Report

제출일 2019년 4월 15일
과목명 운영체제
담당교수 최 종 무 교 수 님
전공 공대 소프트웨어학과
학번 32143180, 32144548
이름 이 명 재, 조 창 연

1. Team

팀 구성원	내용
이명재 (32143180)	Round Robin Scheduling 구현 MLFQ Scheduling 구현 보고서 작성 & 소스 묶음 작업
조창연 (32144548)	FCFS Scheduling 구현 Lottery Scheduling 구현 보고서 작성 & 마무리

2. Description

본 소스코드에선 최대 수행시간(total_time)을 나타내는 변수와 가장 마지막으로 프로세스가 종료된 시점(deadline)을 나타내는 변수, 현재 CPU에 프로세스가 올라와있는지 체크하는 변수(process)와 모든 프로세스의 스케줄링이 종료되었는지 확인하는 변수(allstop), 출력 데이터(throughtput)를 담는 2차원 배열을 각 함수별 공통변수로 가진다.

➜ FIFO (First In, First Out)

FIFO는 스케줄링 기법 중 가장 명백하고 단순하며 구현하기에도 굉장히 쉬운 것 중 하나로 먼저 들어온 프로세스가 먼저 수행하는 스케줄링입니다. 해당 스케줄링 기법은 FCFS (First Come First Serve)라고 부르기도 하며, 비선점형 스케줄링으로 프로세스가 한 번 스케줄되면 그 프로세스가 끝날 때까지 다른 프로세스가 끼어들 수 없는 방식입니다.

먼저 변수들을 선언한 후 메모리 동적할당을 통해 프로세스의 도착시간과 수행시간을 받습니다. 그리고나서 모든 프로세스의 도착시간과 수행시간을 더하여 최대 수행시간을 구합니다.

프로세스가 cpu에 올라갈 때 걸리는 최대 시간을 구하고, 현재 진행 시간에 프로세스가 도착했는지 또 스케줄링 할 프로세스의 처리시간이 남아있는지 확인합니다. 그리고나서 process 변수에 데이터가 들어갈 수 있으면, data를 삽입합니다.

만약 process 변수에 처리해야 할 프로세스가 존재한다면 프로세스의 실행시간을 1씩 줄이고 throughtput data에 입력합니다. 이 후 모든 프로세스의 스케줄링이 종료되었다면 프로세스가 종료된 마지막 시점으로부터 총 스케줄링 시간과 deadline 변수의 시간이 1 차이가 나기 때문에 ++i를 통해 증가시켜줍니다.

다음과 같은 과정을 계속 반복하면서 스케줄링이 진행이 되며, 모든 과정이 끝났다면 모든 프로세스 스케줄링이 끝났다고 판단하고 프로그램을 종료합니다.

➜ RR (Round-robin)

RR은 정해진 타임 슬라이스에 따라 도착한 프로세스를 타임슬라이스 만큼 수행 한 뒤, 기아상태를 측정해 우선순위를 두어서 해당 프로세스를 타임 슬라이스만큼 수행하는 것을 반복하는 스케줄링입니다. 타임슬라이스를 줄일수록 반응시간이 짧아져서 사용자에게 즉각적으로 반응이 가게 되는데 너무 짧아지면 context switch를 자주해야하기 때문에 복잡해집니다. 이 프로그램에선 타임 슬라이스를 1로 두었으며 이는 소스코드에서 수정이 가능합니다.

