CPU Scheduling Simulator Implementation

과목	운영체제 (COSE341)
학과	건축사회환경공학부
학번	2020170393
이름	최준환

목차

I. 서론
    1. CPU Scheduler
    2. 구현사항 요약

II. 본론
    1. 선행연구 분석
    2. 시스템 계획
    3. 구현 결과
    4. 실행 및 성능 비교

III. 결론 및 고찰
    1. 시뮬레이터 기능 정리
    2. 개선사항
    3. 프로젝트 소감

IV. References

I. 서론

1. CPU Scheduler

1) 개념

운영체제의 역할 중 하나는 사용자의 프로세스를 효과적으로 처리하여 대기시간과 지연 등 사용자의 경험을 최적화하는 것이다. 프로세스는 실행을 위해 메인 메모리에 로드된 프로그램 파일로, 운영체제는 CPU 스케줄링 알고리즘에 따라 여러 개의 프로세스에 대한 CPU 할당을 결정한다.

프로세싱 유닛이 한 개뿐인 시스템이라면 같은 시간동안 최대 한 개의 프로세스만이 CPU 자원에 할당될 수 있다. 그러나 현실의 컴퓨터 환경에서는 사용자가 여러 개의 프로세스를 실행하기 원하기 때문에, 운영체제는 CPU 스케줄링을 통해 사람의 시점에서 매우 짧은 시간마다 CPU에 서로 다른 프로세스를 할당하는 방식으로 여러 개의 프로세스가 동시에 실행되는 것처럼 보이도록 한다. 따라서 사용자의 입장에서 프로세스들의 긴 대기시간이나 지연으로 인한 불편을 최소화하기 위해서는 적절한 방식의 CPU 스케줄링이 필요하다. 현존하는 수많은 운영체제는 각자의 스케줄링 알고리즘을 통해 CPU 스케줄링을 수행한다.

2) 기준

CPU 스케줄러를 통해 얻고자 하는 효과는 같은 시간동안 가능한 많은 프로세스의 실행을 완료하고, 각 프로세스의 총 수행 시간이나 대기 시간 등을 최적화하는 것이다. 이러한 관점에서 다음의 요소들을 스케줄링을 위한 기준으로 볼 수 있다.

(1) CPU utilization

CPU utilization은 일정 시간동안 CPU가 실제로 프로세스 실행에 사용된 비율을 의미한다. 즉 시스템이 측정한 시간동안 CPU가 busy 상태였던 시간의 비율이다. CPU utilization이 높으면 CPU자원이 거의 항상 사용되고 있다는 것이므로 프로세스 실행이 끊기지 않고 이루어지고 있을 뜻한다. 하지만 실제 시스템에서는 너무 높은 CPU utilization은 CPU에 걸리는 부하가 크다는 의미를 가지기도 하므로 적정 수준으로 관리된다. 본 프로젝트에서는 context switching에 의한 오버헤드를 시간 계산에 포함하지 않으므로 직접적인 성능 지표로는 활용하지 않았다. 하지만 특정 프로세스와 스케줄링 알고리즘 조합에서 불필요한 CPU의 idle이 발생해 총 스케줄링 시간이 다른 알고리즘보다 길어지는 상황은 확인할 수 있었다.

(2) Throughput

CPU 스케줄링에서 throughput은 단위시간동안 처리한 프로세스의 수를 의미한다.

(3) Turnaround time

Turnaround time은 프로세스의 요청이 발생한 순간부터 완료시까지의 시간 간격을 의미한다. 즉 프로세스가 생성되어 대기를 시작하면서부터 완전히 실행될 때까지 소요된 시간이다. Average turnaround time은 여러 개의 프로세스에 대한 turnaround time의 평균값으로서, 스케줄링 알고리즘에 따라 변한다. Average turnaorund time이 짧을수록 더 효율적인 스케줄링 알고리즘으로 생각할 수 있다.

(4) Waiting time

프로세스가 ready queue에서 실행을 대기한 시간의 총합이다. I/O 작업 완료 interrupt에 대한 대기의 경우 해당 시간동안 프로세스가 실행될 수 없으므로 포함하지 않는다. Average waiting time이 짧다면 프로세스들이 평균적으로 대기한 시간이 짧음을 뜻하므로 CPU 할당이 적절히 이루어지고 있음을 판단할 수 있다.

(5) Response time

Response time은 특정 프로세스가 최초로 실행 대기를 시작한 순간부터 최초로 CPU에 할당될 때까지 걸리는 시간을 의미한다. Average response time이 짧다면 요청된 프로세스가 CPU에 할당되기까지 필요한 시간이 짧음을 의미하므로 효율이 높을 것으로 예상할 수 있다.

결과적으로 CPU 스케줄러는 CPU 사용률과 시간 당 프로세스 처리율 등의 지표는 최대화하고, 프로세스 당 전체 소요시간이나 대기시간, 응답시간 등 프로세스 완료까지 필요한 시간은 최소화하는 방향으로 CPU에 프로세스를 할당해야 한다. 이를 위해 다양한 스케줄링 알고리즘들이 제안되었다.

3) 알고리즘

프로세스 스케줄링은 결국 현재 실행을 위해 CPU 할당을 기다리는 프로세스 중에서 일정한 기준을 통해 다음에 CPU에 할당할 프로세스를 결정하는 것이다. 즉 스케줄링 알고리즘은 어떤 기준으로 다음에 실행할 프로세스를 판단할 것인지 그 방법을 포함한다. 다음은 기초적인 CPU 스케줄링 알고리즘의 예시이다.

(1) First Come, First Served (FCFS)

FCFS 스케줄링은 문장 그대로 CPU를 요청한 순서대로 프로세스를 할당하는 스케줄링 방식이다. FCFS 방식에서 프로세스는 단순한 FIFO 큐를 사용해 관리할 수 있다. 이 스케줄링 방식에서는 preemption이 발생하지 않으므로, CPU에 할당된 프로세스가 종료되거나 스스로 interrupt가 발생하여 중단되기 전까지는 실행이 계속된다. 따라서 CPU burst time이 긴 프로세스가 다른 프로세스보다 먼저 큐에 들어오게 되면 보다 CPU burst time이 짧은 프로세스들까지 turnaround time이 길어지는 convoy effect가 발생한다.

(2) Shortest Job First (SJF)

SJF는 실행을 기다리는 프로세스들 중 남아있는 CPU burst time이 가장 짧은 프로세스를 CPU에 할당하는 스케줄링 방식이다. 따라서 shortest-next-CPU-burst algorithm으로 볼 수 있다. SJF에서는 nonpreemptive와 preemptive의 두 가지 scheme이 존재한다. 비선점형 (nonpreemptive) 스케줄링에서는 현재 실행중인 프로세스보다 더 짧은 CPU burst time을 가진 프로세스가 추가되더라도 현재 CPU 할당을 보장한다. 반면 선점형 (preemptive) 스케줄링에서는 언제든 현재 실행중인 프로세스보다 더 짧은 잔여 burst time을 가진 프로세스가 나타나면 해당 프로세스가 즉시 CPU에 할당된다. Preemptive SJF는 특별히 Shortest-remaining-time-first (SRTF) 스케줄링으로 칭하기도 한다.

Preemptive SJF는 이론적으로 가장 짧은 average turnaround time을 갖지만, 현실의 시

스템에서 실행될 프로세스의 CPU burst time을 예측하는 것은 어렵기 때문에 실제로 적용하기에는 무리가 있는 방법이다.

(3) Priority Scheduling

Priority scheduling은 각각의 프로세스가 우선순위(priority)를 부여받아 우선순위가 높은 프로세스부터 CPU에 할당되는 스케줄링 방식이다. 이 방식 역시 선점형 scheme이 존재한다. 비선점형 방식의 경우 더 높은 우선순위를 가진 프로세스가 새로 추가되더라도 실행중인 프로세스는 CPU 할당을 보장받는다. 반면 선점형 방식에서는 더 높은 우선순위를 갖는 프로세스가 추가될 때마다 즉시 해당 CPU 할당이 해당 프로세스로 변경된다.

프로세스 각각에 우선순위를 부여하면 더 빠른 수행이 필요한 중요 프로세스를 우선적으로 수행 가능하다는 장점이 존재한다. 반면 우선순위가 고정적인 시스템에서 우선순위가 높은 프로세스가 계속 추가되면 우선순위가 낮은 프로세스는 영원히 실행되지 못하는 starvation 문제가 발생할 가능성이 존재한다. 따라서 이러한 경우 ready queue에서 대기하는 시간에 따라 우선순위가 점차 높아지는 aging 기법을 사용하여 무한히 대기하는 프로세스가 발생하지 않도록 하는 것이 권장된다.

(4) Round Robin

Round Robin 방식의 스케줄링에서는 실행 대기 중인 모든 프로세스가 돌아가면서 지정된 time quantum동안 CPU에 할당된다. 이 때 할당 순서는 FCFS 방식과 동일하게 FIFO 구조로 결정된다. 즉 Round Robin 방식에서 time quantum의 길이가 매우 길어지면 FCFS와 동일하게 동작하게 되고, convoy effect 또한 발생할 수 있다. 반면 반대로 time quantum이 너무 짧아지면 context switching에 의한 오버헤드가 지나치게 커질 수 있다. 따라서 Round Robin 적용 시에는 time quantum을 적절한 길이로 설정하여 모든 프로세스가 공평하게 CPU에 할당하는 장점을 얻으면서 대기시간이 과도하게 길어지는 문제를 피해야 한다.

(5) Multilevel Queue Scheduling

Multilevel Queue Scheduling (MQ Scheduling)은 프로세스가 CPU 할당을 대기하는 ready queue를 여러 층으로 만들어 관리하는 방식이다. 즉 그 자체로 CPU에 할당할 프로세스를 결정하는 알고리즘을 의미하지는 않고, 프로세스를 관리하는 방법으로 볼 수 있다.

MQ scheduling의 한 예로, priority scheduling에서 우선순위에 따라 큐를 분리하면 priority scheduling을 단순한 FIFO 큐 여러 개로 관리할 수 있는 장점이 있다. 또한 기존에 큐를 1개로 하였을 때 탐색에 최대 O(n)이 필요했던 것과 달리 큐를 계층 형태로 관리하면 O(nlgn) 시간 내에 원하는 프로세스를 찾을 수 있다.

MQ scheduling은 프로세스의 종류에 따라 서로 다른 스케줄링 알고리즘을 사용하는 여러 개의 레디큐를 관리하는 것으로 볼 수도 있다. 예를 들어 foreground 프로세스에 Round Robin 방식을, background 프로세스에 FCFS 방식을 적용하기 위해 두 개의 분리된 큐를 사용할 수 있다. 이때는 큐 간에도 우선순위가 존재하거나 CPU 할당 시간의 비율을 달리하는 방법으로 프로세스를 선택할 큐를 선택하는 스케줄링 과정도 수행되어야 한다.

Multilevel Queue에서 우선순위가 낮은 큐에 프로세스가 진입하면 starvation이 발생할 수 있다. 이를 해결하기 위해 CPU 사용시간이나 대기시간에 따라 프로세스를 우선순위가 더 낮거나 높은 큐로 이동하는 Multilevel Feedback Queue Scheduling도 존재한다.

2. 구현사항 요약

본 프로젝트에서 구현한 CPU 스케줄러의 실행 흐름을 요약하면 다음과 같다.

1) 입력

cpu_scheduler 최초 실행 시 0 또는 1을 입력하여 전체 알고리즘 또는 특정 알고리즘에 대한 스케줄링을 수행한다. 단일 알고리즘에 대한 시뮬레이션 시 실행할 알고리즘의 번호를 추가로 입력한다 (0~8).

다음으로 생성할 프로세스의 수를 1~10 중의 값으로 입력한다. 이전 입력 결과에 따라 시뮬레이션에 time quantum이 필요한 알고리즘이 있다면 1~5 중의 값을 입력하여 지정한다.

