CPU Scheduling Simulator

2022100052 이정희

목차

1. 서론

- CPU Scheduler의 개념

CPU Scheduler는 process가 CPU를 사용할 수 있도록 Scheduling하는 작업을 의미한다. 여러 process들이 존재할 때, 각 process들이 동시에 CPU를 할당 받고 실행될 수 없기 때문에 Operating System은 어떻게 CPU를 할당해야 하는지 결정해야 한다. 이를 통해 CPU Utilization을 최대화하고 system의 성능을 향상시킨다.

CPU Scheduler에는 여러 Algorithm이 존재하는데 그 중 이번 project에서는 FCFS(First-Come, First-Served), SJF(Shortest Job First), RR(Round Robin), Priority, Preemptive SJF, Preemptive Priority 6개를 구현했다.

FCFS(First-Come, First-Served)는 process가 Ready Queue에 도착한 순서대로 CPU를 할당 받는 구조로 되어 있다. 즉, FIFO(First-In, First-Out) 구조로 되어 있는 Queue와 매우 흡사한 모습을 보인다. FCFS는 구현이 간단하다는 장점이 있지만 Convoy Effect로 인해 Waiting Time이 길어질 수 있다는 단점이 있다.

SJF(Shortest Job First)는 가장 짧은 CPU Burst를 가진 process가 CPU를 할당 받는 구조로 되어 있다. SJF의 경우 Average Waiting Time을 최소화할 수 있다는 장점이 있지만 CPU Burst Time을 미리 알 수 있는 경우가 거의 없기 때문에 현실적으로 구현아기 어렵다는 단점이 있다.

Priority는 각 process에게 Priority를 제공해주고 Priority가 높은 process에게 CPU를 제공하는 구조로 되어 있다. 이는 중요한 작업을 우선적으로 처리할 수 있다는 장점이 있지만 Priority가 낮은 process는 무한정 대기를 하는 Starvation이 발생할 수 있다는 단점을 가지고 있다.

RR(Round Robin)은 각 process에게 동일한 시간 할당량(time quantum)을 설정하고 그 시간동안 작동하도록 설계된 구조이다. RR은 응답시간이 짧아지고 공평하게 모든 process가 CPU를 할당 받을 수 있다는 장점이 있지만 time quantum을 잘못 설정하면 Algorithm이 이상하게 변질될 수 있다는 단점이 있다(time quantum이 너무 짧으면 Context Switching이 많아지고, 너무 길면 response time이 길어질 수 있다.)

Preemptive SJF와 Preemptive Priority는 원래 Algorithm에서 선점형의 특성을 추가한 Algorithm 이다. 즉, 특정 process가 Ready Queue에 들어왔을 때, 그 process가 가장 먼저 CPU를 할당 받아야 한다면 현재 CPU를 점유하고 있는 process에게서 CPU를 빼앗아서 점유하는 방식으로 되어 있다. 이는 원래 Algorithm보다 더욱 효율적으로 process들에게 CPU를 제공할 수 있다는 장점이 있지만 구현하기 어렵다는 단점을 가지고 있다.

- CPU Scheduling Simulator 요약 정리

이 project에서 CPU Scheduling Simulator는 random으로 process 5개를 받고(pid를 제외하고 CPU Burst Time, IO Burst Time, Arrival Time, Priority, IO Burst Time은 random으로 받는다.)

이후 FCFS, SJF, Priority, RR, Preemptive SJF, Preemptive Priority순으로 Algorithm을 작동시켜서 각 Algorithm에서 나온 Average Waiting Time, Average Turnaround Time, CPU Utilization을 출력시킨 뒤 마지막으로 Average Waiting Time, Average Turnaround Time, CPU Utilization의 순위를 매겨서 출력시키도록 만들었다.

2. 본론

- CPU Scheduling Simulator 구조 설명

먼저 Create_Process 함수를 통해 random으로 process 5개를 만든 이후 Print_Process 함수를 통해 각 Process의 특징들을 출력했다. 이후 Schedule 함수를 통해 CPU Scheduler를 실행했다(여기서 각 process들은 1번의 IO Burst를 반드시 수행하도록 코드를 작성했고 IO_Burst_Timing은 CPU를 몇 초 할당 받고 난 뒤 IO Burst가 일어나는지 알려주는 변수이다.)

Schedule 함수는 FCFS, SJF, Priority, RR, P_SJF, P_Priority 6가지의 함수를 순차적으로 실행하도록 만들었다. 각 algorithm들은 CPU Schedule을 실행한 뒤, Print_Gantt_Chart 함수를 통해서 Gantt Chart를 출력하고 이후 Calculate_Average_Waiting_Time, Calculate_Average_Turnaround_Time, Calculate_CPU_Utilization 함수들을 실행시켜 자신들의 Average Waiting Time, Average Turnaround Time, CPU Utilization을 계산하고 저장하도록 만들었다.

6가지의 Algorithm을 수행한 뒤, 저장한 Average Waiting Time, Average Turnaround Time, CPU Utilization 값들을 비교해서 순위를 매기는 Evaluation 함수를 만들었다.

- CPU Scheduler module 설명

이제부터 핵심인 각 Algorithm의 CPU Scheduler module에 대해서 설명해 볼 예정이다. 순서는 FCFS, SJF, Priority, RR, P_SJF, P_Priority 순이다.

FCFS Algorithm은 먼저 Ready_Queue와 Waiting_Queue를 만든 뒤, 모든 process의 CPU_Burst_Time이 0이 될 때까지 while문을 돌리면서 CPU를 할당해줬다. FCFS Algorithm의 경우 Ready_Queue에 새로운 process가 진입하는 경우는 Arrival_Time이 될 때 혹은 IO_Burst_Time이 0이 되어 다시 Ready_Queue에 들어가는 경우 2가지가 존재하기 때문에 이 점을 고려하여 이 두 가지의 경우가 나타난 경우, 그 process는 Ready_Queue의 맨 뒤로 이동시켰다.

다음으로 CPU를 할당하는 방법은 크게 3가지로 나누었다. 먼저 첫 번째로 현재 CPU를 할당 받고 있는 process가 없으며 Ready_Queue에도 process가 없는 경우, CPU를 IDLE 상태로 만들었다. 그 다음으로 현재 CPU를 할당 받고 있는 process가 없지만 Ready_Queue에 process가 존재하는 경우 Ready_Queue를 pop해서 process를 추출한 후 그 process가 CPU를 할당 받도록 만들었다. 마지막으로 세 번째는 IO Burst를 해야 하거나 혹은 CPU Time이 0이 되어 CPU를 다시 돌려줬을 때, Ready_Queue에 process가 있으면 pop을 해서 새로운 process에 CPU를 할당하거나 없다면 CPU를 IDLE 상태로 만들었다.

IO Burst의 경우 단일 Queue로 구성했으며 FCFS 구조로 설정을 했다. 만약 특정 process가 IO Burst Time이 끝나고 CPU Burst Time이 남아 있다면 다시 Ready_Queue로 돌려주는 작업을 한다.

이런 방식으로 FCFS Algorithm을 수행한 후, Average Waiting Time과 Average Trunaround Time, CPU Utilization을 계산한 이후 값들을 저장하고 출력을 하도록 코드를 작성했다.

SJF Algorithm은 먼저 Ready_Queue와 Waiting_Queue를 만든 뒤, 모든 process의 CPU_Burst_Time이 0이 될 때까지 while문을 돌리면서 CPU를 할당해줬다. SJF Algorithm의 경우 Ready_Queue에 새로운 process가 진입하는 경우는 Arrival_Time이 될 때 혹은 IO_Burst_Time이 0이 되어 다시 Ready_Queue에 들어가는 경우 2가지가 존재하기 때문에 이 점을 고려하여 이 두 가지의 경우가 나타난 경우, Ready_Queue에 존재하는 process들과 CPU Burst Time을 비교하여 다시 Ready_Queue를 정리했다.

다음으로 CPU를 할당하는 방법은 크게 3가지로 나누었다. 먼저 첫 번째로 현재 CPU를 할당 받고 있는 process가 없으며 Ready_Queue에도 process가 없는 경우, CPU를 IDLE 상태로 만들었다. 그 다음으로 현재 CPU를 할당 받고 있는 process가 없지만 Ready_Queue에 process가 존재하는 경우 Ready_Queue를 pop해서 process를 추출한 후 그 process가 CPU를 할당 받도록 만들었다. 마지막으로 세 번째는 IO Burst를 해야 하거나 혹은 CPU Time이 0이 되어 CPU를 다시 돌려줬을 때, Ready_Queue에 process가 있으면 pop을 해서 새로운 process에 CPU를 할당하거나 없다면 CPU를 IDLE 상태로 만들었다.

