CPU scheduler report

2013150133 Jeongho Yoon

서론

CPU 스케줄링은 실험실 사용을 위한 시간표를 짜는 것에 비유할 수 있다. 만약 실험실이 하나이고 여러 반이 그 실험실을 사용하려고 한다고 하면, 그 실험실을 나누어 사용하기 위한 시간표가 필요하다. 이와 유사하게, 여러 개의 프로세스가 프로세서를 사용해야 한다면, 어떤 프로세스가 먼저 프로세서를 사용할 것이고 어떤 프로세스가 나중에 사용할 것인지를 결정하기 위한 일종의 스케줄링이 필요하다. 이것이 CPU 스케줄링이다.

CPU 스케줄링은 멀티태스킹을 지원하는 체제에서 필수적이다. 전력 장벽에 다다른 뒤 코어의 수가 증가하고 있다고는 하지만, 여전히 많은 경우에 프로세스의 수가 프로세서의 수보다 많다. 여러 개의 프로세스들이 한 개의 프로세서를 나누면서 문제를 일으키지 않게 하기 위해서는, CPU 스케줄러가 필요하다.

CPU 스케줄러를 평가하는 지표에는 여러 가지가 있다. 여기에는 프로세스가 waiting queue 에서 기다리는 시간을 의미하는 waiting time, 단위 시간 동안 완성한 프로세스의 수를 의미하는 throughput, 프로세스가 프로세서를 할당받기 전까지 기다려야 하는 시간인 latency 등이 있다. 또한, 프로세서가 여러 프로세스들 간에 얼마나 공평하게 할당되었는지를 의미하는 fairness, 데드라인을 지켰는지, 또한 priority 가 존중되었는지도 CPU 스케줄러의 평가 항목이 될 수 있다. 이러한 항목들 중 일부는 서로 충돌하는 경우가 있다. 예를 들어, throughput 과 latency 는 서로 충돌할 가능성이 있다. 이러한 경우에는, 스케줄러는 적절한 결정을 내려야 할 것이다.

본론

이 부분에서는 CPU 스케줄러 시뮬레이터를 설명한다.

먼저, 시뮬레이터가 스케줄링하는 것은 프로세스이다. 따라서, 프로세스들을 나타내기 위한 struct 가 만들어졌다.

```
struct ProcStruct {
   int pid;
   int CPUburst;
   int IOburst;
   int priority;
   int arrival;
   int finishedTime;
   int turnaroundTime;
   int waitingTime;
};
```

pid 는 프로세스를 identify 하는 고유 번호, CPUburst 는 CPU 를 사용해야 하는 시간, IOburst 는 IO 버스트 시간, Priority 는 우선순위, arrival 은 프로세스가 waiting queue 에 도착한 시간, finishedTime 은 프로세스가 끝난 시간, turnaroundTime 은 프로세스가 도착하여 끝날 때까지의 시간, 그리고 waitingTime 은 프로세스가 waiting queue 에서 보낸 시간이다.

프로세스들은 randomProcessAlloc() 을 통해 랜덤하게 만들어진다.

스케줄러를 만들 때 자료구조를 활용하면 더욱 효율적인 구성이 가능하다. 예를 들어, priority scheduling 에서는 priority queue 를 사용하면 효율적이다. Priority queue 는 max heap 을 이용하면 효율적이다. 또한, First come

first served 나 round robin 에서는 queue 가 효율적이다. 이러한 자료구조들 또한 만들어졌다. 그리고 각 자료구조에 해당하는 작업들, 즉 enqueue(), dequeue(), enheap(), deheap(), minHeapify() 등도 만들어졌다.

```
struct NodeArray {
   Process *arr[512];
   int head;
   int tail;
};
typedef struct NodeArray Queue;
typedef struct NodeArray Heap;
```

보고서 맨 뒤에 이에 해당하는 코드가 있다.

isEmpty() 와 isFull() 은 자료구조에 해당하는 배열이 비어있거나 차있는지를 알려준다. enqueue() 는 프로세스를 queue 에 넣는다. dequeue() 는 queue 에서 프로세스를 가져온다. minHeapify() 는 heap 이 min heap property 를 유지하도록 도와준다. enheap() 은 프로세스를 heap 에 넣고, deheap() 은 heap 에서 프로세스를 꺼내온다.

조금 더 세부 사항을 설명하자면, 두 종류의 heap 작업이 만들어졌다. 이는 어떤 경우에는 min heap 이 job length 에 따라서, 또 때로는 priority 에 따라서 min heap 이 되어야 하기 때문이다. 그래서 minHeapifyOnJobLength(), enheapOnJobLength(), deheapOnJobLength(), minHeapifyOnPriority(), enheapOnPriority(), 그리고 deheapOnPriority() 가 만들어졌다.

그리고 스케줄링 알고리즘들이 쓰여졌다. 각 알고리즘은 numOfProcs 를 argument 로 받는데, 이는 프로세스 몇 개를 스케줄링 할 것인지를 의미한다. RR() 의 경우 추가적으로 timeQuantum 을 argument 로 받는데, 이는 time quantum 을 얼마나 길게 할 것인지를 의미한다. 각 알고리즘들은 그 알고리즘에 맞게 프로세스들을 스케줄링 하며, 거기에 해당하는 Gantt chart 를 그린다. 또한, 작업이 끝난 프로세스들을 finishedQueue 에 넣어서 이후 분석에 사용한다.

파이썬과 유사한 syntax 로 쓰여진 pseudocode 와 flow chart 를 통해 이 알고리즘들을 설명하였다. Pseudocode 와 flow chart 에는 작업이 완료된 프로세스들을 finishedQueue 에 넣어서 이후에 분석한다든지, Gantt chart 를 그린다든지 하는 세부 사항은 쓰여있지 않다. 이러한 세부사항을 보기 위해서는 보고서 뒷부분에 있는 code 를 참고할 수 있다.

```
def FCFS(numOfProcs):
    # initialization
    waitingQueue = new Queue()
    running = new Process()
    timePassed = 0
    waitingQueue.enqueue(newRandomProc())
    numOfProcs -= 1 # the first process
    while !waitingQueue.isEmpty() && numOfProcs > 0:
        if numOfProcs > 0:
            waitingQueue.enqueue(newRandomProc())
            numOfProcs -= 1
        if running == null:
            running = waitingQueue.dequeue()
        if running.CPUburst <= 0 :
            running = waitingQueue.dequeue()
        if IOInterruptOccurs():
            running.IOburst -= 1
            timePassed += 1
            foreach waitingProcess in waitingQueue:
                waitingProcess.waitingTime += 1
            continue
        running.CPUburst -= 1
        foreach waitingProcess in waitingQueue:
            waitingProcess.waitingTime += 1
        timePassed += 1
    return

def nonPreemptiveSJF(numOfProcs):
    # initialization
    waitingQueue = new MinHeap('l') # min heapify on job length
    running = new Process()
    timePassed = 0
    waitingQueue.enheap(newRandomProc())
    numOfProcs -= 1 # the first process
    while !waitingQueue.isEmpty() && numOfProcs > 0:
        if numOfProcs > 0:
            waitingQueue.enheap(newRandomProc())
            numOfProcs -= 1
        if running == null:
            running = waitingQueue.deheap()
        if running.CPUburst <= 0 :
            running = waitingQueue.deheap()
        if IOInterruptOccurs():
            running.IOburst -= 1
            timePassed += 1
            foreach waitingProcess in waitingQueue:
                waitingProcess.waitingTime += 1
            continue
```

