**CPU Scheduler Simulation**

**오퍼레이팅시스템-001분반**

**12161567 박기수**

1. **모델 설계**

본 모델은 Multilevel queue scheduling 방식을 사용했다. 큐 간 스케줄링 방식은 SJF scheduling에 착안하여 각 큐의 burst time의 합이 최소인 큐 먼저 처리한다. 각 큐의 waiting time을 최소화할 수 있을 것으로 예상된다. 큐의 개수는 총 5개(**#define Q\_CNT 5**)이며, 각 큐는 최대 10개(**#define CAPACITY 10**)의 프로세스를 저장할 수 있다. Multilevel queue를 구성하는 첫 번째 큐의 scheduling 방식은 FCFS scheduling, 두 번째 큐는 SJF scheduling, 세 번째 큐는 Round Robin scheduling(time quantum = 3), 네 번째 큐는 Priority scheduling, 다섯 번째 큐는 Round Robin scheduling(time quantum = 15)를 사용했다.

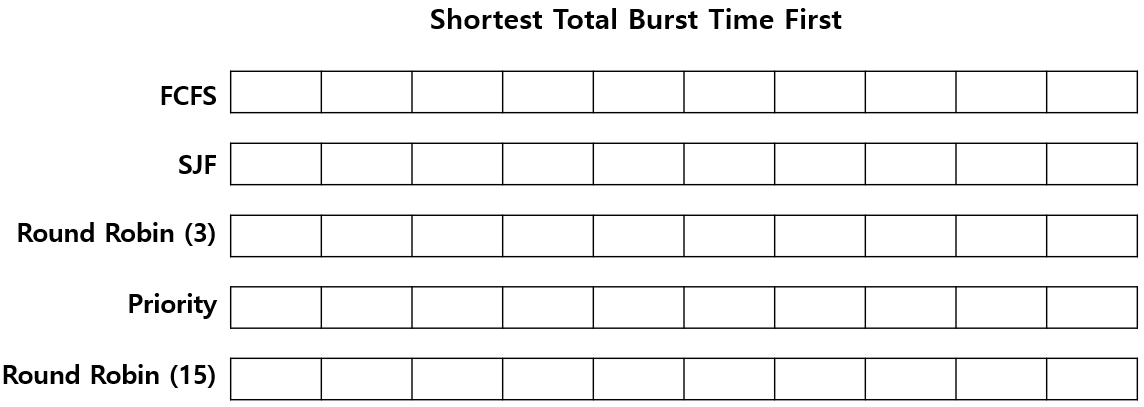


그림 1. Scheduling 방식

1. **구현**

본 모델에서 사용한 전역 변수는 총 3개이다. 각 쓰레드의 동기화를 위한 semaphore 변수(**mutex**)와 가장 마지막으로 처리된 프로세스의 식별자를 저장하는 **lastest\_pid**, context switch 횟수를 저장하는 **context\_switch\_cnt**로 구성되며, **lastest\_pid**와 **context\_switch\_cnt**는 각 쓰레드의 critical section에서 처리된다.

프로세스와 각 큐들을 효과적으로 처리하기 위해서 **Process** 구조체와 **Ready\_queue** 구조체를 선언했다. **Process** 구조체는 process가 어떤 큐에서 처리될 지(**class\_num**), 프로세스의 식별자(**pid**), 프로세스의 우선순위(**priority**), 프로세스의 버스트 시간(**burst\_time**)을 저장한다. **Ready\_queue** 구조체는 circular queue로 구현했으며, Queue의 첫 번째 원소와 마지막 원소를 가리키는 **front, rear** 변수를 가진다. 또한, 저장된 프로세스의 개수(**count**), 저장된 프로세스의 burst time의 합(**total\_burst\_time**), 해당 큐의 scheduling 방식(**sched\_type**), Round robin 방식을 사용할 때 필요한 **time\_quantum** 변수, 각 프로세스를 저장하는 큐(**Process p[]**)로 구성된다.

