본 프로젝트에서는 Linux 환경에서 C를 사용해 CPU 스케줄링 시뮬레이터를 구현했다. Single processor 환경에서 적용 가능한 모델이다. 여러 스케줄링 기법들 중 FCFS, non-preemptive SJF, preemptive SJF, non-preemptive priority, preemptive priority, round robin을 포함시켰다. 무작위 숫자를 부여해 프로세스들을 생성했고, 각 알고리즘에 대해 Gantt Chart를 표시함으로써 프로세스들이 어떤 순서로 수행되는지 한눈에 볼 수 있게 했다. 또한 평균 대기 시간, 평균 반환 시간을 측정하여 각 알고리즘 간 비교가 가능하다.

1) 초기 화면

이 시뮬레이터를 실행하면 자동으로 1-10개의 프로세스가 임의로 생성된다. 이 프로세스들의 도착 시간, CPU burst time, I/O start time, I/O burst time, Priority도 임의로 부여된다. 그리고 그 아래 8개의 선택지가 나온다. 1-6 사이의 숫자를 선택하면 각 스케줄링 기법에 따라 CPU 스케줄링을 한 결과가 화면에 간트 차트가 average waiting time, average turnaround time과 함께 뜬다. 7을 선택하면 현재 생성된 프로세스들에 대해 이전에 선택지 1-6을 선택한 적이 있을 경우에만 1-6의 스케줄링 기법을 적용했을 때의 average waiting time과 average turnaround time을 보여준다. 스케줄링을 한 적이 없는 경우 average waiting time과 average turnaround time이 모두 9999로 나타난다. FCFS만 수행하고 7을 선택하면 FCFS의 average waiting time과 average turnaround time만 반영되고 나머지 스케줄링 기법에 대해서는 모두 9999로 나타난다. 8을 선택하면 이전의 결과를 다 지우고 새로운 프로세스 랜덤 개를 다시 생성한다.

2) 모듈 소개

이 시뮬레이터는 1.c, FCFS.c, Priority_NP.c, Priority_P.c, RR.c, SJF_NP.c, SJF_P.c, create_process.c, create_process.h, evaluation.c, queue.c, queue.h, simulator.h로 총 c 파일 10개와 헤더파일 3개로 구성된다. 1.c는 초기 화면을 구성하는 파일이다.

2-1) 1.c

int main(void){

//각 스케줄링 기법별 waiting time과 turnaround time을 각각 다른 변수에 저장한다. 그리고 포인터를 이용해 다른 함수에서도 값을 변화시킬 수 있도록 한다. 초깃값은 9999이다. 따라서 선택지 1-6 실행 전 7을 출력하면 9999가 나오는 것이다.

Set wt1, tt1, wt2, tt2, ..., tt6 to 9999

Set *WT1 to &wt1, *TT1 to &tt1, ...

Set n to a random number between 1-10

Create n processes //create_process.c의 Create_Process 함수를 사용한다.

Create ready queue and waiting queue which can fit at least 10 processes //queue.c의 createQueue 함수를 사용한다. 1-10개의 프로세스를 생성하므로 최소 10개

REPEAT

Show options

CASE choice of

1 : First Come First Served

2 : Shortest Job First (non-preemptive)

3 : Shortest Job First (preemptive)

4 : Priority (non-preemptive)

5 : Priority (preemptive)

6 : Round Robin

7 : evaluation

8 : create new processes

ENDCASE

}

2-2) create_process.h

Process 구조체 정의. PID, arrival time, cpu burst time, I/O start time, I/O burst time, priority, waiting time, termination time, turnaround time으로 구성된다. 이 시뮬레이터에서 생성하는 프로세스들은 I/O가 한 번 이하로만 발생한다.

2-3) create_process.c

PID는 프로세스가 생성된 순서대로 정수를 부여한다. 첫 번째 생성된 프로세스의 PID는 0이다. 도착 시간은 0에서 19 사이의 값이, CPU burst time은 2 이상 16 이하의 값이 무작위로 배정된다. I/O start time은 CPU burst가 얼마만큼 진행된 후 I/O 작업이 시작되는지를 가리키는 수다. CPU burst time보다 짧게 배정한다. I/O burst time은 I/O 작업 시간으로 0에서 4가 들어갈 수 있다. 단 I/O burst time이 0이면 I/O가 발생하지 않은 것으로 보고, I/O start time은 아무 의미도 갖지 않는다. 예를 들어 CPU burst time이 5이고 I/O start time이 2, I/O burst time이 3이라면 CPU 작업을 2 수행하고 I/O가 시작되어 I/O를 3 수행한 후 다시 나머지 CPU 작업을 3 수행하고 종료된다. Priority는 0 이상 6 이하의 값이 들어가고 priority 숫자가 작을수록 우선순위가 더 높다(중요하다). Waiting time, termination time, turnaround time은 프로세스가 생성되면 0으로 초기화된다.

Process* Create_Process(int n){

Dynamic memory allocation for Process* *process*

Initialize variables of *process*

Display values

}

2-4) queue.h

Ready queue와 waiting queue로 쓸 구조체 Queue를 선언한다. Queue의 변수로는 in, out, size, capacity와 Process가 들어가는 배열인 array가 있다. In은 enqueue되는 Process가 들어갈 칸을, out은 dequeue될 칸을 가리킨다. Size는 현재 차 있는 칸의 수를 말하고 capacity는 큐의 크기를 가리킨다.

2-5) queue.c

큐 조작에 필요한 함수들을 정의한다. Ready queue와 waiting queue가 경우에 따라 정렬을 필요로 하므로 기존 큐에는 없는 sort 함수들을 추가했다. createQueue, isFull, isEmpty, enqueue, dequeue, showQueue, sort_by_cpuburst, sort_by_priority, sort_by_ioburst가 있다.

createQueue는 capacity를 parameter로 받아 capacity 크기의 array를 만들고 in, out, size를 0으로 초기화한다.

isFull은 큐의 size와 capacity를 비교해 같으면 true를 리턴한다.

isEmpty는 size가 0이면 true를 리턴한다.

enqueue는 parameter로 큐와 넣을 item을 받아서 큐가 full인지 확인하고 아닐 경우 item을 array에 넣고 size를 증가시키고 in을 한 칸 뒤로 보낸다. 이때 circular queue의 특성을 고려하여 1만 더하는 대신에 1을 더한 후 capacity로 나눈 나머지를 새로운 in에 대입한다.

dequeue는 out이 가리키는 item을 빼는 함수인데, out을 한 칸 미루고(in에서처럼 1을 더한 후 capacity로 나눈 나머지를 대입한다) size를 줄인다.

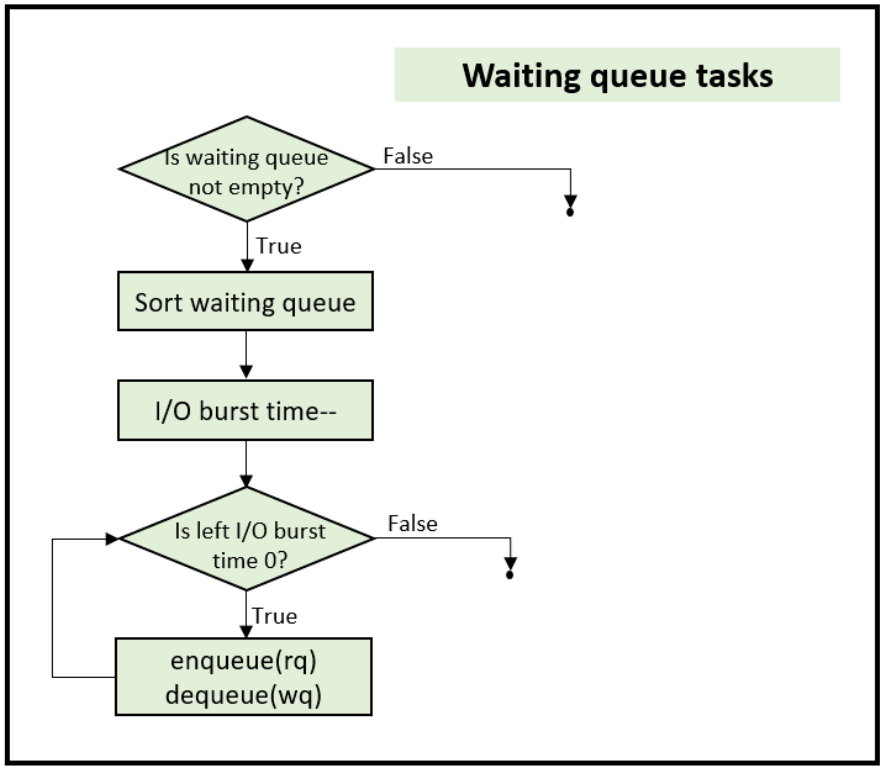
showQueue는 out부터 size 개수만큼 PID를 출력한다. 큐가 비어 있으면 비었다는 메시지를 출력한다.

sort_by_cpuburst/priority/ioburst는 각각 cpu_burst_time, priority, I/O burst 순으로 프로세스를 정렬하는데 숫자가 작은 것이 앞에 오게 정렬한다. 따라서 priority도 낮은 것이 우선순위가 더 높은 것으로 간주한다.

