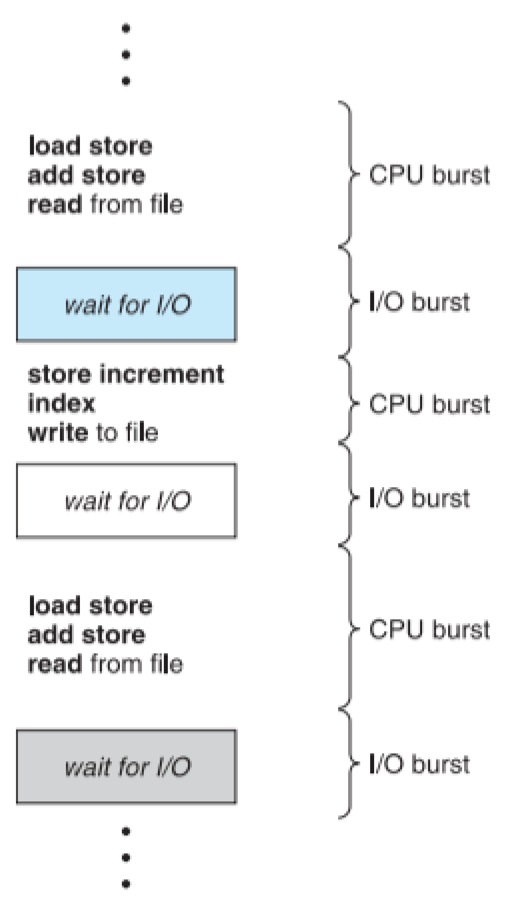
concepts & algorithms

용어 정리

- workload
  - a set of job descriptions
  - ex) arrival time, run time, etc
  - workload 가정? (assumptions)
    - 각각의 job은 같은 시간 동안 실행
    - 모든 job은 동시에 arrive (시스템에 제출)
    - 한번 시작하면, 반드시 끝이 나야 함
    - 모든 job은 CPU만 사용
    - job의 런타임을 알 수 있어야함
- scheduler
  - job을 언제 run시킬지 결정하는 logic
- metric
  - scheduling quality의 측정 단위
  - ex) turnaround time, respond time, fairness, etc

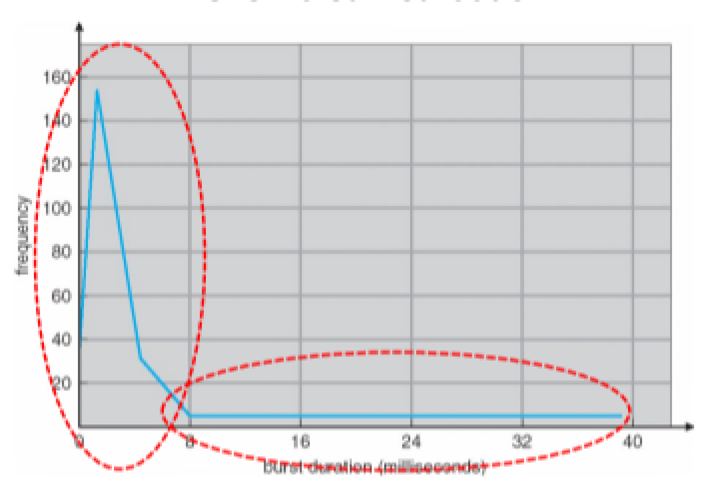
Basic Cooncepts

- CPU를 최대로 이용할 수 있는 건 multiprogramming 덕분
- multiprogramming이 job을 organize해서 CPU는 항상 실행할 job이 있음
- CPU를 집중적으로 사용하는 CPU Burst, IO를 집중적으로 처리하는 IO Burst
- IO Burst 사이 CPU Burst를 적절히 분배하는 것이 관건



- CPU burst는 수가 많지만 걸리는 시간은 적음
- IO burst는 반대로 수가 적지만 오래 걸림

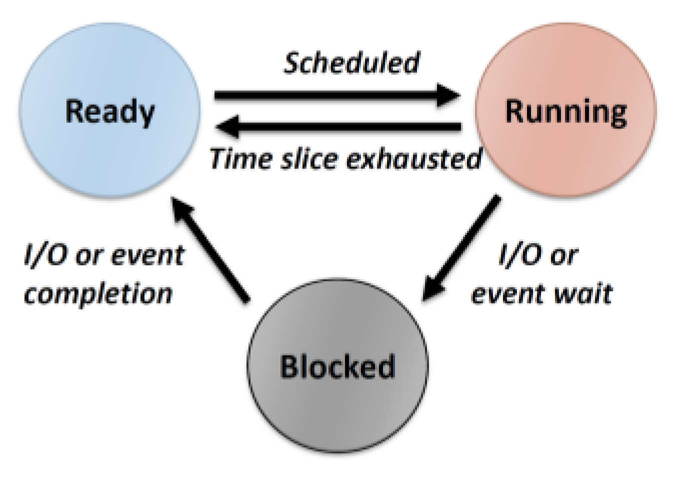
CPU Burst Distribution



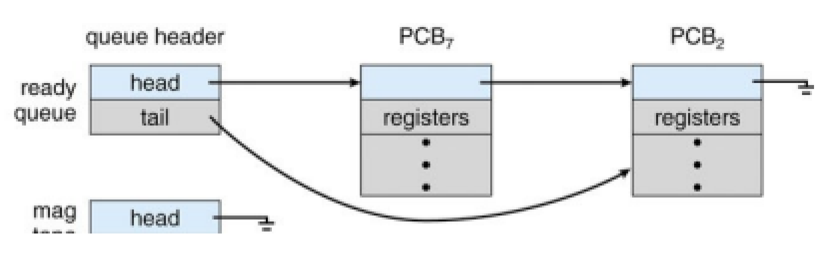
A large number of short CPU bursts and
a small number of long CPU bursts