먼저 변수들을 선언한 후 메모리 동적할당을 통해 프로세스의 도착시간과 수행시간을 받습니다. 그리고나서 모든 프로세스의 도착시간과 수행시간을 더하여 최대 수행시간을 구합니다.
현재 cpu에 할당된 프로세스가 전부 완료되었거나, 타임슬라이스만큼 수행이 되었는지를 확인한 후 둘 중 하나가 만족하면 해당 사항들을 체크하는 변수값들을 초기화 한 후 현재 시간에 도착한 작업이 있는지 판단하고 먼저 도착한 프로세스들 순으로 직전에 수행되었는지 검사 후 기아값을 증가시킵니다. 이를 통해 수행된지 오래된 프로세스일수록 기아값이 늘어나서 우선순위가 높아지게 됩니다.
이 과정을 통해 현재 시간의 프로세스들의 우선순위를 지정하고 cpu에 프로세스를 초기화합니다.
우선순위에 대한 지정이 끝났으니 현재 시간에 대해 도착한 프로세스가 있는지, 그 프로세스가 수행해야할 작업이 남아있는지 확인 한 후 cpu에 프로세스가 올라가 있지 않다면 프로세스를 cpu에 올립니다. 만약 수행가능 한 다른 프로세스의 기아 값이 더 높다면 변경합니다.
이 과정이 한 사이클 돌 때마다 각 프로세스가 끝났는지 체크해서 allstop변수를 증가시켜 줍니다. 이 allstop변수가 총 프로세스갯수가 되면 종료하게끔 설계되어있습니다.
그 뒤 cpu에 최종적으로 놓여진 프로세스를 수행하고 실행시간을 1줄인 뒤 출력을 위한 throughput에 데이터를 입력합니다.
이 과정을 반복하다 모든 프로세스가 종료하면 공통된 출력문을 통해 결과를 출력합니다.

➜ MLFQ (Muti-Level Feedback Queue)

MLFQ는 다중 큐를 이용한 스케줄링으로 프로세스가 도착하면 상위 큐에 도착하며 종료 전 까지 타임 슬라이스 만큼 수행할 때마다 하위레벨의 큐로 우선순위가 내려가게 됩니다. 이는 우선순위를 적절히 분배해 최적의 프로세스 수행을 하도록 도와줍니다. 오랫동안 수행되는 프로세스들은 상위레벨에 많은 프로세스가 존재하면 오랫동안 수행이 안되는 기아상태에 빠질수 있는데 이를 해결 하기위해 주기적으로 하위레벨에 존재하는 프로세스들을 상위레벨로 올려주는 부스팅이 수행되기도 합니다.

먼저 변수들을 선언한 후 메모리 동적할당을 통해 프로세스의 도착시간과 수행시간을 받습니다. 그리고나서 모든 프로세스의 도착시간과 수행시간을 더하여 최대 수행시간을 구합니다.
우선 현재 시간에 스케줄링의 대상이 될 수 있는 프로세스의 개수를 구합니다.(emptyq) 만약 현재 새롭게 들어온 프로세스라면 우선순위를 최우선으로 두고 cpu에 올라간 프로세스가 없다면 할당을 해줍니다. 만약 현재 시간에 도착한 프로세스가 없다면 수행가능한 프로세스가 있는지 확인해서 선점된 프로세스가 있는지 체크 후 할당합니다. 이렇게 할당된 프로세스를 현재시간에 수행 가능한 다른 프로세스들과 우선순위를 비교해서 우선순위가 더 높은 프로세스를 할당합니다. 이를 pr[process]를 통해 우선순위를 정하는데 pr[]이 높을수록 하위단계에 해당하도록 프로세스가 수행될 때 마다 해당 값을 증가시켜줘서 우선순위를 비교하도록 합니다. 기아상태 또한 고려하며 우선순위를 정해야하며 이를 고려하지 않으면 기아상태에 빠지는 프로세스가 생길 수 있습니다.
cpu에 할당된 프로세스를 수행하며 기아상태를 초기화해주며 타임슬라이스 만큼 수행되거나 모든 실행이 끝나게 되면 cpu에서 내려오고 타임슬라이스 체크 변수를 초기화하며 수행가능한 프로세스가 1개보다 많다면 해당 프로세스의 우선순위를 낮춥니다.
그 뒤 cpu에 최종적으로 놓여진 프로세스를 수행하고 실행시간을 1줄인 뒤 출력을 위한 throughput에 데이터를 입력합니다.
이 과정을 반복하다 모든 프로세스가 종료하면 공통된 출력문을 통해 결과를 출력합니다.

➜ Lottery (Bonus)

Lottery는 티켓이라는 메카니즘을 통해 각 프로세스별로 부여하고, 무작위로 뽑는 스케줄링 기법입니다. 이는 많은 티켓을 가지고 있으면 승리할 확률이 높아집니다. 또한, 오랫동안 돌리게되면 원하는 확률게 가까워지지만, 확률이기 때문에 오차가 발생할 수 있습니다. 여기서 ticket 은 자원의 몫을 나타내며, 서로 다른 사용자에게도 공평하게 사용 가능하도록 기존의 자원과 비교하여 각 프로세스별로 배분해주고, 다른 일의 스케줄링 확률을 높여줄 수도 있게 해줍니다. 해당 스케줄링 기법은 MLFQ의 단점인 기아상태를 해결할 수 있습니다.