2) 프로세스 생성

필요한 값을 모두 올바르게 입력하면 CreateProcess()가 호출되어 지정한 개수만큼의 프로세스에 대한 값을 초기화한다. 초기화가 이루어지는 값은 PID, 도착시간, 우선순위, CPU burst time, I/O 요청 발생 시점 및 작업시간이 있다. PID를 제외한 값은 모두 랜덤하게 결정된다.

3) 시뮬레이션 환경 설정

프로세스 생성이 완료되면 Schedule()이 호출되어 시뮬레이션 환경을 구성한다. 이 함수는 프로세스와 스케줄링 알고리즘을 전달받아 해당 알고리즘으로 프로세스를 스케줄링한 결과를 출력한다. 새로운 시뮬레이션을 위해 프로세스의 고유한 값을 제외한 스케줄링 데이터를 초기화하고, 상태 추적을 위한 큐와 포인터를 초기화한다.

초기 입력 과정에서 모든 스케줄링 알고리즘에 대한 시뮬레이션을 선택한 경우 위 과정을 스케줄링 알고리즘이 변경될 때마다 반복한다.

4) 스케줄링

시뮬레이션 환경이 초기화되면 반복문을 통해 시간 단위로 프로세스의 상태를 추적하며 모든 프로세스의 종료를 판단할 때까지 스케줄링을 수행한다. 본 프로젝트에서 구현한 스케줄링 알고리즘에는 FCFS, 선점/비선점형 SJF, 선점/비선점형 Priority scheduling, Round Robin, Lottery Scheduling 및 선점/비선점형 Longest I/O Burst First 방식이 있다.

5) 출력

전체 프로그램에서 출력이 이루어지는 정보는 세 종류가 있다. 첫 번째는 프로세스 생성 완료 시 출력되는 프로세스들의 데이터이다. 두 번째는 매 시뮬레이션 종료 시 출력되는 스케줄링 결과를 나타낸 Gantt chart이다. 마지막으로 매 스케줄링 결과 출력 후 함께 출력되는 성능 평가 결과가 있다. 성능 평가 지표에는 waiting time, turnaround time, response time이 있다.

II. 본론

1. 선행연구 분석

직접 CPU scheduling simulator의 시스템을 구성하기 전에 시뮬레이터에 필요한 조건을 이해하기 위해 Github에 존재하는 CPU 스케줄러 오픈소스를 탐색하여 각 구현상의 특징을 분석해보았다.

Reference [1]

최초 실행 시 생성할 프로세스 개수를 입력받아 해당하는 개수의 프로세스 구조체 포인터를 동적으로 할당하였다. 프로세스 데이터는 랜덤으로 발생한 값을 사용하였다. 이 때, priority 값에 대한 중복을 허용하지 않는 로직이 존재하여 모든 프로세스가 고유한 priority 값을 가지도록 하였다.

특정 스케줄링 알고리즘을 사용한 스케줄링 시뮬레이션 시 먼저 config()를 호출하여 큐를 초기화하고, initialize()를 호출하여 프로세스의 스케줄링 데이터를 초기화하였다. Waiting queue는 최대 프로세스 수 길이의 int 배열로 구현하였다.

각각의 스케줄링 알고리즘은 해당 알고리즘의 명칭을 이름으로 갖는 함수로 구현되었다. 따라서 각각의 스케줄링 함수 내부에 시간 단위 반복문 각각 존재하였다. 반복문 종료는 check라는 값을 정의하여 check값이 프로세스 개수 p_num과 같아질 때 이루어지도록 하였다.

스케줄링 결과 (Gantt chart) 출력은 시뮬레이션 중 실행중인 프로세스의 상태가 변할 때마다 발생 시각과 프로세스의 상태 변경을 출력하는 형태로 구현되었다. 성능 평가의 경우 스케줄링 종료 후 계산이 이루어졌는데, 여기서는 waiting time을 각 프로세스에 대해 직접 추적하여 계산하는 대신 turnaround time - CPU burst time으로 산출하였다. 따라서 I/O 작업 대기시간이 waiting time에 포함되었을 것이다. 소요시간과 대기시간 등 외에 CPU utilization을 계산하여 출력하였다. 이 값은 스케줄링 중 CPU가 idle이 될 때마다 idle time 값을 1만큼 증가시키는 방법으로 계산되었다.

Reference [2]

이 프로그램에서는 터미널에서 프로그램을 실행할 때 전달한 2개의 인자를 프로세스 생성에 사용하였다. 첫 번째 값은 생성할 총 프로세스의 개수를, 두 번째 값은 I/O 요청이 발생하는 프로세스의 수를 의미한다.

프로세스 구조체 배열의 이름을 jobQueue로 하였는데, 실제로 프로세스가 다른 큐에 추가될 때 jobQueue에 변경이 일어나지는 않고, 추상적인 형태를 명시적으로 나타내기 위한 것으로 보인다. 정의한 구조체 중 프로세스 정보에 대한 구조체 이외에 스케줄링 결과를 저장해두는 evaluation 구조체가 존재하였다. 구조체 멤버에는 적용된 알고리즘, 수행시간 및 성능 평가 값이 있다. 각각의 스케줄링 알고리즘에 대한 스케줄링이 순차적으로 수행되기 때문에 전체 프로그램 종료 전에 각 알고리즘에 대한 성능 평가 결과를 일괄적으로 보여주기 위해 값을 저장해두는 것으로 보인다.

큐 초기화, 큐 정렬, 큐에서 프로세스에 대한 접근 등의 조작이 함수로 구현되어 있으나, 모든 큐 구조에 대해 동일한 기능을 가지도록 구현되지는 않았고, 함수 정의 자체에 조작할 큐의 종류가 포함되어 있는 형태였다.

이 구현에서 스케줄링 알고리즘의 이름을 가진 함수(FCFS, SJF 등)의 기능은 알고리즘에 따라 프로세스 구조체 포인터 runningProcess가 가리키는 프로세스를 판단한 뒤 반환하는 것이다. 실제 시간 단위 스케줄링은 startSimulation() 함수에서 simulate()를 반복 호출하여 이루어지며, simulate()는 시간 간격 1에 대한 결과를 schedule()을 통해 받아 해당 시간동안 프로세스의 값을 업데이트한다. schedule()은 실제로는 실행할 알고리즘을 선택하여 결과를 반환하는 기능만 수행하며, runningProcess 판단은 FCFS_alg() 등 스케줄링 알고리즘 함수에서 이루어진다.

여기서도 Gantt chart는 상태 변경이 일어날 때마다 스케줄링 중간에 출력하는 형식으로 구현하였고, 성능 평가의 경우 각 스케줄링 단계에서 analyze()함수에 의해 계산한 뒤 evaluation()에서 일괄적으로 출력하는 형태였다.

Reference [3]

프로그램 실행 시 사용자가 입력하는 값은 프로세스의 개수와 time quantum의 2가지이다. 입력 과정 후 최초 1회 createProcess() 함수를 호출한 뒤, 각각의 스케줄링 알고리즘에 따른 스케줄링을 반복하였다.

```
int main(void) {
    ...
    void (*scheduler[])(t_process process[], int processNum, int timeQuantum, int chart[]) =
        {FCFS, nonPreemptiveSJF, preemptiveSJF, nonPreemptivePriority, preemptivePriority, roundRobin};
    ...
    for (int i = 0; i < SCHEDULER_NUM; i++) {
        ...
        scheduler[i](process, processNum, timeQuantum, chart);
        ...
    }
}
```

구현 상의 특징으로, main함수 내에서 함수 포인터를 활용하여 스케줄링 함수들에 대한 포인터 배열을 만들었다. 이후 for loop를 사용하여 미리 초기화한 순서대로 스케줄링 알고리즘 함수를 호출하였다. 때문에 모든 스케줄링 함수가 동일한 인자 구성을 가지도록 구현되었고, (process, processNum, timeQuantum, chart) 스케줄링 함수 호출 시 선점형과 비선점형을 구분할 수 없으므로 두 스케줄링 알고리즘은 별개의 함수로 구현되었다.

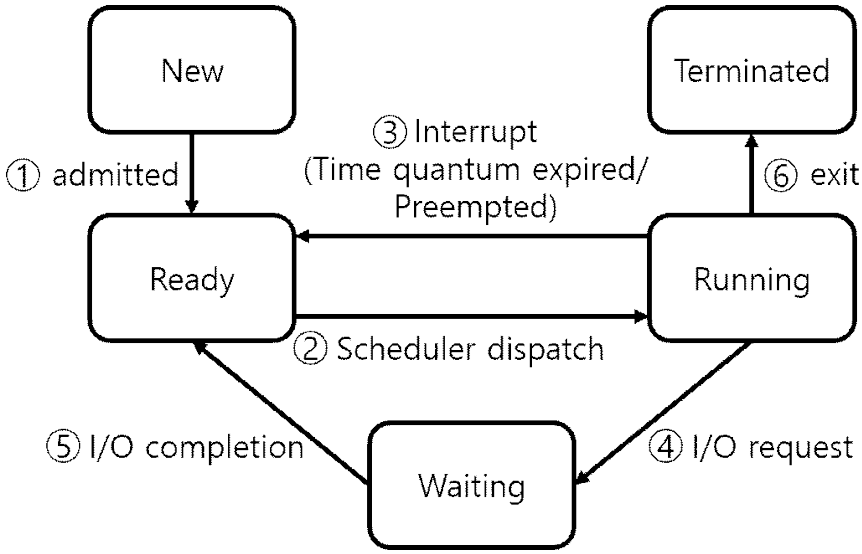
각 스케줄링 함수 내부에 while loop가 존재하여 모든 프로세스 종료 전까지 반복된다. 반복 조건 판단에는 isAllFinished() 함수를 프로세스의 .cpu_remaining이 모두 0에 도달했을 때만 1을 반환하도록 정의하여 !isAllFinished() 조건에서 반복하였다. Ready queue와 waiting queue는 모두 최대 프로세스 개수 +1 크기의 int 배열로 정의하였고, 실제 동작은 원형 큐로 동작하여 최대 프로세스 개수만큼의 PID값이 저장될 수 있는 형태로 구현하였다.

알고리즘 중 프로세스 데이터의 특정 값(priority, remaining time 등)이 우선순위 판단의 기준이 되는 경우 매 루프마다 ready queue를 해당 기준에 따라 정렬하였다. 정렬 기준에 따라 SortReadyQueueBy~ 형태의 함수를 정의하여 사용하였다.

이전 사례와 다르게 스케줄링 결과로서 간트차트를 별도로 출력하였다. 스케줄링 중 chart 정수 배열에 대하여 시간과 인덱스를 대응하여 실행중인 프로세스의 PID를 기록한 뒤, 스케줄링이 완료된 이후에 일괄 출력하였다. 스케줄링 중에는 최종 차트 길이를 알 수 없으므로 chart는 동적배열이 아닌 길이 300의 고정 배열로 생성되었다.

2. 시스템 계획

1) 프로세스의 생명주기



위 그림은 프로세스가 가질 수 있는 상태와 가능한 상태 변경을 그래프의 형태로 나타낸 것이다. 프로세스는 생성 후 실행 가능한 상태가 되면 ready 상태에서 CPU할당을 대기하고, CPU할당 시 running 상태가 되어 실행이 이루어지다가, I/O request가 발생하면 waiting 상태로 변경되어 작업 완료를 대기한다. I/O 작업 완료 interrupt가 발생하면 프로세스는 다시 ready 상태로 돌아가 CPU 할당을 대기한다. 스케줄링 scheme에 따라 실행중인 프로세스는 주어진 time quantum의 만료 또는 다른 프로세스의 선점(preemption)에 의해 즉시 ready 상태로 돌아올 수 있다. 실행이 완료된 프로세스는 terminated 상태가 되고 종료된다.