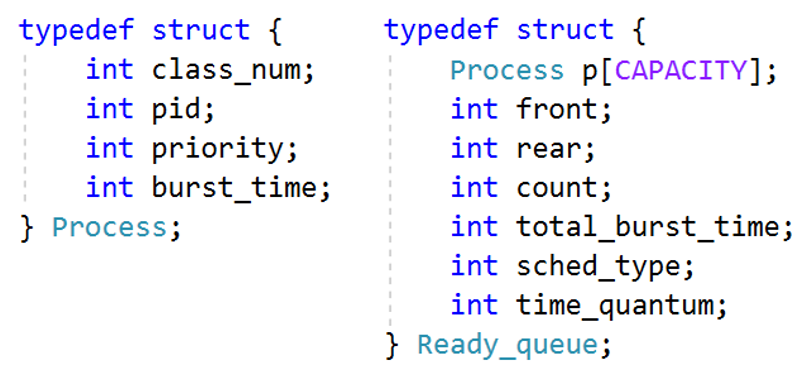


그림 2. Process 구조체, Ready\_queue 구조체

**setReadyQueueType()** 함수는 각 큐를 초기화하고 scheduling 방식을 결정한다. Round Robin 방식을 사용할 경우 **time\_quantum**을 0이 아닌 값으로 설정한다. **insertProcess()** 함수는 처리할 프로세스의 수와 프로세스의 정보(**class, pid, priority, burst\_time**)를 입력한다. 만약 삽입하려는 큐가 가득 찼을 경우 오류 메시지를 출력하고 프로세스 정보를 다시 입력한다. **printProcess()** 함수는 입력된 프로세스의 정보를 출력한다. **compareWithBurstTime(), compareWithPriority()** 함수는 프로세스를 각각 burst time, priority 기준 오름차순 정렬하고, **compareWithTotalBurstTime()** 함수는 각 Ready queue를 total burst time 기준 오름차순 정렬한다. **sched\_FCFS(), sched\_SJF(), sched\_Priority(), sched\_RR()** 함수들은 각각 FCFS, SJF, Priority, Round robin scheduling을 실행한다. **sched\_Queue()** 함수는 각 큐의 scheduling을 실행하는 쓰레드를 실행시킨다.

|  |  |
| --- | --- |
| **함수** | **기능** |
| void setReadyQueueType (Ready\_queue\*); | 큐 초기화 및 scheduling 방식 결정 |
| void insertProcess (Ready\_queue\*); | 프로세스를 각 큐에 삽입 |
| void printProcess (Ready\_queue\*); | 프로세스 정보 출력 |
| int compareWithBurstTime (const void\*, const void\*); | 프로세스를 burst time 순서로 정렬 |
| int compareWithPriority (const void\*, const void\*); | 프로세스를 priority 순서로 정렬 |
| int compareWithTotalBurstTime (const void\*, const void\*); | 큐를 total burst time 순서로 정렬 |
| void\* sched\_FCFS (void\*); | FCFS scheduling 수행 |
| void\* sched\_SJF (void\*); | SJF scheduling 수행 |
| void\* sched\_PRIORITY (void\*); | Priority scheduling 수행 |
| void\* sched\_RR (void\*); | Round Robin scheduling 수행 |
| void\* sched\_Queue (void\* q); | 큐 간 scheduling 수행 |

**표 1. 사용한 함수 목록**

1. **실행**

main 함수에서는 multilevel queue를 생성하고 프로세스들을 입력한다. 프로세스를 모두 입력하면 입력한 프로세스를 출력하고 큐 간 scheduling을 수행하는 쓰레드를 실행한다. 쓰레드가 종료될 때까지 기다렸다가 쓰레드가 종료되면 semaphore 변수를 제거하고 프로그램을 종료한다.

