Lab1
- 긴 보고서

과목: 운영체제(3분반)
담당 교수님: 최종무 교수님

이름: 양혜은, 정윤아
학번: 32182609,32184160
소속: SW융합대학 소프트웨어학과
학년: 3학년

2020년 4월 29일 제출

목차

1. 자료구조 2

1) Process 자료 구조 2

2) Queue 자료 구조 2

2. 알고리즘 구현을 쉽게 하기 위한 함수 3

3. 스케줄러 알고리즘 5

1) FCFS 5

2) Round-Robin 6

3) MLFQ 7

4) Stride Scheduling 10

4. 시뮬레이션 결과 12

1) Lecture Note3 – 24page 워크로드에 대한 시뮬레이션 12

2) 랜덤으로 생성한 워크로드에 대한 시뮬레이션 18

3) Stride 스케줄링 시뮬레이션 28

5. Discussion 31

1. 자료 구조

Lab1에서는 크게 2가지 자료구조를 이용했다.

1) Process 자료 구조: struct Process, struct WaitingInfo를 사용

(1) struct Process

struct Process는 프로세스의 수행 정보를 담는 PCB(Process Control Block)를 흉내 낸 구조체이다. 별칭은 process이다. 멤버변수는 8개가 있다. 프로세스마다 고유한 값을 가지는 id와 프로세스의 도착 시간을 의미하는 t_arrival, 총 서비스되는 시간을 의미하는 t_service, 프로세스가 실행되어야 할 남은 시간을 의미하는 t_remain, 프로세스가 CPU를 얼마나 할당 받았는지를 나타내는 t_keepQ, 프로세스가 처음 실행된 시간을 의미하는 t_firstrun, 프로세스가 끝난 시간을 의미하는 t_completion이 int형으로 선언됐다. 또한 MLFQ에서 I/O프로세스를 정의하기 위한 waitingInfo 구조체를 ioInfo라는 이름으로 사용한다.

(2) struct WaitingInfo

struct WaitingInfo는 오직 mlfq에서만 사용하기 위해 만든 구조체이다. 별칭은 waitingInfo이다. batch job인지, interactive job인지, 높은 우선 순위를 유지하기 위해 타임 슬라이스를 채우기 전에 고의적으로 CPU를 양도하는 fake job인지 분류하기 위해 pkind라는 int형 멤버 변수를 만들었다. pkind가 0이면 batch job, 1이면 interactive job, 2면 fake job임을 의미한다. 또한 wait된 시간부터 wake up 되기까지 남은 시간을 나타내는 t_inWait을 멤버 변수로 갖는다.

2) Queue 자료구조: struct ProcessNode, struct Queue를 사용

(1) struct ProcessNode

struct ProcessNode는 프로세스를 연결리스트로 만들기 위해 정의한 구조체이다. 별칭은 processNode이다. 프로세스 자료형을 가리키는 process* pAddress와 연결된 다음 노드를 가리키는 processNode* next를 멤버변수로 갖는다.

(2) struct Queue

struct Queue는 큐를 만들고 사용하기 위한 구조체이다. 별칭은 queue이다. 큐로 구현하기 위해 연결된 리스트의 가장 첫번째 노드를 가리키는 processNode* head, 가장 마지막 노드를 가리키는 processNode* tail, 각각의 큐에서 정의하여 사용하는 quantum, 그리고 연결된 processNode 개수를 저장하는 qsize를 int형 멤버변수로 갖는다.

2. 알고리즘 구현을 쉽게 하기 위해 사용한 함수

다음 함수들은 여러 가지 스케줄링 함수들에서 공통적으로 이용되는 기능을 구현한 것이다. 스케줄링 알고리즘을 쉽게 코딩하고 쉽게 코드를 이해할 수 있게 하는 기능을 선별하여 작성했다.

(1) process* makeProcesses(int size)

이 함수는 워크로드를 생성하기 위해 작성했다. 매개변수 size 크기의 프로세스 배열을 만들고 초기화하여 반환한다.

만들고 싶은 process의 개수를 매개변수 size로 입력받는다. 입력받은 크기만큼 메모리를 동적 할당하여 프로세스 배열을 생성한다. 생성한 각각의 프로세스에 대해 rand함수를 이용하여 도착 시간과 총 서비스되어야 하는 시간을 자동으로 초기화하였다. t_ arrival은 0부터 7까지, t_service는 1부터 15까지의 난수를 저장한다. t_remain은 t_service로 초기화하여 스케줄링 시뮬레이션을 하면서 process가 앞으로 실행되어야 할 시간을 업데이트할 수 있도록 만들었다.

이외에 프로세스가 실행되면서 결정되는 값은 비정상적인 값인 -1로 초기화했다.

(2) process* copyProcessesArr(process* arr, int size)

이 함수는 워크로드를 복사해주는 함수로, 동일한 워크로드에 대한 스케줄링 결과를 비교하기 위해 필요하다. 워크로드를 복사하지 않고 하나만 사용한다면, 모든 스케줄러가 워크로드를 업데이트할 것이다. 결국 스케줄러는 서로 다른 워크로드에 대해 스케줄링 하게 된다. 이것은 배열이 포인터로 전달되기 때문에 발생한다.

(3) void t_arrivalSort(process* processArr, int size)

FCFS, RoundRobin, MLFQ는 우선 도착한 프로세스의 스케줄링을 나중에 도착한 프로세스보다 먼저 한다. 따라서 스케줄러를 구현할 때 프로세스를 도착시간에 대해 정렬하는 과정이 공통적으로 필요하여 함수로 따로 작성했다.

이 정렬함수는 process 배열의 시작 주소와 배열 사이즈를 전달받는다. 배열 주소를 참조하여 직접 정렬한다. 정렬법으로는 선택 정렬을 사용했다. 이 프로그램에서는 비교적 적은 수의 데이터를 정렬하기 때문에 복잡한 heap이나 merge sort같은 정렬법보다 선택 정렬이 더 유리하다고 판단했다.

(4) void print_run(int size, int pid)

print_run은 스케줄링 결과를 보여 주기 위한 함수이다. 실제로 프로세스는 스케줄링 되면 run 상태가 되어 실행되는데, 시뮬레이터에서는 이 함수를 이용하여 스케줄링 된 프로세스를 특수 문자 #으로 표시하여 출력한다.

프로세스 개수인 size, 현재 실행해야 하는 process의 id인 pid를 매개변수로 받는다. 전역 변수 시스템 시간 상에서 pid가 스케줄링 됐다고 해당 프로세스의 위치에 특수 문자 #을 출력한다. 만약 넘겨받은 pid가 0인 경우는 시스템 시간 상에서 스케줄링 할 프로세스가 없는 것으로, 해당 시간을 비워 놓도록 출력한다.

(5) void push_nodeToQueue(process* processAddress, queue* q)

이 함수는 프로세스의 주소와 큐를 매개변수로 받아서 해당 프로세스에 대한 노드를 큐에 추가한다. 프로세스 노드를 바로 받아 큐에 넣는 함수가 데이터 구조상 자연스럽다. 하지만 처음에는 프로세스들이 큐에 있지 않고 프로세스 배열에 존재한다. 따라서 해당 프로세스에 대한 정보를 프로세스 노드 자료형으로 넘겨 받을 수 없기 때문에 프로세스의 주소를 매개변수로 받았다.

이 함수 내부에서는 processNode*형 지역변수 newNode에 processNode에 대한 동적 메모리를 할당하고, newNode의 pAddress에 매개변수로 받은 프로세스 주소를 넣어준다. 그리고 그것을 큐의 마지막 노드에 연결한다.

(6) bool check_push(process* processArray, int size, queue* nowQue, int* last)

새로 도착한 프로세스가 있는지 체크하고, 있다면 큐에 추가까지 하는 함수이다. 매개변수 processArray와 size는 실행 해야 할 모든 프로세스를 담고 있는 배열과 그 크기이고, nowQueue는 프로세스를 넣어야 할 큐이다. last는 processArray에서 이미 도착해 큐에 들어간 프로세스의 인덱스를 제외하고 가장 작은 인덱스를 저장한다. 도착한 프로세스가 있었다면 true를, 없었다면 false를 반환한다.

이 함수는 push_nodeToQueue 함수와 비슷하게 실행할 프로세스에 대한 정보를 큐에 넣는 역할을 한다. push_nodeToQueue보다 제한적인 역할을 하는 함수를 따로 만든 이유는 스케줄러를 구현하면서 자주 생길 수 있는 실수를 방지하고 이해하기 쉽게 코드를 작성하기 위함이다. 또한 RoundRobin과 MLFQ에서 도착한 프로세스가 있는지 여부에 따라 달라지는 분기문을 작성하기 훨씬 쉬워진다.

(7) void print_performance(process* processArray, int size)

이 함수는 수행이 끝난 후 업데이트된 프로세스 정보를 담고 있는 process 배열을 이용해 성능을 계산하여 출력한다. 프로세스가 도착한 시간과 완료된 시간의 차를 이용하여 평균 반환 시간(turnaround time)을 구하고, 프로세스가 도착한 시간과 처음 실행된 시간의 차를 이용하여 평균 응답 시간(response time)을 구한다.

(8) void print_attributes(process* processes, int size)

스케줄링 알고리즘이 올바르게 구현됐는지 확인하기 위해서는 랜덤으로 속성이 정해진 프로세스에 대해 알아야 한다. 이 함수는 프로세스 배열을 받아 프로세스 id, 도착 시간, 서비스 시간을 출력하고 해당 프로세스의 종류를 알려준다.

이 함수를 이용해 출력된 워크로드에 대해 수기로 스케줄링 하고, 프로그램 결과값을 비교하여 오류가 없는지 확인했다.

(9) queue* makeQueue()

이 함수는 큐를 동적으로 생성하고 멤버를 바르게 초기화하여 큐 주소를 반환한다.

(10) process* pop_address_ofProcess(queue* q)

이 함수는 매개변수로 받은 큐에서 프로세스를 꺼내는 연산을 한다. 큐에 연결된 첫번째 프로세스 노드를 꺼내어 연결을 끊어준다. 이 과정에서 큐는 꺼낸 프로세스 노드의 다음 노드로 연결해준다.

그리고 꺼낸 노드의 멤버인 process 포인터를 반환한다.

만약 큐에서 processNode를 꺼낸다면 프로세스 정보를 이용하기 위해서는 포인터 연산이 추가로 필요하다. 이러한 연산을 줄이고 process를 직관적으로 이용하기 위해 processNode가 아닌 process를 가리키는 포인터를 반환값으로 선택했다. 만약 큐가 비어있다면 NULL을 반환한다.

3. 스케줄러 알고리즘

Lab1에서는 FCFS, Round-Robin, MLFQ, Stride Scheduler의 알고리즘을 구현했다.

1) FCFS 알고리즘

FCFS(First-Come-First-Served = FIFO)는 먼저 도착한 프로세스부터 스케줄링 하는 방식이다. FCFS 스케줄링 알고리즘을 구현하기 위해 fcfs 함수를 만들었다.

(1) void fcfs(int process* processArr, int size)

fcfs 함수는 워크로드에 해당하는 프로세스 배열 processArr과 그 배열의 크기에 해당하는 size를 매개 변수로 받는다. FCFS는 도착 순서대로 프로세스들을 스케줄링 하는 특징을 가졌다. 따라서 fcfs에서 매개변수로 넘겨 받은 프로세스 배열은 t_arrivalSort 함수를 사용해 도착 시간이 빠른 순서대로 정렬된 것이다.

지역 변수 last는 프로세스 배열에서 스케줄링 해 줄 차례인 프로세스의 인덱스를 저장하는 변수이다. 프로세스들이 도착 시간 순으로 정렬돼있는 상태이기 때문에 초기값은 0으로 설정한다. FIFO에서는 가장 늦게 들어온 프로세스가 마지막으로 스케줄링 되므로 그 프로세스가 종료되면 모든 프로세스에 대한 실행이 이미 끝났음을 의미한다. 따라서 last가 프로세스 배열의 마지막 인덱스 값을 초과하기 전인 last <size이면 다음의 과정을 반복한다.

만약 s_time < processArray[last].t_arrival 즉, 현재 시스템 타임 시점에서 도착한 프로세스가 없는 경우 print_run(size, 0)을 실행하고 시스템 타임을 1 증가시킨다. 제일 먼저 스케줄링 돼야 할 프로세스가 이미 도착하여 s_time >= processArray[last].t_arrival인 경우에는 그 프로세스를 스케줄링 해준다. 이때의 시스템 타임을 그 프로세스가 처음 실행되는 시간을 나타내는 t_firstrun에 저장한다. 그 리고 프로세스의 서비스 타임만큼 print_run 함수를 호출하고, 남은 서비스 타임에 해당하는 remain을 감소시켜준다. 그리고 이때마다 시스템 타임을 함께 증가시켜 시스템 타임의 시간을 맞춰준다. 수행 중이던 프로세스의 실행을 마쳤을 때, 현재 시스템 타임을 프로세스의 종료 시간으로 넣어주고, last를 증가시켜 다음 프로세스를 검사할 수 있도록 한다.

모든 프로세스가 실행을 마쳐 while문을 빠져 나왔을 때 print_performance 함수를 호출해 평균 반환 시간과 평균 응답 시간을 구하고, 그 결과를 출력한다.

2) Round-Robin 알고리즘

RR(Round-Robin)은 기본적으로 FCFS와 같이 먼저 도착한 프로세스를 스케줄링 한다. 하지만 그 프로세스가 끝날 때까지 계속 실행시키지 않고 정해진 시간만큼만 실행한 후, 다른 작업으로 넘어가는 방식이다. 이때 한 작업을 실행시키는 정해진 시간을 타임 슬라이스 혹은 타임 퀀텀이라 부른다.

Round-Robin 스케줄링 알고리즘을 구현하기 위해 RoundRobin 함수를 만들었다.

(1) void RoundRobin(process* processArr, int size, int t_quantum)

RoundRobin 함수는 실행시킬 프로세스들을 담고 있는 프로세스 배열 processArr과 프로세스들의 수에 해당하는 size, 그리고 타임 슬라이스에 해당하는 t_quantum을 매개변수로 받는다.

FCFS와 마찬가지로 RR의 기본 알고리즘도 도착한 순서대로 프로세스들을 스케줄링 하는 것이므로 매개변수로 받은 process 배열은 t_arrivalSort 함수를 이용해 도착한 순서대로 정렬된 것이다. 지역변수 last는 도착했는지 검사할 프로세스의 인덱스를 담는다. 먼저 도착한 프로세스부터 검사를 해줄 텐데, 현재 프로세스 배열이 도착 순으로 정렬되어 있으므로 초기값은 0으로 설정한다. makeQueue 함수를 이용하여 스케줄링 대상이 되는 프로세스들을 담을 큐를 생성한다. 그리고 RoundRobin은 모든 프로세스들이 실행을 마치기 전까지 다음의 과정을 반복하며 시뮬레이션 한다.

우선 check_push 함수를 이용하여 현재 시스템 타임에서 도착한 프로세스가 있는지 검사하고, 도착한 프로세스가 있는 경우 큐에 넣는다. 만약 큐가 비어있는 경우에는 스케줄링 할 job이 없는 것이므로 print_run(size, 0)을 호출하여 그 시간을 빈 채로 출력하고, 시스템 타임을 1 증가시킨다. 그리고 continue를 수행하여 다시 while문의 첫 부분으로 돌아가 증가된 시스템 타임 시점에서 도착한 프로세스가 있는지부터 확인한다.

