리눅스 커널(운영체제) 강의노트[5]

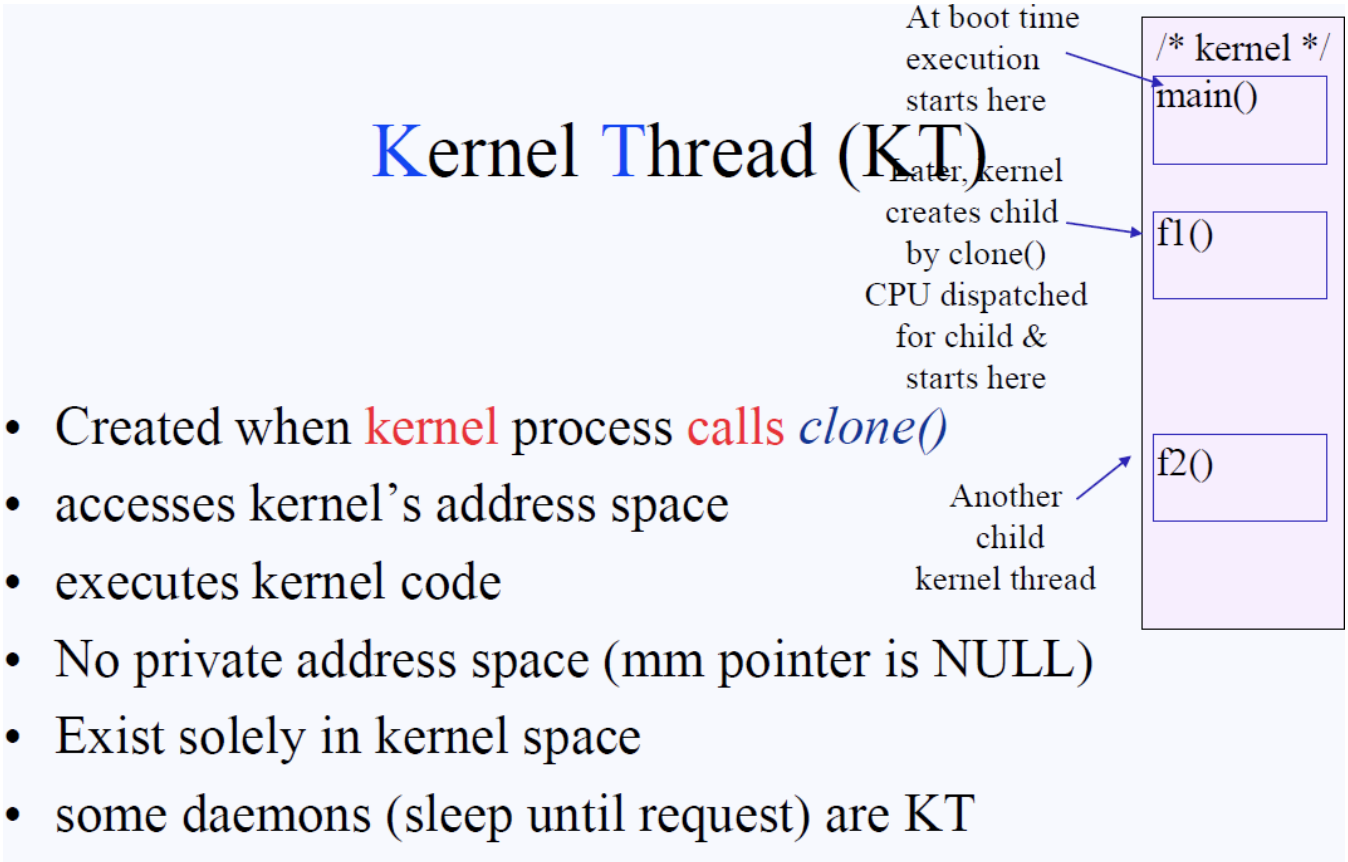
aeharvlee
Nov 19, 2018 · 15 min read

이번 5번째 강의에서는 프로세스간 CPU 점유권의 이동이 어떤 매커니즘으로 이루어지는지를 다루게 된다. CPU를 할당해준다는 것은 단순히 프로세스의 우선순위 말고도 고려해야할 것들이 많다. 리눅스 운영체제는 과연 이러한 숙제를 어떻게 풀고 있는지 지금부터 살펴보려 한다.

1. Kernel Thread(커널 스레드)

먼저 강의노트 4에서 다뤘던 내용들을 잠시 떠올려보자. **스레드(Thread)**가 있고 **프로세스(Process)**가 있었다. **프로세스는 부모의 것(Task basic info + files, fs, tty, mm, signals)**을 전부 그대로 복사한 것(heavy-weight creation)이고,**반대로 최소한으로 복사(light-weight creation)**한 것이 스레드이다.

