**오퍼레이팅 시스템 스케줄러 설계 보고서**

12161569 컴퓨터공학과

박도윤

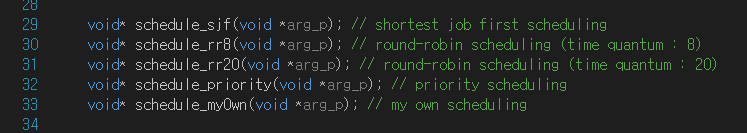
1. **스케줄러 개요**

* 스케줄러는 multi-level queue 방식으로, 각 queue는 총 5개로 구현하였습니다.
* 각 queue는 process형 배열로, 각각 shortest-job first 스케줄링, round-robin 스케줄링 (time-quantum : 8), round-robin 스케줄링 (time-quantum : 20), priority 스케줄링, 나만의 스케줄링 방식으로 구현하였습니다. 한 queue 당 최대 10개의 process를 담을 수 있도록 구현하였습니다. (총 50개) total\_process 배열은 process 정보를 입력 받을 때 임시로 저장하기 위해 필요한 배열입니다.

시계이(가) 표시된 사진

자동 생성된 설명

* Queue 간의 스케줄링은 priority 스케줄링으로 구현하였습니다. 자신보다 더 높은 priority를 가진 queue들의 process가 모두 종료될 때까지 수행되지 않도록 구현하였습니다.
* 각 queue에서의 스케줄링은 별도의 스레드로 처리하였고, queue 간의 스케줄링은 semaphore를 활용하여 동기화하였습니다.
* 스케줄링에 필요한 함수는 다음과 같이 구현하였습니다.

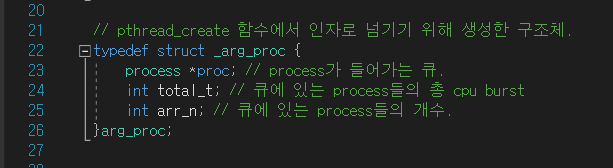


* Process는 process형 구조체를 만들어서 사용하였습니다. exit\_flag는 해당 process가 종료되었는지 아닌지 알기 위한 flag 역할을 하는 변수입니다.

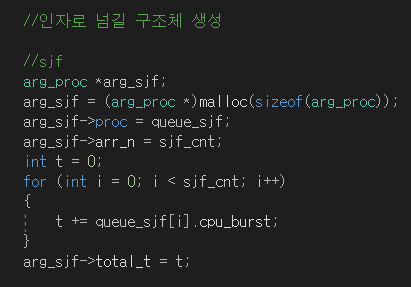
노트북, 화면이(가) 표시된 사진

자동 생성된 설명

* Pthread\_create 함수에서 인자로 필요한 변수들을 넘기기 위해 arg\_proc라는 구조체를 만들어서 사용하였습니다.



인자 구조체 생성 예시는 다음과 같습니다. 이러한 인자 구조체를 각 스케줄링 함수 당 하나씩 총 5개를 생성하여 사용하였습니다.



* 처음에 들어갈 총 process의 수를 입력 받고, 그 다음 process의 수만큼 class number, priority, burst time을 입력 받도록 구현하였습니다. Process id는 들어가는 순서에 따라 자동으로 부여하였습니다. ex) 첫번째로 입력 받은 process id : 1, 두번째로 입력 받은 process의 id : 2



* Class number가 1이면 shortest-job first, 2이면 round-robin (time-quantum : 8), 3이면 round-robin(time-quantum : 20), 4면 priority, 5면 나만의 스케줄링 queue에 들어가도록 구현하였습니다. 먼저 class number, priority, cpu burst를 total\_process에 입력 받고, process id를 지정하고 exit flag를 0으로 지정한 다음, class number에 맞는 스케줄링 queue에 삽입하였습니다.

스크린샷이(가) 표시된 사진

자동 생성된 설명

* 코드는 gcc를 이용하여 컴파일하였습니다.

