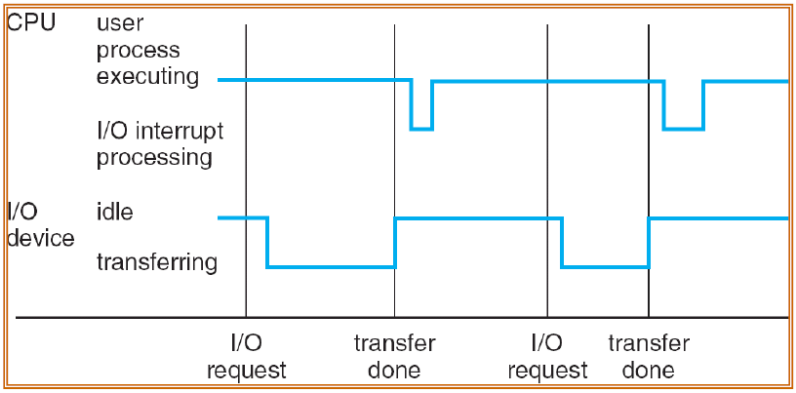
Operating System Report

1. CPU_Scheduler Introduction

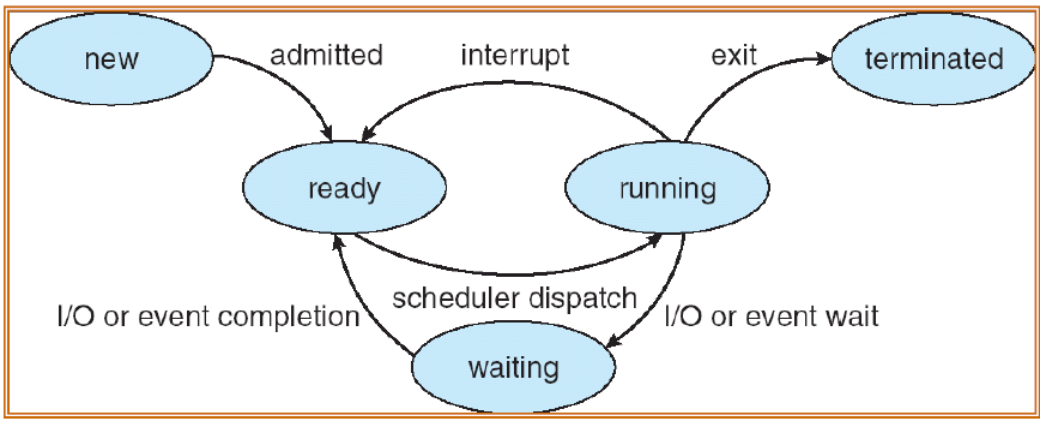
CPU scheduler란 컴퓨터의 operating system의 한 부분으로, 프로세스들에게 CPU 할당을 관리하면서 시스템의 성능에 큰 영향을 줄 수 있다. 메모리 내에 존재하면서, 실행될 준비가 되어 있는 프로세스들 중에서 CPU자원을 할당해줄 프로세스를 선별한다. 특히 프로세서가 하나인 single processor에서는 한 번에 최대 한 개의 프로세스만 CPU자원을 할당받을 수 있다.

1-1. Process State and Scheduling

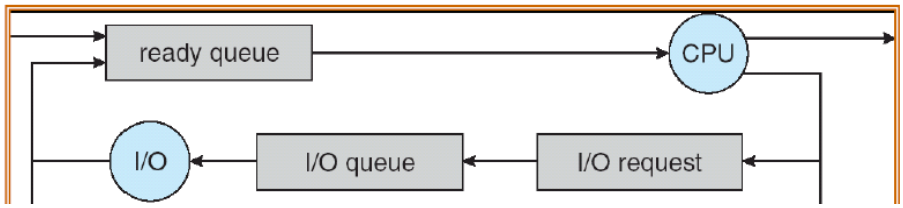
프로세스의 현재 상태를 기준으로 분류하자면, 현재 수행되고 있는 프로세스, 즉 현재 CPU 자원을 할당받아 사용하고 있는 프로세스는 running state에 존재하며, 이 프로세스를 runningProcess로 분류한다. 그리고 수행될 준비가 되어 CPU자원을 할당받을 준비가 되어 있는 프로세스는 ready state에 존재한다. 또한 프로세스는 발생하는 어떠한 event에 대해서 그 event를 처리하기 위한 자원을 대기하는 등, CPU작업 진행 중 block이 될 수 있다. 대표적인 event로 본 프로젝트에서 구현한 I/O request에 의한 event 발생을 예로 들 수 있다. 프로세스는 발생한 I/O request를 수행하기 위해 진행 중이던 CPU 작업이 block되고, waiting state로 돌입한다. 즉, running state에서 waiting state로 프로세스 상태가 변화한다.



위 그림에서 알 수 있듯이, I/O request를 처리한 후, I/O interrupt가 발생하고, waiting state에서 ready state로 상태가 변화한다.



앞서 설명한 개념들을 위 그림으로 나타낼 수 있다. 프로세스들의 state가 구분되고, 각 프로세스들의 state에 따라 여러queue에 위치된다.



CPU자원을 할당받을 준비가 되어있는, ready state의 프로세스들은 위 그림의 ready queue에 위치하고, I/O request가 발생하여, block된 프로세스들은 I/O request를 처리를 위한 I/O device 할당을 위해 waiting queue(I/O waiting queue)에 위치한다.
따라서 CPU scheduler는 ready queue에 존재하는 ready state의 프로세스들 중에서 CPU 자원을 할당받을 프로세스를 scheduling 기법을 통해서 선별한다.

1-2. Scheduling Criteria

1) CPU utilization
프로세스가 수행되면서 CPU가 최대한 idle상태에 들어가지 않고 최대한 바쁘게 프로세스를 수행해야 한다.

2) Throughput
단위시간당 수행되는 프로세스의 수이다.

3) Turnaround time
프로세스가 수행될 준비가 되어서 submission된 시점부터 모든 수행이 완료되어 complete 된 시점까지 걸린 시간을 의미한다.

4) Waiting time
프로세스가 ready queue에서 수행 대기하면서 보내는 모든 시간의 합이다. 다른 프로세스가 schedule되어 수행되는 동안 기다린 시간을 의미한다.

