CPU Scheduling Simulator

2021320086 김주연

1. 서론

CPU 스케줄링은 다중 프로그래밍 환경에서 컴퓨터의 생산성을 높여주는, 운영체제에게 있어 핵심적이고 중요한 기술이다. CPU 스케줄링은 CPU 에서 실행되는 프로세스를 바꿀 수 있게 해주어, 다른 프로세스를 시작하기 위해 한 프로세스가 끝나길 기다려야 했던 비효율을 줄여 주었다.

CPU 코어가 하나뿐인 시스템은 한 번에 한 프로세스만 실행할 수 있다. 다중프로그래밍이 도입되며 CPU 활용률을 최대화하기 위해서는 어떤 프로세스든, 항상 프로세스가 실행되고 있게끔 CPU 를 스케줄링하는 것이 중요해졌다. 원리는 간단하다. 모든 프로세스는 기다리기 전까지 실행되어야 한다. 기다림은 보통 I/O 요청이 마무리될 때까지 이어진다. 기다리는 시간 동안 CPU 가 쉬고 있는 상태는 큰 비효율이다. 프로세스가 기다리는 동안, 같은 메모리에 저장된 다른 프로세스가 CPU 를 점유하면 CPU 활용률을 높일 수 있다. 이때 어떤 프로세스가 실행될지 선택하는 CPU 스케줄링은 운영체제의 근본적인 기능이 되었다. 따라서, CPU 스케줄링은 운영체제 디자인에 핵심적인 역할을 한다.

CPU 스케줄러는 CPU 가 쉬는 상태일 때, 메모리에 있는 프로세스 중 실행될 프로세스를 선택한다. 어떤 프로세스를 선택할지는 다양한 스케줄링 알고리즘에 따라 결정된다. 컴퓨터의 자원은 한정적이기 때문에, CPU 활용을 최대화하며 실행시간이 너무 길어지지 않게 하는 것이 중요하다. 이번 과제에서는 FIFO(First-In-First-Out), non-preemptive SJF(Short Job First), preemptive SJF, non-preemptive priority, preemptive priority, Round Robin 6 가지 스케줄링 알고리즘을 구현하고, waiting time 과 turnaround time 을 계산하여 각 알고리즘을 평가했다.

이번 과제에서는 I/O 작업 구현을 배제하여 순수하게 CPU 집중적인 프로세스들의 스케줄링에 집중했다. 각 시간 단위별 CPU 할당 상태를 Gantt Chart 로 시각화하였으며, 도착 시간, CPU 버스트 시간, 우선순위 등을 랜덤으로 생성하였다.

2. 본론

본 프로젝트를 시작하기에 앞서, 기존에 공개된 여러 CPU 스케줄링 시뮬레이터를 검토하였다. Vercel 과 Netlify 같은 어플은 사용자가 직접 프로세스의 정보를 입력하도록 되어 있었으며,

시각화에 집중하는 모습을 보였다. 순천향대 과제의 시뮬레이터는 알고리즘별 구현을 모아두는 디렉터리를 따로 두고, 실제 실행 로직은 하나의 c 파일에서 관리하여 모듈화된 코드 구조를 보였다. 또한, HRN(Higehst Response-Ratio Next) 알고리즘을 추가로 구현하였다. 과제 요구 사항에 맞춰 본 시뮬레이터는 순수 콘솔 환경에서 동작하면서도 모든 로직을 모듈화하여 확장성과 이식성을 보장하도록 설계하였다.

시뮬레이터의 전체 구성도는 크게 네 가지 모듈로 이루어진다. 첫째, Create_Process() 모듈은 rand() 함수를 이용해 프로세스 ID, 도착 시간, CPU 버스트 시간, 우선순위 등을 생성하고, ready_queue 에 삽입한다. 둘째, Config() 모듈은 ready_queue 와 waiting_queue 를 초기화하며, Round Robin 알고리즘을 위한 타임 퀀텀을 포함한 시뮬레이션 환경을 설정한다. 셋째, Schedule() 모듈에서는 FCFS, SJF(비선점/선점), Priority(비선점/선점), RR 총 여섯 가지 알고리즘을 switch-case 구조로 구현하여 실행한다. 마지막으로 Evaluation() 모듈은 각 알고리즘 실행 후 프로세스별 시작 및 종료 시간을 바탕으로 평균 대기 시간(AWT)과 평균 Turnaround Time(ATAT)을 계산·출력한다.

Create_Process() 함수는 단일 배열과 포인터 기반 큐를 활용하여 메모리 할당 없이 단순하면서도 효율적으로 프로세스의 속성을 관리한다. Config() 함수는 시뮬레이션 전역 변수를 초기화하고, 지정된 타임 퀀텀을 scheduler 구조체에 설정함으로써 Round Robin 실행 시점을 제어할 수 있도록 한다. Schedule() 함수 내부에서는 각 알고리즘별 핵심 로직만 교체하고, 프로세스 선택 조건만 변경하도록 구현하여 코드 중복을 최소화하였다.