시계, 그리기이(가) 표시된 사진

자동 생성된 설명

* Main 함수에서의 개략적인 내용은 다음과 같습니다.

텍스트, 앉아있는, 테이블이(가) 표시된 사진

자동 생성된 설명

* 예시 input은 다음과 같습니다.

22 // process의 총 개수

2 0 17 // class number, priority, burst time

1 0 43 // process id : 2

5 2 33 // process id : 3

5 5 12

1 0 23

2 0 87

4 5 3

2 0 56

2 0 8

3 0 53

1 0 2

5 4 36

1 0 7

3 0 17

4 7 9

2 0 14

5 1 86

5 3 71

4 1 19

3 0 68

3 0 24

1 0 9 // process id : 22

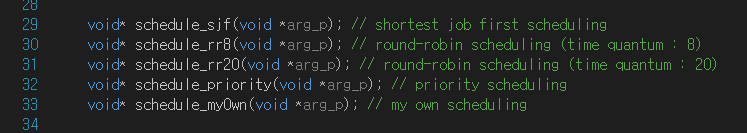
1. **Multi-level queue**

* Queue는 총 5개로 구현하였고, 각 queue는 process형 배열로 최대 10개의 process를 담을 수 있도록 구현하였습니다. (총 50개의 process를 담을 수 있도록 구현하였습니다.)

시계이(가) 표시된 사진

자동 생성된 설명

* Queue 간의 스케줄링은 priority 스케줄링으로 구현하였고, shortest-job first 스케줄링, round-robin 스케줄링 (time quantum : 8), round-robin 스케줄링 (time quantum : 20), priority 스케줄링, 나만의 스케줄링 순서로 priority를 지정하였습니다. 자신보다 더 높은 priority를 가진 queue들의 process가 모두 종료될 때까지 해당 queue가 수행되지 않도록 구현하였습니다.
* 각 queue에서의 스케줄링은 별도의 스레드로 처리하였고, queue 간의 스케줄링은 semaphore를 활용하여 동기화하였습니다. Semaphore는 총 3개를 사용하였습니다.



1. Schedule\_sjf -> schedule\_rr8 -> schedule\_rr20 -> schedule\_priority -> schedule\_myOwn 5개의 함수가 순서대로 수행되어야 하므로, 3개의 semaphore를 사용하고 처음에 0으로 초기화해주었습니다.