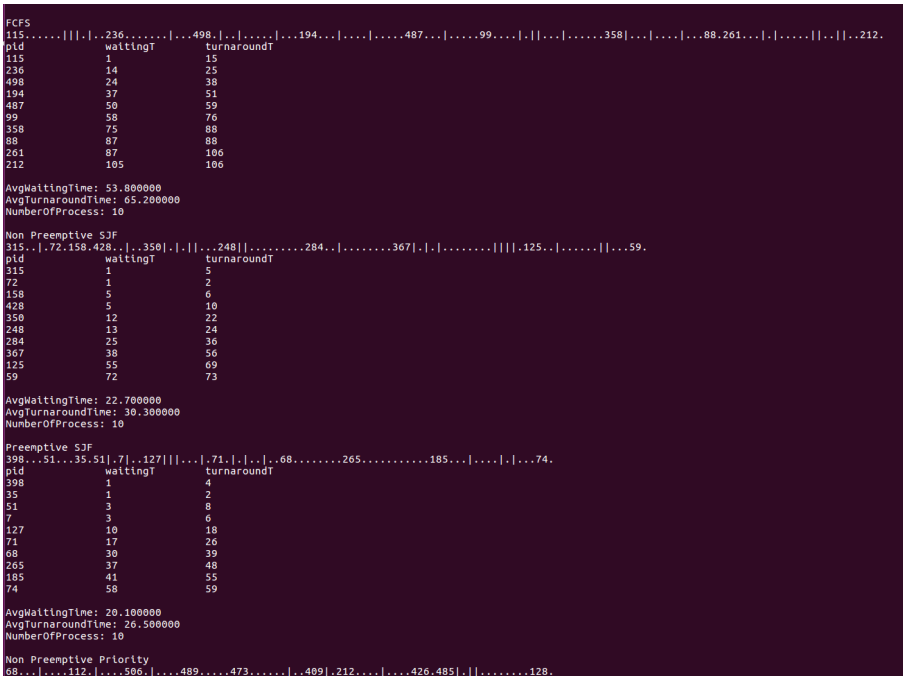
```
        running.CPUburst -= 1
        foreach waitingProcess in waitingQueue:
            waitingProcess.waitingTime += 1
        timePassed += 1
    return

def preemptiveSJF(numOfProcs):
    # initialization
    waitingQueue = new MinHeap('l') # min heapify on job length
    running = new Process()
    timePassed = 0
    waitingQueue.enheap(newRandomProc())
    numOfProcs -= 1 # the first process
    while !waitingQueue.isEmpty() && numOfProcs > 0:
        if numOfProcs > 0:
            waitingQueue.enheap(newRandomProc())
            numOfProcs -= 1
        if running == null:
            running = waitingQueue.deheap()
        if running.CPUburst <= 0 :
            running = waitingQueue.deheap()
        elif running.CPUburst > waitingQueue.arr[0].CPUburst:
            waitingQueue.enheap(running)
            running = waitingQueue.deheap()
            # waitingQueue.arr[0] is the root of the min heap,
            # which is the process with the shortest job length
        if IOInterruptOccurs():
            running.IOburst -= 1
            timePassed += 1
            foreach waitingProcess in waitingQueue:
                waitingProcess.waitingTime += 1
            continue
        running.CPUburst -= 1
        foreach waitingProcess in waitingQueue:
            waitingProcess.waitingTime += 1
        timePassed += 1
    return

def nonPreemptivePriority(numOfProcs):
    # initialization
    waitingQueue = new MinHeap('p') # min heapify on priority
    running = new Process()
    timePassed = 0
    waitingQueue.enheap(newRandomProc())
    numOfProcs -= 1 # the first process
    while !waitingQueue.isEmpty() && numOfProcs > 0:
        if numOfProcs > 0:
            waitingQueue.enheap(newRandomProc())
            numOfProcs -= 1
```

```
        if running == null:
            running = waitingQueue.deheap()
        if running.CPUburst <= 0 :
            running = waitingQueue.deheap()
        if IOInterruptOccurs():
            running.IOburst -= 1
            timePassed += 1
            foreach waitingProcess in waitingQueue:
                waitingProcess.waitingTime += 1
            continue
        running.CPUburst -= 1
        foreach waitingProcess in waitingQueue:
            waitingProcess.waitingTime += 1
        timePassed += 1
    return

def preemptivePriority(numOfProcs):
    # initialization
    waitingQueue = new MinHeap('p') # min heapify on priority
    running = new Process()
    timePassed = 0
    waitingQueue.enheap(newRandomProc())
    numOfProcs -= 1 # the first process
    while !waitingQueue.isEmpty() && numOfProcs > 0:
        if numOfProcs > 0:
            waitingQueue.enheap(newRandomProc())
            numOfProcs -= 1
        if running == null:
            running = waitingQueue.deheap()
        if running.CPUburst <= 0 :
            running = waitingQueue.deheap()
        elif running.priority > waitingQueue.arr[0].priority:
            waitingQueue.enheap(running)
            running = waitingQueue.deheap()
            # lower number for priority means higher priority
            # waitingQueue.arr[0] is the root of the min heap,
            # which is the process with the highest priority
        if IOInterruptOccurs():
            running.IOburst -= 1
            timePassed += 1
            foreach waitingProcess in waitingQueue:
                waitingProcess.waitingTime += 1
            continue
        running.CPUburst -= 1
        foreach waitingProcess in waitingQueue:
            waitingProcess.waitingTime += 1
        timePassed += 1
    return
```

