Ch5. Process Scheduling (완)

Created	@May 27, 2020 8:48 PM
Tags	운영체제

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/2b2a11fc-1cd8-4e23-a0d0-7c6cde67fc34/CH5._Process_Scheduling.pdf

Process Scheduling

💡 **스케쥴링의 단위는 Thread가 맞다.**
옛날에는 프로세스 단위로 생각했고, 지금도 교과서에서는 프로세스 스케쥴링이라는 말을 쓰기도 한다.
"CPU 스케쥴링", "프로세스 스케쥴링", "스레드 스케쥴링" 세 용어가 혼용되어 사용.

💡 Remind: CPU Scheduling : 어느 프로세스가 돌아갈 것이냐?

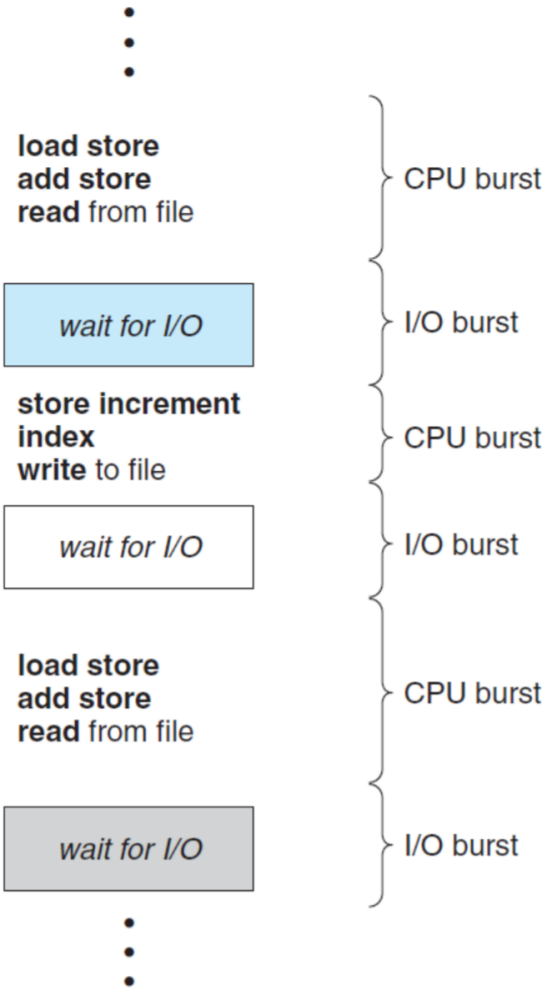
프로세스 스케쥴링은 왜 필요한가?

- **CPU Utilization을 최대화하기 위해.**
- **프로세스들 사이의 fairness** : 어느 프로세스는 계속 돌아가고, 어느 프로세스는 조금만 돌아간다면 불공평할 것이다.

CPU-I/O Burst Cycle

burst : 갑자기 막 증가하는 것

- **CPU Burst** : CPU가 많이 사용되는 것
- **I/O Burst** : I/O가 많이 사용되는 것

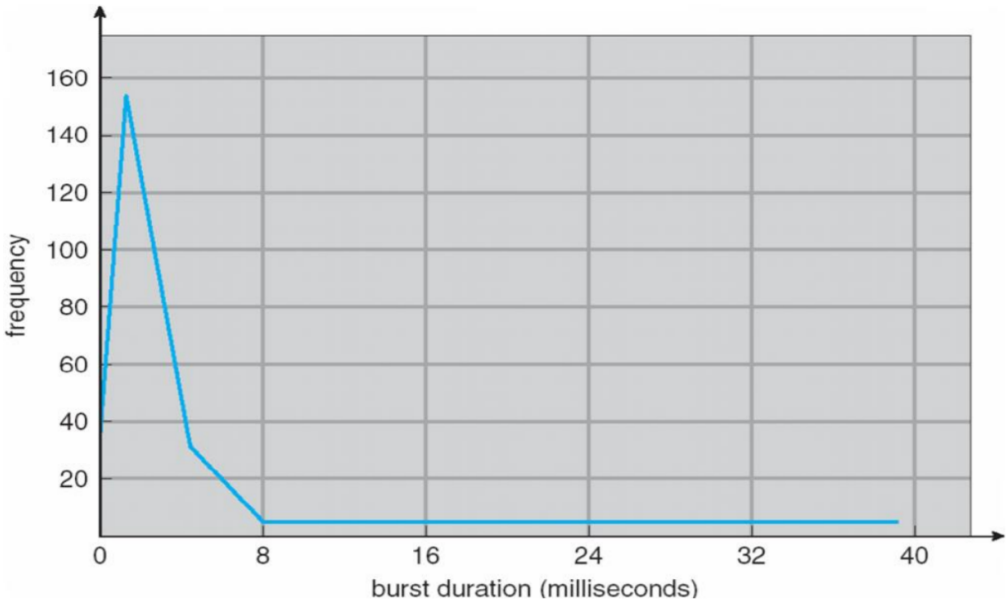


CPU 쓰다가, 'read from file' 호출하고 I/O 기다리다가, 다시 CPU 쓰다가, I/O 기다리다가, ...