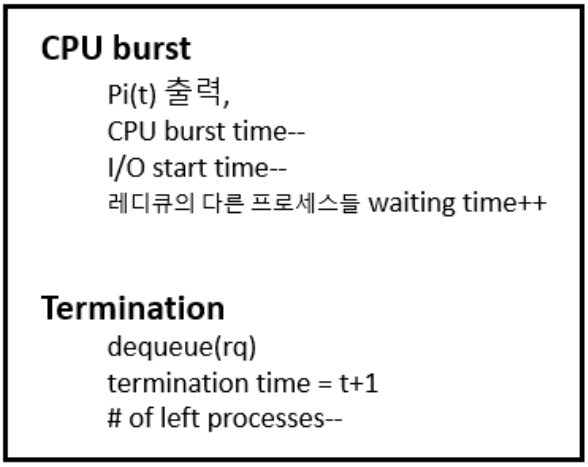
```
void sort_by_cpuburst(Queue* q){
        Create an array of size q->size
        Fill up the new array with filled parts of q->array
        Sort the new array by CPU burst time
        Overwrite q->array with the elements of the new array
}
```

sort_by_priority/ioburst도 같은 방식이다.

아래부터는 스케줄링 알고리즘을 flowchart를 사용해 소개한다. Ready queue와 waiting queue의 작업은 단위 시간이 한 번 흐를 때마다 순서대로, 독립적으로 한 번씩 실행된다. Flowchart의 waiting queue tasks는 다음을 말한다. (자세한 설명은 FCFS.c에 있다.)



또한, 알고리즘들의 flowchart에서 등장할 CPU burst와 Termination은 다음을 의미한다.



CPU burst는 Pi(t) 출력, CPU burst time--, I/O start time--, ready queue의 다른 프로세스들의

waiting time 1 증가를 말한다. Termination은 해당 프로세스가 완전히 종료되었음을 의미하는데, 그 프로세스를 ready queue에서 빼고, termination time에 t+1을 저장하고, 남은 프로세스의 개수를 하나 감소시키는 것을 말한다.

모든 프로세스는 도착하면 먼저 I/O burst time이 0인지 확인한다. I/O burst time이 0인 경우는 I/O가 발생하지 않는 것으로 간주하고 I/O start time을 -1로 변경하여 I/O가 발생할 시점인지 확인하는 if문을 피해가도록 조치한다.

2-6) FCFS.c

프로세스를 도착한 순서대로 ready queue에 줄 세운다. 이제 단위 시간에 한 번씩 아래 작업을 반복한다. 단위 시간이 흘러갈 동안 Ready queue에서의 작업이 있을 것이고 이어서 waiting queue에서의 작업이 있을 것이다.

Ready queue에 작업이 있다면 큐의 맨 앞 프로세스가 I/O 작업을 시작할 시점인지 확인한다.

I/O start time이 0이라면 I/O 작업을 할 수 있도록 ready queue에서 dequeue한다. 이 시뮬레이터에서 I/O 작업은 한 번만 발생하므로 다음에 이 단계에서 I/O start time이 0인지 묻는 것을 지나치기 위해 I/O start time을 음수로 변경한다. 이 프로세스가 waiting queue로 빠진 후 ready queue에 여전히 실행할 프로세스가 남아 있는지 확인한다. 프로세스가 있다면 CPU를 할당하고 없다면 Pidle(t)를 출력하고 waiting queue task로 넘어간다. 프로세스가 있어서 CPU를 할당한 경우 CPU 작업을 했다고 Pi(t)를 출력하고, CPU burst time을 1 감소시킨다. I/O 작업을 해야 할 시간도 가까워지므로 I/O start time도 1 감소시킨다. 이때 ready queue 안의 다른 프로세스들은 한 단위시간만큼 ready queue에서 기다린 것이므로 모두 waiting time을 1씩 증가시킨다. 프로세스가 CPU 작업을 한 뒤 남은 CPU burst time을 확인하여 0이면 종료하고 아니면 waiting queue task를 한다.

만약 I/O start time이 0이 아니었다면 ready queue의 첫 프로세스를 waiting queue로 보내지 않을 뿐 위와 같은 작업이 실행된다.

Ready queue가 비어 있었다면 프로세스가 다 종료되어 빈 것인지 I/O 작업 중이라 빈 것인지 확인하여 아무것도 출력하지 않거나 Pidle(t)를 출력한다.

이렇게 ready queue의 작업들이 완료되면 같은 단위 시간 내에 waiting queue의 작업들도 수행해줘야 한다. Waiting queue task에 프로세스가 하나 이상 있다면 남은 I/O burst 순으로 정렬한다. 늦게 waiting queue에 들어온 프로세스도 I/O burst가 끝나면 dequeue되어야 하기 때문에 I/O burst 순으로 정렬해둔다. 한 단위시간이 흘러가는 만큼 프로세스들의 I/O 작업도 이루어지고 있을 것이므로 waiting queue의 모든 프로세스들의 I/O burst time을 1씩 감소시킨다. I/O burst time이 0인 모든 프로세스들을 dequeue시키기 위해 while loop을 돌린다. 다

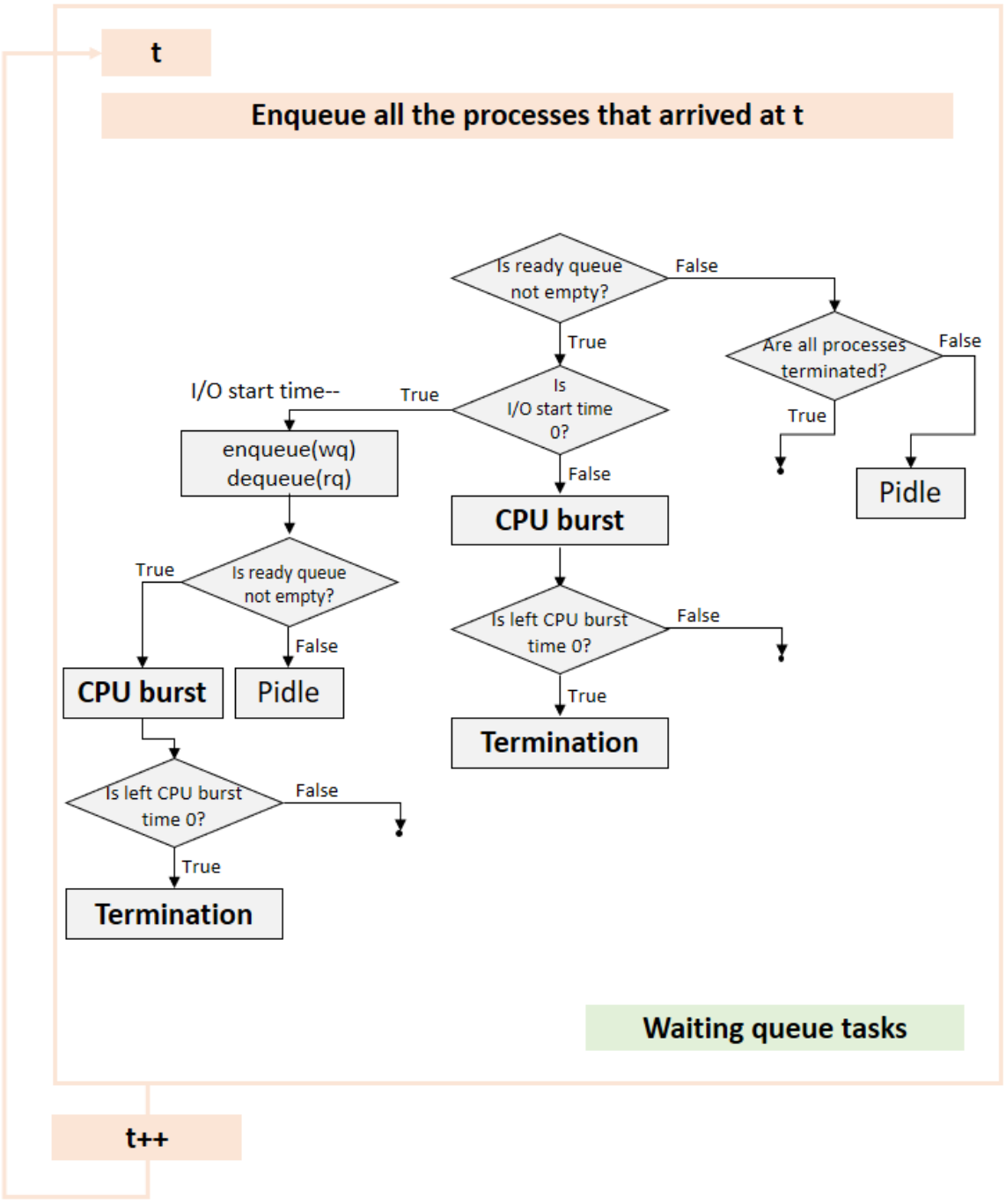
dequeue 시키고 ready queue로 보낸 후 더 이상 I/O burst time이 0인 프로세스가 없으면 waiting queue 작업을 종료하고 다음 단위시간의 작업들을 flowchart 위부터 다시 실행한다.