CPU Scheduler

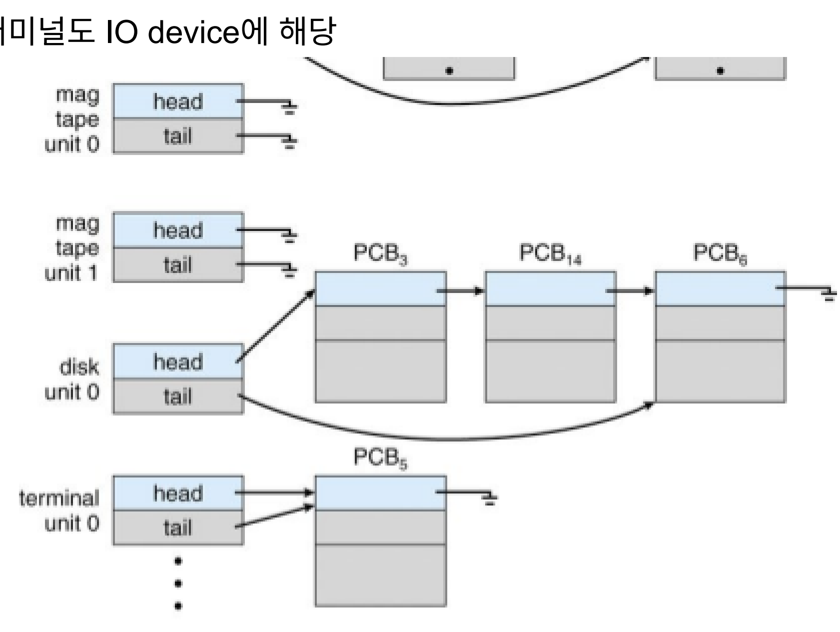
- CPU Scheduler가 메모리에 있는 프로세스 중 하나를 골라 CPU에 할당
- 프로세스가 다음과 같은 상태변화 상황에 놓일 때 CPU Scheduling
  - running -> waiting (ex : IO request) [nonpreemptive]
  - running -> ready (ex : interrupt, especially time slice exhausted) [preemptive]
  - waiting -> ready (ex : IO completion) [preemptive]
  - 혹은 termination [nonpreemptive]



- nonpreemptive scheduling에서 프로세스는 terminating이나 switching to the waiting state로 CPU에 대한 제어권을 유지하며 이는 직접 release 하기 전 까지 유지됨
- preemptive scheduling 은 특정 HW 플랫폼에서만 쓰임
  - 타이머같은 특정 HW가 preempive scheduling에 사용됨
  - 주의점
    - 공유 데이터에 접근할 때 공유 데이터를 보호할 메커니즘 필요
      - 적절한 조치가 없으면 data inconsistency 발생
    - 커널 모드 안에 있을 때 preemption이 발생하면 커널 코드가 중간 상태에서 멈추므로 꼬일 수 있음
    - 중요 OS acitivities(ex : context switch)를 수행하는 동안 인터럽트가 발생하면 작업이 중간에 끊길 수 있음
- CPU Scheduler는 프로세스의 scheduling queue를 관리
  - job queue : 시스템에 존재하는 모든 프로세스의 집합
    - 어떤 사용자가 프로그램을 실행하려고 하면 그 요청은 일단 job queue에 등록
    - 메모리 공간이 충분해지면 그 때 메모리에 올라가고 ready queue에 등록
  - ready queue : 메인 메모리에 존재하는 프로세스 집합



  - device queue : IO device를 waiting 중인 프로세스 집합
    - 터미널도 IO device에 해당



  - wait queue : 특정 이벤트들(device, timer, message, ...)을 기다리는 프로세스들의 큐
  - 프로세스는 이 다양한 큐들을 여기저기 이동

Dispatcher

- CPU를 선택된 프로세스에게 파견
- Dispatcher module이 CPU scheduler가 선택한 프로세스에게 CPU 제어권을 줌
- Dispatcher가 하는 일
  - switching context
  - user mode로 전환
  - user program을 재시작하기 위한 다음 명령어 위치로 이동
- Dispatch latency
  - Dispatcher가 한 프로세스를 중단하고 다른 프로세스를 시작하는 데 걸리는 시간

Scheduling Criteria

- 프로세스 스케쥴링 성능을 평가할 때 사용하는 지표
- CPU Utilization : 얼마만큼 CPU를 바쁘게 유지하는가
- Throughput : 단위 시간 당 임무를 완수한 프로세스의 개수

시간 종류

- Turnaround time : 특정 프로세스를 수행하는 데 걸리는 시간
  - completion time : 프로세스가 종료된 시간
  - arrival time : 프로세스가 시스템에 도착한 시간
  - turnaround time : completion time - arrival time
  - 갔다가 되돌아오는 걸 turnaround라 함
- Waiting time : 프로세스가 ready queue에 머무른 시간
- Response time : CPU가 request를 받고 프로세스의 첫 실행 시각까지 걸린 시간
  - firstrun time : 프로세스가 처음 시작된 시간
  - response time : firstrun time - arrival time

Scheduling Criteria를 하는 이유?

- CPU Utilization과 throughput을 극대화하고 turnaround time, waiting time, respose time을 최소화
- average를 optimize하기 보단 minimum과 maximum value를 optimize
- respose time에 있어서 편차를 최대한 줄이기 위해

Scheduling Algorithms

- 6가지의 스케쥴링 알고리즘

알고리즘 목록

- FCFS (First-Come, First-Served) 스케쥴링
- SJF (Shortest-Job-First) 스케쥴링
- Priority 스케쥴링
- RR (Round-Robin) 스케쥴링
- Multilevel Queue 스케쥴링
- Multilevel Feedback Queue 스케쥴링

Scheduling Metrics

- 성능 평가 기준 : Turnaround time
  - job이 시스템에 도착한 시점에서 job이 complete되는 데 걸린 시간
- 또다른 평가 기준은 fairness
- 스케쥴링에는 performance와 fairness의 trade-off가 존재

FCFS(FIFO)

- 특징
  - 구현하기 간단
  - convoy efftect
    - convoy : 호송대, 행렬
  - long process 때문에 뒤에 있는 short processes들이 오래 기다려야 하는 경우가 생김
  - non-preemptive므로 프로세스의 CPU 제어권을 강탈할 수 없음
- 10초가 걸리는 A, B, C가 차례대로 도착했다고 가정
- average turnaround time
  - A의 turnaround time은 10초
  - B는 도착하고 10초 기다린 후 10초 실행해야하므로 turnaround time은 20초
  - C는 30초
  - 그러므로 (10 + 20 + 30) / 3 = 20s

