[Chapter 5]

경영학과/2018100233/김정민

I. CPU burst vs. I/O burst

A. 다른 프로세스가 없다는 가정 하에 하나의 프로세스가 동작하는 sequence

1. CPU burst (CPU 연산)과 I/O burst (I/O 요청 및 처리)가 번갈아 일어난다.
2. CPU-bound process
   - CPU 연산이 I/O 작업보다 많은 비중을 차지하는 경우
   - 머신러닝 프로그램
     - 연산을 수행하기 시작하면 한참을 기다려야 한다.
     - 타임 퀀텀(q)을 거의 다 쓴다 -> 타이머 인터럽트에 의해 레디 큐로 넘어간다.
3. I/O-bound process
   - 일반적으로 우리가 컴퓨터로 사용하는 대부분의 과정
   - GUI(클릭조차도 입출력), 게임 등등
   - 프로세스가 웨이팅 큐, 레디 큐, CPU를 왔다갔다 한다.
   - CPU burst time이 비교적 짧다. (계속해서 I/O 요청 및 처리가 발생)
   - 통계적으로 대부분의 프로세스들이 I/O-bound process

B. Dispatcher

1. Dispatch란?
   - 문맥 전환하는 과정 (스케줄링보다는 조금 더 디테일한 개념)
   - 스케줄링 정책에 따라서 디스패치가 결정된다. (최적화가 필요)
   - Context switch
   - Switching to user mode
   - 프로그램을 다시 수행하기 위해 유저 프로그램의 적정한 위치로 이동하는 것
2. Dispatch latency
   - 컨텍스트 스위치 과정에서 발생하는 지연 시간과 비슷
   - A 프로세스를 멈추고, B 프로세스를 실행하는 사이에 일어나는 지연
   - A의 PCB 상태를 저장하고, B의 PCB 상태를 CPU에 불러오는 과정
   - 이게 짧을수록 시스템 전체의 효율이 좋다.

II. CPU scheduling

A. Preemptive vs. Non-preemptive

1. Non-preemptive scheduling (비선점)
   - 공평하다.
   - 스케줄러는 수행 중인 job이 자발적으로 멈출 때까지 기다린다.
   - 스케줄링이 일어나는 경우
     - 프로세스의 상태가 running에서 waiting이 된 경우 (I/O 요청)
     - 프로세스가 종료된 경우
2. Preemptive scheduling (선점)
   - 스케줄러가 실행 중인 job을 멈추고, 컨텍스트 스위치를 발생시킬 수 있다.
   - 중요한 논점 (후에 나올 개념들)
     - 프로세스가 shared data를 업데이트하는 중간에 스위치가 일어난다면? (critical section)
     - 시스템 콜에서의 프로세스가 선점된다면?

B. Scheduling Criteria

1. CPU Utilization

- CPU가 쉬지 않고 최대한 활용되는 게 좋다. (낭비를 줄임)
- 높을수록 좋다.
- CPU-burst 관점에서 최대화 good

2. Throughput
   - 단위시간당 수행을 완료하는 프로세스의 개수 (성능)
   - 높을수록 좋다.
   - CPU-burst 관점에서 최대화 good
3. Turnaround Time
   - 한 프로세스가 CPU에 들어온 후에 terminated 상태가 될 때까지의 시간
   - 짧을수록 좋다.
   - I/O-burst 관점에서 최소화 good
4. Waiting Time
   - 한 프로세스가 레디 큐에서 스케줄링되기를 기다리는 시간
   - 짧을수록 좋다.
   - I/O-burst 관점에서 최소화 good
5. Response Time
   - 요청이 들어왔을 때 첫 반응이 나타날 때까지의 시간
   - 짧을수록 사용자에게 I/O 빠르게 제공
   - I/O-burst 관점에서 최소화 good

