Introduction

Workload Assumptions

1. 모든 job이 같은 시간만큼 걸린다
2. 모든 job이 같은 시간에 도착한다
3. 모든 job이 CPU만 사용한다
4. 각각 job의 실행 시간은 알고있다

Metrics

- 성능 척도 :
  Turnaround Time = 작업의 완료시간 - 도착시간
  // T(around time) = T(completion) - T(arrival)
  Response Time = 첫 실행 - 도착시간
- Fairness 척도 : 보통 성능과 반비례함

FIFO (First In First Out)

- 먼저 온거 먼저 실행함 : 간단하고 구현하기 쉬움
- ABC가 각각 10만큼 걸리고, ABC순서로 왔다고 치면 Average Turnaround time = (10+20+30)/3 = 20

Convoy Effect

- 모든 작업이 같은 시간을 소모하지 않을때, 동시에 도착했지만 오래걸리는 작업이 먼저 실행 될 경우 평균 시간이 너무 크게 늘어남

SJF(Shortest Job First)

- 실행시간이 짧은 순으로 작업 수행(Non-Preemptive : 선매권이 없음)
- 짧은 실행시간을 가진 작업이 다른 작업 수행 도중 도착한 경우, 여전히 Turnaround Time이 크게 늘어남

STCF(Shortest Time-to-Completion First)

- SJF에 preemption(선매권)을 추가
- 새로운 작업이 시스템에 들어오면, 원래 있던 작업들과 새로 들어온 작업의 Completion까지 걸리는 시간을 비교해 순서를 재조정
- response time에 별로 효율적이지 못함

Round Robin(RR) Scheduling

- Time Slicing Scheduling(Scheduling Quantum) : 정해진 Time Slice만큼 작업을 수행하고, **Run Queue** 에 있는 다음 작업 수행
- 모든 작업이 끝날때까지 반복
- Time Slice의 길이는 Timer Interrupt길이의 배수여야 함
- Response Time에 굉장히 효율적
- 모든 작업에 공평한 방법이지만, 성능 면(Turnaround time)에서 좋지 않음

Time Slice

- 길이가 상당히 중요함
- 짧을 때 : Response Time 향상, Context Switching의 비용이 전체 성능을 압도
- 길 때 : Switching의 비용을 분할상환, Response Time 안좋아짐
- 길이를 어떻게 할 지는 디자이너의 선택(Trade-Off)

Incorporating I/O

- 모든 작업이 입출력을 포함한다고 가정
- 작업 하나가 입출력 작업 요청을 하면, 해당 작업이 Blocked쪽으로 이동, 이 때 스케쥴러가 다른 작업을 CPU에서 돌려야함
- 작업 하나가 완료되면, Interrupt가 raise되고, OS가 해당 작업을 Blocked List에서 Ready List로 옮김

Multi-Level Feedback Queue(MLFQ)

- 과거로부터 배워서 미래를 예측하는 스케쥴러
- 목적 : Turnaround time 최적화(SJF), Response Time 최적화(과거의 지식(작업길이)을 바탕으로)

Basic Rules

- 서로 구별되는 큐를 여러개 가지고 있음(각자 우선순위가 다름)
- 준비가 완료 된 작업은 하나의 큐에 있음 -> 더 높은 우선순위 큐에 속해있는 작업 우선 실행,
  같은 우선순위 큐에 속할 경우 RR방식 적용(Time Slice)
- 관찰된 행동을 바탕으로 작업의 우선순위를 설정
  ex) I/O대기로 인해 CPU를 자주 포기함 -> 우선순위 상승, CPU를 지나치게 오래사용 -> 우선순위 하락

Changing Priority

- Rule 3 : 작업이 처음 시스템에 들어오면, 가장 높은 우선순위에 배치
- Rule 4a : 작업이 time slice 전체를 잡아먹으면, 우선순위 감소
- Rule 4b : 작업이 time slice가 끝나기 전 CPU를 포기할경우, 동일한 우선순위 유지
- I/O를 사용하는 작업의 우선순위가 높음
- 이 측면에서 SJF와 비슷함