모든 프로세스는 CPU Burst, I/O Burst 사이를 전환하면서 실행된다.

CPU Burst가 짧고 I/O Burst가 긴 프로그램도 있고, 그 반대인 프로그램도 있다.

CPU Burst duration



|—CPU—| |IO| **|——CPU——|** |IO| **|—CPU—|**
대충 이런 식으로 진행될 때, 모든 CPU Burst의 길이의 빈도를 측정해봤더니 위처럼 히스토그램이 그려지더라.

- **짧은 길이의 CPU Burst의 빈도가 아주 많았다.**
- **긴 CPU Burst는 별로 없었다.**
  - → **일반적인 프로그램은 CPU Burst 길이가 짧고, I/O에 대부분의 시간을 쓴다.**
    I/O를 하지 않고 CPU만 계속 하는 프로그램은 거의 없다는 소리.
- I/O bound 프로그램이라면 짧은 CPU Burst의 빈도가 더 많이 나올 것이다. (왼쪽 위로 치우침)
- CPU bound 프로그램이라면 CPU에 대부분의 시간을 쓰니 긴 CPU Burst의 빈도가 더 많이 나올 것이다. (오른쪽 위로 치우침)

CPU Scheduler

Ready Queue에서 다음에 실행할 프로세스를 선택하는 것 → **CPU 스케쥴러, 또는 Short-term scheduler가 한다.**

> 💡 Remind: Ready Queue : PCB의 Linked List.
> 링크드 리스트를 순회하면서 다음 실행할 프로세스를 선택한다.

스케쥴러가 결정을 내리는 경우 4가지

1. 프로세스가 **Runnng → Waiting**으로 바뀔 때 - **I/O request 또는 wait()**

   → 반드시 스케줄링 결정을 해야 한다.

2. 프로세스가 **Running → Ready**로 바뀔 때 - **(timer) interrupt**

   리눅스에서는 주기적으로 (1ms) timer가 돌아서, timer expired(timeout)된 애들은 다시 Ready Queue로 들어간다.

   → **스케쥴링 결정을 해도 되고 안 해도 된다.** 다음에 계속 똑같은 프로세스 돌려도 되니까, 새 프로세스를 안 골라도 되는 것. (Policy에 따름)

3. 프로세스가 **Waiting → Ready**로 바뀔 때 - I/O 완료. 이제 User CPU를 쓸 때가 됐다는 것.

   → **스케쥴링 결정을 해도 되고 안 해도 된다.** 새 프로세스가 Ready에 들어왔다고 해서, 무조건 새 프로세스를 골라야 한다는 건 아니다.

4. 프로세스가 **Terminated**된 경우

   → 반드시 스케쥴링 결정을 해야 한다. 프로세스가 종료되면 당연히 다음 실행할 프로세스를 정해줘야 한다.

Preemptive vs. Nonpreemptive

스케쥴러를 두 종류로 나눠볼 수 있다.

- **Nonpreemptive scheduler**
  - **1번, 4번 경우에만 스케쥴링이 일어남.**
  - 프로세스가 끝나거나 Wait queue로 들어갈 때만, 필요할 때만 스케쥴링이 일어남.
  - 스케쥴링을 강제하지 않음.
  - 프로세스가 자발적으로 CPU를 release하거나, terminate될 때까지 계속 돌아간다.
- **Preemptive scheduler**
  - **모든 경우에 스케쥴링이 일어남.**
  - 스케쥴링이 강제됨.

돌아가고 있는 프로세스를 Reschedule할 수 있는가?

- Nonpreemptive : X
- Preemptive : O

Preemptive scheduling

Preemptive : Forcing the process to give up the processor.