Waiting queue tasks에서 I/O burst가 끝나면 바로 ready queue로 enqueue 시키므로 두 프로세스가 ready queue에 t(시간)에 도착한다고 할지라도 flowchart 맨 위에서 enqueue 시키는 새로 도착한 프로세스보다 I/O를 마친 프로세스가 ready queue의 줄에서 앞서 있다.

시간을 1씩 증가시키는 for loop을 다 돌면 프로세스들이 순서를 따라 실행된 후 모두 종료되어 있을 것이다. 알고리즘을 수행하면서 각 프로세스는 waiting time과 termination time을 저장해왔다. 프로세스가 종료된 시각인 termination time에서 arrival time을 빼서 각 프로세스의 turnaround time을 구한다. 그리고 모든 프로세스의 waiting time의 평균을 구해서 *WT(숫자)에 저장한다. *WT1은 FCFS, *WT2는 SJF_NP, *WT3은 SJF_P 등 *WT(숫자)는 프로세스들의 평균 waiting time을 저장할 주소이다. 포인터를 사용해 저장함으로써 이후 evaluation 모듈을 수행할 때도 여기 저장된 값을 불러올 수 있도록 한다. Turnaround time도 마찬가지로 평균을 구해서 *TT(숫자)에 저장한다. Evaluation을 할 때뿐만 아니라 알고리즘을 실행한 후 Gantt chart 아래에서도 방금 실행한 알고리즘의 average waiting time과 average turnaround time을 보여준다.

```
3 processes are created.

[Process ID, Arrival time, CPU Burst time, I/O Start time, I/O Burst time, Prior
ity]
[0 6 10 5 0 1]
[1 13 11 6 3 2]
[2 18 14 10 2 5]



- - - - - - - - - - - - - - - - - - - - -
1. FCFS       2. Non-Preemptive SJF       3. Preemptive SJF
4. Non-Preemptive Priority       5. Preemptive Priority       6. Round Robin
7. Evaluation(Only after you did 1~6, you can see the result.)
8. Create new processes

Enter a number : 1

************************************
        FCFS scheduling
************************************

Pidle(0) Pidle(1) Pidle(2) Pidle(3) Pidle(4) Pidle(5) P0(6) P0(7) P0(8) P0(9) P0
(10) P0(11) P0(12) P0(13) P0(14) P0(15) P1(16) P1(17) P1(18) P1(19) P1(20) P1(21
) P2(22) P2(23) P2(24) P2(25) P2(26) P2(27) P2(28) P2(29) P2(30) P2(31) P1(32) P
1(33) P1(34) P1(35) P1(36) P2(37) P2(38) P2(39) P2(40)

avg_waiting_time : 5.66667
avg_turnaround_time : 19
```

FCFS



2-7) Shortest Job First (non-preemptive)

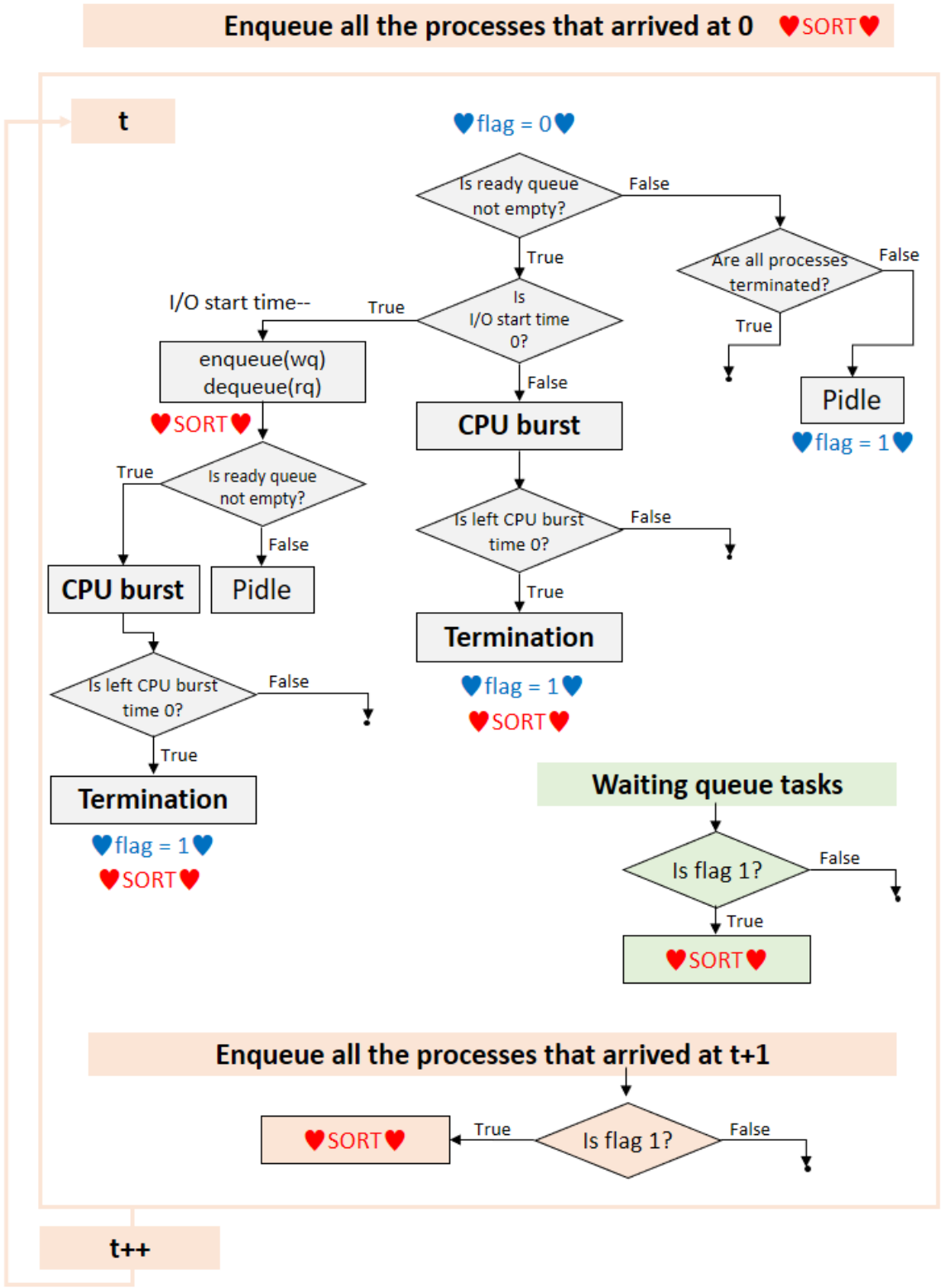
SJF 알고리즘도 ready queue가 비었는지, I/O start time이 0인지 등을 체크해야 하기 때문에 FCFS와 flowchart 모양은 비슷하게 그려진다. 다만 shortest job first 알고리즘에서는 남은 CPU burst time이 적은 게 먼저 실행되어야 하고, 다른 프로세스가 수행하는 중에는 순서를 바꾸면 안 되므로 적시에 남은 CPU burst time 순으로 ready queue의 프로세스들을 정렬해야 해야 한

다. 정렬이 필요한 시점은 **context switch**가 발생하는 때이다. 즉, 실행하던 프로세스가 I/O 작업을 하러 갔을 때, 종료되었을 때, 빈 ready queue에 프로세스들이 동시에 도착했을 때 그들 중 가장 CPU burst time이 짧은 것부터 실행할 수 있도록 정렬해야 한다. Ready queue의 첫 프로세스가 I/O 작업을 하러 가는 경우는 해당 프로세스가 ready queue에서 빠진 후 바로 ready queue를 정렬한다. 정렬된 큐에서 가장 CPU burst가 짧은 프로세스가 CPU를 할당 받는다. Termination이 발생하거나 큐가 아예 빈 경우는 상황이 약간 다르다. 다음 단위시간이 시작되기 전에 waiting queue에서 I/O를 끝낸 프로세스들, 그리고 새로 도착한 프로세스들이 ready queue에 유입되기 때문이다. 이러한 때를 체크하기 위해 flag라는 변수를 도입했다. 위와 같은 상황에 flag를 1로 바꾼다. 어떤 프로세스가 종료된 후 ready queue에 새로 들어오는 프로세스가 있으면, 즉 그때의 flag가 1이면 ready queue를 새로 정렬한다. 이 flag는 새로운 단위시간마다 0으로 초기화한다.