IO Burst의 경우 단일 Queue로 구성했으며 FCFS 구조로 설정을 했다. 만약 특정 process가 IO Burst Time이 끝나고 CPU Burst Time이 남아 있다면 다시 Ready_Queue로 돌려주는 작업을 한다.

이런 방식으로 SJF Algorithm을 수행한 후, Average Waiting Time과 Average Trunaround Time, CPU Utilization을 계산한 이후 값들을 저장하고 출력을 하도록 코드를 작성했다.

Priority Algorithm은 먼저 Ready_Queue와 Waiting_Queue를 만든 뒤, 모든 process의 CPU_Burst_Time이 0이 될 때까지 while문을 돌리면서 CPU를 할당해줬다. Priority Algorithm의 경우 Ready_Queue에 새로운 process가 진입하는 경우는 Arrival_Time이 될 때 혹은 IO_Burst_Time이 0이 되어 다시 Ready_Queue에 들어가는 경우 2가지가 존재하기 때문에 이 점을 고려하여 이 두 가지의 경우가 나타난 경우, Ready_Queue에 존재하는 process들과 Priority를 비교하여 다시 Ready_Queue를 정리했다.

다음으로 CPU를 할당하는 방법은 크게 3가지로 나누었다. 먼저 첫 번째로 현재 CPU를 할당 받고 있는 process가 없으며 Ready_Queue에도 process가 없는 경우, CPU를 IDLE 상태로 만들었다. 그 다음으로 현재 CPU를 할당 받고 있는 process가 없지만 Ready_Queue에 process가 존재하는 경우 Ready_Queue를 pop해서 process를 추출한 후 그 process가 CPU를 할당 받도록 만들었다. 마지막으로 세 번째는 IO Burst를 해야 하거나 혹은 CPU Time이 0이 되어 CPU를 다시 돌려줬을 때, Ready_Queue에 process가 있으면 pop을 해서 새로운 process에 CPU를 할당하거나 없다면 CPU를 IDLE 상태로 만들었다.

IO Burst의 경우 단일 Queue로 구성했으며 FCFS 구조로 설정을 했다. 만약 특정 process가 IO Burst Time이 끝나고 CPU Burst Time이 남아 있다면 다시 Ready_Queue로 돌려주는 작업을 한다.

이런 방식으로 Priority Algorithm을 수행한 후, Average Waiting Time과 Average Trunaround Time, CPU Utilization을 계산한 이후 값들을 저장하고 출력을 하도록 코드를 작성했다.

RR Algorithm은 먼저 Ready_Queue와 Waiting_Queue를 만든 뒤, 모든 process의 CPU_Burst_Time이 0이 될 때까지 while문을 돌리면서 CPU를 할당해줬다. RR Algorithm의 경우 Ready_Queue에 새로운 process가 진입하는 경우는 Arrival_Time이 될 때, time quantum으로 인해 다시 Ready_Queue에 들어갈 때, IO_Burst_Time이 0이 되어 다시 Ready_Queue에 들어가는 경우 3가지가 존재하기 때문에 이 점을 고려하여 이 세 가지의 경우가 나타난 경우, 그 process는 Ready_Queue의 맨 뒤로 이동시켰다.

다음으로 CPU를 할당하는 방법은 크게 3가지로 나누었다. 먼저 첫 번째로 현재 CPU를 할당 받고 있는 process가 없으며 Ready_Queue에도 process가 없는 경우, CPU를 IDLE 상태로 만들었다. 그 다음으로 현재 CPU를 할당 받고 있는 process가 없지만 Ready_Queue에 process가 존재하는 경우 Ready_Queue를 pop해서 process를 추출한 후 그 process가 CPU를 할당 받도록 만들었다. 마지막으로 세 번째는 IO Burst를 해야 하거나 혹은 CPU Time이 0이 되어 CPU를 다시 돌려줬을 때, Ready_Queue에 process가 있으면 pop을 해서 새로운 process에 CPU를 할당하거나 없다면 CPU를 IDLE 상태로 만들었다.

다른 Algorithm과 달리 RR Algorithm은 time quantum으로 인해 CPU를 다시 반납해야 하는 경우가 있어서 특정 process가 CPU를 할당 받을 때, Quantum_Time을 0으로 설정한 뒤 계속 1씩 증가시켜서 Quantum_Time이 time quantum과 동일하게 된다면 CPU를 반납하도록 코드를 작성했다.

IO Burst의 경우 단일 Queue로 구성했으며 FCFS 구조로 설정을 했다. 만약 특정 process가 IO Burst Time이 끝나고 CPU Burst Time이 남아 있다면 다시 Ready_Queue로 돌려주는 작업을 한다.

이런 방식으로 RR Algorithm을 수행한 후, Average Waiting Time과 Average Trunaround Time, CPU Utilization을 계산한 이후 값들을 저장하고 출력을 하도록 코드를 작성했다.

P_SJF와 **P_Priority**의 경우 기존의 SJF와 Priority의 구조와 흡사한 모습을 하고 있지만 Preemptive의 특성 때문에 일부분에서 차이 한 가지가 존재한다. 바로 새로운 process가

Ready_Queue에 들어왔을 때, 특정 조건을 만족시킨다면 현재 CPU를 할당 받고 있는 process로부터 CPU를 빼앗아 자기가 CPU를 점유할 수 있다는 점이다.

위와 같은 특징을 표현하기 위해서 나는 새로운 process가 Ready_Queue에 들어왔을 때, 현재 CPU를 할당 받고 있는 process와 비교를 한다. P_SJF의 경우라면 CPU_Burst_Time, P_Priority의 경우라면 Priority를 비교하여 만약 새로 들어온 process가 현재 CPU를 점유하고 있는 process보다 우위에 있다면 CPU를 새로 들어온 process에게 제공하고 CPU를 빼앗긴 process는 Ready_Queue의 맨 앞자리에 배치하도록 설정했다.

먼저 새로 들어온 process가 Ready_Queue에 있는 procecss들은 비교하지 않고 CPU를 점유하고 있는 process만 비교하는 이유는 현재 CPU를 점유하고 있는 process가 Ready_Queue에 있는 다른 process들보다 우위에 있기 때문이고 새로운 process가 CPU를 빼앗았을 때 CPU를 빼앗긴 process가 Ready_Queue의 맨 앞에 위치하는 이유는 앞에서도 이야기했지만 다른 process들보다도 우위를 점하고 있어 Ready_Queue 내에서 1순위가 되어야 하기 때문이다.

위와 같은 방식들로 FCFS, SJF, Priority, RR, P_SJF, P_Priority 6가지의 Algorithm을 만들었다.

- CPU Scheduling Simulator 결과

```
Process 1:
CPU_Burst_Time = 2
IO_Burst_Time = 3
Arrival_Time = 15
Priority = 1
IO_Burst_Timing = 1

Process 2:
CPU_Burst_Time = 5
IO_Burst_Time = 4
Arrival_Time = 19
Priority = 3
IO_Burst_Timing = 1

Process 3:
CPU_Burst_Time = 6
IO_Burst_Time = 1
Arrival_Time = 2
Priority = 8
IO_Burst_Timing = 5

Process 4:
CPU_Burst_Time = 12
IO_Burst_Time = 1
Arrival_Time = 3
Priority = 8
IO_Burst_Timing = 12

Process 5:
CPU_Burst_Time = 12
IO_Burst_Time = 5
Arrival_Time = 3
Priority = 4
IO_Burst_Timing = 8
```

<각 process의 특징>

```
FCFS:
|      |      |      |      |      |      |      |      |      |      |      |      |
| IDLE |  P3  |  P4  |  P5  |  P3  |  P1  |  P2  | IDLE |  P5  |  P1  | IDLE |  P2  |
|      |      |      |      |      |      |      |      |      |      |      |      |
0      2      7      19     27     28     29     30     32     36     37     39     43

Average Waiting Time :
( 14 + 10 + 19 + 4 + 16 ) / 5 = 12.6

Average Turnaround Time :
( 22 + 24 + 26 + 16 + 33 ) / 5 = 24.2

CPU Utilization :
37 / 43 = 86%
```

