID=temp->processID;
       replacebeforemin=beforemin;
      }
   beforemin=front1;
 }
 if(front1->remainingbursttime<min)
      {
       min=front1->remainingbursttime;
       temp=front1;
       replacebeforemin=beforemin;
       previousprocessID=temp->processID;
      }
//delete min, temp stores the minimum's pointer
 if(temp==front && front->ptr!=NULL)//first node is the minimum
  {
    replacebeforemin=replacebeforemin->ptr;
    free(front);
    front=replacebeforemin;
  }
 else if(front->ptr==NULL)
  {
    free(front);
    front=NULL;
    rear=NULL;
  }
else if(temp==rear)//last node is minimum
  {
    rear=replacebeforemin;
    rear->ptr=NULL;
    free(temp);
  }
else if(front->ptr==temp)//2nd node is the minimum
  {
    front->ptr=temp->ptr;
    free(temp);
  }
else //middle node is minimum
  {
    replacebeforemin->ptr=temp->ptr;
    free(temp);
```

```
     }
   count--;
  return min;
}

int peekprocessID()
{
    if ((front != NULL) && (rear != NULL))
        return(front->processID);
    else
        return 0;
}

int getprocessID()
{
  return previousprocessID;
}

int peekburst()
{
    if ((front != NULL) && (rear != NULL))
        return(front->remainingbursttime);
    else
        return 0;
}


int empty()
{
     if ((front == NULL) && (rear == NULL))
        return 1;
    else
        return 0;
}

#endif
```