CPU 스케줄링 시뮬레이터 구성 시 명시적으로 추적되어야 하는 상태는 ready, running, waiting이다. 이 세 가지 state에서는 프로세스가 매 순간 서로 다른 상태로 이동할 수 있기 때문이다. 반면 new와 terminated의 경우 swap out이 발생하는 medium-term scheduling을 고려하지 않는다면 스케줄링의 관점에서 스케줄링 대상의 '추가' 또는 '감소'의 의미로만 생각해도 충분하다. 따라서 두 상태의 프로세스는 포인터로 상태를 명시하는 대신 스케줄링 데이터 값으로부터 상태를 판단하는 형태로 구현할 수 있다.

2) 프로세스 상태 정의 및 시스템 설계

본 프로젝트에서는 ready 상태의 프로세스는 ready_queue를, running 상태의 프로세스는 *running 포인터를, waiting 상태의 프로세스는 wait_queue를 사용하여 관리하였다. 반면 new와 terminated 상태의 프로세스는 각각 (current_time == arrival_time) 과 (process.completion_time != -1) 라는 조건의 형태로만 존재한다. 모든 프로세스 종료 시 스케줄링도 종료되어야 하므로 IsAllTerminated() 함수를 정의하여 모든 프로세스가 상술한 조건을 만족하는지 확인하는 것으로 스케줄링 종료 여부를 판단하였다.

프로세스에서 발생할 수 있는 상태 변경에는 총 6가지가 있다. 다음은 위에서 정의한 각 상태의 프로세스 추적 수단을 바탕으로 각각의 상태 변화를 판단하는 방법을 수도코드로 표현한 것이다.

① admitted

```
for (i = 0부터 마지막 프로세스의 pid까지) {
  if (process[i].arrival_time == current_time) {
    ready_queue에 process[i]를 가리키는 노드 추가
  }
}
```

② Scheduler dispatch

스케줄링 알고리즘에 따라 판단이 이루어진다. 각각의 스케줄링 알고리즘은 매 시간 단위마다 호출되어 실행 중인 프로세스의 중단 여부를 판단한 뒤 ready queue에서 CPU에 할당할 프로세스를 선택하는 함수로 구현하였다. 알고리즘별 상세한 판단 논리는 이후 구현 단계에서 함수별로 수도코드를 사용하여 서술하였다.

```
if ((running == NULL && ready_queue->front != NULL) {
  ready queue의 '선택된' 노드가 가리키는 프로세스를 running이 가리킴
}
```

③ Interrupt (time quantum expired / preempted)

```
// time quantum expiry
if ((running != NULL) and (연속으로 실행된 시간 > time_quantum)) {
  ready_queue에 *running이 가리키는 프로세스를 가리키는 노드 추가
  running = NULL
}
```

Time quantum 만료는 시뮬레이션이 이루어지는 함수 내에서 occupancy_time이라는 값을 정의하여 현재 실행중인 프로세스가 연속으로 실행된 시간을 계산하여 판단한다.

```
// preempted
if (preemptive and (ready_queue->front의 우선순위 > running의 우선순위)) {
  ready_queue에 *running이 가리키는 프로세스를 가리키는 노드 추가
  running = NULL
}
```

스케줄링 방식에 따라 preemption이 발생하며, 이 과정은 preemptive scheme이 존재하는 스케줄링 알고리즘 함수 내부에서 새로운 프로세스를 CPU에 할당하기 이전에 먼저 수행된다. 프로세스의 CPU 할당은 CPU에 할당된 프로세스가 중단된 이후에 이루어져야하기 때문이다.

④ I/O request

```
// running 프로세스에 대하여
for (i = 0부터 MAX_IO_NUM-1 까지) {
  if ((running의) io_timing[i] == CPU burst 진행도) {
    running의 I/O 작업 남은시간을 io_burst[i]로 설정
    running의 total_remaining_io를 io_burst[i]만큼 감소
    wait_queue에 *running이 가리키는 프로세스를 가리키는 노드 추가
    running = NULL
  }
}
```

I/O request 요청 시점은 프로세스 생성 시 io_timing[]에 저장된다. 이 배열의 전체 길이는 MAX_IO_NUM이며, 이 중 일부만 랜덤 과정을 통해 유효한 값으로 초기화되고 나머지는 -1로 초기화하여 구분한다. 실제 구현에서는 프로세스 구조체 멤버에 진행도 값 대신 cpu_burst와 remaining_time만이 존재하여, (cpu_burst-remaining_time)이 io_timing의 특정 값에 도달하였는지 판단하는 것으로 I/O 요청을 정의하였다.

⑤ I/O completion

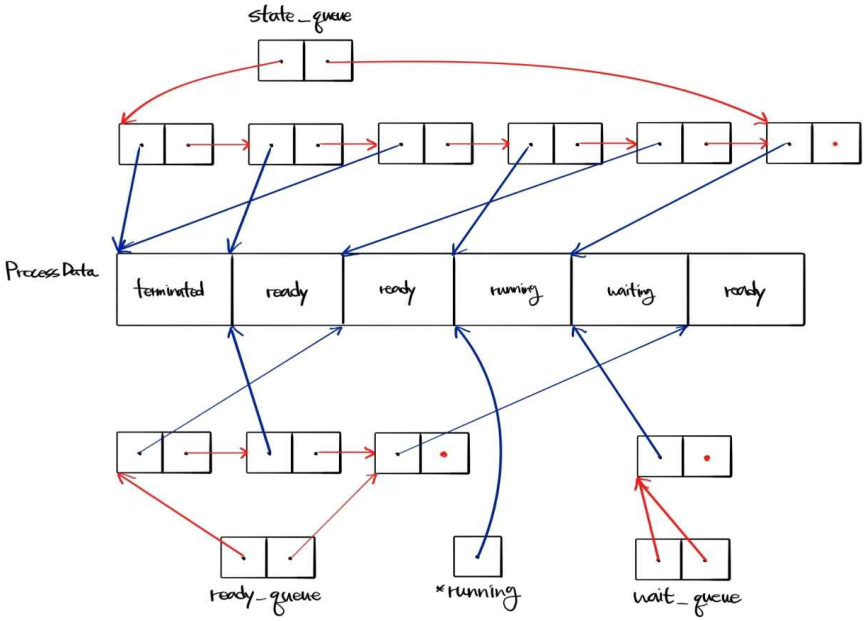
```
for (wait_queue의 노드들) {
  if (노드가 가리키는 프로세스의 remaining_io == 0) {
    ready_queue에 현재 노드의 프로세스를 가리키는 노드를 추가하고 기존 노드는 제거
  } else{
    임시 큐에 현재 노드의 프로세스를 가리키는 노드를 추가하고 기존 노드는 제거
  }
}
wait_queue = 임시 큐
```

Waiting queue등 모든 큐는 Node 구조체의 연결 리스트로 구현하였다. 그러나 waiting queue의 경우 노드의 추가 순서와 제거 순서가 상이하기 때문에 큐 형태로 관리하는 대신 매 시간마다 기존의 큐를 순회하면서 아직 대기 중인 프로세스는 새로운 waiting queue에 추가하고, I/O 작업이 완료된 프로세스는 ready queue에 추가하는 형태로 구현하였다.

⑥ exit

```
if (running != NULL and running->remaining_time == 0) {
  running->completion_time = current_time // 완료시각을 현재시각으로 설정
  running = NULL
}
```

종료된 프로세스는 (completion_time!=-1) 조건으로 판단하고, 실행이 완료되어 종료될 프로세스의 판단에는 (remaining_time==0) 을 사용하였다. 종료와 함께 completion_time은 시뮬레이션 상의 해당 시각으로 변경된다.



프로세스의 상태는 두 개의 큐와 한 개의 포인터로 추적할 수 있지만, 본 프로젝트에서는 스케줄링 종료 후 해당 시뮬레이션의 결과를 간트 차트의 형태로 출력하고자 하였기에 상태 변경 내용과 시점(순서)를 저장하기 위한 추가 변수를 정의하였다. state_queue는 current_time에서 current_time+1이 될 때 CPU 할당이 변할 경우 노드를 추가하고 current_time에 CPU에 할당되어 있던 프로세스를 가리킨 후 (CPU가 idle이었다면 NULL) current_time 및 상태 변경 종류 (char type)을 각각 Node 구조체의 time과 data에 저장한다. 위 그림은 이상의 세 가지 큐와 running 포인터가 Schedule()함수 내에서 유지되는 상황을 실제 코드 구현에 가깝게 표현한 예이다.

2) 선행연구 기반 구현요소 점검

선행 프로젝트의 시스템 구성과 구현 방법을 분석한 결과, 본 프로젝트의 목표에 부합하는 스케줄링 시뮬레이터를 구현하기 위해서는 다음 부분에 대한 보완이 필요하다고 판단하였다.

1. 프로세스 진행 중 I/O request가 다수(2회 이상) 발생하는 상황을 구현
2. 전체 스케줄링 결과를 일정한 형태의 간트차트로 출력하여 비교 용이성 향상
3. 각 스케줄링 알고리즘 구현 시 공통된 기능을 함수로 정의하여 사용
   3.1. Scheudle() 함수를 하나로 두고 다음 CPU할당 논리만 분리된 함수로 구현
   3.2. Ready queue 정렬 시 정렬 기준을 함수 인자로 전달하여 함수 재사용
4. 프로젝트를 기능별로 모듈화

위 요소를 반영하여, 각각 0~10의 랜덤한 시각에 도착하여 1~10의 CPU burst time과 priority 값을 갖고, 0~2회의 시점에 1~3 길이의 대기를 유발하는 I/O request가 발생하는 프로세스를 동시에 최대 10개까지 서로 다른 9가지 스케줄링 알고리즘으로 시뮬레이션 할 수 있는 프로그램을 실제로 구현하였다.

3. 구현 결과

1) 모듈 구성

```c
/* functions */
/* process.c    프로세스 생성 및 초기화 함수*/
void CreateProcess(ProcessData p[], int pNum);
void AssignProcessValues(ProcessData *p, int i);
void shuffle(int *arr, int n);
int compare(const void* a, const void* b);
void InitRuntimeData(ProcessData p[], int pNum);
int IsAllTerminated(ProcessData p[], int pNum);

/* simulator.c  시뮬레이션 환경 초기화 및 시간 단위 루프 반복 관련 함수 */
void Config(ProcessData p[], int pNum, int qNum, ...);
void Schedule(int alg_id, ProcessData p[], int pNum, int tq);

/* queue.c      큐 생성 및 조작 관련 함수 */
void InitQueue(Queue *q);
void Enqueue(Queue *q, ProcessData *p);
ProcessData* Dequeue(Queue *q);
int CompareProcess(ProcessData *a, ProcessData *b, int criteria);
void SortReadyQueue(Queue *q, int criteria);

/* scheduler.c  스케줄링 알고리즘 함수 */
void FCFS(Queue *ready, ProcessData **running, int current_time);
void SJF(Queue *ready, ProcessData **running, int current_time,
    int preemptive, Queue *state);
void Priority(Queue *ready, ProcessData **running, int current_time,
    int preemptive, Queue *state);
void RoundRobin(Queue *ready, ProcessData **running, int current_time,
    int time_quantum, int *occupancy_time, Queue *state);
void LotteryScheduling(Queue *ready, ProcessData **running, int current_time,
    int time_quantum, int *occupancy_time, Queue *state);
void LongestIOFirst(Queue *ready, ProcessData **running, int current_time,
    int preemptive, Queue *state);

/* display.c    프로세스 생성 결과, 스케줄링 결과(간트 차트 및 성능) 등 출력 관련 함수 */
void DisplayProcessInfo(ProcessData p[], int pNum);
void PrintGanttChart(ProcessData p[], int pNum, Queue *state_queue);
void Evaluation(int alg_id, ProcessData p[], int pNum);
```

구현된 CPU scheduling simulator는 역할에 따라 main.c를 포함한 6개의 소스코드 파일에 함수를 정의하였으며, 각 파일에 정의된 함수들의 대략적인 쓰임은 위 그림과 같다. main 함수는 서론에서 요약한 구현사항과 동일하게 프로그램이 동작할 수 있도록 조건에 따라 함수를 호출한 뒤 종료된다.