<FCFS Algorithm>

```
SJF:
|      |      |      |      |      |      |      |      |      |      |
| IDLE |  P3  |  P4  |  P3  |  P1  |  P2  |  P5  |  P1  |  P2  |  P5  |
|      |      |      |      |      |      |      |      |      |      |
0      2      7      19     20     21     22     30     31     35     39

Average Waiting Time :
( 11 + 5 + 11 + 4 + 19 ) / 5 = 10.0

Average Turnaround Time :
( 16 + 16 + 18 + 16 + 36 ) / 5 = 20.4

CPU Utilization :
37 / 39 = 95%
```

<SJF Algorithm>

```
Priority:
|      |      |      |      |      |      |      |      |      |      |      |
| IDLE |  P3  |  P4  |  P3  |  P5  |  P2  |  P1  | IDLE |  P5  |  P2  |  P1  |
|      |      |      |      |      |      |      |      |      |      |      |
0      2      7      19     20     28     29     30     33     37     41     42

Average Waiting Time :
( 15 + 9 + 11 + 4 + 17 ) / 5 = 11.2

Average Turnaround Time :
( 27 + 22 + 18 + 16 + 34 ) / 5 = 23.4

CPU Utilization :
37 / 42 = 88%
```

<Priority Algorithm>

```
RR:
|      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |
| IDLE |  P3  |  P4  |  P5  |  P3  |  P4  |  P5  |  P1  |  P3  |  P4  |  P2  |  P5  |  P1  |  P2  |
|      |      |      |      |      |      |      |      |      |      |      |      |      |      |
0      2      6      10     14     15     19     23     24     25     29     30     34     35     39

Average Waiting Time :
( 11 + 10 + 16 + 14 + 14 ) / 5 = 13.0

Average Turnaround Time :
( 20 + 20 + 23 + 26 + 31 ) / 5 = 24.0

CPU Utilization :
37 / 39 = 95%
```

<RR Algorithm>

```
P_SJF:
|      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |
| IDLE |  P3  |  P4  |  P3  |  P4  |  P1  |  P4  |  P1  |  P4  |  P2  |  P5  |  P2  |  P5  | IDLE |  P5  |
|      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |
0      2      7      8      9      15     16     19     20     22     23     27     31     35     40     44

Average Waiting Time :
( 0 + 3 + 0 + 7 + 24 ) / 5 = 6.8

Average Turnaround Time :
( 5 + 12 + 7 + 19 + 41 ) / 5 = 16.8

CPU Utilization :
37 / 44 = 84%
```

<P_SJF Algorithm>

```
P_Priority:
|      |      |      |      |      |      |      |      |      |      |      |      |      |      |
| IDLE |  P3  |  P4  |  P3  |  P5  |  P2  |  P1  | IDLE |  P5  |  P2  |  P1  |  P2  |  P1  |
|      |      |      |      |      |      |      |      |      |      |      |      |      |      |
0      2      7      19     20     28     29     30     33     37     40     40     41     42

Average Waiting Time :
( 15 + 9 + 11 + 4 + 17 ) / 5 = 11.2

Average Turnaround Time :
( 27 + 22 + 18 + 16 + 34 ) / 5 = 23.4

CPU Utilization :
37 / 42 = 88%
```

<P_Priority Algorithm>

```
Rank of Average Waiting Time:
1. P_SJF : 6.8
2. SJF : 10.0
3. Priority : 11.2
4. P_Priority : 11.2
5. FCFS : 12.6
6. RR : 13.0
```

<Rank of Average Waiting Time>

```
Rank of Average Turnaround Time:
1. P_SJF : 16.8
2. SJF : 20.4
3. Priority : 23.4
4. P_Priority : 23.4
5. RR : 24.0
6. FCFS : 24.2
```

<Rank of Average Turnaround Time>

```
Rank of CPU Utilization:
1. SJF : 95%
2. RR : 95%
3. Priority : 88%
4. P_Priority : 88%
5. FCFS : 86%
6. P_SJF : 84%
```

<Rank of CPU Utilization>

- CPU Scheduler module 성능 비교

CPU Scheduler module를 여러 번 실행한 결과, Rank of Average Waiting Time, Rank of Average Turnaround Time에서 P_SJF, SJF가 항상 1, 2위를 차지했다. 즉, SJF Algorithm이 Waiting Time과 Turnaround Time의 측면에서 매우 높은 효율성을 보이고 있다는 사실을 알 수 있다. 하지만 현재 Project에서는 각 process들의 CPU_Burst_Time을 알고 있기 때문에 SJF Algorithm을 구현할 수 있었지만 현실에서는 CPU_Burst_Time을 알기 매우 힘들기 때문에 현실적으로 구현하기 어렵다는 점이 있다.

이외의 순위는 매번 돌릴 때마다 조금씩 바뀌고 CPU Utilization 또한 SJF를 포함해서 다양하게 순위가 나온다는 사실을 알 수 있다.

3. 결론

- Project 수행 소감 및 향후 발전 방향

이번 project에서 구현한 CPU Scheduling Simulator는 6가지의 CPU Scheduling Algorithm의 동작 방식과 결과를 분석하는 것을 초점으로 진행했다. 이를 통해서 각 Algorithm들의 장단점을 비교하고, 이에 대한 성능을 평가할 수 있었다.

먼저 Operating System에서 배운 CPU Scheduling을 직접 구현했다는 점에서 매우 흥미로웠고 이론으로 배운 내용들이 실제로도 적용이 되는지 두 눈으로 볼 수 있어서 좋은 경험이 되었다. 특히 SJF Algorithm이 실제로도 Average Waiting Time과 Average Turnaround Time의 측면에서 가장 효율적인지 확인하는 것과 Preemptive 방식이 적용되고 안 적용되는 차이점을 확인할 수 있다는 점이 매우 좋았다.

하지만 SJF나 Priority Algorithm에서 Starvation이 나타날 수 있다는 문제점을 해결하기 위한 Aging 기법을 구현하는 것까지 하고 싶었지만 시간상 할 수 없었다는 점이 아쉬웠다. 그래서 이번 project 이후 시간을 따로 내서 Aging 기법을 실제로 구현하고 더 나아가서 MLFQ(Multilevel Feedback Queue)와 같은 다른 CPU Scheduling Algorithm을 구현을 할 계획이다.