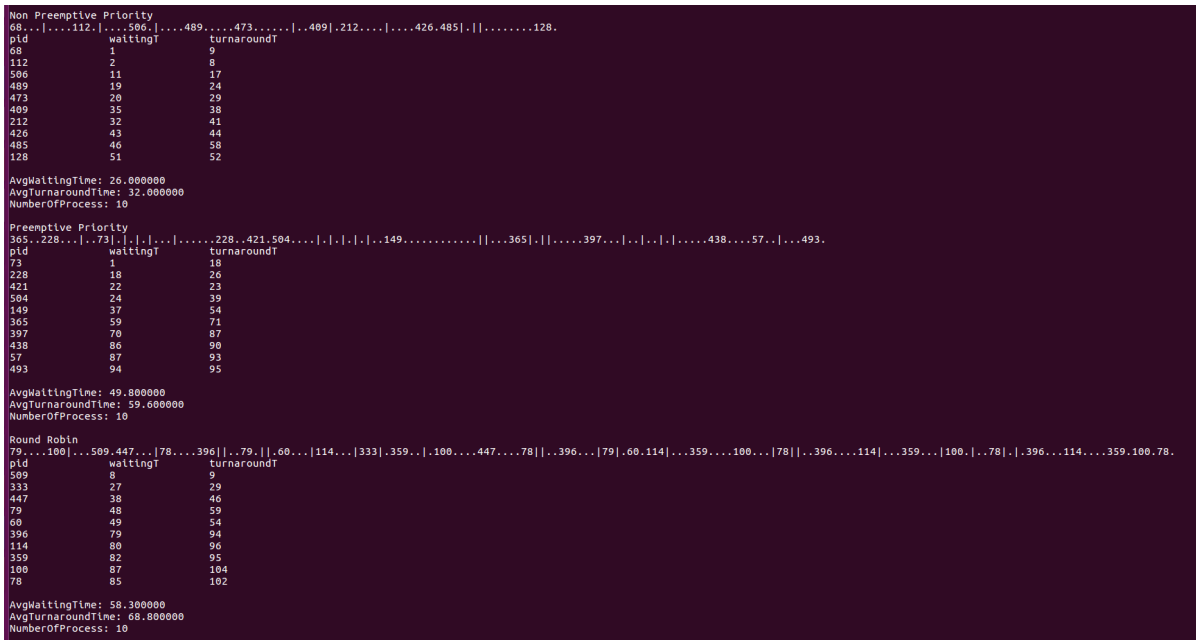
```
def RR(numOfProcs, timeQuantum):
    # initialization
    waitingQueue = new Queue()
    running = new Process()
    timePassed = 0
    currentTimeQuantum = 0
    waitingQueue.enqueue(newRandomProc())
    numOfProcs -= 1 # the first process
    while !waitingQueue.isEmpty() && numOfProcs > 0:
        if numOfProcs > 0:
            waitingQueue.enqueue(newRandomProc())
            numOfProcs -= 1
        if running == null:
            running = waitingQueue.dequeue()
        if running.CPUburst <= 0 :
            running = waitingQueue.dequeue()
        if currentTimeQuantum >= timeQuantum:
            waitingQueue.enqueue(running)
            running = waiting.dequeue()
            currentTimeQuantum = 0
        if IOInterruptOccurs():
            running.IOburst -= 1
            timePassed += 1
            currentTimeQuantum += 1
            foreach waitingProcess in waitingQueue:
                waitingProcess.waitingTime += 1
            continue
        running.CPUburst -= 1
        foreach waitingProcess in waitingQueue:
            waitingProcess.waitingTime += 1
        timePassed += 1
        currentTimeQuantum += 1
    return
```

실행 화면

10 개의 프로세스들로 스케줄러를 run 하면 다음과 같은 실행 화면이 나온다.



```
FCFS
115......|||.|..236.......|...498.|..|.....|...194...|....|.....487...|.....99....|.||...|......358|...|....|...88.261...|.|.....||..||..212.
pid             waitingT        turnaroundT
115             1               15
236             14              25
498             24              38
194             37              51
487             50              59
99              58              76
358             75              88
88              87              88
261             87              106
212             105             106

AvgWaitingTime: 53.800000
AvgTurnaroundTime: 65.200000
NumberOfProcess: 10

Non Preemptive SJF
315..|.72.158.428..|..350|.|.||...248||.........284..|........367|.|.|........||||.125..|......||...59.
pid             waitingT        turnaroundT
315             1               5
72              1               2
158             5               6
428             5               10
350             12              22
248             13              24
284             25              36
367             38              56
125             55              69
59              72              73

AvgWaitingTime: 22.700000
AvgTurnaroundTime: 30.300000
NumberOfProcess: 10

Preemptive SJF
398...51...35.51|.7|..127|||...|.71.|.|..|..68........265...........185...|....|.|...74.
pid             waitingT        turnaroundT
398             1               4
35              1               2
51              3               8
7               3               6
127             10              18
71              17              26
68              30              39
265             37              48
185             41              55
74              58              59

AvgWaitingTime: 20.100000
AvgTurnaroundTime: 26.500000
NumberOfProcess: 10

Non Preemptive Priority
68...|....112.|....506.|....489.....473......|..409|.212....|....426.485|.||........128.
```



```
Non Preemptive Priority
68...|....112.|....506.|....489.....473......|..409|.212....|....426.485|.||........128.
pid             waitingT        turnaroundT
68              1               9
112             2               8
506             11              17
489             19              24
473             20              29
409             35              38
212             32              41
426             43              44
485             46              58
128             51              52

AvgWaitingTime: 26.000000
AvgTurnaroundTime: 32.000000
NumberOfProcess: 10

Preemptive Priority
365..228...|..73|.|.|.|...|......228..421.504....|.|.|.|.|..149............||...365|.||.....397...|..|..|.|.....438....57..|...493.
pid             waitingT        turnaroundT
73              1               18
228             18              26
421             22              23
504             24              39
149             37              54
365             59              71
397             70              87
438             86              90
57              87              93
493             94              95

AvgWaitingTime: 49.800000
AvgTurnaroundTime: 59.600000
NumberOfProcess: 10

Round Robin
79....100|...509.447...|78....396||..79.||.60...|114...|333|.359..|.100....447....78||..396...|79|.60.114|...359....100...|78||..396....114|...359...|100.|..78|.|.396...114....359.100.78.
pid             waitingT        turnaroundT
509             8               9
333             27              29
447             38              46
79              48              59
60              49              54
396             79              94
114             80              96
359             82              95
100             87              104
78              85              102

AvgWaitingTime: 58.300000
AvgTurnaroundTime: 68.800000
NumberOfProcess: 10
```

간트 차트에서 숫자는 각 프로세스의 pid, ‘.’ 은 CPU burst, ‘|’ 는 IO burst 를 나타낸다. 각 프로세스의 waiting time 과 turnaround time 또한 출력되며, 전체적인 평균 waiting time 과 turnaround time 도 출력된다.