텍스트, 앉아있는, 테이블이(가) 표시된 사진

자동 생성된 설명

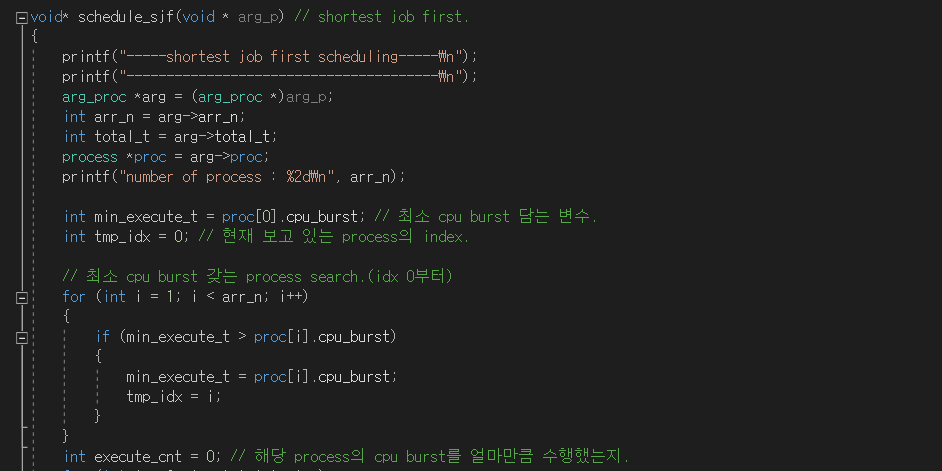
1. schedule\_sjf를 수행한 후, sem\_post(&sem1);을 하여 sem1 값을 1 증가시켰습니다. Sem1 = 1, sem2 = 0, sem3 = 0 상태를 만들었습니다.
2. Schedule\_sjf의 수행이 끝나고 schedule\_rr8을 수행해야 하므로 schedule\_rr8의 맨 앞부분에 sem\_wait(&sem1);을 추가하였습니다. (sem1이 0 이상일 때만 실행 가능하도록) sem1 값은 다시 0이 되고Schedule\_rr8을 수행한 후, sem\_post(&sem2);를 하여 sem2 값을 1 증가시켜 sem1 = 0, sem2 = 1, sem3 = 0 상태를 만들었습니다.
3. Schedule\_rr8의 수행이 끝나고 schedule\_rr20을 수행해야 하므로 schedule\_rr20의 맨 앞부분에 sem\_wait(&sem2);을 추가하였습니다. (sem2가 0 이상일 때만 실행 가능하도록) sem2 값이 0이 되고Schedule\_rr20을 수행한 후, sem\_post(&sem1);과 sem\_post(&sem2);를 하여 sem1과 sem2 값을 1 증가시켜 sem1 = 1, sem2 = 1, sem3 = 0 상태를 만들었습니다.
4. Schedule\_rr20의 수행이 끝나고 schedule\_priority를 수행해야 하므로 맨 앞부분에 sem\_wait(&sem1);과 sem\_wait(&sem2);을 추가하였습니다. (sem1과 sem2가 0 이상일 때만 실행 가능하도록) sem1과 sem2가 다시 0이 되고 schedule\_priority를 수행한 후, sem\_post(&sem3);을 하여 sem3 값을 1 증가시켜 sem1 = 0, sem2 = 0, sem3 = 1 상태를 만들었습니다.
5. Schedule\_priority의 수행이 끝나고 schedule\_myOwn을 수행해야 하므로 맨 앞부분에 sem\_wait(&sem3);을 추가하였습니다. (sem3이 0 이상일 때만 실행 가능하도록)
6. 이렇게 semaphore에 의해 Schedule\_sjf -> schedule\_rr8 -> schedule\_rr20 -> schedule\_priority -> schedule\_myOwn의 수행이 끝나고 나면 sem\_destroy를 이용해 semaphore와 관련된 리소스들을 소멸시켜 주었습니다.

