DKU 단국대학교 DANKOOK UNIVERSITY

REPORT

멀티태스킹 운영체제 시뮬레이터

주 제 : 멀티태스킹 운영체제 시뮬
교 수 : 김하연
소속학과: 소프트웨어
학 번 : 32207825
이 름 : 정재호
마 감 : 2024.04.21

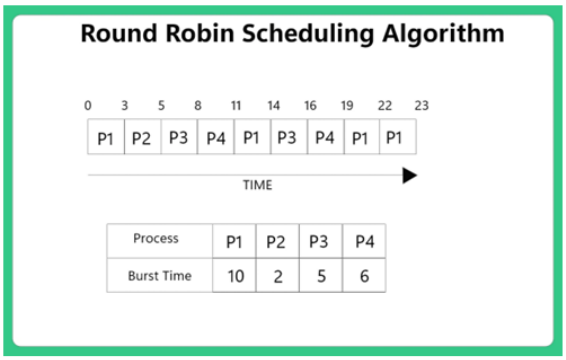
단국대학교 DANKOOK UNIVERSITY

- 목차

■ 프로젝트 설계

➢ 요구 사항 분석

멀티태스킹의 기본을 Round Robin 스케줄링 알고리즘으로 구현하고 프로세스 사이의 공정한 시간분배를 해줘야 합니다.
우선 프로세스는 사용자가 정의한 간단한 태스크면 충분하기 때문에 time 라이브러리의 time.sleep과 print 출력문으로 간단하게 구현가능할 것 같습니다.



수업 자료에 있는 프로세스 예시를 활용해서 P1~P4 프로세스를 각각 버스트타임 10,2,5,6으로 설정하여 라운드 로빈 스케줄링을 테스트해 보려 합니다.

Round Robin 스케줄링을 구현하는데 하나의 CPU를 4개의 프로세스가 각각의 시간 할당량만큼 차지하게 하며 배분해주는것이 목표입니다.
4개 프로세스의 상태는 대기, 실행, 완료로 구별해서 관리가 편리해지게 할 것이고, 전체 프로세스가 현재 어떤 상태인지 체크하는 UI를 넣어 진행과정을 알 수 있게 해주려 합니다.

■ 주요 구현 사항

➢ 라이브러리

```
1  import time
```

사용하는 라이브러리는 time입니다.

➢ class Process

```
3  class Process:
4      def __init__(self, name, burstTime): # 프로세스는 이름과 버스트타임을 가집니다
5          self.name = name
6          self.burstTime = burstTime
7          self.state = "준비" # 준비로 초기화
8
9      # 프로세스 실행
10     def execute(self, quantum):
11         print(f"{self.name} 실행중...")
12
13         if self.burstTime <= quantum:
14             time.sleep(self.burstTime)
15             self.state = "완료"
16
17             print(f"프로세스 {self.name}가 완료로 바뀝니다.")
18
19         else:
20             time.sleep(quantum)
21             self.burstTime -= quantum
22
23             print(f"프로세스 {self.name}가 {quantum}초만큼 실행되었습니다.")
24
```

하나의 CPU를 싱글 쓰레드 프로세스 4개가 라운드 로빈 알고리즘을 통해 할당 받게 해주기 위해서는 multiprocessing 라이브러리까지는 이용하지 않아도 되겠다는 생각이 들었습니다. 그래서 Process 클래스를 만들어주고 이름과 버스트타임, 상태를 속성으로 가지게 해줍니다. 상태는 '준비'로 초기화된 상태로 시작해야합니다.

execute 메소드로 프로세스를 실행 시키는 과정에서 라운드 로빈 퀀텀에 맞춰서 프로세스를 동작(time.sleep으로 동작과정 생략)시켜주고 버스트타임을 동작한 퀀텀만큼 빼주다가 퀀텀보다 버스트타임이 적게 남게된다면 완료상태로 바꾸는 코드를 짜줬습니다.

➢ class RoundRobin

```
25 class RoundRobin:
26     def __init__(self, processes, quantum): # 프로세스 리스트와 퀀텀을 가집니다
27         self.processes = processes
28         self.quantum = quantum
```

라운드 로빈 스케줄링 알고리즘을 구현할 클래스의 속성은 먼저 퀀텀이 필요합니다. 그리고 대기열을 관리해주기 위한 다양한 방법이 있을 텐데 저 같은 경우에는 리스트를 사용하게 되었습니다.