보다 많은 수의 프로세스들로 심층적인 비교 분석을 하기에 앞서 앞의 10 개의 프로세스들로 run 한 결과로 각 스케줄링 알고리즘들을 waiting time 과 turnaround time 을 기준으로 나열하면 다음과 같다.

Order by average waiting time

1. preemptiveSJF, 20.1
2. nonPreemptiveSJF, 22.7
3. nonPreemptivePriority, 26.0
4. preemptivePriority, 49.8
5. FCFS, 53.8
6. RR, 58.3

Order by average turnaround time
preemptiveSJF, 26.5
nonPreemptiveSJF, 30.3
nonPreemptivePriority, 32.0
preemptivePriority, 59.6
FCFS, 65.2
RR, 68.1

위의 결과에서 preemptiveSJF 와 nonPreemptiveSJF 가 waiting time 과 turnaround time 을 최소화하는 데에는 유리하다는 것을 알 수 있다.

보다 심층적인 분석을 위하여 이번에는 100 개, 200 개, 300 개의 프로세스들로 run 해보았다. 자리를 아끼기 위해 개별 프로세스들의 waiting time, turnaround time 과 Gantt chart 는 출력하지 않고, 평균 waiting time 과 turnaround time 만 출력했다. 출력 결과는 다음과 같다.

Results from 100 processes

FCFS

AvgWaitingTime: 427.870000
AvgTurnaroundTime: 437.310000
NumberOfProcess: 100

Non Preemptive SJF

AvgWaitingTime: 243.330000
AvgTurnaroundTime: 252.270000
NumberOfProcess: 100

Preemptive SJF

AvgWaitingTime: 248.580000
AvgTurnaroundTime: 257.440000
NumberOfProcess: 100

Non Preemptive Priority

AvgWaitingTime: 437.520000
AvgTurnaroundTime: 447.340000
NumberOfProcess: 100

Preemptive Priority

AvgWaitingTime: 416.190000
AvgTurnaroundTime: 425.820000
NumberOfProcess: 100

Round Robin

AvgWaitingTime: 443.900000
AvgTurnaroundTime: 452.590000
NumberOfProcess: 100

Results from 200 processes

FCFS

AvgWaitingTime: 850.215000
AvgTurnaroundTime: 859.675000
NumberOfProcess: 200

Non Preemptive SJF

AvgWaitingTime: 546.990000
AvgTurnaroundTime: 556.295000
NumberOfProcess: 200

Preemptive SJF

AvgWaitingTime: 557.825000
AvgTurnaroundTime: 567.615000
NumberOfProcess: 200

Non Preemptive Priority

AvgWaitingTime: 810.470000
AvgTurnaroundTime: 819.750000
NumberOfProcess: 200

Preemptive Priority

AvgWaitingTime: 775.380000
AvgTurnaroundTime: 784.245000
NumberOfProcess: 200

Round Robin

AvgWaitingTime: 1015.565000
AvgTurnaroundTime: 1113.545000
NumberOfProcess: 200

Results from 300 processes

FCFS

AvgWaitingTime: 1299.466667
AvgTurnaroundTime: 1309.313333
NumberOfProcess: 300

Non Preemptive SJF

AvgWaitingTime: 763.236667
AvgTurnaroundTime: 772.486667
NumberOfProcess: 300

Preemptive SJF

AvgWaitingTime: 806.416667
AvgTurnaroundTime: 815.690000
NumberOfProcess: 300

Non Preemptive Priority

AvgWaitingTime: 1307.563333
AvgTurnaroundTime: 1317.263333
NumberOfProcess: 300

Preemptive Priority

AvgWaitingTime: 1106.086667
AvgTurnaroundTime: 1114.640000

NumberOfProcess: 300

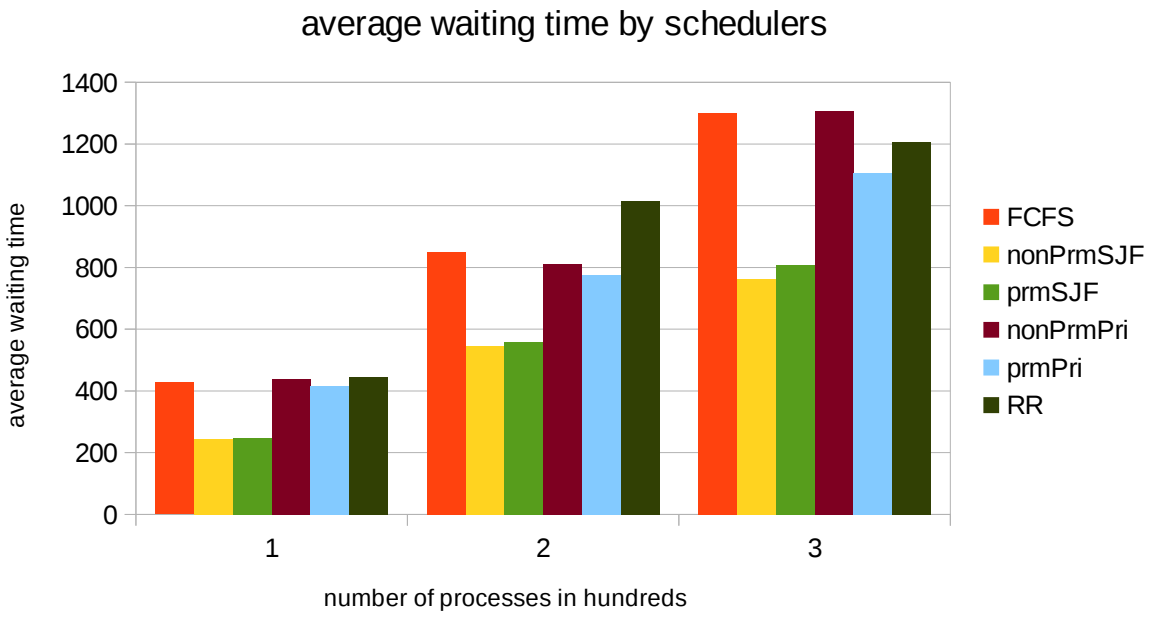
Round Robin

AvgWaitingTime: 1206.226667
AvgTurnaroundTime: 1801.400000
NumberOfProcess: 300

이를 표와 차트로 나타내보면 다음과 같다.