FIFO의 단점

- 만약 특정 job이 유독 동작시간이 길다면?
- average turnaround time이 커짐
  - 작업이 n개 들어오고 맨 앞 작업이 30초가 걸린다면 뒤의 n개 작업들은 각자 30초를 더 기다려야해서 turnaround time이 배로 커짐

SJF(Shortes Job First)

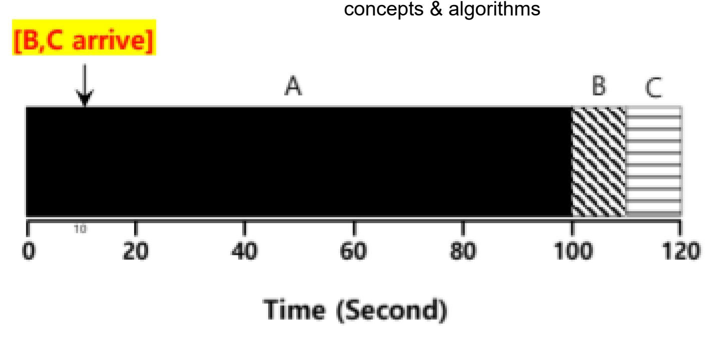
- 가장 짧은 job부터 수행, min heap처럼.
- Scheduler가 강제로 CPU 제어권을 뺏지 않는 non-preemptive scheduler여야 함
- (A, 100), (B, 10), (C, 10)가 왔어도 B, C 부터 수행
- average turnaround time : (10 + 20 + 120) / 3 = 50s

SJF의 단점

- 만약 A, B, C가 동시에 오지 않고 A가 충분히 먼저 온 경우

- SJF에선 non-preemptive로 A의 제어권을 뺏을 수 없어 A가 오래 독점하게 됨....

concepts & algorithms

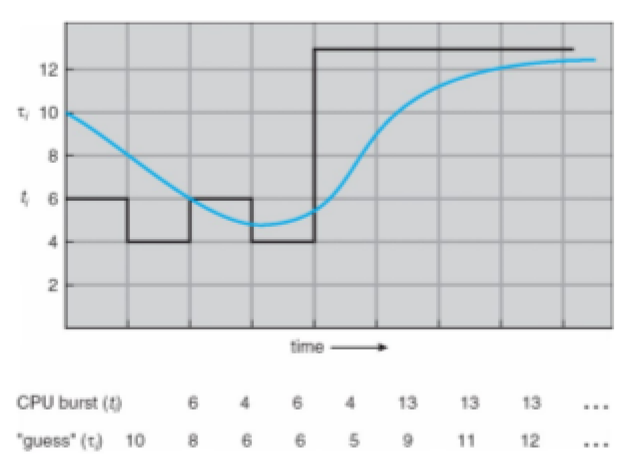


$$\text{Average turnaround time} = \frac{100 + (110 - 10) + (120 - 10)}{3} = 103.33\ sec$$

- B, C가 10초에 도착했으므로 각각 - 10

SJF는 최적의 스케쥴링 방식?

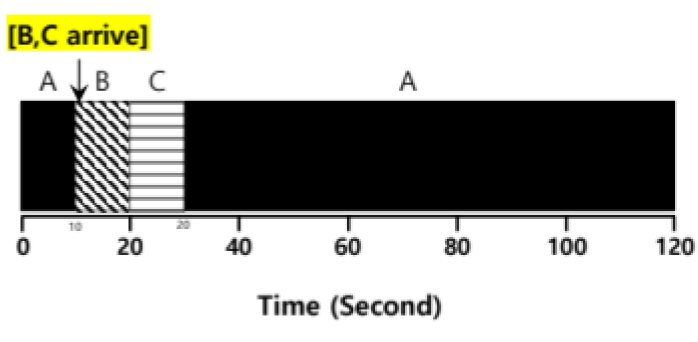
- average waiting time을 가장 적게 만들 수 있는 방식 (이상적인 경우에서만)
- 문제는 다음 CPU burst가 얼마일지 모름
- 해결 방법?
  - 유저가 직접 CPU burst 값을 입력 : 불가능...
  - 이전 CPU 사용 기록을 바탕으로 예측
    - linear regression 사용
    - exponential average of the measured lengths of previous CPU burst 사용
      - tn = n번째 실제 CPU burst
      - Tn+1 = 예측한 n + 1번째 CPU burst
      - 상수 a, 보통 0.5
      - $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n.$



      - guess 행 한 열씩 밀림 (....)
      - a를 0.5라 가정하고 2번째부터 예측
      - T1이 8, t1이 4이므로 T2 = 4 + 2 = 6

STCF (Shortest Time - to - Completion First)

- SJF에 preemption 특징 추가
- AKA, PSJF (Preemptive Shortest Job First)
- new job과 remaining job을 결정
- 가장 적게 남아있을 job을 스케쥴링



$$\text{Average turnaround time} = \frac{(120 - 0) + (20 - 10) + (30 - 10)}{3} = 50\ sec$$

- A를 처리 도중 B, C가 오면 A를 중단하고 B, C를 처리
- A 작업이 0초에 시작해서 120초에 끝났으므로 120 - 0

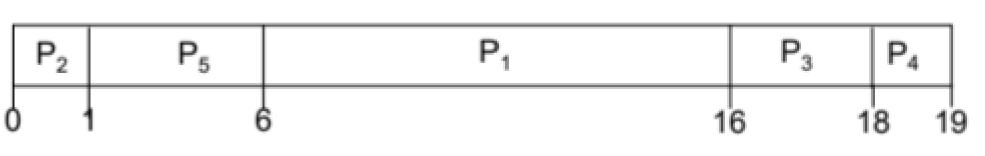
New scheduling metric : Response time

- job이 도착해서 scheduling 되는 데 걸리는 시간
- STCF는 response time가 좋지 않음
  - 늦게 온 긴 작업은 도착한 뒤에도 CPU를 받는 데 오래 걸림