<Source Code>

```
#pragma warning(disable:4996)

#include <stdio.h>

#include <string.h>

#include <malloc.h>

#include <math.h>

#include <stdlib.h>

#include <time.h>


int Rank_Waiting_Time[6] = { 0, 1, 2, 3, 4, 5 };

int Rank_Turnaround_Time[6] = { 0, 1, 2, 3, 4, 5 };

int Rank_CPU_Utilization[6] = { 0, 1, 2, 3, 4, 5 };

float Average_Waiting_Time[6], Average_Turnaround_Time[6], CPU_Utilization[6];

int left_IO[6][5];


typedef struct {

	int Process_ID;

	int CPU_Burst_Time;

	int IO_Burst_Time;

	int Arrival_Time;

	int Priority;

	int IO_Burst_Timing;

}Process;
```

```
typedef struct {
	int* ID;
	int cnt;
}QUEUE;

QUEUE* Create_Queue(void) {
	QUEUE* buf = (QUEUE*)malloc(sizeof(QUEUE));
	buf->ID = (int*)malloc(sizeof(int) * 1000);
	buf->cnt = 0;
	return buf;
}

void Enqueue(QUEUE* queue, int name) {
	*(queue->ID + (queue->cnt)) = name;
	queue->cnt++;
}

void Dequeue(QUEUE* queue) {
	queue->cnt--;
	memmove(queue->ID, queue->ID + 1, sizeof(int) * (queue->cnt));
}

Process Create_Process(int ID) {
	Process buf;

	buf.Process_ID = ID;
```

```
	buf.CPU_Burst_Time = rand() % 20 + 1;
	buf.IO_Burst_Time = rand() % 5 + 1;
	buf.Arrival_Time = rand() % 20 + 1;
	buf.Priority = rand() % 10 + 1;
	buf.IO_Burst_Timing = rand() % buf.CPU_Burst_Time;

	return buf;
}

Process Copy_Process(Process process) {
	Process buf;

	buf.Process_ID = process.Process_ID;
	buf.CPU_Burst_Time = process.CPU_Burst_Time;
	buf.IO_Burst_Time = process.IO_Burst_Time;
	buf.Arrival_Time = process.Arrival_Time;
	buf.Priority = process.Priority;
	buf.IO_Burst_Timing = process.IO_Burst_Timing;

	return buf;
}

void Print_Process(Process process_Info[], int Process_Cnt) {

	for (int i = 0; i < Process_Cnt; i++) {
		printf("Process %d: ₩n", i + 1);
```

```
                printf("CPU_Burst_Time = %d₩n", process_Info[i].CPU_Burst_Time);
                printf("IO_Burst_Time = %d₩n", process_Info[i].IO_Burst_Time);
                printf("Arrival_Time = %d₩n", process_Info[i].Arrival_Time);
                printf("Priority = %d₩n", process_Info[i].Priority);
                printf("IO_Burst_Timing = %d₩n", process_Info[i].CPU_Burst_Time - process_Info[i].IO_Burst_Timing);
                printf("₩n");
        }
}

void Print_Gantt_Chart(int** CPU_Info, int flag) {
        for (int i = 0; i < flag; i++) printf("|       ");
        printf("|₩n");
        for (int i = 0; i < flag; i++) {
                if (CPU_Info[1][i] == -1) printf("| IDLE ");
                else printf("|   P%d   ", CPU_Info[1][i]);
        }
        printf("|₩n");
        for (int i = 0; i < flag; i++) printf("|       ");
        printf("|₩n");
        printf("0       ");
        for (int i = 0; i < flag; i++) {
                if (CPU_Info[0][i] < 10) printf("%d       ", CPU_Info[0][i]);
                else printf("%d      ", CPU_Info[0][i]);
        }
```

```
	printf("₩n₩n");
}

float Calculate_Average_Waiting_Time(Process Process_Info[], int Process_Cnt, int** CPU_Info, int flag, int left_IO_flag) {
	int** Keep_Value = (int**)malloc(sizeof(int*) * 2);
	int Total = 0;
	int Future_Time = 0;
	int Currunt_Time = 0;

	for (int i = 0; i < 2; i++) Keep_Value[i] = (int*)malloc(sizeof(int) * Process_Cnt);
	for (int i = 0; i < Process_Cnt; i++) {
		Keep_Value[0][i] = 0;
		Keep_Value[1][i] = 0;
	}
	for (int i = 0; i < flag; i++) {
		if (i == 0) Currunt_Time = 0;
		else Currunt_Time = CPU_Info[0][i - 1];
		Future_Time = CPU_Info[0][i];
		if (CPU_Info[1][i] != -1) {
			if (Keep_Value[1][CPU_Info[1][i] - 1] == 0) {
				Keep_Value[0][CPU_Info[1][i] - 1] = Currunt_Time - Process_Info[CPU_Info[1][i] - 1].Arrival_Time;
				Keep_Value[1][CPU_Info[1][i] - 1] = Future_Time;
			}
			else {
```

```
                                        Keep_Value[0][CPU_Info[1][i] - 1] += (Currunt_Time - Keep_Value[1][CPU_Info[1][i] - 1]);
                                        Keep_Value[1][CPU_Info[1][i] - 1] = Future_Time;
                                }
                        }
                }
                for (int i = 0; i < Process_Cnt; i++) {
                        if(Process_Info[i].IO_Burst_Timing > 0) Keep_Value[0][i] -= (Process_Info[i].IO_Burst_Time + left_IO[left_IO_flag][i]);
                }
                printf("Average Waiting Time :₩n");
                printf("( ");
                for (int i = 0; i < Process_Cnt; i++) {
                        if (i != Process_Cnt - 1) printf("%d + ", Keep_Value[0][i]);
                        else printf("%d )", Keep_Value[0][i]);
                        Total += Keep_Value[0][i];
                }

                float ans = (float)Total / Process_Cnt;

                printf(" / %d = %.1f₩n₩n", Process_Cnt, ans);
                return ans;
}

float Calculate_Average_Turnaround_Time(Process Process_Info[], int Process_Cnt, int** CPU_Info, int flag) {
```

```
	int* Keep_Value = (int*)malloc(sizeof(int) * Process_Cnt);
	int Total = 0;

	for (int i = 0; i < Process_Cnt; i++) Keep_Value[i] = 0;

	for (int i = 0; i < flag; i++) if (CPU_Info[1][i] != -1) Keep_Value[CPU_Info[1][i] - 1] = CPU_Info[0][i];

	for (int i = 0; i < Process_Cnt; i++) Keep_Value[i] -= Process_Info[i].Arrival_Time;

	//for (int i = 0; i < Process_Cnt; i++) Keep_Value[i] -= Process_Info[i].IO_Burst_Time;

	printf("Average Turnaround Time :₩n");
	printf("( ");
	for (int i = 0; i < Process_Cnt; i++) {
		if (i != Process_Cnt - 1) printf("%d + ", Keep_Value[i]);
		else printf("%d )", Keep_Value[i]);
		Total += Keep_Value[i];
	}

	float ans = (float)Total / Process_Cnt;

	printf(" / %d = %.1f₩n₩n", Process_Cnt, ans);
	return ans;
}
```

```
float Calculate_CPU_Utilization(int** CPU_Info, int flag) {
	int CPU_Idle = 0;
	int Last_Time = CPU_Info[0][flag - 1];
	for (int i = 0; i < flag; i++) {
		if (CPU_Info[1][i] == -1) {
			if (i == 0) CPU_Idle += CPU_Info[0][i];
			else CPU_Idle += CPU_Info[0][i] - CPU_Info[0][i - 1];
		}
	}

	int CPU_Not_Idle = Last_Time - CPU_Idle;
	float ans = (float)CPU_Not_Idle / Last_Time;

	printf("CPU Utilization :₩n");
	printf("%d / %d = %.0f%%₩n₩n", CPU_Not_Idle, Last_Time, ans * 100);

	return ans;
}

void FCFS(Process Process_Info[], int Process_Cnt) {
	QUEUE* Ready_Queue = Create_Queue();
	QUEUE* Waiting_Queue = Create_Queue();

	Process* process = (Process*)malloc(sizeof(Process) * Process_Cnt);
	for (int i = 0; i < Process_Cnt; i++) process[i] = Copy_Process(Process_Info[i]);
```

```
	int Process_End_Cnt = 0;
	int flag = 0;

	int** CPU_Info = (int**)malloc(sizeof(int) * 2);
	for (int i = 0; i < 2; i++) CPU_Info[i] = (int*)malloc(sizeof(int) * 500);

	int Currunt_CPU_Process = -1;
	int Currunt_IO_Process = -1;
	int CPU_Idle_Time = 0;
	int CPU_Time = 0;

	while (Process_End_Cnt < Process_Cnt) {
		for (int i = 0; i < Process_Cnt; i++) if (CPU_Time == process[i].Arrival_Time) Enqueue(Ready_Queue, i + 1);

		CPU_Time++;

		if (Currunt_CPU_Process == -1 && Ready_Queue->cnt == 0) CPU_Idle_Time++;
		else {
			if (Currunt_CPU_Process == -1) {
				if (CPU_Idle_Time != 0) {
					CPU_Info[0][flag] = CPU_Time - 1;
					CPU_Info[1][flag] = -1;
					flag++;
					CPU_Idle_Time = 0;
				}
```

```
                Currunt_CPU_Process = Ready_Queue->ID[0];
                Dequeue(Ready_Queue);
            }
            process[Currunt_CPU_Process - 1].CPU_Burst_Time--;
            if (process[Currunt_CPU_Process - 1].CPU_Burst_Time == 0) {
                CPU_Info[0][flag] = CPU_Time;
                CPU_Info[1][flag] = Currunt_CPU_Process;
                flag++;
                Currunt_CPU_Process = -1;
