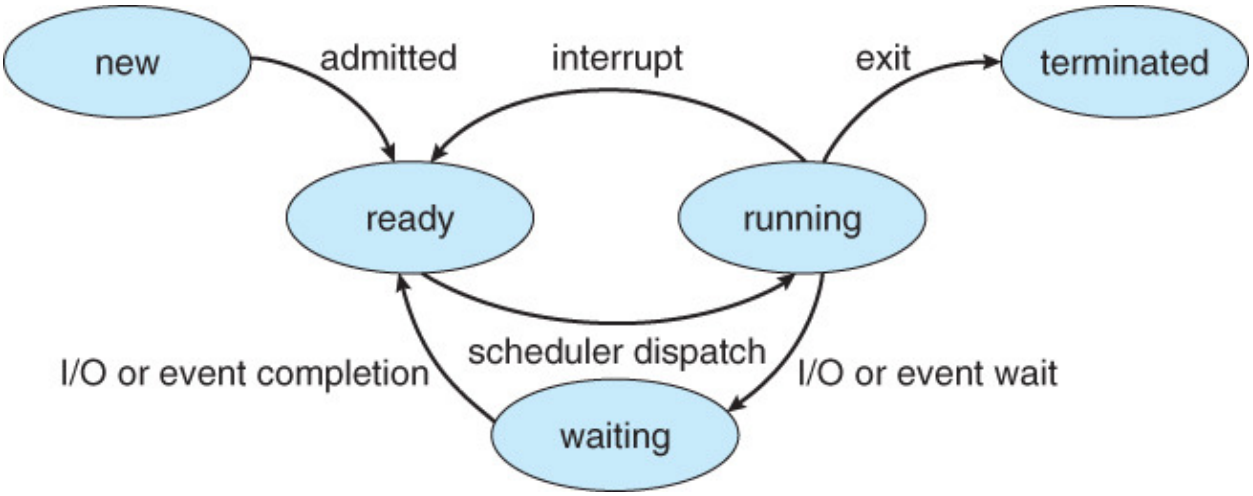
서론

CPU 스케줄링

프로세스(Process)가 구동하려면 다양한 시스템 자원이 필요하다. 대표적으로 CPU(중앙처리장치)와 입출력장치가 있는데, 최고의 성능을 내기 위해 **자원을 어떤 프로세스에 얼마나 할당하는지 정책을 짜는 것**을 CPU스케줄링이라고 한다.



위의 그림은 프로세스의 생명주기를 나타낸다. 이러한 프로세스의 상태 변화 중 다음의 4가지 경우에서 CPU 스케줄링이 일어난다.

1. 수행 → 대기 (I/O or event wait)
2. 수행 → 준비 (interrupt)
3. 대기 → 준비 (I/O or event completion)
4. 수행 → 종료 (exit)

이 중 1, 2, 4의 경우 수행하던 프로세스의 변화를 나타내며 3은 레디 큐의 변화를 나타낸다. 즉, CPU 스케줄링은 **레디 큐에 변화가 있을 때** 혹은 **수행하던 프로세스에 변화가 있을 때** 일어난다.

스케줄링 방식에는 **비선점(Nonpreemptive) 스케줄링**과 **선점(Preemptive) 스케줄링**이 있는데, 비선점 스케줄링의 경우 1, 4의 상황에서만 수행되며 선점 스케줄링의 경우 모든 상황에서 수행된다.

각 스케줄링 기법에 대한 설명과 아래의 5개 프로세스에 대한 스케줄링 결과에 대해 알아보자.

PID	Arrival	Priority	CPUburst	IOburst	Remain	Wait	Turnaround
1	29	2	41	0	41	0	0
2	13	1	20	0	20	0	0
3	22	5	39	0	39	0	0
4	26	3	27	0	27	0	0
5	7	2	29	0	29	0	0

비선점 스케줄링(Nonpreemptive Scheduling)

어떤 프로세스가 CPU를 할당 받으면 그 프로세스가 종료되거나 입출력 요구가 발생하여 자발적으로 중지될 때까지 계속 실행되도록 보장하는 스케줄링 기법이다. 순서대로 처리되기 때문에 공정성이 있고 다음에 처리해야 할 프로세스와 관계없이 응답 시간을 예상할 수 있으며 선점 방식보다 스케줄러 호출 빈도가 낮고 문맥 교환에 의한 오버헤드가 적다. 일괄 처리 시스템에 적합하며, CPU 사용 시간이 긴 하나의 프로세스가 CPU 사용 시간이 짧은 여러 프로세스를 오랫동안 대기시킬 수 있으므로, 처리율이 떨어질 수 있다는 단점이 있다.

1. FCFS 스케줄링(First-Come First-Served Scheduling)

CPU를 처음 요구한 프로세스가 CPU를 처음 사용한다는 의미의 스케줄링 알고리즘이다. FCFS는 스케줄링 특성상 비선점 스케줄링 알고리즘이기 때문에 프로세스의 실행 순서에 따라 평균 대기시간의 차이가 크다. 이를 호위 효과라고 하는데, 호위 효과는 CPU를 매우 오래 사용하는 프로세스가 도착하게 되면, 다른 프로세스가 CPU를 사용하는데 기다리는 대기 시간이 매우 커지는 현상을 말한다.

- 장점 : 모든 프로세스가 공평한 조건으로 CPU를 사용할 수 있게 해주며, 반응 시간을 예측할 수 있다.
- 단점 : 호위 효과로 인해 긴 작업을 수행 하느라 여러 개의 짧은 일이 대기하는 비효율적인 상태가 될 수 있다.

PID	Arrival	Priority	CPUburst	IOburst	Remain	Wait	Turnaround
5	7	2	29	0	0	0	29
2	13	1	20	0	0	23	43
3	22	5	39	0	0	34	73
4	26	3	27	0	0	69	96
1	29	2	41	0	0	93	134

average waiting time : 43.80 **average turnaround time** : 75.00

FCFS 스케줄링의 성능은 프로세스의 도착 순서에 따라 달라진다. 아래의 예시를 통해 대기 시간과 반환 시간이 어떻게 달라지는지 알아보자.

1) 긴 작업이 먼저 도착하는 경우

PID	burst time	waiting time	turnaround time
1	24	0	24
2	3	24	27
3	3	27	30
평균	-	17	27

2) 짧은 작업이 먼저 도착하는 경우

PID	burst time	waiting time	turnaround time
2	3	0	3
3	3	3	6
1	24	6	30
평균	-	3	13

이처럼 FCFS 스케줄링의 경우 프로세스의 실행 순서에 따라 성능이 크게 달라질 수 있다.

2. 비선점 SJF 스케줄링(Nonpreemptive Shortest Job First Scheduling)

평균 대기 시간을 최소화하기 위해 CPU 점유 시간이 가장 짧은 프로세스에 CPU를 먼저 할당하는 방식의 CPU 스케줄링 알고리즘으로 평균 대기시간을 최소로 만드는 것을 최적으로 두고 있는 알고리즘이다. 요구 시간이 긴 프로세스가 요구 시간이 짧은 프로세스에게 항상 양보되어 기아 상태가 발생할 수 있으며, 대기 상태에 있는 프로세스의 요구 시간에 대한 정확한 자료를 얻기 어렵다는 문제점이 있다. 단기 스케줄링 보다는 장기 스케줄링에 유리하다. 기아 상태를 해결하기 위해서는 에이징(aging) 기법을 사용해야하는데, 이는 낮은 우선순위의 프로세스들의 무한 대기 문제에 대한 해결책으로 오랫동안 시스템에서 대기하는 프로세스들의 우선순위를 점진적으로 증가시키는 방법이다.

- 장점 : 평균 대기시간을 최소화할 수 있다.
- 단점 : 기아 상태가 발생할 수 있다. 모든 프로세스의 CPU 점유 시간을 알기 어렵기 때문에 구현하기 힘들다.

PID	Arrival	Priority	CPUburst	IOburst	Remain	Wait	Turnaround
5	7	2	29	0	0	0	29
2	13	1	20	0	0	23	43
4	26	3	27	0	0	30	57
3	22	5	39	0	0	61	100
1	29	2	41	0	0	93	134

average waiting time : 41.40 **average turnaround time** : 72.60

3. 비선점 Priority 스케줄링(Nonpreemptive Priority Scheduling)

각 프로세스 마다 가지고 있는 우선순위를 가지고 우선순위가 가장 높은 프로세스에게 CPU를 할당하는 방식의 CPU 스케줄링 알고리즘으로 우선순위가 같은 경우 FCFS로 처리된다. SJF 스케줄링은 우선순위를 CPU 점유시간을 기준으로 정하는 우선순위 알고리즘의 특수한 형태라고 할 수 있다.

- 장점 : 우선순위가 높은 프로세스를 우선적으로 처리할 수 있다.
- 단점 : 기아 상태가 발생하여 우선순위가 낮은 프로세스의 대기 시간이 무한정 늘어날 수 있다.

PID	Arrival	Priority	CPUburst	IOburst	Remain	Wait	Turnaround
5	7	2	29	0	0	0	29
3	22	5	39	0	0	14	53
4	26	3	27	0	0	49	76
1	29	2	41	0	0	73	114
2	13	1	20	0	0	130	150

average waiting time : 53.20 **average turnaround time** : 84.40

선점 스케줄링(Preemptive Scheduling)

어떤 프로세스가 CPU를 할당받아 실행 중에 있어도 다른 프로세스가 실행 중인 프로세스를 중지하고 CPU를 강제로 점유할 수 있는 스케줄링 기법이다. 모든 프로세스에게 CPU 사용 시간을 동일하게 부여할 수 있다. 빠른 응답시간을 요하는 대화식 시분할 시스템에 적합하며 긴급한 프로세서를 제어할 수 있다. '운영 체제가 프로세서 자원을 선점'하고 있다가 각 프로세스의 요청이 있을 때 특정 요건들을 기준으로 자원을 배분하는 방식이다. 선점이 일어날 경우 오버헤드가 발생하며 처리 시간을 예측하기 힘들다.

1. 선점 SJF 스케줄링(Preemptive Shortest Job First Scheduling)

SRTF 스케줄링(Shortest Remaining Time First Scheduling)이라고도 하며, 현재 실행 중인 프로세스보다 남은 시간이 적은 프로세스가 들어오게 되면 실행 중인 프로세스를 중지하고 CPU를 넘겨준다.

PID	Arrival	Priority	CPUburst	IOburst	Remain	Wait	Turnaround
2	13	1	20	0	0	0	20
5	7	2	29	0	0	20	49
4	26	3	27	0	0	30	57
3	22	5	39	0	0	61	100
1	29	2	41	0	0	93	134

average waiting time : 40.80 **average turnaround time** : 72.00

2. 선점 Priority 스케줄링(Preemptive Priority Scheduling)

현재 실행 중인 프로세스보다 우선순위가 높은 프로세스가 들어오게 되면 실행 중인 프로세스를 중지하고 CPU를 넘겨준다.

PID	Arrival	Priority	CPUburst	IOburst	Remain	Wait	Turnaround
3	22	5	39	0	0	0	39
4	26	3	27	0	0	35	62
5	7	2	29	0	0	66	95
1	29	2	41	0	0	73	114
2	13	1	20	0	0	130	150

average waiting time : 60.80 **average turnaround time** : 92.00

3. Round Robin 스케줄링(Round Robin Scheduling)

Round Robin 스케줄링은 각각의 프로세스가 공평하게 시간 간격 동안 CPU를 활용할 수 있도록 하는 스케줄링 알고리즘이다. 시간 할당량(time quantum) 단위로 CPU를 할당하며 알고리즘의 성능은 시간 할당량의 크기에 좌우된다. 시간 할당량이 매우 크면 FCFS 스케줄링과 같아지고, 시간 할당량이 매우 작으면 프로세서 공유와 같아진다. 그러나 시간 할당량이 매우 적은 경우에는 문맥 교환을 위한 오버헤드가 발생하기 때문에 비효율적이다.

(time quantum = 10)

PID	Arrival	Priority	CPUburst	IOburst	Remain	Wait	Turnaround
2	13	1	20	0	0	34	54
5	7	2	29	0	0	50	79
4	26	3	27	0	0	80	107
3	22	5	39	0	0	91	130
1	29	2	41	0	0	93	134

average waiting time : 69.60 **average turnaround time** : 100.80

Round Robin 스케줄링의 성능은 시간 할당량에 따라 달라진다. 아래의 예시를 통해 대기 시간과 반환 시간이 어떻게 달라지는지 알아보자.