먼저 변수들을 선언한 후 메모리 동적할당을 통해 프로세스의 도착시간과 수행시간 그리고 스케줄링에 필요한 티켓을 받습니다. 이 때 티켓은 매시간 랜덤하게 돌아갑니다. 그리고나서 전체 티켓의 합을 구하고, 티켓의 구간을 정해줍니다. 티켓의 합과 구간 설정이 다 되었다면, 모든 프로세스의 도착시간과 수행시간을 더하여 최대 수행시간을 구합니다. 그럼 이제 최대 수행시간동안 티켓이 어떻게 뽑히는지 알아보겠습니다.
해당 프로그램의 에러를 방지하고자 기본적으로 allstop 변수를 초기화시켜주고, 모든 프로세스의 스케줄링이 끝났는지 확인을 시켜줍니다. 그런 후에 랜덤 티켓을 생성합니다. 그러면 임의의 각 구간별로 티켓이 생성이 될 텐데, 이때 티켓이 어느 값 사이에 위치하는지 모르기 때문에 확인작업이 필요합니다. 각 구간별로 티켓의 위치를 확인하면서 내가 정한 값이 프로세스가 도착하고 실행할 시간이 남아있다면 다음 실행시킬 프로세스로 정하고 아무 곳이나 값이 교체하는게 아니라 다 돌았는데도 티켓 사이 값에 해당하는 프로세스를 못 찾았을 경우 다시 뽑습니다. 다시 뽑았기 때문에 변수 i 를 1씩 감소하면서 다시 비교합니다. 이 과정이 전체 수행시간동안 계속 반복됩니다.
만약 process 변수에 처리해야 할 프로세스가 존재한다면 프로세스의 실행시간을 1씩 줄이고 throughtput data에 입력합니다. 이 후 모든 프로세스의 스케줄링이 종료되었다면 프로세스가 종료된 마지막 시점으로부터 총 스케줄링 시간과 deadline 변수의 시간이 1 차이가 나기 때문에 ++i를 통해 증가시켜줍니다.
모든 과정이 끝났다면 모든 프로세스 스케줄링이 끝났다고 판단하고 프로그램을 종료합니다.

3. Demonstration

처음 어려웠던 부분은 일정 조율을 하는데 많은 어려움을 겪었습니다. 우리가 본 과제를 위해 처음 미팅을 가졌던 날이 과제가 나오고 10일이 지난 후인 4월 6일 토요일이였으며, 두 번째 미팅이 4월 8일 월요일이었는데, 실질적으로 두 번째 미팅이 마지막이었습니다. 그 이후로는 서로 만날 수 있는 시간이 없어서 연락을 주고받으며 과제를 진행하였습니다.

두번째로 어려웠던 부분은 제가 과제가 나오던 날 대외행사로 인해 수강하지 못하여 Lottery 스케줄링에 대해서 알고 있는 내용이 전혀 없었고, 또 앞서 배웠던 FIFO, RR, MLFQ 스케줄링에 대한 개념도 완전히 잡혀 있질 않아 본 과제를 처음 시작할 때 많은 어려움이 있었으나, 구글이나 깃허브의 여러 가지 글을 보면서 개념을 보강하였습니다.

마지막으로 어려웠던 부분은 바로 RR 스케줄링과 MLFQ 스케줄링의 코드 작성이었습니다. 출력 결과물을 뽑아내는 것은 어렵지 않았지만, 프로세스 우선순위를 정해주는 알고리즘이다보니 어려움이 있었습니다. 만약 소스 코드를 조금만 잘못 작성했다면 완전히 다른 결과를 보여줬을 것입니다. 또한, 다르게 작성해더라도 출력하고자 하는 워크로드가 크거나 길지 않아서 우리가 우선순위를 모를 수 있는 문제도 발생할 수 있게 됩니다.