5) Response time
프로세스가 수행되기 위해서 request를 submission한 후, 첫 response가 발생하는, 즉 처음으로 CPU에서 수행될 때 까지 걸리는 시간이다.

위 Criteria에 따라, 가장 효율적인 Scheduling을 위해서는 CPU utilization과 throughput을 최대화하고, turnaround time, waiting time, response time을 최소화해야 한다.

1-3. Scheduling Algorithms

1) FCFS (First Come First Served)
FCFS 알고리즘은 기본적으로 FIFO queue를 사용한다. 가장 먼저 도착한 프로세스 순으로 CPU자원을 할당받는다. 또한 non-preemptive, 비선점 알고리즘이므로, 한번 수행 시작된 프로세스가 완료되거나 I/O 작업 수행 등, 각종 interrupt가 발생하지 않는 한, 다른 프로세스가 수행되는 경우는 없다.

2) SJF (Shortest Job First)
SJF알고리즘은 CPU burst time이 작은 프로세스 순으로 CPU자원을 할당받는다. 이 알고리즘은 preemptive와 non-preemptive 가 둘 다 가능한데, non-preemptive로 구현하면, FCFS와 마찬가지로, 한번 수행 시작된 프로세스가 완료되거나 interrupt가 발생하지 않는

한, 다른 프로세스가 수행될 수 없다. 하지만 preemptive로 구현하면, 남아있는 CPU burst time, remaining CPU burst time이 더 적은 프로세스가 존재하면, 기존에 수행 중 이던 프로세스의 수행을 중단시키고, remaining CPU burst time이 더 적은 프로세스를 우선적으로 수행한다.

3) Priority
Priority 알고리즘이 수행되기 위해서는 프로세스들에게 priority, 즉 우선순위가 부여되어야 한다. 그리고 사전에 부여된 이 우선순위를 기준으로 우선순위가 높은 프로세스 순으로 CPU자원을 할당한다. preemptive, non-preemptive방식이 가능한데, non-preemptive는 앞서 설명한 것과 마찬가지로 한 프로세스가 수행중일 때에는, 수행 완료, interrupt 등의 상황을 제외하면 다른 프로세스가 수행될 수 없고, preemptive는 한 프로세스가 수행 중일 때, 우선순위가 더 높은 프로세스가 submission되면, 수행 중이던 프로세스의 수행이 중단되고, 더 높은 우선순위의 프로세스가 수행된다.

4. Round Robin
Round Robin 알고리즘이 수행되기 위해서는 먼저 Time quantum이 도입된다. 한 프로세스는 CPU자원을 할당받아 한번에 Time quantum의 시간만큼 연속적으로 작업을 수행할 수 있고, Time quantum의 시간을 모두 소비하면, context switch가 발생하여, 다시 ready queue로 돌아가고 다음 schedule된 프로세스가 Time quantum만큼 수행된다. scheduling의 기본적인 틀은 FCFS와 동일하여, 먼저 도착한 프로세스가 먼저 schedule된다.

2. CPU_Scheduler_Simulator

2-1. Other Existing Simulator

본 프로젝트에서 구현한 simulator에 대해서 서술하기 이전에, 기존에 구현되어 있는 simulator에 대해서 간단하게 분석하였다.
기존에 구현되어 있던 simulator는 시간에 따른 각 프로세스의 state를 분석하여 전체적인 시스템이 어떻게 돌아가고 있는 지에 대해서 보여주었다. 예를 들어, 프로세스의 현재 활동에 대해서 simulator가 묘사를 해주는데, 'I/O operation이 완료될 때까지 대기하는 중' 또는 '현재 프로세스가 CPU자원을 사용하는 중' 등 프로세스의 state를 변화시킬 수 있는 여러 event들을 simulator가 보여주면서 사용자가 시간에 따른 각 프로세스들의 state를 파악할 수 있게 구현하였다.
기존 simulator는 각 프로세스의 state를 분류하기 위해, 프로세스에게 발생하는 event들에 따라, 프로세스들의 state가 변화하게 구현되었다.

2-2. System Construction

이번 목차부터 본 프로젝트에서 본인이 구현한 simulator에 대해서 서술하였다.

CPU_scheduler의 simulation은 time_count라는 가상의 시간 단위 마다 해당 time_count 시점에서의 시스템의 상태를 파악한다.
설계한 CPU_Scheduler가 simulate될 시스템의 구성은 아래와 같다. 앞서 설명한 Process State and Scheduling의 내용을 기반으로 하여, 프로세스가 생성될 때부터, 수행되고, 완료될 때까지의 state에 따라서 시스템을 구성하였다.

1) Job Queue
CPU_scheduling 이전 단계로, 실행될 준비가 되어 있는 프로세스들을 job scheduling하여, ready queue로 보내주기 위해, 먼저 처음 생성된 프로세스들은 job queue에 위치하게 된다. 그리고 해당 arrival time에 프로세스들은 job scheduling 되어 ready queue로 이동한다. 처음 생성된 프로세스 set은 job queue에 모두 위치하게 된다. 이 상태에서 이 job queue의 copy를 생성하여, 처음 생성된 프로세스 set들이 여러 scheduling 알고리즘들을 통해 scheduling 될 수 있도록 하였다. 그리하여 모든 scheduling 알고리즘들이 동일한 프로세스 set을 가지고 수행된다.

2) Ready Queue
앞서 job queue에서 설명했듯이, 프로세스의 해당 arrival time에 job scheduling에 의해 프로세스는 ready queue로 들어간다. 매 time_count마다 해당 arrival time의 프로세스가 ready queue로 들어가므로, ready queue안의 프로세스들은 arrival time 순으로 자동 정렬된다. 만약 arrival time이 같은 프로세스들이 존재한다면, 그 프로세스들 사이에서 pid순으로, 즉, 먼저 생성되어 job queue에 들어간 프로세스가 ready queue로 먼저 들어간다.