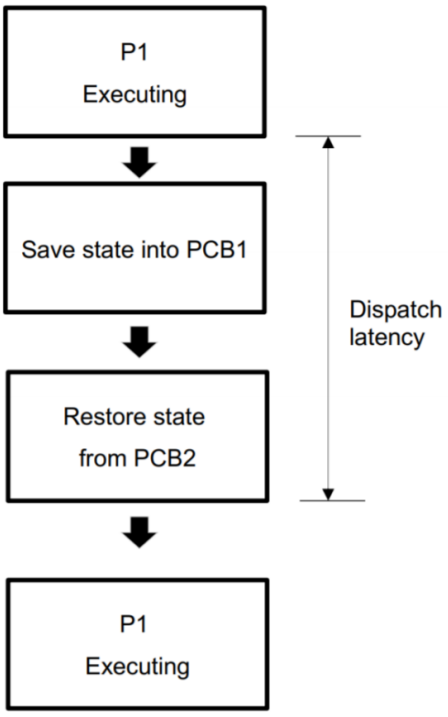
문제가 있다!

- **Race condition의 가능성.**
  - Race condition : 대충 두 개를 병렬 실행하는데 실행할때마다 결과가 다르게 나오는 것. → *Ch6. Synchronization*
  - **한 프로세스가 데이터를 업데이트하고 있는데 Preemption이 일어날 수 있다.**
    → 다른 프로세스는 일관성 없는 데이터를 읽게 된다.
    Data consistency가 깨질 수 있다.
- 커널 디자인의 문제
  - **프로세스가 커널 모드에서 시스템 콜을 처리하고 있는데, Preemption이 일어날 수 있다.**
    → 마찬가지로 커널 자료구조가 inconsistent state에 빠진다.
  - 커널 영역은 Application보다 더 critical해서, 작은 값이라도 잘못되면 crash나기 쉽다. 더 큰 문제다.
    - One solution : **시스템 콜 핸들링할 때는 Preemption을 잠시 Disallow.**
      → Downside : Preemption을 Disallow하면 (당연히) **당장 시작해야 하는 프로세스가 시작 안 될 수도 있음.**
      Real-time computing 같이 계산이 바로바로 끝나야 하는 시스템에서는, 중요한 계산을 먼저 해야 할 수도 있는데...

Dispatcher

Dispatcher : 선택한 프로세스에게 CPU 컨트롤을 넘겨주는 일을 하는 Component.

Dispatch latency : 한 프로세스가 끝나고 다른 프로세스가 시작될 때까지의 시간.



마지막에 P1 아니고 P2

디스패처의 책임

- Context switch
- User mode로의 switch
- User program을 재시작할 때, 적절한 위치로 jumping.

Scheduling Criteria

스케쥴링 결정을 할 때, 무엇을 기준으로 삼을 수 있는가?

- **CPU utilization**
  - 이상적으로는 CPU busy를 100%로 유지하고 싶지만, 이건 실제로는 문제가 있는 것.
    - → CPU saturation : CPU가 모든 걸 사용해서 더 이상 공간이 없는 것.
  - 실제로는 30~40% 정도가 최적. 이 정도로 유지한다.
- Throughput
  - 단위 시간당 처리되는 일의 수를 최대화 → 짧은 것만 선택하면 되긴 된다.

- Fairness
- 등등...

스케쥴링 알고리즘 네 가지

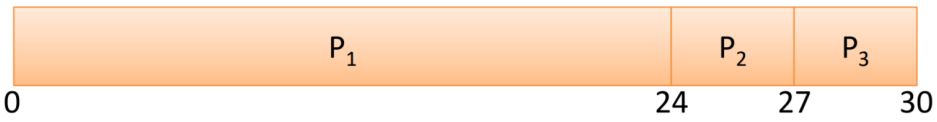
- FCFS, SJF, SRF, RR

FCFS : First Come, First Served

- **먼저 들어온 프로세스를 먼저 할당**
- **Nonpreemptive**. 구현 쉬움.
- 평균 대기 시간(AWT)이 길다.

Process	Burst Time
P_1	24 ms
P_2	3 ms
P_3	3 ms

- Suppose processes arrive in the order of P_1, P_2, P_3. The Gantt chart for the schedule is:



  - waiting time for P1=0, P2=24, P3=27
  - Average waiting time

    (0+24+27)/3 = 17 ms

- 그런데 [P2, P3, P1] 순서대로 오면 평균 대기시간 = (0+3+6)/3 = 3ms 나온다.
  - **Convoy effect** : Burst time이 큰 프로세스가 먼저 들어오면, 그 뒤에 들어온 프로세스들은 오래 기다려야 함.

SJF : Shortest-Job-First

- **짧은 거 먼저!**
- 최적 AWT 나온다.
- Preemptive일 수도, Nonpreemptive일 수도.
  - Nonpreemptive인 경우.
    - CPU burst 시간을 미리 알 수 없다. → 예측.

- **Exponential Average**
- t_n = n번째 CPU burst 시간, τ_n = n번째 CPU burst 예측치.