1) 시간 할당량이 매우 클 경우 (time quantum = 30)

PID	burst time	waiting time	turnaround time
1	24	0	24
2	12	24	36
3	18	36	54
평균	-	20	38

2) 시간 할당량이 매우 작을 경우 (time quantum = 3)

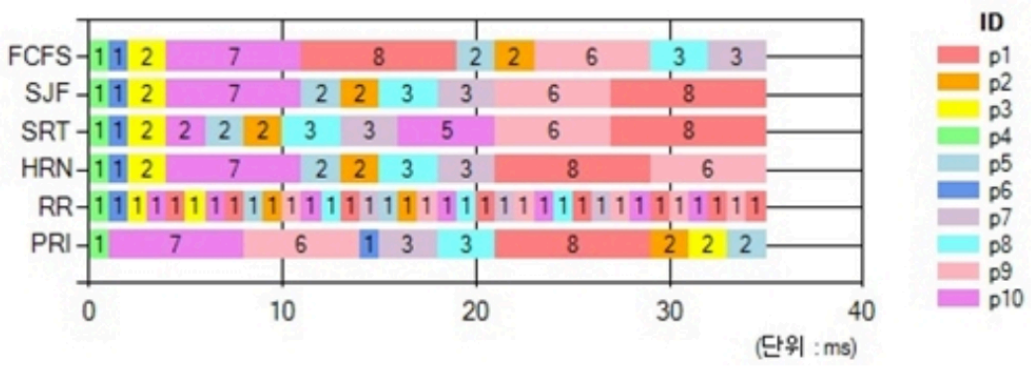
PID	burst time	waiting time	turnaround time
1	24	30	54
2	12	21	33
3	18	30	48
평균	-	27	45

본론

[기존 시뮬레이터] CPUsim2

CPUsim2은 학습자가 다양한 상황을 설정하여 알고리즘의 동작 방식과 실행 결과를 확인하고 이들을 비교함으로써 알고리즘에 대한 올바른 이해를 돕는 웹 기반교육용 시뮬레이터이다.

```
Language :  C#, ASP.NET
Tool : Visual Studio 2008, IIS 7
```



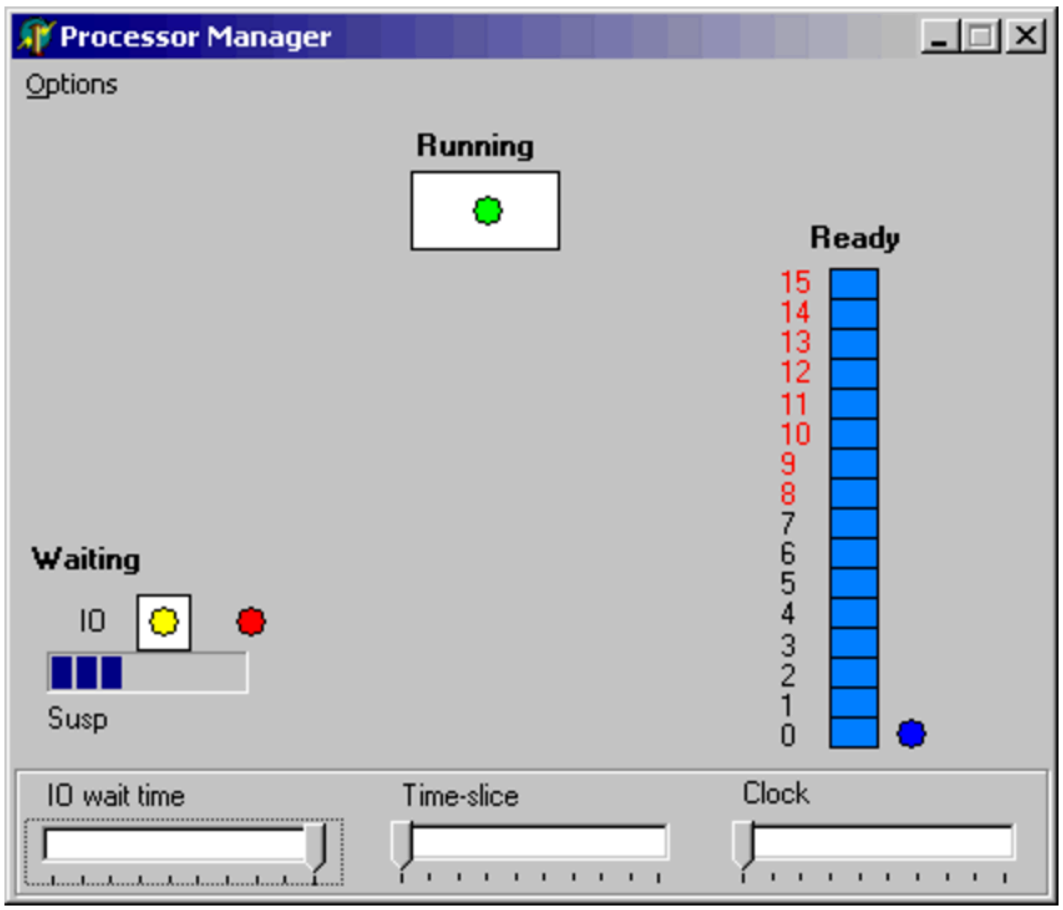
특징

- 프로세스 개수 : 2~10개
- 프로세스 관련 정보 생성 : 수동 입력 혹은 자동 입력

- 알고리즘 종류 : FCFS, HRN, SJF, SRT, Priority, RR
- [RR] CPU 시간(Time quantum) 설정 기능
- 실행 방식 : 일괄 실행 혹은 단계별 실행
- 재설정 가능
- 출력 : 간트 차트로 출력, 해당 알고리즘의 성능 수치(평균 대기시간과 평균 반환시간), 성능 수치가 누적되어 표시(알고리즘 간 성능 비교 가능)

[기존 시뮬레이터] Sosim

Sosim은 다중 프로그래밍, 프로세스 관리, 스케줄링, 메모리 관리 기능을 제공하는 독립 실행형 범용 시뮬레이터이다.

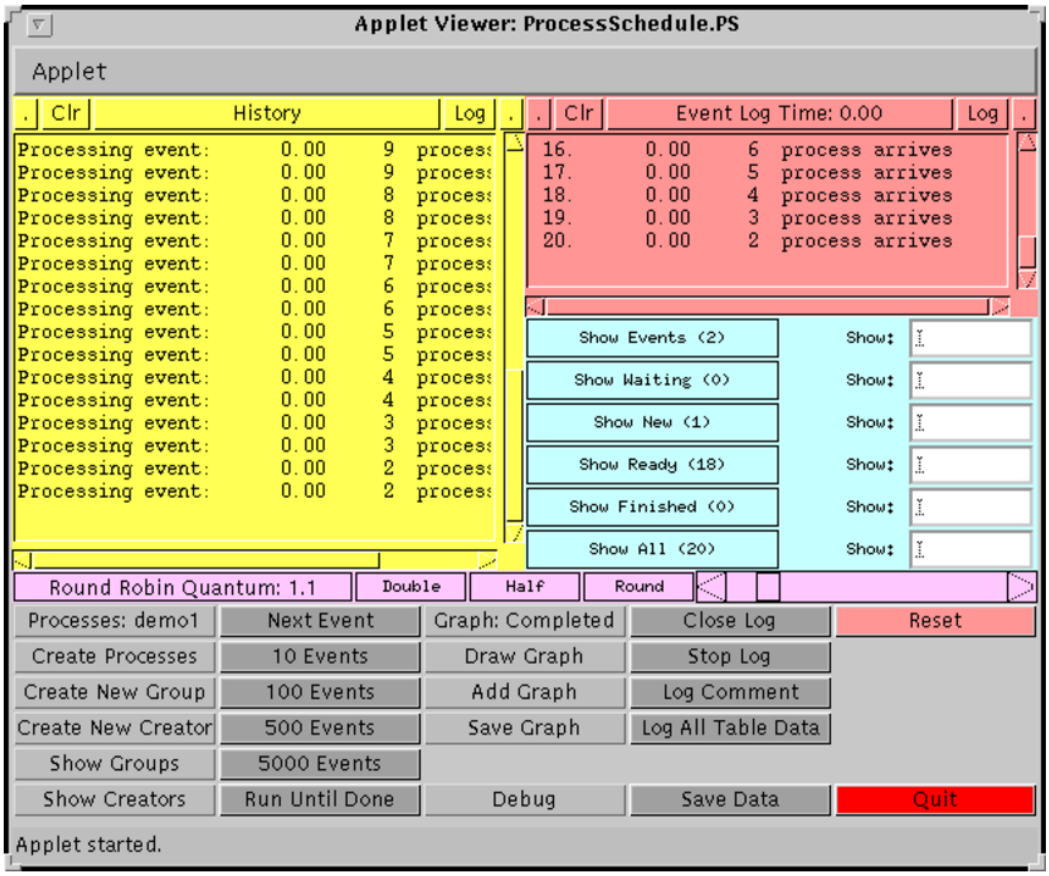


특징

- GUI를 이용한 시각화 (PCB, ready queue 등)
- 알고리즘 종류 : 비선점형, 선점형, RR
- 페이징 기법을 이용한 가상 메모리 관리 제공
    - 로그 저장 가능

[기존 시뮬레이터] Robbins

Robbins는 웹 브라우저와 독립 실행이 가능한 단일용도 시뮬레이터이다.



특징

- 알고리즘 종류 : FCFS, SJF, Preemptive SJF, SJFA, RR, Priority
- 매개변수 설정과 스케줄링 결과 저장 기능 제공
- 결과 위주의 정보 제공

프로젝트 시뮬레이터

구성 모듈

```
void CreateProcess()
```

- 사용자에게 생성할 프로세스 개수(processNum)를 입력받아 프로세스 노드를 생성한다. ProcessID는 0부터 processNum-1까지 차례로 증가하며, 그 외의 CPUburstTime, ArrivalTime, Priority는 랜덤으로 생성된다. CPUburstTime과 ArrivalTime은 1~50의 범위를 가지며 Priority는 1~processNum의 범위를 가진다.

```
void Enqueue(
   PROCESS *node, // 큐에 저장할 노드
   PROCESS head, // 저장할 큐의 head
   PROCESS tail // 저장할 큐의 tail
)
```

- 해당 프로세스큐가 비어있을 경우 head에 node를 저장하고, 비어있지 않을 경우 tail의 뒤에 node를 저장한다. 저장 후에는 tail을 node로 바꾸어준다.

```
void Menu()
```

- 선택 가능한 메뉴를 출력하고 사용자에게 원하는 메뉴를 입력받아 해당 입력값을 인자로 Select를 호출한다.

```
void Select(
  int select // 선택한 메뉴
)
```

- 해당 select 값에 맞는 코드를 실행한다. select 값은 선택한 메뉴를 의미한다. 1~6은 각 스케줄링 알고리즘이 실행되며 7은 Evaluation, 8은 프로그램이 종료된다.

```
void PrintProcess(
  PROCESS **head // 프로세스 정보를 출력할 큐의 head
)
```

- 해당 큐에 존재하는 모든 프로세스 정보를 표 형식으로 출력한다.

```
void PrintEval()
```

- Evaluation이 끝난 후 모든 스케줄링 알고리즘의 평균 대기 시간과 평균 반환 시간을 출력한다. 또한 맨 마지막에는 가장 작은 평균 대기 시간을 가지는 알고리즘과 가장 작은 평균 반환 시간을 가지는 알고리즘을 구해 PrintMin을 호출한다.

```
void PrintMin(
  int result // 가장 작은 평균 시간을 가지는 알고리즘 종류
)
```

- result 값에 따라 알고리즘 종류를 출력한다.

```
void InitEval()
```

- 스케줄링 시뮬레이션을 반복하기 위해 Eval 값들을 초기화한다. 초기화되는 변수는 각 알고리즘의 평균 대기 시간과 평균 반환 시간을 저장하는 변수이다.

```
void SortByArrival(
  PROCESS head, // 정렬할 큐의 head
  PROCESS tail // 정렬할 큐의 tail
)
```

- 프로세스 큐에 존재하는 노드들을 ArrivalTime을 기준으로 정렬한다.

```
void SortByBurst(
  PROCESS head, // 정렬할 큐의 head
  PROCESS tail // 정렬할 큐의 tail
)
```

- 프로세스 큐에 존재하는 노드들을 CPUburstTime을 기준으로 정렬한다.

```
void SortByPriority(
  PROCESS head, // 정렬할 큐의 head
  PROCESS tail // 정렬할 큐의 tail
)
```

- 프로세스 큐에 존재하는 노드들을 Priority를 기준으로 정렬한다.

```
void SortByRemain(
  PROCESS head, // 정렬할 큐의 head
  PROCESS tail // 정렬할 큐의 tail
)
```

- 프로세스 큐에 존재하는 노드들을 RemainTime을 기준으로 정렬한다.

```
void CopyNode(
  PROCESS *node, // 복사할 정보를 가진 노드
  PROCESS *new // 복사한 정보를 저장할 노드
)
```

- 노드의 프로세스 정보를 새로운 노드에 복사한다.

```
void CopyQueue(
  PROCESS queue, // 복사할 큐
  PROCESS **head, // 복사한 큐를 저장할 큐의 head
  PROCESS **tail // 복사한 큐를 저장할 큐의 tail
)
```

- 큐에 들어있는 모든 노드를 CopyNode로 복사한 후 새로운 큐에 Enqueue한다.

```
void FCFS()
```

- FCFS 스케줄링을 실행한다. 간트 차트와 평균 대기 시간, 평균 반환 시간을 출력한다.

```
void NPSJF()
```

- Nonpreemptive SJF 스케줄링을 실행한다. 간트 차트와 평균 대기 시간, 평균 반환 시간을 출력한다.

```
void NPPriority()
```

- Nonpreemptive Priority 스케줄링을 실행한다. 간트 차트와 평균 대기 시간, 평균 반환 시간을 출력한다.

```
void PSJF()
```

- Preemptive SJF 스케줄링을 실행한다. 간트 차트와 평균 대기 시간, 평균 반환 시간을 출력한다.