3) Waiting Queue
프로세스가 처음 생성되면 CPU burst time과 I/O burst time이 부여된다. 이때, 각각 random한 값이 부여되는데, CPU burst time의 경우는 최소 5이상, I/O burst time은 최소 0 이상부터 부여되므로, 프로세스마다 I/O burst가 발생할 수도 있고, 안 할 수도 있다. I/O burst가 발생하는 프로세스는 매 time_count마다 1/3의 확률로 I/O request가 발생한다. 만약 I/O request가 발생하게 되면, 발생한 time_count에 해당 프로세스는 waiting queue로 이동한다. 그리고 waiting queue에서 프로세스에게 부여된 I/O burst time만큼의 시간을 머무르고, (waiting queue에 존재하는 것 자체로 I/O작업을 수행한다고 가정하였다.), 다시 ready queue로 복귀하게 된다.

4) Terminate Queue
프로세스에게 부여된 모든 CPU burst time과 I/O burst time을 모두 수행한 프로세스들은 수행이 완료되었다는 의미로 terminate state로 돌입하고, 시스템 구성상 terminate queue로 이동하게 된다. terminate queue내의 프로세스들의 수와 처음 생성한 프로세스들의 수가 같아지면, 즉, 모든 프로세스들이 수행되고 terminate되면, simulation을 종료한다. 이후, 각 scheduling 알고리즘들의 평가를 위해, terminate된 프로세스들의 평가 값들을 저장한다.

5) runningProcess
프로세스의 state들 중 running state에 해당하는 프로세스들이 저장된다. 프로세스는 runningProcess가 되어 running state에 있는 동안 CPU 작업을 수행하여, CPU burst time이 감소한다.

2-3. Process Construction
simulator에서 사용할 프로세스 셋을 구성하기 위해 프로세스 구조체를 선언해주었다. 구조체에는 pid, priority, arrival time, CPU burst time, I/O burst time, CPU remaining time, I/O remaining time, waiting time, turnaround time, 추가로 response time까지 가지고 있다. 이 중 waiting time과 turnaround time, response time은 각 scheduling 기법들의 성능을 평가하기 위해서 사용될 것이다.
생성될 프로세스 셋의 프로세스 수는 simulator 실행 시, 입력값으로 받는다. 최대 입력 개수는 10으로 설정하였으므로, 본 simulator 에서는 최대 10개의 프로세스까지 scheduling 할 수 있다.

프로세스가 생성되기 위해서 createProcess()가 실행이 된다. 프로세스 구조체가 가지는 특성 값들을 인자로 받아 하나의 프로세스 구조체를 생성한다. createProcess()에게 인자를 넘겨주는 함수가 createProcesses()이다. 이 함수를 살펴보면, 프로세스가 생성될 때 가지는 특성 값들의 결정 방식을 알 수 있다. createProcesses()에서 아래와 같은 코드를 볼 수 있는데,

createProcess(i+1, rand() % total_num + 1, rand() % (total_num + 5), rand() % total_num + 5, rand() % (total_num + 10));

즉 n개의 프로세스가 생성될 때, pid는 1부터 n까지의 값을 가지고, priority도 1부터 n까지의 값을 가지지만, pid와는 다르게 random하게 값이 부여된다. 또한 priority는 낮은 숫자일수록, 우선순위가 높다고 정의하였다. arrival time은 0부터 n+4까지의 값을 random하게 부여한다. CPU burst time은 5부터 n+4까지의 값이 random하게 부여된다. I/O burst time은 CPU burst time보다는 좀 더 넓은 범위로, 0부터 n+9까지의 값을 random하게 부여한다.
CPU remaining time, I/O remaining time은 각각 처음 인자로 받은 CPU burst time과 I/O burst time들로 초기화된다. waiting time과 turnaround time은 0으로 초기화시켜준다. response time의 경우에는, response time의 정의인 '프로세스의 submission이후, 첫 request가 발생할 때 까지 걸린 시간'을 반영하여, (해당 프로세스가 처음으로 수행된 time_count) - (해당 프로세스의 arrival time)으로 설정하였다. response time의 초기 값은 -1이다.

2-4. Predefined Conditions

1) I/O request 발생
앞서 '2-2 System Construction'에서 설명하였다시피, 매 time_count마다 1/3의 확률로 I/O burst가 발생하도록 설정하였다. 이를 좀 더 구체적으로 설명하면, 한 time_count시점에 running state로 돌입하여, runningProcess가 된 프로세스의 현 작업 수행 상태를 3가지로 분류하였다. 첫째로, 프로세스가 CPU remaining time, I/O remaining time 모두 0이하인, CPU burst와 I/O burst가 모두 수행된 상태이면, terminate된다. 둘째로, CPU remaining time, I/O remaining time 모두 0보다 큰, CPU burst와 I/O burst가 모두 완료되지 않은 상황에서는 1/3의 확률로 I/O burst가 발생한다. 셋째로, CPU burst는 모두 수행되었는데, I/O burst가 아직 완료되지 않은 상황에서는 무조건 I/O burst를 발생시킨다. 이것은 정해진 시간 안에 모든 프로세스들을 complete시키고 simulation을 종료하기 위해서이다. 계속 1/3의 확률로 I/O burst를 발생시키면 정해진 시간 안에 프로세스가 terminate되지 않을 수도 있기 때문이다.

2) Time Quantum / elapsedTime
Round Robin 알고리즘에서 사용할 time quantum은 3으로 설정하였다. 또한 시스템에는 elapsedTime, 즉 경과 시간이라는 변수가 있는데, runningProcess가 교체될 때마다, 즉 이전 time_count시점과 다른 프로세스가 runningProcess로 존재할 때, 0으로 초기화된다. elapsedTime을 통해 프로세스가 얼마나 연속적인 time_count만큼 running state로 머물렀는지를 알 수 있다.

2-5. Scheduling Algorithms in Simulation

본 프로젝트에서 구현한 scheduling 알고리즘은 총 6가지이다. FCFS, non-preemptive SJF, preemptive SJF, non-preemptive Priority, preemptive Priority, Round Robin을 구현하였

다.

1) FCFS
FCFS는 앞서 서론에서 서술하였다시피, 먼저 도착하여 submission된 프로세스부터 CPU자원을 할당하여 준다. job queue의 프로세스들이 job scheduling을 통해 해당 arrival time에 ready queue로 이동한다. 그렇게 되면 자동으로 ready queue에서의 프로세스들은 arrival time 순서로 정렬된다. FCFS 알고리즘은 이렇게 매 time_count마다 순차적으로 ready queue에 도착하는 프로세스들 중에서 가장 먼저 도착한 프로세스를 candidate(scheduling 될 후보 프로세스를 의미함)로 저장한다. 그리고 preemptive 상황을 FCFS에서는 고려하지 않으므로, candidate로 저장된 ready queue의 첫 번째 프로세스를 scheduling한다.