Priority Scheduling

- 우선순위에 따라 스케쥴링
- 우선순위는 internally or extrenally하게 define
- priority가 같으면 FCFS 방식대로

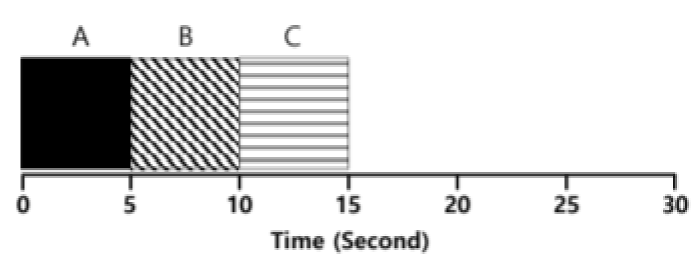
Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2



- average waiting time = 8.2
- SJF는 predicted next CPU burst가 priority인 priority scheduling임
- 단점
  - 계속해서 higher-priority process가 들어오면 low-priority는 영원히 실행되지 못 함 -> infinite blocking / starvation
  - starvation을 aging으로 해결
    - 기다리는 동안 priority를 조금씩 늘림

RR (Round Robin)

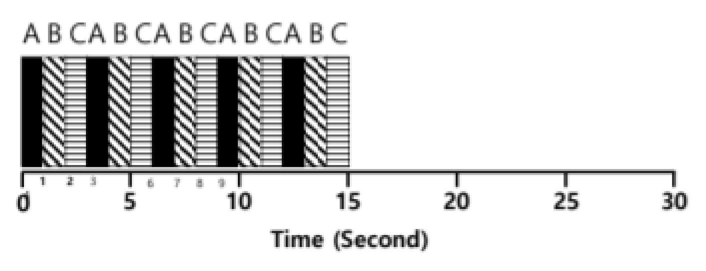
- 중세시대 회의 방식
- time slicing scheduling
- time slice (time quantum)
  - 보통 10~100ms
  - 매우 짧게 쪼갠 시간 (a.k.a scheduling quantum)
  - time slice는 timer-interrupt의 주기의 배수여야 함
- job을 time slice만큼 수행하고 run queue에 있는 다음 job으로 switch
  - 기존 job은 ready queue의 끝으로 이동
  - run queue는 circular queue
- job들이 끝날 때 까지 반복



SJF (Bad for Response Time)

$$T_{average\ response} = \frac{0 + 5 + 10}{3} = 5sec$$

- 위와 같은 상황에선 response time이 5초로 매우 느림



RR with a time-slice of 1sec (Good for Response Time)

$$T_{average\ response} = \frac{0 + 1 + 2}{3} = 1sec$$

- 하지만 잘게 쪼갠다면 매우 짧은 response time을 기대할 수 있음
- 짧은 response지만 complete가 늦게 되므로 bad turnaround time
- preemptive, no starvation, fair

the length of time slice

- 짧게 쪼갤 수록 resonse time은 좋아지지만 잦은 context switching이 전체 비용에 큰 비중을 차지하게 됨
- length는 시스템 디자이너에게 주어진 trade-off 과제

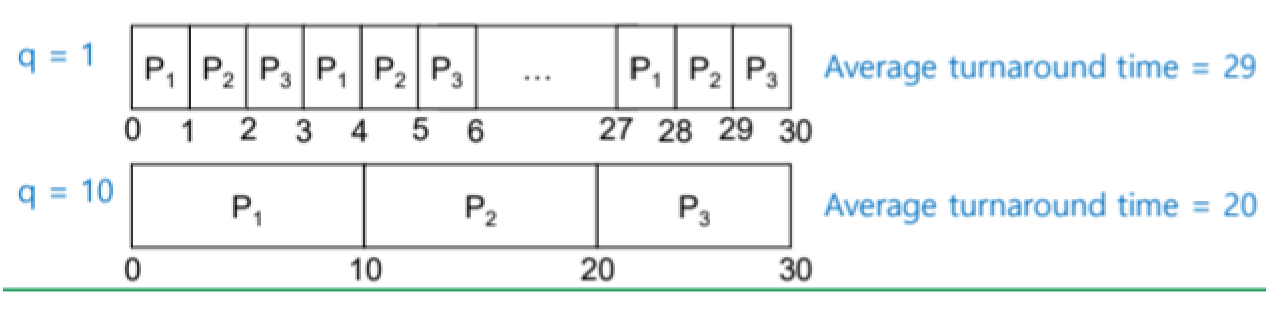
RR의 성능

- time quantime의 크기가 좌지우지
- q가 매우 크면 FCFS
- q가 매우 작으면 process sharing (속도가 1 / n가 되어버림)
- 보통 10ms ~ 100ms이며 context swtich는 10 usec
  - time quantum은 context switch보다 커야함

turaround time도 time quantum의 영향을 받음

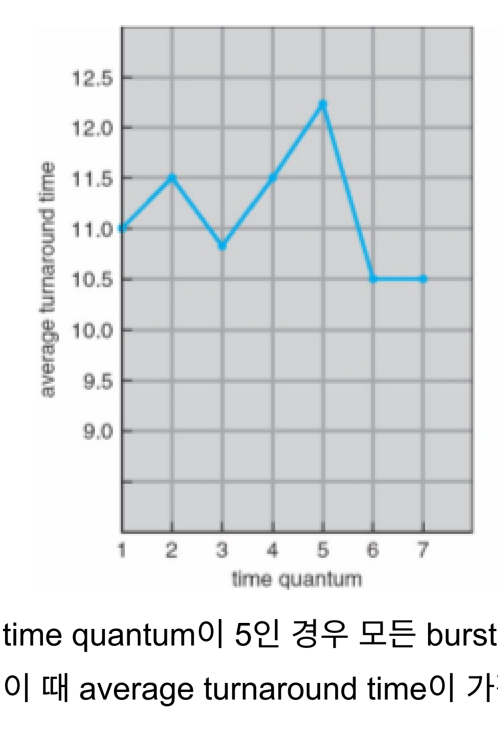
- 평균적으로 SJF보다 turnaround time이 높지만 response가 좋음
- 프로세스가 single time quantum으로 burst time을 끝내면 평균 turnaround time이 개선됨