                Process_End_Cnt++;
            }
            if (process[Currunt_CPU_Process - 1].CPU_Burst_Time == process[Currunt_CPU_Process - 1].IO_Burst_Timing) {
                left_IO[0][Currunt_CPU_Process - 1] = -CPU_Time;
                Enqueue(Waiting_Queue, Currunt_CPU_Process);
                CPU_Info[0][flag] = CPU_Time;
                CPU_Info[1][flag] = Currunt_CPU_Process;
                flag++;
                Currunt_CPU_Process = -1;
            }
        }

        if (Currunt_IO_Process != -1) process[Currunt_IO_Process - 1].IO_Burst_Time--;
        if (Currunt_IO_Process != -1 && process[Currunt_IO_Process - 1].IO_Burst_Time == 0) {
            if(process[Currunt_IO_Process - 1].CPU_Burst_Time != 0)
```

```
Enqueue(Ready_Queue, Currunt_IO_Process);
                        Currunt_IO_Process = -1;
                }
                if (Currunt_IO_Process == -1 && Waiting_Queue->cnt != 0) {
                        Currunt_IO_Process = Waiting_Queue->ID[0];
                        left_IO[0][Currunt_IO_Process - 1] += CPU_Time;
                        Dequeue(Waiting_Queue);
                }

        }

        printf("FCFS:₩n");
        Print_Gantt_Chart(CPU_Info, flag);

        float FCFS_Average_Waiting_Time = Calculate_Average_Waiting_Time(Process_Info, Process_Cnt, CPU_Info, flag, 0);
        float FCFS_Average_Turnaround_Time = Calculate_Average_Turnaround_Time(Process_Info, Process_Cnt, CPU_Info, flag);
        float FCFS_CPU_Utilization = Calculate_CPU_Utilization(CPU_Info, flag);

        Average_Waiting_Time[0] = FCFS_Average_Waiting_Time;
        Average_Turnaround_Time[0] = FCFS_Average_Turnaround_Time;
        CPU_Utilization[0] = FCFS_CPU_Utilization;

        /*
        free(Ready_Queue->ID);
```

```
	free(Waiting_Queue->ID);
	free(Ready_Queue);
	free(Waiting_Queue);
	free(process);
	for (int i = 0; i < 2; i++) free(CPU_Info[i]);
	free(CPU_Info);
	*/
}

void SJF(Process Process_Info[], int Process_Cnt) {
	QUEUE* Ready_Queue = Create_Queue();
	QUEUE* Waiting_Queue = Create_Queue();

	Process* process = (Process*)malloc(sizeof(Process) * Process_Cnt);
	for (int i = 0; i < Process_Cnt; i++) process[i] = Copy_Process(Process_Info[i]);

	int Process_End_Cnt = 0;
	int flag = 0;

	int** CPU_Info = (int**)malloc(sizeof(int) * 2);
	for (int i = 0; i < 2; i++) CPU_Info[i] = (int*)malloc(sizeof(int) * 500);

	int Currunt_CPU_Process = -1;
	int Currunt_IO_Process = -1;
	int CPU_Idle_Time = 0;
	int CPU_Time = 0;
```

```
while (Process_End_Cnt < Process_Cnt) {
	for (int i = 0; i < Process_Cnt; i++) if (CPU_Time == process[i].Arrival_Time) {
		Enqueue(Ready_Queue, i + 1);
		for (int j = 0; j < Ready_Queue->cnt; j++) {
			for (int k = j + 1; k < Ready_Queue->cnt; k++) {
				if (process[Ready_Queue->ID[j] - 1].CPU_Burst_Time > process[Ready_Queue->ID[k] - 1].CPU_Burst_Time) {
					int temp = Ready_Queue->ID[j];
					Ready_Queue->ID[j] = Ready_Queue->ID[k];
					Ready_Queue->ID[k] = temp;
				}
			}
		}
	}

	CPU_Time++;

	if (Currunt_CPU_Process == -1 && Ready_Queue->cnt == 0) CPU_Idle_Time++;
	else {
		if (Currunt_CPU_Process == -1) {
			if (CPU_Idle_Time != 0) {
				CPU_Info[0][flag] = CPU_Time - 1;
				CPU_Info[1][flag] = -1;
				flag++;
				CPU_Idle_Time = 0;
```

```
                }
                Currunt_CPU_Process = Ready_Queue->ID[0];
                Dequeue(Ready_Queue);
            }
            process[Currunt_CPU_Process - 1].CPU_Burst_Time--;
            if (process[Currunt_CPU_Process - 1].CPU_Burst_Time == 0) {
                CPU_Info[0][flag] = CPU_Time;
                CPU_Info[1][flag] = Currunt_CPU_Process;
                flag++;
                if (process[Currunt_CPU_Process - 1].CPU_Burst_Time == process[Currunt_CPU_Process - 1].IO_Burst_Timing) {
                    left_IO[1][Currunt_CPU_Process - 1] = -CPU_Time;
                    Enqueue(Waiting_Queue, Currunt_CPU_Process);
                }
                Currunt_CPU_Process = -1;
                Process_End_Cnt++;
            }
            if (process[Currunt_CPU_Process - 1].CPU_Burst_Time == process[Currunt_CPU_Process - 1].IO_Burst_Timing) {
                left_IO[1][Currunt_CPU_Process - 1] = -CPU_Time;
                Enqueue(Waiting_Queue, Currunt_CPU_Process);
                CPU_Info[0][flag] = CPU_Time;
                CPU_Info[1][flag] = Currunt_CPU_Process;
                flag++;
                Currunt_CPU_Process = -1;
            }
```

```
		}

		if (Currunt_IO_Process != -1) process[Currunt_IO_Process - 1].IO_Burst_Time--;
		if (Currunt_IO_Process != -1 && process[Currunt_IO_Process - 1].IO_Burst_Time ==
0) {
			if (process[Currunt_IO_Process - 1].CPU_Burst_Time > 0) {
				Enqueue(Ready_Queue, Currunt_IO_Process);
				for (int i = 0; i < Ready_Queue->cnt; i++) {
					for (int j = i + 1; j < Ready_Queue->cnt; j++) {
						if (process[Ready_Queue->ID[i] -
1].CPU_Burst_Time > process[Ready_Queue->ID[j] - 1].CPU_Burst_Time) {
							int temp = Ready_Queue->ID[i];
							Ready_Queue->ID[i] = Ready_Queue-
>ID[j];
							Ready_Queue->ID[j] = temp;
						}
					}
				}
			}
			Currunt_IO_Process = -1;
		}
		if (Currunt_IO_Process == -1 && Waiting_Queue->cnt > 0) {
			Currunt_IO_Process = Waiting_Queue->ID[0];
			left_IO[1][Currunt_IO_Process - 1] += CPU_Time;
			Dequeue(Waiting_Queue);
		}
```

```
        }

        printf("SJF:₩n");
        Print_Gantt_Chart(CPU_Info, flag);

        float SJF_Average_Waiting_Time = Calculate_Average_Waiting_Time(Process_Info, Process_Cnt, CPU_Info, flag, 1);
        float SJF_Average_Turnaround_Time = Calculate_Average_Turnaround_Time(Process_Info, Process_Cnt, CPU_Info, flag);
        float SJF_CPU_Utilization = Calculate_CPU_Utilization(CPU_Info, flag);

        Average_Waiting_Time[1] = SJF_Average_Waiting_Time;
        Average_Turnaround_Time[1] = SJF_Average_Turnaround_Time;
        CPU_Utilization[1] = SJF_CPU_Utilization;

        /*
        free(Ready_Queue->ID);
        free(Waiting_Queue->ID);
        free(Ready_Queue);
        free(Waiting_Queue);
        free(process);
        for (int i = 0; i < 2; i++) free(CPU_Info[i]);
        free(CPU_Info);
        */
}
```

```
void Priority(Process Process_Info[], int Process_Cnt) {
	QUEUE* Ready_Queue = Create_Queue();
	QUEUE* Waiting_Queue = Create_Queue();

	Process* process = (Process*)malloc(sizeof(Process) * Process_Cnt);
	for (int i = 0; i < Process_Cnt; i++) process[i] = Copy_Process(Process_Info[i]);

	int Process_End_Cnt = 0;
	int flag = 0;

	int** CPU_Info = (int**)malloc(sizeof(int) * 2);
	for (int i = 0; i < 2; i++) CPU_Info[i] = (int*)malloc(sizeof(int) * 500);

	int Currunt_CPU_Process = -1;
	int Currunt_IO_Process = -1;
	int CPU_Idle_Time = 0;
	int CPU_Time = 0;

	while (Process_End_Cnt < Process_Cnt) {
		for (int i = 0; i < Process_Cnt; i++) if (CPU_Time == process[i].Arrival_Time) {
			Enqueue(Ready_Queue, i + 1);
			for (int j = 0; j < Ready_Queue->cnt; j++) {
				for (int k = j + 1; k < Ready_Queue->cnt; k++) {
					if (process[Ready_Queue->ID[j] - 1].Priority < process[Ready_Queue->ID[k] - 1].Priority) {
						int temp = Ready_Queue->ID[j];
```

```
                                        Ready_Queue->ID[j] = Ready_Queue->ID[k];
                                        Ready_Queue->ID[k] = temp;
                                }
                        }
                }
        }

        CPU_Time++;

        if (Currunt_CPU_Process == -1 && Ready_Queue->cnt == 0) CPU_Idle_Time++;
        else {
