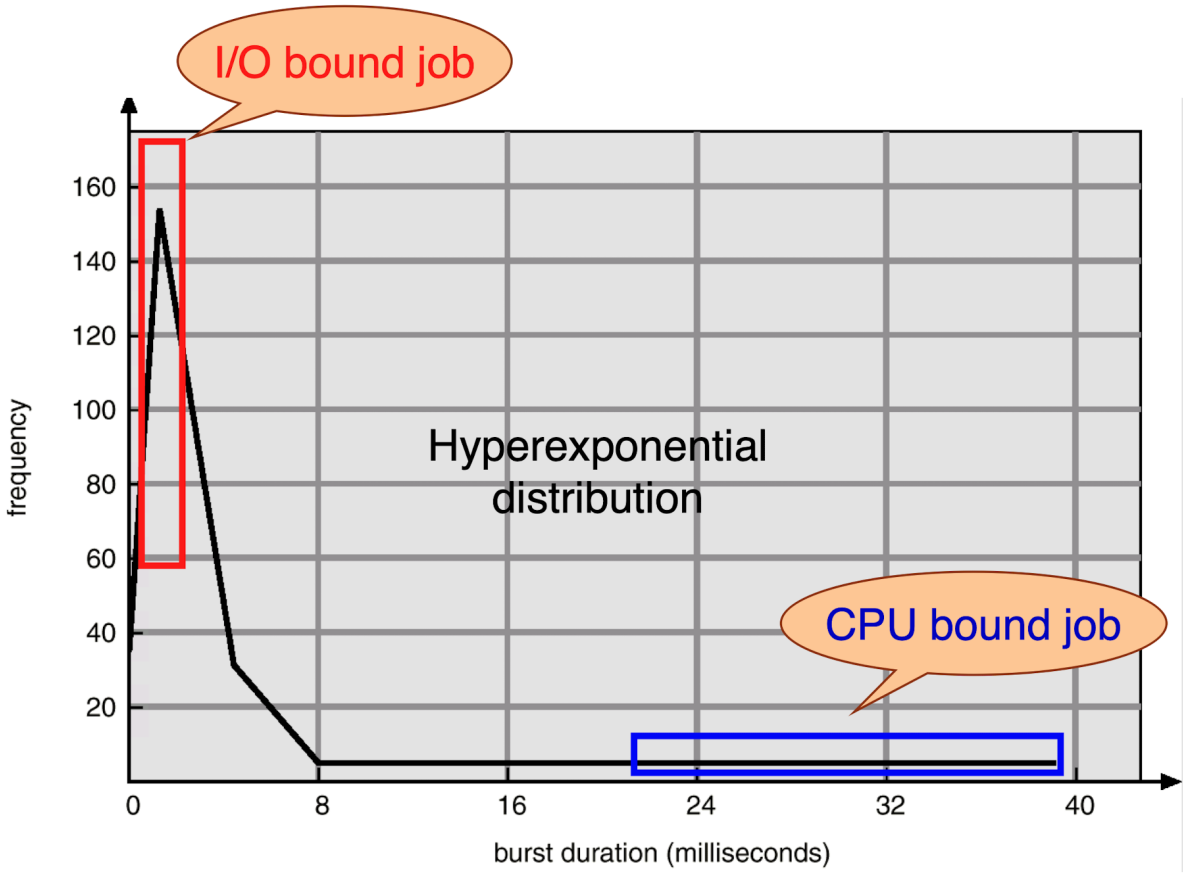
# 4. CPU Scheduling

### CPU burst vs IO Burst

CPU burst: 정말 말 그대로 CPU가 연산처리하는 거

IO burst: IO 장치가 일하는거




IO bound같이 사용자의 편의가 중요한 경우에는 최우선적으로 처리하는게 맞아

⇒ 카톡보내는데 10초 20초씩 보내면 속 터지니까

그래서 최대한 점유시간이 짧은 애들을 앞에서 많이 잡으면서 처리하고 한번 잡을 때 오래걸리는 애들을 나중으로 미뤄