4. Result

```
이 프로그램은 5개의 데이터 셋을 사용합니다.
데이터 셋의 갯수는 소스에서 수정이 가능합니다.
1. FIFO
2. RR
3. MLFQ
4. Lottery
5. Exit
>>>1
Enter the 'Arrival-Time' and 'Service(Run) time': 0 3
Enter the 'Arrival-Time' and 'Service(Run) time': 2 6
Enter the 'Arrival-Time' and 'Service(Run) time': 4 4
Enter the 'Arrival-Time' and 'Service(Run) time': 6 5
Enter the 'Arrival-Time' and 'Service(Run) time': 8 2
         0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
    ------------------------------------------------------------------------------------
 [A] >> ■ ■ ■ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □
 [B] >> □ □ □ ■ ■ ■ ■ ■ ■ □ □ □ □ □ □ □ □ □ □ □
 [C] >> □ □ □ □ □ □ □ □ □ ■ ■ ■ ■ □ □ □ □ □ □ □
 [D] >> □ □ □ □ □ □ □ □ □ □ □ □ □ ■ ■ ■ ■ ■ □ □
 [E] >> □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ ■ ■

>>>2
Enter the 'Arrival-Time' and 'Service(Run) time': 0 3
Enter the 'Arrival-Time' and 'Service(Run) time': 2 6
Enter the 'Arrival-Time' and 'Service(Run) time': 4 4
Enter the 'Arrival-Time' and 'Service(Run) time': 6 5
Enter the 'Arrival-Time' and 'Service(Run) time': 8 2
RR 의 현재 Time quantom은 1 입니다. 소스에서 수정이 가능합니다.
         0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
    ------------------------------------------------------------------------------------
 [A] >> ■ ■ □ ■ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □
 [B] >> □ □ ■ □ ■ □ ■ □ □ ■ □ □ □ ■ □ □ □ ■ □ □
 [C] >> □ □ □ □ □ ■ □ □ ■ □ □ □ ■ □ □ □ ■ □ □ □
 [D] >> □ □ □ □ □ □ □ ■ □ □ □ ■ □ □ □ ■ □ □ ■ ■
 [E] >> □ □ □ □ □ □ □ □ □ □ ■ □ □ □ ■ □ □ □ □ □

>>>3
Enter the 'Arrival-Time' and 'Service(Run) time': 0 3
Enter the 'Arrival-Time' and 'Service(Run) time': 2 6
Enter the 'Arrival-Time' and 'Service(Run) time': 4 4
Enter the 'Arrival-Time' and 'Service(Run) time': 6 5
Enter the 'Arrival-Time' and 'Service(Run) time': 8 2
arrival_timeLFQ 의 현재 Time quantom은 1 입니다. 소스에서 수정이 가능합니다.
         0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
    ------------------------------------------------------------------------------------
 [A] >> ■ ■ □ ■ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □
 [B] >> □ □ ■ □ □ ■ □ □ □ □ □ ■ □ □ ■ □ □ ■ □ ■ □
 [C] >> □ □ □ □ ■ □ □ ■ □ □ □ □ ■ □ □ ■ □ □ □ □ □
 [D] >> □ □ □ □ □ □ ■ □ □ ■ □ □ □ ■ □ □ ■ □ □ □ □
 [E] >> □ □ □ □ □ □ □ □ ■ □ ■ □ □ □ □ □ □ □ □ □ □

>>>4
pick the ticket: 10
pick the ticket: 40
pick the ticket: 20
pick the ticket: 30
pick the ticket: 50
Enter the 'Arrival-Time' and 'Service(Run) time': 0 3
Enter the 'Arrival-Time' and 'Service(Run) time': 2 6
Enter the 'Arrival-Time' and 'Service(Run) time': 4 4
Enter the 'Arrival-Time' and 'Service(Run) time': 6 5
Enter the 'Arrival-Time' and 'Service(Run) time': 8 2
This is the ticket list that give each process
A :10 B :40 C :20 D :30 E :50
         0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
    ------------------------------------------------------------------------------------
 [A] >> ■ ■ □ □ □ ■ □ □ □ □ □ □ □ □ □ □ □ □ □ □
 [B] >> □ □ ■ ■ ■ □ ■ □ ■ ■ □ □ □ □ □ □ □ □ □ □
 [C] >> □ □ □ □ □ □ □ □ □ □ □ ■ □ □ ■ □ □ □ ■ ■
 [D] >> □ □ □ □ □ □ □ ■ □ □ ■ □ ■ ■ □ □ □ ■ □ □
 [E] >> □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ ■ ■ □ □ □
```

5. Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <time.h>
#include <sys/time.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <assert.h>
#include <pthread.h>
#include <math.h>
#define arrival_time 6 // number of data
#define run_time 2
#define size 6


void FIFO();
void RR();
void MLFQ();
void Lottery();
int int_pow(int base, int exp);

int main(int argc, char *argv[]){

    int choice = 0;


    while (1) {
        printf("이 프로그램은 5개의 데이터 셋을 사용합니다.₩n");
        printf("데이터 셋의 갯수는 소스에서 수정이 가능합니다.₩n");
        printf("1. FIFO₩n");
        printf("2. RR₩n");
        printf("3. MLFQ₩n");
        printf("4. Lottery₩n");
        printf("5. Exit₩n");
        printf (">>>");
        scanf("%1d",&choice);

        while(1) {
            if (choice < 6) break;
            else {
                    printf("번호를 다시 입력하세요.₩n₩n₩n");
                    printf("1. FIFO₩n");
                    printf("2. RR₩n");
                    printf("3. MLFQ₩n");
                    printf("4. Lottery₩n");
                    printf("5. Exit₩n");
                    printf (">>>");
                    scanf("%1d",&choice);
            }
        }
        switch(choice) {
            case 1:
                FIFO();
                break;
            case 2:
                RR();
                break;
            case 3:
                MLFQ();
                break;
            case 4:
                Lottery();
```

```
                break;
            case 5:
                return 0;
        }
    }
}

int int_pow(int base, int exp)
{
    int result = 1;
    while (exp)
    {
        if (exp & 1)
            result *= base;
        exp /= 2;
        base *= base;
    }
    return result;
}

void FIFO()
{
        int i,data,j,k,l;
        int total_time = 0; // all time
        int process = 0; // now process start
        int deadline = 0; // last process exit space
        int allstop = 0; // all scheduling exit
        int throughput[arrival_time][100] = {}; // throughput data
        int **arr;

        arr = (int **) calloc (sizeof (int*), arrival_time);

        for(i = 0; i < arrival_time; i++) {
                arr[i] = (int*) calloc ( sizeof(int), run_time);
        }

        for (i = 1; i < arrival_time; i++) {
                printf("Enter the 'Arrival-Time' and 'Service(Run) time': ");
                for(j = 0; j < run_time; j++) {
                        scanf("%d", &arr[i][j]);
                }
        }

        for (i = 1; i < arrival_time; i++) {
                total_time += arr[i][1] + arr[i][0];
        }

        // all turnaroud time
        for (i = 0; i < total_time; i++) {
                process = 0;
                for (data = 1; data < arrival_time; data++) {
                        if (i >= arr[data][0] && arr[data][1] >= 1) {
                                if (process == 0) {
                                        process = data;
                                        break;
                                }
                        }
                }

                if (process > 0) {
                        arr[process][1]--;
                        throughput[process][i] = 1;
                }
                // all scheduling exit checking
                if (arr[arrival_time - 1][1] < 1) {
                        deadline = ++i;
                        break;
                }
```

```
        }

        printf("            ");
    for (l = 0; l < deadline; l++)  {
            printf("%2d   ",l);
        }
        printf("₩n      ");

        for (l = 0; l < deadline; l++)  {
            printf("------");
        }
    printf("₩n");

    for (i = 1; i < arrival_time; i++) {
        printf(" [%c] >> ",i+64);
        for (k = 0; k < deadline; k++)  {
            if (throughput[i][k] == 1) {
                printf("■   ");
            }
            else
                printf("□   ");
        }
        printf("₩n");
    }
}

void RR()
{
        int i,data,j,k,l;
        int total_time = 0; // all time
        int process = 0; // now process start
        int deadline = 0; // last process exit space
        int allstop = 0; // all scheduling exit
        int throughput[arrival_time][100] = {}; // throughput data
        int **arr;
    int hunger[arrival_time] = {0}; // 기아상태 측정용
    int check[arrival_time] = {0}; // 프로세스가 타임슬라이스를 다 사용했는지 체크
    int timeslice = 1;