2) SJF
SJF에서는 프로세스의 특성 값인 CPUremainingTime을 scheduling을 위해 사용한다. SJF에서의 candidate 프로세스는 처음에는 FCFS와 마찬가지로 ready queue의 첫 번째 프로세스로 설정한다. 이후 ready queue내의 다른 프로세스들과 CPUremainingTime들을 비교하여 candidate 프로세스를 update해 나간다. SJF알고리즘의 정의에 의해, CPUremainingTime이 적은 프로세스가 candidate로 선정이 되는데, 만약 CPUremainingTime이 같은 프로세스들이 비교가 된다면, arrival time을 비교한다. CPUremainingTime이 같은 상황에서 먼저 도착한, 즉, arrival time이 적은 프로세스가 candidate로 선정된다. 만약 arrival time까지 같은 상황이라면, 굳이 candidate를 update하지 않는다.
위 과정을 수행하여 candidate 프로세스를 설정한 후, preemptive와 non-preemptive 상황을 분류하여 남은 scheduling 과정을 진행한다.
 -non-preemptive
preempt을 고려하지 않는 상황이라면 이전 time_count시점부터 수행 중인 프로세스(runningProcess)가 terminate되거나 I/O burst가 발생하지 않는 한 다른 프로세스가 수행될 수 없고, 대기하여야 한다. 이후에 runningProcess가 NULL인, 즉 이전에 수행 중이던 프로세스가 terminate되거나 I/O burst가 발생하여, 현재 수행중인 프로세스가 없다면, 앞서 update해주었던 candidate 프로세스를 scheduling한다.
 -preemptvie
위와 반대로 preempt을 고려하는 상황에서는 이전 time_count시점부터 수행 중인 프로세스 (runningProcess)와 앞서 update해주었던 candidate프로세스 간에 CPUremainingTime을 비교하여야 한다. 이때 candidate를 update해주었던 방식과 유사하게 CPUremainingTime이 같다면, arrival time을 비교하여 더 빨리 도착한 프로세스를 scheduling한다. 만약 arrival time까지 같은 상황이라면 굳이 preemt을 발생시켜 불필요한 context switch overhead를 야기하지는 않는다. 즉 preempt없이 수행중이던 프로세스를 계속 수행한다. preempt이 발생하게 되면, 수행 중이던 프로세스는 다시 ready queue의 맨 뒤로 보내지고, candidate 프로세스가 최종 scheduling된다.

3) Priority
Priority 알고리즘에서는 프로세스의 특성 값인 priority를 scheduling을 위해 사용한다. 앞서 설명한 SJF와 유사한 방식으로 candidate를 update한다. 단지 비교 지표가 priority로 바뀐 것일 뿐이다. 사전에 priority가 낮을수록, 우선순위가 높다고 설정하였으므로, scheduling 상황에서 ready queue에 존재하는 프로세스들 중 priority값이 작은 프로세스를 candidate로 update한다. 물론 priority가 같다면 arrival time을 비교하여, 더 빨리 도착한 프로세스를 선택한다. 도착 시간까지 같다면 역시 굳이 update해주지는 않는다.
위 과정을 수행하여 candidate 프로세스를 설정한 후, preemptive와 non-preemptive 상황을 분류하여 남은 scheduling 과정을 진행한다.

-non-preemptive
앞서 언급하였다시피, preempt을 고려하지 않는 상황이라면 이전 time_count시점부터 수행 중인 프로세스 (runningProcess)가 terminate되거나 I/O burst가 발생하지 않는 한 다른 프로세스가 수행될 수 없고, 대기하여야 한다. 이후에 runningProcess가 NULL인, 즉 이전에 수행 중이던 프로세스가 terminate되거나 I/O burst가 발생하여, 현재 수행중인 프로세스가 없다면, 앞서 update해주었던 candidate 프로세스를 scheduling한다.
-preemptvie
위와 반대로 preempt을 고려하는 상황에서는 이전 time_count시점부터 수행 중인 프로세스 (runningProcess)와 앞서 update해주었던 candidate프로세스 간에 priority를 비교하여야 한다. 이때 candidate를 update해주었던 방식과 유사하게 priority가 같다면, arrival time을 비교하여 더 빨리 도착한 프로세스를 scheduling한다. 만약 arrival time까지 같은 상황이라면 굳이 preemt을 발생시켜 불필요한 context switch overhead를 야기하지는 않는다. 즉 preempt없이 수행중이던 프로세스를 계속 수행한다. preempt이 발생하게 되면, 수행 중이던 프로세스는 다시 ready queue의 맨 뒤로 보내지고, candidate 프로세스가 최종 scheduling된다.

4) Round Robin
Round Robin 알고리즘은 앞서 서론에서 서술하였다시피, FCFS와 기본적인 scheduling 기준이 같으므로, candidate 프로세스의 선정 방식이 FCFS와 같다. scheduling이 이루어지는 time_count시점에서 ready queue의 첫 번째 프로세스를 candidate로 설정한다.
다만, Round Robin에서는 프로세스의 수행 경과 시간이 time quantum에 도달하면 context switch가 발생한다. 그러므로 프로세스의 연속된 수행 경과 시간을 측정하기 위해 사전에 정의해놓은 elapsedTime변수를 사용한다.
이전 time_count시점부터 수행 중이던 프로세스 (runningProcess)의 elapsedTime이 time quantum에 도달하였다면, context switch가 발생하고, 수행 중이던 프로세스는 다시 ready queue의 맨 뒤로 보내진다. 그리고 미리 선정해둔 candidate 프로세스를 scheduling한다.
수행 중이던 프로세스의 elapsedTime이 time quantum에 도달하지 않은 상황이라면, Round Robin에서는 preemptive 방식을 고려하지 않으므로 수행 중이던 프로세스를 그대로 계속 scheduling하여 수행한다.

