운영체제
Lab 1. CPU Virtualization

주 제 : Linux에서 4가지 Scheduler Simulator 구현
교 수 : 최건희
학 과 : 소프트웨어학과
학 번 : 32190789
이 름 : 김승호
마 감 : 25.05.01.

소스코드 설명

1. RR (Round Robin)

첫 번째 스케줄러는 Round Robin(RR) 기법을 기반으로 작동합니다. 생성자에서 전달된 작업 큐(job_queue_)와 context switch 시간(switch_time_)을 기반으로, 주어진 time slice만큼 각 작업을 분할하여 순차적으로 실행하는 방식입니다.

- **waiting_queue**는 현재 시점까지 도착한 작업들을 보관하는 큐입니다.
- **current_job_**은 현재 CPU에서 실행 중인 작업이며, left_slice_는 해당 작업의 남은 time slice를 관리합니다.
- 매 tick(run() 호출)마다 다음 작업을 1초 실행하며, 이때 남은 시간과 time slice를 감소시키고 상황에 따라 문맥 교환(context switch)을 수행합니다.
- 작업이 완료되면 completion_time을 기록하고 end_jobs_에 저장합니다.

Round Robin 코드는 도착한 작업을 즉시 waiting_queue에 추가하며, 실행 중인 작업이 없으면 대기열에서 작업을 꺼내 CPU에 할당합니다. Slice가 끝난 작업은 다시 waiting_queue 뒤로 보냅니다.

RR::run() 함수 단계별 설명

1. 도착한 작업을 waiting_queue에 추가

```
// 1. 현재 시간에 도착하는 작업들을 waiting_queue에 추가
while (!job_queue_.empty() && job_queue_.front().arrival_time <= current_time_) {
    waiting_queue.push(job_queue_.front());
    job_queue_.pop();
}
```

- 아직 처리 전인 작업들(job_queue_) 중에서 현재 시간보다 **도착 시간이 빠르거나 같은 작업들**을 waiting_queue로 옮긴다.
- **job_queue_는 도착 시간 순 정렬**된 큐라고 가정되어 있음.

2. 현재 실행 중인 작업이 없는 경우

```
// 2. 현재 실행 중인 작업이 없는 경우
if (current_job_.name == 0) {
    if (waiting_queue.empty()) {
        if (!job_queue_.empty()) {
            current_time_ = job_queue_.front().arrival_time;
            return run();
        }
        return -1;
    }

    current_job_ = waiting_queue.front();
    waiting_queue.pop();
    left_slice_ = time_slice_;

    if (current_job_.remain_time == current_job_.service_time) {
        current_job_.first_run_time = current_time_;
    }
}

// 실행 직전에 현재 실행중인 작업 이름 저장
int running_job_name = current_job_.name;
```

아무 작업도 실행하고 있지 않다면

- waiting_queue도 비어 있고, job_queue_는 아직 작업이 남아 있다면 -> 시간 점프 (current_time_을 다음 작업의 도착 시간으로) -> run() 재호출.
- 둘 다 비어 있다면 작업 다 끝난 상태 → -1 반환.

실행할 작업이 있다면

- waiting_queue에서 하나 꺼내서 현재 작업으로 설정.

3. 작업을 1초간 실행

```
// 3. 작업 1초 실행
current_time_ += 1.0;
current_job_.remain_time -= 1;
left_slice_ -= 1;
```

- 1초가 지났으니 현재 시간 증가.
- 현재 작업의 남은 시간, 슬라이스 시간 각각 1씩 감소.

4. 1초 실행 후 새로 도착한 작업들 추가

```
// 4. 실행 후 도착한 작업 waiting_queue에 추가
while (!job_queue_.empty() && job_queue_.front().arrival_time <= current_time_) {
    waiting_queue.push(job_queue_.front());
    job_queue_.pop();
}
```

방금 실행하는 동안 도착한 작업도 waiting_queue에 추가.

5. 작업이 끝났다면

```
// 5. 작업 완료
if (current_job_.remain_time == 0) {
    current_job_.completion_time = current_time_;
    end_jobs_.push_back(current_job_);
    current_job_ = Job();
    left_slice_ = time_slice_;

    if (!waiting_queue.empty() || !job_queue_.empty()) {
        current_time_ += switch_time_;
    }
}
```

작업 완료되면:

- 완료 시간 기록
- 완료 리스트에 넣음 (end_jobs_)
- 현재 작업 초기화
- 문맥 교환 시간 추가 (다음 작업이 있다면)

6. time slice가 끝났다면

```
// 6. time slice 끝
else if (left_slice_ == 0) {
    waiting_queue.push(current_job_);
    current_job_ = Job();
    left_slice_ = time_slice_;

    if (!waiting_queue.empty() || !job_queue_.empty()) {
        current_time_ += switch_time_;
    }
}
```

아직 작업은 끝나지 않았지만 time slice가 다 됐으면:

- 다시 waiting_queue에 넣고 > 새 작업 준비
- context switch 시간 추가

7. 실행한 작업 이름 반환

```
// 7. 저장한 "실행한 작업 이름" 리턴
return running_job_name;
```

위 실행 전에 따로 저장한 작업 이름 반환

Discussion

준비

- 다양한 Workload (A, B)에 대해 다양한 Context Switch Time (0.01, 0.1, 0.2 등)과 여러 스케줄링 알고리즘(RR, FB, Lottery, Stride)을 적용했다.
- RR의 time slice는 1 또는 4로 실험하였다.

분석

- **Workload A**, Context Switch Time = **0.01**일 때
  - **Turnaround Time**이 가장 낮은 스케줄러: **RR (time slice = 4)**
  - 이유: RR은 time slice 단위로 context switch가 자주 발생하지만, slice가 4일 경우 context switch 빈도가 낮아져 전환 오버헤드가 줄어든다.
  - 반면, FB는 job starvation이 발생할 수 있고, Lottery나 Stride는 randomness 나 정밀도 이슈로 인한 오차가 존재했다.
- **Workload B**의 경우 job들이 중후반에 몰려 있어서 RR보다 FB가 유리하게 작동

하기도 했다. 초기 priority가 높은 job이 우선 배치되며 평균 응답 시간이 감소했다.

배운 점 및 어려웠던 점

VirtualBox를 처음 사용해봤는데, OS 이미지를 직접 다운로드 받아 직접 가상머신 구동시키는 것이 재미있었습니다. 남는 노트북이 있다면, ubuntu를 VirtualBox를 이용해 구동시켜 개인 서버로 사용해봐도 좋을 것 같다는 생각을 했습니다.

어려웠던 점은, 코드 구현이였습니다. 제공해주신 깃허브를 fork해서 clone한 다음, 로컬에서 작업 후 git push, 가상머신에서 git pull해 구동시켜보는 식으로 진행했는데, 원하는 결과가 잘 나오지 않아서 아쉬웠습니다.