        arr = (int **) calloc (sizeof (int*), arrival_time);

        for(i = 0; i < arrival_time; i++) {
            arr[i] = (int*) calloc ( sizeof(int), run_time);
        }

        for (i = 1; i < arrival_time; i++) {
            printf("Enter the 'Arrival-Time' and 'Service(Run) time': ");
            for(j = 0; j < run_time; j++) {
                scanf("%d", &arr[i][j]);
            }
        }


    for (i = 1; i < arrival_time; i++) {
        total_time += arr[i][1] + arr[i][0];
    }

    for (i = 0; i < total_time; i++) {
        if (check[process] % timeslice == 0 || arr[process][1] < 1) { // 프로세스가 타임슬라이스 만큼 실행
되었거나 수행이 완료되면
            check[process] = 0; // 타임슬라이스를 초기화
            hunger[process] = 0; // 스케쥴링이 되어 기아상태가 사라짐
            for (k = 1; k < arrival_time; k++) {
                if (i >= arr[k][0] && k != process) // 현재 진행 시간에 작업이 도착했는지 판단하고 k 값
이 직전에 스케쥴링되었는지 확인
                    hunger[k]++; // 스케쥴링이 직전에 되지 않았다면 기아상태를 증가시킴
            }
        }
```

```
        allstop = 0; // 초기화를 해 주어야 매 루프마다 모든 프로세스의 스케쥴링이 끝났는지 체크 가능
        process = 0; // 실행할 프로세스를 고르기 전에 초기화

        for (j = 1; j < arrival_time; j++) {
            if (i >=arr[j][0] && arr[j][1] >= 1) { // 현재 진행 시간에 프로세스가 도착했는지 판단하고 스케쥴링 할 프로세스의 처리시간이 남았는지 확인
                if (process == 0) { // 실행할 프로세스를 아직 고르지 못했다면
                    process = j; // 프로세스를 고름
                }
                else if (hunger[process] < hunger[j]) { // 프로세스를 이미 골랐음에도 지금 스케쥴링할 다른 프로세스가 존재한다면 배고픔 정도를 비교
                    process = j; // 더 배고프다면 스케쥴링할 프로세스를 교체
                }
            }

            if (arr[j][1] < 1) // 모든 프로세스 스케쥴링이 끝났는지 체크
                allstop++;
        }

        if (process > 0) { // 레디 큐에 처리할 프로세스가 존재한다면
            arr[process][1]--; // 프로세스의 실행시간을 1 줄이고
            throughput[process][i] = 1; // 아웃풋 데이터에 입력
            check[process]++;
        }

        if (allstop == arrival_time-1) { // 스케쥴링이 종료되었다면
            deadline = ++i; // deadline이 현재시간을 기억하고 루프 종료
            break;
        }
    }

    printf("RR 의 현재 Time quantom은 1 입니다. 소스에서 수정이 가능합니다.₩n");
    printf("          ");
    for (l = 0; l < deadline; l++)  {
        printf("%2d  ",l);
    }
    printf("₩n      ");

    for (l = 0; l < deadline; l++)  {
        printf("------");
    }
    printf("₩n");

    for (i = 1; i < arrival_time; i++) {
        printf(" [%c] >> ",i+64);
        for (k = 0; k < deadline; k++)  {
            if (throughput[i][k] == 1) {
                printf("■  ");
            }
            else
                printf("□  ");
        }
        printf("₩n");
    }
}



void MLFQ()
{
    int i,data,j,k,l;
        int total_time = 0; // all time
        int process = 0; // now process start
        int deadline = 0; // last process exit space
        int allstop = 0; // all scheduling exit
        int throughput[arrival_time][100] = {}; // throughput data
        int **arr;
        int hunger[arrival_time] = {0}; // 기아상태 측정용
```