                if (Currunt_CPU_Process == -1) {
                        if (CPU_Idle_Time != 0) {
                                CPU_Info[0][flag] = CPU_Time - 1;
                                CPU_Info[1][flag] = -1;
                                flag++;
                                CPU_Idle_Time = 0;
                        }
                        Currunt_CPU_Process = Ready_Queue->ID[0];
                        Dequeue(Ready_Queue);
                }
                process[Currunt_CPU_Process - 1].CPU_Burst_Time--;
                if (process[Currunt_CPU_Process - 1].CPU_Burst_Time == 0) {
                        CPU_Info[0][flag] = CPU_Time;
                        CPU_Info[1][flag] = Currunt_CPU_Process;
                        flag++;
```

```
                    if (process[Currunt_CPU_Process - 1].CPU_Burst_Time == process[Currunt_CPU_Process - 1].IO_Burst_Timing) {
                        left_IO[2][Currunt_CPU_Process - 1] = -CPU_Time;
                        Enqueue(Waiting_Queue, Currunt_CPU_Process);
                    }
                    Currunt_CPU_Process = -1;
                    Process_End_Cnt++;
                }
                if (process[Currunt_CPU_Process - 1].CPU_Burst_Time == process[Currunt_CPU_Process - 1].IO_Burst_Timing) {
                    left_IO[2][Currunt_CPU_Process - 1] = -CPU_Time;
                    Enqueue(Waiting_Queue, Currunt_CPU_Process);
                    CPU_Info[0][flag] = CPU_Time;
                    CPU_Info[1][flag] = Currunt_CPU_Process;
                    flag++;
                    Currunt_CPU_Process = -1;
                }
            }

            if (Currunt_IO_Process != -1) process[Currunt_IO_Process - 1].IO_Burst_Time--;
            if (Currunt_IO_Process != -1 && process[Currunt_IO_Process - 1].IO_Burst_Time == 0) {
                if (process[Currunt_IO_Process - 1].CPU_Burst_Time > 0) {
                    Enqueue(Ready_Queue, Currunt_IO_Process);
                    for (int i = 0; i < Ready_Queue->cnt; i++) {
                        for (int j = i + 1; j < Ready_Queue->cnt; j++) {
```

```
                                        if (process[Ready_Queue->ID[i] - 1].Priority < process[Ready_Queue->ID[j] - 1].Priority) {
                                            int temp = Ready_Queue->ID[i];
                                            Ready_Queue->ID[i] = Ready_Queue->ID[j];
                                            Ready_Queue->ID[j] = temp;
                                        }
                                    }
                                }
                            }
                            Currunt_IO_Process = -1;
                        }
                        if (Currunt_IO_Process == -1 && Waiting_Queue->cnt > 0) {
                            Currunt_IO_Process = Waiting_Queue->ID[0];
                            left_IO[2][Currunt_IO_Process - 1] += CPU_Time;
                            Dequeue(Waiting_Queue);
                        }
                    }

                    printf("Priority:₩n");
                    Print_Gantt_Chart(CPU_Info, flag);

                    float Priority_Average_Waiting_Time = Calculate_Average_Waiting_Time(Process_Info, Process_Cnt, CPU_Info, flag, 2);
                    float Priority_Average_Turnaround_Time = Calculate_Average_Turnaround_Time(Process_Info, Process_Cnt, CPU_Info, flag);
                    float Priority_CPU_Utilization = Calculate_CPU_Utilization(CPU_Info, flag);
```

```
	Average_Waiting_Time[2] = Priority_Average_Waiting_Time;
	Average_Turnaround_Time[2] = Priority_Average_Turnaround_Time;
	CPU_Utilization[2] = Priority_CPU_Utilization;

	/*
	free(Ready_Queue->ID);
	free(Waiting_Queue->ID);
	free(Ready_Queue);
	free(Waiting_Queue);
	free(process);
	for (int i = 0; i < 2; i++) free(CPU_Info[i]);
	free(CPU_Info);
	*/
}

void RR(Process Process_Info[], int Process_Cnt, int quantum) {
	QUEUE* Ready_Queue = Create_Queue();
	QUEUE* Waiting_Queue = Create_Queue();

	Process* process = (Process*)malloc(sizeof(Process) * Process_Cnt);
	for (int i = 0; i < Process_Cnt; i++) process[i] = Copy_Process(Process_Info[i]);

	int Process_End_Cnt = 0;
	int flag = 0;
```

```
	int** CPU_Info = (int**)malloc(sizeof(int*) * 2);
	for (int i = 0; i < 2; i++) CPU_Info[i] = (int*)malloc(sizeof(int) * 500);

	int Currunt_CPU_Process = -1;
	int Currunt_IO_Process = -1;
	int CPU_Idle_Time = 0;
	int CPU_Time = 0;
	int Quantum_Time = 0;

	while (Process_End_Cnt < Process_Cnt) {
		for (int i = 0; i < Process_Cnt; i++) if (CPU_Time == process[i].Arrival_Time) Enqueue(Ready_Queue, i + 1);

		CPU_Time++;

		if (Currunt_CPU_Process == -1 && Ready_Queue->cnt == 0) CPU_Idle_Time++;
		else {
			if (Currunt_CPU_Process == -1) {
				if (CPU_Idle_Time != 0) {
					CPU_Info[0][flag] = CPU_Time - 1;
					CPU_Info[1][flag] = -1;
					flag++;
					CPU_Idle_Time = 0;
				}
				Currunt_CPU_Process = Ready_Queue->ID[0];
				Dequeue(Ready_Queue);
```

```
                }
                process[Currunt_CPU_Process - 1].CPU_Burst_Time--;
                Quantum_Time++;
                if (process[Currunt_CPU_Process - 1].CPU_Burst_Time == 0) {
                        CPU_Info[0][flag] = CPU_Time;
                        CPU_Info[1][flag] = Currunt_CPU_Process;
                        flag++;
                        Process_End_Cnt++;
                        Quantum_Time = 0;
                        Currunt_CPU_Process = -1;
                }
                else if (process[Currunt_CPU_Process - 1].CPU_Burst_Time ==
process[Currunt_CPU_Process - 1].IO_Burst_Timing) {
                        left_IO[3][Currunt_CPU_Process - 1] = -CPU_Time;
                        Enqueue(Waiting_Queue, Currunt_CPU_Process);
                        CPU_Info[0][flag] = CPU_Time;
                        CPU_Info[1][flag] = Currunt_CPU_Process;
                        flag++;
                        Quantum_Time = 0;
                        Currunt_CPU_Process = -1;
                }
                else if (Quantum_Time == quantum) {
                        CPU_Info[0][flag] = CPU_Time;
                        CPU_Info[1][flag] = Currunt_CPU_Process;
                        flag++;
                        Enqueue(Ready_Queue, Currunt_CPU_Process);
```

```
                    Quantum_Time = 0;
                    Currunt_CPU_Process = -1;
                }
            }

            if (Currunt_IO_Process != -1) process[Currunt_IO_Process - 1].IO_Burst_Time--;
            if (Currunt_IO_Process != -1 && process[Currunt_IO_Process - 1].IO_Burst_Time ==
0) {
                if (process[Currunt_IO_Process - 1].CPU_Burst_Time != 0)
Enqueue(Ready_Queue, Currunt_IO_Process);
                Currunt_IO_Process = -1;
            }
            if (Currunt_IO_Process == -1 && Waiting_Queue->cnt != 0) {
                Currunt_IO_Process = Waiting_Queue->ID[0];
                left_IO[3][Currunt_IO_Process - 1] += CPU_Time;
                Dequeue(Waiting_Queue);
            }

        }

        printf("RR:₩n");
        Print_Gantt_Chart(CPU_Info, flag);

        float RR_Average_Waiting_Time = Calculate_Average_Waiting_Time(Process_Info,
Process_Cnt, CPU_Info, flag, 3);
        float RR_Average_Turnaround_Time = Calculate_Average_Turnaround_Time(Process_Info,
Process_Cnt, CPU_Info, flag);
```

```
	float RR_CPU_Utilization = Calculate_CPU_Utilization(CPU_Info, flag);

	Average_Waiting_Time[3] = RR_Average_Waiting_Time;
	Average_Turnaround_Time[3] = RR_Average_Turnaround_Time;