➢ def run

```
30     def run(self):
31         # 프로세스 리스트가 전부 완료가 아니면 True
32         while any(process.state != "완료" for process in self.processes):
33 
34             #프로세스 리스트 for루프
35             for process in self.processes:
36 
37                 # 준비 상태이면 실행시켜줍니다
38                 if process.state == "준비":
39                     process.state == "실행"
40                     process.execute(self.quantum)
41 
42                     # 버스트타임이 0이 됐으면 완료상태로 전환
43                     if process.burstTime <= 0:
44                         process.state = "완료"
45 
46                 # 이미 완료 상태이면 컨티뉴
47                 elif process.state == "완료":
48                     continue
49 
50             # 전체 프로세스 상태 출력
51             print("\n    | 프로세스 |  상태  |")
52             for process in self.processes:
53                 print(f'\t{process.name}\t {process.state}')
54             print("")
```

라운드 로빈의 run메소드는 실질적인 동작을 통제해주게 해줬습니다. 우선 프로세스의 리스트에 하나라도 상태가 '완료'가 아닌 프로세스가 있다면 실행되게 조건문을 넣어줍니다. 프로세스들을 반복문을 통해 순차적으로 돌리며 준비상태를 발견하면 실행(process.execute)를 해주고, 상태는 '준비'지만 버스트타임이 0이 된 프로세스가 있으면 프로세스 상태를 '완료'로 바꿔줍니다. 그 외에 완료 상태인 프로세스는 continue해주면 됩니다. 그리고 메소드 끝에는 전체 프로세스 상태를 출력해

주는 텍스트 UI를 넣어 작동하는 중간에 어떻게 변화하는지 확인하게 해줬습니다.

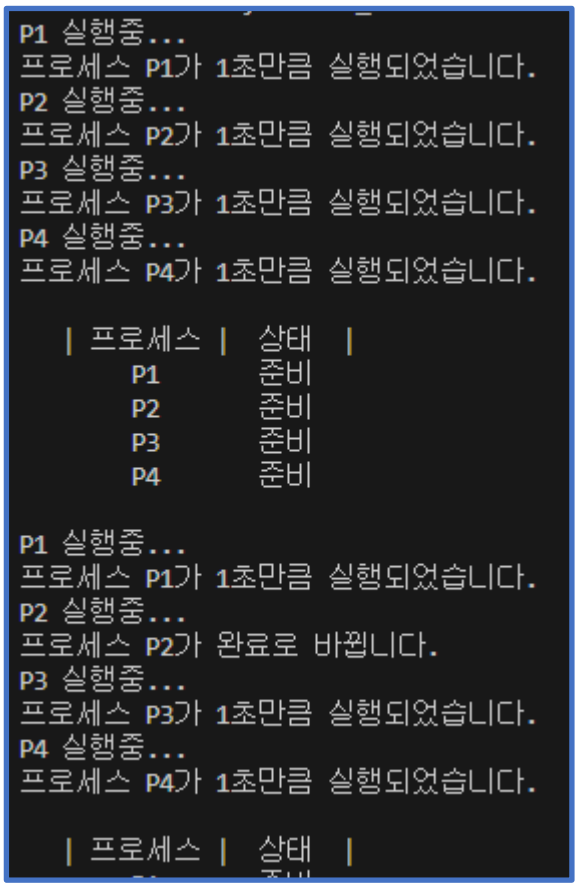
- main

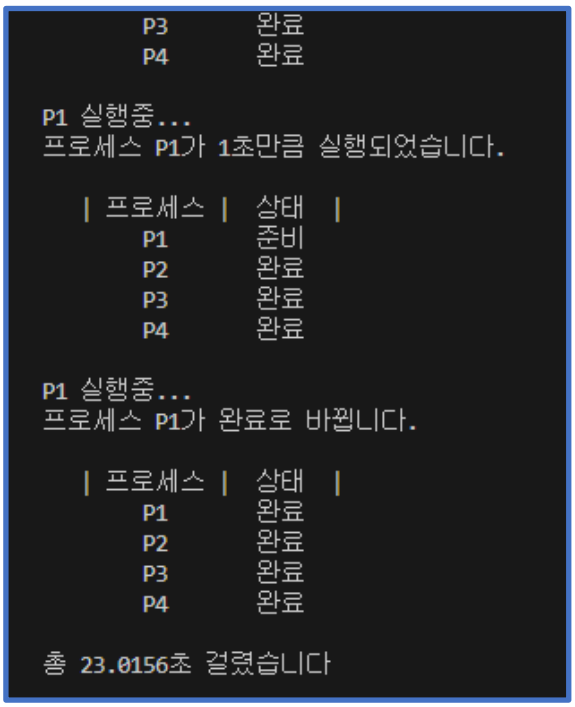
```
56  if __name__ == "__main__":
57      # 시간 측정 시작
58      start = time.perf_counter()
59
60      # 프로세스 생성
61      p1 = Process("P1", 10)
62      p2 = Process("P2", 2)
63      p3 = Process("P3", 5)
64      p4 = Process("P4", 6)
65
66      # 프로세스 리스트
67      processes = [p1, p2, p3, p4]
68
69      # 라운드 로빈 생성, 퀀텀 설정
70      RRS = RoundRobin(processes, quantum=1)
71      RRS.run()
72
73      # 시간 측정 종료
74      finish = time.perf_counter()
75
76      print(f'총 {round(finish - start, 4)}초 걸렸습니다')
```