C. Scheduling Goals (시스템마다 목표가 다르다.)

1. All systems (모든 시스템에서의 기본적인 목적)
   - Fairness : 프로세스들이 공평하게 CPU를 사용할 수 있도록 스케줄링
   - Balance : 시스템의 하드웨어 자원들 중 어디 하나 노는 것 없도록 유지
2. Batch systems (하나의 프로세스 독점)
   - Throughput : 시간당 수행하는 job을 극대화
   - Turnaround time : 레디 큐에 들어온 뒤에 종료될 때까지의 시간이 최소화
   - CPU utilization : CPU를 항상 바쁘게 유지하도록 (극대화)
3. Interactive systems (I/O-bound, 상호작용이 많은 시스템)
   - Response time : 레디 큐에서 대기하는 평균 시간 최소화 (빠르게 처리해서 입출력 많이)
   - Waiting time : 웨이팅 큐에서 대기하는 평균 시간 최소화
   - Proportionality : 사용자의 기대를 충족
4. Real-time systems
   - Meeting deadlines : 데드라인에 맞춰서 job을 수행할 수 있도록 스케줄링
   - Predictability : 퀄리티 감소를 피할 수 있도록 스케줄링

D. Scheduling Non-goals

1. Starvation
   - Fairness 깨진 상태
   - CPU를 점유하지 못하고 계속해서 큐에 존재하고 있는 프로세스가 존재하는 상황
   - 잘못된 스케줄링 정책은 starvation 유발 가능하다.
   - 동기화 또한 starvation 유발 가능하다.
     - Ex. Lock이 걸린 동안 다른 쓰레드는 critical section에 접근할 수 없다.
2. Preemptive scheduling (선점)
   - 스케줄러가 실행 중인 job을 멈추고, 컨텍스트 스위치를 발생시킬 수 있다.

- 중요한 논점 (후에 나올 개념들)
  - 프로세스가 shared data를 업데이트하는 중간에 스위치가 일어난다면? (critical section)
  - 시스템 콜에서의 프로세스가 선점된다면?

E. Scheduling Algorithms

1. FCFS (First-Come, First-Served)
   - FIFO (First In, First Out)
   - 레디큐에 들어온 순서대로 CPU에서 연산 수행
   - 실세계와 비슷
   - Non-preemptive
     - 앞선 프로세스가 수행 중이면, 스케줄링에 의해 그 프로세스를 레디큐로 내리지 않는다.
   - Starvation이 발생하지 않는다.
   - 문제
     - Convoy effect
       - 먼저 도착한 프로세스들이 엄청나게 긴 cpu burst time을 가진다면, 평균 대기 시간이 상당히 길어진다.
       - 이는 I/O와 CPU의 불균형이 발생한다.
   - Burst time을 예측하기 어렵기 때문에 실효성이 낮다.
2. SJF (Shortest Job First)
   - CPU burst가 가장 작은 프로세스들부터 수행
   - 이상적인 평균 대기 시간이 나온다.
     - 모든 job들이 동시에 레디큐에 존재할 때
     - 현실성 떨어진다.
   - Non-preemptive
   - 문제
     - 미래에 레디큐에 들어올 프로세스의 CPU burst를 예측할 수 없다.
     - P1 : 1, P2 : 10, Pn : 1
       - 이러한 CPU burst 상황에서 P1이 끝나고 1단위시간마다 Pn이 계속해서 레디큐에 들어온다면 P2는 **starvation** 발생
3. SRTF (Shortest Remaining Time First)
   - Preemptive SJF
     - 우선순위 고려
   - 한 프로세스가 작업을 완료할 때, 혹은 새로운 프로세스들이 들어올 때마다 가장 burst time이 적게 남은 프로세스를 CPU로 스케줄링한다.
   - 이 방법도 현실성이 떨어진다.
4. Priority
   - 각 프로세스마다 우선순위 부여
   - SJF는 예측한 CPU burst time이 다음으로 짧은 프로세스를 기준으로 우선순위를 부여하는 것과 같다.
   - Starvation 문제 발생한다.
     - 우선순위가 낮은 프로세스는 CPU를 할당받지 못할 수도 있다.
     - 계속해서 우선순위가 높은 프로세스가 추가된다면 ..
   - 해결책 : Aging
     - 우선순위가 낮은 채로 레디큐에 존재하는 프로세스들은 일정한 시간 임계점을 넘을 때마다 우선순위를 하나씩 증가시켜준다.