반대로 실행 큐가 비어 있지 않은 경우에는 pop_address_ofProcess 함수를 호출하여 실행 큐로부터 스케줄링 할 프로세스의 주소에 해당하는 프로세스 포인터를 넘겨받고, 실행 큐에서 없애준다. 이때 프로세스 포인터가 가리키는 프로세스의 firstrun이 처음에 프로세스 생성했을 때의 초기화 값인 -1이면, 처음 스케줄링 되는 것을 의미한다. 이 경우 프로세스가 처음 실행된 시간을 의미하는 firstrun을 현재 시스템 시간으로 설정해준다. 그리고 print_run 함수를 타임 슬라이스만큼 반복해서 호출하여 스케줄링 결과를 출력한다. 그때마다 프로세스의 남은 서비스 타임에 해당하는 t_remain값은 1씩 감소시키고, 시스템 타임은 1씩 증가시킨다. 만약 반복문을 수행하던 중 t_remain이 0이 되면, 그 프로세스의 타임 슬라이스가 지나기 전에 그 프로세스가 종료된 것이므로 반복문을 탈출한다. 그 다음, 새로 들어온 프로세스들이 있는지를 확인한다. 만약 도착한 프로세스가 있다면 큐에 넣어준다. 그리고 방금 수행했던 프로세스의 t_remain이 남아있는지를 확인하여 큐에 넣어준다.

지금까지의 과정을 반복하다가 모든 프로세스가 실행을 마쳤으면 while문을 벗어나면 된다. 모든 프로세스가 종료했다는 조건은 마지막 프로세스까지 큐에 도착을 이미 한 상태여서 더 이상 새롭게 들어올 프로세스가 없고, 큐가 비어있는 경우이다. 즉 last==size && rrqueue->qsize == 0으로 표현할 수 있다. 그리고 종료 조건을 만족하여 while문을 빠져나오면 평균 반환 시간과 평균 응답 시간을 출력한다.

3) MLFQ 알고리즘

MLFQ는 응답 시간을 최소화하면서 반환 시간을 최적화하기 위해 고안된 스케줄링 방식이다. MLFQ는 여러 개의 큐로 구성되고, 각각의 큐들은 서로 다른 우선 순위를 가진다. MLFQ는 기본적으로 (규칙1)우선순위가 더 높은 큐에 있는 job을 먼저 스케줄링하며, (규칙2)같은 큐에 존재하는 job들의 경우에는 Round-Robin 방식으로 스케줄링한다.

각각의 프로세스가 들어갈 큐는 다음과 같은 규칙들을 통해 결정된다.

(규칙3) 새로운 프로세스가 시스템에 들어오면 가장 높은 우선순위의 큐에 들어간다.

(규칙4) 현재 속한 레벨의 큐에서 시간 할당량을 모두 다 쓰면 우선순위가 내려간다.

(규칙5) 일정 시간이 지나면 모든 job들을 가장 높은 우선순위의 큐로 올린다.

이러한 규칙들을 만족하는 MLFQ 알고리즘을 MLFQ 함수로 구현했다. 그리고 이 함수를 구현하기 위해 몇 가지 다른 함수들도 만들었다.

(1) void print_allElement(queue* q)

print_allElement는 MLFQ 시뮬레이션 시 스케줄링이 어떻게 이루어지고 있는지 쉽게 확인할 수 있도록 큐의 상태를 출력할 때 사용되는 함수이다. 매개변수로 큐를 받으면, 그 큐에 속한 모든 노드들의 프로세스 id를 출력해준다.

(2) void print_Queues(queue** multilevel, int level, queue* wait)

print_Queues는 print_allElement와 함께 MLFQ 시뮬레이션 시 큐의 상태를 출력하기 위해 사용되는 함수이다. print_Queue는 큐들을 배열로 만든 멀티 레벨 큐, 큐의 level 수, wait 큐를 매개변수로 받고, 멀티 레벨 큐에 속한 모든 큐들과 wait 큐에 대해 print_allElement 함수를 호출한다. 그 결과 wait 큐와 멀티 레벨 큐의 상태가 보기 좋게 출력된다.

(3) void doBoost(queue* multiQ[], int level)

doBoost는 MLFQ 스케줄링에서 규칙5에 해당하는 부스팅을 수행하는 함수이다. 멀티 레벨 큐와 큐의 레벨 수를 매개 변수로 받는다. 모든 레벨의 큐를 탐색하면서 만약 올릴 job이 있다면 가장 높은 우선순위를 가진 multiQ[0]으로 옮겨준다. 이 함수가 호출되면 두 번째로 높은 우선순위 큐부터 가장 낮은 우선순위 큐까지 차례로 방문하여 프로세스를 가장 높은 우선순위 큐로 올린다.

(4) void pushjob_again(process* p, queue* q, int waitTime)

pushjob_again은 이미 스케줄링 되어 일정 시간 동안 실행된 후에도 아직 수행이 종료되지 않은 job들을 스케줄링 하기 위해 다시 큐에 넣어주는 역할을 하는 함수이다. 직전에 실행한 프로세스와, 그 프로세스를 넣을 큐, 그리고 I/O job이 wait큐에 들어갔을 때 wake up되기까지의 시간인 waitTime을 매개 변수로 받는다. 만약 전달 받은 프로세스가 batch job이거나 fake job이면 함수를 호출할 때 같이 넘겨줬던 큐에 그 프로세스를 추가한다. 그리고 프로세스가 I/O job인 경우, 해당 프로세스를 인자로 넘겨 받은 wait 큐에 넣어줄 뿐 아니라, 프로세스의 멤버 변수인 ioInfo.t_inWait를 waitTime으로 설정

해준다. inWait을 설정해줌으로써 wait 큐에 올라간 프로세스는 몇 초 동안 wait 상태였는지를 기억할 수 있고, waitTime 만큼의 시간이 지나면 ready 큐로 내려 올 수 있게 한다.

(5) void pushWakejob(queue* waitQueue, queue* q0)

pushWakejob은 wait 큐와 가장 높은 우선순위 큐를 매개 변수로 받는다. 이 함수는 wait 큐를 탐색하면서 만약 멤버 변수 ioInfo.t_inWait이 0 이하가 된 프로세스가 있다면, 즉 wake up된 job이 있다면 wait 큐로부터 꺼내어 가장 높은 우선순위 큐에 넣어주는 함수이다. 실제 시스템에서는 wait 상태로 머무는 시간이 정해지지 않은 경우도 있고, 각 job들마다 wait 상태로 머무는 시간이 다르므로 큐의 중간에 위치한 프로세스를 꺼내야 할 수 있다. 하지만 우리는 모든 I/O job들에 대해 wait 상태의 시간을 고정시켰기 때문에 FIFO 방식으로 job들이 wake up된다. 따라서 pop_address_ofProcess 함수를 이용해서 큐에서 프로세스를 꺼내도록 했다.

(6) void decreaseTime_waitQ(queue* q)

decreaseTime_waitQ는 시스템 타임을 증가시킬 때, wait 큐에 있는 프로세스들이 wake up하기까지 얼마나 남았는지를 나타내는 멤버변수 ioInfo.t_inWait도 함께 업데이트를 하기 위해 사용하는 함수이다. wait 큐를 매개 변수로 받고, 시스템 타임을 증가시킬 때 이 함수도 함께 호출하여 흐른 시간이 wait 큐에도 반영될 수 있도록 한다.

(7) bool isEmptyQueue(queue* multiQ[], int level)

isEmptyQueue는 멀티 레벨 큐와 큐의 레벨 수를 매개변수로 받아 각 레벨을 탐색하면서 만약 멀티 레벨 큐가 비어있다면 true를 반환하고, 비어있지 않다면 false를 반환한다. 이 함수는 MLFQ에서 실행했던 프로세스를 다시 큐에 넣어줄 때의 조건을 위해 활용된다. 강의 노트 3장의 24페이지 워크로드로 예를 들자면, 0ms에 도착하여 1ms동안 수행된 A프로세스는 1ms 시점에 큐가 비어있고, 새로 도착한 다른 프로세스도 없기 때문에 한 번 더 스케줄링 되었다가 다음 큐로 들어가게 된다. 이 부분을 구현하는 과정에서 큐가 비었는지 확인하기 위해 isEmptyQueue 함수를 이용한다.

(8) void MLFQ(process* processArray, int size, int level, int quantumSame)

MLFQ 함수는 프로세스 배열과 프로세스 배열의 크기, 그리고 큐 레벨의 수, 큐 레벨 별로 타임 퀀텀이 같은지를 나타내는 quantumSame을 매개변수로 받는다. quantumSame은 1, 2, 3 중 하나의 값이 들어온다. 인자의 값이 1인 경우 모든 레벨의 큐의 타임 슬라이스를 1로 둔다. 2인 경우에는 큐의 레벨을 i로 표현 할 때, 타임 슬라이스를 2^i로 둔다. 마지막으로 3인 경우, 타임 슬라이스를 2^{i+1}로 설정한다.

MLFQ 함수를 호출하면 먼저 매개변수로 넘겨 받은 level 수만큼의 큐를 원소로 갖는 배열을 만든다. 이것이 MLFQ에서의 멀티 레벨 큐에 해당한다. 그리고 매개 변수로 넘겨 받은 quantumSame의 값에 따라 큐 별로 타임 슬라이스를 설정하고 wait 큐를 생성한다. 이 외에도 다른 변수들을 초기화하는데, I/O job이 wait 큐로 들어가서 wake up 될 때까지의 시간을 나타내는 waitTime을 여기서는 3으로 고정시켰다. 그리고 t_IOprunning은 I/O job이 스케줄링 됐을 때 wait 상태가 되어 wait 큐로 들어가기 전에 CPU를 점유하는 시간을 의미한다. t_IOprunning은 waitTime과 마찬가지로 실제 시스템에서는 다양하게 나타나겠지만, 우리는 1로 고정시켰다. 즉 모든 I/O job들이 스케줄

링 되면 타임 퀀텀보다 작거나 같은 1초 동안만 실행되다가 wait 큐로 들어가서 3초를 기다린 후에 다시 멀티 레벨 큐로 들어오게 설정했다. t_usingCPU는 fake job이 스케줄링 된 후 CPU를 점유하는 시간을 나타낸다. fake job이 높은 우선순위를 유지하기 위해, 타임 슬라이스를 채우기 전에 고의적으로 CPU를 반납하는 상황을 가정하여 1로 설정했다. 그리고 부스트 주기는 사용자로부터 입력을 받아 설정한다.

MLFQ 함수는 크게 두 영역으로 구성된다. 첫 번째 영역은 새로 도착한 프로세스와 실행하다가 남은 프로세스를 다시 큐에 넣어주는 부분이다. 두 번째는 멀티 레벨 큐에서 적절한 프로세스를 꺼내어 실행하는 부분이다. 이 과정을 계속 반복하다가 모든 프로세스가 수행을 마치고, wait 큐도 비어있을 때 MLFQ 함수가 종료된다.

새로 도착한 프로세스를 큐에 넣어준 후, 직전에 실행한 프로세스가 더 수행돼야 한다면 큐에 다시 넣어줘야 한다. 이때 I/O job인지, fake job인지, batch job인지에 따라 경우를 나누어 큐에 넣어준다. 만약 I/O job인 경우 wait 상태가 되어 실행을 중단한 것이므로 pushjob_again 함수를 이용하여 해당 프로세스를 wait 큐에 넣는다. 그리고 fake job인 경우, 타임 슬라이스를 다 채우지 못한 채 CPU를 반납한 것이므로 pushjob_again 함수를 호출하여 같은 레벨의 큐에 프로세스를 다시 넣어준다. 그런데 이때 만약에 해당 프로세스가 현재 큐에서 CPU를 할당 받은 시간을 누적한 t_keepQ가 그 큐의 타임 슬라이스와 같거나 크고, 현재 큐가 가장 낮은 우선순위 큐가 아니면, 우선 순위를 낮춰 아래의 큐에 집어 넣고, 다시 t_keepQ는 0으로 초기화한다. batch job인 경우에는 타임 슬라이스를 다 채워서 실행하므로 기본적으로 다시 큐에 넣을 때는 아래 단계의 큐에 집어넣는다. 하지만 만약 마지막 큐에 위치해있거나, 멀티 레벨 큐에 실행할 다른 프로세스가 없고, 새로 도착한 프로세스가 없는 경우에는 같은 레벨을 유지해준다.

프로세스를 스케줄링할 때는 멀티 레벨 큐의 높은 우선 순위 큐부터 탐색하면서 프로세스를 꺼내고 todoProcess라는 변수에 담아 놓는다. 이때 꺼낼 프로세스가 없어서 todoProcess가 NULL이라면 print_run(size, 0)을 호출하여 해당 시스템 타임에 아무것도 수행하지 않음을 출력하고 시스템 타임을 증가시킨다. 그리고 decreaseTime_waitQ 함수를 함께 호출하여 waitQ에 있는 프로세들의 대기 시간도 업데이트 한다. 다음으로 부스트 시간이 됐는지, wake된 job이 있는지, 새로 들어온 프로세스들이 있는지를 검사하고 적절한 프로세스들을 큐에 넣어준다. 이는 시스템 타임이 바깥쪽 while 문을 한 번 돌 때마다 1씩 증가되는 것이 아니라 print_run함수를 호출할 때마다 시간이 증가되는 것이므로 매 시간마다 검사를 하기 위한 것이다. todoProcess가 NULL이 아니라면 I/O job인지, fake job인지, batch job인지에 따라 경우를 나누어 실행한다.

만약 batch job이라면 해당 큐의 퀀텀만큼 print_run 함수를 호출하고 decreaseTime_waitQ 함수도 함께 호출한다. t_remain은 퀀텀만큼 감소하고, 시스템 타임은 퀀텀만큼 증가한다. 그리고 todoProcess가 NULL일 때와 마찬가지로, 부스트할 시간이 됐다면 doBoost 함수를 호출하여 부스트시키고, wake된 job이 있는지, 새로 들어온 job이 있는지 pushWakejob, check_push 함수를 이용해 검사하고 큐에 넣어준다. 만약 타임 퀀텀만큼 이 과정을 반복하던 중 t_remain이 0과 같거나 작아진다면 반복문을 종료시킨다. todoProcess가 i/o job이나 fake job인 경우도 이 과정과 동일하지만, 타임 퀀텀만큼 이 과정을 반복하는 것이 아니라, 각각 IOprunning, t_usingCPU의 값만큼 이 과정을 반복한다. 이렇게 프로세스를 실행하는 부분을 마치면 다시 바깥쪽 while문의 첫 부분으로 돌아가 새로운 프로세스가 있는지 검사하고, 남은 프로세스들을 다시 큐에 넣게 된다.

4) Stride Scheduling 알고리즘

Stride 스케줄링은 앞에서 살펴본 다른 스케줄링 기법들처럼 반환 시간이나 응답 시간의 최적화에 초점을 맞춘 스케줄링 방식이 아니다. Stride는 각 job들이 CPU 사용 시간에 대한 일정한 비율을 갖도록, 공정 지분을 보장하기 위한 스케줄링 방식이다. Stride 스케줄링의 알고리즘은 다음과 같다. 먼저 티켓의 수에 따라 stride값을 구하고, passvalue가 가장 작은 job을 수행한다. 그리고 수행한 job의 passvalue에 stride값을 더한다. 다시 passvalue가 가장 job을 찾아 수행하고 이 과정을 반복한다.

Stride 스케줄링 알고리즘을 구현하기 위해 Stride 함수를 만들었다. 그리고 Stride 스케줄링 방식에서는 반환 시간, 응답 시간보다 각각의 job들이 얼마나 원하는 비율로 스케줄링 됐는지가 의미 있기 때문에 Stride 스케줄러에 대한 프로세스 자료 구조를 따로 만들었다. 이외에도 Stride 알고리즘을 구현하기 위해 만든 몇 가지 함수가 있다.