	CPU_Utilization[3] = RR_CPU_Utilization;

	/*
	free(Ready_Queue->ID);
	free(Waiting_Queue->ID);
	free(Ready_Queue);
	free(Waiting_Queue);
	free(process);
	for (int i = 0; i < 2; i++) free(CPU_Info[i]);
	free(CPU_Info);
	*/
}

void P_SJF(Process Process_Info[], int Process_Cnt) {
	QUEUE* Ready_Queue = Create_Queue();
	QUEUE* Waiting_Queue = Create_Queue();

	Process* process = (Process*)malloc(sizeof(Process) * Process_Cnt);
	for (int i = 0; i < Process_Cnt; i++) process[i] = Copy_Process(Process_Info[i]);

	int Process_End_Cnt = 0;
	int flag = 0;
```

```
	int** CPU_Info = (int**)malloc(sizeof(int) * 2);
	for (int i = 0; i < 2; i++) CPU_Info[i] = (int*)malloc(sizeof(int) * 500);

	int Currunt_CPU_Process = -1;
	int Currunt_IO_Process = -1;
	int CPU_Idle_Time = 0;
	int CPU_Time = 0;

	while (Process_End_Cnt < Process_Cnt) {
		for (int i = 0; i < Process_Cnt; i++) if (CPU_Time == process[i].Arrival_Time) {
			Enqueue(Ready_Queue, i + 1);
			for (int j = 0; j < Ready_Queue->cnt; j++) {
				for (int k = j + 1; k < Ready_Queue->cnt; k++) {
					if (process[Ready_Queue->ID[j] - 1].CPU_Burst_Time > process[Ready_Queue->ID[k] - 1].CPU_Burst_Time) {
						int temp = Ready_Queue->ID[j];
						Ready_Queue->ID[j] = Ready_Queue->ID[k];
						Ready_Queue->ID[k] = temp;
					}
				}
			}
			if (Currunt_CPU_Process != -1 && process[Currunt_CPU_Process - 1].CPU_Burst_Time > process[Ready_Queue->ID[0] - 1].CPU_Burst_Time) {
				CPU_Info[0][flag] = CPU_Time;
				CPU_Info[1][flag] = Currunt_CPU_Process;
```

```
                flag++;
                int temp = Ready_Queue->ID[0];
                Ready_Queue->ID[0] = Currunt_CPU_Process;
                Currunt_CPU_Process = temp;
        }
}

CPU_Time++;

if (Currunt_CPU_Process == -1 && Ready_Queue->cnt == 0) CPU_Idle_Time++;
else {
        if (Currunt_CPU_Process == -1) {
                if (CPU_Idle_Time != 0) {
                        CPU_Info[0][flag] = CPU_Time - 1;
                        CPU_Info[1][flag] = -1;
                        flag++;
                        CPU_Idle_Time = 0;
                }
                Currunt_CPU_Process = Ready_Queue->ID[0];
                Dequeue(Ready_Queue);
        }
        process[Currunt_CPU_Process - 1].CPU_Burst_Time--;
        if (process[Currunt_CPU_Process - 1].CPU_Burst_Time == 0) {
                CPU_Info[0][flag] = CPU_Time;
                CPU_Info[1][flag] = Currunt_CPU_Process;
                flag++;
```

```
                    if (process[Currunt_CPU_Process - 1].CPU_Burst_Time == process[Currunt_CPU_Process - 1].IO_Burst_Timing) {
                        left_IO[4][Currunt_CPU_Process - 1] = -CPU_Time;
                        Enqueue(Waiting_Queue, Currunt_CPU_Process);
                    }
                    Currunt_CPU_Process = -1;
                    Process_End_Cnt++;
                }
                if (process[Currunt_CPU_Process - 1].CPU_Burst_Time == process[Currunt_CPU_Process - 1].IO_Burst_Timing) {
                    left_IO[4][Currunt_CPU_Process - 1] = -CPU_Time;
                    Enqueue(Waiting_Queue, Currunt_CPU_Process);
                    CPU_Info[0][flag] = CPU_Time;
                    CPU_Info[1][flag] = Currunt_CPU_Process;
                    flag++;
                    Currunt_CPU_Process = -1;
                }
            }

            if (Currunt_IO_Process != -1) process[Currunt_IO_Process - 1].IO_Burst_Time--;
            if (Currunt_IO_Process != -1 && process[Currunt_IO_Process - 1].IO_Burst_Time == 0) {
                if (process[Currunt_IO_Process - 1].CPU_Burst_Time > 0) {
                    Enqueue(Ready_Queue, Currunt_IO_Process);
                    for (int i = 0; i < Ready_Queue->cnt; i++) {
                        for (int j = i + 1; j < Ready_Queue->cnt; j++) {
```

```
                                                if (process[Ready_Queue->ID[i] - 1].CPU_Burst_Time > process[Ready_Queue->ID[j] - 1].CPU_Burst_Time) {
                                                        int temp = Ready_Queue->ID[i];
                                                        Ready_Queue->ID[i] = Ready_Queue->ID[j];
                                                        Ready_Queue->ID[j] = temp;
                                                }
                                        }
                                }
                                if (Currunt_CPU_Process != -1 && process[Currunt_CPU_Process - 1].CPU_Burst_Time > process[Ready_Queue->ID[0] - 1].CPU_Burst_Time) {
                                        CPU_Info[0][flag] = CPU_Time;
                                        CPU_Info[1][flag] = Currunt_CPU_Process;
                                        flag++;
                                        int temp = Ready_Queue->ID[0];
                                        Ready_Queue->ID[0] = Currunt_CPU_Process;
                                        Currunt_CPU_Process = temp;
                                }
                        }
                        Currunt_IO_Process = -1;
                }
                if (Currunt_IO_Process == -1 && Waiting_Queue->cnt > 0) {
                        Currunt_IO_Process = Waiting_Queue->ID[0];
                        left_IO[4][Currunt_IO_Process - 1] += CPU_Time;
                        Dequeue(Waiting_Queue);
                }
```

```
        }

        printf("P_SJF:₩n");
        Print_Gantt_Chart(CPU_Info, flag);

        float P_SJf_Average_Waiting_Time = Calculate_Average_Waiting_Time(Process_Info, Process_Cnt, CPU_Info, flag, 4);
        float P_SJF_Average_Turnaround_Time = Calculate_Average_Turnaround_Time(Process_Info, Process_Cnt, CPU_Info, flag);
        float P_SJF_CPU_Utilization = Calculate_CPU_Utilization(CPU_Info, flag);

        Average_Waiting_Time[4] = P_SJf_Average_Waiting_Time;
        Average_Turnaround_Time[4] = P_SJF_Average_Turnaround_Time;
        CPU_Utilization[4] = P_SJF_CPU_Utilization;

        /*
        free(Ready_Queue->ID);
        free(Waiting_Queue->ID);
        free(Ready_Queue);
        free(Waiting_Queue);
        free(process);
        for (int i = 0; i < 2; i++) free(CPU_Info[i]);
        free(CPU_Info);
        */
}
```

```
void P_Priority(Process Process_Info[], int Process_Cnt) {
	QUEUE* Ready_Queue = Create_Queue();
	QUEUE* Waiting_Queue = Create_Queue();

	Process* process = (Process*)malloc(sizeof(Process) * Process_Cnt);
	for (int i = 0; i < Process_Cnt; i++) process[i] = Copy_Process(Process_Info[i]);

	int Process_End_Cnt = 0;
	int flag = 0;

	int** CPU_Info = (int**)malloc(sizeof(int) * 2);
	for (int i = 0; i < 2; i++) CPU_Info[i] = (int*)malloc(sizeof(int) * 500);

	int Currunt_CPU_Process = -1;
	int Currunt_IO_Process = -1;
	int CPU_Idle_Time = 0;
	int CPU_Time = 0;

	while (Process_End_Cnt < Process_Cnt) {
		for (int i = 0; i < Process_Cnt; i++) if (CPU_Time == process[i].Arrival_Time) {
			Enqueue(Ready_Queue, i + 1);
			for (int j = 0; j < Ready_Queue->cnt; j++) {
				for (int k = j + 1; k < Ready_Queue->cnt; k++) {
					if (process[Ready_Queue->ID[j] - 1].Priority < process[Ready_Queue->ID[k] - 1].Priority) {
```

```
                                        int temp = Ready_Queue->ID[j];
                                        Ready_Queue->ID[j] = Ready_Queue->ID[k];
                                        Ready_Queue->ID[k] = temp;
                                    }
                                }
                            }
                        }