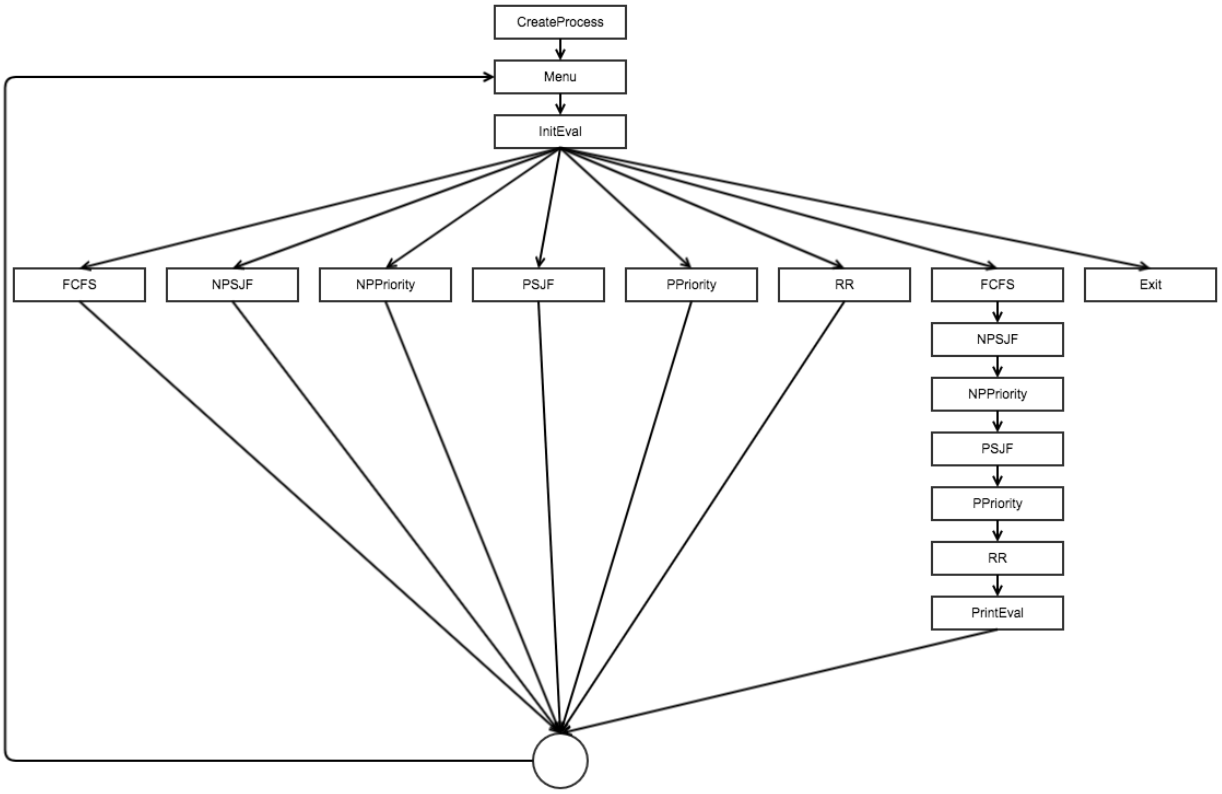
```
void PPriority()
```

- Preemptive Priority 스케줄링을 실행한다. 간트 차트와 평균 대기 시간, 평균 반환 시간을 출력한다.

```
void RR()
```

- Round Robin 스케줄링을 실행한다. 간트 차트와 평균 대기 시간, 평균 반환 시간을 출력한다.

프로그램 흐름도



비선점 스케줄링 알고리즘(FCFS, NPSJF, NPPriority)

비선점 스케줄링은 프로세스들을 도착 시간 순으로 정렬한 후 도착 시간에 도달하면 ready queue에 넣는다. CPU의 상태는 3가지가 존재하는데 각 경우에 따라 다른 동작이 실행된다.

- CPU 상태가 ProcessID일 때 : 간트 차트 출력하고 실행 시간, 잔여 시간 등 계산
- CPU 상태가 SCHEDULE일 때 : IDLE 상태로 변경하고 ready queue에서 해당 프로세스 제거
- CPU 상태가 IDLE일 때 : ready queue에 프로세스가 존재하면 CPU 할당

```
Nonpreemptive Scheduling {

    SortByArrival // 도착 시간 순으로 정렬

    while(endNum < processNum) { // 모든 프로세스가 작업을 마칠 때까지 반복
```

```
      /* Job Scheduling */ { // 도착 시간이 되면 Job Scheduling
        Copy to ReadyQueue
        Remove job scheduled process from process list
      }

      /* if CPUstate is SCHEDULE */ { // 한 프로세스가 작업을 마쳤을 때
        Make CPU state IDLE
        Remove end process from ready queue
      }

      /* CPU Scheduling */ {
        while(ReadyQueue is not empty) {
          if current executing process {
            Print for Gantt chart
            if (remain time is 1) { // 작업이 끝나는 시점
              Calculate turnaround time, average time
              Make CPUstate SCHEDULE
              Increase endNum
            }
            Increase execute time
            Decrease remain time
          }
          if CPU state is IDLE {
            Print whitespace for Gantt chart
            CPUstate = ready node's PID
            if (remain time is 1) { // 작업이 끝나는 시점
              Calculate turnaround time, average time
              Make CPUstate SCHEDULE
              Increase endNum
            }
            Increase execute time
            Decrease remain time
          }
          else { // 대기하고 있는 프로세스
            increase waiting time
          }
        }
      }

      if(readyHead==NULL && CPUstate==IDLE) { // 아무것도 수행되고 있지 않을 때 공백
출력
            Print whitespace for Gantt chart
      }
      Increase time
    }

    Print Average Time // 평균 대기 시간, 평균 반환 시간 출력
}
```

선점 스케줄링 알고리즘(PSJF, PPriority)

선점 스케줄링은 비선점 스케줄링과 동작 방식은 동일하나 매 시간마다 ready queue에 있는 노드들을 기준에 따라 정렬해준다는 점이 다르다.

```
Preemptive Scheduling {

      ...

      /* if CPUstate is SCHEDULE */ { // 한 프로세스가 작업을 마쳤을 때
        Make CPU state IDLE
        Remove end process from ready queue
      }

      Sort ready nodes

      /* CPU Scheduling */ {
        while(ReadyQueue is not empty) {
          if current executing process {
            Print for Gantt chart
            if (remain time is 1) { // 작업이 끝나는 시점
              Calculate turnaround time, average time
              Make CPUstate SCHEDULE
              Increase endNum
            }
            Increase execute time
            Decrease remain time
          }
     ...
}
```

Round Robin 알고리즘

선점 스케줄링은 프로세스가 CPU를 점유하는 시점이 아니라 프로세스가 작업을 끝마치는 시점에서 ready queue에서 제거된다. 이는 선점 스케줄링이 무조건 ready queue의 첫 번째에 존재하는 프로세스를 실행하기 때문에 가능하다. 그러나 Round Robin 스케줄링의 경우 ready queue의 모든 프로세스에 순서대로 CPU가 주어져야 한다. 그래서 Round Robin 스케줄링은 time quantum이 만료되었을 때 ready queue의 head를 다음 프로세스에 넘기고 다시 ready queue의 맨 뒷부분에 넣어주도록 구현했다.

```
/* time expired - move to ready queue */
if(CPUstate > 0) {
  node = (PROCESS *)malloc(sizeof(PROCESS));
  CopyNode(readyHead, node);
  Enqueue(node, &readyHead, &readyTail);
  readyHead = readyHead->Next;
}
```

실행 결과

실행 환경 설정

처음 프로그램을 실행하면 생성할 프로세스 개수를 입력받는다.

```
❯ ./CPUscheduling
+-------------------------------------------------------------------+
|                          ProcessCreation                          |
+-------------------------------------------------------------------+
                  Enter the number of process :
```

개수를 입력하면 입력한 개수만큼 생성된 프로세스들의 정보가 출력된다. (본 실험에서는 10개를 입력했다.)

PID	Arrival	Priority	CPUburst	IOburst	Remain	Wait	Turnaround
1	27	1	6	0	6	0	0
2	46	2	50	0	50	0	0
3	46	9	27	0	27	0	0
4	20	5	5	0	5	0	0
5	30	5	6	0	6	0	0
6	36	10	46	0	46	0	0
7	48	6	7	0	7	0	0
8	39	1	44	0	44	0	0
9	25	2	49	0	49	0	0
10	34	6	36	0	36	0	0

그 후 알고리즘을 선택할 수 있도록 메뉴가 보여진다. Exit를 선택하기 전까지 동일한 프로세스 정보에 대해 반복하여 스케줄링 해볼 수 있다.

```
+-------------------------------------------------------------------+
|                    CPU Scheduling Simulator                       |
+-------------------------------------------------------------------+
|                                                                   |
|   1. FCFS(First Come First Served)                                |
|   2. Nonpreemptive SJF(Shortest Job First)                        |
|   3. Preemptive SJF(Shortest Job First)                           |
|   4. Nonpreemptive Priority                                       |
|   5. Preemptive Priority                                          |
|   6. RR(Round Robin)                                              |
|   7. Evaluation                                                   |
|   8. Exit                                                         |
|                                                                   |
+-------------------------------------------------------------------+

                  Select CPU Scheduling Algorithm :
```

각 알고리즘을 선택하면 해당 알고리즘을 이용한 스케줄링이 실행되고 간트 차트와 스케줄링 결과가 출력된다. 출력되는 프로세스들은 작업이 끝난 순서대로 정렬되어 있다.

1. FCFS

```
+--------------------------------------------------------------+
|                     FCFS Scheduling Report                   |
+--------------------------------------------------------------+
************************* Gantt Chart ***************************

                    4----9---------------------------------------
-------1-----5-----10-----------------------------------6----------
-----------------------------------8-----------------------------
-----------2-------------------------------------------------3----
----------------------7------

**************************** Result *****************************
```

PID	Arrival	Priority	CPUburst	IOburst	Remain	Wait	Turnaround
4	20	5	5	0	0	0	5
9	25	2	49	0	0	0	49
1	27	1	6	0	0	47	53
5	30	5	6	0	0	50	56
10	34	6	36	0	0	52	88
6	36	10	46	0	0	86	132
8	39	1	44	0	0	129	173
2	46	2	50	0	0	166	216
3	46	9	27	0	0	216	243
7	48	6	7	0	0	241	248

```
  average waiting time : 98.70 average turnaround time : 126.30
```

2. Nonpreemptive SJF

```
+-------------------------------------------------------------+
|              Non-Preemptive SJF Scheduling Report           |
+-------------------------------------------------------------+
************************* Gantt Chart **************************

                     4----9-----------------------------------------
-------1-----5-----7------3-------------------------10------------
------------------------8-----------------------------------------
6-------------------------------------------2--------------------
----------------------------

***************************** Result *******************************

+-----+-------+--------+--------+-------+------+------+----------+
| PID |Arrival|Priority|CPUburst|IOburst|Remain| Wait |Turnaround|
+-----+-------+--------+--------+-------+------+------+----------+
|   4 |   20  |    5   |    5   |    0  |   0  |    0 |     5    |
+-----+-------+--------+--------+-------+------+------+----------+
|   9 |   25  |    2   |   49   |    0  |   0  |    0 |    49    |
+-----+-------+--------+--------+-------+------+------+----------+
|   1 |   27  |    1   |    6   |    0  |   0  |   47 |    53    |
+-----+-------+--------+--------+-------+------+------+----------+
|   5 |   30  |    5   |    6   |    0  |   0  |   50 |    56    |
+-----+-------+--------+--------+-------+------+------+----------+
|   7 |   48  |    6   |    7   |    0  |   0  |   38 |    45    |
+-----+-------+--------+--------+-------+------+------+----------+
|   3 |   46  |    9   |   27   |    0  |   0  |   47 |    74    |
+-----+-------+--------+--------+-------+------+------+----------+
|  10 |   34  |    6   |   36   |    0  |   0  |   86 |   122    |
+-----+-------+--------+--------+-------+------+------+----------+
|   8 |   39  |    1   |   44   |    0  |   0  |  117 |   161    |
+-----+-------+--------+--------+-------+------+------+----------+
|   6 |   36  |   10   |   46   |    0  |   0  |  164 |   210    |
+-----+-------+--------+--------+-------+------+------+----------+
|   2 |   46  |    2   |   50   |    0  |   0  |  200 |   250    |
+-----+-------+--------+--------+-------+------+------+----------+

  average waiting time : 74.90 average turnaround time : 102.50
```