텍스트, 앉아있는, 테이블이(가) 표시된 사진

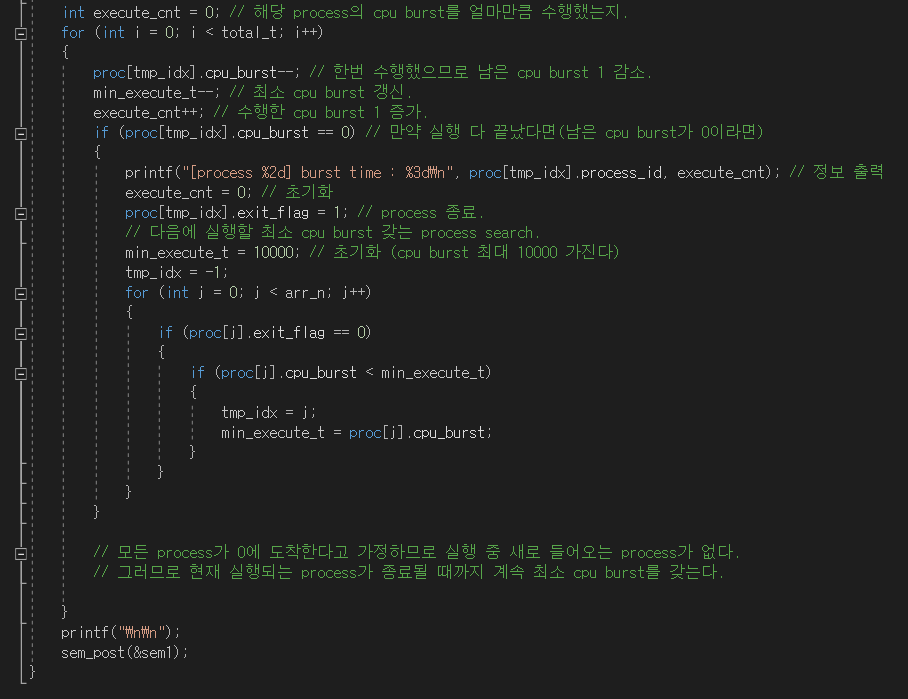
자동 생성된 설명

1. **Shortest-job first scheduling**

* 가장 최소 시간의 cpu burst time을 갖는 process가 우선적으로 수행되는 스케줄링 기법입니다. Minimum average waiting time을 갖기 때문에, queue 중 가장 높은 우선순위에 두었습니다.
* 모든 process가 0에 도착한다고 가정하였기 때문에 process 수행 중 새로 들어오는 process가 없으므로 현재 실행되는 process가 수행이 종료될 때까지 계속 최소 cpu burst time을 갖는 process가 됩니다.
* Class number가 1인 process가 이 스케줄링 queue에 들어가게 되므로, process 2, 5, 11, 13, 22가 이 queue에서 스케줄링 됩니다.



1. 우선 인자 구조체로 담아 온 변수들을 정리하고, queue에서 최소 cpu burst time을 갖는 process를 index 0부터 탐색합니다.



1. 최소 cpu burst time을 갖는 process를 찾은 후에, 해당 cpu burst time이 0이 될 때까지 수행합니다. (수행 중 새로 들어오는 process가 없으므로 해당 process의 cpu burst time이 0이 될 때까지 계속 수행합니다)
2. Cpu burst time이 0이 되면, process의 id와 수행한 burst time을 출력하고 exit\_flag를 1로 갱신하여 종료합니다.
3. 필요한 변수들을 초기화 한 다음 종료되지 않은 process 중 최소 cpu burst time을 갖는 process를 찾아서 다시 해당 cpu burst time이 0이 될 때까지 수행하고 정보를 출력합니다.
4. 이 과정을 queue에 있는 모든 process가 수행될 때까지 반복합니다.

* 출력은 다음과 같습니다.

텍스트이(가) 표시된 사진

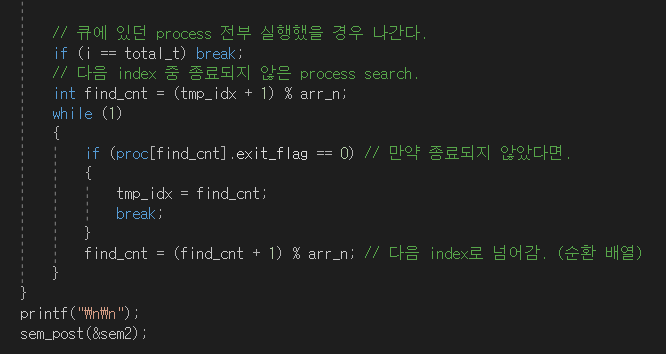
자동 생성된 설명

1. **Round-robin scheduling**

* 정해진 time quantum마다 돌아가면서 process를 수행하는 스케줄링 방식입니다.
* Time quantum을 8로 설정한 queue와 20으로 설정한 queue 두 가지를 구현하였습니다.
* Class number가 2인 process가 time quantum이 8인 queue에 들어가게 되므로, process 1, 6, 8, 9, 16이 해당 queue에 스케줄링 됩니다.
* Class number가 3인 process가 time quantum이 20인 queue에 들어가게 되므로, process 10, 14, 20, 21이 해당 queue에 스케줄링 됩니다.
* 스크린샷, 컴퓨터이(가) 표시된 사진

  자동 생성된 설명예시는 time quantum이 8인 코드를 가져왔습니다.