막말로 컴파일 타임 1시간 짜리가 1시간 10초 된다고해서 큰 영향 없잖아

### State Transition Diagram

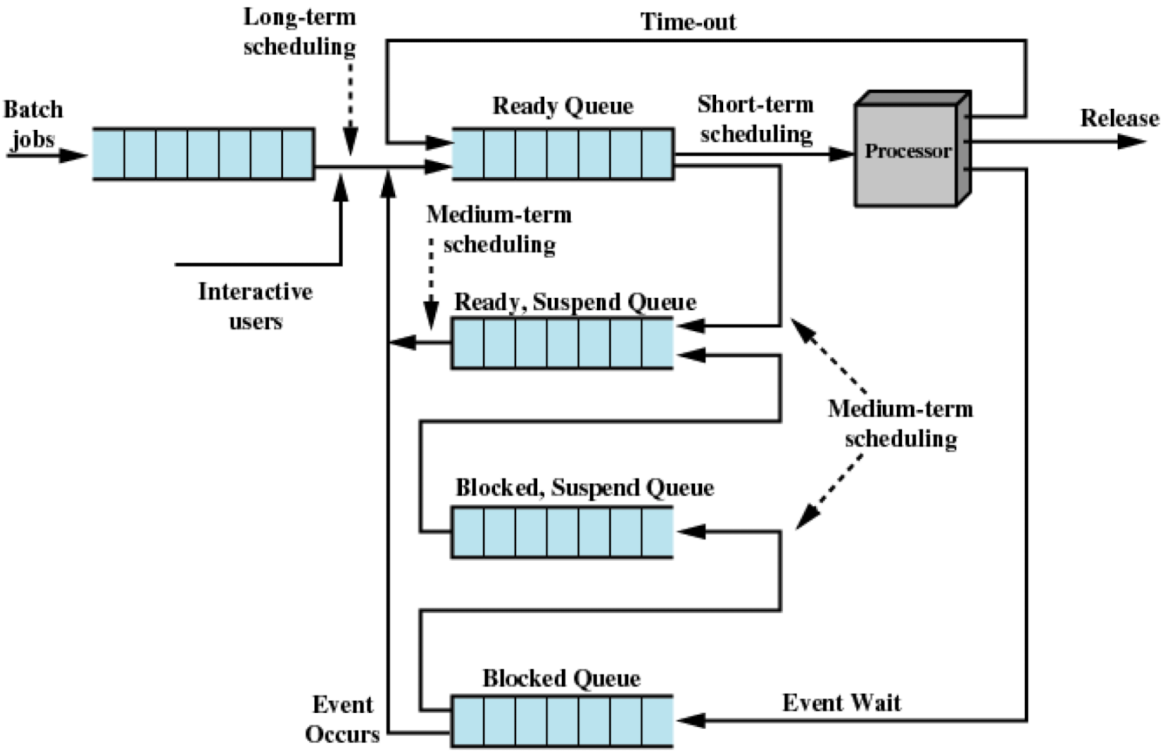



**Figure 9.3 Queuing Diagram for Scheduling**

일단 이게 저번에 본 state transition을 좀 더 구체화해서 표현한 그림인데

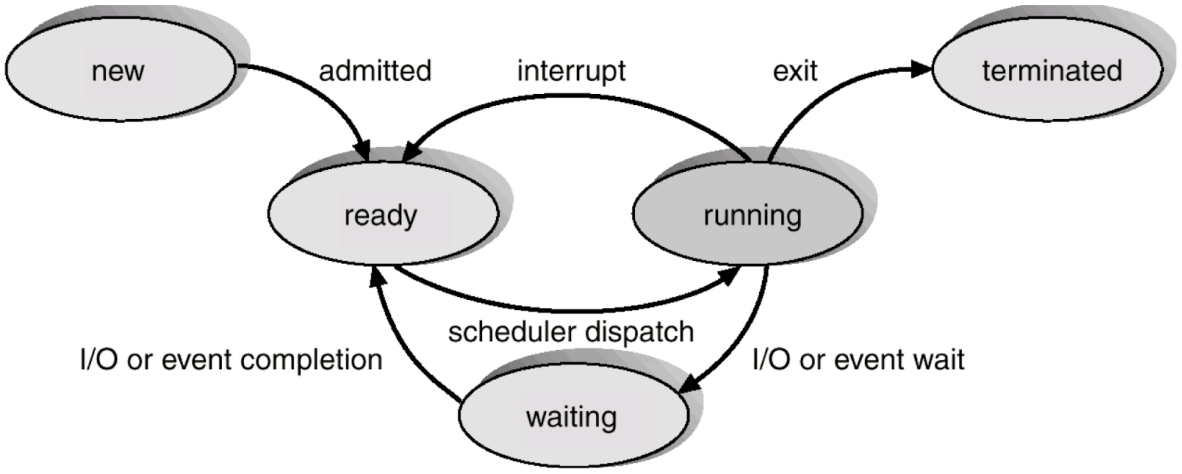
(그 new, ready, running, waiting, terminate 그림)