Flowchart의 SORT는 ready queue를 CPU burst time이 적은 순으로 정렬하는 것을 말한다.

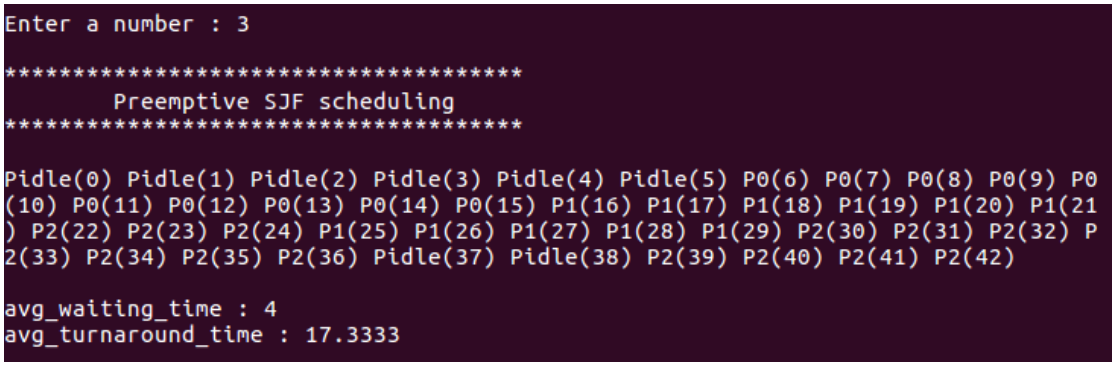
```
Enter a number : 4

************************************
        Non-Preemptive Priority scheduling
************************************

Pidle(0) Pidle(1) Pidle(2) Pidle(3) Pidle(4) Pidle(5) P0(6) P0(7) P0(8) P0(9) P0
(10) P0(11) P0(12) P0(13) P0(14) P0(15) P1(16) P1(17) P1(18) P1(19) P1(20) P1(21
) P2(22) P2(23) P2(24) P2(25) P2(26) P2(27) P2(28) P2(29) P2(30) P2(31) P1(32) P
1(33) P1(34) P1(35) P1(36) P2(37) P2(38) P2(39) P2(40)

avg_waiting_time : 5.66667
avg_turnaround_time : 19
```

SJF_NP

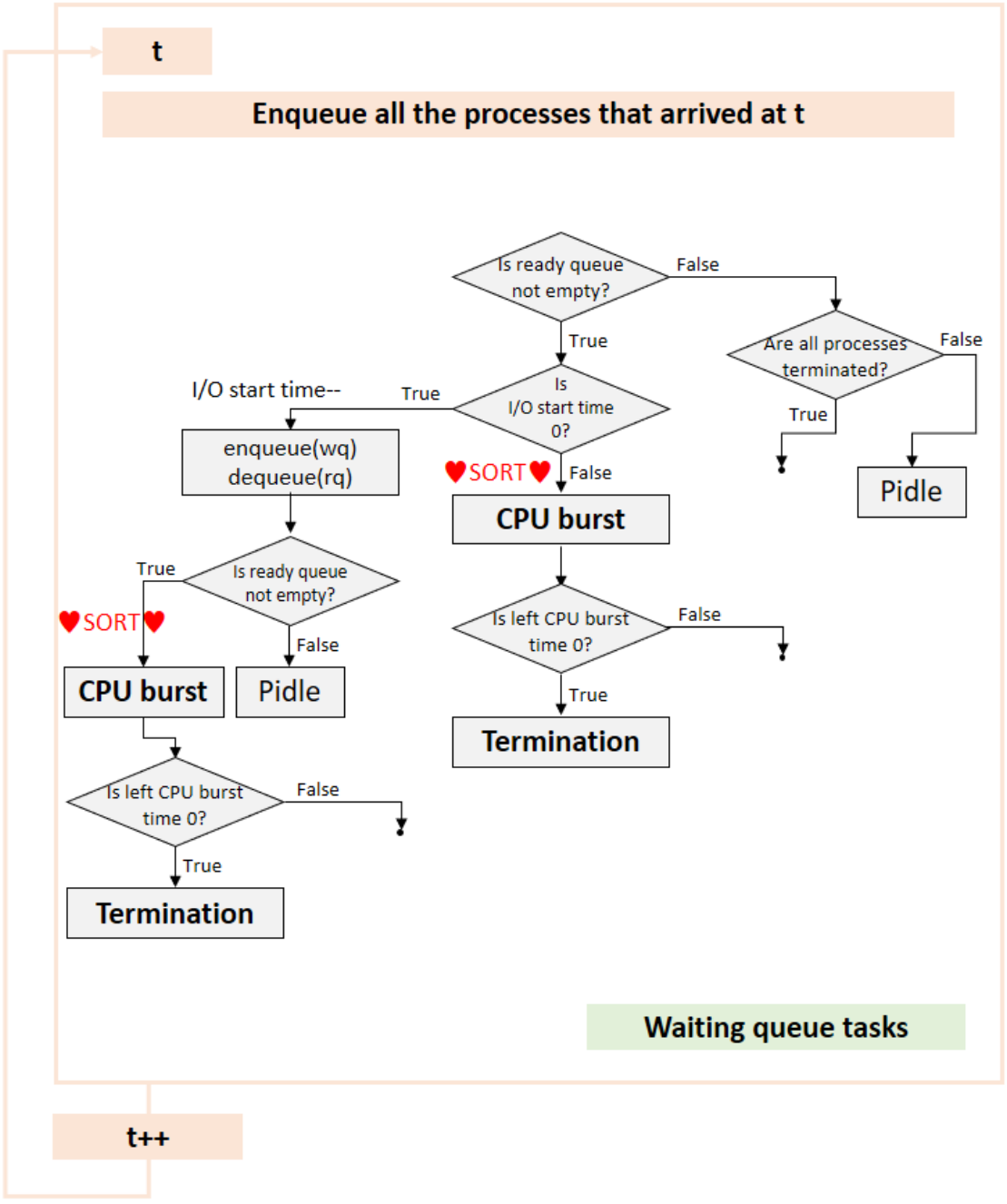

Enqueue all the processes that arrived at 0
♥SORT♥
t
♥flag = 0♥
Is ready queue not empty?
False
True
Are all processes terminated?
False
True
Pidle
♥flag = 1♥
Is I/O start time 0?
True
I/O start time--
False
enqueue(wq) dequeue(rq)
♥SORT♥
CPU burst
Is ready queue not empty?
True
False
CPU burst
Pidle
Is left CPU burst time 0?
False
True
Termination
♥flag = 1♥
♥SORT♥
Is left CPU burst time 0?
False
True
Termination
♥flag = 1♥
♥SORT♥
Waiting queue tasks
Is flag 1?
False
True
♥SORT♥
Enqueue all the processes that arrived at t+1
♥SORT♥
True
Is flag 1?
False
t++

2-8) Shortest Job First (preemptive)

SJF preemptive 알고리즘은 정렬하는 시점이 non-preemptive 방식과 다르다. 선점 가능하므로 이전에 실행하던 프로세스가 끝났는지 여부와 상관없이 CPU 할당을 할 때마다 남은 CPU burst time 순으로 정렬해서 실행한다.