Problems

- Starvation : Interactive 작업이 시스템이 너무 많으면, 작업시간이 긴 작업은 절대 CPU 시간을 할애받지 못함
- Game the scheduler : 99퍼센트의 Time slice를 사용 한 뒤 I/O 작업 수행 -> 높은 우선순위 유지
- 프로그램의 성향이 바뀔 수 있음(CPU위주 -> I/O 위주)

Priority Boost

- Rule 5 : 일정시간 S가 지나면, 모든 작업을 제일 높은 우선순위로 올림

Better Accounting

- Scheduler Gaming을 방지하기 위함
- Rule 4 : 작업이 일정 레벨에서 일정 수준 이상의 시간을 사용하면, 우선순위가 내려감

Turning MLFQ and Other Issues

- 우선순위가 낮으면 Time Slice를 길게 함
- 높은 우선순위 -> 짧은 Time Slice, 낮은 우선순위 -> 긴 Time Slice

Solaris MLFQ Implemenation

- TS(Time-sharing scheduling class)
- 60 Queues, 점진적으로 조금씩 Time slice를 높임(우선순위 밑으로 갈수록)
- 1초정도마다 우선순위 BOOST

Refined Rules

1. Priority(A) > Priority(B) -> A 실행
2. Priority(A) = Priority(B) -> Round Robin Scheduling을 통해 A와 B 실행
3. 새로운 작업이 들어오면 가장 높은 우선순위에 배치
4. 한 작업이 일정 수준 이상의 시간총합을 사용했을 경우, 우선순위가 내려감
5. 일정시간 S가 지나면, 모든 작업을 제일 높은 우선순위로 Boost

Proportional Share

- Fair-Share Scehduler : 모든 작업이 일정 비율 이상의 CPU time을 받도록 보장
- Turnaround/Response Time 측면에서 효율적이지 못함

Basic Concept

- Tickets : 프로세스가 받아야하는 자원의 지분을 표현
  -> 전체 티켓 중 해당 프로세스의 티켓이 몇개인지를 결정

Lottery Scheduling

- 스케쥴러가 승리 티켓을 뽑음 -> 해당 티켓의 프로세스를 실행
- 작업들이 경쟁하는 횟수가 늘어날수록, 정해진 퍼센테지를 따라갈 확률이 높음

Ticket Mechanisms

- Ticket Currency
  i. 환율을 신경쓰지않고 사용자가 작업별로 티켓 할당
  ii. 시스템이 작업별 currency를 global currency로 변경
- Ticket Transfer : 프로세스가 다른 프로세스에게 Ticket을 줄 수가 있음
- Ticket Inflation : 프로세스가 일시적으로 가지고 있는 티켓을 늘리거나 줄일 수 있음

Implemenation

- 불공정지표(Unfairness Metric) = U : 첫번째 작업이 완료되는 시간 / 두번째 작업이 완료되는 시간
  -> 두 작업이 비슷하게 끝나면 1에 가까워짐
- Lottery Fairness : 작업 길이가 길어지면 확률에 따라서 1에 가까워지지만, 길이가 짧을 시 굉장히 불공정함

Stride Scheduling

- 작업별로 한 걸음의 단위를 설정

- 큰 수 / 프로세스의 티켓 수
- 가장 이동한 거리가 작은 작업을 한 걸음 이동시킴 -> 모든 작업을 완료할때까지 계속 반복
- Deterministic : 랜덤이 대략적으로 정확하고 간단한 스케쥴러를 가지게 해주지만, 확률이기때문에 바르게 작동 안 할 수 있음
  -> Stride Scheduling = A Deterministic fair-share scheduler
- **문제점 :** 다른 작업들이 적당히 진행 된 상태에서, 새로운 작업이 들어오면, 거리가 역전 될 때까지 새로 들어온 작업이 CPU를 독점하게 됨