Process	Burst Time
P_1	10
P_2	10
P_3	10



- rule of thumb이라는 게 있음
  - 모든 CPU 버스트들의 80퍼센트가 time quantum보다 짧아야 함

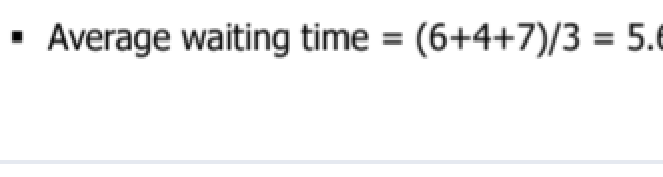
process	time
P_1	6
P_2	3
P_3	1
P_4	7



  - time quantum이 5인 경우 모든 burst들의 80퍼센트가 time quantum보다 짧음
  - 이 때 average turnaround time이 가장 좋은 결과를 보임

예제

Process	Burst Time	Time Quantum
P_1	24	4
P_2	3	
P_3	3	



- Average waiting time = (6+4+7)/3 = 5.66

Multilevel Queue Scheduling

- 다양한 큐들로 분리된 ready queue에 프로세스가 들어감
- memory size, process priority, process type 등 여러 할당 기준이 존재
- 큐 각자 자신만의 스케쥴링 알고리즘을 소유
  - 크게 foreground와 background로 나뉨
  - foreground - RR, background - FCFS

2가지 방식

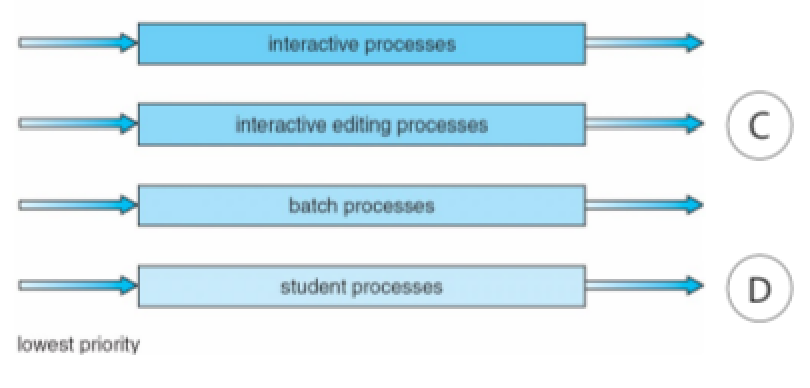
- fixed priority scheduling
  - foreground를 전부 처리하고 난 뒤 bacground 처리
  - starvation 우려
- time slice
  - 80 : 20으로 fore, back가 CPU time을 가짐

왜 사용하는가?

- turnaround time 최적화
- interactive한 job의 response time 최소화

MLQ (basic rules)

- MLQ에 여러 개의 서로 다른 큐가 있음
  - ex) system queue, interactive queue, batch queue, background queue
- 각 큐는 서로 다른 우선순위를 가짐
- 프로세스는 한 번 할당 시 다른 큐로 이동 불가
- 같은 큐 안에서는 RR 스케쥴링 동작



Solaris ML(F)Q 구현

- 60개의 큐가 존재
- 큐의 우선순위가 높을 수록 time-slice가 짧고 낮을 수록 길
- 우선순위가 매 초마다 높아짐

Multilevel Feedback Queue Scheduling

- 프로세스가 큐를 이동할 수 있음
- starvation을 막기 위한 aging
- 각 큐마다 allotment가 존재
  - allotment : 큐에 할당되는 CPU time

MLFQ는 다음 요소들을 정해야한다

- 큐의 수
  - Solaris는 60개 사용
  - 많을수록 더 섬세한 우선순위 조절 가능, but 구현 복잡
- 각 큐의 scheduling 알고리즘
- 언제 process를 upgrade/demote할지 결정하는 method
  - 프로세스를 어느 queue에 넣을지 결정하는 method

Basic MLFQ의 문제점

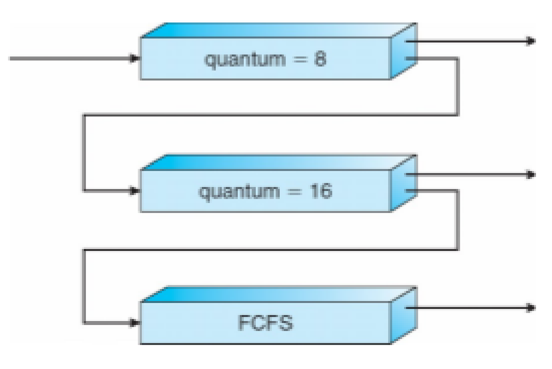
- Starvation
  - interactive job이 너무 많아 long-running job이 CPU time을 못 받는 경우
- Game the scheduler
  - scheduler의 우선순위 정책을 악용
  - time slice를 99퍼센트 쓰고 남은 1퍼센트를 IO operation issue에 사용
  - 무의미한 IO 작업이지만 scheduler는 프로세스의 우선순위를 유지 or upgrade
- 프로그램이 CPU 위주 작업이다가 IO 위주 작업으로 바뀌어야 할 수 있음
  - CPU bound process -> IO bound process

MLFQ rules (refined set)

- 프로세스 A와 B가 있다고 가정하자
- A의 우선순위를 PA, B의 우선순위를 PB라 하자
- rule 1 : PA > PB 시 A 실행
- rule 2 : PA == PB 시 RR로 처리
- rule 3 : job이 시스템으로 들어오면 가장 높은 우선순위에 배치
- rule 4 : job이 allotment를 다 쓰면 강등
- rule 5 : 특정 주기마다 모든 작업들을 topmost 큐로 이동