2-6. Simulation Progress
앞서 서술한 구성과 구현된 알고리즘들을 바탕으로 각 scheduling 알고리즘을 가지고 CPU_Scheduler의 simulation을 진행할 것이다.

simulation의 진행 단계는 크게
1. 사용할 자료구조들 초기화
2. scheduling할 프로세스 생성
3. simulation에서 사용할 알고리즘을 선택하여 simulation시작
4. simulation 종료 후 산출된 성능 평가 값들 display
5. 다음 simulation 진행을 위해 사용한 자료구조들 초기화
로 총 5가지 단계로 나눌 수 있다.

1). 사용할 자료구조들 초기화
먼저 initialize 함수를 통해 simulation에서 사용할 state queue들을 초기화 해준다.

2). scheduling할 프로세스 생성
1단계에서 simulation에서 사용할 시스템 자료구조들이 사용될 준비가 되면, 2단계에서는

사용자의 입력 값(scheduling할 프로세스 수)을 받아 프로세스를 생성한다. 앞서 '2-3 Process Construction'에서 서술하였다시피, createProcess()함수는 프로세스의 특성 값을 인자로 받아서 프로세스 구조체 하나를 생성하고, job queue에 해당 프로세스를 보낸다. createProcesses()는 사용자의 입력 값인 생성할 전체 프로세스 수를 인자로 받은 다음 그 수만큼 createProcess()함수를 수행한다. 그러면 자연스럽게 job queue가 완성되게 된다.
'2-2 System Construction'에서 서술하였다시피, 본 프로젝트에서는 job queue의 copy를 생성하여 여러 알고리즘을 동일한 프로세스 set으로 진행할 수 있게 하였다. 그러므로 createProcess()함수에서는 완성된 Copy_JQ()함수를 통해서 앞서 완성된 job queue의 복사본을 생성한다. 그리고 완성된 job queue를 display하여, 사용자가 생성된 프로세스들의 특성 값들을 볼 수 있게 하였다.

3). simulation에서 사용할 알고리즘을 선택하여 simulation시작
simulation을 실질적으로 진행하는 simulation()함수는 사용할 알고리즘과 preemptive방식 사용 여부, time_quantum, 그리고 simulation이 진행되는 시간을 나타내는 time_count를 인자로 받는다. 본 프로젝트에서는 time_count를 100으로 설정하였다. 즉 본 프로젝트의 simulation은 시간단위 time_count가 0에서 100이 될 때까지 진행된다. time_quantum은 3으로 설정하였다.
simulation()에서는 앞서 2단계에서 생성한 복사본을 scheduling에 사용하기 위해 load한다. loadCopy_JQ()함수는 먼저 실제 job queue를 초기화 시켜준 다음 미리 생성해둔 job queue의 copy본을 그대로 실제 job queue에 복사해준다. 다음 simulation을 진행할 때도 위와 같은 방식을 사용함으로, 계속해서 동일한 프로세스 set을 사용할 수 있다.
이제 scheduling할 프로세스 set이 사용할 준비가 다 되었으므로, 실제로 time_count를 증가시키면서 매 시간 단위마다 scheduling을 진행한다. 매 시간 단위마다 RunSystem()함수가 시행된다.
RunSystem()함수는 실질적으로 매 time_count마다 scheduling이 이루어지고 있는 시스템의 상태를 update해준다. 현재 time_count값과 사용하는 알고리즘, preemptive방식 사용 여부, time_quantum을 인자로 받는다.
RunSystem()함수에서는 인자로 받은 현재 time_count값과 job queue에 존재하는 프로세스들의 arrival time값을 비교하여 도착 시간이 된 프로세스를 ready queue로 보내주는 job scheduling이 먼저 수행된다. 만약 도착시간이 같은 프로세스들이 존재한다면, 자동으로 pid순으로(생성된 순으로) job scheduling된다.
RunSystem()함수에서는 앞서 설명한 알고리즘을 통해 실제 scheduling이 이루어지는 schedule()함수가 사용된다. schedule()함수는 실행할 알고리즘과 preemptive 여부, time_quantum을 인자로 받고, 인자로 받은 알고리즘으로 scheduling하여 CPU자원을 할당받을 선별된 프로세스를 return한다.
schedule()함수를 통해 선별된 프로세스는 runningProcess로 저장되고, running state로 돌입한다. RunSystem()함수에서는 위의 과정으로 선별된 runningProcess의 response time의 값이 아직 update되지 않은 초기 값 –1이라면, 즉, 해당 프로세스가 submission된 이후 첫 request가 발생한 것이라면, 해당 프로세스의 response time을 현재 time_count – arrival time으로 update해준다.
RunSystem()함수에서는 scheduling 뿐만 아니라, 실제 시스템이 진행되고 있는 상황이고, 매 time_count마다 시스템의 상태를 update해준다. 그러므로 매 time_count마다 시스템 내부에 존재하는 모든 프로세스들의 상태도 update해줘야 한다. ready queue에 존재하는 모든 프로세스들의 waiting time과 turnaround time을 1씩 증가 시켜준다. 또한 waiting queue에 존재하는 프로세스들은 I/O burst를 수행 중이므로, I/O remaining time을 감소시켜주고, turnaround time은 증가시켜준다. 만약 I/O remaining time이 0에 도달하여, 작업이 완료되면, 해당 프로세스는 다시 ready queue로 돌아간다.
또한 수행 중인 프로세스인 runningProcess의 CPU remaining time을 감소시켜주고,

turnaround time은 증가시켜준다. 그리고 수행 경과 시간인 elapsedTime역시 증가시켜준다.
RunSystme()함수에서는 runningProcess의 상태를 파악하여 terminate또는 random하게 I/O burst를 발생시킨다. 자세한 설명은 '2-4. Predefined Conditions'에서 서술하였으므로 생략한다.
위의 과정을 통해 RunSytem()함수가 반복적으로 실행되다 보면, 프로세스들이 계속 terminate되고, 이후 모든 프로세스들이 terminate되면 simulation이 마무리된다. simulation()은 마지막으로 사용한 알고리즘에 대한 평가 값들을 저장하고 다음 simulation을 진행하기 위해 자료구조들을 모두 초기화하고 종료된다.

4). simulation 종료 후 산출된 성능 평가 값들 display
모든 알고리즘에 대한 simulation이 종료되고 나면, simulation()함수에서 Partial_Evaluate()를 통해 저장해놓은 평가 값들을 Total_Evaluate()함수를 통해 출력한다.