(1) struct stride_Process

struct stride_Process는 Stride 스케줄러를 시뮬레이션 할 때 사용할 프로세스 자료구조로, 별칭은 stride_process이다. stride_process는 앞에서 다른 스케줄러를 위한 프로세스 자료구조로 사용하는 process와 다르게 멤버변수를 5개 가진다. 프로세스마다 고유한 값을 가지는 id, 스케줄링 순서를 정해주기 위한 passvalue, CPU 사용 지분에 해당하는 tickets, 티켓 값의 역수에 해당하는 stride, 스케줄링 된 횟수를 의미하는 sched_time가 모두 int형으로 선언됐다.

(2) stride_process* makeStride_Processes(int size)

이 함수는 Stride 스케줄러에 대한 워크로드를 랜덤으로 생성하기 위해 작성했다. 인자로 넘겨 받은 size 크기의 프로세스 배열을 만들고 초기화하여 그 배열의 주소를 반환한다.

만들고 싶은 stride_process의 개수를 매개변수 size로 입력받는다. 입력받은 크기만큼 메모리를 동적 할당하여 프로세스 배열을 생성한다. 모든 프로세스들에 대하여 passvalue를 0으로 초기화하고, 각 프로세스마다 tickets는 5에서 50사이의 임의의 5의 배수를 저장했다. stride는 stride값을 Stride 함수 안에서 구할 것이므로 의미가 없는 -1로 초기화시켰다. sched_time은 0으로 초기화시켰고, 해당 프로세스가 스케줄링 될 때마다 업데이트 될 것이다.

(3) void print_stride(int scheduler time, int size, stride_process* processArr)

print_stride 함수는 stride 스케줄링을 시뮬레이션 할 때 스케줄링 되는 과정을 보여주기 위한 함수로 시스템 타임에 해당하는 scheduler time, 프로세스 배열의 크기에 해당하는 size, 프로세스 배열인 processArr을 매개변수로 받는다. 넘겨 받은 시스템 타임에서 각 프로세스들이 어떤 passvalue값들을 갖고 있는지 출력한다. 따라서 print_stride 함수의 출력 결과를 통해 어떻게 스케줄링이 이뤄지고 있는지 이해할 수 있다.

(4) void run_stride(stride_process task)

run_stride는 스케줄링 되는 프로세스를 매개변수로 넘겨받고, 그 프로세스의 id를 출력해준다.

(5) int getGCD(int a, int b)

getGCD는 두 수의 최대공약수를 구하는 함수로, int형 정수 2개를 매개변수로 넘겨 받는다. Stride 스케줄링 방식에서 stride값을 구하기 위해서는 각 프로세스들의 티켓들의 최소공배수가 필요하다. 어떤 두 수의 최소공배수는 두 수의 곱을 두 수의 최대공약수로 나눠 구할 수 있다. 따라서 최소공배수를 쉽게 구하기 위해 최대공약수를 구하는 getGCD 함수를 만들었다. 최대공약수를 구하는 알고리즘으로는 유클리드 호제법을 이용했다. getGCD는 두 수의 최대공약수를 return 한다.

(6) int getLCM(stride_process* processArr, int size)

getLCM은 stride값 계산을 위해 티켓들의 최소공배수를 구하는 함수이다. stride값은 모든 프로세스 티켓 값들의 공배수를 각 프로세스의 티켓 값으로 나누어 구한다. 공배수를 구할 때 단순히 프로세스 티켓 값들을 곱하게 되면 stride값이 너무 커지기 때문에, 최소공배수로 구해줬다.

프로세스 배열에 해당하는 processArr과 그 배열의 크기인 size를 매개변수로 넘겨 받는다. 반복문을 돌며 프로세스 배열에 담긴 프로세스 티켓 값들의 최소공배수 연산을 2개씩 차례대로 수행한다. 최종적으로 프로세스 배열에 있는 모든 프로세스 티켓 값들에 대한 최소공배수를 구한다. 그리고 최소공배수 값을 return한다.

(7) void Stride(stride_process* processArr, int size, int runtime, int t_quantum)

Stride 함수는 워크로드에 해당하는 프로세스의 배열과 그 배열의 크기 size, 사용자로부터 입력 받은 시뮬레이션 시간인 runtime, 타임 슬라이스에 해당하는 t_quantum을 매개변수로 받는다. 본격적으로 스케줄링을 하기에 앞서 매개변수로 넘겨 받은 프로세스 배열에 있는 각 프로세스 원소들의 stride값을 구해준다. stride 값을 구하기 위해 getLCM 함수를 사용하고, return 받은 최소공배수 값을 지역 변수 commonMultiple에 저장한다. 그리고 for문을 돌며 commonMultiple을 각 프로세스의 ticket값으로 나눠 stride 값을 구하고 저장한다.

Stride 방식은 가장 작은 passvalue를 가진 프로세스를 스케줄링 해준다. 따라서 프로세스 배열에서 가장 작은 passvalue를 갖는 프로세스의 인덱스를 저장하기 위한 지역변수 index_minPassvalue를 선언해줬다. 프로세스 배열들을 탐색하면서 가장 작은 passvalue를 가진 프로세스의 인덱스를 index_minPassvalue에 저장한다. 아예 동일한 passvalue를 갖는 프로세스들이 있다면 id값이 더 작은 프로세스를 우선으로 저장했다. 이때 index_minPassvalue 값을 인덱스로 갖는 프로세스가 스케줄링 되어야 하므로 타임 슬라이스만큼 print_stride와 run_stride 함수를 호출하여 스케줄링 결과를 출력하고, 그만큼 시스템 타임을 증가시킨다. 그리고 실행한 프로세스의 passvalue에는 stride값을 더해주고, 스케줄링 횟수에 해당하는 sched_time을 증가시켜준다. 이러한 스케줄링 과정은 시스템 타임이 메인 함수에서 입력 받은 시뮬레이션 시간과 같아지기 전까지 반복된다.

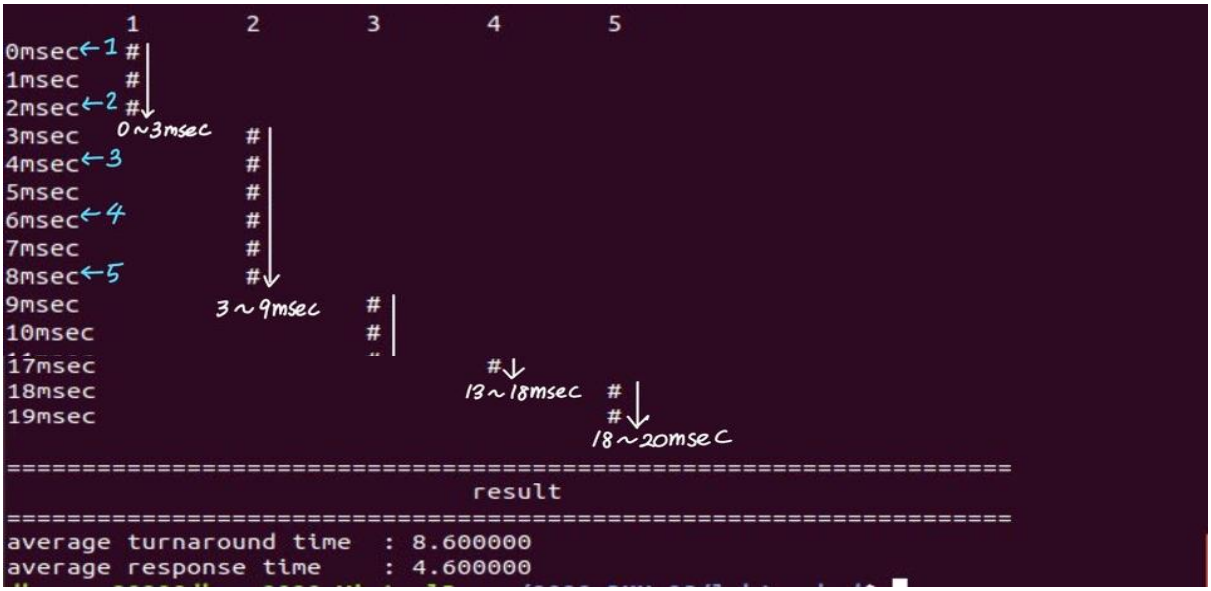
4. 시뮬레이션 결과

Lab1에서는 FCFS, Round-Robin, MLFQ, Stride Scheduler의 알고리즘을 구현한 후 Lecture Note3의 24페이지에 나와있는 workload와 makeProcesses 함수를 이용하여 랜덤으로 생성한 Workload에 대한 스케줄링 시뮬레이션을 했다.

1) Lecture Note3 – 24page 워크로드에 대한 시뮬레이션

```
**********Attributes of processes(FCFS, RR, MLFQ)*******
* id                    1
* Arrive time           0
* Service time          3
* Bach job
**************************************************************
* id                    2
* Arrive time           2
* Service time          6
* Bach job
**************************************************************
* id                    3
* Arrive time           4
* Service time          4
* Bach job
**************************************************************
* id                    4
* Arrive time           6
* Service time          5
* Bach job
**************************************************************
* id                    5
* Arrive time           8
* Service time          2
* Bach job
**************************************************************
```

(1) FCFS

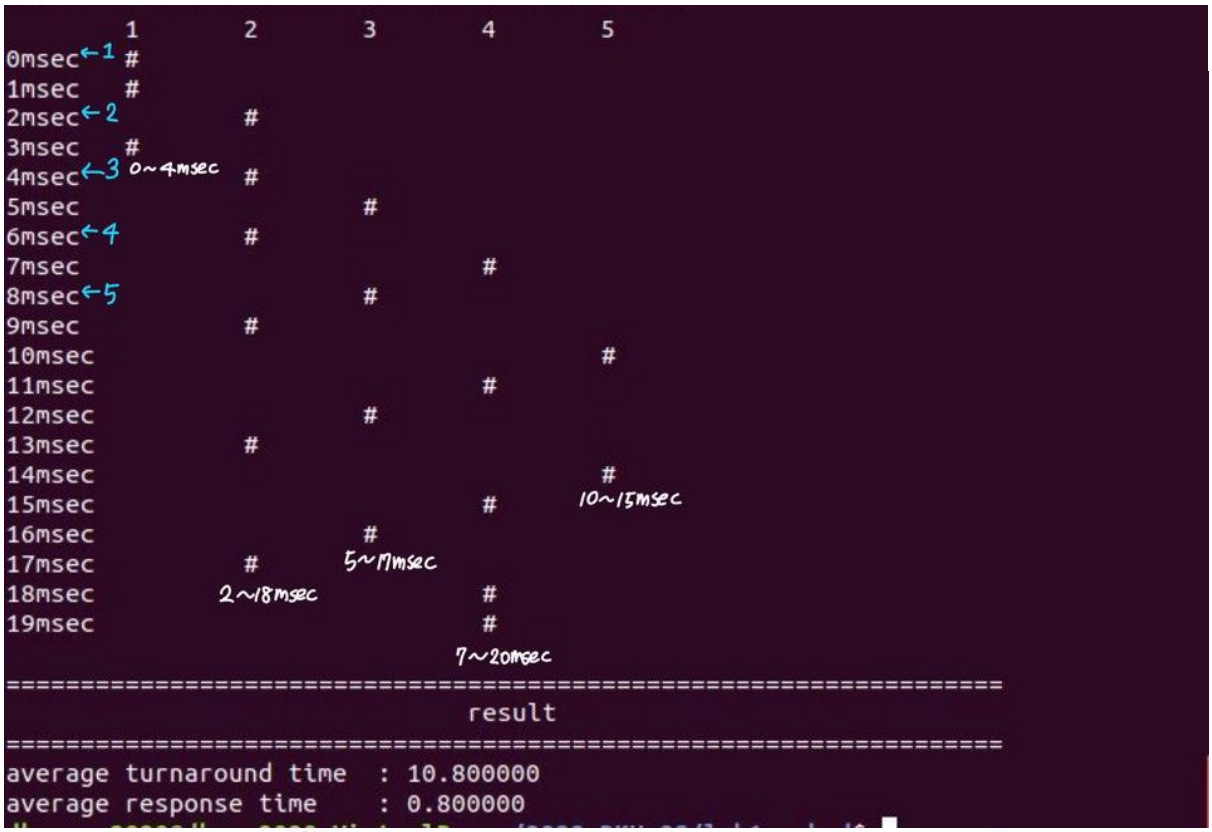


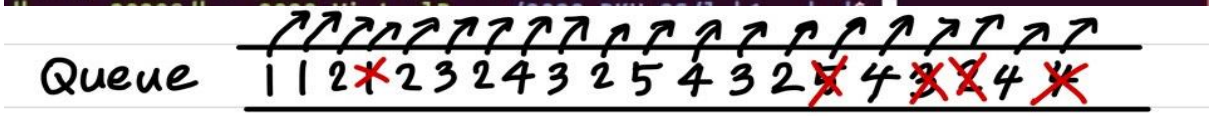
● 평균 반환 시간: $\frac{3+7+9+12+12}{5} = 8.6$ ● 평균 응답 시간: $\frac{0+1+5+7+10}{5} = 4.6$

Lecture Note 3의 24페이지에 나와있는 Workload에 대한 FCFS 알고리즘을 시뮬레이션 한 결과이다. 1번 프로세스가 0msec에 도착했기 때문에 제일 먼저 스케줄링 된다. 1번 프로세스를 실행하던 중인 2msec에 2번 프로세스가 도착하고, 2번 프로세스는 1번이 끝난 후인 3msec에 스케줄링 된다. 2번 프로세스가 실행 중일 때는 3, 4, 5번 프로세스가 도착한다. 2번 프로세스가 수행을 마친 후에는 그 중 먼저 도착했던 3번 프로세스부터 스케줄링 된다. 나머지 4, 5번 프로세스도 그 전 프로세스의 수행이 끝난 후 도착 순서대로 실행된다.

(2) Round-Robin

a: time slice = 1일 때

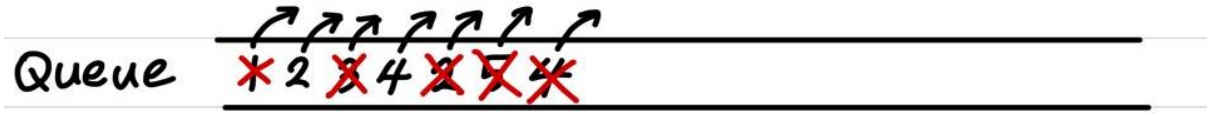




● 평균 반환 시간: $\frac{4+16+13+14+7}{5} = 10.8$ ● 평균 응답 시간: $\frac{0+0+1+1+2}{5} = 0.8$

b: time slice = 4일 때

```
        1       2       3       4       5
0msec←1 #
1msec   #
2msec←2 #
3msec   0~3msec #
4msec←3         #
5msec           #
6msec←4         #
7msec                   #
8msec←5                 #
9msec                   #
10msec                  #
11msec                  7~11msec #
12msec                           #
13msec                           #
14msec                           #
15msec          #
16msec          #
17msec          3~17msec                #
18msec                                  #
19msec                                  17~19msec
                                #
                                11~20msec
======================================================================
                            result
======================================================================
average turnaround time  : 10.000000
average response time    : 3.600000
```



● 평균 반환 시간: $\frac{3+15+7+14+11}{5} = 10$ ● 평균 응답 시간: $\frac{0+1+3+5+9}{5} = 3.6$

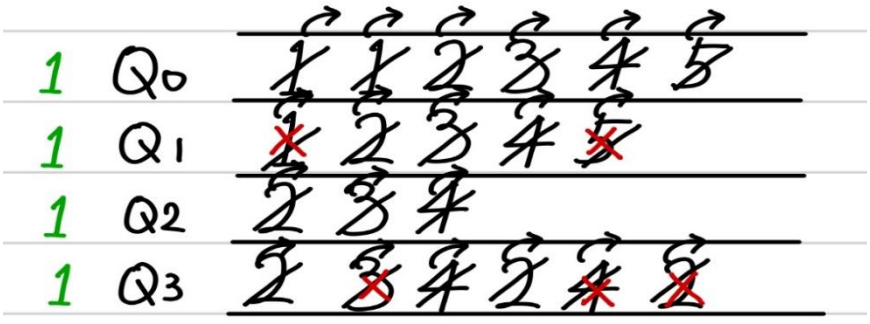
위의 실행 결과들은 Lecture Note 3의 24페이지에 나와있는 Workload에 대한 Round-Robin 알고리즘을 적용하여 스케줄링한 결과이다. 각각 타임 슬라이스를 1msec, 4msec로 주고 시뮬레이션했다. 프로세스들의 속성을 가지고 직접 큐를 그려가면서 결과를 비교해본 결과, 출력 결과가 예측한 대로 잘 나왔다.