예시

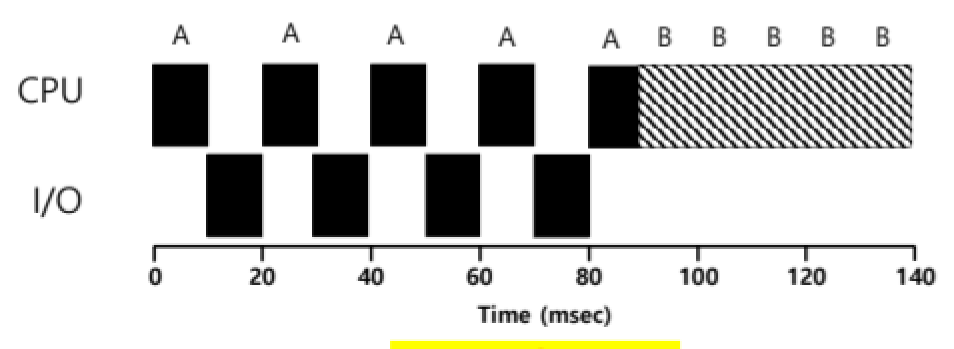
- 큐가 3개인 경우



- 처음엔 스케쥴러는 Q0의 프로세스를 처리
- Q0이 비면 Q1 처리
- Q2는 Q0과 Q1이 비어야 처리
- 만약 Q1에 프로세스가 들어오면 Q2는 중단

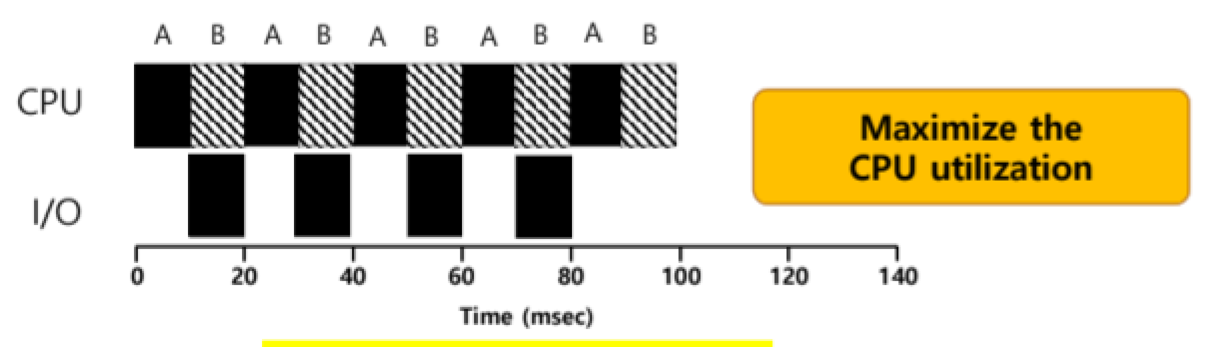
Incorporation I/O

- turnaround time이나 response time과는 별개로 고려해봐야할 상황이 있음
- 모든 프로그램은 IO를 사용
- IO를 사용하는 동안 CPU를 사용하지 않는다는 문제



Poor Use of Resources

- A가 IO를 여러번 사용
- CPU를 사용하지 않는 동안 B를 처리하는 것이 합리적



Overlap Allows Better Use of Resources

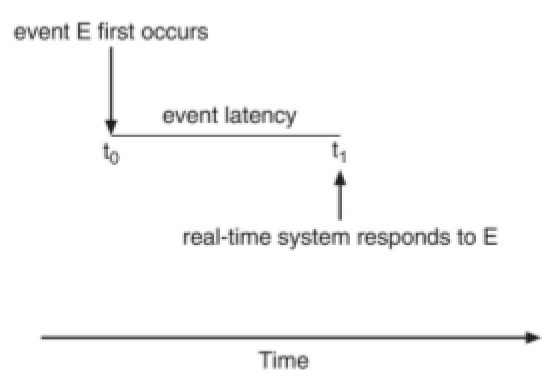
- job이 IO request를 보내면 job은 blocked 되고 IO의 completion을 기다리게 됨
- 이때 scheduler는 다른 job을 CPU에 scheduling
- IO가 completion되면 interrupt 발생
- OS는 blocked된 job으로 돌아가 ready state로 바꿈

Real-Time CPU Scheduling

- 매순간순간 처리해야하는 프로세스들이 갱신되는 상황에서 scheduling
- 실시간으로 CPU를 스케쥴링 한다는 건 어려움
- soft real-time systems : 중요한 real-time 작업을 최우선순위로 두지만 꼭 지켜지지 않아도 됨
  - ex) 스트리밍
- hard real-time systems : 반드시 deadline을 지켜야함
  - ex) 에어백, 심장박동감지 의료기기

latency

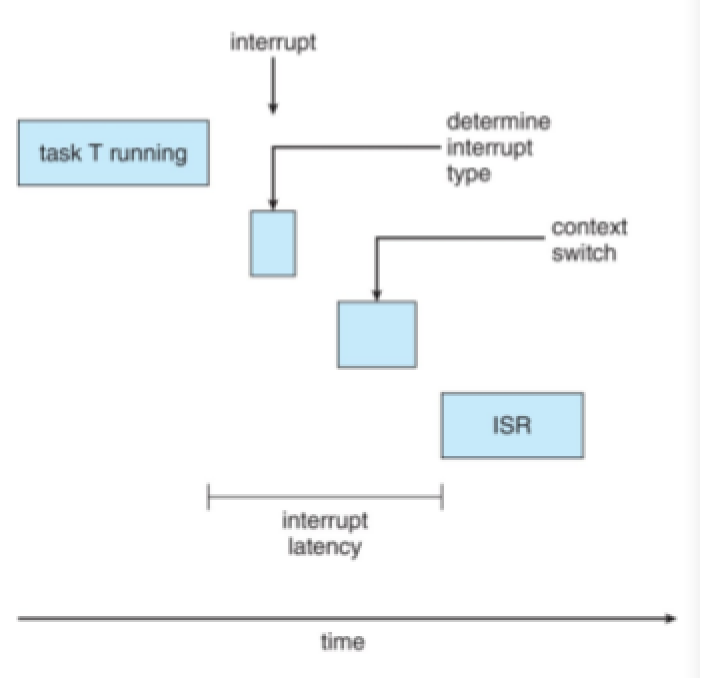
- event latency : 이벤트가 발생하고 난 후 처리되기까지 걸린 시간



- 성능에 영향을 미치는 두 유형의 latency가 있음
  - interrupy latency : 인터럽트 도착부터 인터럽트 서비스 루틴 실행까지 걸린 시간
  - dispatch latency : 현재 프로세스를 중단하고 다른 프로세스를 시작하는 데 걸린 시간