또한 커널은 **메모리 상주 프로그램(memory resident program)**이다. `main()` 함수가 있는 **평범한 프로그램의 특성 +부팅할 때 부터 메모리에 올라와서 컴퓨터의 전원을 완전히 차단할 때까지 메모리에 상주**한다는 특성을 함께 가지고 있다. 여기까지 떠올렸다면 이제 아래 그림을 보면서 오늘 다룰 주제 중 하나인 **커널 스레드**를 살펴보자.

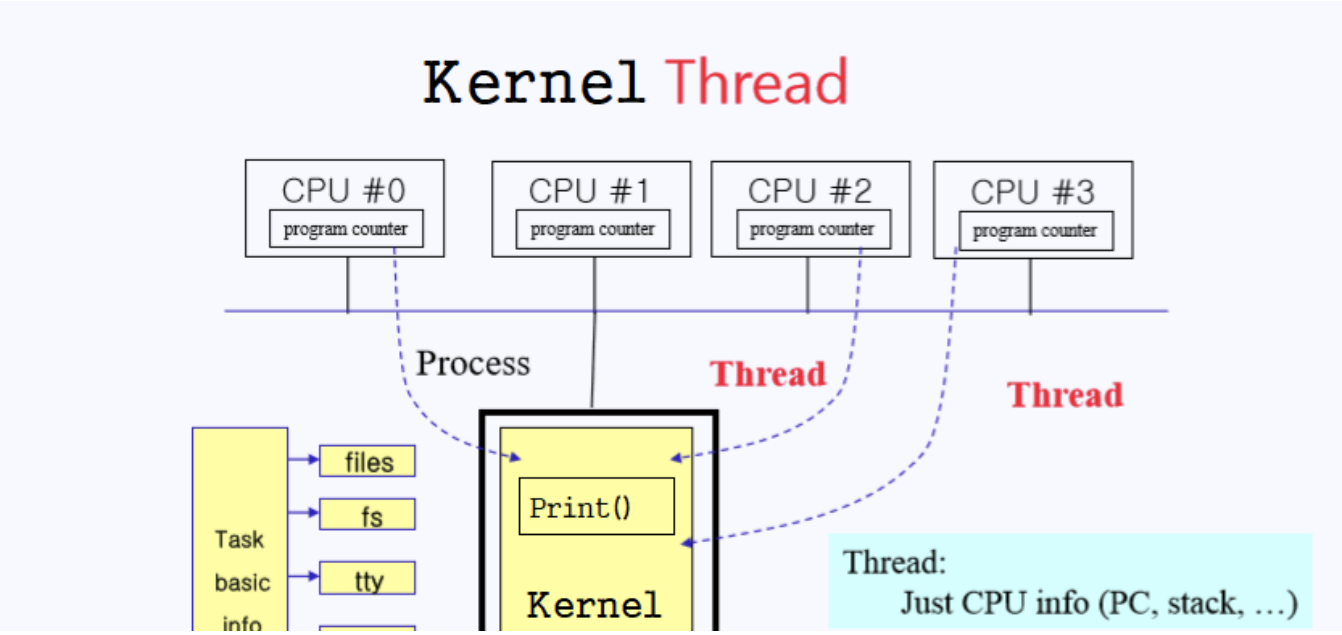


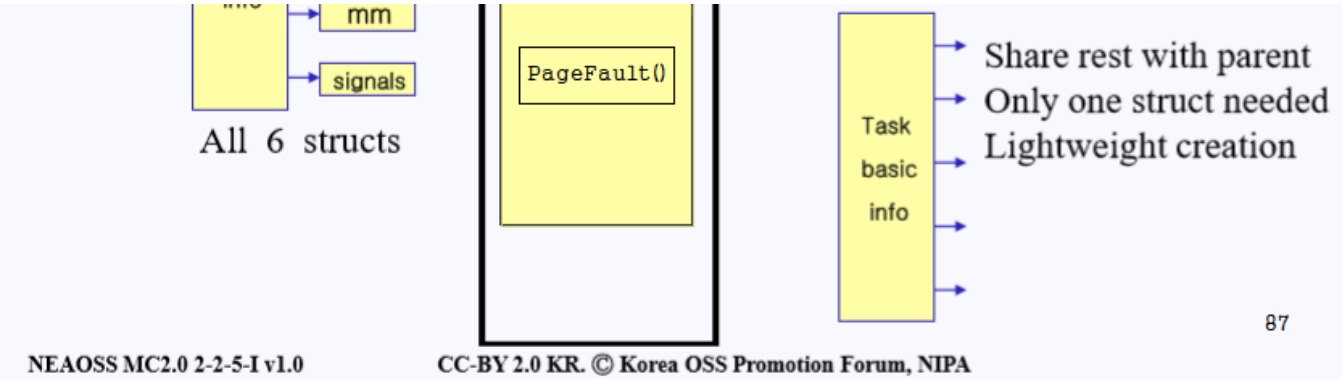
위 그림의 우측 보라색으로 칠해진 부분은 커널영역을 의미한다. 컴퓨터가 **맨 처음 부팅(booting)하면 분명 커널의** `main()` **부터** 실행할 것이다. 그 후 커널 프로세스가 동작하던 중 커널 프로세스 내에서 **시스템 콜** `clone()` **을 호출**하게 되면 자식 프로세스가 생기는데, 이때 **부모 프로세스가 가르키는 PC(Program Counter) 와 자식 프로세스가 가리키는 PC**는 각각 다른 곳을 가리키고 있을 수 있다.