3. Preemptive SJF

```
+-------------------------------------------------------------+
|                Preemptive SJF Scheduling Report             |
+-------------------------------------------------------------+
************************* Gantt Chart ***************************

                    444449911111155555510101010101033777777733333
33333333333333333331010101010101010101010101010101010101010101010101
01010101088888888888888888888888888888888888888888888888666666666666
66666666666666666666666666666669999999999999999999999999999999999999
99999999999992222222222222222222222222222222222222222222222222222222

***************************** Result ****************************

+-----+-------+--------+--------+-------+------+------+----------+
| PID |Arrival|Priority|CPUburst|IOburst|Remain| Wait |Turnaround|
+-----+-------+--------+--------+-------+------+------+----------+
|   4 |   20  |    5   |    5   |    0  |   0  |    0 |     5    |
+-----+-------+--------+--------+-------+------+------+----------+
|   1 |   27  |    1   |    6   |    0  |   0  |    0 |     6    |
+-----+-------+--------+--------+-------+------+------+----------+
|   5 |   30  |    5   |    6   |    0  |   0  |    3 |     9    |
+-----+-------+--------+--------+-------+------+------+----------+
|   7 |   48  |    6   |    7   |    0  |   0  |    0 |     7    |
+-----+-------+--------+--------+-------+------+------+----------+
|   3 |   46  |    9   |   27   |    0  |   0  |    7 |    34    |
+-----+-------+--------+--------+-------+------+------+----------+
|  10 |   34  |    6   |   36   |    0  |   0  |   39 |    75    |
+-----+-------+--------+--------+-------+------+------+----------+
|   8 |   39  |    1   |   44   |    0  |   0  |   70 |   114    |
+-----+-------+--------+--------+-------+------+------+----------+
|   6 |   36  |   10   |   46   |    0  |   0  |  117 |   163    |
+-----+-------+--------+--------+-------+------+------+----------+
|   9 |   25  |    2   |   49   |    0  |   0  |  172 |   221    |
+-----+-------+--------+--------+-------+------+------+----------+
|   2 |   46  |    2   |   50   |    0  |   0  |  200 |   250    |
+-----+-------+--------+--------+-------+------+------+----------+

  average waiting time : 60.80 average turnaround time : 88.40
```

4. Nonpreemptive Priority

```
+-------------------------------------------------------------+
|           Non-Preemptive Pritority Scheduling Report        |
+-------------------------------------------------------------+
************************* Gantt Chart **************************

                    4----9-----------------------------------------
-------6-----------------------------------------3-------------
------------10-----------------------------------7------5-----2---
-----------------------------------------1-----8-------------
---------------------------
```

```
**************************** Result ******************************
```

PID	Arrival	Priority	CPUburst	IOburst	Remain	Wait	Turnaround
4	20	5	5	0	0	0	5
9	25	2	49	0	0	0	49
6	36	10	46	0	0	38	84
3	46	9	27	0	0	74	101
10	34	6	36	0	0	113	149
7	48	6	7	0	0	135	142
5	30	5	6	0	0	160	166
2	46	2	50	0	0	150	200
1	27	1	6	0	0	219	225
8	39	1	44	0	0	213	257

```
  average waiting time : 110.20 average turnaround time : 137.80
```

5. Preemptive Priority

```
+-------------------------------------------------------------------+
|             Preemptive Priority Scheduling Report                 |
+-------------------------------------------------------------------+
************************** Gantt Chart ***************************

                     4444499999555510106666666666666666666666666666
66666666666666666333333333333333333333333331010101010101010101010101
01010101010101010101010101010101010101010107777777559999999999999
99999999999999999999999999999992222222222222222222222222222222222
22222222222221111118888888888888888888888888888888888888888888888

***************************** Result *****************************

+-----+-------+--------+--------+-------+------+------+----------+
| PID |Arrival|Priority|CPUburst|IOburst|Remain| Wait |Turnaround|
+-----+-------+--------+--------+-------+------+------+----------+
|   4 |   20  |    5   |    5   |    0  |   0  |    0 |      5   |
+-----+-------+--------+--------+-------+------+------+----------+
|   6 |   36  |   10   |   46   |    0  |   0  |    0 |     46   |
+-----+-------+--------+--------+-------+------+------+----------+
|   3 |   46  |    9   |   27   |    0  |   0  |   36 |     63   |
+-----+-------+--------+--------+-------+------+------+----------+
|  10 |   34  |    6   |   36   |    0  |   0  |   73 |    109   |
+-----+-------+--------+--------+-------+------+------+----------+
|   7 |   48  |    6   |    7   |    0  |   0  |   95 |    102   |
+-----+-------+--------+--------+-------+------+------+----------+
|   5 |   30  |    5   |    6   |    0  |   0  |  116 |    122   |
+-----+-------+--------+--------+-------+------+------+----------+
|   9 |   25  |    2   |   49   |    0  |   0  |  122 |    171   |
+-----+-------+--------+--------+-------+------+------+----------+
|   2 |   46  |    2   |   50   |    0  |   0  |  150 |    200   |
+-----+-------+--------+--------+-------+------+------+----------+
|   1 |   27  |    1   |    6   |    0  |   0  |  219 |    225   |
+-----+-------+--------+--------+-------+------+------+----------+
|   8 |   39  |    1   |   44   |    0  |   0  |  213 |    257   |
+-----+-------+--------+--------+-------+------+------+----------+

  average waiting time : 102.40 average turnaround time : 130.00
```

6. RR(Round Robin)

```
+--------------------------------------------------------------+
|                     Round Robin Scheduling                   |
+--------------------------------------------------------------+
                      Enter quantum: 10
+--------------------------------------------------------------+
|                 Round Robin Scheduling Report                |
+--------------------------------------------------------------+
************************* Gantt Chart ***************************

                     444449999999999111111555555101010101010101010
99999999996666666666888888888822222222233333333377777771010101010
10101010109999999999666666666668888888888222222222333333333331010101
01010101010109999999999666666666668888888888222222222233333331010101
01010999999999666666666688888888882222222222666666688882222222222

***************************** Result ****************************

+-----+-------+--------+--------+-------+------+------+----------+
| PID |Arrival|Priority|CPUburst|IOburst|Remain| Wait |Turnaround|
+-----+-------+--------+--------+-------+------+------+----------+
|   4 |   20  |    5   |    5   |    0  |   0  |    0 |     5    |
+-----+-------+--------+--------+-------+------+------+----------+
|   1 |   27  |    1   |    6   |    0  |   0  |    8 |    14    |
+-----+-------+--------+--------+-------+------+------+----------+
|   5 |   30  |    5   |    6   |    0  |   0  |   11 |    17    |
+-----+-------+--------+--------+-------+------+------+----------+
|   7 |   48  |    6   |    7   |    0  |   0  |   59 |    66    |
+-----+-------+--------+--------+-------+------+------+----------+
|   3 |   46  |    9   |   27   |    0  |   0  |  158 |   185    |
+-----+-------+--------+--------+-------+------+------+----------+
|  10 |   34  |    6   |   36   |    0  |   0  |  167 |   203    |
+-----+-------+--------+--------+-------+------+------+----------+
|   9 |   25  |    2   |   49   |    0  |   0  |  172 |   221    |
+-----+-------+--------+--------+-------+------+------+----------+
|   6 |   36  |   10   |   46   |    0  |   0  |  200 |   246    |
+-----+-------+--------+--------+-------+------+------+----------+
|   8 |   39  |    1   |   44   |    0  |   0  |  203 |   247    |
+-----+-------+--------+--------+-------+------+------+----------+
|   2 |   46  |    2   |   50   |    0  |   0  |  200 |   250    |
+-----+-------+--------+--------+-------+------+------+----------+

  average waiting time : 117.80 average turnaround time : 145.40
```

7. Evaluation

Evaluation에서는 위의 모든 알고리즘 수행 결과가 출력되고, 마지막에는 모든 알고리즘의 평균 대기 시간과 반환 시간을 정리한 Final Report가 출력된다. 이 때, 최소의 평균 대기 시간과 최소의 평균 반환 시간을 가진 알고리즘을 함께 출력한다.

```
+-------------------------------------------------------------------+
|                          Final Report                             |
+-------------------------------------------------------------------+
| FCFS Average Waiting Time = 98.70
| FCFS Average Turnaround Time = 126.30
+-------------------------------------------------------------------+
| NonPreemptive SJF Average Waiting Time = 74.90
| NonPreemptive SJF Average Turnaround Time = 102.50
+-------------------------------------------------------------------+
| NonPreemptive Priority Average Waiting Time = 110.20
| NonPreemptive Priority Average Turnaround Time = 137.80
+-------------------------------------------------------------------+
| Preemptive SJF Average Waiting Time = 60.80
| Preemptive SJF Average Turnaround Time = 88.40
+-------------------------------------------------------------------+
| Preemptive Priority Average Waiting Time = 102.40
| Preemptive Priority Average Turnaround Time = 130.00
+-------------------------------------------------------------------+
| Round Robin Average Waiting Time = 117.80
| Round Robin Average Turnaround Time = 145.40
+-------------------------------------------------------------------+
| Minimum Waiting Time Algorithm : Preemptive SJF
| Minimum Turnaround Time Algorithm : Preemptive SJF
+-------------------------------------------------------------------+
```

8. Exit

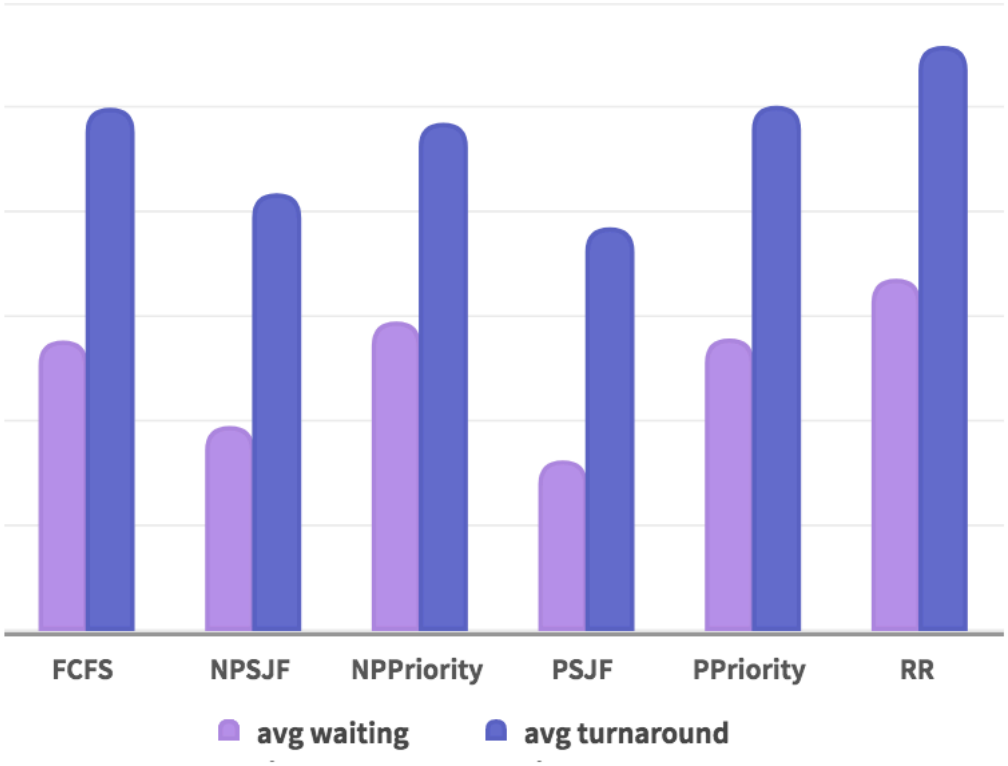
Exit을 선택할 경우 프로그램이 종료된다.

결론

알고리즘간 성능 비교

프로세스 5개에 대해 총 20번의 실험을 진행하였고 대기 시간과 반환 시간의 평균치를 구해 성능을 비교했다. 동일한 알고리즘일 경우 선점 스케줄링일 때 더 좋은 성능을 보였으며, 평균 대기 시간과 평균 반환 시간 모두 Preemptive Shortes Job First 스케줄링에서 가장 작게 나타났다. 구체적인 수치는 다음과 같다.