Interrupt latency

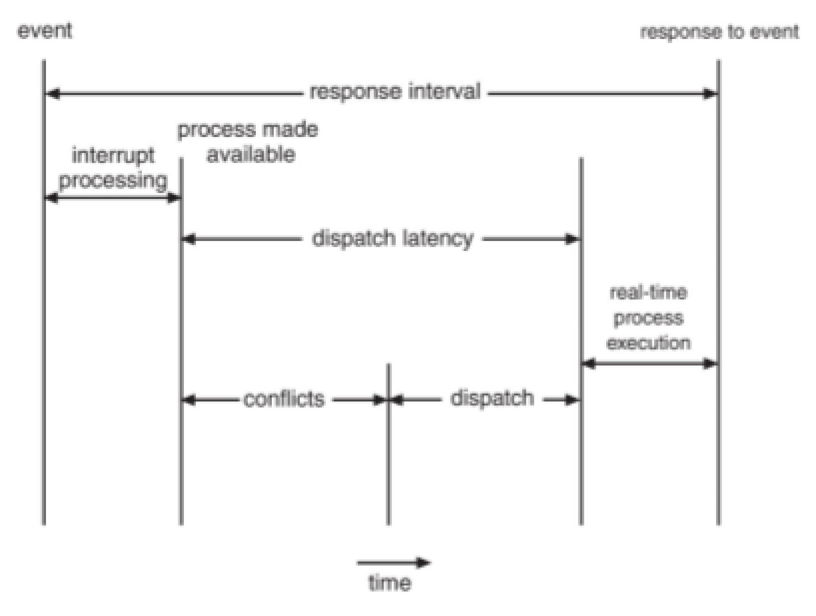
- 인터럽트가 발생한 순간부터, 그 인터럽트를 처리 시작하기까지 걸리는 시간
- real-time에서는 짧을수록 좋고 예측 가능해야 함
- components
  - instruction이 pipeline을 처리하는 동안 인터럽트를 check하는 데 걸린 시간
    - CPU는 명령어를 처리하는 도중에도 인터럽트를 감지해야 함
    - 특정 시점엔 인터럽트를 일부러 무시하거나 지연시킴
    - 그 시점이 끝날 때 까지 기다리는 시간
  - interrupt를 check 하고 난 후 어느 interrupt type으로 할지 결정하는 데 걸린 시간
  - ISR (interrupt service routine)을 실행하기 위해 context switch



- other components
  - kernel data strucre가 업데이트 되는 동안 interrupt가 비활성화되는 시간
    - OS Kernel이 중요한 데이터를 갱신할 때 인터럽트를 일시적으로 꺼놓음
    - 데이터 꼬임 방지
    - ex) 리스트 삽입 중간에 인터럽트 발생 시 무결성 깨짐
- real-time OS에서 인터럽트가 disabling되는 건 매우 짧은 시간이어야 함

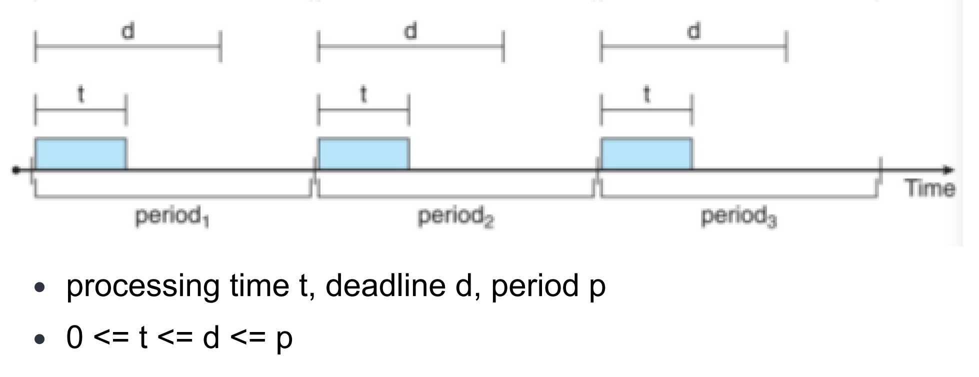
Dispatch Latency

- periodic real-time task에서 중요한 개념
  - 스트리밍 : 주기적으로 버퍼에 데이터를 채워야함
  - 공장 자동화 시스템 : 매 순간마다 센서를 체크해야함
- 실행 준비가 되었더라도 CPU에서 실행되기까지 시간이 지연되면 안 됨
- high-priority task가 실행 준비된 순간부터 CPU에서 실행되기까지 걸린 시간
- conflict phase
  - 커널모드에서 실행 중인 어느 low-prioirty 프로세스가 있을 경우
    - high-priority task를 위해 preemption
    - 커널모드라 쉽게 끊기 힘든 점 때문에 선점 지연
  - low-priority task가 high-priority task가 필요한 resource를 갖고 있을 경우
    - low-prioirty task가 resource를 release하는 데 걸림
- dispatch phase
  - conflict phase 해결 후 available CPU를 high-priority 프로세스에 scheduling
  - 당연히 scheduling에도 시간이 필요



Priority-based Scheduling

- real-time scheduling을 위해 preemptive, priority-based scheduling이 필요
- 하지만 이 방법은 soft real-time만 guarantee
- hard real-time은 deadline을 만족시킬 ability를 제공해야 함
- 각 프로세스에 일정한 cpu time을 제공

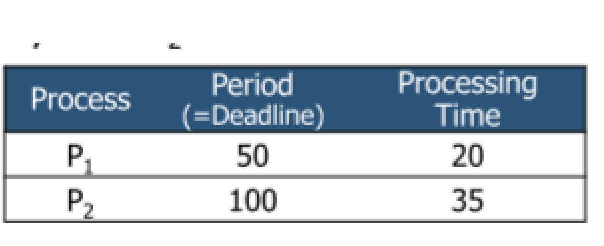


  - processing time t, deadline d, period p
  - 0 <= t <= d <= p
- admision-control algorithm
  - 지정된 시간 내에 프로세스가 완료될 수 있을 거 같으면 admission
  - 만약 데드라인을 넘길 거 같으면 deny