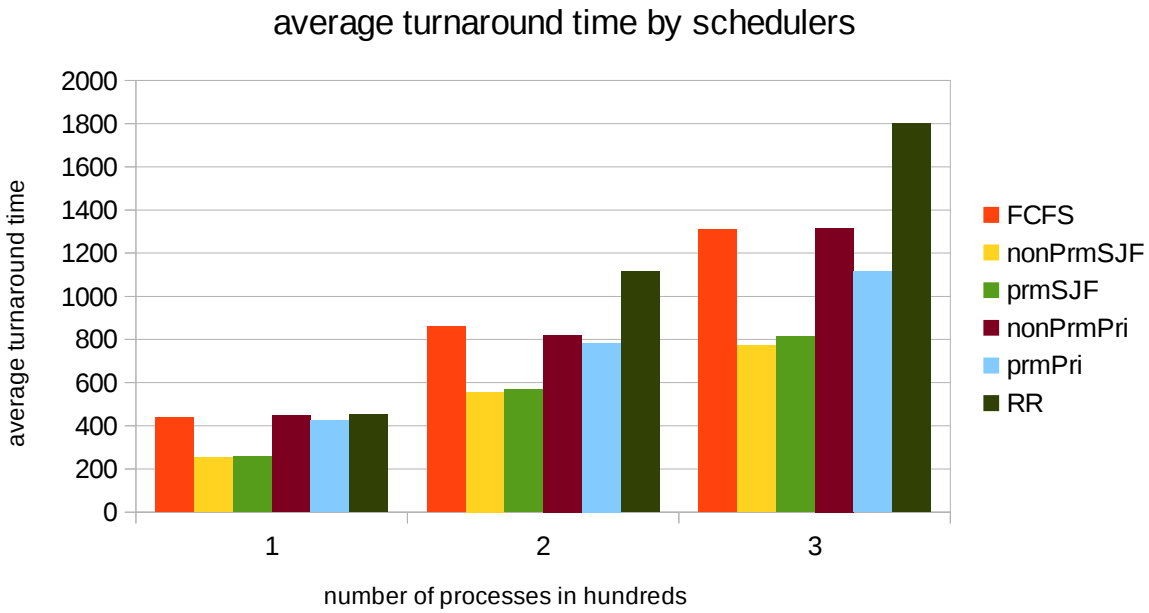
Average waiting time by schedulers

	FCFS	nonPrmSJF	prmSJF	nonPrmPri	prmPri	RR
100	427.87	243.33	248.58	437.52	416.19	443.9
200	850.215	546.99	557.825	810.47	775.38	1015.565
300	1299.466667	763.23667	806.41667	1307.5633	1106.0867	1206.2267



Average turnaround time by schedulers

	FCFS	nonPrmSJF	prmSJF	nonPrmPri	prmPri	RR
100	437.31	252.27	257.44	447.34	425.82	452.59
200	859.675	556.295	567.615	819.75	784.245	1113.545
300	1309.313333	772.486667	815.69	1317.263333	1114.64	1801.4



이번에도 non preemptive SJF 와 preemptive SJF 스케줄러들이 waiting time 과 turnaround time 면에서 가장 성능이 좋았다.

하지만, SJF 는 실제 상황에서 만들기 꽤 힘든 스케줄러이다. 왜냐하면 각 프로세스들의 CPU burst 시간을 정확하게 예측해야 하기 때문이다. 또한, 이 시뮬레이션은 우선순위가 존중되었는지, 데드라인이 지켜졌는지, 리소스가 공평하게 분배되었는지 등을 나타내지는 않는다. 따라서, SJF 계열 스케줄러들이 waiting time 과 turnaround time 면에서 가장 성능이 좋다고 이야기할 수는 있지만, 이들이 가장 좋은 스케줄러라고 하기에는 힘들다.

결론

이 시뮬레이터는 여러 가지 스케줄링 알고리즘을 시뮬레이트 하고 각자 해당하는 waiting time 과 turnaround time 을 출력해준다. 이를 통해 여러 가지 스케줄링 알고리즘들의 성능을 알아볼 수 있고, 비교 분석해볼 수 있다.

이 시뮬레이터를 더욱 발전시키기 위해서는 우선순위 존중이나 공평성, 데드라인 등의 feature 을 추가하면 좋을 것이다. 또한, Gantt chart 를 더욱 그래픽 라이브러리를 사용하여 그리는 것도 좋은 방향이 될 것 같다.

이 다음부터는 코드이다.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
/* ------------------------------------------------------------------------------------------------------------------
*/
struct ProcStruct {
    int pid;
    int CPUburst;
    int IOburst;
    int priority;
    int arrival;
    int finishedTime;
    int turnaroundTime;
    int waitingTime;
};
typedef struct ProcStruct Process;
Process *processAlloc(int a, int b, int c, int d, int e){
    Process *p;
    p = (Process*)malloc(sizeof(Process));
    p->pid = a;
    p->CPUburst = b;
    p->IOburst = c;
    p->priority = d;
    p->arrival = e;
    p->finishedTime = 0;
    p->turnaroundTime = 0;
    p->waitingTime = 1;
    return p;
}
Process *randomProcessAlloc(int n){
    Process *p;
    p = processAlloc(rand()%512, rand()%16, rand()%8+8, rand()%101 - 50, n);
    return p;
}
/* ------------------------------------------------------------------------------------------------------------------
*/

struct NodeArray {
    Process *arr[512];
    int head;
    int tail;
};
typedef struct NodeArray Queue;
Queue *queueAlloc(){
    Queue *q;
    q = (Queue*)malloc(sizeof(Queue));
    q->head = 0;
    q->tail = 0;
    return q;
```