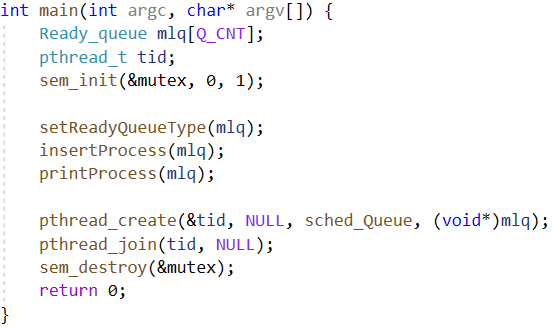


그림 . main 함수

큐 간 scheduling을 수행하는 **sched\_queue()** 함수는 **stdlib.h**의 **qsort()** 함수를 호출하여 각 큐를 **total\_burst\_time** 순서로 정렬한다. 이후 for문에서 각 큐를 scheduling하는 쓰레드를 실행하는데, 각 큐의 scheduling type에 맞는 함수를 호출한다. 하나의 쓰레드가 실행하면 해당 쓰레드가 종료될 때까지 기다린다.

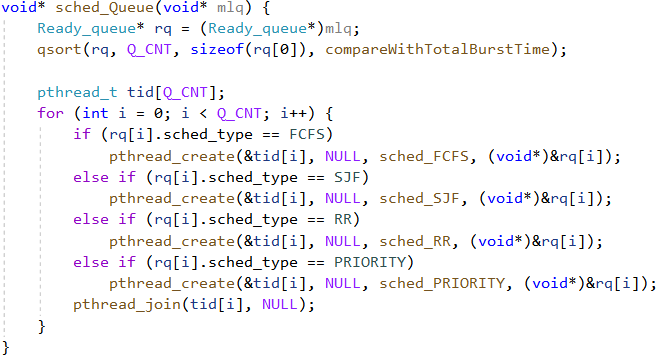


그림 . sched\_Queue() 함수

**sched\_FCFS()** 함수는 현재 인덱스를 저장하는 **current\_idx**, 큐의 실행시간을 저장하는 **running\_time** 변수를 가진다. FCFS scheduling은 프로세스가 입력된 순서대로 처리하므로, 별도의 정렬을 하지 않는다. while 문을 **total\_burst\_time**만큼 반복 실행한다. 만약 **current\_idx**의 프로세스의 burst time이 0보다 클 경우 해당 프로세스의 **pid**를 출력하고 burst time을 1만큼 감소시키고 running time을 1만큼 증가시킨다. 그 후에 critical section으로 진입하여 현재 처리중인 프로세스와 마지막으로 실행한 프로세스가 다를 경우, **context\_switch\_cnt**를 1만큼 증가시킨다. **lastest\_pid**를 갱신한다. 만약 **current\_idx** 프로세스의 burst time이 0일 경우 해당 프로세스의 처리가 완료되었음을 의미하므로 다음 프로세스를 수행하기 위해 **current\_idx**를 1만큼 증가시키고 **count**를 1만큼 감소시킨다. 또한 **front**를 1만큼 증가시켜 다음 프로세스를 가리킨다. while문의 수행이 끝나면 마지막으로 **count**를 1만큼 감소시키고, **front**를 1만큼 증가시킨다. 모든 프로세스가 처리됐으므로 **front**와 **rear**는 같은 곳을 가리킨다.

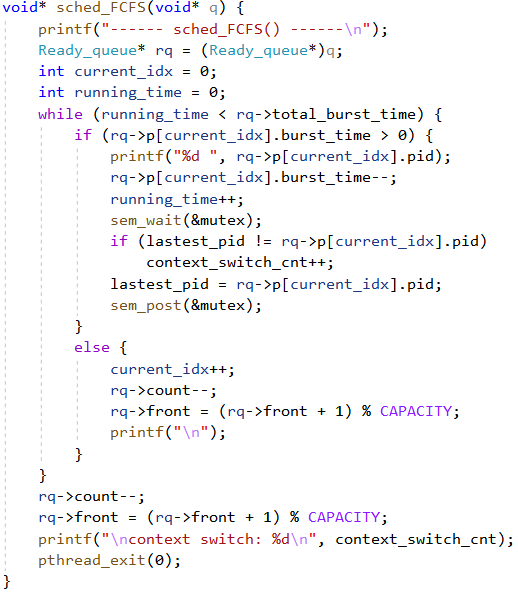


그림 . sched\_FCFS() 함수

**sched\_SJF(),** **sched\_Priority()** 함수는 **sched\_FCFS()** 함수의 while문 앞에 각각 burst time, priority 순으로 정렬시키는 작업을 추가로 수행해주고, 나머지 부분은 **sched\_FCFS()**와 같다.