SJF_P



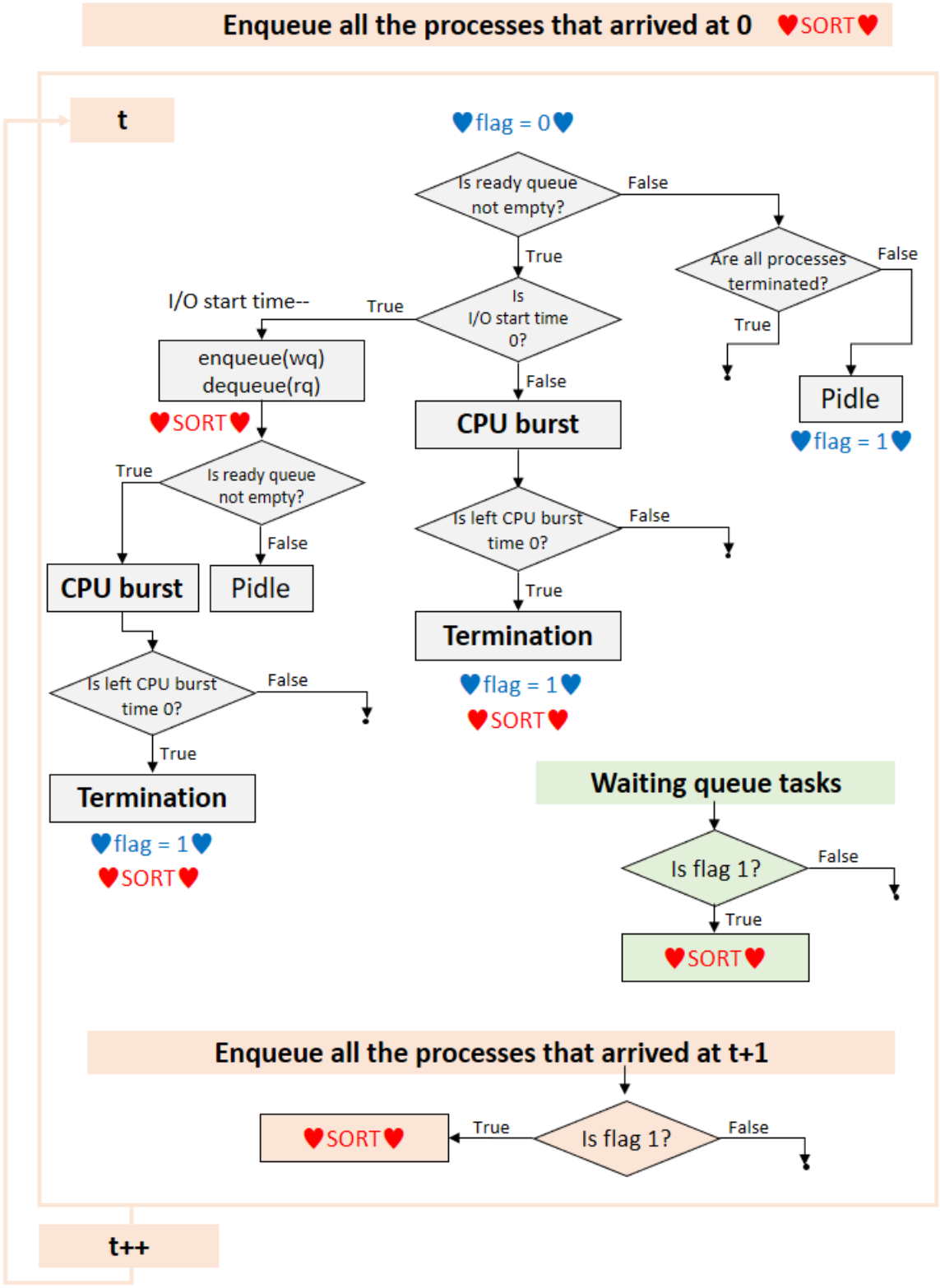
Waiting queue tasks에서 I/O를 마친 프로세스가 enqueue되므로, 두 프로세스가 ready queue에 똑같이 t에 도착한다면 새로 도착한 프로세스보다 I/O를 마치고 온 프로세스가 먼저 수행된다.

2-9) Priority (non-preemptive)

Shortest Job First (non-preemptive)와 기본적인 구조가 같다. 그때의 SORT가 CPU burst time 기준 정렬이었다면, Priority (non-preemptive) 방식에서는 priority를 기준으로 정렬한다.

```
Enter a number : 4

************************************
        Non-Preemptive Priority scheduling
************************************

Pidle(0) Pidle(1) Pidle(2) Pidle(3) Pidle(4) Pidle(5) P0(6) P0(7) P0(8) P0(9) P0
(10) P0(11) P0(12) P0(13) P0(14) P0(15) P1(16) P1(17) P1(18) P1(19) P1(20) P1(21
) P2(22) P2(23) P2(24) P2(25) P2(26) P2(27) P2(28) P2(29) P2(30) P2(31) P1(32) P
1(33) P1(34) P1(35) P1(36) P2(37) P2(38) P2(39) P2(40)

avg_waiting_time : 5.66667
avg_turnaround_time : 19
```

Priority_NP

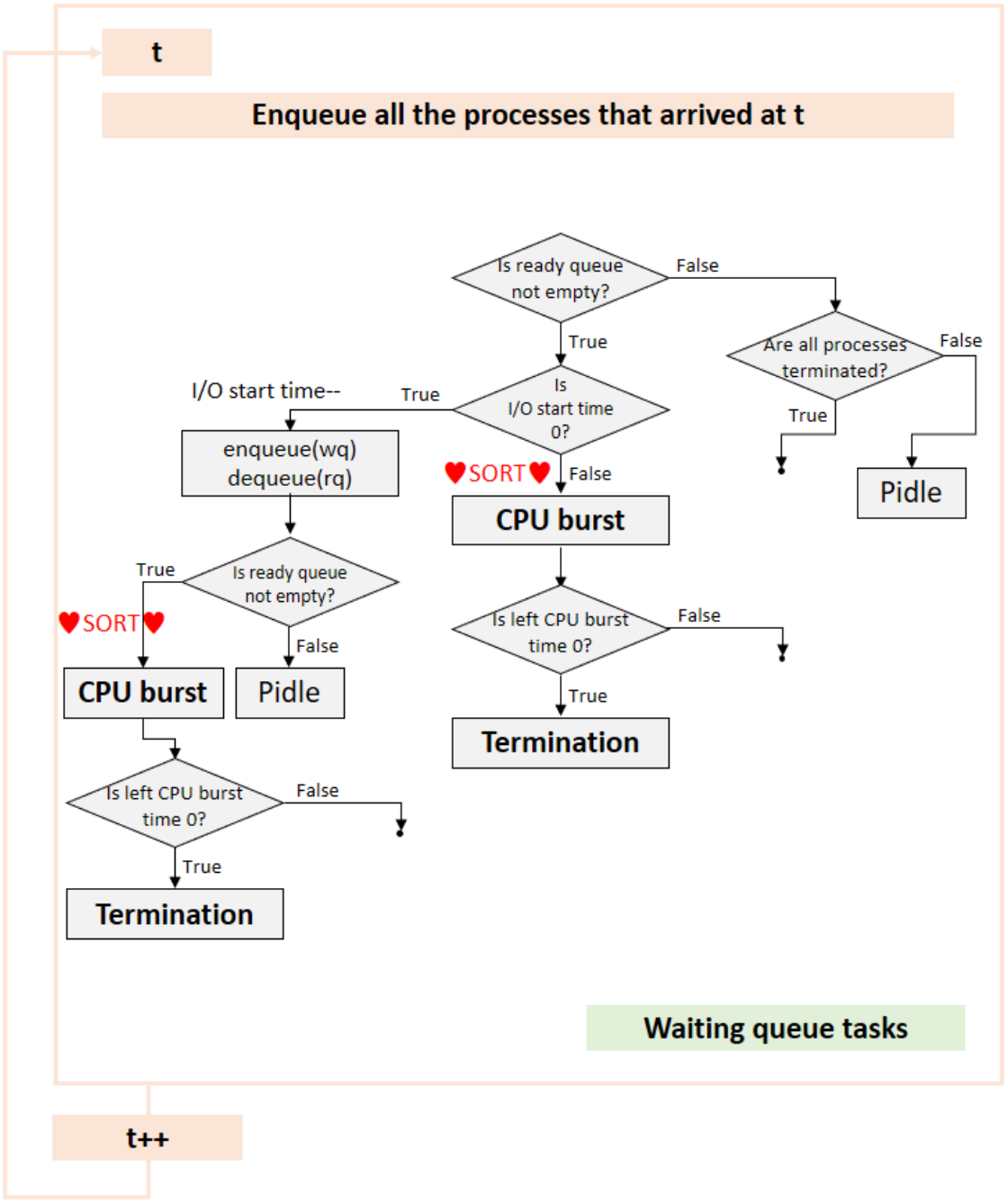


2-10) Priority (preemptive)

이것 역시 Shortest Job First (preemptive) 방식과 기본적인 구조가 같다. 여기서의 SORT는 priority를 기준으로 한 정렬이다.

```
Enter a number : 5

************************************
        Preemptive Priority scheduling
************************************

Pidle(0) Pidle(1) Pidle(2) Pidle(3) Pidle(4) Pidle(5) P0(6) P0(7) P0(8) P0(9) P0
(10) P0(11) P0(12) P0(13) P0(14) P0(15) P1(16) P1(17) P1(18) P1(19) P1(20) P1(21
) P2(22) P2(23) P2(24) P1(25) P1(26) P1(27) P1(28) P1(29) P2(30) P2(31) P2(32) P
2(33) P2(34) P2(35) P2(36) Pidle(37) Pidle(38) P2(39) P2(40) P2(41) P2(42)

avg_waiting_time : 4
avg_turnaround_time : 17.3333
```

Priority_P



Waiting queue tasks에서 I/O를 마친 프로세스가 enqueue되므로, 두 프로세스가 ready queue에 똑같이 t에 도착한다면 새로 도착한 프로세스보다 I/O를 마치고 온 프로세스가 먼저 수행된다.