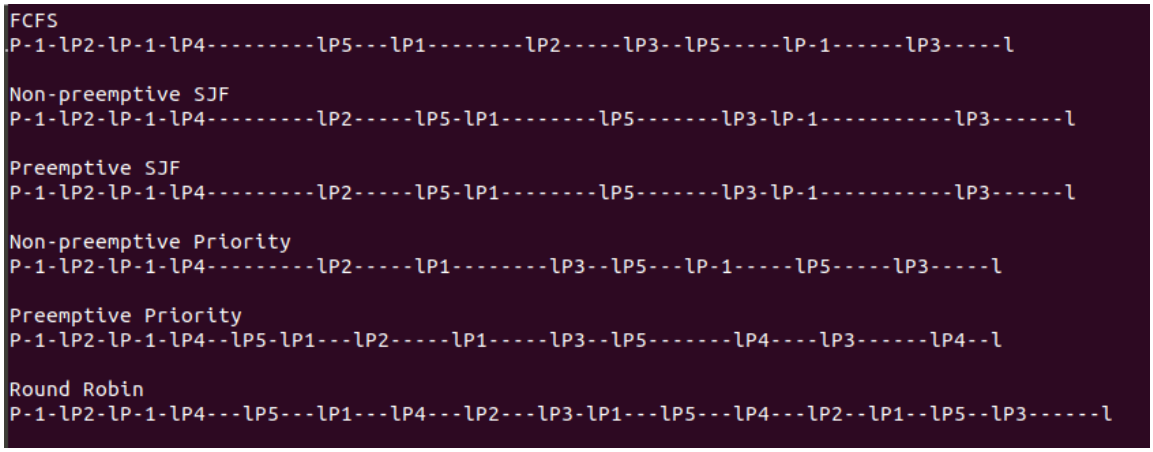
5). 다음 simulation 진행을 위해 사용한 자료구조들 초기화
모든 알고리즘에 대한 simulation과 evaluation까지 종료되고 난 후에는 사용한 모든 자료구조들을 초기화 시키고 사용한 메모리를 반환한다.

3. Results Analysis and Review of Project

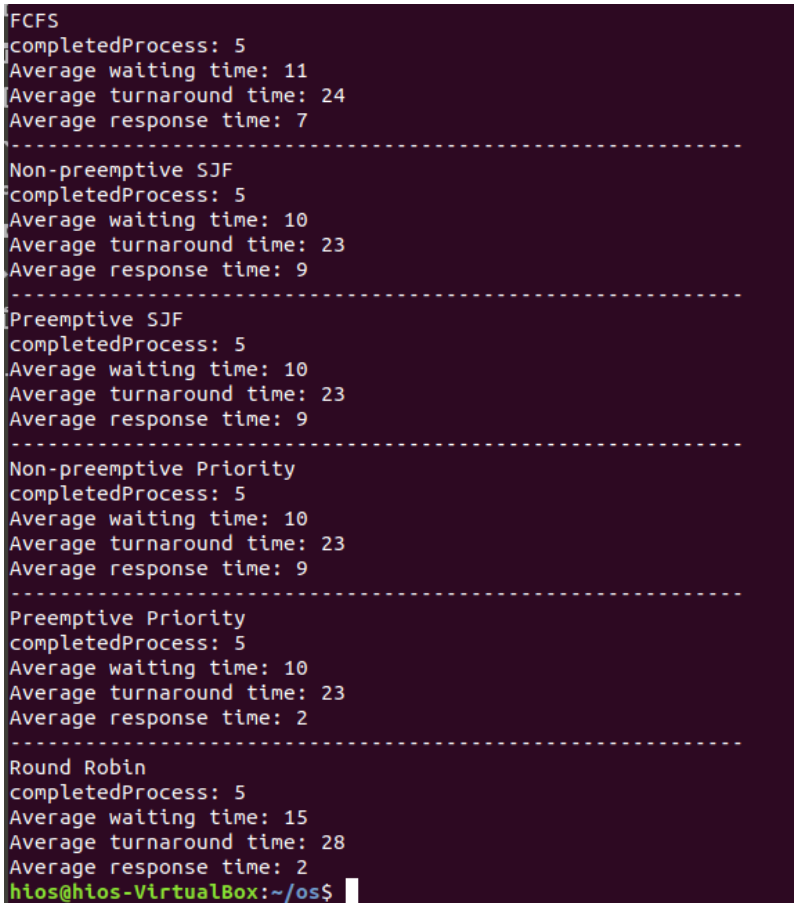
3-1 Comparison Analysis of Algorithms

```
hios@hios-VirtualBox:~/os$ gcc -o reporttest OS_CPUScheduler.c
hios@hios-VirtualBox:~/os$ ./reporttest 5
프로세스 : 5개
pid    priority    arrival_time    CPU burst    IO burst
-------------------------------------------------------------
  1          1              6           8           0
-------------------------------------------------------------
  2          1              1           6           7
-------------------------------------------------------------
  3          1              9           8          11
-------------------------------------------------------------
  4          5              3           9           0
-------------------------------------------------------------
  5          3              5           8           5
```

위 그림은 리눅스 환경의 가상머신에서 본 프로젝트 simulator를 실행시킨 결과 중 생성된 프로세스들을 display한 것이다. terminal을 보면 알 수 있듯이 simulator를 실행시킬 때 사용자는 scheduling할 총 프로세스 수를 입력한다. 본 예시에는 5개의 프로세스를 scheduling 하였다.
실행과 동시에 job queue에 pid의 오름차순으로 프로세스가 생성되어 들어갔다. pid를 제외한 모든 특성 값들이 random하게 할당되었다. 본 예시에서는 I/O burst를 수행하는 프로세스가 총 3개가 되었다.