```
}
void enqueue(Queue *q, Process *p){
    if((q->tail+1)%512 == q->head){
        fprintf(stdout, "The queue is full\n");
        return;
    }
    q->arr[(q->tail)] = p;
    q->tail = (q->tail + 1)%512;
    return;
}
Process *dequeue(Queue *q){
    if(q->head == q->tail){
        fprintf(stdout, "The queue is empty\n");
        return NULL;
    }
    Process *result;
    result = q->arr[q->head];
    q->head = (q->head+1)%512;
    return result;
}
int isEmpty(Queue *q){
    if(q->head == q->tail){
        return 1;
    } else {
        return 0;
    }
}
int isFull(Queue *q){
    if((q->tail+1)%512 == q->head){
        return 1;
    } else {
        return 0;
    }
}
/* ----------------------------------------------------------------------------------------------------------------------
*/
typedef struct NodeArray Heap;
Heap *heapAlloc(){
    Heap *h;
    h = (Heap*)malloc(sizeof(Heap));
    h->head = 0;
    h->tail = 0;
}
int isEmptyHeap(Heap *h){
    if(h->tail == 0){
        return 1;
    } else {
        return 0;
    }
```

```
}
void minHeapifyOnJobLength(Heap *h, int i){
    int left = i*2;
    int right = i*2+1;
    int min = i;
    if(left<h->tail && h->arr[left]->CPUburst < h->arr[min]->CPUburst){
        min = left;
    }
    if(right<h->tail && h->arr[right]->CPUburst < h->arr[min]->CPUburst){
        min = right;
    }
    if(min!=i){
        Process *temp;
        temp = h->arr[i];
        h->arr[i] = h->arr[min];
        h->arr[min] = temp;
        minHeapifyOnJobLength(h, min);
    }
    return;
}
void enheapOnJobLength(Heap *h, Process *p){
    h->arr[h->tail] = p;
    int i = h->tail;
    h->tail += 1;
    Process *temp;
    while(i>0 && h->arr[i]->CPUburst < h->arr[i/2]->CPUburst){
        temp = h->arr[i];
        h->arr[i] = h->arr[i/2];
        h->arr[i/2] = temp;
        i = i/2;
    }
    return;
}
Process *deheapOnJobLength(Heap *h){
    Process *result;
    result = h->arr[0];
    h->arr[0] = h->arr[h->tail-1];
    h->arr[h->tail-1] = NULL;
    h->tail -= 1;
    minHeapifyOnJobLength(h, 0);
    return result;
}
void minHeapifyOnPriority(Heap *h, int i){
    int left = i*2;
    int right = i*2+1;
    int min = i;
    if(left<h->tail && h->arr[left]->priority < h->arr[min]->priority){
        min = left;
    }
```

```
    if(right<h->tail && h->arr[right]->priority < h->arr[min]->priority){
        min = right;
    }
    if(min!=i){
        Process *temp;
        temp = h->arr[i];
        h->arr[i] = h->arr[min];
        h->arr[min] = temp;
        minHeapifyOnPriority(h, min);
    }
    return;
}
void enheapOnPriority(Heap *h, Process *p){
    h->arr[h->tail] = p;
    int i = h->tail;
    h->tail += 1;
    Process *temp;
    while(i>0 && h->arr[i]->priority < h->arr[i/2]->priority){
        temp = h->arr[i];
        h->arr[i] = h->arr[i/2];
        h->arr[i/2] = temp;
        i = i/2;
    }
    return;
}
Process *deheapOnPriority(Heap *h){
    Process *result;
    result = h->arr[0];
    h->arr[0] = h->arr[h->tail-1];
    h->arr[h->tail-1] = NULL;
    h->tail -= 1;
    minHeapifyOnPriority(h, 0);
    return result;
}
/* ----------------------------------------------------------------------------------------------------------------------
*/
void FCFS(int numOfProcs){
    Queue *waiting;
    waiting = queueAlloc();
    Queue *finished;
    finished = queueAlloc();
    int i,j;
    int timePassed = 0;
    Process *running;
    running = NULL;
    enqueue(waiting, randomProcessAlloc(0));
    numOfProcs -= 1;
    while(!isEmpty(waiting) || numOfProcs > 0){
        if(numOfProcs>0){
```

```
            enqueue(waiting, randomProcessAlloc(timePassed +1));
            numOfProcs -= 1;
        }
        if(running == NULL){
            running = dequeue(waiting);
            fprintf(stdout, "%d", running->pid);
        }
        if(running->CPUburst <= 0){
            running->finishedTime = timePassed + 1;
            enqueue(finished, running);
            running = dequeue(waiting);
            fprintf(stdout, "%d", running->pid);
        }
        if(rand()%100<20 && running->IOburst > 0){
            running->IOburst -= 1;
            timePassed += 1;
            fprintf(stdout, "|");
            for(i=waiting->head;i<waiting->tail;i++){
                waiting->arr[i]->waitingTime += 1;
            }
            continue;
        }
        running->CPUburst -= 1;
        for(i=waiting->head;i<waiting->tail;i++){
            waiting->arr[i]->waitingTime += 1;
        }
        timePassed += 1;
        fprintf(stdout, ".");
    }
    running->finishedTime = timePassed + 1;
    enqueue(finished, running);
    fprintf(stdout, "\n");
    fprintf(stdout, "pid\t\twaitingT\tturnaroundT\n");
    double avgWaitingT = 0;
    double avgTurnaroudT = 0;
    int num = 0;
    while(!isEmpty(finished)){
        running = dequeue(finished);
        avgWaitingT += running->waitingTime;
        avgTurnaroudT += (running->finishedTime - running->arrival + 1);
        num += 1;
        fprintf(stdout, "%d\t\t%d\t\t%d\n", running->pid, running->waitingTime, running->finishedTime -
running->arrival + 1);
    }
    fprintf(stdout, "\n");
    avgWaitingT /= num;
    avgTurnaroudT /= num;
    fprintf(stdout, "AvgWaitingTime: %f\nAvgTurnaroundTime: %f\nNumberOfProcess: %d\n\n",
avgWaitingT, avgTurnaroudT, num);
```

```
    return;
}
void nonPreemptiveSJF(int numOfProcs){
    Heap *waiting;
    waiting = heapAlloc();
    Queue *finished;
    finished = queueAlloc();
    int timePassed = 0;
    Process *running;
    running = NULL;
    int i, j;
    enheapOnJobLength(waiting, randomProcessAlloc(0));
    numOfProcs -= 1;
    while(!isEmptyHeap(waiting) || numOfProcs > 0){
        if(numOfProcs>0){
            enheapOnJobLength(waiting, randomProcessAlloc(timePassed+1));
            numOfProcs -= 1;
        }
        if(running == NULL){
            running = deheapOnJobLength(waiting);
            fprintf(stdout, "%d", running->pid);
        }
        if(running->CPUburst <= 0){
            running->finishedTime = timePassed + 1;
            enqueue(finished, running);
            running = deheapOnJobLength(waiting);
            fprintf(stdout, "%d", running->pid);
        }
        if(rand()%100<20 && running->IOburst > 0){
            running->IOburst -= 1;
            timePassed += 1;
            fprintf(stdout, "|");
            for(i=waiting->head;i<waiting->tail;i++){
                waiting->arr[i]->waitingTime += 1;
            }
            continue;
        }
        running->CPUburst -= 1;
        for(i=waiting->head;i<waiting->tail;i++){
            waiting->arr[i]->waitingTime += 1;
        }
        timePassed += 1;
        fprintf(stdout, ".");
    }
    running->finishedTime = timePassed + 1;
    enqueue(finished, running);
    fprintf(stdout, "\n");
    fprintf(stdout, "pid\t\twaitingT\tturnaroundT\n");
    double avgWaitingT = 0;
```