타임 슬라이스를 1msec로 짧게 줬을 때는 4msec로 줬을 때보다 평균 응답 시간이 빨랐으며, 평균 반환 시간은 약간 늦었다. 또한 이 결과를 FCFS 알고리즘을 적용했을 때의 결과와 비교했다. 평균 응답 시간이 4.6msec이었던 FCFS에 비해 평균 응답 시간은 0.8msec(타임 슬라이스=1msec 기준)로 훨씬 빨라졌다. 반면에 평균 반환 시간은 4.6msec였던 FCFS에 비해 10.8msec(타임 슬라이스=1msec 기준)로 늦어졌다. Round-Robin 스케줄링은 타임 슬라이스를 작게 줄수록 응답 시간이 빨라져 대화형 프로세스들이 주를 이루는 시스템에서 사용하기 좋다는 것을 확인할 수 있었다.

(3) MLFQ

a: time slice=1일 때

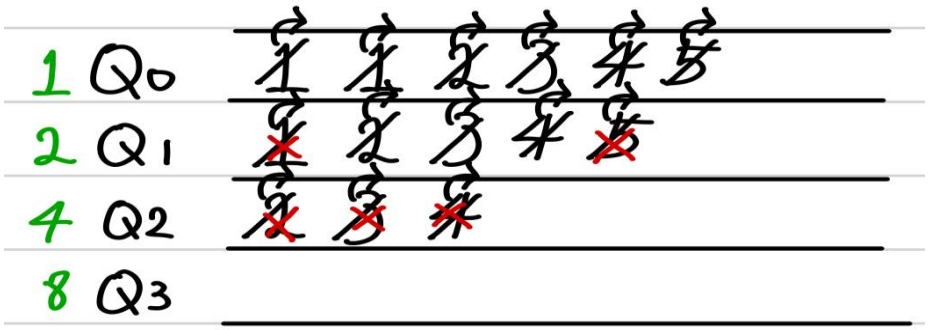
```
*********************************MLFQ*********************************
[MLFQ]Input boosting time>> 100
        1       2       3       4       5
0msec   #
1msec   #
2msec           #
3msec   #
4msec           #
5msec           #
6msec                   #
7msec                   #
8msec                           #
9msec                   #
10msec                          #
11msec          #
12msec                  #
13msec                  #
14msec          #
15msec                  #
16msec                  #
17msec          #
18msec                  #
19msec          #
=============================================================================
                                result
=============================================================================
average turnaround time  : 10.000000
average response time    : 0.000000
```



● 평균 반환 시간: $\frac{4+18+12+13+3}{5} = 10$ ●평균 응답 시간: $\frac{0+0+0+0+0}{5} = 0$

b: time slice = 2^i 일 때

```
*********************************MLFQ*********************************
[MLFQ]Input boosting time>> 100
        1       2       3       4       5
0msec   #
1msec   #
2msec           #
3msec   #
4msec           #
5msec           #
6msec           #
7msec                   #
8msec                           #
9msec                   #
10msec                  #
11msec                  #
12msec                  #
13msec                          #
14msec          #
15msec          #
16msec          #
17msec          #
18msec                  #
19msec                  #
==========================================================================
                                result
==========================================================================
average turnaround time  : 10.600000
average response time    : 0.200000
```



●평균 반환 시간: $\frac{4+15+14+14+6}{5} = 10.6$ ●평균 응답 시간: $\frac{0+0+0+1+0}{5} = 0.2$

a는 모든 레디 큐가 타임 슬라이스를 동일하게 갖는 경우를, b는 높은 우선순위 큐일수록 짧은 타임 슬라이스를 갖는 경우를 시뮬레이션한 결과이다. 부스트되는 경우를 제외하기 위해 부스트 타임은 매우 큰 수로 주었다.

평균 반환시간과 응답시간 모두 a가 b보다 나은 것을 확인할 수 있다. 이것은 a의 모든 큐의 퀀텀이 1이기 때문이다. 새로운 프로세스가 들어왔을 때, 스케줄러는 이것을 곧장 최상위 큐에 넣어 실행한다. 따라서 a의 경우, 프로세스의 도착 시간이 처음 수행되는 시간과 동일하다. 따라서 반환 시간과 응답 시간이 b보다 짧다.

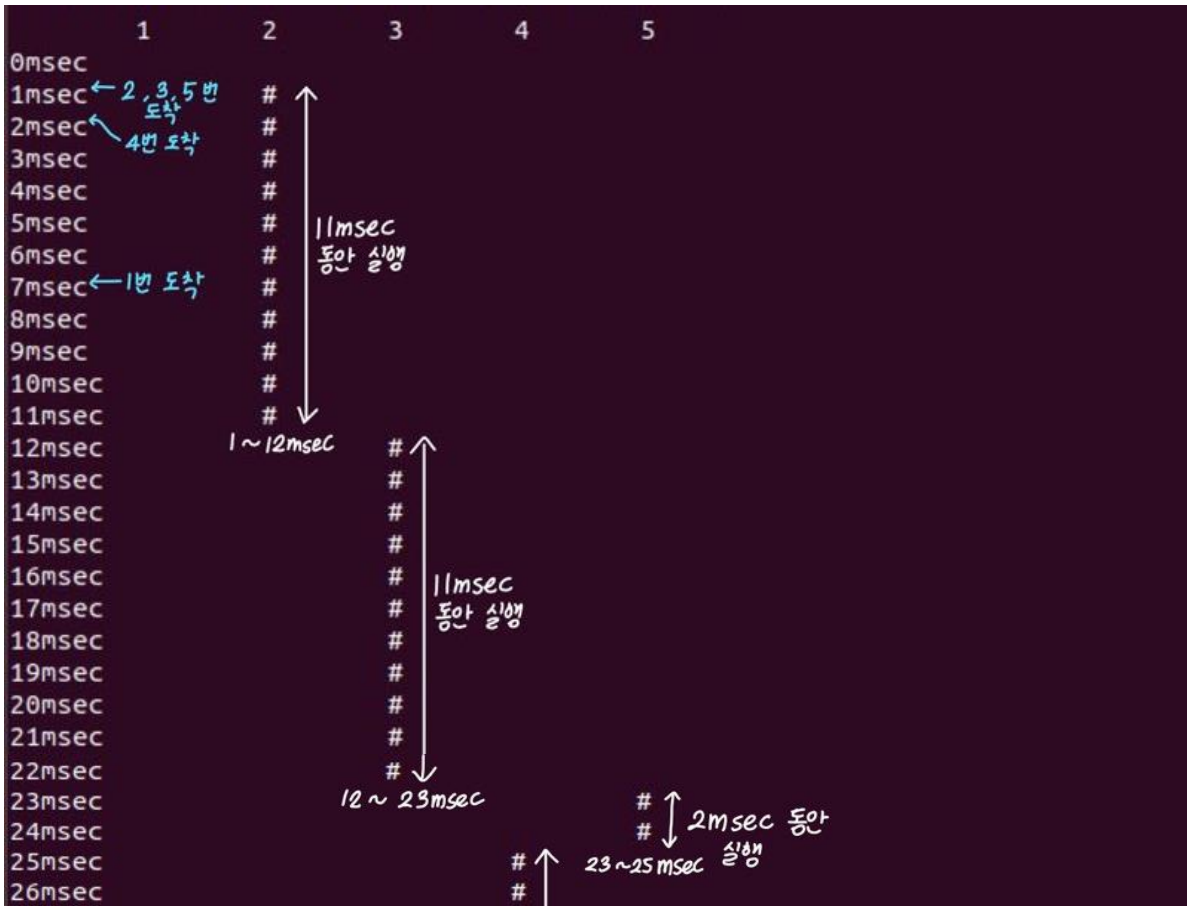
Round Robin의 a와 MLFQ의 a를 비교하면 MLFQ가 RR보다 효율적인 방식임이 확인된다. RR방식은 큐가 하나밖에 없기 때문에 도착한 프로세스가 바로 수행될 수 없다. 큐에 있던 모든 프로세스가 먼저 수행되어야 한다. 하지만 MLFQ는 큐를 다층으로 두고 우선순위를 둠으로써 ready상태로 전이된 프로세스가 우선적으로 수행되게 만든다. 따라서 MLFQ는 RR보다 대화형 작업에 더 유리하다.

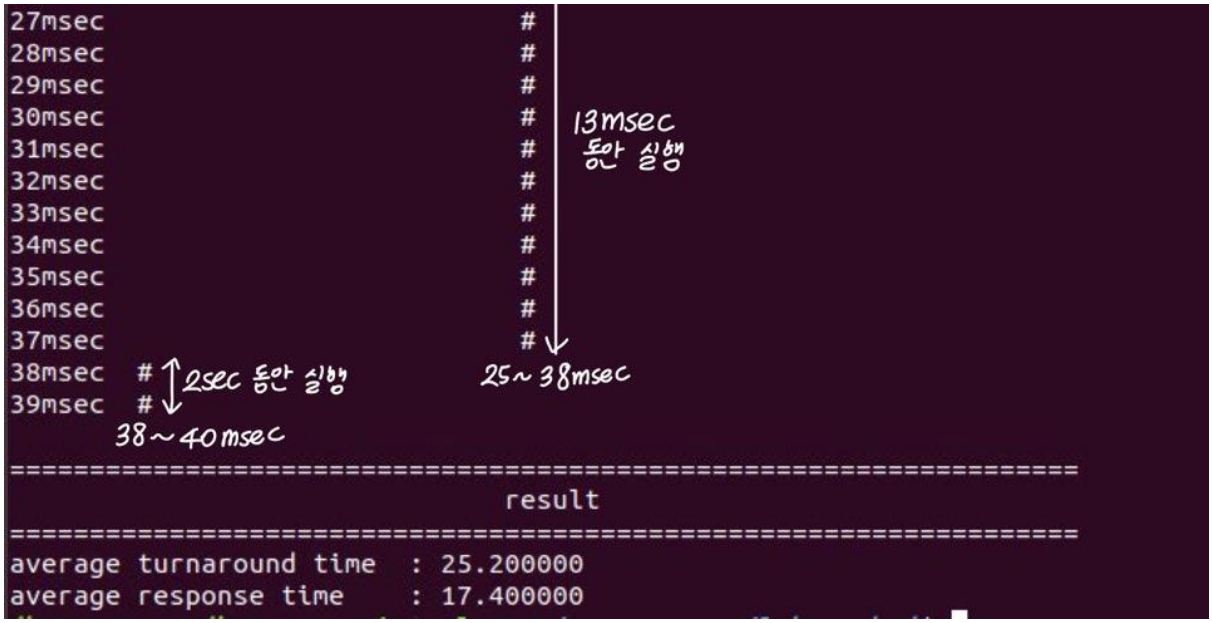
하지만 RR과 MLFQ의 a가 항상 좋은 스케줄러는 아니다. 실제 프로세스가 상태전이하기 위해서는 문맥교환이 필요하다. 수행중인 프로세스는 현재 상태를 PCB에 저장하고 레디 상태로, 레디 상태였던 프로세스는 PCB에 저장했던 문맥을 복원한다. 문맥교환은 프로세스 수행을 지연시킨다. 따라서 잦은 문맥교환이 발생하는 a는 오버헤드를 전혀 고려하지 않은 스케줄링 방식이다. 수행해야 할 프로세스가 많을 때에도 a방식의 스케줄러가 실제로 위와 같은 좋은 반환 시간을 보여줄지는 장담할 수 없다.

2-1) 랜덤으로 생성한 워크로드에 대한 시뮬레이션 (FCFS, Round-Robin)

```
********** Attribution of processes **********
* id                1
* Arrive time       7
* Service time      2
*******************************************
* id                2
* Arrive time       1
* Service time     11
*******************************************
* id                3
* Arrive time       1
* Service time     11
*******************************************
* id                4
* Arrive time       2
* Service time     13
*******************************************
* id                5
* Arrive time       1
* Service time      2
*******************************************
```

(1) FCFS



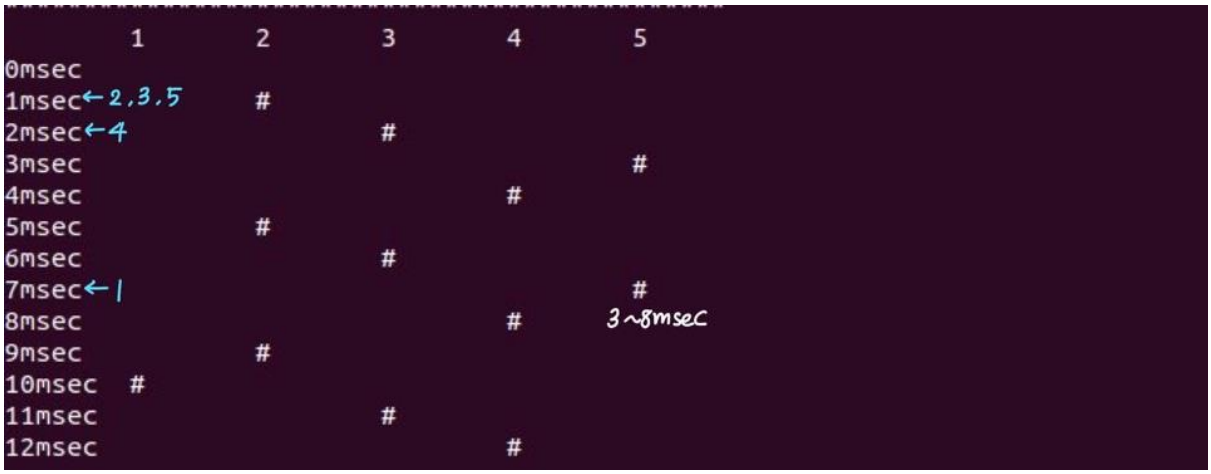


● 평균 반환 시간: $\frac{33+11+22+36+24}{5} = 25.2$ ● 평균 응답 시간: $\frac{31+0+11+23+22}{5} = 17.4$

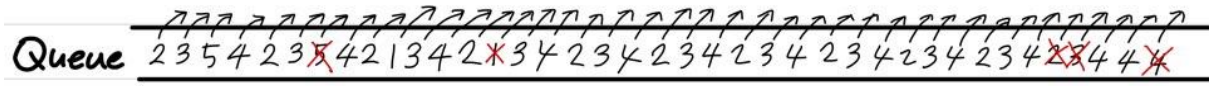
출력된 프로세스들의 정보를 바탕으로 시뮬레이션 결과를 비교해봤더니 스케줄링이 예측한대로 이루어졌다. 도착 시간이 빠른 프로세스가 우선적으로 스케줄링 되고, 수행 중인 프로세스의 실행이 끝나야 다른 프로세스가 스케줄링 될 수 있는 FCFS의 특징대로 스케줄링이 이뤄짐을 확인할 수 있었다.