(1) 프로세스 생성: CreateProcess()

프로세스의 데이터와 스케줄링 정보를 저장하는 ProcessData 구조체의 구성 멤버는 아래 그림과 같다. main에서 초기 입력이 성공적으로 수행되면 프로세스 생성 단계로 넘어가 주어진 조건에서 랜덤으로 프로세스에 데이터가 부여된다.

```c
typedef struct {
    // Process data (1번만 초기화)
    int pid;
    int arrival_time;
    int priority;
    int cpu_burst;
    int io_timing[MAX_IO_NUM];
    int io_burst[MAX_IO_NUM];
    // Scheduling data (시뮬레이션마다 초기화)
    int remaining_time;
    int remaining_io;
    int waiting_time;
    int response_time;
    int completion_time;
    int total_remaining_io;
} ProcessData;
```

void CreateProcess(ProcessData p[], int pNum);

프로세스 구조체 배열과 값을 부여할 프로세스의 개수를 전달받아 각각의 프로세스에 랜덤한 데이터를 부여한 후 결과를 출력한 뒤 종료되는 함수이다.

void AssignProcessValues(ProcessData *p, int i);

개별 프로세스에 대한 실제 랜덤 데이터 부여가 이루어지는 함수이다. 시드 생성은 CreateProcess()에서 1차례만 이루어지고, 시드값을 사용하여 rand()의 나머지 연산을 활용해 랜덤한 값으로 전체 시뮬레이션에서 사용되는 프로세스 데이터를 초기화한다.

I/O request 발생 시점과 작업시간의 경우 전처리로 정의된 MAX_IO_NUM 길이의 정수 배열 io_timing[]과 io_burst[] 전체를 -1로 초기화한 뒤 index 0부터 랜덤한 부분까지만 유효한 값으로 다시 부여하여 정의한다. 결과적으로 프로세스가 CPU에 io_timing[0] 만큼 할당된 이후 첫 번째 I/O 요청이 발생하면 프로세스는 io_burst[0] 시간동안 작업 완료를 대기하는 형태가 된다.

void InitRuntimeData(ProcessData p[], int pNum);

초기에 1회만 설정되고 모든 스케줄링 알고리즘에서 동일하게 사용되는 프로세스 정보와 달리, 스케줄링 정보는 다른 알고리즘을 사용한 시뮬레이션 시 항상 초기화되어야 한다. 따라서 ProcessData 구조체의 scheduling data들을 알맞은 값으로 초기화한다. remaining_time은 초기에 cpu_burst와 동일하게, waiting_time은 0으로 초기화되고 특정 조건의 판단에 쓰이는 값들은 -1로 초기화한다. total_remaining_io는 앞서 설정한 io_burst[]에서 유효한 값을 모두 더한 값으로 초기화한다.

int IsAllTerminated(ProcessData p[], int pNum);

프로세스 배열과 개수를 입력받아 지정된 프로세스 개수에 대하여 어떤 프로세스라도 terminated 상태인 (completion_time != -1)을 만족하지 않을 경우 0을 반환하고, 모두 종료 판단 시 1을 반환한다. 시뮬레이션 시 반복문의 종료 조건으로 사용한다.

(2) 큐 조작: SortReadyQueue() 등

큐에 추가하고 제거하는 기능을 구현하여 간결성을 높이기 위한 함수들을 정의하였다. 이 중 SortReadyQueue() 함수는 스케줄링 알고리즘에서 다음에 CPU에 할당할 프로세스를 판단하는 데에도 중요한 역할을 한다. 이 함수의 자세한 동작은 스케줄링 알고리즘의 수도코드와 함께 후술하였다.

(3) 시뮬레이션 환경 구성: Schedule()

void Config(ProcessData p[], int pNum, int qNum, ...);

InitRuntimeData()와 InitQueue()를 호출하여 스케줄링 정보와 이전 시뮬레이션 종료 후에 남았을 수 있는 큐 연결을 초기화한다. 가변 인자 함수로 구현하여 프로세스와 프로세스 개수를 입력한 뒤 나머지 인자로 전달받은 만큼의 큐를 전부 초기화한다.

void Schedule(int alg_id, ProcessData p[], int pNum, int tq);

Schedule() 함수는 ready_queue, wait_queue, state_queue, *running이 정의되고 이상의 초기화 과정을 거친 뒤 반복문을 사용하여 시간 단위로 프로세스의 값과 상태를 업데이트하는 실제 시뮬레이션 환경을 구성한다. Schedule() 전체의 구성은 세 부분으로 나누어진다.

1. 큐, 포인터, 지역변수(time quantum, idle 추적용) 정의 및 초기화
2. 스케줄링 시뮬레이션 (모든 프로세스 종료 시까지 반복)
3. 스케줄링 결과를 간트차트로 출력 (PrintGanttChart())

그리고 시뮬레이션이 이루어지는 반복문은 하나의 시간 간격 current_time 동안 다음 작업이 순차적으로 수행된다.

1. current_time이 되었을 때의 값 업데이트
  waiting: I/O 완료 대기시간 1 감소
  running: 남은 CPU burst 1 감소
  ready: waiting time 1 증가

2-1. current_time에 진입한 프로세스의 state 결정
  new: arrival_time == current_time인 프로세스를 레디큐에 추가

2-2. current_time + 1에서 프로세스의 state 결정
  waiting: I/O 작업 완료 시 레디큐로 복귀
  running: CPU burst 완료 시 프로세스 종료, I/O 요청 발생 시 waiting queue 이동
  running, ready: time quantum 초과 또는 preemption 여부 판단
  ready: 실행중인 프로세스가 없는 경우 CPU에 할당할 프로세스 결정후 running 변경

3. 상태 기록: response time과 idle 정보 추적 및 업데이트

이를 수도코드로 다시 표현하면 다음과 같다. 여기서는 Preemption은 포함하지 않았다.

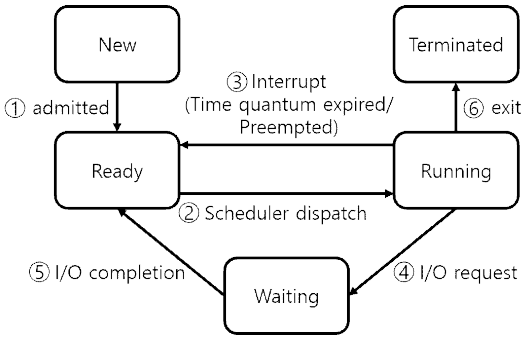
```
Queue ready queue, wait queue
Process *running
큐와 프로세스 포인터 초기화
int current = 0

while (!모든 프로세스 종료)
// 프로세스 상태에 따라 값 업데이트
  // 1. running pointer
  running 프로세스의 진행률, 점유시간 ++
  // 2. waiting queue
  waiting queue 내 프로세스의 I/O 작업 진행률 ++
  // 3. ready queue
  ready queue 내 프로세스의 대기시간 ++

// 업데이트 된 프로세스 값을 사용하여 상태 업데이트
  // 1. job scheduling
  current에 도착한 프로세스를 ready queue에 추가
  // 2. running pointer
  if (running 프로세스 종료) {
    running 프로세스에 완료시간 기록
  } else if (running 프로세스 I/O request 도착) {
    running 프로세스 중단 및 waiting queue에 추가
  }
  // 3. waiting queue
  for (wait queue 프로세스) {
    if (프로세스의 I/O 작업 완료) {
      프로세스를 wait queue에서 제거 및 ready queue에 추가
    }
  }
  // 4. ready queue
  int criterion = 0
  if (running == NULL) {
    for (ready queue 프로세스) {
      if (criterion < (프로세스 criterion)) { // 가장 큰 값을 갖는 프로세스 실행 가정
        criterion = (프로세스 criterion)
        running = 프로세스
      }
    }
  }
  current ++
endwhile
```

waiting queue는 I/O 작업이 완료된 노드부터 제거되므로 큐 형태로 제거할 수 없기 때문에, 매 시간마다 I/O 작업이 남은 프로세스를 가리키는 노드는 새로운 임시 큐에 추가하고, 완료된 프로세스는 ready queue에 추가한 뒤 기존의 모든 노드는 free()시키는 방법으로 매번 새로운 wait queue를 만들어 유지하였다.

Ready queue 역시 스케줄링 알고리즘에 따라 진입 순서와 관계없이 제거가 이루어져야 한다. 이 때 스케줄링의 기준이 되는 값이 프로세스 데이터 내에 존재한다면, 다음 실행 프로세스 판단 시 ready queue를 기준값의 오름차순 또는 내림차순으로 정렬하여 다음에 실행할 프로세스가 큐의 가장 앞으로 오도록 하였다. 본 프로젝트에서 구현한 스케줄링 알고리즘 중에는 SJF, Priority, Longest I/O Burst First 스케줄링에서 위 방법을 적용하였다.

*running 포인터는 NULL으로 초기화되고, 실행중인 프로세스가 없는 경우에도 NULL 값을 가지므로 NULL에 대한 접근을 방지하기 위하여 running!=NULL인 경우에만 프로세스 데이터에 대한 접근이 이루어지도록 조건을 추가하였다.

실제 ready queue의 정렬 및 다음 실행 프로세스의 판단은 모두 정렬이 필요한 스케줄링 알고리즘 함수 내에서 SortReadyQueue()함수를 호출과 함께 이루어진다. 스케줄링 알고리즘의 선택은 switch문을 통해 이루어지며, Schedule() 호출 시 전달받은 alg_id 값에 따라 대응되는 case의 스케줄링 알고리즘으로 스케줄링한다.



반복문 내에서 계산 순서는 프로세스의 상태 변화 조건에 대한 의존성을 기준으로 판단하였다. 프로세스의 생명주기를 다시 살펴보면, 결국 스케줄링에서 가장 중요한 ②와 ③은 나머지 상태 변경을 고려한 다음에 판단해야 한다. 우선 CPU에 대한 새로운 프로세스 할당 고려는 이번 시간이 종료되었을 때 CPU에 할당된 프로세스가 없거나, 이번 시간에 preemption이 발생하는 것을 전제로 한다. 따라서 ② 이전에 ③, ④, ⑥을 판단해야 한다. 이번 시간에 ready queue로 진입한 프로세스도 ②의 판단에 참여할 수 있어야 한다. 따라서 ①, ⑤도 ②보다 먼저 이루어져야 한다.

current_time-1에서 current_time이 되었을 때 새 프로세스의 도착과 함께 나머지 프로세스들은 값이 업데이트되는 것으로 볼 수 있지만, new 프로세스는 이전 시간에 ready queue에 존재하지 않았으므로 값을 업데이트한 이후에 추가(①)해야 한다. 서로 다른 세 가지 상태의 프로세스에 대한 값이 업데이트되므로 값을 업데이트하는 순서끼리는 의존성이 없다.

④와 ⑤의 경우 대기시간이 0인 I/O 작업은 없으므로 순서가 중요하지 않지만, waiting queue를 매번 새로 구성하는 형태로 구현하였기 때문에 ⑤ 이후에 ④를 수행하는 것이 노드가 할당 직후 삭제되는 낭비를 줄일 수 있어 이 순서로 수행하였다.

CPU에 프로세스가 할당된 상태에서 다른 프로세스가 동시에 들어올 수는 없으므로, ②는 반드시 ③ 이후에 이루어진다. ④와 ⑥은 Schedule()에서 직접 판단하지만, ③은 스케줄링 알고리즘에 따라 발생하거나 발생하지 않으므로 스케줄링 알고리즘 함수에서 결정된다. 따라서 모든 스케줄링 알고리즘 함수 역시 ③이후에 ②를 결정한다. 이렇게 구현할 경우 선점/비선점형 scheme이 모두 있는 알고리즘의 경우에도 preemptive 값만 함수에 추가로 전달하여 조건문을 추가하면 하나의 함수로 두 가지 scheme을 모두 구현할 수 있다.