2-11) Round Robin

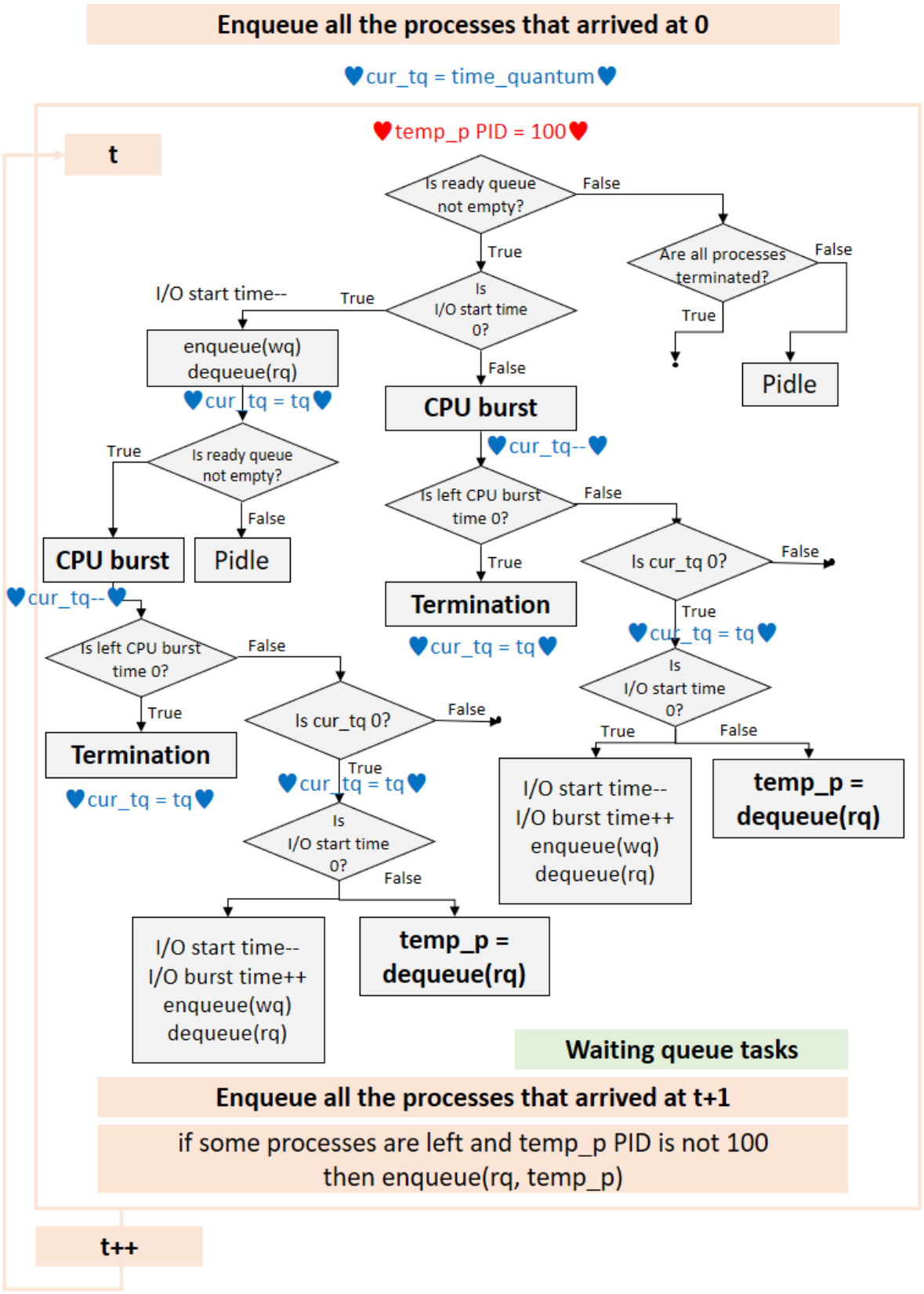
Round Robin 기법을 선택하면 time quantum을 입력하라는 메시지가 나온다. 입력한 time

quantum으로 round robin 스케줄링을 실행한다. Flowchart에서 time_quantum은 입력 받은 time quantum이고 cur_tq는 남은 time quantum을 의미한다. Time quantum이 종료된 프로세스를 다시 ready queue 뒤로 보낼 때 잠시 저장해둘 곳을 Process 구조체를 사용해 temp_p로 선언한다. 이 temp_p는 처음 선언될 때 다른 정보들은 0으로 하고 PID만 100으로 저장되었다. Time quantum이 종료된 프로세스가 temp_p에 저장되지 않은 경우 PID가 100으로 남아있을 것이고 아니면 PID가 저장된 프로세스의 PID로 바뀔 것이다. 프로세스는 10개까지 생성될 수 있으므로 이때의 PID는 0~9 사이이다.

이전까지의 flowchart에 cur_tq가 0인지 확인하는 과정이 들어가서 flowchart가 좀 더 복잡해졌다. Cur_tq가 0일 때마다 dequeue시켜야 하는데 이때 또 확인해야 할 것이 I/O start time이 0인지 여부이다. 자칫하면 I/O가 발생해야 할 시점에 ready queue 뒤로 가느라 waiting queue로 갈 시점을 놓칠 수도 있다. Cur_tq가 0인데 I/O start time은 0이 아닐 경우 dequeue시키고 temp_p에 잠시 저장하여 waiting queue task를 마치고 t+1에 유입될 프로세스들을 다 enqueue시킨 후 ready queue에 다시 줄을 세운다. 따라서 어떤 프로세스들이 t라는 시점에 모두 ready queue에 도착한다면 I/O를 마치고 돌아온 것, 새로 도착한 프로세스, time quantum이 다 되어 다시 줄 선 프로세스 순으로 줄을 서게 된다.

Cur_tq를 기존 time quantum으로 변경해야 하는 시점은 처음에 프로세스들을 생성했을 때, ready queue 맨 앞에 새로운 프로세스가 자리했을 때, 프로세스가 종료되었을 때이다. 그리고 cur_tq가 0이어서 잠시 dequeue되는 프로세스들이 있을 텐데 이때 빠져나갈 때 cur_tq를 기존 time quantum으로 변경해 다음 실행할 프로세스가 문제 없이 실행할 수 있도록 한다. Cur_tq는 CPU burst를 실행할 때마다 1씩 감소시킨다. 한 단위시간 당 한 번의 CPU burst만 발생하므로 cur_tq가 줄어드는 것도 한 번만 발생, temp_p에 값이 저장되는 횟수도 한 번 이하이다. 그리고 새로운 단위시간에는 다시 temp_p의 PID를 100으로 초기화한다.

```
Enter a number : 6

************************************
        RR scheduling
************************************

Enter time quantum : 3
Pidle(0) Pidle(1) Pidle(2) Pidle(3) Pidle(4) Pidle(5) P0(6) P0(7) P0(8) P0(9) P0
(10) P0(11) P0(12) P0(13) P0(14) P1(15) P1(16) P1(17) P0(18) P2(19) P2(20) P2(21
) P1(22) P1(23) P1(24) P2(25) P2(26) P2(27) P1(28) P1(29) P1(30) P2(31) P2(32) P
2(33) P1(34) P1(35) P2(36) Pidle(37) Pidle(38) P2(39) P2(40) P2(41) P2(42)

avg_waiting_time : 7
avg_turnaround_time : 20.3333
```