(2) Round-Robin

a: time slice = 1일 때



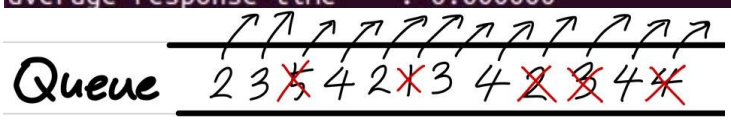
```
13msec          #
14msec  #
15msec 10~15msec          #
16msec                              #
17msec          #
18msec                    #
19msec                              #
20msec          #
21msec                    #
22msec                              #
23msec          #
24msec                    #
25msec                              #
26msec          #
27msec                    #
28msec                              #
29msec          #
30msec                    #
31msec                              #
32msec          #
33msec                    #
34msec                              #
35msec          #
36msec        1~36msec    #
37msec                  2~37msec    #
38msec                              #
39msec                              #
                                  4~40msec
======================================================================
                              result
======================================================================
average turnaround time  : 24.800000
average response time    : 1.600000
```



● 평균 반환 시간: $\frac{8+35+36+38+7}{5} = 24.8$ ● 평균 응답 시간: $\frac{3+0+1+2+2}{5} = 1.6$

b: time slice = 4일 때

```
       1       2       3       4       5
0msec
1msec←2,3,5    #
2msec←4        #
3msec          #
4msec          #
5msec                  #
6msec                  #
7msec←1                #
8msec                  #
9msec                                  #
10msec                                 #
11msec                         #       9~11msec
12msec                         #
13msec                         #
14msec                         #
15msec         #
16msec         #
17msec         #
18msec         #
19msec  #
20msec  #
21msec 19~21msec       #
22msec                 #
23msec                 #
24msec                 #
25msec                         #
26msec                         #
27msec                         #
28msec                         #
29msec         #
30msec         #
31msec         #
32msec       1~32msec  #
33msec                 #
34msec                 #
35msec              5~35msec   #
36msec                         #
37msec                         #
38msec                         #
39msec                         #11~40msec
================================================================
                          result
================================================================
average turnaround time  : 25.400000
average response time    : 6.600000
```



● 평균 반환 시간: $\frac{14+31+34+38+10}{5} = 25.4$ ● 평균 응답 시간: $\frac{12+0+4+9+8}{5} = 6.6$

타임 슬라이스를 1msec, 4msec로 달리 설정해서 Round-Robin을 시뮬레이션 해봤다. 첫 번째 워크로드에서 이미 살펴봤듯이 여기서도 타임 슬라이스를 상대적으로 작게 1msec로 설정했을 때의 평균 응답 시간이 5msec 더 빨랐다.

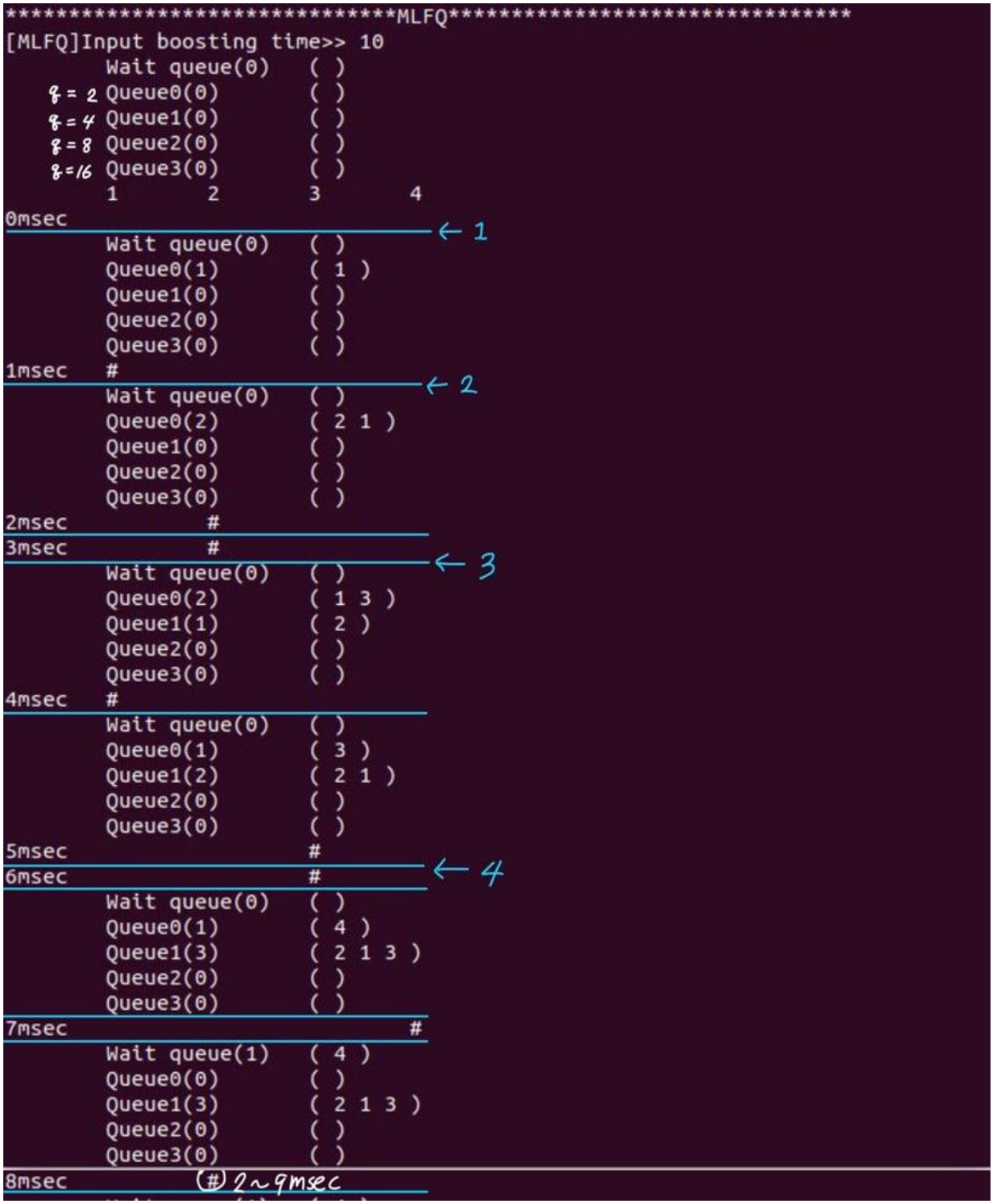
하지만 평균 반환 시간에 대한 결과는 첫 번째 워크로드와 다른 양상을 보였다. 첫 번째 워크로드에 대해서 시뮬레이션했을 때에는 타임 슬라이스가 1msec일 때보다 4mec일 때 평균 반환 시간이 더 작게 나왔고, FIFO일 때는 4msec일 때보다 더 작게 나왔다. 그런데 두 번째 워크로드에 대해서는 FIFO일 때 평균 반환 시간이 25.2msec, Round-Robin에서 타임 슬라이스가 1msec일 때 24.8msec, 타임 슬라이스가 4msec일 때 25.4msec로 나왔다.

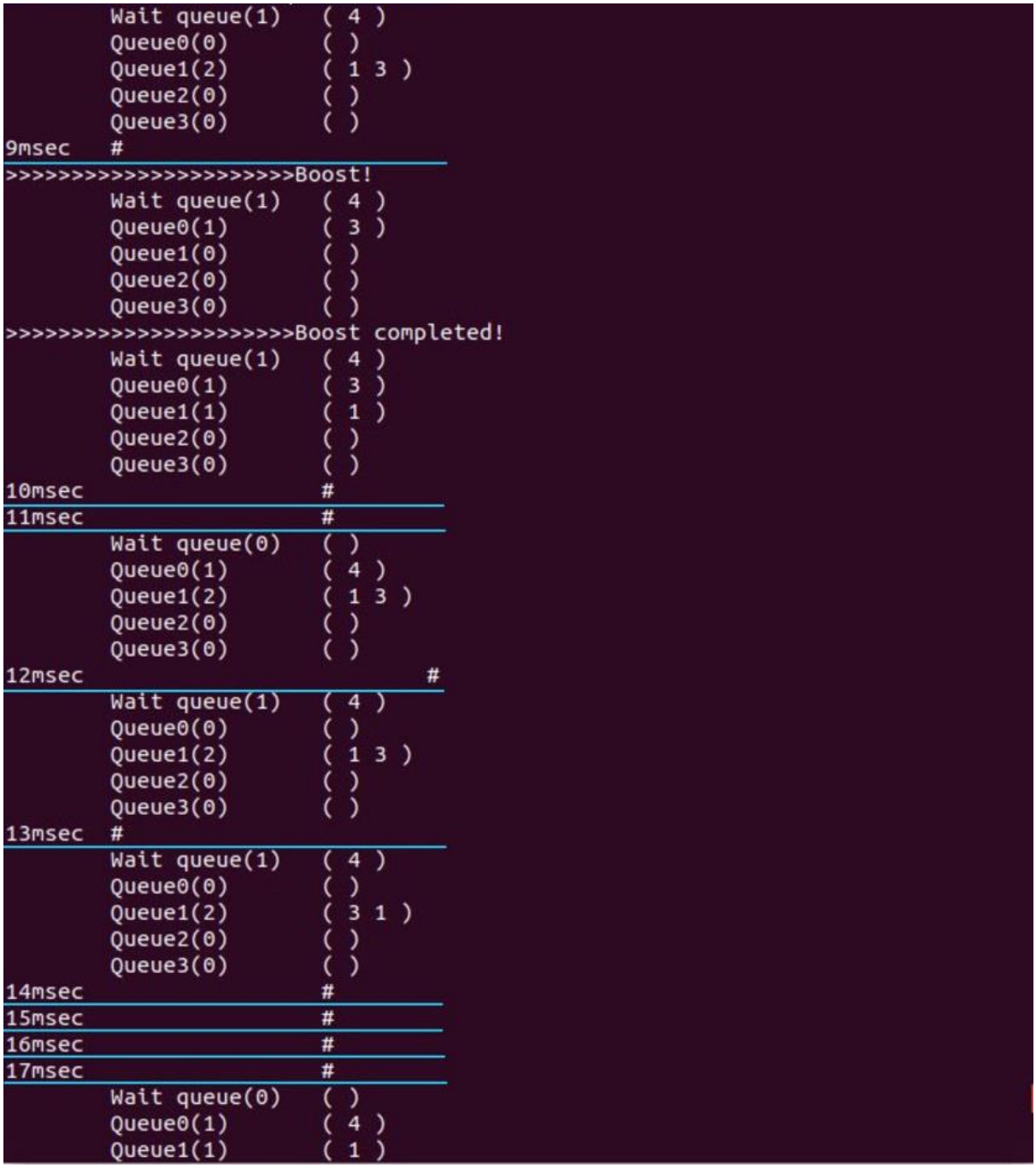
이렇게 첫 번째 워크로드와 두 번째 워크로드에서 결과가 달라진 것은 스케줄링의 대상이 되는 프로세스들의 특성이 다르기 때문이다. 첫 번째 워크로드에서 5개의 프로세스들의 서비스 시간을 살펴보면 제일 짧은 서비스 시간이 2msec이고, 제일 긴 서비스 시간이 6msec로 큰 차이가 나지 않는다. 그리고 동시에 도착하는 프로세스들이 없고 시간 간격을 두고 도착한다. 반면에 두 번째 워크로드는 3개의 프로세스들이 동시에 도착하고, 그 중에는 서비스 시간이 2msec로 짧은 5번 프로세스와 서비스 시간이 11msec로 긴 2, 3번 프로세스가 포함돼있다. 이때 긴 프로세스인 2번, 3번이 먼저 스케줄링 되고 FIFO에서는 그 프로세스들의 실행을 끝낸 후에 5번 프로세스를 스케줄링한다. 따라서 짧게 수행하고 끝낼 수 있는 5번 프로세스의 반환 시간이 많이 지연된다. 하지만 Round-Robin에서는 선점이 되기 때문에 2, 3번 프로세스들이 먼저 스케줄링 되더라도 5번 프로세스가 금방 스케줄링 되어 수행을 마치므로 반환 시간이 FIFO에 비해 좋게 된다. 즉 응답 시간은 항상 FIFO보다 Round-Robin에서 좋지만, 반환 시간은 워크로드의 특성에 따라 FIFO에서 좋을 수도, Round-Robin에서 좋을 수도 있다.

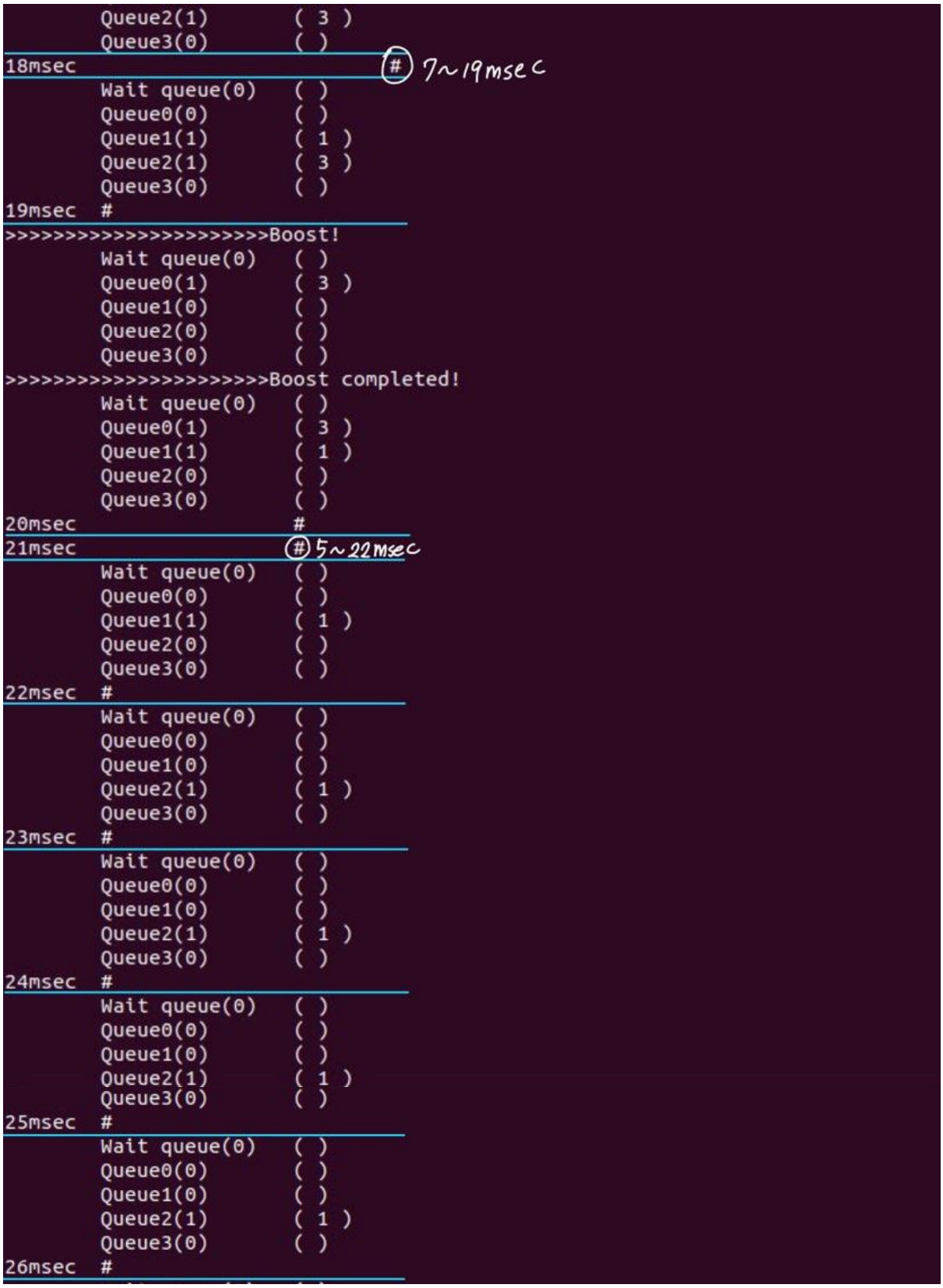
2-2) 랜덤으로 생성한 워크로드에 대한 시뮬레이션 (MLFQ)

```
**********Attributes of processes(FCFS, RR, MLFQ)*******
* id                1
* Arrive time       1
* Service time     11
* Fake job: Return CPU maliciously

*********************************************************************
* id                2
* Arrive time       2
* Service time      3
* Bach job
*********************************************************************
* id                3
* Arrive time       4
* Service time     10
* Bach job
*********************************************************************
* id                4
* Arrive time       6
* Service time      3
* Interactive job
*********************************************************************
```