위 그림은 본 프로젝트에서 구현한 총 6가지 scheduling 알고리즘을 바탕으로 scheduling을 한 결과를 Gantt Chart 형식으로 display한 것이다. 프로세스가 수행된 time_count를 'l'으로 구분하였다. 그리고 각 time_count마다 수행되는 프로세스를 'Pn' (n은 pid) 형식으로 labeling해주었다. P-1은 현재 수행되는 프로세스가 없는 CPU의 idle 상태를 나타낸다. 각 time_count를 '-'으로 표현해주었다. 예를 들어 'lP5---l' 3의 연속된 time_count동안 pid=5인 프로세스 P5가 수행되었다는 것을 의미한다.
총 3개의 프로세스 I/O burst를 할당받았으므로, I/O burst 수행에 의해서 time_count 중간마다 CPU의 idle이 발생함을 위 Gantt Chart를 통해 관찰할 수 있다.
모든 프로세스들이 terminate되어 scheduling 종료되면 'l'을 삽입하여 Gantt Chart를 마무리한다. Chart를 통해 알 수 있듯이, 모든 프로세스들이 자신의 CPU burst와 I/O burst를 정상적으로 완료하여 terminate된다.



위 그림은 Total_Evaluation()함수를 통해 각 알고리즘 별 평가 지표들을 display한 것이다. 모든 알고리즘 simulation에서 총 5개의 프로세스들이 정상적으로 완료되었다.
average waiting time같은 경우, 처음에는 SJF가 가장 optimal한 값이 나올 것이라고 생각했는데, 본 simulation의 결과로 살펴보았을 때는 다른 알고리즘에 비교하였을 때, 특별히 optimal 하다고 보이지는 않는다. Round Robin과는 어느 정도의 차이를 보이지만, Priority와는 같은 값을 보이면서, 예상과 달리, SJF가 average waiting time 면에서 가장 optimal하지가 않았다. 특히 FCFS와도 큰 차이를 보이지가 않는데, 원인을 생성된 프로세스 set을 통해 분석해보자면,

```
hios@hios-VirtualBox:~/os$ gcc -o reporttest OS_CPUScheduler.c
hios@hios-VirtualBox:~/os$ ./reporttest 5
프로세스 : 5개
pid    priority    arrival_time    CPU burst    IO burst
-------------------------------------------------------------
  1          1              6           8           0
-------------------------------------------------------------
  2          1              1           6           7
-------------------------------------------------------------
  3          1              9           8          11
-------------------------------------------------------------
  4          5              3           9           0
-------------------------------------------------------------
  5          3              5           8           5
```

위에서 알 수 있듯이, 생성된 프로세스들의 CPU burst time이 큰 차이가 없다. 특히 가장 먼저 도착한 P2가 가장 작은 CPU burst time을 가지고, 나머지 프로세스들의 CPU burst time은 큰 차이가 없으므로, FCFS와 SJF의 average waiting time에서 큰 차이가 없었을 것이라 판단하였다.
average turnaround time은 예상대로 Round Robin이 가장 컸다. CPU burst time을 할당해줄 때, 최소 5이상으로 할당해주는 policy가 영향을 줬을 거라고 생각한다. Round Robin의 average turnaround time을 optimal하게 해주려면, 한 time quantum 안에 자신의 CPU burst를 완전히 수행하는 것에 가깝게 time quantum을 맞춰줘야 하는데, simulation을 시작할 때, 사전에 정의한 time quantum값이 3으로, 모든 프로세스들의 CPU burst time을 채우기에는 부족하다고 생각한다. 결과적으로 Round Robin의 average turnaround time이 not optimal 하게 도출되었다.
그에 반해 예상대로 average response time 측면에서는 Round Robin이 Preemptive Priority와 함께 가장 optimal 하였다. 모든 프로세스가 자신에게 한번 할당된 time quantum을 모두 소비하면 preempt되므로, average response time 측면에서는 가장 optimal하다고 결론 내렸다.
preemptive와 non-preemptive로 구분되는 알고리즘에서의 차이점을 분석해보면, SJF같은 경우, 두 알고리즘의 Gantt Chart가 동일하였다. 그 원인으로는 아마도 도착 시간이 상이한 프로세스들 간에 처음 주어진 CPU burst time에 큰 차이가 없어서 preempt일어날 상황이 없었다고 분석하였다. Priority같은 경우, average response time측면에서 preempt 여부가 큰 영향을 주었는데, 일반적으로 뒤늦게 도착한 프로세스가 우선순위가 더 높아서 preempt가 발생하여 첫 response가 발생하면, response time이 적어진다고 판단하였다.

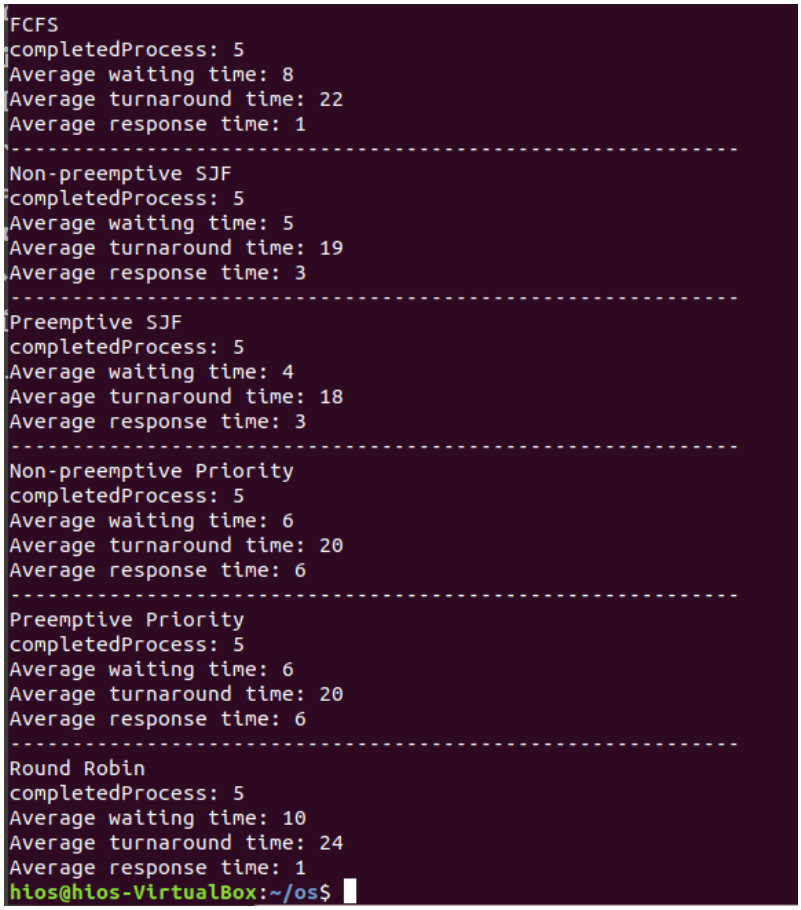
위와 다른 프로세스 set으로 simulation해보기 위해 추가로 simulation을 더 실행해보았다.

```
hios@hios-VirtualBox:~/os$ gcc -o report OS_CPUScheduler.c
hios@hios-VirtualBox:~/os$ ./report 5
프로세스 : 5개
pid     priority     arrival_time     CPU burst     IO burst
-----------------------------------------------------------
  1           5               8           6           6
-----------------------------------------------------------
  2           3               1           9           7
-----------------------------------------------------------
  3           3               0           6           1
-----------------------------------------------------------
  4           5               1           6          13
-----------------------------------------------------------
  5           4               1           5          10
```