```
    double avgTurnaroudT = 0;
    int num = 0;
    while(!isEmpty(finished)){
        running = dequeue(finished);
        avgWaitingT += running->waitingTime;
        avgTurnaroudT += (running->finishedTime - running->arrival + 1);
        num += 1;
        fprintf(stdout, "%d\t\t%d\t\t%d\n", running->pid, running->waitingTime, running->finishedTime -
running->arrival + 1);
    }
    fprintf(stdout, "\n");
    avgWaitingT /= num;
    avgTurnaroudT /= num;
    fprintf(stdout, "AvgWaitingTime: %f\nAvgTurnaroundTime: %f\nNumberOfProcess: %d\n\n",
avgWaitingT, avgTurnaroudT, num);
    return;
}
void preemptiveSJF(int numOfProcs){
    Heap *waiting;
    waiting = heapAlloc();
    Queue *finished;
    finished = queueAlloc();
    int timePassed;
    Process *running;
    running = NULL;
    enheapOnJobLength(waiting, randomProcessAlloc(0));
    numOfProcs -= 1;
    int i,j;
    while(!isEmptyHeap(waiting) || numOfProcs > 0){
        if(numOfProcs > 0){
            enheapOnJobLength(waiting, randomProcessAlloc(timePassed+1));
            numOfProcs -= 1;
        }
        if(running == NULL){
            running = deheapOnJobLength(waiting);
            fprintf(stdout, "%d", running->pid);
        }
        if(running->CPUburst <= 0){
            running->finishedTime = timePassed + 1;
            enqueue(finished, running);
            running = deheapOnJobLength(waiting);
            fprintf(stdout, "%d", running->pid);
        } else if(running->CPUburst > waiting->arr[0]->CPUburst){
            enheapOnJobLength(waiting, running);
            running = deheapOnJobLength(waiting);
            fprintf(stdout, "%d", running->pid);
        }
        if(rand()%100<20 && running->IOburst > 0){
            running->IOburst -= 1;
```

```
            timePassed += 1;
            fprintf(stdout, "|");
            for(i=waiting->head;i<waiting->tail;i++){
               waiting->arr[i]->waitingTime += 1;
            }
            continue;
         }
         running->CPUburst -= 1;
         for(i=waiting->head;i<waiting->tail;i++){
            waiting->arr[i]->waitingTime += 1;
         }
         timePassed += 1;
         fprintf(stdout, ".");
      }
      running->finishedTime = timePassed + 1;
      enqueue(finished, running);
      fprintf(stdout, "\n");
      fprintf(stdout, "pid\t\twaitingT\tturnaroundT\n");
      double avgWaitingT = 0;
      double avgTurnaroudT = 0;
      int num = 0;
      while(!isEmpty(finished)){
         running = dequeue(finished);
         avgWaitingT += running->waitingTime;
         avgTurnaroudT += (running->finishedTime - running->arrival + 1);
         num += 1;
         fprintf(stdout, "%d\t\t%d\t\t%d\n", running->pid, running->waitingTime, running->finishedTime -
running->arrival + 1);
      }
      fprintf(stdout, "\n");
      avgWaitingT /= num;
      avgTurnaroudT /= num;
      fprintf(stdout, "AvgWaitingTime: %f\nAvgTurnaroundTime: %f\nNumberOfProcess: %d\n\n",
avgWaitingT, avgTurnaroudT, num);
      return;
}
void nonPreemptivePriority(int numOfProcs){
      Heap *waiting;
      waiting = heapAlloc();
      Queue *finished;
      finished = queueAlloc();
      int timePassed = 0;
      Process *running;
      running = NULL;
      int i, j;
      enheapOnPriority(waiting, randomProcessAlloc(0));
      numOfProcs -= 1;
      while(!isEmptyHeap(waiting) || numOfProcs > 0){
         if(numOfProcs>0){
```

```
            enheapOnPriority(waiting, randomProcessAlloc(timePassed + 1));
            numOfProcs -= 1;
        }
        if(running == NULL){
            running = deheapOnPriority(waiting);
            fprintf(stdout, "%d", running->pid);
        }
        if(running->CPUburst <= 0){
            running->finishedTime = timePassed + 1;
            enqueue(finished, running);
            running = deheapOnPriority(waiting);
            fprintf(stdout, "%d", running->pid);
        }
        if(rand()%100<20 && running->IOburst > 0){
            running->IOburst -= 1;
            timePassed += 1;
            fprintf(stdout, "|");
            for(i=waiting->head;i<waiting->tail;i++){
                waiting->arr[i]->waitingTime += 1;
            }
            continue;
        }
        running->CPUburst -= 1;
        for(i=waiting->head;i<waiting->tail;i++){
            waiting->arr[i]->waitingTime += 1;
        }
        timePassed += 1;
        fprintf(stdout, ".");
    }
    running->finishedTime = timePassed + 1;
    enqueue(finished, running);
    fprintf(stdout, "\n");
    fprintf(stdout, "pid\t\twaitingT\tturnaroundT\n");
    double avgWaitingT = 0;
    double avgTurnaroudT = 0;
    int num = 0;
    while(!isEmpty(finished)){
        running = dequeue(finished);
        avgWaitingT += running->waitingTime;
        avgTurnaroudT += (running->finishedTime - running->arrival + 1);
        num += 1;
        fprintf(stdout, "%d\t\t%d\t\t%d\n", running->pid, running->waitingTime, running->finishedTime -
running->arrival + 1);
    }
    fprintf(stdout, "\n");
    avgWaitingT /= num;
    avgTurnaroudT /= num;
    fprintf(stdout, "AvgWaitingTime: %f\nAvgTurnaroundTime: %f\nNumberOfProcess: %d\n\n",
avgWaitingT, avgTurnaroudT, num);
```