- 우선순위는 프로세스의 기능과 시스템의 목적에 따라 다를 수 있다.
- 각 우선순위마다 priority queues를 링크드 리스트 형태로 유지한다.
    - 하나의 priority queue 안에서 대기하는 프로세스들은 SRTF, SJF, FCFS 등의 방법으로 스케줄링

5. RR (Round Robin)
- Time quantum(q) 개념 등장
    - 타임 퀀텀이 아주 길면 FCFS (들어온 순서대로 모든 프로세스 수행)
    - 타임 퀀텀이 0에 가까우면 계속해서 컨텍스트 스위치 발생 (커널 모드 동작, 스케줄러 호출 증가) -> 커널 오버헤드 증가
- 멀티프로세스 상황 가정
- 타임 퀀텀이 다 차면 running 상태에서 ready 큐로 돌아가서 다음 기회를 기다린다.
- 단점
    - SJF에 비해서 큰 turnaround time을 가진다. (계속해서 컨텍스트 스위치가 발생하므로)
- 그 대신 response time이 비교적 짧다.
    - 타임 퀀텀이 작을수록 response가 좋다.
    - 하지만 dispatch latency보다 타임퀀텀이 작으면 손해가 크다.
    - 컨텍스트 스위치 타임보다는 타임퀀텀을 길게 잡아야 한다.
- 프로세스가 하나 남아서 남은 시간 동안 쭉 수행해도 될 때에도 타임퀀텀마다 스케줄러 호출은 해야한다.
- 타임 퀀텀이 작으면, 커널 오버헤드 증가 -> CPU Utilization 감소 (컨텍스트 스위치 자주 발생)
- 타임 퀀텀이 크면, response time이 안좋다.
- A rule of thumb
    - CPU bursts의 80퍼센트가 타임퀀텀보다 짧아야 한다.

6. Multilevel Queue
- 레디 큐를 여러 개로 나눈다.
    - Foreground (interactive) -> I/O bound processes
    - Background (batch) -> CPU bound processes
        - 하나의 프로세스 쭉 수행하면 좋다.
- 각 큐는 각자만의 스케줄링 알고리즘
    - Foreground – RR
        - Response time이 중요하다 (priority 활용)
    - Background – FCFS
        - CPU utilization 증가
- 단점
    - 우선순위 높은 프로세스가 계속 들어오면, Starvation 가능성 있다.
- Aging이 아니라 Time slice를 통해 해결한다.
    - 80% -> foreground, 20% -> background
    - 이런 식으로 단위 시간을 쪼개서 해결
- 어떤 프로세스가 어떤 큐에 들어갈지를 결정해야 하는데, 스케줄링 알고리즘 만든 사람이 임의로 정해야 한다.
    - 신뢰 이슈가 존재 -> Multilevel Feedback Queue Scheduling 등장

7. Multilevel Feedback Queue
- 유닉스 계열에서 실제로 사용
- 높은 우선순위를 가진 큐에서 아래로 내려올수록 타임 퀀텀을 점점 길게 설정

        - 상위 큐에서 타임 퀀텀을 다 채우고도 할 일이 남은 프로세스(CPU bound일 확률 높음)는 하위 큐로 내려간다.
        - 타임 퀀텀을 채우지 못하고 웨이팅 큐에 들어간 프로세스는 I/O bound 프로세스일 가능성이 높고, response time이 중요하니까 해당 큐의 레디큐로 다시 들어간다.
    - 하나의 큐 안에서는 타임 퀀텀을 이용해서 RR의 방식으로 스케줄링한다.
    - 프로세스의 우선순위는 타임퀀텀을 다 썼는지 여부에 따라서 동적으로 변한다.
    - 이 방법은 하위 우선순위를 가지는 CPU-bound 프로세스들에서 지연이 생긴다.
        - 그래도 기존의 것들보다는 좋은 편

III. Multiple-Processor Scheduling

A. Multi CPUs or Multi-core

1. Symmetric multiprocessing vs. Asymmetric multiprocessing
    - 최근에는 SMP 추세
2. Load balancing
    - 각 코어마다 태스크의 밸런스가 중요하다.
    - Push vs. pull migration
3. Processor affinity
    - Soft affinity
        - 통합 레디 큐
        - 통합 레디큐에 각 코어가 접근해서 태스크를 수행
        - 스케줄링 알고리즘 만들기 쉬움
        - Cache hit rate가 낮다.
    - Hard affinity
        - 각 코어마다 비슷한 태스크끼리 레디 큐 형성
        - Cache hit rate 높다. (CPU 레지스터, L2 cache 등 활용) -> 프로세스 효율이 높다.
        - 스케줄링 알고리즘이 무겁고 어렵다.
        - 코어간에 태스크를 옮기는 작업 (migration)도 필요하다. 이는 쉽지 않다.