Enqueue all the processes that arrived at 0
♥cur_tq = time_quantum♥
♥temp_p PID = 100♥
t
Is ready queue not empty?
False
True
Are all processes terminated?
False
True
Pidle
I/O start time--
True
Is I/O start time 0?
enqueue(wq)
dequeue(rq)
♥cur_tq = tq♥
False
CPU burst
♥cur_tq--♥
True
Is ready queue not empty?
False
CPU burst
Pidle
Is left CPU burst time 0?
False
True
Termination
♥cur_tq = tq♥
Is cur_tq 0?
False
True
♥cur_tq = tq♥
Is I/O start time 0?
True
False
I/O start time--
I/O burst time++
enqueue(wq)
dequeue(rq)
temp_p = dequeue(rq)
♥cur_tq--♥
Is left CPU burst time 0?
False
True
Termination
♥cur_tq = tq♥
Is cur_tq 0?
False
True
♥cur_tq = tq♥
Is I/O start time 0?
False
I/O start time--
I/O burst time++
enqueue(wq)
dequeue(rq)
temp_p = dequeue(rq)
Waiting queue tasks
Enqueue all the processes that arrived at t+1
if some processes are left and temp_p PID is not 100
then enqueue(rq, temp_p)
t++

2-12) Evaluation

각 알고리즘을 실행하면서 average waiting time과 average turnaround time을 *WT(숫자)와 *TT(숫자)에 저장해두기 때문에 각 알고리즘을 실행한 후 evaluation을 하면 아래와 같은 화면이 나온다.

```
Enter a number : 7

*************************************
        Evaluation
*************************************

                      FCFS     SJF_NP   SJF_P    Pri_NP   Pri_P     RR
avg_waiting_time :    22.75    15.38    14.38    20.38    21.62    26.62
avg_turnaround_time : 31.62    24.25    23.25    29.25    30.50    35.50
```

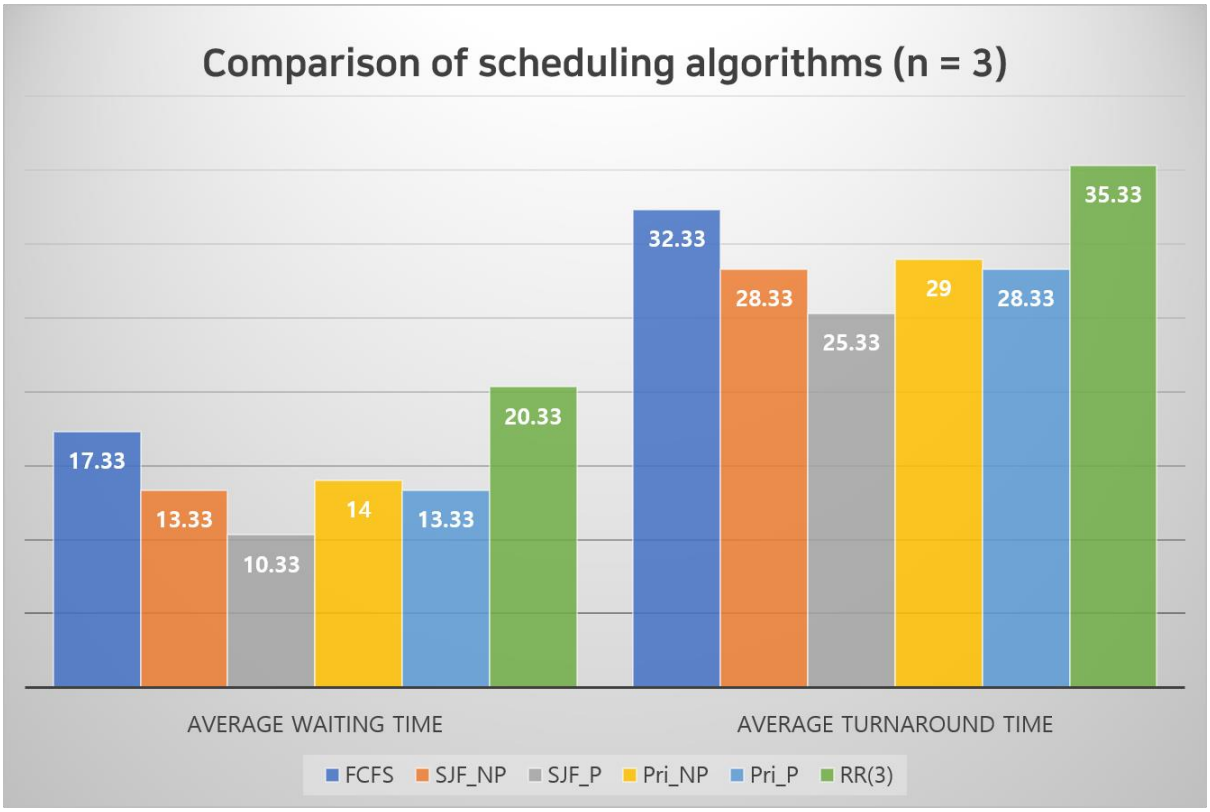
만약 어떤 알고리즘을 실행하지 않고서 evaluation을 하면 *WT(숫자), *TT(숫자)의 초기값이 9999이므로 average waiting time과 average turnaround time이 9999로 뜬다.

3) 알고리즘들 간 비교 / 분석

같은 프로세스들을 생성했을 때 각 알고리즘별 average waiting time과 average turnaround time을 분석해보았다. 모든 경우 Round robin 방식의 time quantum은 3으로 했다.

프로세스가 3개가 다음과 같이 생성되었을 때 각 알고리즘들의 average waiting time과 average turnaround time 결과이다.

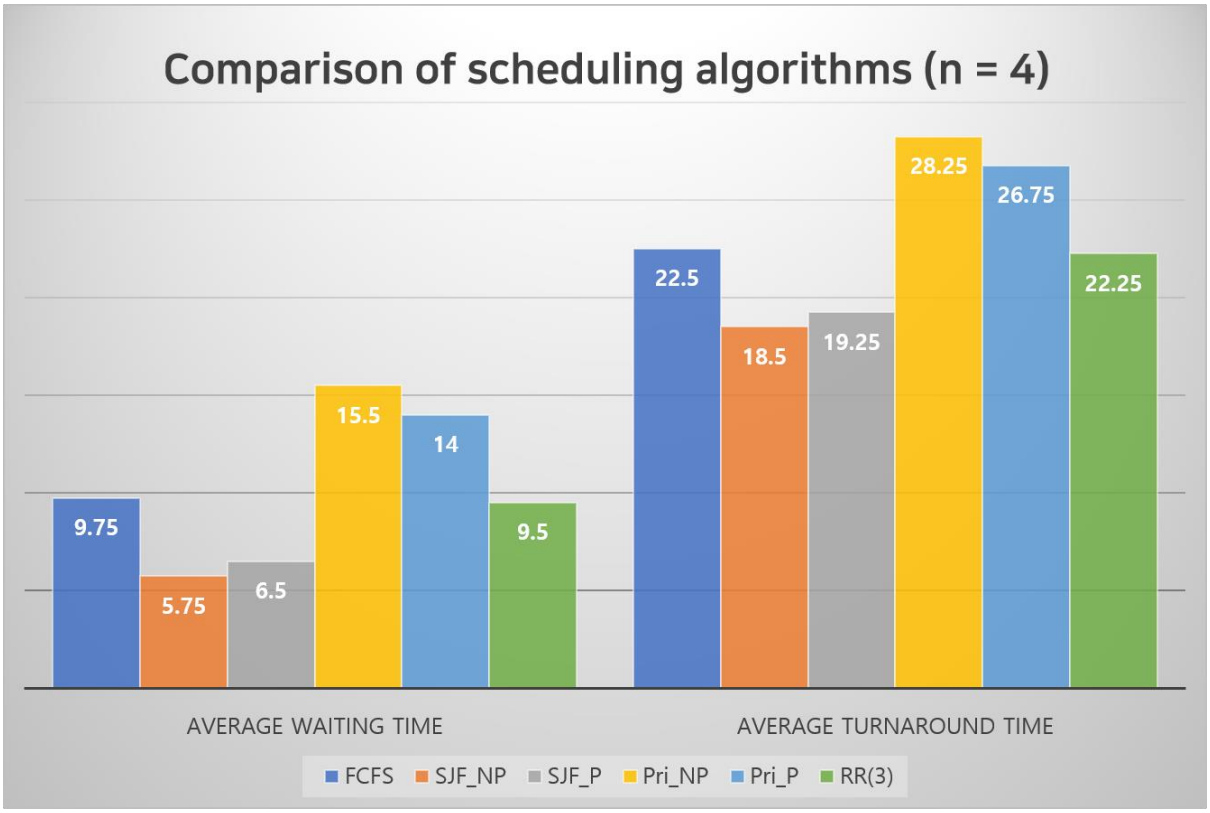
PID	Arrival time	CPU burst time	I/O start time	I/O burst time	Priority
P0	5	15	14	1	3
P1	10	14	3	2	6
P2	10	11	10	2	6



SJF_P < SJF_NP = Pri_P < Pri_NP < FCFS < RR

프로세스가 4개가 다음과 같이 생성되었을 때 각 알고리즘들의 average waiting time과 average turnaround time 결과이다.