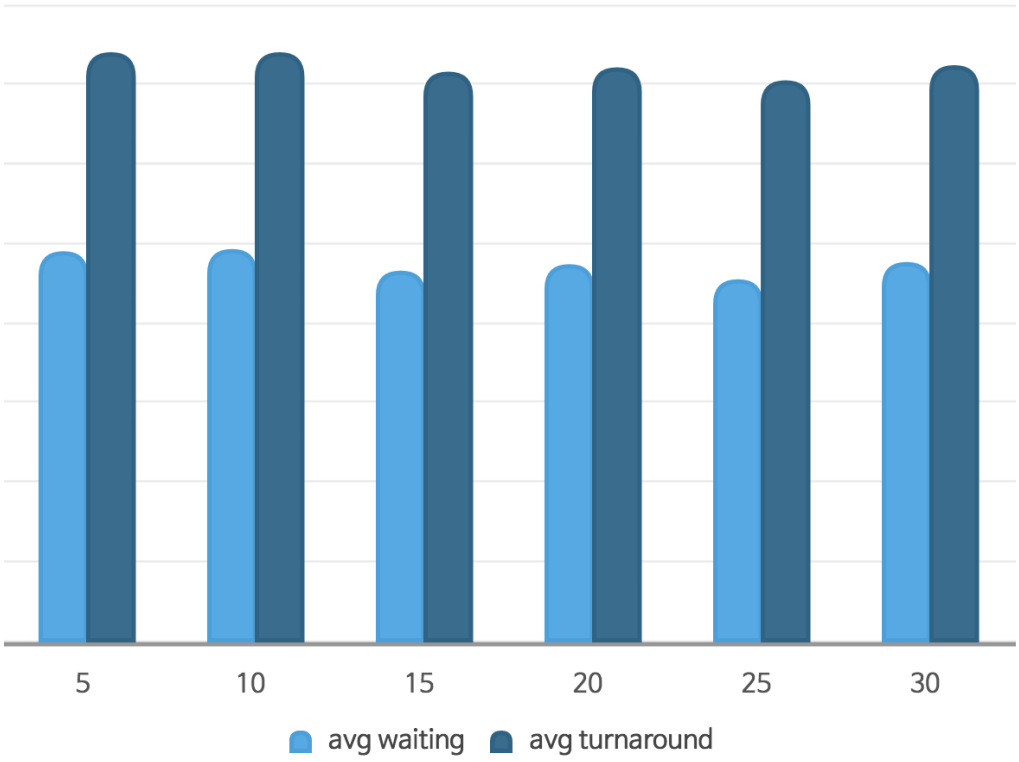
	FCFS	NPSJF	NPPriority	PSJF	Priority	RR
AVG waiting time	27.62	19.54	29.56	16.32	27.84	33.58
AVG turnaround time	49.8	41.72	48.5	38.5	50.02	55.76



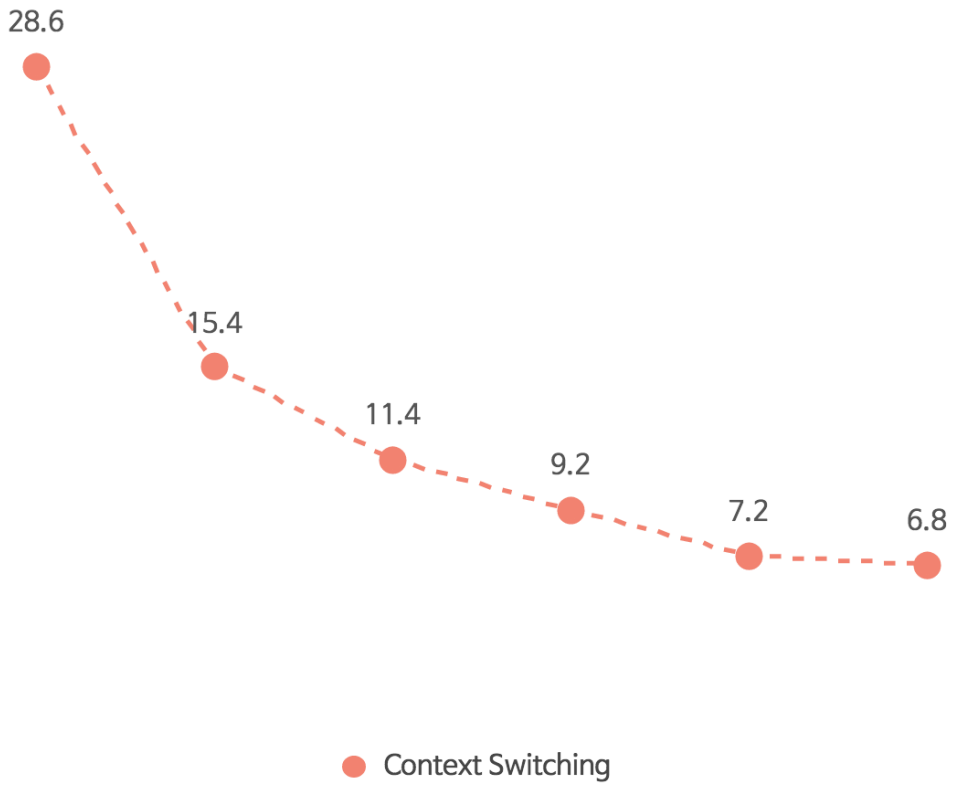
[Round Robin] 시간 할당량(time quantum)에 따른 성능 비교

프로세스 5개에 대해 5ms 간격으로 시간 할당량을 바꾸어가며 성능을 비교했다. 평균 대기 시간과 평균 반환 시간 모두 시간 할당량이 25ms일 때 가장 작게 나타났지만, 평균 대기 시간과 평균 반환 시간만 보면 시간 할당량 변화에 따른 성능 변화는 미미하다.

time quantum	5ms	10ms	15ms	20ms	25ms	30ms
AVG waiting time	48.96	49.1	46.54	47.2	45.46	47.5
AVG turnaround time	73.84	73.98	71.42	72.08	70.34	72.38



이는 문맥 교환에 따른 오버헤드를 고려하지 않았기 때문이다. 따라서 각 시간 할당량에 따른 문맥 교환 횟수를 비교했다.



본 실험에서는 작업 시간의 범위를 1ms 이상 50ms 이하로 설정했는데 이 경우 평균적으로 시간 할당량이 25ms 혹은 30ms 일 때 문맥 교환이 가장 적게 일어나는 것으로 나타났다.

프로젝트 소감 및 추후 발전 방안

스케줄링에 대한 개념은 이해하고 있었지만 코드로 구현하려고 하니 어려움이 많았다. 특히 구조체와 포인터를 사용해서 구현했는데 동적할당에 서툴러서 새로운 노드를 만들어서 정렬하거나 큐에 삽입할 때 시간이 오래 걸렸다. 하지만 프로젝트를 진행하면서 개념적으로만 이해하고 있었던 스케줄링 과정을 CPU가 실행 중일때, 스케줄링이 끝났을 때, idle 상태일 때 등으로 나누어서 생각해볼 수 있어서 스케줄링을 더 깊게 이해할 수 있었다. 또한 I/O operation이나 추가 구현 사항을 구현하지 못했는데 데모 시간에 다른 사람들이 발표하는 내용을 들으며 어떻게 구현할지 생각해 볼 수 있었다. 기존 시뮬레이터에 대한 내용은 "웹 기반의 교육용 CPU 스케줄링 시뮬레이터의 설계 및 구현"이라는 논문을 참고했는데 다른 여러 시뮬레이터들이 동작하는 방식을 보고 다음에 다시 구현해 볼 기회가 있다면 수정또는 추가해보고 싶은 것을 다음과 같이 생각해보았다.

- I/O operation 기능
- CPUsim2처럼 간트 차트를 한꺼번에 출력하여 비교해볼 수 있는 기능
- ready queue를 시각화하여 실시간으로 보여주는 기능

오류 및 해결 방법

#1 Random data 부여

- process data 생성을 위한 random 값 필요
- rand 사용시 계속 동일값 생성

[해결] 난수 발생

링크 참고
srand와 time.h 이용하여 seed 만든 후 rand 사용

```
seed = time(NULL);
srand(seed);
...
node->CPUburstTime = rand() % CPUtime + 1;
```

```
srand(time(NULL));
```

#2 프로세스 정렬

`shead` -> `head` -> ... -> `tail` -> `stail` 형식으로 프로세스 큐 앞뒤에 임의의 노드를 붙여 정렬을 하려고 했으나 다음과 같은 오류가 발생했다.

```
temp.c:151:6: error: member reference base type 'PROCESS *'
      (aka 'struct process *') is not a structure or union
        tail->Next = stail;
```

노드 동적할당

`shead`와 `stail` 노드를 동적할당하여 연결해주었다.

```
/* shead -> head -> ... -> tail -> stail */
shead = (PROCESS *)malloc(sizeof(PROCESS));
stail = (PROCESS *)malloc(sizeof(PROCESS));
(*tail) = next = (PROCESS *)malloc(sizeof(PROCESS));

shead->Next = *head;
stail = (*tail)->Next;
stail->Next = NULL;
```

그러나 세 개 단위로 묶어서 정렬되어 프로세스의 개수가 3개 이하일 경우 오류가 발생했다.

```
+-----+-------+--------+--------+-------+------+------+----------+
| PID |Arrival|Priority|CPUburst|IOburst|Remain| Wait |Turnaround|
+-----+-------+--------+--------+-------+------+------+----------+
|   1 |   41  |    3   |   38   |    0  |  38  |    0 |     0    |
+-----+-------+--------+--------+-------+------+------+----------+
|   0 | -1342177280  |    4   | -1342177280   |    0  |   0  |    0 |
0    |
+-----+-------+--------+--------+-------+------+------+----------+
```

SWAP 포인터 사용

swap을 위한 포인터를 따로 만들어서 정렬할 때 이용했다.

```
PROCESS *FRONT, *MID, *END; /* for swap */
...
/* bubble sort */
for (i=0; i<processNum; i++)
{
    FRONT = shead;
    MID = FRONT->Next;
    END = MID->Next;

    while(END!=stail) {

        if(MID->ArrivalTime > END->ArrivalTime) {
            FRONT->Next = END;
            MID->Next = END->Next;
            END->Next = MID;
        }
        FRONT = FRONT->Next;
        MID = FRONT->Next;
        END = MID->Next;
    }
}
```

또한 정렬이 끝난 후에 head는 맨 첫 노드, tail은 맨 끝 노드로 설정해주었다.

```
/* set head and tail */
*head = shead->Next;
*tail = FRONT->Next;
```

```
+-----+-------+--------+--------+-------+------+------+----------+
| PID |Arrival|Priority|CPUburst|IOburst|Remain| Wait |Turnaround|
+-----+-------+--------+--------+-------+------+------+----------+
|   4 |   27  |    1   |   13   |    0  |  13  |    0 |      0   |
+-----+-------+--------+--------+-------+------+------+----------+
|   1 |   28  |    4   |   31   |    0  |  31  |    0 |      0   |
+-----+-------+--------+--------+-------+------+------+----------+
|   2 |   28  |    4   |   38   |    0  |  38  |    0 |      0   |
+-----+-------+--------+--------+-------+------+------+----------+
|   5 |   36  |    4   |   41   |    0  |  41  |    0 |      0   |
+-----+-------+--------+--------+-------+------+------+----------+
|   3 |   40  |    2   |   44   |    0  |  44  |    0 |      0   |
+-----+-------+--------+--------+-------+------+------+----------+
|   0 |    0  |    0   |    0   |    0  |   0  |    0 |      0   |
+-----+-------+--------+--------+-------+------+------+----------+
```

정렬 후에 0번 process가 추가되어 출력되는 문제가 나타났는데 tail에서 stail로의 연결이 끊어지지 않아 stail 노드가 출력되는 것이었다.

[해결] 불필요한 연결 해제

tail을 설정하기 전 stail의 연결을 끊어주었다.

```
*head = shead->Next;
MID->Next = NULL; /* to cut off stail */
*tail = FRONT->Next;
```

#3 스케줄링 후 프로세스 큐에 결과값 적용

스케줄링한 결과를 그대로 프로세스 큐에 적용하도록 했는데 이 방식은 동일한 데이터로 다른 알고리즘을 돌릴 때 다시 초기화해주는 과정이 필요했다. 그래서 따로 endHead와 endTail을 만들어 스케줄링이 끝난 프로세스 데이터를 복사해 큐에 삽입하는 방식을 사용했다.