(4) 스케줄링 알고리즘: FCFS(), SJF(), etc.

Schedule() 함수가 시뮬레이션 환경을 초기화하고 시간 단위 반복이 이루어지는 환경을 구성한다면, 각각의 스케줄링 알고리즘 함수들은 시뮬레이션 내에서 변화하는 값들로부터 다음에 CPU에 할당할 프로세스를 판단하는 핵심적인 역할을 수행한다. Schedule()의 반복문에서 스케줄링 알고리즘 함수 이전의 과정은 모두 스케줄링 함수 진입 시점에 각 프로세스의 상태를 current_time+1 시점의 것으로 업데이트해두기 위한 준비과정으로 볼 수 있다.

본 프로젝트에서는 총 9개의 스케줄링 알고리즘을 구현하였으며, 선점/비선점형 scheme이 존재하는 알고리즘의 경우 하나의 함수로 구현하여 총 6개의 함수를 정의하였다. 다음은 각 스케줄링 알고리즘의 동작을 서술한 수도코드와 그에 대한 설명이다. 모든 수도코드에서 state_queue에 대한 노드 추가 및 상태 업데이트 동작은 제외하였다.

① First-Come First-Served: FCFS()

```
void FCFS(Queue *ready, ProcessData **running, int current_time) {
    if (*running == NULL && ready->front != NULL) {
        *running = Dequeue(ready);
    }
}
```

조건) 실행중인 프로세스가 없고 레디큐에 프로세스가 존재
1. 가장 앞의 프로세스를 실행

결과적으로 ready queue에 진입한 순서대로 실행된다. 예를 들어 I/O request가 발생하여 waiting queue로 진입하면 ready queue의 가장 뒤로 들어가게 되므로 다른 프로세스보다 arrival time은 빠르더라도 ready queue로 재진입한 시점이 더 늦어 더 늦은 시각에 다시 CPU에 할당된다. 단, Schedule()에서 도착시간의 판단이 PID 순으로 이루어지므로 arrival time이 같다면 PID가 작은 프로세스가 우선 실행된다.

② Nonpreemptive/Preemptive SJF: SJF()

조건) 레디큐에 프로세스가 존재
1. 레디큐를 remaining_time 오름차순으로 정렬
조건) preemptive scheme에서 실행 중인 프로세스의 remaining_time이 더 긺
1.1 running 프로세스가 ready가 되고 running 포인터는 NULL로 변경
조건) running 포인터가 NULL
2. 가장 앞의 프로세스를 실행

(1) Nonpreemptive Scheme

```
void NonpreemptiveShortestJobFirst(Queue *ready, ProcessData **running, int current_time) {
    if (ready->front == NULL)
        return;
    do remaining_time 오름차순으로 레디큐를 정렬
    if (*running == NULL)
        *running = Dequeue(ready);
}
```

(2) Preemptive Scheme

```
void PreemptiveShortestJobFirst(Queue *ready, ProcessData **running, int current_time, int preemptive) {
    if (ready->front == NULL)
        return;
    do remaining_time 오름차순으로 레디큐를 정렬
    if (preemptive && *running != NULL) {
        if (레디큐 첫 번째 프로세스의 remaining_time >= 실행중인 프로세스의 remaining_time)
            return;
        Enqueue(ready, *running);
        *running = NULL;
    }
    if (*running == NULL)
        *running = Dequeue(ready);
}
```

실제 구현 시에는 두 scheme을 통합하여 preemptive=0일 때 비선점형으로, preemptive=1일 때 선점형으로 동작하는 SJF() 함수를 정의하였다. 실제 함수 구현 시에는 preemption 발생 시 상태 변화를 추적하기 위해 state_queue를 Queue *state로 받아 *running을 NULL로 초기화하기 전에 state queue에 추가하는 과정을 포함하였다.

③ Nonpreemptive/Preemptive Priority : Priority()

SJF 함수와 완전히 동일한 구성으로 이루어져 있으며, ready queue 정렬과 실행중인 프로세스에 대한 비교 기준값이 priority라는 차이점만 존재한다. 결과적으로 priority '값'이 가장 작은 프로세스가 높은 우선순위를 가진다.

(1) Nonpreemptive Scheme

```
void NonpreemptivePriority(Queue *ready, ProcessData **running, int current_time) {
    if (ready->front == NULL)
        return;
    do priority 오름차순으로 레디큐를 정렬 // priority 값이 작을 때 높은 우선순위를 가짐
    if (*running == NULL)
        *running = Dequeue(ready);
}
```

(2) Preemptive Scheme

```
void PreemptivePriority(Queue *ready, ProcessData **running, int current_time, int preemptive) {
    if (ready->front == NULL)
        return;
    do priority 오름차순으로 레디큐를 정렬 // priority 값이 작을 때 높은 우선순위를 가짐
    if (preemptive && *running != NULL) {
        if (레디큐 첫 번째 프로세스의 priority >= 실행중인 프로세스의 priority)
            return;
        Enqueue(ready, *running);
        *running = NULL;
    }
    if (*running == NULL)
        *running = Dequeue(ready);
}
```

실제 구현 시에는 두 scheme을 통합하여 preemptive=0일 때 비선점형으로, preemptive=1일 때 선점형으로 동작하는 Priority() 함수를 정의하였다. 실제 함수 구현 시에는 preemption 발생 시 상태 변화를 추적하기 위해 state_queue를 Queue *state로 받아 *running을 NULL로 초기화하기 전에 state queue에 추가하는 과정을 포함하였다.

void SortReadyQueue(Queue *q, int criteria);

큐와 정렬 기준에 대응하는 정수값을 입력받아 전달된 프로세스의 기준값에 따라 ready queue를 정렬한다. SortReadyQueue()의 알고리즘은 다음과 같다.

```
// 오름차순 예시 (입력받은 큐: queue)
if (큐가 유효하지 않거나 큐의 노드 개수가 0개 또는 1개) return
sorted = NULL
while (큐에 노드가 존재) {
  insert = 큐의 맨 앞에서 분리한 노드
  if (insert의 기준값이 sorted의 첫 번째 프로세스 기준값보다 작음(빈 경우 포함)) {
    insert가 sorted의 가장 앞으로 추가
  } else {
    do sorted에서 insert보다 기준값이 크거나 같은 노드를 찾을 때까지 탐색
    insert를 해당 노드 앞에 삽입
  }
}
do queue의 front, rear 연결을 sorted와 동일하게 만듦
```

이 때 실제 코드에서는 기준값 비교에 CompareProcess()를 사용한다. CompareProcess()는 SortReadyQueue() 호출 시 입력된 criteria를 그대로 전달받아 내부의 switch문을 통해 어떤 값을 비교할 것인지 결정한다. 예를 들어 CompareProcess(a, b, 1)에서는 priority를 비교하게 되고, 내부적으로는 a->priority - b->priority를 return한다. SortReadyQueue()에서 삽입 조건이 (CompareProcess(a, b, 1) < 0) 이므로 a의 priority 값이 더 작거나 같을 때 b 앞에 삽입된다. 즉, SortReadyQueue(q, 1) 호출 시 큐는 priority값에 대해 오름차순으로 정렬된다. 이는 곧 priority scheduling에서 priority 값이 작을수록 높은 우선순위를 가짐을 의미하고, 의도한 구현사항과 일치한다.

④ Round Robin : RoundRobin()

1. 시간초과 발생 여부 판단 후 초과 시 running 프로세스를 ready로 변경
2. 실행중인 프로세스가 없고 ready queue가 비어있지 FCFS로 다음 프로세스 결정
3. occupancy_time 초기화

```
void RoundRobin(Queue *ready, ProcessData **running, int current_time,
                int time_quantum, int *occupancy_time, Queue *state) {
    // running -> ready (대기중인 프로세스가 없더라도 우선 중단)
    if (실행중인 프로세스가 존재하고 time quantum 초과) {
        Enqueue(ready, *running);
        *running = NULL;
    }
    // ready -> running
    if (레디큐에 프로세스가 존재하고 실행 중인 프로세스가 없음) {
        *running = Dequeue(ready);
        *occupancy_time = 0; // 초기화
    }
}
```

Round Robin algorithm에서는 각각의 프로세스가 동등한 time quantum을 가지고 돌아가면서 CPU에 할당된다. 따라서 ready queue의 정렬은 따로 이루어지지 않고, 기본적으로 FCFS와 비슷하지만, 프로세스가 연속으로 CPU에 할당된 시간을 의미하는 occupancy_time을 추가로 전달받는다. 최초 occupancy_time의 정의 및 초기화는 Schedule()의 루프 이전에 이루어지며, 이후의 모든 초기화는 RoundRobin() 내에서 수행되고 Schedule()의 루프 내에서는 증가 연산만 수행하는 형태로 분리하여 관리된다.

실행 중인 프로세스가 없을 때는 FCFS와 완전히 동일하게 동작하며, 실행중인 프로세스가 있는 경우 time_quantum 값과 occupancy_time을 비교하여 expire 여부를 판단한다. 시간초과가 발생한 경우 즉시 실행 중이던 프로세스는 ready queue로 이동한다. 따라서 만약 시간 초과가 발생한 시점에 ready queue에 프로세스가 없더라도 우선 중단되고, 바로 다음 조건문에서 다시 time quantum 초기화와 함께 CPU를 할당받는 형태로 동작한다.

⑤ Lottery Scheduling : LotteryScheduling()

1. 시간초과 발생 여부 판단 후 초과 시 running 프로세스를 ready로 변경
2. 실행중인 프로세스가 없고 ready queue가 비어있지 않으면 추첨 진행
3. 실행중인 프로세스가 없는 경우 추첨 결과 winner가 running이 되고 점유시간 초기화

Lottery Scheduling 알고리즘에서는 'priority'값을 각 프로세스의 티켓 개수로 사용하여 추첨을 진행한다. 결과적으로 priority 값이 클수록 CPU에 할당될 확률이 높아진다. 티켓 추첨은 실행 중인 프로세스가 없고, ready queue에 프로세스가 존재할 때만 진행된다.

Ready queue에 있는 모든 프로세스의 'priority'값을 합하고, 이를 전체 티켓 개수로 한다. 이후 0부터 총 티켓 수-1 사이의 값을 랜덤으로 선택하고, 레디큐를 순회하면서 각 프로세스의 priority를 더하다가 합산한 값이 선택된 값을 초과할 때의 프로세스가 winner가 된다.

이 알고리즘의 경우 프로세스의 정해진 값이 아닌 추첨을 통해 확률적으로 다음 실행 프로세스를 결정하기 때문에 레디큐의 정렬이 이루어지지 않는다. 따라서 큐에서 winner 노드를

제거하고 연결을 조정하는 과정이 동반된다.