Suspended Process를 Suspend Queue에 넣어서 별도로 관리하는 걸 볼 수 있어

이 때 Waiting상태의 process를 Medium-term scheduler가 동일하게 관리하는 거 확인 가능

### CPU Scheduling 상황 4가지

1. running → waiting (e.g IO request)
2. running → ready (e.g time runout)
3. waiting → ready (e.g IO finished interrupt)
4. terminate




설마 이거 기억 안 나진 않겠지

그래서 OS가 강제로 탈취해서 CPU에 할당해주는게 2, 3

반대로 자연스럽게 만료돼서 CPU가 다른 프로세스를 잡게 되는거

그럼 이런 스케줄링을 해주는 주체가 누구다? ⇒ OS

그래서 강제성이 있는거(preemptive) ⇒ 2,3

강제성 없는거(nonpreemptive) ⇒ 1, 4

## Dispatcher

설마 Dispatch Servlet이 여기서 나온건가?

아무튼 Dispatcher가 process에 CPU를 할당하는 겁니다 그러니까 Dispatcher도 OS의 기능이겠죠?

참고로 Dispatcher는 kernel code입니다.

그래서 얘가 뭘하냐?

1. switching context
2. switching to user mode (kernel mode로 왔으니 다시 user mode로 돌려줘야지)
3. jumping to the proper location in the user program to restart that program

   ⇒ 이거는 interrupt 부분 좀 자세히 보면 좋은데, 이게 프로세스는 다른 프로세스로 변경될 때 스택 영역에 현재 쓰고 있는 내용(레지스터, PC 등)을 저장하고 옮기는데 이걸 그대로 불러와야돼 여기서 말하는 proper location이 PC값이기 때문에 (정확히는 PC의 이전값이겠지) 그 주소로 이동

Dispatch latency: 이게 context switch overhead

## Scheduling Criteria

1. CPU utilization: 최대로
2. Throughput: 최대로
3. Turnaround time: 최소로
   a. ready queue에서 대기한 시간 + CPU에서 실행하는 시간 + IO 시간
4. Waiting time: 최소로
   a. ready queue에서 대기한 시간

5. Response time: 최소로

### Scheduling Algorithms

1. First Come First Served

   장점: 제일 공평해

   단점: convoy effect, 앞에 프로세스가 작업시간이 너무 길어지면 뒤에 프로세스는 너무 오래 기다려야돼
2. Shortest Job First(nonpreemptive, 그리디 알고리즘)

   장점: 전체 프로세스의 대기 시간이 줄어들어

   단점: starvation problem, 새로 들어오는 작업이 계속 짧은 애만 들어오면 특정 프로세스가 실행되지 못해

   또한 프로세스에 사용되는 CPU time이 어느정도인지 계산할 수 있어야 하는데, 현실적으로 불가능해
3. Shortest Remain Time First(preemptive)