Rate-Monotonic Scheduling

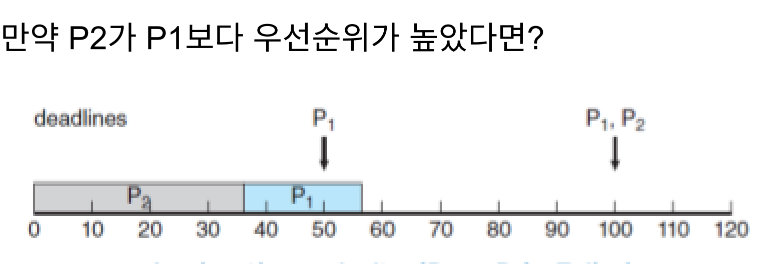
- 프로세스의 우선순위를 period의 역수를 기반으로 할당
  - shorter period, lower priority
  - longer period, higher priority
- frequently하게 CPU를 차지하는 task에 높은 priority

예시



Process	Period (=Deadline)	Processing Time
P_1	50	20
P_2	100	35

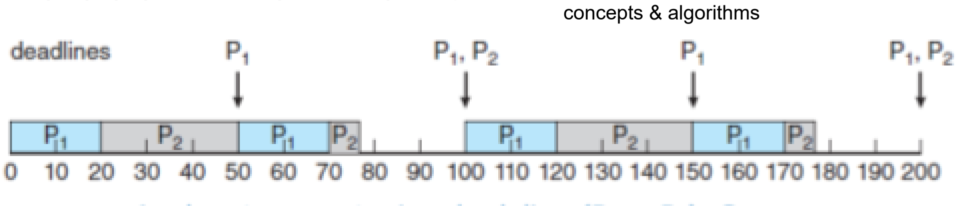
- P1의 CPU utilization은 20/50으로 0.4, P2는 35/100으로 0.35
- 합 0.75이므로 schedulable
- period가 곧 deadline
- P1은 50s 마다 들어오고 P2는 100s 마다 들어옴
- P1이 50s 마다 들어오므로 다음 P1이 들어오기 전 이전 P1이 끝나야 함
- 만약 P2가 P1보다 우선순위가 높았다면?



부산대학교 Apply other priority (P_2 > P_1): Failed

- P1이 나중에 실행되어 deadline을 넘겨버림...

- P1이 처리되는 도중 다음 P1이 와버림



Apply rate-monotonic scheduling ($P_1 > P_2$): Success

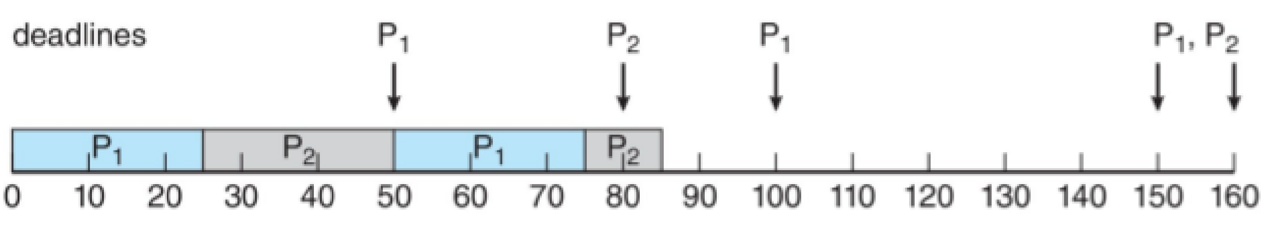
- 우선순위로 인해 P2가 처리되는 도중 P1의 도착으로 P1부터 처리
- 그 후 P2를 처리해도 deadline을 안 넘겨서 ok

missed deadlines with rate-monotnoic scheduling

- 만약 프로세스들의 CPU utilization이 너무 크면?

Process	Period (=Deadline)	Processing Time
P_1	50	25
P_2	80	35

- 두 프로세스의 cpu utilization 합은 0.94 -> schedulable?



- P2가 80s에서 deadline을 miss
- CPU utilization은 얼마여야 하는가?

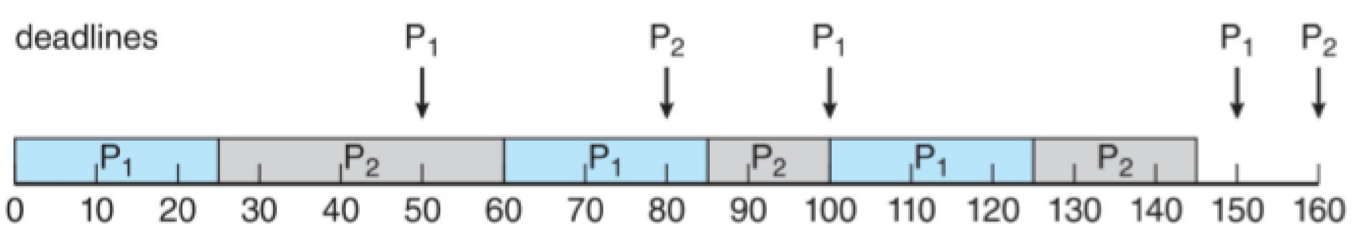
  When scheduling N processes, the worst-case CPU utilization is $N(2^{1/N} - 1)$

  - A reasonable CPU utilization is 100% for one, 83% for two, and about 69% for a very large number of processes
- period 보다 더 좋은 기준은??

Earliest Deadline First Scheduling (EDF)

- deadline이 가장 급한 프로세스에 higher priority 제공

Process	Period (=Deadline)	Processing Time
P_1	50	25
P_2	80	35



- priority가 갱신되는 시점 : pdf에는 명시되어있지 않은데 새 프로세스가 들어올 때 인 듯
- P1이 들어올 때 : P1의 deadline은 100이지만 P2는 80이므로 P2부터
- P2가 들어올 때 : P2의 deadline은 160이지만 P1은 100이므로 P1부터
- P1이 들어올 때 : P2의 deadline은 160이지만 P1은 150이므로 P1부터

Proportional Share Scheduling

- 직접 프로세스의 CPU time에 할당 비율 조정
- 별로 중요한 내용은 아닌 듯...?