Lottery Scheduling 알고리즘의 경우에도 time quantum이 존재하여, ready queue의 프로세스 존재 여부와 관계 없이 시간 초과 시 즉시 ready queue로 이동한다. 이 경우 즉시 재추첨이 이루어져 자기 자신만 존재하는 상황에서 추첨을 통해 자기 자신이 다시 CPU에 할당된다. 이러한 알고리즘을 적용해야 하는 이유는 Lottery scheduling이 정말 확률적으로 CPU 할당을 결정할 수 있도록 해야 하기 때문이다. 만약 자기 자신이 즉시 해당 시간에 ready queue에 진입했다가 다시 추첨될 확률이 존재하지 않는다면, 전체 프로세스가 2개밖에 없는 경우 실제 티켓 분배 상황과 관계없이 Round Robin과 같은 방식으로 번갈아 CPU에 할당된다. 이는 Lottery scheduling 알고리즘의 의도와 맞지 않는다. 따라서 함수 호출 시 먼저 time quantum 초과 여부를 확인하고, 초과 시 실행 중인 프로세스를 ready queue로 돌려보내는 과정을 먼저 수행하여 다음 시간에 중단될 프로세스도 다시 다음 시간의 추첨에 참여할 수 있도록 하였다.

```
void LotteryScheduling(Queue *ready, ProcessData **running, int current_time,
                       int time_quantum, int *occupancy_time, Queue *state) {
    Node *winner = NULL;
    int total_tickets = 0, ticket_count = 0;
    // running -> ready (대기중인 프로세스가 없더라도 우선 중단)
    if (*running != NULL && *occupancy_time >= time_quantum) {
        Enqueue(ready, *running);
        *running = NULL;
    }
    // ready queue가 비어 있지 않을 때 추첨 진행
    if (*running == NULL) {
        if (ready->front == NULL) return;
        else {
            winner = ready->front;
            // priority 값을 각 프로세스의 ticket 개수로 사용 (higher priority value higher probability)
            for (레디큐 첫 node to 레디큐 마지막 node) {
                do node에 해당하는 프로세스의 priority를 total_tickets에 합산
            }
            int win = 0 ~ total_tickets 사이의 랜덤값
            for (레디큐 첫 node to 레디큐 마지막 node) {
                do node에 해당하는 프로세스의 priority를 ticket_count에 합산
                if (win <= ticket_count) {
                    winner = node;
                    break;
                }
            }
        }
    }
    if (*running == NULL) {
        do winner를 ready queue에서 제거 후 연결 재조정
        // ready -> running
        *running = winner->p_process;
        *occupancy_time = 0;
    }
}
```

⑥ Nonpreemptive/Preemptive LongestIOFirst()

I/O request가 발생하여 프로세스가 해당 요청을 수행하면 그 동안 cpu는 다른 프로세스를 실행할 수 있다. 따라서 i/o burst가 긴 프로세스를 먼저 실행시키면 cpu 사용률을 높일 수 있을 것으로 기대하였다.

io_burst[]에 나누어져 있는 I/O burst의 총합을 매번 계산하는 것은 비효율적이므로 프로세스 구조체에 total_remaining_io 필드를 추가하였다. total_remaining_io는 매 스케줄링 알고리즘 시작 시 모든 유효한 io_burst[j]의 합으로 초기화되며, 스케줄 함수 내에서 I/O request 발생 시점에 도달하여 waiting queue로 진입할 때마다 갱신된다. 남아있는 I/O burst의 합이 가장 큰 프로세스를 우선으로 실행하기 위해 total_remaining_io 내림차순으로 레디큐를 정렬한다.

```
void LongestIOFirst(Queue *ready, ProcessData **running, int current_time, int preemptive, Queue *state) {
    if (레디큐가 비었거나 nonpreemptive이고 실행 중인 프로세스가 있음) return;

    do 프로세스의 남아있는 전체 I/O burst 내림차순으로 ready queue 정렬

    if (preemptive && *running != NULL) {
        do 정렬된 ready queue 첫 번째 프로세스의 remaining I/O burst time이 실행중인 프로세스의 것보다 클 때 preempted
    }
    // ready -> running
    if (*running == NULL) {
        *running = Dequeue(ready);
    }
}
```

(5) 출력: Evaluation() 등

생성된 프로세스 정보의 출력, 간트차트 출력, 성능 평가 결과 출력 등 스케줄링 정보와 결과를 실제 사용자가 쉽게 확인할 수 있도록 출력하는 함수이다. 각 함수가 호출되는 위치는 다르지만 특성상 일정한 출력을 위한 반복적인 구성이 포함되어 있어 가독성을 높이기 위해 분리하였다.

void PrintGanttChart(ProcessData p[], int pNum, Queue *state_queue);

본 프로젝트에서는 Gantt chart를 스케줄링이 끝난 후에 일정한 형태로 출력하고자 하였다. Schedule()의 state_queue는 간트 차트를 출력하기 위해 추가한 큐이다. 스케줄링이 완료되면 우선 전체 프로세스 중 마지막으로 끝난 프로세스의 completion_time을 chartSize로 하여 해당 길이의 int, char 배열을 각각 동적할당한다. 이후 정수 배열은 -1, char 배열은 공백(‘ ’)으로 초기화한 뒤 state_queue를 순회하면서 node에 해당하는 시간(node->time)에 대응하는 index의 값만 PID나 상태 변경 원인을 뜻하는 문자로 업데이트한다.

이 함수를 통해 총 스케줄링 시간에 관계없이 항상 일정한 행의 출력을 유지할 수 있다.

void Evaluation(int alg_id, ProcessData p[], int pNum);

프로세스 구조체 배열을 받아 스케줄링 종료 시점의 스케줄링 데이터를 바탕으로 average waiting time, average turnaround time, average response time을 계산한 뒤 출력한다. 이 때 평균값과 함께 각 프로세스의 성능값을 같이 출력하여 프로세스별 시간 차이를 확인할 수 있도록 구현하였다.

4. 실행 및 성능 비교

1) 전체 알고리즘 출력 결과 비교

구현한 CPU scheduling simulator의 정상 동작을 확인하기 위해 1회 실행한 결과를 텍스트 파일로 저장하였다. 다음은 모든 스케줄링 알고리즘에 대하여 6개의 프로세스를 (time quantum=2로) 스케줄링한 결과이다.

(1) 생성된 프로세스 정보

PID	Arrival time	Priority	CPU burst	I/O (timing, burst)
0	1	8	6	(3, 2)
1	1	5	8	(4, 2), (5, 3)
2	3	1	3	(2, 2)
3	2	9	7	(5, 1)
4	1	5	8	-
5	3	9	9	(7, 1)

(2) 알고리즘별 스케줄링 결과

- FCFS

```
================================================================================
Gantt Chart     (T: terminated, I: I/O request, P: preempted, E: time quantum expired)
Total scheduling time : 42

        ____________________________________________________________________________________________
process|X|    0|      1|              4|        3|  2|           5|    0|1|  3|2|  5|    1|
event  |_|____I|______I|_______________T|_________I|__I|____________I|____T|I|__T|T|__T|____T|
time   0 1     4       8              16        21  23            30    3334  3637  39    42

Performance of FCFS algorithm
average waiting time    : 23.33      (24, 28, 29, 26, 7, 26)
average turnaround time : 32.00      (32, 41, 34, 34, 15, 36)
average response time   : 10.33      (0, 3, 18, 14, 7, 20)
```

- Nonpreemptive SJF

```
================================================================================
Gantt Chart     (T: terminated, I: I/O request, P: preempted, E: time quantum expired)
Total scheduling time : 43

        ____________________________________________________________________________________________
process|X|    0|  2|    0|2|       3|      1|  3|1|             4|    1|           5|X|  5|
event  |_|____I|__I|____T|T|________I|______I|__T|I|______________T|____T|____________I|_|__T|
time   0 1     4   6     910        15      19  2122              30    33            4041  43

Performance of Nonpreemptive SJF algorithm
average waiting time    : 13.83      (0, 19, 2, 11, 21, 30)
average turnaround time : 22.50      (8, 32, 7, 19, 29, 40)
average response time   : 12.33      (0, 14, 1, 8, 21, 30)
```

- Preemptive SJF

```
================================================================================
Gantt Chart     (T: terminated, I: I/O request, P: preempted, E: time quantum expired)
Total scheduling time : 43

      ____________________________________________________________________________
process|X|  0|  2|0|3|2|    0|      3|1|  3|    1|  4|1|          4|    1|           5|X|  5|
event  |_|__P|__I|I|P|T|____T|______I|P|__T|____I|__P|I|___________T|____T|____________I|_|__T|
time   0 1   3   5 6 7 8    11      1516  18    21  2324          30    33            4041  43

Performance of Preemptive SJF algorithm
average waiting time    : 13.33      (2, 19, 0, 8, 21, 30)
average turnaround time : 22.00      (10, 32, 5, 16, 29, 40)
average response time   : 11.33      (0, 14, 0, 4, 20, 30)
```

- Nonpreemptive Priority

```
================================================================================
Gantt Chart     (T: terminated, I: I/O request, P: preempted, E: time quantum expired)
Total scheduling time : 42

      ____________________________________________________________________________
process|X|      1|  2|              4|2|1|    0|    1|    0|        3|            5|  3|  5|
event  |_|______I|__I|_______________T|T|I|____I|____T|____T|________I|_____________I|__T|__T|
time   0 1       5   7               151617    20    23    26        31            38  40  42

Performance of Nonpreemptive Priority algorithm
average waiting time    : 16.50      (17, 9, 8, 30, 6, 29)
average turnaround time : 25.17      (25, 22, 13, 38, 14, 39)
average response time   : 12.67      (16, 0, 2, 24, 6, 28)
```

- Preemptive Priority

```
================================================================================
Gantt Chart     (T: terminated, I: I/O request, P: preempted, E: time quantum expired)
Total scheduling time : 42

      ____________________________________________________________________________
process|X|  1|  2|  4|2|  1|          4|1|    0|    1|    0|        3|            5|  3|  5|
event  |_|__P|__I|__P|T|__I|___________T|I|____I|____T|____T|________I|_____________I|__T|__T|
time   0 1   3   5   7 8  10           1617    20    23    26        31            38  40  42

Performance of Preemptive Priority algorithm
average waiting time    : 15.33      (17, 9, 0, 30, 7, 29)
average turnaround time : 24.00      (25, 22, 5, 38, 15, 39)
average response time   : 12.00      (16, 0, 0, 24, 4, 28)
```

- Round Robin (time quantum = 2)

```
================================================================================
Gantt Chart     (T: terminated, I: I/O request, P: preempted, E: time quantum expired)
Total scheduling time : 42

        ____________________________________________________________________________________________
process|X|  0|  1|  4|  3|  2|  5|0|  1|  4|  3|2|  5|  0|1|  4|3|  5|0|  4|  1|  3|5|1|  5|
event  |_|__E|__E|__E|__E|__I|__E|I|__I|__E|__E|T|__E|__E|I|__E|I|__E|T|__T|__E|__T|I|T|__T|
time   0 1   3   5   7   9  11  1314  16  18  2021  23  2526  2829  3132  34  36  383940  42

Performance of RoundRobin algorithm
average waiting time    : 24.00      (23, 26, 13, 28, 25, 29)
average turnaround time : 32.67      (31, 39, 18, 36, 33, 39)
average response time   : 4.17      (0, 2, 6, 5, 4, 8)
```

- Lottery Scheduling (time quantum = 2)

```
================================================================================
Gantt Chart     (T: terminated, I: I/O request, P: preempted, E: time quantum expired)
Total scheduling time : 42

        ____________________________________________________________________________________________
process|X|  1|  1|  0|  3|  4|  3|  5|  5|  5|0|5|3|  5|1|  3|  0|  1|  4|0|  2|  4|1|2|  4|
event  |_|__E|__I|__E|__E|__E|__E|__E|__E|__E|I|I|I|__T|I|__T|__E|__E|__E|T|__I|__E|T|T|__T|
time   0 1   3   5   7   9  11  13  15  17  19202122  2425  27  29  31  3334  36  383940  42

Performance of Lottery Scheduling algorithm
average waiting time    : 23.83      (25, 25, 32, 17, 33, 11)
average turnaround time : 32.50      (33, 38, 37, 25, 41, 21)
average response time   : 9.67      (4, 0, 31, 5, 8, 10)
```