[해결] endHead와 endTail 사용

```
Enqueue(node, &endHead, &endTail);
...
PrintProcess(&endHead);
```

#4 Gantt Chart 출력과정에서 무한루프 도는 경우 발생

- 3개 프로세스 입력시 스케줄링이 끝난 프로세스를 다시 스케줄링 후 83초에서 무한루프

```
74 : CPUscheduling 2
75 : executing
76 : executing
77 : executing
78 : executing
79 : executing
80 : executing
81 : executing
82 : executing
83 : CPUscheduling 2
83 : executing
83 : executing
...
```

- 4개,5개 프로세스 입력시 정상스케줄링하다가 83초에서 멈춤

```
73 : executing
74 : CPUscheduling 2
75 : executing
76 : executing
77 : executing
78 : executing
79 : executing
80 : executing
81 : executing
82 : executing
83 : CPUscheduling 4
```

무한루프 발생 상황 - 디버깅

- 스케줄링 과정 중 발생하는 문제는 없음
- CPUstate 변화 정상적으로 이루어짐
- 다음 노드로 넘어가는 과정 혹은 노드 삽입 과정에서 문제 예상

[해결] 매번 노드 동적할당

node 사용시 처음에만 malloc하고 후에는 그냥 사용했는데 node는 구조체이기 때문에 사용할 때마다 malloc을 해주어야 한다.

```
node = (PROCESS *)malloc(sizeof(PROCESS));
CopyNode(readyHead, node);
Enqueue(node, &endHead, &endTail);
...
CopyNode(readyHead, node);
Enqueue(node, &endHead, &endTail);
```

#5 CPU idle일 때 Gantt Chart 공백 출력

readyQueue에 아무런 프로세스가 존재하지 않고 CPU가 idle일 때 공백을 출력하도록 했다.

```
if(readyHead==NULL && CPUstate==IDLE) {
    printf(" ");
}
```

#6 [Round Robin] time quantum이 종료된 프로세스 readyQueue에 재삽입

프로세스 큐에서 해당 프로세스를 복사하여 readyQueue에 넣도록 했는데 segmentation fault가 발생했다.

[해결] readyQueue에서 복사 후 삽입

time quantum이 만료된 프로세스는 processQueue가 아닌 readyHead에서 노드를 복사하여 다시 readyQueue에 넣어주어야한다.

```
/* time expired - move to ready queue */
if(CPUstate > 0) {
  node = (PROCESS *)malloc(sizeof(PROCESS));
  CopyNode(readyHead, node);
  Enqueue(node, &readyHead, &readyTail);
  readyHead = readyHead->Next;
}
```

#7 [Round Robin] 스케줄링이 끝난 후 remain time이 1로 출력

```
**************************** Result *******************************

+-----+-------+--------+--------+-------+------+------+----------+
| PID |Arrival|Priority|CPUburst|IOburst|Remain| Wait |Turnaround|
+-----+-------+--------+--------+-------+------+------+----------+
|   2 |   13  |    1   |   20   |    0  |   1  |   34 |    54    |
+-----+-------+--------+--------+-------+------+------+----------+
|   5 |    7  |    2   |   29   |    0  |   1  |   50 |    79    |
+-----+-------+--------+--------+-------+------+------+----------+
|   4 |   26  |    3   |   27   |    0  |   1  |   80 |   107    |
+-----+-------+--------+--------+-------+------+------+----------+
|   3 |   22  |    5   |   39   |    0  |   1  |   91 |   130    |
+-----+-------+--------+--------+-------+------+------+----------+
|   1 |   29  |    2   |   41   |    0  |   1  |   93 |   134    |
+-----+-------+--------+--------+-------+------+------+----------+
```

[해결] remain time이 1일 때 endQueue에 복사하기 전 remain time 감소

remain time == 1일 때 endQueue에 해당 프로세스를 복사하는데
복사 후에 remain time을 감소시켜서 endQueue에는 remain time이 0이 되는 것이 적용되지 않음

```
/* end of execute */
if(readyNode->RemainTime == 1) {
  readyNode->Turnaround = readyNode->WaitingTime + readyNode->ExecutingTime
+ 1;
  waiting[5] += readyNode->WaitingTime;
  turnaround[5] += readyNode->Turnaround;
  endNum++;
  CPUstate = SCHEDULE;

  readyNode->ExecutingTime++;
  readyNode->RemainTime--;

  node = (PROCESS *)malloc(sizeof(PROCESS));
  CopyNode(readyNode, node);
  Enqueue(node, &endHead, &endTail);
}
else {
  readyNode->ExecutingTime++;
  readyNode->RemainTime--;
}
timequantum--;
```

부록 (소스코드)

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

#define CPUtime 50
#define SCHEDULE 0
#define IDLE -1

typedef struct process PROCESS;
typedef struct process {
    int ProcessID;
    int CPUburstTime;
    int IOburstTime;
    int ArrivalTime;
    int Priority;
    int RemainTime;
    int WaitingTime;
    int ExecutingTime; /* to compute Turnaround time */
    int Turnaround;
    PROCESS *Next;
} PROCESS;

PROCESS *head = NULL, *tail = NULL;
```

```
PROCESS *readyHead = NULL, *readyTail = NULL;
PROCESS *endHead = NULL, *endTail = NULL;
int processNum, CPUstate, quantum;

double waiting[6] = {0,0,0,0,0,0};
double turnaround[6] = {0,0,0,0,0,0};

/*--- function declaration ---*/
void CreateProcess();
void Enqueue(PROCESS *node, PROCESS **head, PROCESS **tail);

void Menu();
void Select(int select);

void PrintProcess(PROCESS **head);
void PrintEval();
void PrintMin(int result);
void InitEval();

void SortByArrival(PROCESS **head, PROCESS **tail);
void SortByBurst(PROCESS **head, PROCESS **tail);
void SortByPriority(PROCESS **head, PROCESS **tail);
void SortByRemain(PROCESS **head, PROCESS **tail);

void CopyNode(PROCESS *node, PROCESS *new);
void CopyQueue(PROCESS **queue, PROCESS **head, PROCESS **tail);

void FCFS();
void NPSJF();
void NPPriority();

void PSJF();
void PPriority();
void RR();

/*--- function definition ---*/
void Menu() {

    int select;
    printf("+-----------------------------------------------------------------
-+\n");
    printf("|                    CPU Scheduling Simulator
|\n");
    printf("+-----------------------------------------------------------------
-+\n");
    printf("|
|\n");
    printf("|   1. FCFS(First Come First Served)
|\n");
```

```
    printf("|   2. Nonpreemptive SJF(Shortest Job First)
|\n");
    printf("|   3. Preemptive SJF(Shortest Job First)
|\n");
    printf("|   4. Nonpreemptive Priority
|\n");
    printf("|   5. Preemptive Priority
|\n");
    printf("|   6. RR(Round Robin)
|\n");
    printf("|   7. Evaluation
|\n");
    printf("|   8. Exit
|\n");
    printf("|
|\n");
    printf("+-----------------------------------------------------------------
-+\n\n");
    printf("                  Select CPU Scheduling Algorithm : ");
    scanf("%d", &select);
    Select(select);

}

void Select(int select) {

    switch(select) {
        case 1:
            InitEval();
            FCFS();
            break;
        case 2:
            InitEval();
            NPSJF();
            break;
        case 3:
            InitEval();
            PSJF();
            break;
        case 4:
            InitEval();
            NPPriority();
            break;
        case 5:
            InitEval();
            PPriority();
            break;
        case 6:
            InitEval();
```

```
            RR();
            break;
        case 7:
            InitEval();
            FCFS();
            NPSJF();
            PSJF();
            NPPriority();
            PPriority();
            RR();
            PrintEval();
            break;
        case 8:
            //Exit
            exit(0);
        default:
            //Error
            break;
    }

}

void CreateProcess() {

    int i;
    long seed;

    head = NULL, tail = NULL;
    PROCESS *node;

    seed = time(NULL);
    srand(seed);

    printf("+-----------------------------------------------------------------
-+\n");
    printf("|                          ProcessCreation
|\n");
    printf("+-----------------------------------------------------------------
-+\n");
    printf("                  Enter the number of process : ");
    scanf("%d", &processNum);

    for(i = 0;i < processNum; i++) {
        node = (PROCESS *)malloc(sizeof(PROCESS));
        node->ProcessID = i+1;
        node->CPUburstTime = rand() % CPUtime + 1;
        node->IOburstTime = 0;
        node->ArrivalTime = rand() % CPUtime + 1;
        node->Priority = rand() % processNum + 1;
```

```
        node->RemainTime = node->CPUburstTime;
        node->WaitingTime = 0;
        node->ExecutingTime = 0;
        node->Turnaround = 0;
        node->Next = NULL;
        Enqueue(node, &head, &tail);
    }

    PROCESS *temp;
    temp = head;
    PrintProcess(&temp);
}

void Enqueue(PROCESS *node, PROCESS **head, PROCESS **tail) {

    if(*head == NULL) /* if queue is empty */
        *head = node;
    else (*tail)->Next = node;
    *tail = node;

}

void PrintProcess(PROCESS **head) {

    printf("+-----+-------+--------+--------+-------+------+------+---------
-+\n");
    printf("| PID |Arrival|Priority|CPUburst|IOburst|Remain| Wait
|Turnaround|\n");
    printf("+-----+-------+--------+--------+-------+------+------+---------
-+\n");

    for(; *head != NULL; *head = (*head)->Next) {
        printf("| %3d |  %3d  |  %3d   |  %3d   |  %3d  | %3d  | %4d |   %3d
   |\n",
            (*head)->ProcessID, (*head)->ArrivalTime, (*head)->Priority,
(*head)->CPUburstTime,
            (*head)->IOburstTime, (*head)->RemainTime, (*head)->WaitingTime,
(*head)->Turnaround);
        printf("+-----+-------+--------+--------+-------+------+------+-----
-----+\n");
    }

}

void PrintEval() {

    int i, result = 0;
    double minWaiting;
    double minTurnaround;
```

```
    printf("+-------------------------------------------------------------
-+\n");
    printf("|                            Final Report
|\n");
    printf("+-------------------------------------------------------------
-+\n");
    printf("| FCFS Average Waiting Time = %.2lf
\n", waiting[0]/(double)processNum);
    printf("| FCFS Average Turnaround Time = %.2lf
\n", turnaround[0]/(double)processNum);
    printf("+-------------------------------------------------------------
-+\n");
    printf("| NonPreemptive SJF Average Waiting Time = %.2lf
\n", waiting[1]/(double)processNum);
    printf("| NonPreemptive SJF Average Turnaround Time = %.2lf
\n", turnaround[1]/(double)processNum);
    printf("+-------------------------------------------------------------
-+\n");
    printf("| NonPreemptive Priority Average Waiting Time = %.2lf
\n", waiting[2]/(double)processNum);
    printf("| NonPreemptive Priority Average Turnaround Time = %.2lf
\n", turnaround[2]/(double)processNum);
    printf("+-------------------------------------------------------------
-+\n");
    printf("| Preemptive SJF Average Waiting Time = %.2lf
\n", waiting[3]/(double)processNum);
    printf("| Preemptive SJF Average Turnaround Time = %.2lf
\n", turnaround[3]/(double)processNum);
    printf("+-------------------------------------------------------------
-+\n");
    printf("| Preemptive Priority Average Waiting Time = %.2lf
\n", waiting[4]/(double)processNum);
    printf("| Preemptive Priority Average Turnaround Time = %.2lf
\n", turnaround[4]/(double)processNum);
    printf("+-------------------------------------------------------------
-+\n");
    printf("| Round Robin Average Waiting Time = %.2lf
\n", waiting[5]/(double)processNum);
    printf("| Round Robin Average Turnaround Time = %.2lf
\n", turnaround[5]/(double)processNum);
    printf("+-------------------------------------------------------------
-+\n");

    minWaiting = waiting[0];
    minTurnaround = turnaround[0];

    for(i = 1; i < 6; i++) {
        if(minWaiting > waiting[i]) {
```

```
            minWaiting = waiting[i];
            result = i;
        }
    }
    printf("| Minimum Waiting Time Algorithm : ");
    PrintMin(result);

    result = 0;
    for(i = 1; i < 6; i++) {
        if(minTurnaround > turnaround[i]) {
            minTurnaround = turnaround[i];
            result = i;
        }
    }
    printf("| Minimum Turnaround Time Algorithm : ");
    PrintMin(result);
    printf("+-----------------------------------------------------------------
-+\n");

}

void PrintMin(int result) {
    switch(result) {
        case 0:
            printf("FCFS\n");
            break;
        case 1:
            printf("NonPreemptive SJF\n");
            break;
        case 2:
            printf("NonPreemptive Priority\n");
            break;
        case 3:
            printf("Preemptive SJF\n");
            break;
        case 4:
            printf("Preemptive Priority\n");
            break;
        case 5:
            printf("Round Robin\n");
            break;
    }
}

void InitEval() {
    int i;
    for(i = 0; i < 6; i++) {
        waiting[i] = 0;
        turnaround [i] = 0;
```

```
    }
}

void SortByArrival(PROCESS **head, PROCESS **tail) {

    PROCESS *FRONT, *MID, *END; /* for swap */
    PROCESS *shead, *stail, *temp;
    int i;