메인에는 시간측정을 위해서 start와 finish를 time라이브러리를 통해 측정해주는 방식을 사용했습니다. 프로세스 4개를 클래스 Process를 통해 생성해주고 리스트 processes를 만들어 저장해줬습니다. 라운드로빈의 퀀텀은 일단 1로 설정해주고 테스트를 해본 이후에 다양한 퀀텀으로 변경하여 무슨 차이가 발생하는지 관찰해보려 합니다.

■ 실행 결과

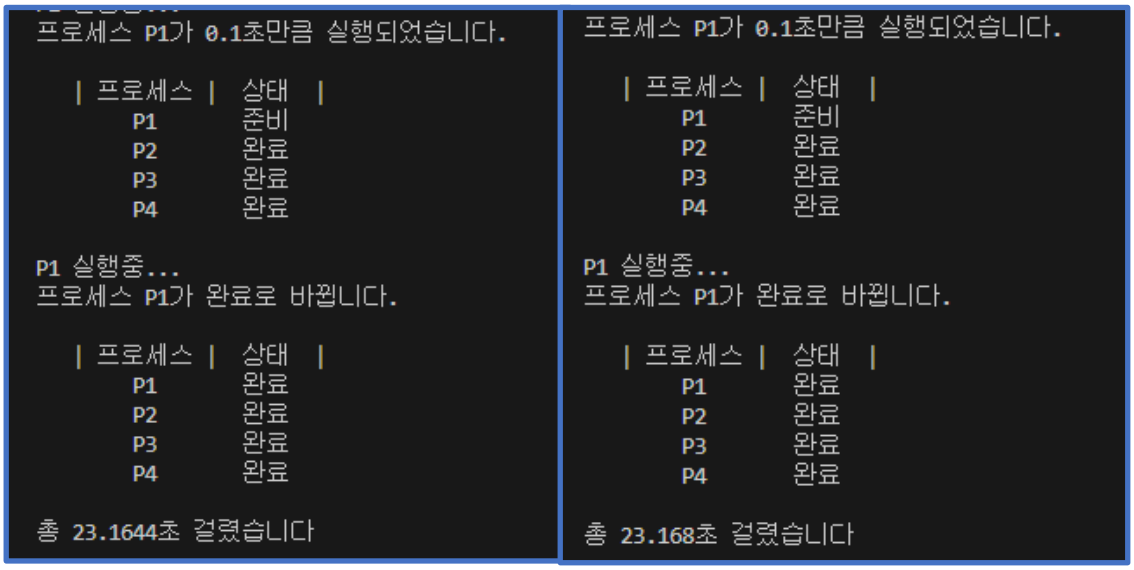
➢ Quantum = 1





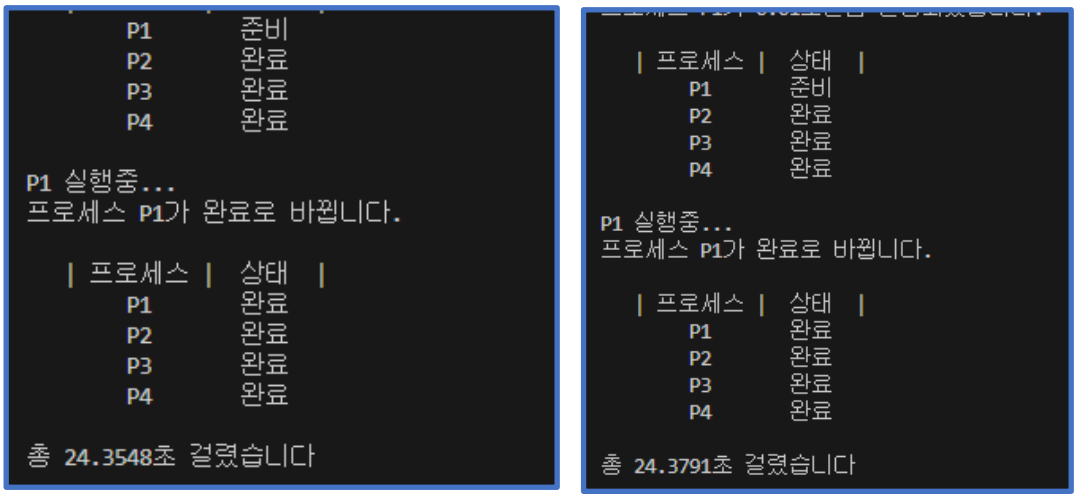
10번 정도의 테스트를 해보았고, 대략적으로 비슷한 종료시간을 가졌습니다. 테스트 결과 P1부터 P4까지 순차적으로 라운드 로빈 스케줄링을 통해 1초씩 실행되며, 실행 중에 완료가 되면 자연스럽게 다음 프로세스로 넘어가는 것을 확인했습니다. 그리고 프로세스 리스트 한 사이클을 돌 때 마다 전체 프로세스를 확인하는 UI도 정상적으로 작동하고 있었습니다. 여기서 퀀텀을 더 작게 쪼개면 어떤 현상이 나타날지 궁금해서 퀀텀을 조정해보려 합니다.

- Quantum = 0.1



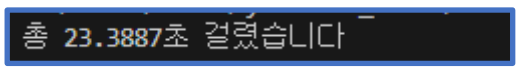
이번에도 10번의 테스트를 진행했고 대략적으로 다 비슷한 결과를 보였습니다. 퀀텀을 1로 설정했을때랑 비교하면 0.15초정도는 더 느린 것을 확인할 수 있었습니다.

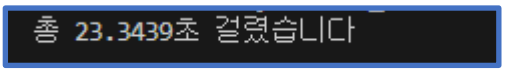
- Quantum = 0.01



10번의 테스트를 해보았고 평균적으로 24.38초 정도가 나왔습니다. 퀀텀 1과 비교하면 1.37초 정도의 큰 차이를 보였습니다.

- Quantum = 0.01 , print문 제거