- Nonpreemptive Longest I/O Burst First

```
================================================================================
Gantt Chart     (T: terminated, I: I/O request, P: preempted, E: time quantum expired)
Total scheduling time : 42

        ____________________________________________________________________________________________
process|X|      1|    0|1|  2|        3|           5|             4|    0|    1|2|  3|  5|
event  |_|______I|____I|I|__I|________I|___________I|______________T|____T|____T|T|__T|__T|
time   0 1       5     8 9  11        16           23             31    34    3738  40  42

Performance of Nonpreemptive Longest I/O Burst First algorithm
average waiting time    : 26.50      (25, 23, 30, 30, 22, 29)
average turnaround time : 35.17      (33, 36, 35, 38, 30, 39)
average response time   : 9.00      (4, 0, 6, 9, 22, 13)
```

- Preemptive Longest I/O Burst First

```
================================================================================
Gantt Chart     (T: terminated, I: I/O request, P: preempted, E: time quantum expired)
Total scheduling time : 42

        ____________________________________________________________________________
process|X|      1|  0|1|  2|0|        3|            5|              4|    1|2|    0|  3|  5|
event  |_|______I|__P|I|__I|I|________I|_____________I|_______________T|____T|T|____T|__T|__T|
time   0 1       5   7 8  1011        16            23              31    3435    38  40  42

Performance of Preemptive Longest I/O Burst First algorithm
average waiting time    : 26.17      (29, 20, 27, 30, 22, 29)
average turnaround time : 34.83      (37, 33, 32, 38, 30, 39)
average response time   : 8.83      (4, 0, 5, 9, 22, 13)
```

(3) 전체 결과 종합 및 분석

Scheduling algorithm	전체 소요시간	Average waiting time	Average turnaround time	Average response time
FCFS	42	23.33	32.00	10.33
비선점형 SJF	43	13.83	22.50	12.33
선점형 SJF	43	13.33	22.00	11.33
비선점형 Priority	42	16.50	25.17	12.67
선점형 Priority	42	15.33	24.00	12.00
Round Robin	42	24.00	32.67	4.17
Lottery scheduling	42	23.83	32.50	9.67
선점형 I/O bound 우선	42	26.50	35.17	9.00
비선점형 I/Obound 우선	42	26.17	34.83	8.83

Average turnaround time은 Preemptive SJF algorithm에서 최솟값을 가졌다. Nonpreemptive scheme의 경우에도 두 번째로 짧은 평균 소요 시간을 보였다.

반면 응답 시간 측면에서는 Round Robin 방식이 가장 짧았다. Time quantum이 2로 설정되어 더 자주 context switching이 발생한 탓에 프로세스가 추가되어 처음으로 CPU에 할당되기까지 걸리는 시간은 가장 짧았다. 하지만 time quantum 대비 생성된 프로세스들의 CPU burst 값이 컸기 때문에 average turnaround time은 긴 편이었다.

잔여 I/O burst time이 긴 프로세스를 우선하는 알고리즘의 경우 응답시간 면에서는 Round Robin 다음으로 좋은 결과를 나타내었지만, 평균 소요 시간에서는 가장 나쁜 결과를 나타냈다. 그 이유는 모든 프로세스의 I/O request가 끝나면 해당 알고리즘은 단순한 FCFS로 동작하게 되는데, I/O burst가 발생하지 않아 가장 우선순위가 낮은 프로세스가 긴 CPU burst time을 가진 탓에 나머지 프로세스들은 대부분의 작업을 마치고 종료만을 기다리는 상황에서 convoy effect가 발생한 것으로 보인다.

2) 랜덤 프로세스 집합에서 반복 수행

알고리즘의 성능이 프로세스 집합의 특성에 따라 변할 수 있으므로 성능 평가 값만을 출력하도록 코드를 일부 변형하여 각 알고리즘에 대하여 1000개의 서로 다른 프로세스 집합으로 스케줄링한 성능을 분석하였다.

(1) 코드 변형

우선 main함수에서 입력 없이 고정된 조건에서 프로그램이 실행되도록 하였고, 프로세스의 개수는 10개, time quantum은 3으로 고정하였다. 프로세스와 간트차트에 대한 출력 함수는 주석처리하였고, Evaluation()은 알고리즘 이름과 성능값만 쉼표로 구분되도록 출력하여 CSV 파일로 저장하였다.

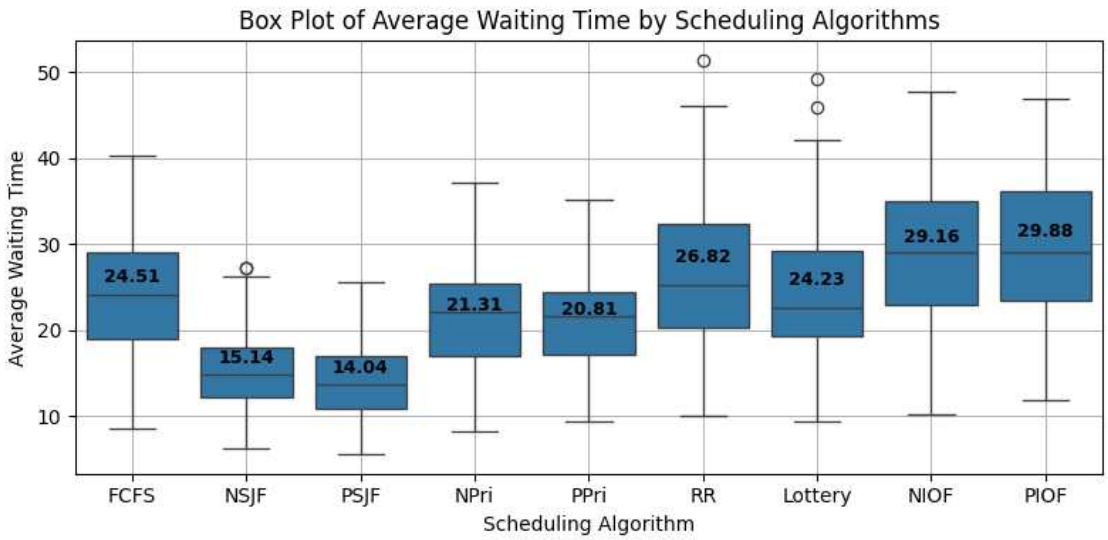
```c
int main(void) {
    ProcessData processes[MAX_PROCESSES];
    int processNum = 10;
    int timeQuantum = 3;
    printf("Alg,waiting,turnaround,response\n");
    srand(time(NULL));
    for (int i = 0; i < 1000; i++) {
        CreateProcess(processes, processNum);
        for (int j = 0; j < SCHEDULERS; j ++) {
            Schedule(j, processes, processNum, timeQuantum);
            Evaluation(j, processes, processNum);
        }
    }
}
```

```c
switch (alg_id) {
    case 0:
        printf("FCFS,");
        break;
    ...
}
printf("%.2f,", avg_waiting_time);
printf("%.2f,", avg_turnaround_time);
printf("%.2f\n", avg_response_time);
```

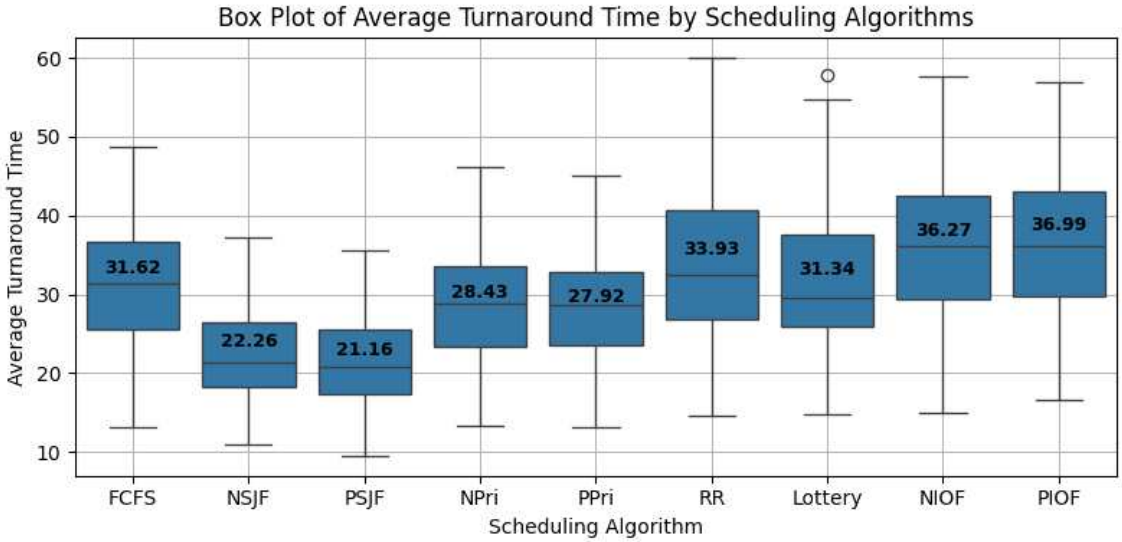
(2) 결과 요약

출력된 CSV 파일을 Python 시각화 라이브러리를 활용하여 시각화하였다. 다음은 각 스케줄링 결과의 성능값의 분포를 나타낸 Box plot이다. 상자에 표시된 숫자는 모든 1회 스케줄링 결과의 평균 성능값에 대한 1000회 평균값이다.

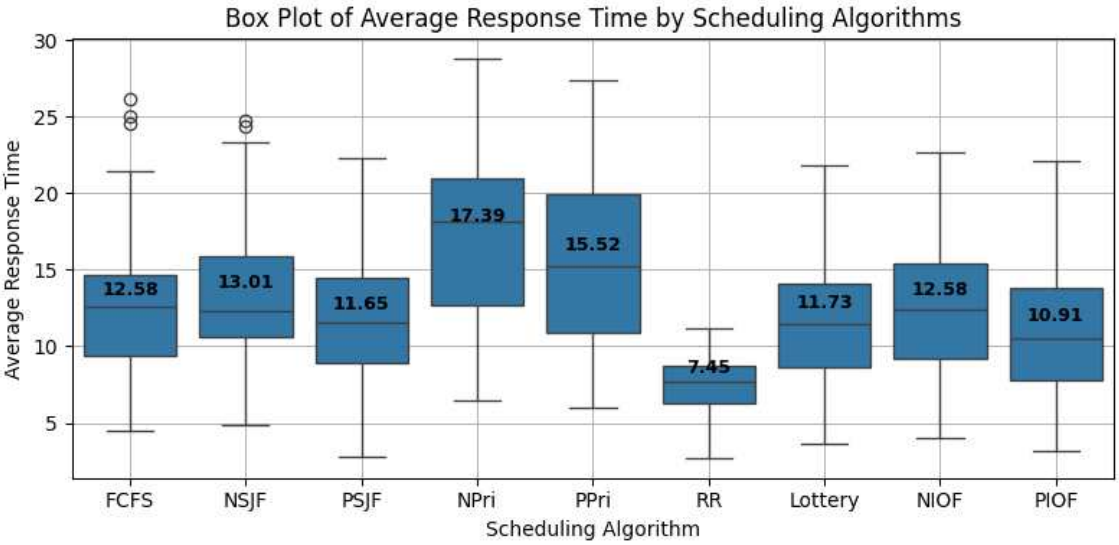
- Average Waiting Time



- Average Turnaround Time



- Average Response Time



(3) 알고리즘별 특징 분석

Average waiting time과 average turnaround time의 경우 Shortes-Job-First 알고리즘에서 가장 좋은 결과가 나타났다. 또한 SJF와 Priority scheduling 알고리즘 공통으로 preemption이 존재하는 scheme에서 존재하지 않는 scheme보다 미세하게 더 나은 성능을 보였다.

