CPU scheduling

2022년 4월 25일 월요일 오후 2:45

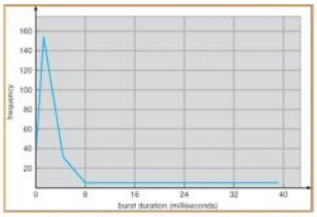
CPU scheduling

CPU Brust

- Cpu가 일을 수행하는 시간 (한번 cpu에 들어가서 최대한 실행되고 나오는 시점까지의 시간)
- 프로세스가 실행완료되어 소멸될때까지의 시간과는 다름(여러이유로 프로세스는 cpu를 여러 번 사용할수있기 때문이다)
-

IO Brust

- I/O요청 ~ I/O 완료의 기다리는 시간
- 프로세스의 실행 과정은 수많은 cpu버스트와 io 버스트의 순환 과정이다
- Cpu 버스트는 짧은데 비해 io 버스트는 길다



한 프로세스에 대한 CPU 버스트와 I/O 버스트 대한 그래프]

화면 캡처: 2022-04-25 오후 2:53

CPU Scheduling

Ready Queue상태 프로세스 중 하나를 골라 해당 프로세스에 cpu코어에 할당하고 특정 프로세스가 cpu 사용을 독점하지 못하도록 (time sharing)관리

Preemptive scheduling

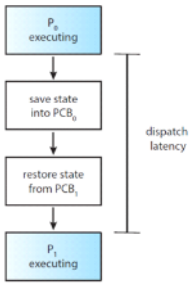
- 한 프로세스에 할당된 cpu를 강제로 뺏음(우선순위,time 만기 등의 이유) 다른 프로세스 사용하게 한다
- 현대 컴퓨터는 이 방식을 사용한다

Non Preemptive scheduling

- 강제로 뺏지 않고 다 할때까지 기다린다 한번 할당 받은 cpu 가능한 계속 사용

Dispatcher

CPU 스케쥴러가 선택한 프로세스를 실제로 실행할수있도록 한다
다음으로 CPU 코어를 할당 받을 프로세스가 선택되면 기존의 실행 중이던 프로세스를 멈추고 선택된 프로세스에 cpu 제어권을 넘기는 역할 수행



[디스패치 지연시간의 모식도]

화면 캡처: 2022-04-25 오후 2:58

역할

- Context switching 을 발생, 진행 시킨다
- 프로세스가 중단되었던 시점을 찾아 실행 시킨다
- Cpu 스케줄링을 하기 위해 진입한 커널 영역을 빠져 나와 해당 cpu 스케줄러에 의해 선택된 프로세스가 이전에 실행 중이던 실행 시점을 찾아 (context switching 과정)해당 시점으로 복귀 시킴으로써 해당 프로세스의 제어권을 사용자에게 반납 (사용자 모드로 전환 시킴)

Dispatch Latency

순수하게 하나의 프로세스가 모종의 이유 (우선순위,스케줄러등)로 문맥 전환 될때 다른 프로세스 선택을 위해 cpu 스케줄러를 호출한 시점 ~ 스케줄러에 의해 선택된 프로세스의 PCB 정보를 복구한 다음 실행 영역을 사용자 영역으로 전환시킨 시점까지의 모든 작업을 수행하는데 걸리는 시간 그림에서 확인 가능

Algorithm

성능 평가 척도

- **CPU 이용률 (CPU utilization)**
  - 보통 한 클럭 틱 단위 동안 해당 CPU 혹은 코어가 일을 한 시간의 양
  - 가능한 한 CPU에 계속 일을 시킴으로써 CPU 이용시간을 최대한 극대화해야함
  - 실제 CPU 이용류은 40-90% 왜냐하면 실제는 IO Brust 시간이 더 길기 때문이다

- **처리율(Throughput)**
  - 단위 시간 동안 얼마나 다양한 프로세스들을 처리 하는가?