10번의 테스트를 해보았고 평균적으로 23.35초정도가 나왔습니다. 퀀텀 1과 비교하면 0.34초의 차이이며, 퀀텀 0.01과 비교하면 1.03초의 차이가 발생합니다.

- **실행 결과 분석**

실행 케이스는 1, 0.1, 0.01, 0.01 + print문 제거 총 4개의 케이스로 설정해서 테스트를 해보았습니다. 퀀텀이 작으면 작을수록 모든 프로세스를 더 빠르게 신경 써주는 모습을 확인했고, 실행시간이 더 길어졌다는 것을 알 수 있었습니다. 퀀텀이 0.01초일때는 1초였을때와 비교해서 큰 차이를 보였는데, 이에 대한 이유로 문맥교환 혹은 print문과 같은 I/O 작업을 하면서 시간차이가 발생한 것 아닐까 하는 생각을 하게 되었습니다. 따라서 먼저 print문을 주석 처리하여 비교해보았는데 일단 I/O 작업이 빈번하게 발생하면 작업시간이 길어질 수 있다는 것을 확인했습니다. 그리고 print문이 없더라도 퀀텀이 1이었을때와 비교해서 차이가 있다는 것을 보았을 때 많은 문맥교환을 통한 오버헤드 증가가 나타났을 수도 있겠다는 생각을 했습니다.

따라서 라운드 로빈 스케줄링 알고리즘은 모든 프로세스를 공평한 시간을 돌아가며 사용하게 해주기 때문에 퀀텀을 줄이면 줄일수록 사용자 입장에서 비교적 빠른 응답속도를 느낄 수 있을 것이고 실시간으로 다양한 것을 작동시키면서 작업들을 하고싶은 경우에 장점이 될 수 있겠다는 것을 느꼈습니다. 하지만 프로세스가 10개 100개 이런 식으로 계속 늘어나게 되면 모든 프로세스를 신경 쓰느라 정작 중요한 작업은 빠르게 작업하지 못하는 단점이 발생할 것이고 이런 것들도 신경쓰기 위해 우선순위도 적용시키고 적절한 퀀텀을 찾는 노력이 필요하겠다는 생각을 하며 이번 과제를 마칩니다.

읽어주셔서 감사합니다.