```
    int check[arrival_time] = {0}; // 프로세스가 타임슬라이스를 다 사용했는지 체크
    int timeslice = 1;
        int pr[arrival_time] = { 0 }; // 처리된 회수를 저장
        int chktime = 0; // 현재 프로세스가 타임슬라이스이내에서 실행되고있는지
        int emptyq = 0; // 현재 큐에 단 하나의 프로세스만 존재하는지
        arr = (int **) calloc (sizeof (int*), arrival_time);

        for(i = 0; i < arrival_time; i++) {
                arr[i] = (int*) calloc ( sizeof(int), run_time);
        }

        for (i = 1; i < arrival_time; i++) {
                printf("Enter the 'Arrival-Time' and 'Service(Run) time': ");
                for(j = 0; j < run_time; j++) {
                        scanf("%d", &arr[i][j]);
                }
        }

    for (i = 1; i < arrival_time; i++) {
        total_time += arr[i][1] + arr[i][0];
    }

        for (i = 0; i < total_time; i++) {

                if (chktime == 0) { // cpu에 프로세스가 올라가 있는지 확인합니다.
                        for (k = 1; k < arrival_time; k++) {
                                if (i >= arr[k][0] && k != process) {// 현재 진행 시간에 프로세스가 도착했는지 판단하고 k 값이 직전에 스케쥴링되었는지 확인
                                        hunger[k]++; // 스케쥴링이 직전에 되지 않았다면 배고픔을 증가시킴
                                        check[k] = 0; // 스케쥴링이 이뤄지지 않아서 타임퀀텀 초기화
                                }
                        }
                }

                if (chktime == 0) // cpu에 선점되었는지 확인
                        process = 0; // 실행할 프로세스를 고르기 전에 초기화
                emptyq = 0; // 현재 스케쥴링의 대상인 프로세스 수
        allstop = 0; // 초기화를 해 주어야 매 루프마다 모든 프로세스의 스케쥴링이 끝났는지 체크 가능

                for (j = 1; j < arrival_time; j++) {
                        if (i+1 >= arr[j][0] && arr[j][1] > 0) { emptyq++; } // 현재 스케쥴링의 대상이 될 수 있는 프로세스 조사

                        if (i == arr[j][0] && arr[j][1] >= 1) { // 새롭게 들어온 프로세스라면
                                hunger[j] = 9999; // 가장 높은 우선순위 할당
                                if (process == 0) { // 현재 스케쥴링할 대상을 정하지 않은 상태라면
                                        process = j; // 후보에 넣음
                                }
                        } // 새로 들어온 프로세스는 가장 우선순위 큐에 들어감

                        if (i >= arr[j][0] && arr[j][1] >= 1) { // 현재 진행 시간에 프로세스가 도착했는지 판단하고 스케쥴링 할 프로세스의 처리시간이 남았는지 확인
                                if (process == 0 && chktime == 0) { // 실행할 프로세스를 아직 고르지 못했고 선점된 프로세스가 없다면
                                        process = j; // 프로세스를 고름
                                }
                                else if (pr[process] > pr[j] && hunger[j] != 0) { // 실행될 프로세스가 새로 발견한 프로세스보다 더 낮은 우선순위의 큐에 있는지 확인하고 배고픔을 체크 (배고픔을 체크 안할경우 더 낮은 상태의 큐가 무조건 먼저 돌아감)
                                        if (chktime == 0) { // 선점되지 않았다면
                                                process = j;
                                        } // 현재 process에 들어간 프로세스가 타임퀀텀만큼 충분히 돌았는가 판단
                                }
                        }

                if (arr[j][1] < 1)  // 모든 프로세스 스케쥴링이 끝났는지 체크
```

```
                allstop++;
                }

                if (process > 0) { // 스케쥴링 후보에 오른 프로세스를 스케쥴링함
                        arr[process][1]--; // 프로세스의 실행시간을 1 줄이고
                        throughput[process][i] = 1; // 아웃풋 데이터에 입력
                        check[process]++; // 프로세스가 타임퀀텀 사용
                        chktime = 1; // 현재 프로세스가 cpu를 사용 중임
                        hunger[process] = 0; // 스케쥴링이 되어 배고픔이 사라짐
                        if (check[process] == (int)int_pow(timeslice, pr[process]) || arr[process][1] < 1) { //
프로세스가 타임퀀텀만큼 다 돌았으면 우선순위 낮은 큐로 들어감
                                check[process] = 0; // 주어진 시간만큼 스케쥴링을 다 했기 때문에 초기화
                                chktime = 0; // 프로세스가 cpu에서 내려감
                                if (emptyq > 1) // 단 하나의 프로세서만 스케쥴링의 대상이 아니라면
                                pr[process]++; // 프로세스의 우선순위를 낮춤
                        }
                }