1. 우선 인자 구조체로 담아 온 변수들을 정리하고, time quantum 8만큼 들어온 순서대로 process를 수행합니다.
2. 만약 해당 process가 time quantum 동안 다 수행이 되었다면 exit\_flag을 1로 갱신하여 종료하고 수행되기 전 기존의 burst time, time quantum 동안 수행된 burst time, 남은 burst time을 출력합니다. 수행된 burst time이 꼭 8이 아닐 수도 있으므로 따로 계산을 하였습니다.
3. 남은 cpu burst time이 있다면 수행되기 전 기존의 burst time, time quantum 동안 수행된 burst time (여기서는 8), 남은 burst time을 출력합니다.



1. 만약 queue에 있던 process가 전부 실행되었다면 (i = total\_t) while문을 벗어납니다.
2. 그렇지 않은 경우, 다음 process 중 종료되지 않은 process를 탐색합니다. 다음 process를 탐색할 때, 현재 process가 queue의 마지막이고 다음 process가 queue의 처음일 수 있으니 index를 나타내는 변수(find\_cnt)에 1을 더한 뒤 항상 (% arr\_n) 나머지 연산을 합니다.

* Time quantum이 20인 round-robin 스케줄링도 time quantum이 20이라는 점만 빼면 위의 방식과 동일합니다.
* time quantum이 8인 round robin 스케줄링의 출력은 다음과 같습니다.

테이블이(가) 표시된 사진

자동 생성된 설명

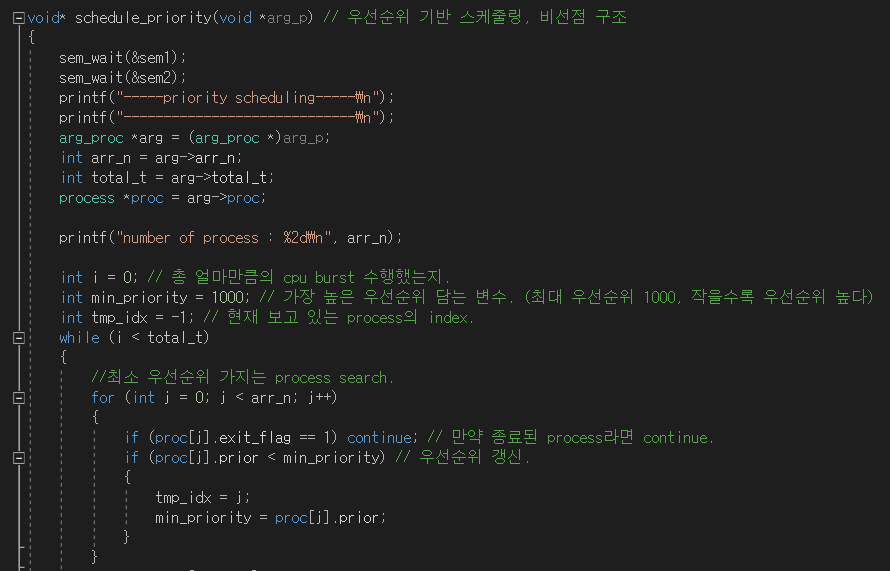
* time quantum이 20인 round robin 스케줄링의 출력은 다음과 같습니다. 이 입력은 강의자료에 있는 round-robin 예시 입력을 그대로 가져왔습니다.

테이블, 앉아있는, 검은색, 컴퓨터이(가) 표시된 사진

자동 생성된 설명

1. **priority scheduling**

* 우선순위가 높은 순서대로 process를 스케줄링 하는 방식입니다.
* 우선순위가 작을수록 높도록 구현하였습니다.
* Class number가 4인 process가 해당 queue에 들어가므로, process 7, 15, 19가 이 queue에서 스케줄링 됩니다.