    /* shead -> head -> ... -> tail -> stail */
    shead = (PROCESS *)malloc(sizeof(PROCESS));
    stail = (PROCESS *)malloc(sizeof(PROCESS));
    (*tail)->Next = (PROCESS *)malloc(sizeof(PROCESS));

    shead->Next = *head;
    stail = (*tail)->Next;
    stail->Next = NULL;

    /* bubble sort */
    for (i=0; i<processNum; i++)
    {
        FRONT = shead;
        MID = FRONT->Next;
        END = MID->Next;

        while(END!=stail) {

            if(MID->ArrivalTime > END->ArrivalTime) {
                FRONT->Next = END;
                MID->Next = END->Next;
                END->Next = MID;
            }
            FRONT = FRONT->Next;
            MID = FRONT->Next;
            END = MID->Next;
        }
    }

    /* set head and tail */
    *head = shead->Next;
    MID->Next = NULL; /* to cut off stail */
    *tail = FRONT->Next;

    free(shead);
    free(stail);

}

void SortByBurst(PROCESS **head, PROCESS **tail) {
```

```
    PROCESS *FRONT, *MID, *END; /* for swap */
    PROCESS *shead, *stail, *temp;
    int i;

    /* shead -> head -> ... -> tail -> stail */
    shead = (PROCESS *)malloc(sizeof(PROCESS));
    stail = (PROCESS *)malloc(sizeof(PROCESS));
    (*tail)->Next = (PROCESS *)malloc(sizeof(PROCESS));

    shead->Next = *head;
    stail = (*tail)->Next;
    stail->Next = NULL;

    /* bubble sort */
    for (i=0; i<processNum; i++)
    {
        FRONT = shead;
        MID = FRONT->Next;
        END = MID->Next;

        while(END!=stail) {

            if(MID->CPUburstTime > END->CPUburstTime) {
                FRONT->Next = END;
                MID->Next = END->Next;
                END->Next = MID;
            }
            FRONT = FRONT->Next;
            MID = FRONT->Next;
            END = MID->Next;
        }
    }

    /* set head and tail */
    *head = shead->Next;
    MID->Next = NULL; /* to cut off stail */
    *tail = FRONT->Next;

    free(shead);
    free(stail);

}

void SortByPriority(PROCESS **head, PROCESS **tail) {

    PROCESS *FRONT, *MID, *END; /* for swap */
    PROCESS *shead, *stail, *temp;
    int i;
```

```
    /* shead -> head -> ... -> tail -> stail */
    shead = (PROCESS *)malloc(sizeof(PROCESS));
    stail = (PROCESS *)malloc(sizeof(PROCESS));
    (*tail)->Next = (PROCESS *)malloc(sizeof(PROCESS));

    shead->Next = *head;
    stail = (*tail)->Next;
    stail->Next = NULL;

    /* bubble sort */
    for (i=0; i<processNum; i++)
    {
        FRONT = shead;
        MID = FRONT->Next;
        END = MID->Next;

        while(END!=stail) {

            if(MID->Priority < END->Priority) {
                FRONT->Next = END;
                MID->Next = END->Next;
                END->Next = MID;
            }
            FRONT = FRONT->Next;
            MID = FRONT->Next;
            END = MID->Next;
        }
    }

    /* set head and tail */
    *head = shead->Next;
    MID->Next = NULL; /* to cut off stail */
    *tail = FRONT->Next;

    free(shead);
    free(stail);

}

void SortByRemain(PROCESS **head, PROCESS **tail) {

    PROCESS *FRONT, *MID, *END; /* for swap */
    PROCESS *shead, *stail, *temp;
    int i;

    /* shead -> head -> ... -> tail -> stail */
    shead = (PROCESS *)malloc(sizeof(PROCESS));
    stail = (PROCESS *)malloc(sizeof(PROCESS));
```

```
    (*tail)->Next = (PROCESS *)malloc(sizeof(PROCESS));

    shead->Next = *head;
    stail = (*tail)->Next;
    stail->Next = NULL;

    /* bubble sort */
    for (i=0; i<processNum; i++)
    {
        FRONT = shead;
        MID = FRONT->Next;
        END = MID->Next;

        while(END!=stail) {

            if(MID->RemainTime > END->RemainTime) {
                FRONT->Next = END;
                MID->Next = END->Next;
                END->Next = MID;
            }
            FRONT = FRONT->Next;
            MID = FRONT->Next;
            END = MID->Next;
        }
    }

    /* set head and tail */
    *head = shead->Next;
    MID->Next = NULL; /* to cut off stail */
    *tail = FRONT->Next;

    free(shead);
    free(stail);

}

void CopyNode(PROCESS *node, PROCESS *new) {

    new->ProcessID = node->ProcessID;
    new->CPUburstTime = node->CPUburstTime;
    new->IOburstTime = node->IOburstTime;
    new->ArrivalTime = node->ArrivalTime;
    new->Priority = node->Priority;
    new->RemainTime = node->RemainTime;
    new->WaitingTime = node->WaitingTime;
    new->ExecutingTime = node->ExecutingTime;
    new->Turnaround = node->Turnaround;
    new->Next = NULL;
```

```
}

void CopyQueue(PROCESS **queue, PROCESS **head, PROCESS **tail) {

    PROCESS *node;
    for(; *queue != NULL; *queue = (*queue)->Next) {
        node = (PROCESS *)malloc(sizeof(PROCESS));
        CopyNode(*queue, node);
        Enqueue(node, head, tail);
    }

}

void FCFS() {

    SortByArrival(&head, &tail);

    /* Process Queue */
    PROCESS *processHead = NULL, *processTail = NULL;
    PROCESS *processQueue = head, *node, *readyNode;

    int time = 0, endNum = 0;
    CPUstate = IDLE;

    /* Initialize ReadyQueue */
    readyHead = NULL;
    readyTail = NULL;

    /* Copy ProcessQueue */
    CopyQueue(&processQueue, &processHead, &processTail);

    printf("+-----------------------------------------------------------------
-+\n");
    printf("|                      FCFS Scheduling Report
|\n");
    printf("+-----------------------------------------------------------------
-+\n");
    printf("************************* Gantt Chart
*************************\n\n");

    while(endNum < processNum) {

        processQueue = processHead;

        /*  Insert ReadyQueue - Job Scheduling */
        while(processQueue != NULL && processQueue->ArrivalTime == time) {
            node = (PROCESS *)malloc(sizeof(PROCESS));
            CopyNode(processQueue, node);
            Enqueue(node, &readyHead, &readyTail);
```

```
            /* remove job scheduled process from process list */
            processQueue = processQueue->Next;

            /* next process */
            processHead = processHead->Next;
        }

        /* remove end process from readyqueue and make CPU idle */
        if(CPUstate == SCHEDULE) {
            CPUstate = IDLE;
            node = (PROCESS *)malloc(sizeof(PROCESS));
            CopyNode(readyHead, node);
            Enqueue(node, &endHead, &endTail);
            /* remove end process from ready queue */
            readyHead = readyHead->Next;
        }

        /* CPU Scheduling */
        readyNode = readyHead;
        while(readyNode != NULL) {

            /* current executing process */
            if(CPUstate == readyNode->ProcessID) {
                printf("-");
                /* end of execute */
                if(readyNode->RemainTime == 1) {
                    readyNode->Turnaround = readyNode->WaitingTime +
readyNode->ExecutingTime + 1;
                    waiting[0] += readyNode->WaitingTime;
                    turnaround[0] += readyNode->Turnaround;
                    endNum++;
                    CPUstate = SCHEDULE;
                }
                readyNode->ExecutingTime++;
                readyNode->RemainTime--;
            }

            /* cpu state is idle */
            else if(CPUstate == IDLE) {
                printf("%d",readyNode->ProcessID);
                CPUstate = readyNode->ProcessID;
                /* end of execute */
                if(readyNode->RemainTime == 1) {
                    readyNode->Turnaround = readyNode->WaitingTime +
readyNode->ExecutingTime + 1;
                    waiting[0] += readyNode->WaitingTime;
                    turnaround[0] += readyNode->Turnaround;
                    endNum++;
```

```
                    CPUstate = SCHEDULE;
                }
                readyNode->ExecutingTime++;
                readyNode->RemainTime--;
            }

            /* waiting process */
            else {
                readyNode->WaitingTime++;
            }

            readyNode = readyNode->Next;
        }

        if(readyHead==NULL && CPUstate==IDLE) {
            printf(" ");
        }

        time++;
    }

    node = (PROCESS *)malloc(sizeof(PROCESS));
    CopyNode(readyHead, node);
    Enqueue(node, &endHead, &endTail);
    printf("\n\n");

    printf("*************************** Result
*****************************\n\n");
    PrintProcess(&endHead);
    printf("\n  average waiting time : %.2lf average turnaround time : %.2lf
\n",
        waiting[0]/(double)processNum, turnaround[0]/(double)processNum);

}

void NPSJF() {

    SortByArrival(&head, &tail);

    /* Process Queue */
    PROCESS *processHead = NULL, *processTail = NULL;
    PROCESS *processQueue = head, *node, *readyNode;

    int time = 0, endNum = 0;
    CPUstate = IDLE;

    /* Initialize ReadyQueue */
    readyHead = NULL;
    readyTail = NULL;
```

```
    /* Copy ProcessQueue */
    CopyQueue(&processQueue, &processHead, &processTail);

    printf("+-----------------------------------------------------------------
-+\n");
    printf("|              Non-Preemptive SJF Scheduling Report
|\n");
    printf("+-----------------------------------------------------------------
-+\n");
    printf("************************* Gantt Chart
*************************\n\n");

    while(endNum < processNum) {

        processQueue = processHead;

        /*  Insert ReadyQueue - Job Scheduling */
        while(processQueue != NULL && processQueue->ArrivalTime == time) {
            node = (PROCESS *)malloc(sizeof(PROCESS));
            CopyNode(processQueue, node);
            Enqueue(node, &readyHead, &readyTail);

            /* remove job scheduled process from process list */
            processQueue = processQueue->Next;

            /* next process */
            processHead = processHead->Next;
        }

        /* remove end process from readyqueue and make CPU idle */
        if(CPUstate == SCHEDULE) {
            CPUstate = IDLE;
            node = (PROCESS *)malloc(sizeof(PROCESS));
            CopyNode(readyHead, node);
            Enqueue(node, &endHead, &endTail);
            /* remove end process from ready queue */
            readyHead = readyHead->Next;
            if(readyHead != NULL)
                SortByBurst(&readyHead, &readyTail);
        }

        /* CPU Scheduling */
        readyNode = readyHead;
        while(readyNode != NULL) {

            /* current executing process */
            if(CPUstate == readyNode->ProcessID) {
                printf("-");
```

```
                /* end of execute */
                if(readyNode->RemainTime == 1) {
                    readyNode->Turnaround = readyNode->WaitingTime +
readyNode->ExecutingTime + 1;
                    waiting[1] += readyNode->WaitingTime;
                    turnaround[1] += readyNode->Turnaround;
                    endNum++;
                    CPUstate = SCHEDULE;
                }
                readyNode->ExecutingTime++;
                readyNode->RemainTime--;
            }

            /* cpu state is idle */
            else if(CPUstate == IDLE) {
                printf("%d",readyNode->ProcessID);
                CPUstate = readyNode->ProcessID;
                /* end of execute */
                if(readyNode->RemainTime == 1) {
                    readyNode->Turnaround = readyNode->WaitingTime +
readyNode->ExecutingTime + 1;
                    waiting[1] += readyNode->WaitingTime;
                    turnaround[1] += readyNode->Turnaround;
                    endNum++;
                    CPUstate = SCHEDULE;
                }
                readyNode->ExecutingTime++;
                readyNode->RemainTime--;
            }

            /* waiting process */
            else {
                readyNode->WaitingTime++;
            }

            readyNode = readyNode->Next;
        }

        if(readyHead==NULL && CPUstate==IDLE) {
            printf(" ");
        }

        time++;
    }

    node = (PROCESS *)malloc(sizeof(PROCESS));
    CopyNode(readyHead, node);
    Enqueue(node, &endHead, &endTail);
    printf("\n\n");
```

```
    printf("************************** Result
****************************\n\n");
    PrintProcess(&endHead);
    printf("\n  average waiting time : %.2lf average turnaround time : %.2lf
\n",
        waiting[1]/(double)processNum, turnaround[1]/(double)processNum);
}

void NPPriority() {

    SortByArrival(&head, &tail);

    /* Process Queue */
    PROCESS *processHead = NULL, *processTail = NULL;
    PROCESS *processQueue = head, *node, *readyNode;

    int time = 0, endNum = 0;
    CPUstate = IDLE;

    /* Initialize ReadyQueue */
    readyHead = NULL;
    readyTail = NULL;