        if (allstop == arrival_time-1) { // 스케쥴링이 종료되었다면
            deadline = ++i; // deadline이 현재시간을 기억하고 루프 종료
            break;
        }
        }

    printf("arrival_timeLFQ 의 현재 Time quantom은 1 입니다. 소스에서 수정이 가능합니다.₩n");
        printf("          ");
    for (k = 0; k < deadline; k++)  {
            printf("%2d  ",k);
        }
    printf("₩n      ");

    for (k = 0; k < deadline; k++)  {
            printf("------");
        }
    printf("₩n");

    for (i = 1; i < arrival_time; i++) {
        printf(" [%c] >> ",i+64);
        for (k = 0; k < deadline; k++)  {
            if (throughput[i][k] == 1) {
                printf("■  ");
            }
            else
                printf("□  ");
        }
        printf("₩n");
    }
}

void Lottery()
{

    int ticket_sum = 0; //all ticket
    int scpr[size] = {}; // ticket space
    int pick = -1; // pick the ticket
    int i,j,k,l;
    int total_time = 0; // all time
    int process = 0; // now process
    int deadline = 0; // exit space that last process
    int allstop = 0; // all process scheduling exit
    int throughput[arrival_time][100] = {}; // throughput data
    int **arr;

    arr = (int **) calloc (sizeof (int*), arrival_time);

        int *ticket = malloc(sizeof(int) * size);
    for (int i = 1; i < size; i++) {
        printf("pick the ticket: ");
```

```
        ticket[i] = i;
        scanf("%d", &ticket[i]);
}

for(i = 0; i < arrival_time; i++) {
        arr[i] = (int*) calloc ( sizeof(int), run_time);
}

for (i = 1; i < arrival_time; i++) {
        printf("Enter the 'Arrival-Time' and 'Service(Run) time': ");
        for(j = 0; j < run_time; j++) {
                scanf("%d", &arr[i][j]);
        }
}

srand((unsigned int)time(NULL));

printf("This is the ticket list that give each process₩n");
for (i = 1; i < arrival_time; i++) {
        ticket_sum += ticket[i];
        printf("%c :%d ",i+64,ticket[i]);
}
printf("₩n");
for (i = 1; i < arrival_time; i++) {
        scpr[i] = scpr[i-1] + ((float)ticket[i] / ticket_sum + 0.005) * 100;
}
scpr[arrival_time-1] += 1;

for (i = 1; i < arrival_time; i++) {
        total_time += arr[i][1] + arr[i][0];
}


for (i = 0; i < total_time; i++) {

        allstop = 0;

        for (j = 1; j < arrival_time; j++) {

                if (arr[j][1] < 1)
                        allstop++;

                if (pick == -1) {
                        pick = rand() % scpr[size-1];
                }
                if (pick >= scpr[j-1] && pick < scpr[j]) {
                        if (i >= arr[j][0] && arr[j][1] >= 1)
                        {
                                process = j;
                                break;
                        }
                        if (i < arr[j][0] || arr[j][1] < 1) {
                                if (process == 0) {
                                        pick = -1;
                                        i--;
                                        break;
                                }
                        }
                }
        }

        if (process > 0) {
                arr[process][1]--;
                throughput[process][i] = 1;
                process = 0;
                pick = -1;
        }
```

```
            if (allstop == arrival_time-1) {
                deadline = ++i;
                break;
            }
        }

            printf("            ");
        for (l = 0; l < deadline; l++)  {
                printf("%2d   ",l);
        }
        printf("₩n      ");

        for (l = 0; l < deadline; l++)  {
                printf("------");
            }
        printf("₩n");

        for (i = 1; i < arrival_time; i++) {
            printf(" [%c] >> ",i+64);
            for (k = 0; k < deadline; k++)  {
                if (throughput[i][k] == 1) {
                    printf("■   ");
                }
                else
                    printf("□   ");
            }
            printf("₩n");
        }
}
```