IV. Real-Time Scheduling

A. Deadline 맞추는 것이 중요하다. (임베디드 시스템에서 주로 활용)

1. Hard real-time systems
    - "반드시" 데드라인을 지켜야 하는 스케줄링 필요
2. Soft real-time systems
    - 웬만하면 데드라인을 지켜야하지만, 안지킨다고 해서 큰 문제가 일어나지는 않는다.
3. Periodic task in real-time systems
    - 주기마다 태스크를 하나씩 수행하고, 주기보다 작은 데드라인이 정해져있다.
    - 한 주기에서 할 일을 다 했으면, 다음 주기가 찾아올 때까지 레디 큐에 들어가지 않는다.

B. Static vs. Dynamic priority scheduling

1. Static : Rate-Monotonic algorithm
    - 현존하는 모든 임베디드 OS들이 채택하는 방법
    - 스케줄링보다는 우선순위 할당하는 알고리즘
    - 우선순위는 주기의 역수에 의해서 높게 설정된다. (주기가 짧을수록 우선순위)
    - 데드라인, 주기, 실행시간을 고려해서 설계를 잘해야만 데드라인을 지킬 수 있다.
2. Dynamic : EDF (Earliest Deadline First) algorithm
    - 데드라인이 가까울수록 더 높은 우선순위를 가진다.

- 미스가 발생하지 않는다. (이상적)
- 근데 왜 static을 쓰는가?
  - Static 방법을 정확하게 설계하면 문제가 없다. (대부분의 임베디드 시스템에서는 단순하고 반복적인 태스크들이 존재)
  - Dynamic 방법은 스케줄러에 레디큐에 있는 모든 프로세스의 데드라인을 모두 넣어서 비교해야 하는 과정이 필요하다.
  - 이는 오버헤드가 크다.
  - 임베디드 장치는 이런 오버헤드를 감당할 만큼 성능이 좋지 않다.

V. OS별 스케줄링 방법

A. Linux Scheduling

1. Real-time scheduling은 POSIX를 활용한다.
   - Real-time 태스크들의 우선순위는 static algorithm을 활용
2. Global priority scheme (Real-time + Normal)
   - Real-time 태스크(커널이 사용하는 중요한 프로세스들)들은 0부터 99까지의 높은 우선순위를 가진다.
   - Normal 태스크(사용자가 실제로 실행시키는 프로세스)들은 100부터 139까지의 낮은 우선순위를 가진다.
   - Normal 부분은 binary search tree로 관리해서 탐색이 용이하다.

B. Windows Scheduling

1. 프로세스마다 고정적으로 우선순위를 나눠둔다. (우선순위 테이블)
2. 우선순위 테이블 상에서 동일한 우선순위를 가지는 프로세스들은 RR 방식을 통해 스케줄링한다.

C. Solaris Scheduling (Windows와 비슷)

1. 각 쓰레드들은 고정적인 우선순위를 부여받는다.
2. 동일한 우선순위를 가지는 쓰레드들은 RR 방식을 통해 스케줄링한다.

VI. Algorithm Evaluation

A. 알고리즘을 평가하는 4가지 방법

1. Deterministic modeling (이론)
   - 수행해야 하는 일의 양을 정해놓고, 각 알고리즘마다 해당 일을 수행하기 위한 성능을 정의한다.
2. Queueing models (수학적 모델)
   - 기대되는 시스템 매개변수를 연산하는 데 사용되는 수학적 모델
3. Simulation
   - 구현을 할 수 없는 경우에 사용
   - 환경요소들을 유리하게 튜닝할 여지가 있기 때문에 신뢰성이 떨어진다.
   - 시간과 노력은 비교적 덜 든다.
   - 프로세스에 필요한 정보들은 수행되고 있는 프로세스의 실제 정보를 트래킹하는 것이 좋다.
     - 난수 생성보다 좋다는 것
   - Ex. emulation
4. Implementation
   - 가장 좋은 방법
   - 실제 환경에서 실제로 구현해서 실제 데이터를 다룬다.
   - 하지만 시간과 노력이 많이 든다.