시뮬레이터에서 가장 기본이 되는 FCFS(First Come, First Served) 알고리즘은 ready_queue 에서 순차적으로 프로세스를 꺼내 한 번 시작되면 끝까지 실행시키는 방식으로 구현되었다. 구체적으로 dequeue(&ready_queue)로 꺼낸 프로세스의 도착 시각(arrival time)이 현재 시각(current_time)보다 뒤처져 있으면 곧바로 current_time 을 그 도착 시각으로 이동시키고, burst time 만큼 current_time 을 증가시켜 종료 시각(end_time)을 기록한다. 이 과정에서 별도의 우선순위 비교나 남은 실행 시간 관리 없이, 단순히 큐에 들어온 순서대로 CPU 를 할당하므로 코드가 가장 직관적이고 간결하다.

다음으로 SJF(Shortest Job First) 비선점 방식은 준비 큐를 단순 선형 구조 대신 "남은 실행 시간(remaining time)이 짧은 순"으로 꺼낼 수 있는 최소 힙(min-heap)으로 관리한다. 먼저

모든 프로세스를 도착 시각(arrival time) 기준으로 정렬한 뒤, 현재 시각까지 도착한 프로세스들을 힙에 push_heap(heap, proc, by_burst) 형태로 삽입한다. 힙에서 pop_heap() 으로 꺼낸 프로세스는 burst 가 가장 짧은 녀석이므로, start_time 과 end_time 을 기록한 다음 한 번에 실행을 마친다. 만약 힙이 비어 있으면 다음 프로세스의 도착 시각까지 current_time 을 점프하는 방식으로 구현하였다.

SJF Preemptive(또는 Shortest Remaining Time First) 방식은 매 시간 단위로 남은 실행 시간이 가장 짧은 프로세스를 선점해서 실행하는 형태다. 구현 시에는 비선점형 SJF 와 같은 최소 힙을 사용하되, 프로세스마다 remaining 필드를 추가해 남은 버스트 시간을 추적한다. 루프 안에서 현재 시각까지 도착한 프로세스들을 힙에 넣고, top_heap() 으로 꺼낸 프로세스의 remaining-- 연산을 수행한 뒤 current_time++ 을 한 단위씩 증가시킨다. 만약 remaining 이 0 이 되면 그때야 end_time 을 기록하고 힙에서 제거한다.

우선순위 기반 스케줄링인 Priority Non-Preemptive 는 arrival time 기준으로 도착한 프로세스를 우선순위(priority) 값에 따라 우선순위 큐에 push_pq() 한 다음, pop_pq()으로 가장 높은(숫자가 가장 낮은) 우선순위 프로세스를 한 번에 실행하도록 구현하였다. Priority Preemptive 버전은 이 구조에 SJF Preemptive 와 유사한 선점 로직을 결합해, 매 시간 top_pq()으로 현재 가장 높은 priority 프로세스를 실행하고 remaining-- 후 다시 큐에 남으면 재삽입하는 방식으로 작성했다.

마지막으로 Round Robin(RR) 은 classic circular queue 방식으로 구현하였다. 도착 순서대로 ready_queue 에 enqueue 한 뒤, dequeue() 로 꺼낸 프로세스에게 time_quantum 만큼 CPU 를 할당하고 남은 시간이 있으면 다시 enqueue() 해서 큐 뒤로 보내는 형태다. 실행 중 새로 도착한 프로세스는 언제든 dequeue 가능한 위치로 삽입되며, 각 start_time, end_time 은 첫 실행 시각과 remaining 이 0 이 된 순간에 기록한다. 이처럼 모듈별로 함수(fcfs(), sjf_nonpreemptive(), sjf_preemptive(), priority_nonpreemptive(), priority_preemptive(), round_robin())을 분리해 두었기 때문에, Schedule(int algo) 의 switch-case 분기만으로 손쉽게 원하는 알고리즘을 실행하고 비교할 수 있다.

Evaluation() 함수는 각 프로세스의 start_time 과 end_time 을 이용하여 총 대기 시간과 총 응답 시간을 집계하며, 이를 프로세스 수로 나누어 성능을 계산한다. 이를 위해 double 형으로

변환하여 종합적인 비교가 가능하도록 하였다. 이를 통해 FCFS, SJF, Priority, RR 방식 간의 성능 차이를 객관적으로 분석할 수 있다.

main()은 전체 실행 흐름을 제어하는 역할을 한다. 먼저 Config()를 호출해 ready_queue 와 waiting_queue 를 초기화하고, RR 알고리즘에서 사용할 time_quantum 값을 설정한다. 이후, Create_Process()를 통해 N 개의 프로세스를 난수 기반으로 생성하여 준비 큐에 삽입하고, 실제 시뮬레이션을 수행할 스케줄링 알고리즘을 선택한다. Algo 매개변수는 FCFS, SJF, Priority, RR 등의 숫자 상수로 맵핑되어 있으며, 반복문이나 사용자 입력을 통해 다양한 알고리즘을 순차 실행해 비교할 수 있게 구성되어 있다. 스케줄링이 완료되면 Evaluation() 함수를 호출해 각 프로세스의 start_time 과 end_time 을 기반으로 평균 대기 시간(AWT)과 평균 Turnaround Time(ATAT)을 계산·출력한다. 마지막으로 print_gantt_chart() 같은 보조 함수를 통해 각 알고리즘별 Gantt 차트를 콘솔에 시각화한다.