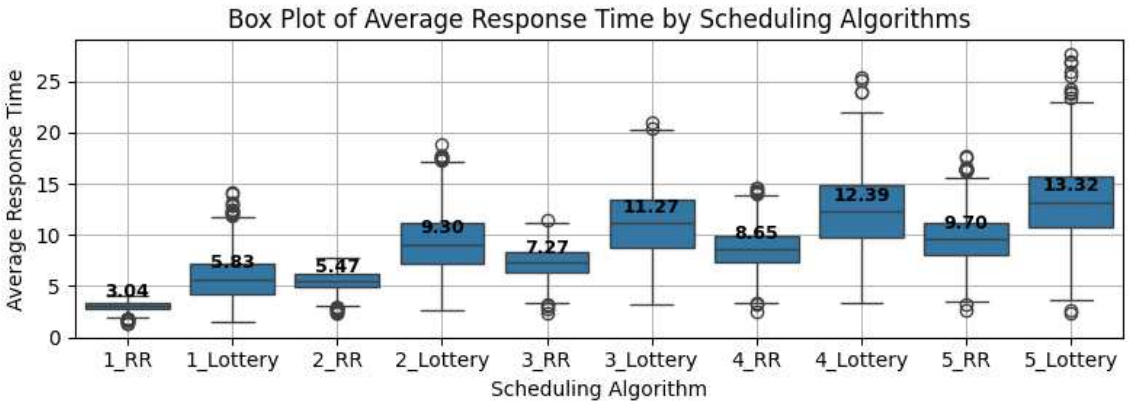
반면 average response time은 Round Robin이 항상 가장 좋은 결과를 나타냈다. 해당 시뮬레이션에서 time quantum은 3으로 고정하였는데, 만약 time quantum이 더 작아진다면 average response time은 더 짧아지겠지만 전체적인 프로세스 종료는 거의 끝부분에서 동시에 일어나게 되어 turnaround time은 증가할 것이다.

Lottery scheduling algorithm의 경우 평균적인 성능은 항상 중간 정도의 결과를 보였다. SJF나 Priority scheudling보다는 낮은 성능을 보였지만, FCFS나 Round Robin보다는 미세하게 나은 성능을 보였다.

마지막으로 I/O burst time이 긴 프로세스를 우선 실행하는 알고리즘의 경우 average response time을 제외하면 성능 면에서 뒤떨어지는 결과를 보였다. 이는 해당 알고리즘의 Gantt chart를 통해 이유를 알 수 있다.

I/O request가 존재하는 시간까지는 상대적으로 전체 프로세스가 고르게 할당된 반면, 그 이후로는 FCFS로 동작하여 문제가 발생하였다. 처음부터 FCFS로 동작할 경우 차라리 CPU burst가 긴 프로세스가 빨리 실행되어 종료되면 나머지 프로세스도 순차적으로 종료될 수 있다. 하지만 이 알고리즘에서는 I/O 작업을 모두 끝내고 종료를 위한 CPU 할당만을 남겨두고 있는 프로세스가 다른 프로세스들에 의해 대기하면서 FCFS 알고리즘보다도 성능이 나쁜 모습을 보였다. 이 현상은 I/O request가 발생하지 않아 우선순위가 가장 낮은 CPU bound process가 긴 CPU burst time을 가질 때 가장 심했다.

따라서 해당 알고리즘이 정상적인 성능을 보이게 하려면 I/O request가 모두 완료된 이후에 새로운 스케줄링 알고리즘을 적용하여 프로세스의 마무리 단계가 효과적으로 이루어질 수 있도록 해야 할 것이다.



추가적으로 time quantum을 사용하는 Round Robin과 Lottery Scheduling 알고리즘의 경우 time quantum을 변화시켜가며 시뮬레이션하였다. 그 결과 average response time에서는 예상할 수 있듯 time quantum이 작을수록 더 짧은 응답시간을 보였다. 하지만 이외에 전체 소요시간이나 대기시간에서는 큰 차이가 나타나지 않았다. 프로세스의 구성에 따라 time quantum이 길어질수록 average waiting time과 turnaround time이 감소하는 경우도 있었지만, 랜덤한 프로세스 조합에 대하여 평균을 낸 결과 차이가 나타나지 않게 되었다.

결론적으로 average waiting time, average turnaround time, average response time의 관점에서 스케줄링 알고리즘으로서 가장 우수한 성능을 보이는 알고리즘은 Shortest Job First algorithm이었다. 소요시간과 대기시간 면에서 평균적으로 가장 낮은 수치를 기록했고, priority에 따라 지나치게 편차가 커지는 Priority scheduling과 달리 응답시간 면에서도 다른 프로세스들과 비슷한 수준의 수치를 보였다. 하지만 SJF 알고리즘은 CPU burst time 예측의 어려움이라는 문제 때문에 실제 시스템에서 적용할 수 없다는 한계가 있다.

만약 response time이 중요한 환경이라면 Round Robin 방식이 가장 적합한 스케줄링 알고리즘이다. Time quantum에 따른 차이는 있지만, RR 알고리즘은 모든 프로세스를 돌아가면서 CPU에 할당하기 때문에 프로세스의 특성에 구애받지 않고 적은 편차로 항상 낮은 average response time을 보장한다.

III. 결론 및 고찰

1. 시뮬레이터 기능 정리

1) 기능 요약

본 프로젝트에서 구현한 CPU scheduling simulator에서는 FCFS, 선점/비선점형 SJF, 선점/비선점형 Priority scheduling, Round Robin, Lottery Scheduling 등 총 9개의 스케줄링 알고리즘을 사용하여 최대 10개의 프로세스가 동시에 존재하는 상황에 대한 스케줄링을 시뮬레이션 할 수 있다. 이 때 각 프로세스는 0~10의 랜덤한 시각에 도착하여 1~10의 CPU burst time과 priority 값을 갖고, 0~2회의 시점에 1~3 길이의 대기를 유발하는 I/O request가 발생한다.

프로세스 생성 시 생성된 프로세스 정보가 테이블 형태로 출력되고, 특정 알고리즘을 적용한 스케줄링 시뮬레이션이 종료될 때마다 스케줄링 결과를 정리한 간트 차트와 스케줄링 성능을 평가한 평균 waiting time, turnaround time, response time 정보도 출력된다.

2) 프로세스 데이터 확장성

생성하는 프로세스의 개수나 CPU burst time, priority, arrival time 등의 값을 편의를 위해 1~10 정도의 값으로 제한하였다. 하지만 실제로 시뮬레이션을 진행할 때 제한된 값에 대해서만 스케줄링이 가능하도록 구현하지 않았기 때문에 프로세스 데이터에 대한 제한값을 변경해도 정상적으로 작동할 것이다.

먼저 최대 프로세스 개수의 경우 #define으로 전처리하여 해당 값만 변경하면 더 많은 프로세스에 대한 스케줄링이 가능하다. 모든 큐가 노드 구조체의 연결리스트로 구현되어 있기 때문에 큐의 크기가 부족한 상황도 발생하지 않는다. 같은 관점에서 간트차트 출력 또한 state_queue와 차트 데이터를 저장하는 배열이 모두 동적으로 할당되므로 시뮬레이션 시간이 더 길어져도 정상적으로 작동한다.

Time quantum의 경우 역시 #define으로 최댓값을 5로 제한하였지만 실제로는 occupancy_time으로 시간 초과를 판단하기 때문에 5보다 큰 값으로 설정하여도 문제가 발생하지 않는다. 프로세스별 최대 I/O request 발생 가능 횟수인 MAX_IO_NUM의 경우에도 I/O 요청 횟수 랜덤 결정 시 MAX_IO_NUM과 cpu_burst-1 중 더 작은 값을 사용하도록 제한되어 있으므로 더 크게 설정 가능하다.

SCHEDULERS 값은 구현한 스케줄링 알고리즘의 수를 의미한다. 이 경우에는 스케줄링 알고리즘을 추가하려면 해당 값을 변경한 뒤 Schedule()의 switch문에서 새로운 가장 큰 값의 case에 해당 알고리즘을 배치하고, Evaluation() 함수에서도 switch문에 case 추가 후 알고리즘 명칭을 정상적으로 출력하도록 수정해야 한다. main함수에서 입력값 제한 범위는 SCHEDULERS로 자동으로 변경되지만, 알고리즘 별로 대응하는 번호를 설명하기 위한 출력문을 수정해야 한다.

프로세스 구조체의 경우 값에 접근할 때 프로세스 단위로만 탐색이 이루어지고 멤버 단위의 주소 탐색이 이루어지는 경우는 존재하지 않는다. 따라서 구조체에 자유롭게 멤버를 추가할 수 있다. 따라서 다른 스케줄링 알고리즘을 구현할 때 새로운 기준값을 프로세스 구조체에 추가하고 해당 기준으로 연산하는 case를 CompareProcess()에 추가하면 기존에 정의한 SortReadyQueue() 함수를 활용할 수 있다.

2. 개선사항

1) 성능 평가 및 일괄 출력 기능

현재는 성능 출력 시 평균값과 함께 각 프로세스 별 개별 수치를 함께 출력하여 프로세스별 처리 시간의 차이를 확인할 수 있도록 하였다. 여기서 더 나아가 직접 최솟값/최댓값이나 편차 등 스케줄링 알고리즘의 공정성을 평가할 수 있는 요소도 추가할 수 있을 것이다. 또한 현재 프로그램은 전체 시뮬레이션 종료 후 스케줄링한 모든 알고리즘에 대한 평가 정보를 일괄적으로 출력하지는 않는다. 성능 지표만을 저장할 수 있는 구조체를 새로 정의한 뒤, 시뮬레이션이 끝날 때마다 성능 지표 값들을 동적할당되 새로운 구조체에 저장하고, 이를 큐로 관리하여 모든 시뮬레이션 종료 시점에 순서대로 출력하도록 하면 알고리즘별 성능 차이를 확인하기 더 쉬울 것으로 생각한다.

2) 스케줄링 알고리즘

마지막에 구현한 I/O burst time이 길수록 높은 우선순위를 가지는 형태의 알고리즘의 경우 성능 면에서 가장 나쁜 모습을 보였다. 처음에 이 알고리즘을 구현한 이유는 I/O 요청이 발생할 프로세스의 실행이 뒤로 지연되면 해당 프로세스들만 남은 상태에서 I/O 대기로 인해 CPU가 idle 상태가 될 가능성이 있다고 생각했기 때문이다. 그래서 I/O burst가 잦거나 긴 I/O bound 프로세스에 먼저 CPU를 할당하여 전체 프로세스 수가 충분할 때 I/O 대기에 들어갈 수 있도록 의도하여 구현한 알고리즘이다. 그러나 모든 I/O 요청이 끝난 이후에 다른 스케줄링 기준이 없다는 점을 고려하지 않아서 문제가 발생했다. I/O bound 프로세스를 우대한다는 것은 CPU bound 프로세스의 우선순위가 낮다는 것인데, 그 상태에서 CPU burst만 일어나는 프로세스가 뒤쪽에 몰린 결과 I/O bound 프로세스까지 종료되지 못하는 문제가 생겼다. 이를 해결하기 위해서는 I/O burst와 CPU burst를 종합적으로 고려하거나, I/O burst보다는 우선순위가 낮은 제2의 스케줄링 기준을 함께 적용하는 방법으로 대응해야 할 것이다.

3. 프로젝트 소감

CPU scheduling simulator를 직접 구현하면서 스케줄링 알고리즘에 대한 이해도와 해당 알고리즘이 제안된 배경을 더 잘 이해할 수 있었다. 현재 거의 모든 컴퓨터 시스템은 멀티프로그래밍을 기본으로 하는 만큼, CPU 스케줄링에 대한 이해는 컴퓨터 시스템을 더 잘 활용하고 관리하는데 큰 도움이 될 것이라고 생각한다.

IV. References

[1] Daeyoung Kim. (2017). CPU-Scheduler. github.com. https://github.com/cyc1am3n/CPU-Scheduler

[2] Inho Kim. (2016). CPU-Scheduling-Simulator. github.com. https://github.com/arkainoh/CPU-Scheduling-Simulator

[3] Guijung Woo. (2023). https://github.com/woog2roid/Korea-University-COSE-Assignments