프로세스 수는 5로 동일하며, random하게 특성 값들을 부여하였으므로, 앞서 실행한 프로세스 set과는 확연히 다른 프로세스 set으로 scheduling이 진행되었다.

```
FCFS
P3-lP2-lP4---lP5-lP3-----lP1----lP2--------lP5----lP4---lP1-l

Non-preemptive SJF
P3-lP5-lP3-----lP4--lP1-lP2-lP-1-lP5----lP1-----lP2--------lP4---l

Preemptive SJF
P3-lP5-lP3-----lP4--lP1-lP2-lP-1-lP5----lP1-----lP2-lP4----lP2------l

Non-preemptive Priority
P3-lP2-lP3-----lP5--lP2--------lP4-lP1-lP5---lP-1---lP1-----lP-1-lP4----l

Preemptive Priority
P3-lP2-lP3-----lP5--lP2--------lP4-lP1-lP5---lP-1---lP1-----lP-1-lP4----l

Round Robin
P3-lP2-lP4---lP5-lP3---lP1-lP2---lP3--lP2---lP5---lP1---lP4---lP2--lP5-lP1-l
```

새롭게 실행한 simulation의 Gantt Chart이다.



새로운 프로세스 set으로 실행한 결과, average waiting time 측면에서 이번에는 SJF가 가장 optimal 하였다. 하지만 Priority에 비해 확연하게 optimal하다고 볼 수가 없었다. CPU burst time의 범위를 좀 더 넓게 설정하여, 프로세스들 간의 CPU burst time 차이를 크게 만들어 주면 SJF의 좀 더 상대적인 optimal을 관찰할 수 있을 것 같다고 분석하였다. 현재 5부터 9까지로 설정된 CPU burst time 범위는 프로세스들 간의 CPU burst time의 확연한 차이를 보여주기 어렵다고 판단하였다.

average turnaround time측면에서는 예상대로 새로운 프로세스 set으로의 실행에서도 Round Robin이 worst한 알고리즘으로 나타났다. 원인은 앞서 시행한 simulation과 같이 time quantum이 프로세스들의 CPU burst time에 비해 턱없이 부족하기 때문이라고 판단하였다.

그에 반해 average response time측면에서는 역시 Round Robin 알고리즘이 가장 best한 알고리즘으로 나타났다. 하지만 특이하게도 FCFS 역시 best하다고 판단되었는데 원인으로는 I/O interrupt의 발생이 Round Robin과 유사한 패턴으로 이루어진 것 때문이라고 분석하였다. FCFS와 Round Robin은 도착시간을 기준으로 scheduling한다는 공통점이 있는데, 이러한 기본 전제 하에 유사한 I/O interrupt 발생 패턴이 겹쳐지면서 response time이 유사한 경향을 보여주었다고 분석하였다.

preemptive와 non-preemptive 여부는 SJF와 Priority들의 optimal 여부에 크게 영향을 주지는 못했다.

3-2 Review of Project

이번 프로젝트를 수행하면서 막연하게 느껴졌던 CPU scheduling을 좀 더 명확하게 이해할 수 있게 되었다. 수업 시간에 학습했던 내용을 토대로 알고리즘과 시스템을 구현하고, 최대

한 real system 환경에 가깝게 randomness를 도입하였지만, 본 프로젝트의 simulator로는 아직 구현하지 못한 real system 요인들이 존재하는 것 같다. 예를 들면, I/O request가 발생하여 프로세스가 waiting queue에 진입했을 때, I/O burst의 수행, 즉 I/O device의 할당이 즉각적으로 이루어진다고 가정하고 simulator를 구현하였는데, 사실 real system에서는 waiting queue에서 I/O device의 할당을 대기하여야 하고, 즉각적으로 I/O device의 할당이 이루어지는 것은 굉장히 ideal한 환경이라고 생각한다. 추후에 이 점을 보완하여 좀 더 real system에 가까운 simulator를 구현해볼 생각이다.
또한 추가적으로 CPU utilization 또한 관찰해보고 싶다. 특히 job scheduling이 이루어질 때, CPU bound 프로세스와 I/O bound 프로세스가 적절하게 distributed되는지의 여부가 CPU utilization에 영향을 준다는 것을 관찰해보고 싶다.
또한 textbook의 exercise 문제에 묘사되어있는 다양한 scheduling 알고리즘들에 대해서도 추가적으로 구현해보고 싶다. 특히 현재 프로젝트에서는 preemptive의 여부에 따른 평가 지표의 확연한 차이가 드러나지 않아 아쉽다. 그래서 preemptive-Round Robin 등 preemptive 개념을 추가한 또 다른 알고리즘들을 구현하여, preemptive의 성능 향상 기여도를 좀 더 면밀하게 관찰해보고 싶다.
본 프로젝트를 수행할 때, 어려웠던 점은 C programming이였다. 신입생 이후로는 C 언어로 코딩 작업을 해볼 기회가 없어서 C 언어를 다루는데 어려운 점이 있었다. C 언어 대해서 review해볼 수 있는 좋은 기회였다. 그리고 평소에는 다루지 않았던 리눅스 환경에서의 코딩도 새로운 경험이었다. 익숙하지 않은 새로운 환경에서의 프로그래밍은 프로그래밍에 대한 시야를 넓힐 수 있던 좋은 기회였다.