**sched\_RR()** 함수는 time quantum을 고려해야 하므로, **current\_time** 변수를 추가했다. while문을 **total\_burst\_time**만큼 반복 실행한다. 만약 현재 프로세스를 **time\_quantum**만큼 처리했을 경우 **current\_time**을 0으로 초기화하고 **current\_idx**를 1만큼 증가시킨다. 여기서 주의할 점은 프로세스를 번갈아 가면서 처리해야 하므로 circular queue에서 index를 증가시키는 것처럼 **current\_idx**를 증가시킨 후 프로세스 개수와 나머지 연산을 한다. Burst time에 따른 프로세스 처리는 다른 scheduling 방식과 동일하며 **current\_time** 처리를 추가적으로 한다. 만약 burst time이 0인데 **current\_time**이 0이 아닐 경우 **current\_time**을 0으로 변경한다.

1. **성능 분석**

위의 Gantt chart는 큐를 순서대로 처리한 결과이고, 아래의 Gantt chart는 큐를 본 모델의 방식대로 처리한 결과이다. 첫번째 Gantt chart의 waiting time은 0+41+78+149+187 = 455인데, 두번째 Gantt chart의 waiting time은 0+37+75+116+187 = 415로, 본 모델의 scheduling 방식의 waiting time이 더 짧다.

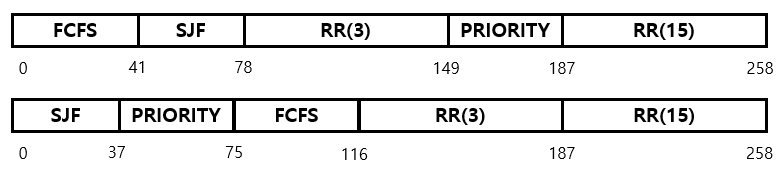


그림 6. Scheduling 방식 별 Gantt chart

다음은 Round robin scheduling에서 time quantum에 따른 context switch 횟수 차이를 비교할 것이다. 3번째 큐에 삽입된 프로세스들의 burst time과 5번째 큐에 삽입된 프로세스들의 burst time은 같지만 time quantum은 각각 3, 15로 다르다. Context switch 횟수는 각각 18, 5로서, time quantum이 3일때의 context switch 횟수가 더 작다.

|  |  |  |
| --- | --- | --- |
|  | 3번째 큐: Round robin  (time quantum: 3) | 5번째 큐: Round robin  (time quantum: 15) |
| **(pid, burst time)** | (9, 11) | (17, 11) |
| (10, 7) | (18, 7) |
| (11, 40) | (19, 40) |
| (12, 13) | (20, 13) |
| **Context switch** | **18** | **5** |

**표 2. time quantum에 따른 context switch의 차이**

1. **결론**

대부분의 multilevel queue의 큐 간 스케줄링 방식은 고정 우선순위의 선점형 스케줄링이지만, 본 모델에서는 큐의 total burst time이 가장 작은 큐 먼저 처리했다. 본 모델의 스케줄링 방식을 따르면 각 큐의 waiting time을 줄일 수 있다. 하지만 우선순위가 가장 높은 큐의 total burst time이 가장 크다면 해당 큐는 가장 마지막에 처리될 것이다. 따라서 우선순위와 waiting time간의 trade-off가 발생한다. 또한, Round robin 방식에서 time quantum이 짧을 수록 context switch가 더 많이 발생한다. 만약 context switch의 overhead가 클 경우 전체적인 실행 시간이 느려 질 것이다.