    /* Copy ProcessQueue */
    CopyQueue(&processQueue, &processHead, &processTail);

    printf("+-----------------------------------------------------------------
-+\n");
    printf("|           Non-Preemptive Pritority Scheduling Report
|\n");
    printf("+-----------------------------------------------------------------
-+\n");
    printf("************************* Gantt Chart
**************************\n\n");

    while(endNum < processNum) {

        processQueue = processHead;

        /*  Insert ReadyQueue - Job Scheduling */
        while(processQueue != NULL && processQueue->ArrivalTime == time) {
            node = (PROCESS *)malloc(sizeof(PROCESS));
            CopyNode(processQueue, node);
            Enqueue(node, &readyHead, &readyTail);

            /* remove job scheduled process from process list */
            processQueue = processQueue->Next;
```

```
            /* next process */
            processHead = processHead->Next;
        }

        /* remove end process from readyqueue and make CPU idle */
        if(CPUstate == SCHEDULE) {
            CPUstate = IDLE;
            node = (PROCESS *)malloc(sizeof(PROCESS));
            CopyNode(readyHead, node);
            Enqueue(node, &endHead, &endTail);
            /* remove end process from ready queue */
            readyHead = readyHead->Next;
            if(readyHead != NULL)
                SortByPriority(&readyHead, &readyTail);
        }

        /* CPU Scheduling */
        readyNode = readyHead;
        while(readyNode != NULL) {

            /* current executing process */
            if(CPUstate == readyNode->ProcessID) {
                printf("-");
                /* end of execute */
                if(readyNode->RemainTime == 1) {
                    readyNode->Turnaround = readyNode->WaitingTime +
readyNode->ExecutingTime + 1;
                    waiting[2] += readyNode->WaitingTime;
                    turnaround[2] += readyNode->Turnaround;
                    endNum++;
                    CPUstate = SCHEDULE;
                }
                readyNode->ExecutingTime++;
                readyNode->RemainTime--;
            }

            /* cpu state is idle */
            else if(CPUstate == IDLE) {
                printf("%d",readyNode->ProcessID);
                CPUstate = readyNode->ProcessID;
                /* end of execute */
                if(readyNode->RemainTime == 1) {
                    readyNode->Turnaround = readyNode->WaitingTime +
readyNode->ExecutingTime + 1;
                    waiting[2] += readyNode->WaitingTime;
                    turnaround[2] += readyNode->Turnaround;
                    endNum++;
                    CPUstate = SCHEDULE;
                }
```

```
                readyNode->ExecutingTime++;
                readyNode->RemainTime--;
            }

            /* waiting process */
            else {
                readyNode->WaitingTime++;
            }

            readyNode = readyNode->Next;
        }

        if(readyHead==NULL && CPUstate==IDLE) {
            printf(" ");
        }

        time++;
    }

    node = (PROCESS *)malloc(sizeof(PROCESS));
    CopyNode(readyHead, node);
    Enqueue(node, &endHead, &endTail);
    printf("\n\n");

    printf("*************************** Result
*****************************\n\n");
    PrintProcess(&endHead);
    printf("\n  average waiting time : %.2lf average turnaround time : %.2lf
\n",
        waiting[2]/(double)processNum, turnaround[2]/(double)processNum);
}

void PSJF() {

    SortByArrival(&head, &tail);

    /* Process Queue */
    PROCESS *processHead = NULL, *processTail = NULL;
    PROCESS *processQueue = head, *node, *readyNode;

    int time = 0, endNum = 0;
    CPUstate = IDLE;

    /* Initialize ReadyQueue */
    readyHead = NULL;
    readyTail = NULL;

    /* Copy ProcessQueue */
    CopyQueue(&processQueue, &processHead, &processTail);
```

```
    printf("+-----------------------------------------------------------------
-+\n");
    printf("|                Preemptive SJF Scheduling Report
|\n");
    printf("+-----------------------------------------------------------------
-+\n");
    printf("************************* Gantt Chart
*************************\n\n");

    while(endNum < processNum) {

        processQueue = processHead;

        /*  Insert ReadyQueue - Job Scheduling */
        while(processQueue != NULL && processQueue->ArrivalTime == time) {
            node = (PROCESS *)malloc(sizeof(PROCESS));
            CopyNode(processQueue, node);
            Enqueue(node, &readyHead, &readyTail);

            /* remove job scheduled process from process list */
            processQueue = processQueue->Next;

            /* next process */
            processHead = processHead->Next;
        }

        /* remove end process from readyqueue and make CPU idle */
        if(CPUstate == SCHEDULE) {
            CPUstate = IDLE;
            node = (PROCESS *)malloc(sizeof(PROCESS));
            CopyNode(readyHead, node);
            Enqueue(node, &endHead, &endTail);
            /* remove end process from ready queue */
            readyHead = readyHead->Next;
        }

        /* sort by remain time */
        if(readyHead != NULL) {
            SortByRemain(&readyHead, &readyTail);
            CPUstate = readyHead->ProcessID;
            printf("%d", readyHead->ProcessID);
        }

        /* CPU Scheduling */
        readyNode = readyHead;
        while(readyNode != NULL) {

            /* current executing process */
```

```
            if(CPUstate == readyNode->ProcessID) {
                /* end of execute */
                if(readyNode->RemainTime == 1) {
                    readyNode->Turnaround = readyNode->WaitingTime +
readyNode->ExecutingTime + 1;
                    waiting[3] += readyNode->WaitingTime;
                    turnaround[3] += readyNode->Turnaround;
                    endNum++;
                    CPUstate = SCHEDULE;
                }
                readyNode->ExecutingTime++;
                readyNode->RemainTime--;
            }

            /* waiting process */
            else {
                readyNode->WaitingTime++;
            }

            readyNode = readyNode->Next;
        }

        if(readyHead==NULL && CPUstate==IDLE) {
            printf(" ");
        }

        time++;
    }

    node = (PROCESS *)malloc(sizeof(PROCESS));
    CopyNode(readyHead, node);
    Enqueue(node, &endHead, &endTail);
    printf("\n\n");

    printf("*************************** Result
*****************************\n\n");
    PrintProcess(&endHead);
    printf("\n  average waiting time : %.2lf average turnaround time : %.2lf
\n",
        waiting[3]/(double)processNum, turnaround[3]/(double)processNum);
}

void PPriority() {

    SortByArrival(&head, &tail);

    /* Process Queue */
    PROCESS *processHead = NULL, *processTail = NULL;
    PROCESS *processQueue = head, *node, *readyNode;
```

```
    int time = 0, endNum = 0;
    CPUstate = IDLE;

    /* Initialize ReadyQueue */
    readyHead = NULL;
    readyTail = NULL;

    /* Copy ProcessQueue */
    CopyQueue(&processQueue, &processHead, &processTail);

    printf("+-----------------------------------------------------------------
-+\n");
    printf("|             Preemptive Priority Scheduling Report
|\n");
    printf("+-----------------------------------------------------------------
-+\n");
    printf("************************* Gantt Chart
*************************\n\n");

    while(endNum < processNum) {

        processQueue = processHead;

        /*  Insert ReadyQueue - Job Scheduling */
        while(processQueue != NULL && processQueue->ArrivalTime == time) {
            node = (PROCESS *)malloc(sizeof(PROCESS));
            CopyNode(processQueue, node);
            Enqueue(node, &readyHead, &readyTail);

            /* remove job scheduled process from process list */
            processQueue = processQueue->Next;

            /* next process */
            processHead = processHead->Next;
        }

        /* remove end process from readyqueue and make CPU idle */
        if(CPUstate == SCHEDULE) {
            CPUstate = IDLE;
            node = (PROCESS *)malloc(sizeof(PROCESS));
            CopyNode(readyHead, node);
            Enqueue(node, &endHead, &endTail);
            /* remove end process from ready queue */
            readyHead = readyHead->Next;
        }

        /* sort by priority */
        if(readyHead != NULL) {
```

```
            SortByPriority(&readyHead, &readyTail);
            CPUstate = readyHead->ProcessID;
            printf("%d", readyHead->ProcessID);
        }

        /* CPU Scheduling */
        readyNode = readyHead;
        while(readyNode != NULL) {

            /* current executing process */
            if(CPUstate == readyNode->ProcessID) {
                /* end of execute */
                if(readyNode->RemainTime == 1) {
                    readyNode->Turnaround = readyNode->WaitingTime +
readyNode->ExecutingTime + 1;
                    waiting[4] += readyNode->WaitingTime;
                    turnaround[4] += readyNode->Turnaround;
                    endNum++;
                    CPUstate = SCHEDULE;
                }
                readyNode->ExecutingTime++;
                readyNode->RemainTime--;
            }

            /* waiting process */
            else {
                readyNode->WaitingTime++;
            }

            readyNode = readyNode->Next;
        }

        if(readyHead==NULL && CPUstate==IDLE) {
            printf(" ");
        }

        time++;
    }

    node = (PROCESS *)malloc(sizeof(PROCESS));
    CopyNode(readyHead, node);
    Enqueue(node, &endHead, &endTail);
    printf("\n\n");

    printf("************************** Result
****************************\n\n");
    PrintProcess(&endHead);
    printf("\n  average waiting time : %.2lf average turnaround time : %.2lf
\n",
```

```
        waiting[4]/(double)processNum, turnaround[4]/(double)processNum);
}

void RR() {

    SortByArrival(&head, &tail);

    /* Process Queue */
    PROCESS *processHead = NULL, *processTail = NULL;
    PROCESS *processQueue = head, *node, *readyNode;

    int time = 0, endNum = 0, timequantum = 0;
    CPUstate = IDLE;

    /* Initialize ReadyQueue */
    readyHead = NULL;
    readyTail = NULL;

    /* Copy ProcessQueue */
    CopyQueue(&processQueue, &processHead, &processTail);
    printf("+-----------------------------------------------------------------
-+\n");
    printf("|                      Round Robin Scheduling
|\n");
    printf("+-----------------------------------------------------------------
-+\n");
    printf("                       Enter quantum: ");
    scanf("%d", &quantum);

    printf("+-----------------------------------------------------------------
-+\n");
    printf("|                  Round Robin Scheduling Report
|\n");
    printf("+-----------------------------------------------------------------
-+\n");
    printf("************************* Gantt Chart
*************************\n\n");

    while(endNum < processNum) {

        processQueue = processHead;

        /*  Insert ReadyQueue - Job Scheduling */
        while(processQueue != NULL && processQueue->ArrivalTime == time) {
            node = (PROCESS *)malloc(sizeof(PROCESS));
            CopyNode(processQueue, node);
            Enqueue(node, &readyHead, &readyTail);

            /* remove job scheduled process from process list */
```

```
            processQueue = processQueue->Next;

            /* next process */
            processHead = processHead->Next;
        }

        if(CPUstate <= 0 || timequantum == 0) {

            /* reinitiate quantum */
            timequantum = quantum;

            /* time expired - move to ready queue */
            if(CPUstate > 0) {
                node = (PROCESS *)malloc(sizeof(PROCESS));
                CopyNode(readyHead, node);
                Enqueue(node, &readyHead, &readyTail);
                readyHead = readyHead->Next;
            }
            /* done */
            else if(CPUstate == SCHEDULE) {
                CPUstate = IDLE;
                readyHead = readyHead->Next;
            }
            /* cpu scheduling */
            if(readyHead != NULL) {
                CPUstate = readyHead->ProcessID;
            }
        }

        readyNode = readyHead;
        while(readyNode != NULL) {

            /* current executing process */
            if(CPUstate == readyNode->ProcessID) {
                printf("%d", readyNode->ProcessID);
                /* end of execute */
                if(readyNode->RemainTime == 1) {
                    readyNode->Turnaround = readyNode->WaitingTime +
readyNode->ExecutingTime + 1;
                    waiting[5] += readyNode->WaitingTime;
                    turnaround[5] += readyNode->Turnaround;
                    endNum++;
                    CPUstate = SCHEDULE;

                    readyNode->ExecutingTime++;
                    readyNode->RemainTime--;

                    node = (PROCESS *)malloc(sizeof(PROCESS));
                    CopyNode(readyNode, node);
```

```
                    Enqueue(node, &endHead, &endTail);
                }
                else {
                    readyNode->ExecutingTime++;
                    readyNode->RemainTime--;
                }
                timequantum--;
            }

            /* waiting process */
            else {
                readyNode->WaitingTime++;
            }

            readyNode = readyNode->Next;

        }

        if(readyHead==NULL && CPUstate==IDLE) {
            printf(" ");
        }

        time++;
    }

    printf("\n\n");

    printf("*************************** Result
*****************************\n\n");
    PrintProcess(&endHead);
    printf("\n  average waiting time : %.2lf average turnaround time : %.2lf
\n",
        waiting[5]/(double)processNum, turnaround[5]/(double)processNum);
}

int main(void) {

    CreateProcess();
    while(1) {
    Menu();
    }

    return 0;

}
```