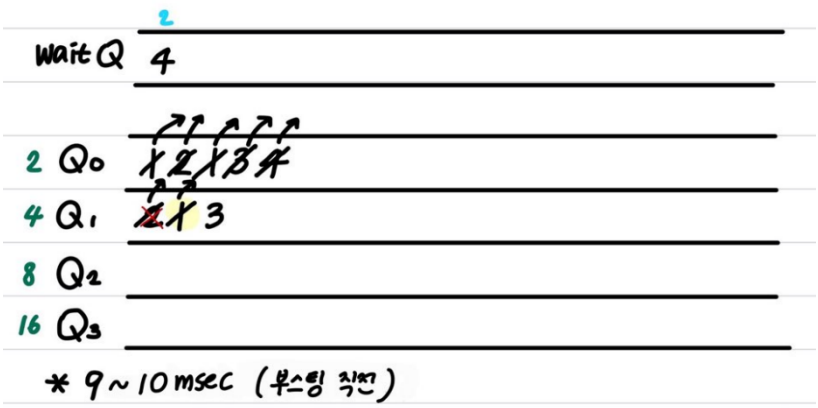
4개의 프로세스를 랜덤으로 생성하고, 큐의 레벨은 4개로 만들어 MLFQ 스케줄링을 시뮬레이션 해봤다. 이때 부스팅 주기는 10msec로 두었고, 큐 별로 타임 슬라이스를 2^{i+1} 로 다르게 두었다. (2, 4, 8, ..)



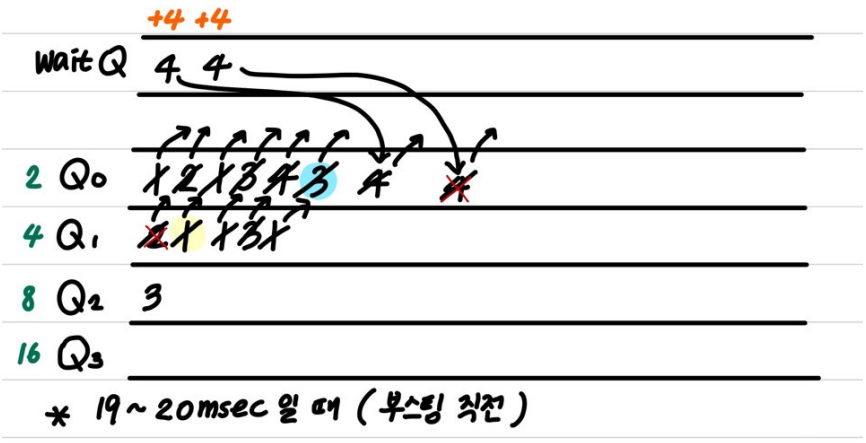

Wait queue(1) (4)
Queue0(0) ( )
Queue1(2) (1 3)
Queue2(0) ( )
Queue3(0) ( )
9msec #
>>>>>>>>>>>>>>>>>>>>>>Boost!
Wait queue(1) (4)
Queue0(1) (3)
Queue1(0) ( )
Queue2(0) ( )
Queue3(0) ( )
>>>>>>>>>>>>>>>>>>>>>>Boost completed!
Wait queue(1) (4)
Queue0(1) (3)
Queue1(1) (1)
Queue2(0) ( )
Queue3(0) ( )
10msec #
11msec #
Wait queue(0) ( )
Queue0(1) (4)
Queue1(2) (1 3)
Queue2(0) ( )
Queue3(0) ( )
12msec #
Wait queue(1) (4)
Queue0(0) ( )
Queue1(2) (1 3)
Queue2(0) ( )
Queue3(0) ( )
13msec #
Wait queue(1) (4)
Queue0(0) ( )
Queue1(2) (3 1)
Queue2(0) ( )
Queue3(0) ( )
14msec #
15msec #
16msec #
17msec #
Wait queue(0) ( )
Queue0(1) (4)
Queue1(1) (1)


Queue2(1) (3)
Queue3(0) ( )
18msec #
7~19msec
Wait queue(0) ( )
Queue0(0) ( )
Queue1(1) (1)
Queue2(1) (3)
Queue3(0) ( )
19msec #
>>>>>>>>>>>>>>>>>>>>>Boost!
Wait queue(0) ( )
Queue0(1) (3)
Queue1(0) ( )
Queue2(0) ( )
Queue3(0) ( )
>>>>>>>>>>>>>>>>>>>>>Boost completed!
Wait queue(0) ( )
Queue0(1) (3)
Queue1(1) (1)
Queue2(0) ( )
Queue3(0) ( )
20msec #
21msec #
5~22msec
Wait queue(0) ( )
Queue0(0) ( )
Queue1(1) (1)
Queue2(0) ( )
Queue3(0) ( )
22msec #
Wait queue(0) ( )
Queue0(0) ( )
Queue1(0) ( )
Queue2(1) (1)
Queue3(0) ( )
23msec #
Wait queue(0) ( )
Queue0(0) ( )
Queue1(0) ( )
Queue2(1) (1)
Queue3(0) ( )
24msec #
Wait queue(0) ( )
Queue0(0) ( )
Queue1(0) ( )
Queue2(1) (1)
Queue3(0) ( )
25msec #
Wait queue(0) ( )
Queue0(0) ( )
Queue1(0) ( )
Queue2(1) (1)
Queue3(0) ( )
26msec #

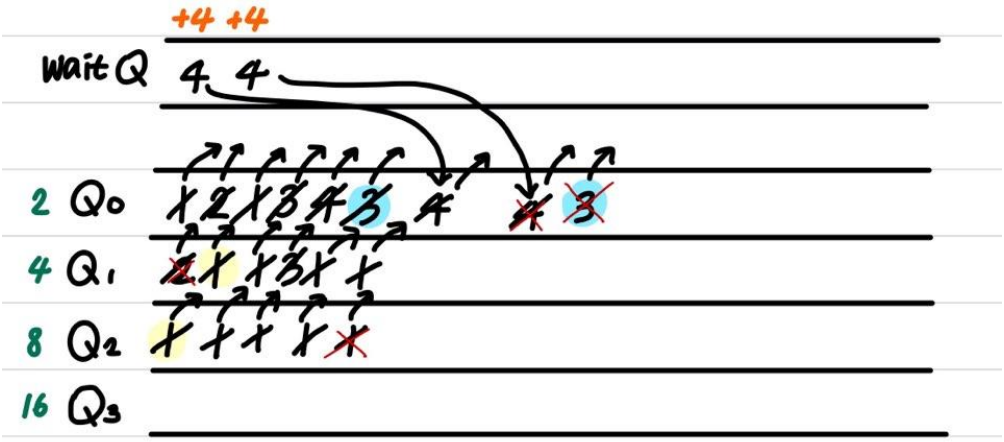
```
        Wait queue(0)   ( )
        Queue0(0)       ( )
        Queue1(0)       ( )
        Queue2(1)       ( 1 )
        Queue3(0)       ( )
27msec  #) 1~28msec
        Wait queue(0)   ( )
        Queue0(0)       ( )
        Queue1(0)       ( )
        Queue2(0)       ( )
        Queue3(0)       ( )

================================================================
                            result
================================================================
average turnaround time  : 16.250000
average response time    : 0.500000
```



위 그림은 9msec에서 1번 프로세스가 실행될 때의 큐의 상태를 나타낸 것이다. 위 그림의 1번 큐에서 연노랑색으로 표시된 1번 프로세스는 fake job이므로 타임 슬라이스를 채워 실행되지는 않지만, 규칙4에 의해 0번 큐에서 할당된 시간을 모두 썼기 때문에 우선 순위가 내려와있다. 4번 프로세스는 I/O를 하는 job이므로 1초간 실행됐다가 wait 상태가 되어 wait큐로 들어간 모습이다. 그리고 이제 10msec에서 부스팅이 일어나게 되어 3번 프로세스가 0번 큐로 올라오게 된다. 다음 그림의 0번 큐에서 하늘색으로 표시된 3번 프로세스가 이에 해당된다.



위 그림은 10msec에서 부스팅이 된 후 계속 스케줄링이 실행되어 19msec를 수행하고 있을 때 큐를 나타낸 것이다. 그동안 wait 큐에 있던 프로세스들도 wake up 되고 0번 큐에 들어와 수행됐음을 볼 수 있다. 그리고 다시 한 번 20msec에서 부스팅이 일어나면 3번 프로세스가 다시 우선 순위가 높아져 0번 큐로 들어가게 된다.



위의 그림은 모든 프로세스가 수행을 종료했을 때의 상태를 그린 것인데, 0번 큐에 하늘 색으로 표시된 두 번째 3번 프로세스가 20msec에서 부스팅될 때 올라왔던 것이다. 수행 시간이 길고, 타임 슬라이스를 꽉 채워서 실행하는 Batch job이라 우선 순위가 내려갔던 3번 프로세스가 부스팅을 통해 기아 상태를 피하고 주기적으로 실행될 수 있었다.

● 평균 반환 시간: $\frac{27+7+18+13+}{4} = 16.25$ ● 평균 응답 시간: $\frac{0+0+1+1}{4} = 0.5$

이 워크로드를 FCFS로 스케줄링했을 때는 평균 반환 시간이 16.75msec, 평균 응답 시간이 10msec였으며, Round-Robin으로 스케줄링했을 때는 평균 반환 시간이 17msec, 평균 응답 시간이 4.25msec로 나타났다. 평균 응답 시간을 최소화하면서 평균 반환 시간을 최적화하려는 MLFQ의 목적을 어느 정도 달성할 수 있었다.

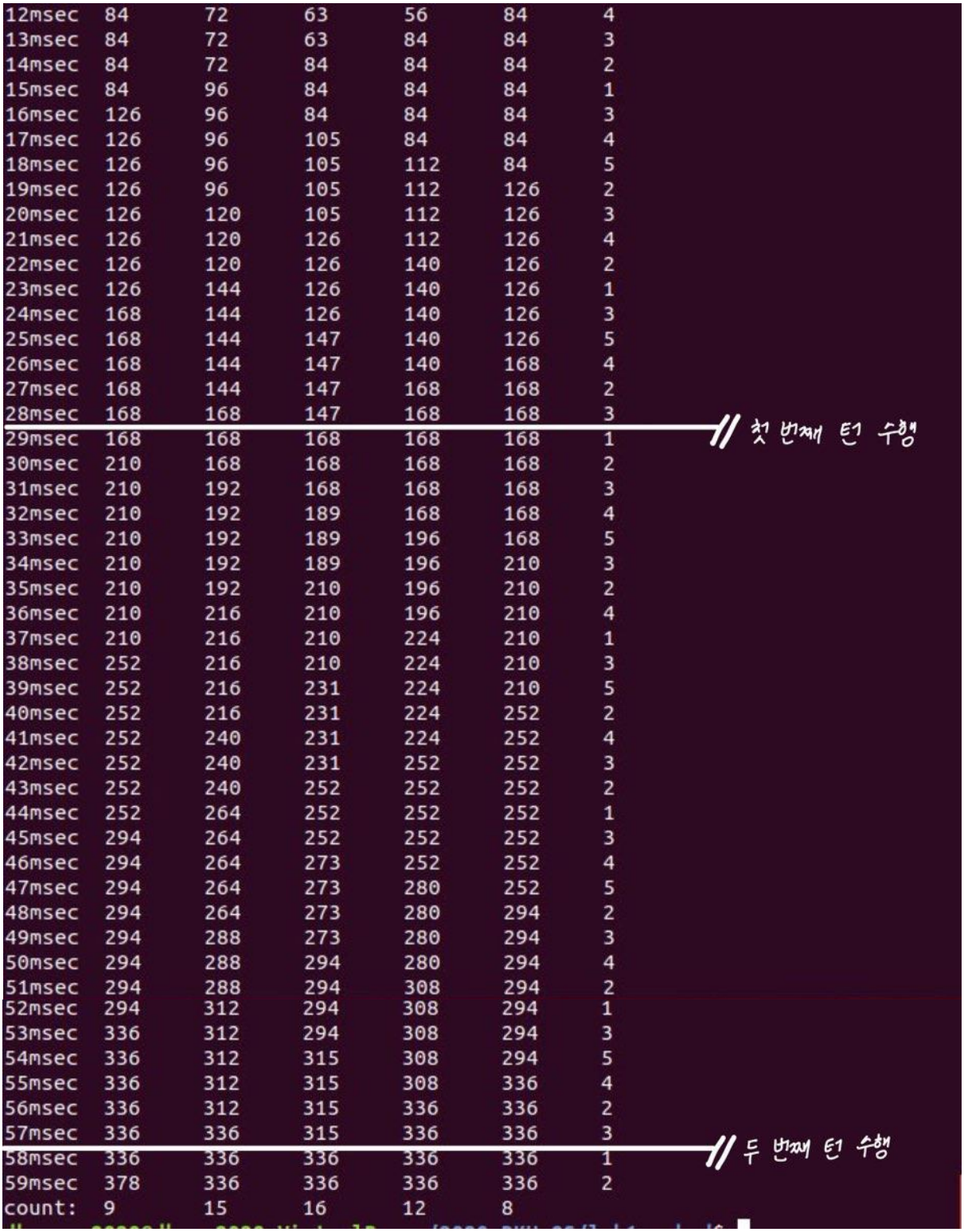
그리고 MLFQ 시뮬레이션을 해보는 과정에서 부스팅 주기를 결정하는 것이 중요하다는 것을 깨달았다. 먼저 부스팅 주기가 짧으면 기아 상태를 피할 수 있다는 장점이 있다. 하지만 부스팅이 자주 일어나면 MLFQ에서 job들에 대해 우선 순위를 부여했던 규칙들이 사실상 지켜지지 않으므로 대화형 작업에 대해서 불리하다. 뿐만 아니라 부스트로 인해 올라오는 job들 때문에 짧은 시간에 끝날 수 있는 job들도 스케줄링이 밀리므로 반환 시간이 좋지 않게 될 수 있다. 또한 같은 부스팅 주기를 가진 시스템이어도 워크로드의 특성이 어떠하냐에 따라 그 성능이 다르게 나타나므로 그 주기의 결정을 신중하게 해야 한다.

3) Stride 스케줄링 시뮬레이션

Stride 스케줄링을 시뮬레이션 할 때는 따로 만든 makeStride_Processes 함수를 이용해 랜덤으로 워크로드를 생성했다.