PID	Arrival time	CPU burst time	I/O start time	I/O burst time	Priority
P0	5	3	1	3	4
P1	11	6	2	2	0
P2	12	16	7	4	3
P3	9	13	3	4	3

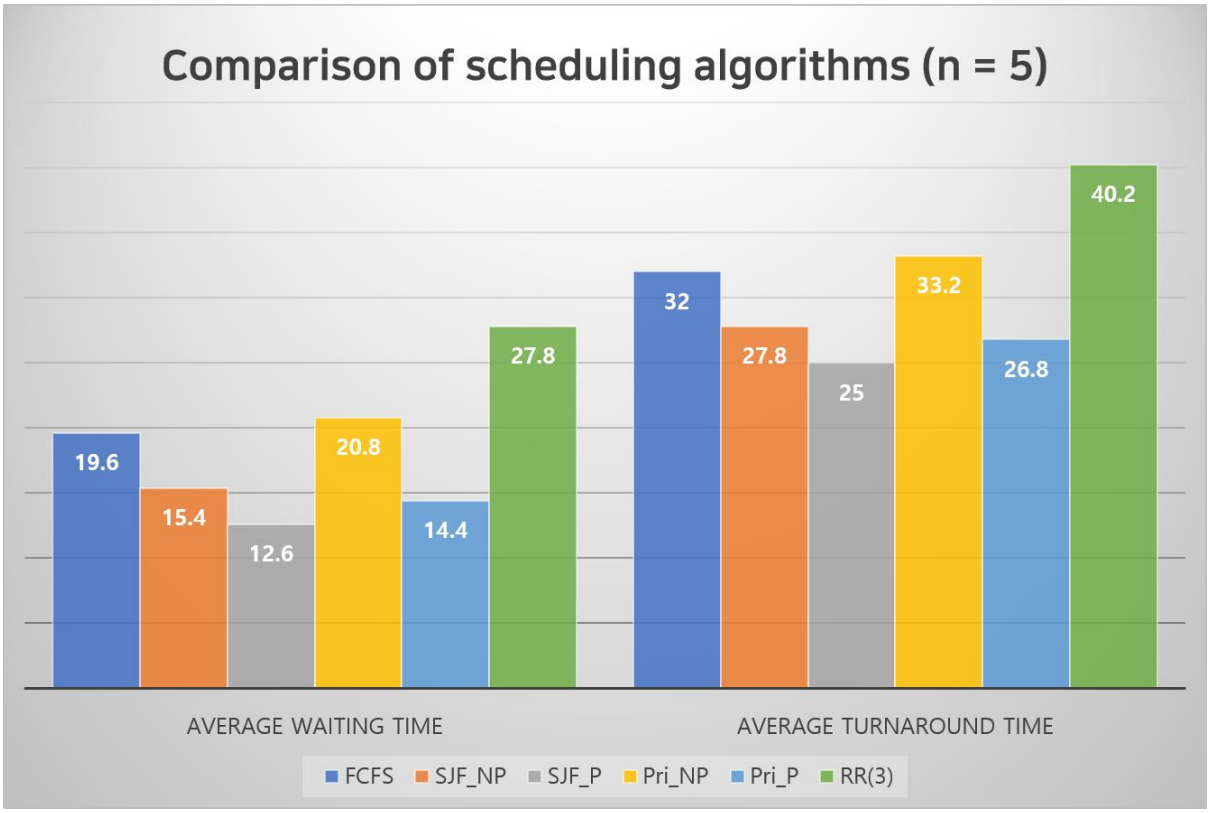


SJF_NP < SJF_P < RR < FCFS < Pri_P < Pri_NP

CPU 작업 시간이 상대적으로 긴 P2와 P3이 CPU 작업 시간이 3밖에 안 되는 P0보다 priority가 높아서 priority 알고리즘들에서 average waiting time과 average turnaround time이 높게 나온 것으로 보인다. (P0는 일찍 도착하지만 I/O를 실행하고 오기 때문에 다른 프로세스들에 의해 막히게 된다.)

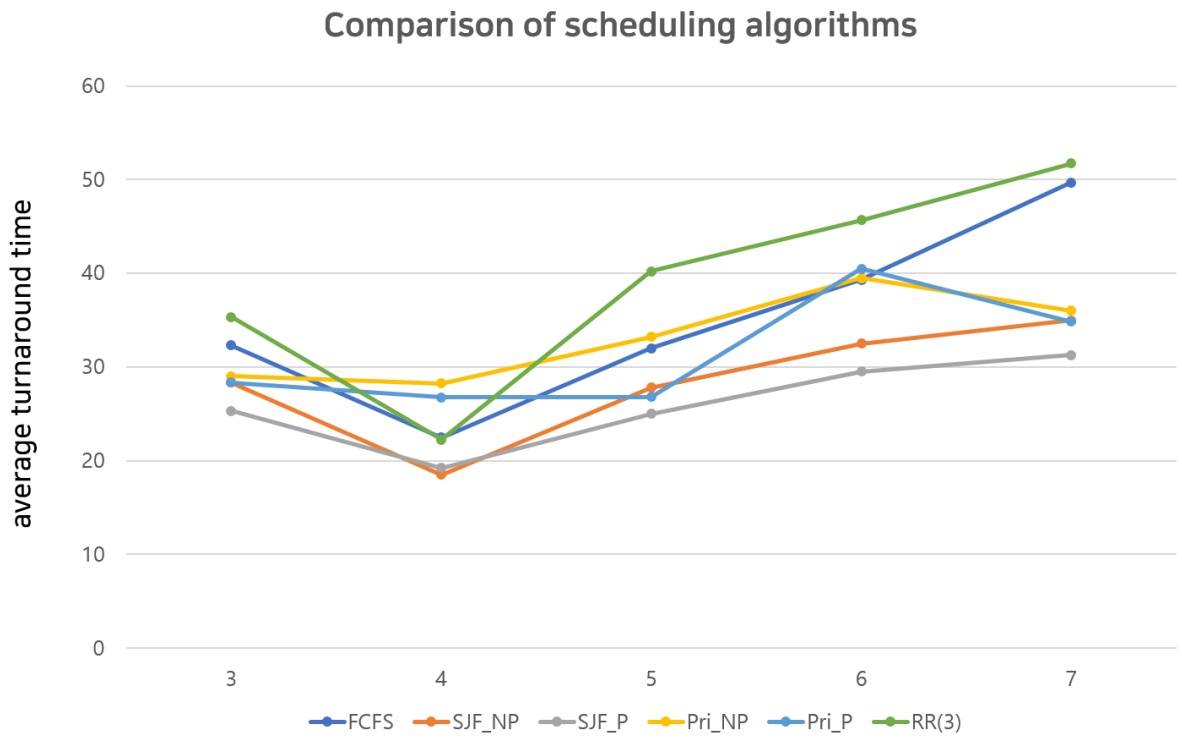
프로세스가 5개가 다음과 같이 생성되었을 때 각 알고리즘들의 average waiting time과 average turnaround time 결과이다.

PID	Arrival time	CPU burst time	I/O start time	I/O burst time	Priority
P0	4	6	3	0	3
P1	3	15	2	2	3
P2	10	9	4	1	0
P3	0	16	12	3	6
P4	19	10	5	0	4



SJF_P < PRI_P < SJF_NP < FCFS < PRI_NP < RR

다음은 프로세스를 6개, 7개 생성해 나온 결과를 위의 결과들과 함께 그래프에 나타낸 것이다.



프로세스 수, 그리고 CPU 작업 시간, I/O burst 등 프로세스 정보들에 따라 어떤 알고리즘이 더 낮은 average turnaround time을 가지는지는 달라진다. 하지만 Shortest Job First 알고리즘이, 특히 preemptive 방식이 확실히 가장 효율적으로 프로세스들을 처리하고 있다는 것을 확인할 수 있다. 짧은 프로세스부터 수행함으로써 짧은 프로세스가 긴 프로세스 뒤에서 한없이 기다리고 있는 현상을 막을 수 있기 때문에 waiting time이 전체적으로 줄어든다. Priority 방식은 중요한 프로세스를 먼저 처리하므로 실제 상황에서는 의미가 있겠지만, 여기서는 확인이 어렵다. Response time이 가장 좋은 것으로 알려진 Round robin 방식은 위 프로세스들의 경우에서 가장 안 좋은 성능을 보였다.