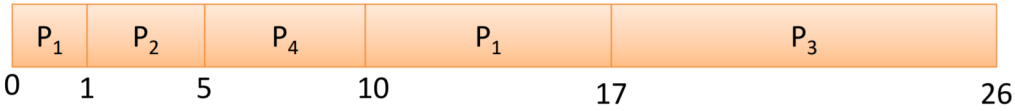
$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

- α, τ_0만 주어지면 구할 수 있다.

- **Preemptive인 경우, SRF(Shortest Remaining Time First)라고 부른다.**
  - **수행되던 도중, 남은 수행 시간이 더 작은 애가 있으면 걔 먼저 수행.**
    - **남은 수행 시간** : Burst time에서 그 프로세스 수행된 시간 뺀 것. 아직 수행 안 된 애, 지금 들어온 애들은 그냥 Burst time 그대로이다.
    - 1ms 단위로 그냥 끊어서, 남은 시간 체크하면서 보자.

  - Shortest-remaining-time-first scheduling

Process	Arrival Time	Burst Time
P_1	0	8 ms
P_2	1	4 ms
P_3	2	9 ms
P_4	3	5 ms



  - Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 6.5 ms

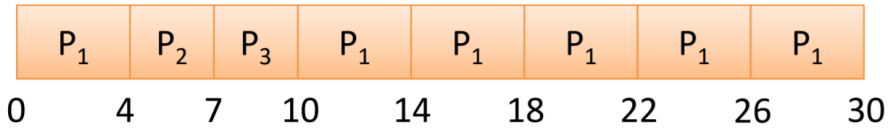
RR

프로세스를 Time quantum만큼 잘라서, FCFS처럼 돌린다. → Preemptive.

- Ready queue 맨 앞에서 하나 골라서, Time quantum만큼 돌리고 (일찍 끝났으면 걍 끝내버리고), 남았으면 큐 맨 뒤에 넣는다.
- 이를 반복.

- Assume the time quantum of 4 ms

Process	Burst Time
P_1	24
P_2	3
P_3	3



- Average waiting time

$$(6 + 4 + 7)/3 = 5.66$$

- **Time quantum**은 어떻게 잡아야 하는가?
  - 너무 크면, FCFS랑 똑같다.
  - 너무 작으면, Context switch 오버헤드가 커진다.
  - 보통 **Time quantum >> Context switch time**이어야 하고,
    Time quantum = 10ms, Context switch time = 10us 정도.

Priority Scheduling

프로세스에 우선순위를 매겨서, 큰 순서대로 스케쥴링.

> 💡 **SJF도 Priority Scheduling의 일종.**
> 여기서 우선순위는 Burst time의 역수. (Burst time이 작을수록 우선순위가 크다.)
> **FCFS도 Priority Scheduling의 일종.**
> 프로세스의 우선순위가 모두 동일.

Priority에는 두 종류가 있다.

- **Internally defined** : 측정 가능한 양 (time limits, open file 개수, ...)
- **Externally defined** : OS 외부에서 정한 기준 (돈을 얼마나 썼는지, ...)

Preemptive, Nonpreemptive 둘 다 가능하다.

- **Preemptive**이면, 우선순위가 더 높으면 현재 프로세스 밀어내기.
- **Nonpreemptive**이면, 바로 다음 순서에.

Starvation Problem (=indefinite blocking)

Priority Scheduling의 문제.

- **낮은 우선순위의 프로세스들은 실행되지 않을 수도 있다.** 얼마나 기다려야 하는지 보장도 없다.
  - → **Aging : Ready queue에 오래 있을수록 프로세스의 우선순위를 높여주자.**
  - Waiting time에 bound를 두는 것.

Multilevel Queue Scheduling

프로세스들을 여러 종류로 나눠서, 종류마다 나눈 Ready queue에 각각 다른 알고리즘 적용. RR, FCFS 등등...

그러나, 문제점.

- **한번 큐에 들어가면, 다른 큐로 이동하는 게 어려움.** 유연하지 않음. 판단이 잘못되면 변경하기 쉽지 않음.
- → **MFQS(Multilevel Feedback Queue Scheduling)**
  - 맨 처음에는 가장 우선순위가 높은 큐로 들어가는데, Time quantum이 작음. 거기서 끝나지 못하고 Preempted되면, Time quantum이 두 배인 그 다음 큐로 이동. 거기서도 완전히 못 끝나고 Preempted 되면, 맨 마지막 큐인 FCFS 큐로 이동.
  - 얘도 문제점 : **Starvation. 우선순위 높은 애만 줄창 들어오면... → Aging.**