다음은 워크로드에 대해 타임 슬라이스를 1msec, 시뮬레이션 시간은 60msec로 지정하여 시뮬레이션 한 결과이다.

```
********** Attribution of processes **********
* id                1
* Ticket Value     20
* Stride value     42
*******************************************
* id                2
* Ticket Value     35
* Stride value     24
*******************************************
* id                3
* Ticket Value     40
* Stride value     21
*******************************************
* id                4
* Ticket Value     30
* Stride value     28
*******************************************
* id                5
* Ticket Value     20
* Stride value     42
*******************************************
        1       2       3       4       5       RUN
=============================================================
0msec   0       0       0       0       0       1
1msec   42      0       0       0       0       2
2msec   42      24      0       0       0       3
3msec   42      24      21      0       0       4
4msec   42      24      21      28      0       5
5msec   42      24      21      28      42      3
6msec   42      24      42      28      42      2
7msec   42      48      42      28      42      4
8msec   42      48      42      56      42      1
9msec   84      48      42      56      42      3
10msec  84      48      63      56      42      5
11msec  84      48      63      56      84      2
```



passvalue가 동일한 경우에는 프로세스의 id가 작은 순서대로 스케줄링 되도록 했기 때문에 처음에는 1번 프로세스가 스케줄링되고, passvalue에 stride값에 해당하는 42를 더한다. 1msec에서는 2, 3, 4, 5번 프로세스의 passvalue가 동일하다. 0msec에서와 마찬가지로 id가 작은 2번 프로세스가 스케줄링되고, passvalue에는 stride값인 24가 더해진다. 이 과정을 계속하여 반복하다 보면 모든 프로세스의

passvalue가 다시 같아지는 시점이 생긴다. 그 시점까지를 한 턴이라고 볼 수 있다. 여기서는 29msec가 첫 번째 턴이다.

이때 각 프로세스들이 스케줄링된 횟수를 세어보면 다음과 같다. 1번 프로세스는 4번, 2번 프로세스는 7번, 3번 프로세스는 8번, 4번 프로세스는 6번, 5번 프로세스는 4번이다. 이 횟수는 각 프로세스의 티켓값인 20:35:40:30:20의 비인 4:7:8:6:4와 같은 비율이다. 그리고 다음 턴까지 프로세스들이 스케줄링된 횟수를 세어보면 또 같은 비율이 나타난다. 시뮬레이션이 끝난 60msec까지 각 프로세스들은 9번, 15번, 16번, 12번, 8번 스케줄링 됐다. 시뮬레이션을 끝냈을 때 한 턴이 완전히 종료된 시점이 아니므로 티켓 값의 비율을 완전히 만족하지는 않지만, 4.5:7.5:8:6:4로 어느 정도 그 비율에 맞춰 스케줄링 됐음을 확인할 수 있다. 즉 Stride 스케줄링에서는 매 턴마다 티켓 값의 비율에 맞춰 동일하게 스케줄링이 이뤄지기 때문에 각 프로세스마다 CPU 할당량을 지정하고 그 비율을 어느 정도 보장하여 실행시키는 것이 가능하다.

5. Discussion

과제를 진행하면서 몇 가지 어려웠던 일이 있었다. 자료구조를 선택하는 일, MLFQ 정책에 대한 자세한 이해에 대해 생각을 많이 했다. 특히 MLFQ 구현이 어려웠다.

1) 자료구조 선택

(1) 워크로드

프로세스 정보를 담은 프로세스 구조체를 만든 후, 우리는 이것을 어떻게 관리할지 고민했다. 워크로드는 어떤 방법으로 만드는 것이 적절할까? 프로세스 배열과 연결리스트 중 어떤 것을 사용해야 원하는 기능을 쉽게 구현할 수 있을까? 토의 끝에 우리는 워크로드 타입으로서 프로세스 구조체 배열을 선택했다.

FCFS, RoundRobin, MLFQ, Stride 정책의 특징을 확인하기 위해서는 스케줄러 함수를 동일한 워크로드로 실행한 결과를 비교해야 한다. 따라서 프로세스 구조체들로 구성된 워크로드는 한번 정해지면 중간에 재정의되거나 추가되지 않는다. 이러한 특성 때문에 연결리스트보다 배열이 더 적합하다고 생각했다.

(2) 멀티 레벨 큐

프로세스를 실행 큐에 어떻게 연결할지 고민했다. 한번에 몇 개의 프로세스가 큐에 달릴지 예측할 수 없으므로 실행 큐는 연결 리스트로 구현해야 하는 것이 명백했다. 그렇다면 MLFQ에서 사용하는 여러 개의 큐는 어떻게 구현해야 할까?

처음에는 MLFQ에서 사용하는 여러 개의 큐를 프로세스를 연결한 것과 같이 연결하고 싶었다. 하지만 라운드로빈에서 사용하는 큐는 오직 한 개이므로 다음 큐를 가리키는 포인터가 필요하지 않았다. 게다가 MLFQ에서 필요하다고 해서 라운드로빈과 관련 없는 멤버를 두 스케줄러가 사용하는 큐 구조체에 추가하는 것은 부적절해 보였다. 기존 라운드로빈 큐를 유지하면서, 라운드로빈의 큐 형태를 포함하는 자료구조가 필요했다. 그래서 〈아래 그림〉과 같은 QueueNode 구조체를 만들었다. 라운드로빈은 queue와 processNode를 사용하고 MLFQ는 queueNode와 queue, processNode를 사용하는 모델이다.

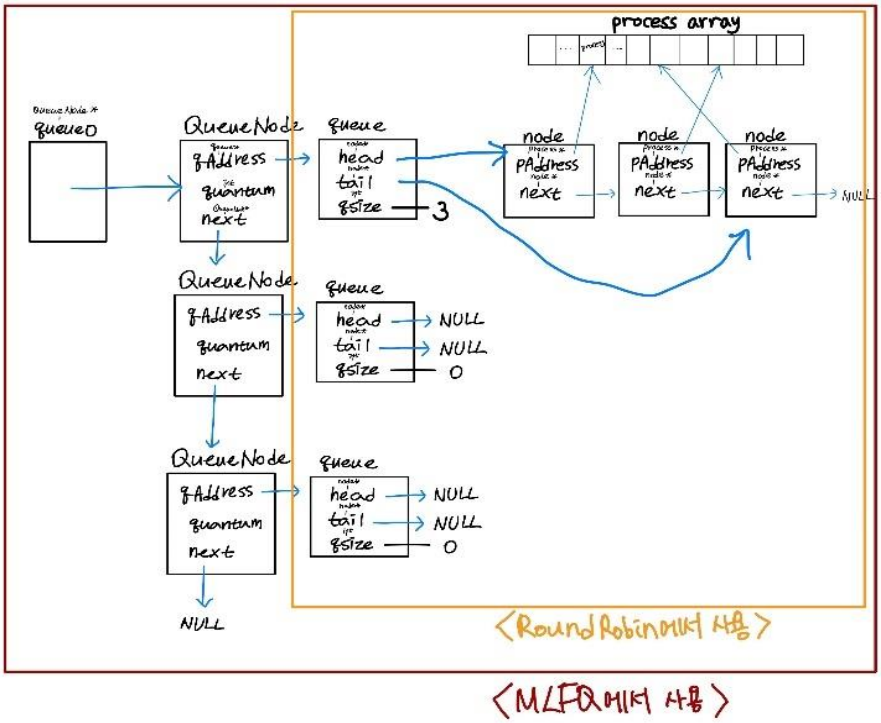


Figure 1 초기 큐 구조

구조는 복잡하지만 필요한 변수만 선언 되어있기 때문에 순조롭게 사용할 수 있을 것이라고 생각했다. 하지만 구조만 복잡할 뿐만 아니라 변수 이름도, 역할도 비슷해 큐와 프로세스를 사용할 때 실수를 많이 했다. 두 자료형을 매번 비교하면서 사용하다 보니 프로그래밍 시간도 점점 길어졌다. 긴 시간을 불편하게 코딩하고 나서야 큐를 다시 만들기로 했다.

이번에는 queue와 queueNode를 통일하는 방향으로 고쳐 생각해보았다. queue는 queueNode에 quantum과 next만 추가된 구조체였다. 라운드로빈은 큐가 하나여서 quantum을 굳이 정의해줄 필요가 없었지만 분명 quantum이라는 개념을 사용하고 있다. 따라서 quantum을 queue의 멤버로서 정의해도 무방했다. next는 여러 개의 큐를 배열로 묶어준다면 필요성이 없어지는 멤버였다. 사실 MLFQ에서 큐의 개수는 고정되어 있으므로 연결리스트로 구현할 필요가 없었다. 큐 배열로 만들어도 된다는 것을 뒤늦게 알아차렸다. 〈아래그림〉은 개선한 큐의 구조이다.

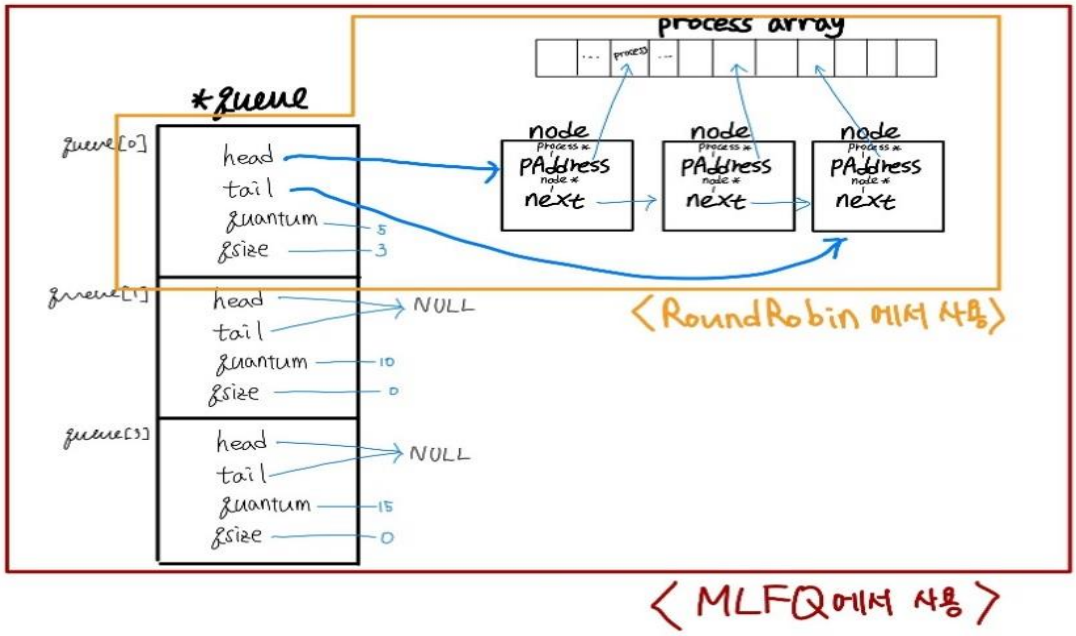


Figure 2 개선한 큐 구조

2) MLFQ에 대한 고민

우리는 MLFQ의 다섯 가지 규칙을 모두 반영하려고 했다. 몇 가지 언급되지 않은 부분을 어떻게 처리해야 할지 고민이 되었다. 또한 어떤 부분은 완전히 잘못 이해하여 난항을 겪었다.

(1) I/O 프로세스를 batch 프로세스와 구분하기 위한 방법.

I/O 프로세스를 배치 프로세스와 구분하는 방식을 고안하는 것이 어려웠다. 프로세스는 특정 시간 동안만 대화형으로 실행될 수도 있다. 이러한 경우 배치 프로세스에서 I/O 프로세스로 전환되는 시간, I/O가 끝나는 시간, I/O로서 실행되는 횟수는 어떻게 저장하며 어떻게 처리해야 할까?

이 많은 것들을 적용하여 프로그래밍 하는 것은 너무 어렵기 때문에 우리는 프로세스를 단순화하기로 했다. 프로세스는 배치거나 대화형 둘 중 하나의 특징만 가지며, 자신이 어떤 특징을 가진 job인지 스케줄러에게 미리 알려줄 수 있다.

이러한 가정을 두어 스케줄러를 구현했다.

(2) boost에 대한 잘못된 이해

처음에는 프로세스 별로 부스트되는 것으로 이해했다. 모든 job들이 각각 마지막으로 부스트된 시간을 기억하고 다시 일정 시간이 지나면 부스트된다고 생각했다.

그런데 만약 동시간에 부스트되는 프로세스들이 우연한 이유로 많아진 경우, 최상위 큐의 quantum 시간만큼 지났을 때 부스트된 프로세스들은 어떻게 처리해야 하는가? 강의노트 LN3의 20페이지를 보

면 최상위에 있던 IO잡들은 부스트에 의해 순서가 밀리는 것을 확인할 수 있다. IO잡은 빠르게 응답해야 하는데 배치 프로세스를 수행하기 위해 IO프로세스를 기약 없이 미루는 것은 정당한 일인가? 또한 부스트는 큐에 대한 함수(enqueue, dequeue와 같은 큐의 성질을 만드는 함수)로 구현할 수 없는 것이 아닌가? 부스트는 큐를 한순간 스택으로 만드는 것으로 보인다. 이것이 가능한지도 의문이지만, 스택처럼 관리된다면 부스트 된 순간에 새로 만들어진 프로세스는 부스트 프로세스보다 앞에 들어가는가? 아니면 뒤에 들어가는가?

꽤 오랫동안 우리는 이러한 질문에 답을 할 수 없었다. MLFQ가 배치 프로세스에 상대적으로 불리하기 때문에 만든 규칙 Rule5는, 이미 정의된 규칙을 무시하고 있었다. 부스트는 기아상태를 해결하지만, 이것을 실행하면서 맞닥뜨리는 상황들에 대해 어떻게 반응해야 하는지는 계획하지 않은 것처럼 보였다. 부스트가 정의되지 않으니 MLFQ 최종 구현은 계속 미뤄졌다.

어디가 잘못됐는지 하나하나 따져보던 중, 우리는 IO 잡이 수행되는 것을 매우 자세하게 그려보았다. 결국 우리는 수업시간에 잠깐 언급된 wait큐를 적용할 수 있었다. 이제껏 wait큐에 주목하지 못했다. wait상태의 프로세스가 달려있는 큐는 ready상태의 큐와 다르다는 것은 이미 알고 있었고 IO잡을 어떻게 처리해야 하는지 논의하면서 자주 등장했던 주제였기 때문에, 우리는 wait큐를 잘 알고 있다고 여겼다. 그래서 이 문제에서 wait큐가 열쇠가 될 것이라고는 굉장히 오랜 시간동안 생각할 수 없었다. 강의노트의 그림처럼 큐에서 프로세스를 실행하는 그림이 아닌 CPU를 그려서 확인하니 부스트에 대한 의문이 풀리기 시작했다. 〈아래 그림〉은 강의노트 20페이지의 일부를 직접 그려본 것이다. 부스트와 IO프로세스가 어떤 순서로 최상위 큐에 들어오게 되는지 그 과정을 확인했다.