- **반환시간(tunaround time)**
  - 프로세스가 처음으로 준비 큐에 도착한 시점~ 프로세스 종료되기까지의 시간

- **준비큐에서 대기시간(waiting time in ready queue)**
  - 프로세스가 준비큐에 들어온 시점 ~ CPU 스케쥴러에 의해 선택되어 준비큐에서 나가는 시점의 시간 (ready queue에서 대기한 시간)

- **응답 시간(response time)**
  - 프로세스가 생성되어 준비큐에 진입한 시점 ~ 처음으로 CPU를 할당 받은 시점의 시간
  - 즉 처음으로 사용자에게 응답을 보인 시간

종류

- **FCFS**
  - 준비 큐에 들어온 순서 대로 프로세스 처리 하는 가장 간단한 알고리즘
  - **Cons**

```
ex) -1) CPU Bound 프로세스가 먼저 오고, 그 뒤에, I/O Bound 프로세스가 오는
        경우
        ⇒ 먼저, 긴 시간 동안 CPU Bound 프로세스 실행
          ☞ 이 긴 시간 동안, I/O 장치는 일을 안함
        ⇒ 이후, CPU Bound 프로세스 실행 후, 비교적 CPU 버스트가 짧은, I/O
          Bound 프로세스 실행
        ⇒ I/O Bound 프로세스는 CPU를 굉장히 조금만 사용
        ⇒ 모든 I/O Bound 프로세스가 아주 짧은 시간 후, I/O 대기 상태가 됨
        ⇒ 이 때 동안, CPU는 노는 상태가 됨. I/O Bound 프로세스는 아직 I/O
          수행
        ⇒ CPU가 일을 안하게 됨

    -2) I/O Bound 프로세스가 먼저 오고, 그 뒤에, CPU Bound 프로세스가 오는
        경우
        ⇒ 모든 I/O Bound 프로세스들이, CPU를 짧게 사용하고 I/O로 이동
        ⇒ 모든 I/O Bound 프로세스들이, CPU Bound 프로세스가 CPU를 다 사
          용할 동안, I/O를 마치고 돌아옴
        ⇒ CPU Bound 프로세스가 일을 마치자마자, 뒤에서 마침 대기하던 I/O
          Bound 프로세스가 CPU에서 실행
        ⇒ 곧바로, CPU Bound 프로세스가 I/O로 이동
        ⇒ 위의 상황보다, CPU 사용률/ I/O 장치 사용률 이 높음
```

화면 캡처: 2022-04-25 오후 3:22

평균 대기시간이 천차만별 ,자원 균등화 실패

- **SJF**
  - 프로세스의 다음 CPU Brust 타임이 작은 순서대로 프로세스 실행
  - **pros**
    - 준비큐 상에서 다음 CPU 버스트가 작은 순서대로 프로세스가 실행되므로 최소의 평균 대기 시간을 항상 보장한다
    - ->
      CPU 버스트가 긴 프로세스의 대기 시간이 늘어나겠지만, CPU 버스트가 짧은 프로세스
      의 경우에는, 대기 시간이, CPU 버스트가 긴 프로세스의 대기 시간이 늘어난 양보다 훨
      씬 줄어들기 때문
  - **Cons**
    - 기근 현상 : 제일 긴 프로세스는 영원히 CPU 할당 받지 않을수도 있다
    - 구현 방법 어렵다

- **ROUND ROBIN (원형 큐)**
  - 준비 큐의 헤드 부분부터 꼬리 부분까지 돌아가며 각각의 프로세스에 굉장히 미세한 단위시간을 부여하고 단위시간이 끝나면 타이머 인터럽트가 걸려서 CPU를 뺏어오는 일련의 과정을 반복하는 스케쥴링 알고리즘이다
  - 스케줄러는 원형 큐를 돌면서 차례차례 단위시간만큼 프로세스들이 실행된다
  - 또한 타이머라는 하드웨어가 추가적으로 필요하며 해당 타이머는 매번 새로운 프로세스가 CPU를 할당 받을때마다 설정된 양자화 시간만큼 세팅되어 해당 프로세스가 양자화 시간을 넘겨서까지 CPU를 사용한다면 타이머 인터럽트를 발생 시킨다. [타이머 인터럽트가 발생한 프로세스는 running -> ready로 constext switching됨]
  - **Preemptive scheduling** 에서 사용되는 알고리즘
  - **Pros**
    - 응답시간이 빨라짐(최소의 응답 시간이 보장 되기때문이다)
    - 어떤 프로세스 하나라도 빠짐 없이 모두 실행됨
  - **Cons**
    - 양자화 시간의 크기에 따라 성능이 더 좋아질수도 나빠질수도있다
    - **극단적으로 큰 양자화 시간**
      - **FCFS 기법과 큰 차이점 없음**
    - **극단적으로 작은 양자화 시간**
      - **Context switching 많이 일어남 overhead증가**
  - 양자 시간이라는게 CPU에 할당된 시간이라고 보면 이해하기 쉬워진다
  - 즉 양자 시간이 너무 작으면 타이머인터럽트가 지나치게 많이 발생하게 된다는 의미이다