```
    return;
}
void preemptivePriority(int numOfProcs){
    Heap *waiting;
    waiting = heapAlloc();
    Queue *finished;
    finished = queueAlloc();
    int timePassed;
    Process *running;
    running = NULL;
    enheapOnPriority(waiting, randomProcessAlloc(0));
    numOfProcs -= 1;
    int i,j;
    while(!isEmptyHeap(waiting) || numOfProcs > 0){
        if(numOfProcs > 0){
            enheapOnPriority(waiting, randomProcessAlloc(timePassed + 1));
            numOfProcs -= 1;
        }
        if(running == NULL){
            running = deheapOnPriority(waiting);
            fprintf(stdout, "%d", running->pid);
        }
        if(running->CPUburst <= 0){
            running->finishedTime = timePassed + 1;
            enqueue(finished, running);
            running = deheapOnPriority(waiting);
            fprintf(stdout, "%d", running->pid);
        } else if(running->priority > waiting->arr[0]->priority){
            enheapOnPriority(waiting, running);
            running = deheapOnPriority(waiting);
            fprintf(stdout, "%d", running->pid);
        }
        if(rand()%100<20 && running->IOburst > 0){
            running->IOburst -= 1;
            timePassed += 1;
            fprintf(stdout, "|");
            for(i=waiting->head;i<waiting->tail;i++){
                waiting->arr[i]->waitingTime += 1;
            }
            continue;
        }
        running->CPUburst -= 1;
        for(i=waiting->head;i<waiting->tail;i++){
            waiting->arr[i]->waitingTime += 1;
        }
        timePassed += 1;
        fprintf(stdout, ".");
    }
    running->finishedTime = timePassed + 1;
```

```
    enqueue(finished, running);
    fprintf(stdout, "\n");
    fprintf(stdout, "pid\t\twaitingT\tturnaroundT\n");
    double avgWaitingT = 0;
    double avgTurnaroudT = 0;
    int num = 0;
    while(!isEmpty(finished)){
        running = dequeue(finished);
        avgWaitingT += running->waitingTime;
        avgTurnaroudT += (running->finishedTime - running->arrival + 1);
        num += 1;
        fprintf(stdout, "%d\t\t%d\t\t%d\n", running->pid, running->waitingTime, running->finishedTime -
running->arrival + 1);
    }
    fprintf(stdout, "\n");
    avgWaitingT /= num;
    avgTurnaroudT /= num;
    fprintf(stdout, "AvgWaitingTime: %f\nAvgTurnaroundTime: %f\nNumberOfProcess: %d\n\n",
avgWaitingT, avgTurnaroudT, num);
    return;
}
void RR(int numOfProcs, int timeQuantum){
    Queue *waiting;
    waiting = queueAlloc();
    Queue *finished;
    finished = queueAlloc();
    Process *running;
    running = NULL;
    int i,j;
    int timePassed = 0;
    int currentTimeQuantum = 0;
    enqueue(waiting, randomProcessAlloc(0));
    numOfProcs -= 1;
    while(!isEmpty(waiting) || numOfProcs > 0){
        if(numOfProcs > 0){
            enqueue(waiting, randomProcessAlloc(timePassed + 1));
            numOfProcs -= 1;
        }
        if(running == NULL) {
            running = dequeue(waiting);
            currentTimeQuantum = 0;
            fprintf(stdout, "%d", running->pid);
        }
        if(running->CPUburst <= 0){
            running->finishedTime = timePassed + 1;
            enqueue(finished, running);
            running = dequeue(waiting);
            fprintf(stdout, "%d", running->pid);
            currentTimeQuantum = 0;
```

```
        }
        if(currentTimeQuantum >= timeQuantum){
            enqueue(waiting, running);
            running = dequeue(waiting);
            fprintf(stdout, "%d", running->pid);
            currentTimeQuantum = 0;
        }
        if(rand()%100<20 && running->IOburst > 0 && currentTimeQuantum < timeQuantum){
            running->IOburst -= 1;
            timePassed += 1;
            currentTimeQuantum += 1;
            fprintf(stdout, "|");
            for(i=waiting->head;i<waiting->tail;i++){
                waiting->arr[i]->waitingTime += 1;
            }
            continue;
        }
        running->CPUburst -= 1;
        timePassed += 1;
        currentTimeQuantum += 1;
        for(i=waiting->head;i<waiting->tail;i++){
            waiting->arr[i]->waitingTime += 1;
        }
        fprintf(stdout, ".");
    }
    running->finishedTime = timePassed + 1;
    enqueue(finished, running);
    fprintf(stdout, "\n");
    fprintf(stdout, "pid\t\twaitingT\tturnaroundT\n");
    double avgWaitingT = 0;
    double avgTurnaroudT = 0;
    int num = 0;
    while(!isEmpty(finished)){
        running = dequeue(finished);
        avgWaitingT += running->waitingTime;
        avgTurnaroudT += (running->finishedTime - running->arrival + 1);
        num += 1;
        fprintf(stdout, "%d\t\t%d\t\t%d\n", running->pid, running->waitingTime, running->finishedTime -
running->arrival + 1);
    }
    fprintf(stdout, "\n");
    avgWaitingT /= num;
    avgTurnaroudT /= num;
    fprintf(stdout, "AvgWaitingTime: %f\nAvgTurnaroundTime: %f\nNumberOfProcess: %d\n\n",
avgWaitingT, avgTurnaroudT, num);
    return;
}
/* ------------------------------------------------------------------------------------ */
int main(int argc, char *argv[]){
```

```
    if(argc != 2){
        fprintf(stdout, "Input a number as an argument\n");
        return 0;
    }
    int numOfProcs;
    numOfProcs = atoi(argv[1]);
    fprintf(stdout, "\n\n");
    fprintf(stdout,"FCFS\n");
    FCFS(numOfProcs);
    fprintf(stdout,"Non Preemptive SJF\n");
    nonPreemptiveSJF(numOfProcs);
    fprintf(stdout,"Preemptive SJF\n");
    preemptiveSJF(numOfProcs);
    fprintf(stdout,"Non Preemptive Priority\n");
    nonPreemptivePriority(numOfProcs);
    fprintf(stdout,"Preemptive Priority\n");
    preemptivePriority(numOfProcs);
    fprintf(stdout, "Round Robin\n");
    RR(numOfProcs, 4);
    fprintf(stdout, "\n");
    return 0;

}
```