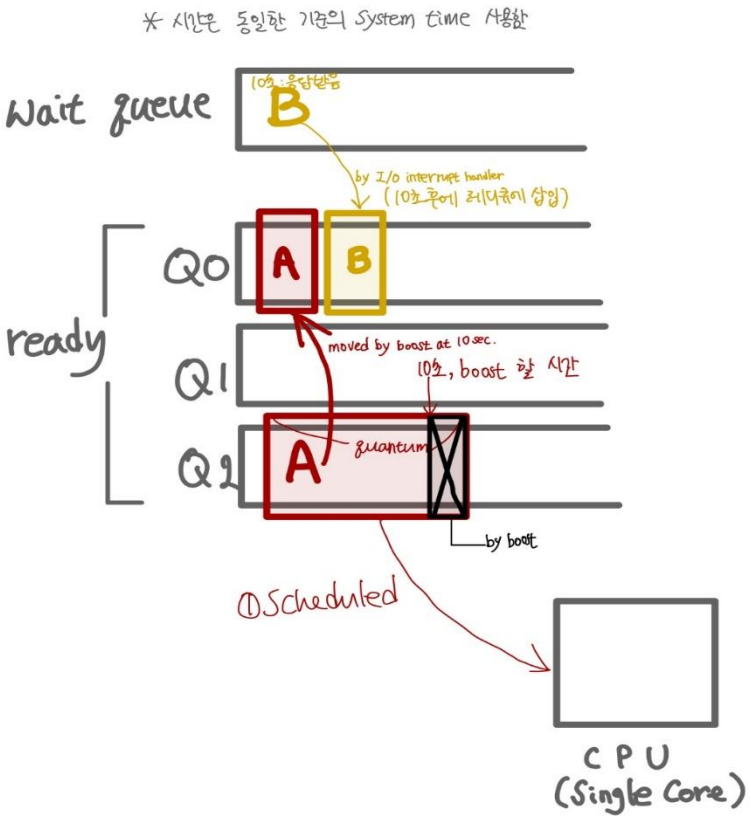


Figure 3 wait, ready 큐와 CPU

우리는 IO보다 부스트 프로세스가 먼저 스케줄링되는 이유를 부스트가 되는 시간과 IO이벤트가 발생하는 시간은 속성이 다르기 때문이라고 추정했다. 부스트는 주기적으로 이루어지기 때문에, 다음 부스트 시간을 예측할 수 있다. 하지만 IO이벤트는 언제 발생할지 예측할 수 없다. 즉, 부스트는 부스트 직전에 프로세스 노드를 최우선 큐로 이동시킬 준비를 미리 하여 정확한 시간에 부스트 할 수 있을 것이다. 반면 IO이벤트는 IO이벤트 발생 후 인터럽트 핸들링을 통해 wake될 것이다. 따라서 동일한 시간에 부스트와 인터럽트가 함께 발생한다면 우선순위 큐에 먼저 들어가는 것은 부스트된 프로세스가 될 것이다.

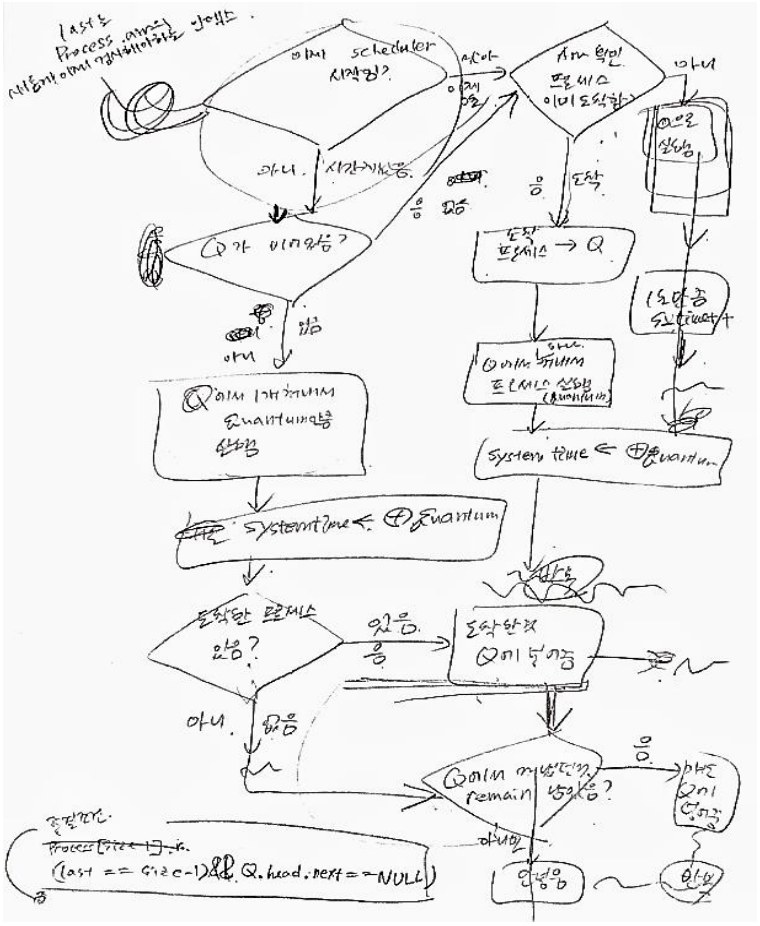
(3) 적절한 부스트 간격에 대한 고민

만약 부스트 주기가 실행중인 프로세스 개수에 비해 너무 짧다면, MLFQ는 가장 높은 우선순위 큐 하나만 사용하게 될 것이다. 라운드로빈과 MLFQ가 다른 양상을 보이게 하려면 적절한 부스트 간격을 설정해야 할 텐데, 이 값을 어떻게 정해야 할지 이야기를 나눴었다.

3) 코딩 과정

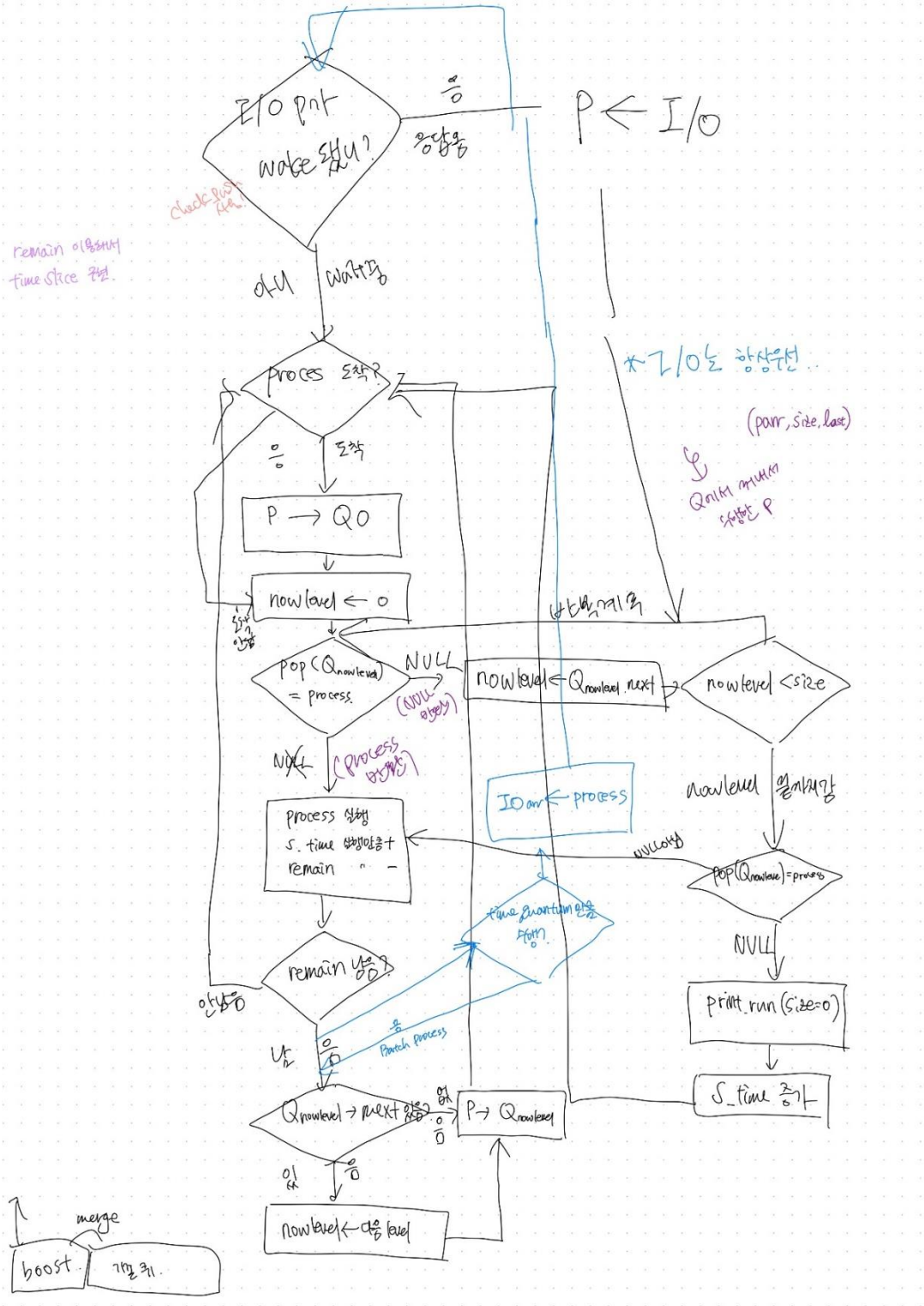
(1) 복잡한 알고리즘과 디버깅

우리는 라운드 로빈을 구현하기 위해 카페에서 함께 코딩하고 있었다. 우리 둘 다 라운드 로빈을 충분히 공부한 후였기 때문에 어떻게 구현할지에 대한 생각도 비슷할 것이라고 여겼다. 어떤 변수를 선언할지 정하고 반복문을 작성하기 시작했는데, 서로 하고 싶은 방향이 조금씩 달랐다. 그래서 이야기하며 종이에 정리해봤다. 아래는 라운드로빈 스케줄링을 구현하기 위해 작성한 순서도이다.



어떤 것부터 작성해야 하는지 합의를 보니 코딩은 금방 끝났다.

문제는 MLFQ였다. 규칙도 많고 큐도 여러 개였기 때문에 신경 써야 하는 것이 많아 보였다. 그래서 이번에는 함께 순서도부터 써보았다. 아래는 처음 작성한 MLFQ 순서도이다.

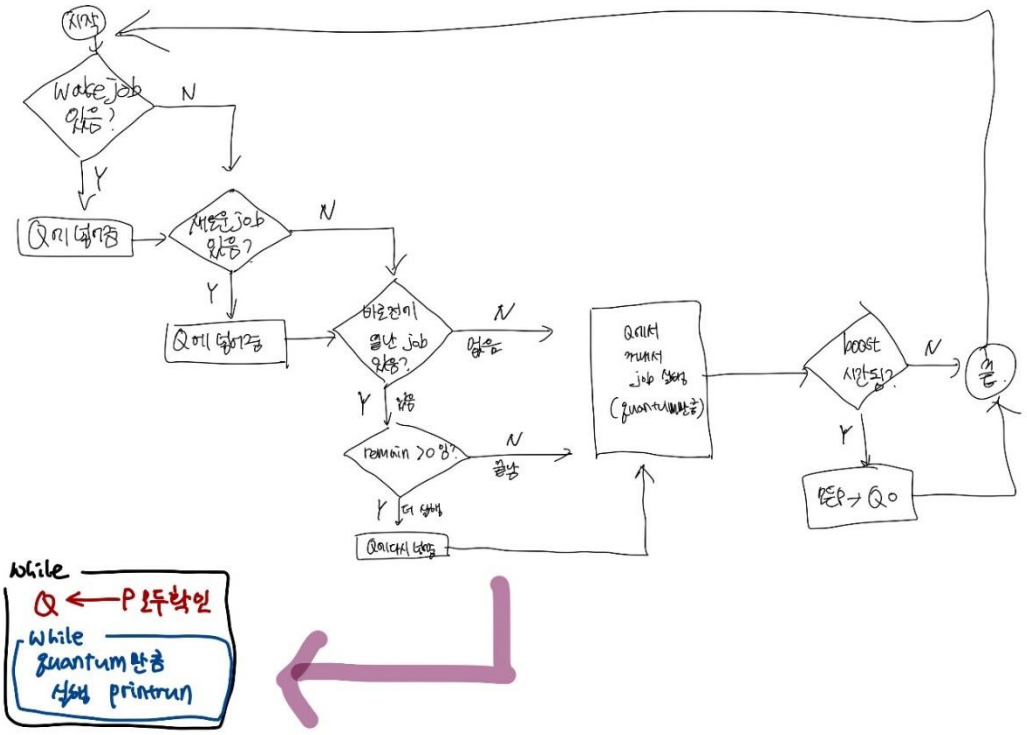


순서가 중복되는 것들이 많아 사용한 도형을 최대한 활용하면서 작성했다. 완성하고 보니 굉장히 복잡해졌다. 그래도 차근차근 코딩하면 금방 완성할 수 있다고 생각했다.

전혀 아니었다. 컴파일 하는 것부터 난제였다. 겨우겨우 컴파일하니, 무한루프를 돈다거나 s_time, last 같이 중요한 변수가 제대로 처리되지 않았다. 디버깅을 FCFS, 라운드로빈, 스트라이드 시뮬레이터를 만드는 것보다 시간을 배로 쓰고 있었다. 고작 한번 쓰이더라도 하나의 동작으로 취급할 수 있는 것을 함수로 묶어도 보고 구조체를 다시 만들어보고, 다시 만든 구조체 때문에 기존에 사용했던 함수도 고쳐봤지만 무리였다. 정상적인 결과가 도저히 나오지를 않았다.

그래서 며칠을 더 붙잡은 끝에 처음부터 다시 하기로 했다. 얻은 교훈은 두가지였다.

1: 비슷한 행동을 하는 것은 미리 함수로 묶어 둘 것.
2: 순서도를 그릴 때, 흐름은 대충 보아도 이해할 수 있어야 할 것
과정이 복잡한 만큼 모듈화를 잘 하는 것이 관건이라는 생각이 들었다. 그래서 순서도를 읽으면서 헷갈리지 않게, 정확면서 단순하게 작성하기로 했다. 아래는 새로 작성한 순서도이다.



비록 코딩할 때 생각해서 작성할 부분은 많아졌지만, 이번에는 몇 시간만에 구현해 낼 수 있었다.

(2) 조원간 코드 공유 방법

처음에는 카카오톡 채팅에 직접 붙여 넣어서 공유했다. 그런데 코드가 몇 백줄 넘어가니 잘려서 전송되었다. 그래서 c파일을 전송하고 받아서 수정했다. 카카오톡에서 c파일은 전송이 안되다 보니 텍스트 파일로 전송했다. 공유 과정이 번거롭다 보니 오류 수정한 파일을 바로 바로 올리지 않을 때도 있었다. 그래서 이미 수정된 에러를 다른 사람이 또 고치고 있는 일도 발생했다. 파일 이름도 규칙 없이 지어서 나중에는

어떤 파일이 어디까지 수정된 파일인지 확인하는 것이 불가능했다. 이러한 요인들이 전체 작업 시간을 더 늘린 것 같다.

4) 개선할 점

앞으로는 코딩하기 전에 사용해야 할 자료형과 함수를 미리 계획해야겠다. FCFS, RR, Stride, MLFQ가 한 프로그램에 들어있는데도 불구하고 하나 하나씩 구현하다 보니 나중에는 엉망이 되어 완성이 계속 미뤄졌다. 그리고 코딩을 본격적으로 하기에 앞서 설계를 꼼꼼히 해야겠다고 느꼈다. 시간을 많이 들인 MLFQ 구현을 할 때도 코딩에 앞서 순서도도 그리면서 체계적으로 생각하려고 했지만, 순서도에서 고려하지 못한 부분들이 코딩을 하면서 나타나서 계속 전체적으로 수정을 해야 했고, 수정을 하면서도 새로운 오류가 생겼다. 또한 소스코드를 잘 관리하기 위해서 적극적으로 깃을 사용해야겠다. 깃을 활용하여 협업한다면 개발 시간을 줄이고 편리하게 코딩할 수 있을 것이다.

5) 역할 분담

공동: 라운드로빈, MLFQ 순서도 작성, 디버깅

양혜은: FIFO, 스트라이드 및 관련 구조체, 함수 구현

정윤아: MLFQ 코딩, 메인 함수 작성, 큐, 프로세스 관련 구조체, 함수 구현