시뮬레이터를 실행한 결과, 비선점형 SJF 는 FCFS 대비 평균 대기 시간에서 약 5 초가량 단축되는 성과를 보였으며, 선점형 SJF 는 평균 대기 시간과 평균 응답 시간 모두에서 가장 우수한 성능을 확인하였다. Priority 알고리즘은 프로세스 우선순위에 따른 실행 순서를 유연하게 제어할 수 있었으나, 특정 프로세스가 장시간 대기하는 현상이 관찰되기도 하였다. Round Robin 은 공정성을 보장하지만, 퀀텀 설정에 따라 컨텍스트 스위칭 오버헤드가 증가하는 단점을 확인하였다.

아래는 시뮬레이터 실행 결과 화면이다. 보고서에는 FCFS 실행 결과만 포함시켰지만, 깃헙에는 모든 알고리즘의 실행 결과 화면을 넣어 두었다. Linux 환경이 아니라 맥 터미널에서 실행한

이유는, linux 환경에서 실행할 시 프로세스와 간트 차트가 한 화면에 다 나오지 않기 때문이다.

```
(base) kimjuyeon@McDreamy OS_simulator % ./scheduler
PID     Arrival Burst   Priority
1       3       7       2
2       3       7       0
3       2       8       1
4       4       6       3
5       1       9       9

Select Scheduling Algorithm:
1. FCFS
2. SJF (Non-Preemptive)
3. SJF (Preemptive)
4. Priority (Non-Preemptive)
5. Priority (Preemptive)
6. Round Robin
Other number to exit.
> 1
---------------------------------

[Gantt Chart - FCFS]
|IDLE  | P5   | P5   | P5   | P5   | P5   | P5   | P5   | P5   | P5   | P3   | P3   | P3   | P3   | P3   | P3   | P3   | P3
 | P1   | P1   | P1   | P1   | P1   | P1   | P1   | P2   | P2   | P2   | P2   | P2   | P2   | P2   | P4   | P4   | P4   | P4
  | P4   | P4   |
0      1      2      3      4      5      6      7      8      9      10     11     12     13     14     15     16     17
 18     19     20     21     22     23     24     25     26     27     28     29     30     31     32     33     34     35
  36     37     38
---------------------------------

[FCFS Evaluation]
PID     WT      TAT
1       15      22
2       22      29
3       8       16
4       28      34
5       0       9
Average WT: 14.60
Average TAT: 22.00

================================
```

3. 결론

본 프로젝트에서 구현한 시뮬레이터는 다양한 스케줄링 알고리즘을 직접 구현하였고, 모듈화를 통해 각 기능을 구분하였으며, 후속 기능 추가 시 최소한의 수정만으로 확장이 가능하게끔 설계하였다. 각 스케줄링 알고리즘의 특성을 명확히 구현하였으며 이를 Gantt chart 로 시각화하고, Waiting Time 과 Turnaround Time 을 통해 성능을 평가한 바 있다.

본 시뮬레이터의 한 가지 아쉬움은, IO 작업을 구현하지 못했다는 점이다. 스케줄러를 짤 기회가 또 주어진다면, IO 작업의 구현까지 신경 써서 실제 환경과 비슷한 시뮬레이터를 구현해 보고 싶다. IO 작업을 데모 전에 구현하기 위해 급하게 코드를 수정했는데, 코드를 수정하는 과정에서 복사 붙여넣기를 하다가 FCFS 알고리즘에 문제가 생겨 데모가 예상한 대로 이어지지 않았다. 깃헙에 코드를 백업해 두는 것의 중요성을 깨달을 수 있는 뜻깊은 경험이었다.

내 코드에서 IO 구현이 실패한 이유는 크게 두 가지로, Gantt Chart 에 IO burst 를 도입했지만 total time 변수에 io burst time 을 추가하지 않아 gantt chart 에 문제가 생긴 점과 io burst 이후 다른 프로세스로 넘어가지 않고 기존의 프로세스를 포함하여 랜덤한 프로세스로 넘어간 점이 그것이다. IO burst time 결정과 IO burst 되는 시점 결성 등에는 문제가 없었으므로, 두 가지 문제점을 개선하면 IO 작업이 구현된 시뮬레이터를 만들 수 있으리라 예상한다.

4. 참고문헌

1. Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley.
2. Tanenbaum, A. S., & Bos, H. (2015). *Modern Operating Systems* (4th ed.). Pearson.