- **Priority scheduling**
  - 우선순위가 높은 프로세스에 CPU를 할당해주는 알고리즘이다
  - 우선순위가 같다면 FCFS를 사용한다
  - Preemptive(우선순위에 따라 그 즉시 높은 프로세스에게 할당), non preemptive(우선순위가 높아도 끝날때까지 기다림)둘다 적용 가능
  - **Pros**
    - 우선순위 따라 우선적으로 실행 가능

- **Cons**
  - 기근 현상 발생 : 우선 순위가 낮다면 영원히 실행이 안될수도 있다.
  - 언제 실행된다는 보장이 없다
  - 해결 방법 : 노화 기법
    - 우선 순위가 낮아서 계속 대기하는 프로세스에게 우선순위 점수를 +1해서 높이는 방법

- **우선 순위 큐 구현**

  - Multilevel Queue Scheduling
  - Multileve Feedback Queue scheduling

  2가지 방법이 있는데 자세한 내용을 보고 싶다면 05장 CPU 스케쥴링 파일 참고할것 + 키워드는 있으니까 구글링도 해보세요

멀티 프로세서 시스템에서의 스케쥴링

멀티 프로세서 (multi processer)-> 처리 방식의 일종일 뿐이다

멀티 프로세서 시스템(CPU 2개 이상)

비대칭 다중 처리 (Asymmertric Multiprocessing ,AMP) -> 단점이 커서 거의 안쓰는 방식

- 하나의 마스터 프로세서가 CPU 스케쥴링 , 입출력 처리 , 하드웨어 접근 제어 및 시스템 호출등 의 커널을 동작 시키고 Server프로세서가 사용자 프로세스를 실행시키기만 하는 시스템
- 운영체제만 실행되는 CPU /코어를 **Master Server**
- 그외의 사용자 프로세스 실행시키기만 하는 CPU /코어를 **Slave**
- **Pros**
  - 하나의 master만이 제어를 하면 되니까 시스템 구조가 간단해진다
  - 커널 영역,사용자 영역이 물리적으로 완전히 분리됨

- **Cons**
  - 하나의 마스터가 모든 시스템 프로세스를 도맡으니까
  - Slave가 커널에 다수의 서비스를 한번에 요구하면 병목 현상이 일어날수도 있다

대칭 다중 처리 (Symmertric Multiprocessing SMP)-> 현대 컴터가 사용하는 방식

- 각각의 CPU /코어가 독자적으로 스케쥴링/커널영역 서비스 실행을 하는 구조
- 각각의 CPU /코어에서 스레드 처리하는 방법 2가지
  - **Common Ready Queue**
    - 모든 CPU/코어가 하나의 공통적인 준비큐를 대상으로 스케쥴링 실행
    - 다수의 스케쥴러가 동시에 준비큐에 접근하는 경우 병목현상 발생
    - 또한 동시에 접근해서 동시에 스케쥴링하여 동일한 프로세스가져오는 경우 일관성 문제 있다
    -
  - **Per Core Run Queue**
    - CPU/코어 각자가 본인의 준비 큐 존재
    - 대칭 다중처리 기법에서 가장 많이 사용하는 기법이다
    - 잘못하면 스레드가 몰리는 가능성이 있다 (unbalance 문제)