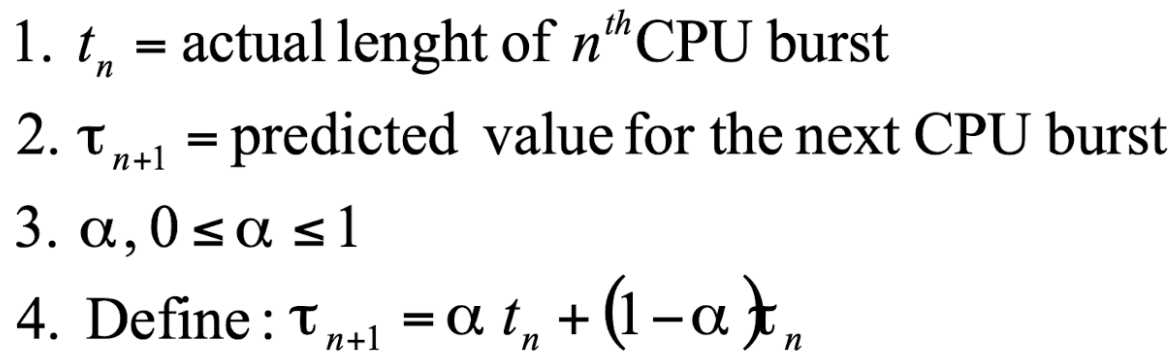
   장점: 마찬가지로 전체 프로세스의 대기 시간이 줄어들어

   단점: Shortest Job First와 동일함
4. Priority Scheduling
5. Round Robin(preemptive)
6. Multilevel Queue
7. Multilevel Feedback Queue

### Determining Length of Next CPU Burst

위에서 SJF, SRTF 할 때 CPU의 시간을 알 수 없는게 문제였는데 이거를 진짜 알아내기보다는 유사하게 예측을 하는거지

그래서 이전 CPU burst의 크기를 통해서 다음을 예측하는거




이거 쓸라 했는데 그냥 이거 자체를 보는게 더 나을듯

alpha가 0이면 그냥 이전 예측값을 그대로 가져오는거

alpha가 1이면 이전에 실제로 사용되었던 값을 그대로 가져오는거

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots$$
$$+ (1 - \alpha)^j \alpha t_{n-j} + \dots$$
$$+ (1 - \alpha)^{n+1} \tau_0$$

공식을 지속적으로 사용하게 되면 이런식으로 나옴 (그냥 대입하면 됨)

단순히 직전값에만 의존한다면 변화가 너무 커 ⇒ 그래서 가중치를 둬서 변화는 반영하면서도 이전의 추이를 같이 반영

### Priority Scheduling

일단 우선순위가 들어온 순간 무조건 starvation 문제는 있다고 생각하면 됨

⇒ 그래서 우선순위에 따른 스케줄링은 Aging을 반드시 도입해야 돼

SJF도 일종의 priority scheduling인거지 (next CPU burst time이 우선순위의 기준이고)

### Round Robin

이건 일단 priority scheduling이 아니야

이 알고리즘의 가장 큰 장점은 모든 프로세스의 상한시간을 알 수 있어

각각의 프로세스는 time quantum을 할당 받아 ⇒ 이 시간만큼 쓰고 다른 프로세스로 CPU를 넘겨야돼

time quantum이 크면 ⇒ FIFO랑 다를게 없어

time quantum이 작으면? ⇒ 프로세스 교체가 활발하게 이뤄지니까 context switch 비용이 너무 커

⇒ 그래서 적절한 time quantum을 선택하는게 중요해

### Multilevel Queue and Multilevel Feedback Queue

일단 둘다 여러개의 ready queue를 둬

그래서 각각의 queue 마다 알고리즘을 다르게 선택해

예를 들어 queue를 3개 쓴다고 가정했을 때

뭐 1, 2번 큐들은 round robin 방식으로 사용하고 3번 큐는 FCFS로 하는 식으로

이렇게 하면 round robin에서 필요한 장점도 채택할 수 있고 좀 더 우선으로 해야하는 작업도 어느정도 먼저 할 수 있어

그럼 둘이 차이가 뭐냐?

MLQ는 하나의 큐에 진입하는 순간 그 큐에 영구 박제

MLFQ는 모든 프로세스가 큐간 이동을 할 수 있어(ex- L1에서 L2로 이동가능)

그러다보니 MLFQ가 좀 더 유연하게 대응할 수 있어 MLQ의 경우는 해당 큐가 priority scheduling인 경우 starvation problem을 해결해야 하니까