                        if (Currunt_CPU_Process != -1 && process[Currunt_CPU_Process - 1].Priority < process[Ready_Queue->ID[0] - 1].Priority) {
                            CPU_Info[0][flag] = CPU_Time;
                            CPU_Info[1][flag] = Currunt_CPU_Process;
                            flag++;
                            int temp = Ready_Queue->ID[0];
                            Ready_Queue->ID[0] = Currunt_CPU_Process;
                            Currunt_CPU_Process = temp;
                        }

                        CPU_Time++;

                        if (Currunt_CPU_Process == -1 && Ready_Queue->cnt == 0) CPU_Idle_Time++;
                        else {
                            if (Currunt_CPU_Process == -1) {
                                if (CPU_Idle_Time != 0) {
                                    CPU_Info[0][flag] = CPU_Time - 1;
                                    CPU_Info[1][flag] = -1;
                                    flag++;
```

```
                    CPU_Idle_Time = 0;
                }
                Currunt_CPU_Process = Ready_Queue->ID[0];
                Dequeue(Ready_Queue);
            }
            process[Currunt_CPU_Process - 1].CPU_Burst_Time--;
            if (process[Currunt_CPU_Process - 1].CPU_Burst_Time == 0) {
                CPU_Info[0][flag] = CPU_Time;
                CPU_Info[1][flag] = Currunt_CPU_Process;
                flag++;
                if (process[Currunt_CPU_Process - 1].CPU_Burst_Time == process[Currunt_CPU_Process - 1].IO_Burst_Timing) {
                    left_IO[5][Currunt_CPU_Process - 1] = -CPU_Time;
                    Enqueue(Waiting_Queue, Currunt_CPU_Process);
                }
                Currunt_CPU_Process = -1;
                Process_End_Cnt++;
            }
            if (process[Currunt_CPU_Process - 1].CPU_Burst_Time == process[Currunt_CPU_Process - 1].IO_Burst_Timing) {
                left_IO[5][Currunt_CPU_Process - 1] = -CPU_Time;
                Enqueue(Waiting_Queue, Currunt_CPU_Process);
                CPU_Info[0][flag] = CPU_Time;
                CPU_Info[1][flag] = Currunt_CPU_Process;
                flag++;
                Currunt_CPU_Process = -1;
```

```
                }
            }

            if (Currunt_IO_Process != -1) process[Currunt_IO_Process - 1].IO_Burst_Time--;
            if (Currunt_IO_Process != -1 && process[Currunt_IO_Process - 1].IO_Burst_Time ==
0) {
                if (process[Currunt_IO_Process - 1].CPU_Burst_Time > 0) {
                    Enqueue(Ready_Queue, Currunt_IO_Process);
                    for (int i = 0; i < Ready_Queue->cnt; i++) {
                        for (int j = i + 1; j < Ready_Queue->cnt; j++) {
                            if (process[Ready_Queue->ID[i] - 1].Priority <
process[Ready_Queue->ID[j] - 1].Priority) {
                                int temp = Ready_Queue->ID[i];
                                Ready_Queue->ID[i] = Ready_Queue-
>ID[j];
                                Ready_Queue->ID[j] = temp;
                            }
                        }
                    }
                    if (Currunt_CPU_Process != -1 && process[Currunt_CPU_Process
- 1].Priority > process[Ready_Queue->ID[0] - 1].Priority) {
                        CPU_Info[0][flag] = CPU_Time;
                        CPU_Info[1][flag] = Currunt_CPU_Process;
                        flag++;
                        int temp = Ready_Queue->ID[0];
                        Ready_Queue->ID[0] = Currunt_CPU_Process;
                        Currunt_CPU_Process = temp;
```

```
                        }
                    }
                    Currunt_IO_Process = -1;
                }
                if (Currunt_IO_Process == -1 && Waiting_Queue->cnt > 0) {
                    Currunt_IO_Process = Waiting_Queue->ID[0];
                    left_IO[5][Currunt_IO_Process - 1] += CPU_Time;
                    Dequeue(Waiting_Queue);
                }
            }

            printf("P_Priority:₩n");
            Print_Gantt_Chart(CPU_Info, flag);

            float P_Priority_Average_Waiting_Time = Calculate_Average_Waiting_Time(Process_Info, Process_Cnt, CPU_Info, flag, 5);
            float P_Priority_Average_Turnaround_Time = Calculate_Average_Turnaround_Time(Process_Info, Process_Cnt, CPU_Info, flag);
            float P_Priority_CPU_Utilization = Calculate_CPU_Utilization(CPU_Info, flag);

            Average_Waiting_Time[5] = P_Priority_Average_Waiting_Time;
            Average_Turnaround_Time[5] = P_Priority_Average_Turnaround_Time;
            CPU_Utilization[5] = P_Priority_CPU_Utilization;

            /*
            free(Ready_Queue->ID);
```

```
		free(Waiting_Queue->ID);
		free(Ready_Queue);
		free(Waiting_Queue);
		free(process);
		for (int i = 0; i < 2; i++) free(CPU_Info[i]);
		free(CPU_Info);
		*/
}

void Schedule(Process Process_Info[], int Process_Cnt, int quantum) {
		FCFS(Process_Info, Process_Cnt);
		SJF(Process_Info, Process_Cnt);
		Priority(Process_Info, Process_Cnt);
		RR(Process_Info, Process_Cnt, quantum);
		P_SJF(Process_Info, Process_Cnt);
		P_Priority(Process_Info, Process_Cnt);
}

void Evaluation() {
		float temp;
		char Rank_Name[6][11];
		strcpy(Rank_Name[0], "FCFS");
		strcpy(Rank_Name[1], "SJF");
		strcpy(Rank_Name[2], "Priority");
		strcpy(Rank_Name[3], "RR");
		strcpy(Rank_Name[4], "P_SJF");
```

```
strcpy(Rank_Name[5], "P_Priority");


for (int i = 0; i < 6; i++) {
	for (int j = i + 1; j < 6; j++) {
		if (Average_Waiting_Time[i] > Average_Waiting_Time[j]) {
			temp = Average_Waiting_Time[i];
			Average_Waiting_Time[i] = Average_Waiting_Time[j];
			Average_Waiting_Time[j] = temp;


			temp = Rank_Waiting_Time[i];
			Rank_Waiting_Time[i] = Rank_Waiting_Time[j];
			Rank_Waiting_Time[j] = temp;
		}
		if (Average_Turnaround_Time[i] > Average_Turnaround_Time[j]) {
			temp = Average_Turnaround_Time[i];
			Average_Turnaround_Time[i] = Average_Turnaround_Time[j];
			Average_Turnaround_Time[j] = temp;


			temp = Rank_Turnaround_Time[i];
			Rank_Turnaround_Time[i] = Rank_Turnaround_Time[j];
			Rank_Turnaround_Time[j] = temp;
		}


		if (CPU_Utilization[i] < CPU_Utilization[j]) {
			temp = CPU_Utilization[i];
			CPU_Utilization[i] = CPU_Utilization[j];
```

```
                    CPU_Utilization[j] = temp;


                    temp = Rank_CPU_Utilization[i];
                    Rank_CPU_Utilization[i] = Rank_CPU_Utilization[j];
                    Rank_CPU_Utilization[j] = temp;
                }

            }
        }

        printf("₩nRank of Average Waiting Time: ₩n");
        for (int i = 0; i < 6; i++) {
            printf("%d. %s : %.1f₩n", i + 1, Rank_Name[Rank_Waiting_Time[i]], Average_Waiting_Time[i]);
        }
        printf("₩n");

        printf("₩nRank of Average Turnaround Time: ₩n");
        for (int i = 0; i < 6; i++) {
            printf("%d. %s : %.1f₩n", i + 1, Rank_Name[Rank_Turnaround_Time[i]], Average_Turnaround_Time[i]);
        }
        printf("₩n");


        printf("₩nRank of CPU Utilization: ₩n");
```

```
	for (int i = 0; i < 6; i++) {
		printf("%d. %s : %.0f%%₩n", i + 1, Rank_Name[Rank_CPU_Utilization[i]], CPU_Utilization[i] * 100);
	}
	printf("₩n");

}

int main(void) {
	int Process_Cnt = 5;
	int quantum = 4;

	srand(time(NULL));

	Process Process_Info[5];
	for (int i = 0; i < Process_Cnt; i++) {
		Process_Info[i] = Create_Process(i + 1);
	}

	Print_Process(Process_Info, Process_Cnt);

	Schedule(Process_Info, Process_Cnt, quantum);

	Evaluation();

	return 0;
```

}