1. 우선 인자 구조체로 담아 온 변수들을 정리하고 가장 높은 우선순위(최소 우선순위)를 가지는 process를 탐색합니다.

모니터, 화면, 검은색, 앉아있는이(가) 표시된 사진

자동 생성된 설명

1. 가장 높은 우선순위를 가지는 process를 전부 수행하고, 해당 process의 exit\_flag을 1로 갱신하여 종료합니다.
2. 해당 process의 id, 우선순위, 수행한 burst time을 출력합니다.
3. 변수들을 초기화 한 뒤, 모든 process들이 전부 수행될 때까지 이 과정을 반복합니다.

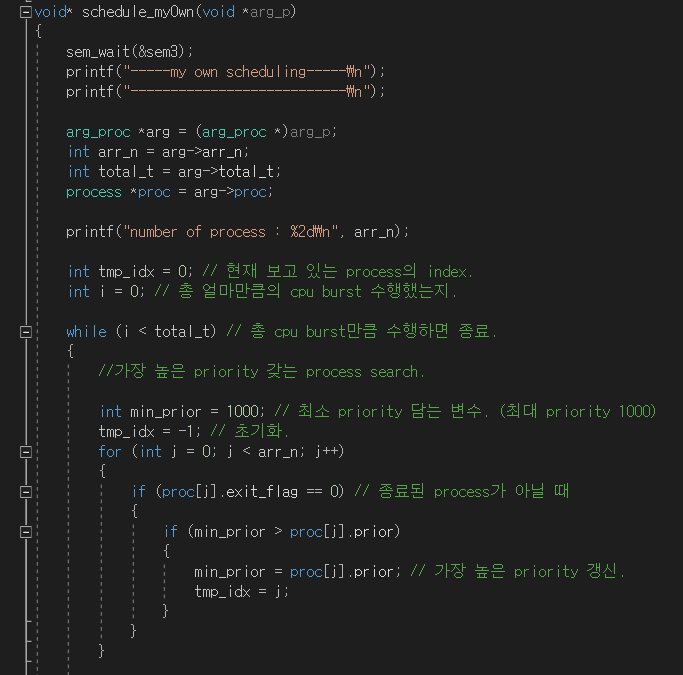
* 출력은 다음과 같습니다.

그리기이(가) 표시된 사진

자동 생성된 설명

1. **My own scheduling**

* 제가 고안한 스케줄링 방식입니다.
* Round-robin 스케줄링 방식과 우선순위 스케줄링 방식을 합쳐 구현하였습니다.
* time quantum마다 우선순위가 가장 높은 process를 수행하되, starvation을 방지하기 위해 수행이 끝나면 해당 process의 우선순위를 1 증가시킵니다. (우선순위를 낮춥니다)
* time quantum은 15로 설정하였습니다.



1. 우선 인자 구조체로 담아 온 변수들을 정리하고 종료되지 않은 process 중에 가장 높은 우선순위를 갖는 process를 탐색합니다.
2. time스크린샷, 모니터, 앉아있는, 검은색이(가) 표시된 사진

   자동 생성된 설명 quantum 15만큼 해당 process를 수행합니다.
3. 만약 해당 process가 전부 수행되었다면, exit\_flag를 1로 갱신하여 종료하고 해당 process의 id, 우선순위, 기존의 burst time, 수행된 burst time, 남은 burst time을 출력합니다. 수행된 burst time이 15가 아닐 수도 있기 때문에 따로 계산을 하였습니다.
4. 전부 수행되지 않았다면 해당 process의 id, 우선순위, 기존의 burst time, 수행된 burst time, 남은 burst time을 출력하고 우선순위를 1 증가시킵니다.
5. Queue에 있는 process를 전부 수행할 때까지 위의 과정을 반복합니다.

* 출력은 다음과 같습니다.

텍스트, 컴퓨터, 테이블이(가) 표시된 사진

자동 생성된 설명