그래서 만약 CPU 코어가 3개가 있다고 한다면, **각 CPU코어의 PC는** `main()` **과** `f1()` **과** `f2()` **를 가리키고 있을 수 있다**(CPU dispatched for child & starts here). 그렇게 만들어진 **자식 프로세스를 커널 스레드**라 한다. 이는**자식 프로세스들이 커널 코드를 실행(execute)하고 있기 때문**이다. 대부분의 경우 이런 함수들은 **서버(server) 혹은 데몬(daemon)**이다.

서버와 데몬의 알고리즘은 **기본적으로 무한 루프(endless loop)**라고 우린 배웠다. 또한 서버와 데몬은 **대부분의 시간을 자면서(sleep)** 보낸다. 그러다 **요청(request)이 오면 깨어나 그 작업을 처리해주고 또 잔다.** 네트워크와 관련된 작업을 처리하는 데몬이라면 네트워크 서버라고 부를 수 있고, 만약 프린트 요청을 기다리고 있다면 프린트 서버라고 할 수 있다.

결국 커널 스레드는 **커널 프로세스가** `clone()` **을 호출해서 light weight overhead로 자식을 만들어준 것**이다. 또한 커널 메모리 영역과 코드를 똑같이 접근하고, 커널 코드를 실행한다. 당연하게도 커널 스레드는 커널 영역에만 존재한다. 많은 데몬(웹서버, 프린트 등)들이 커널 스레드이다.



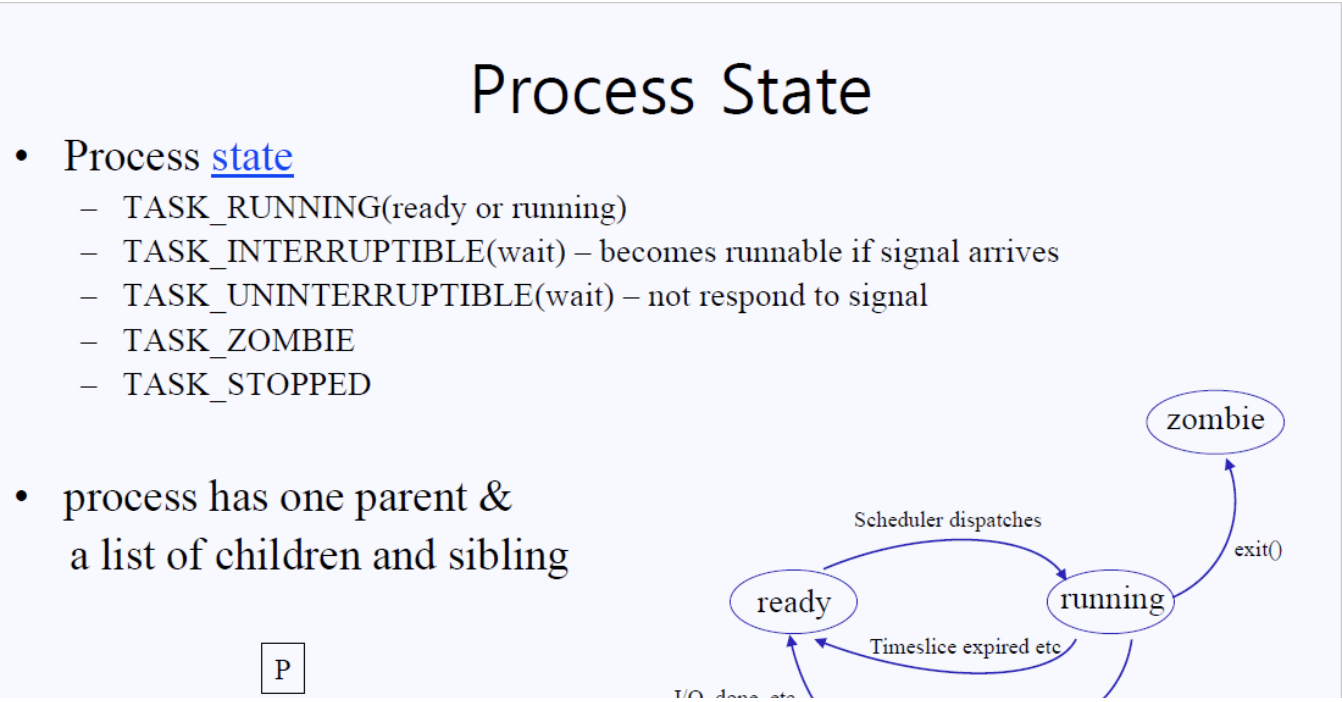


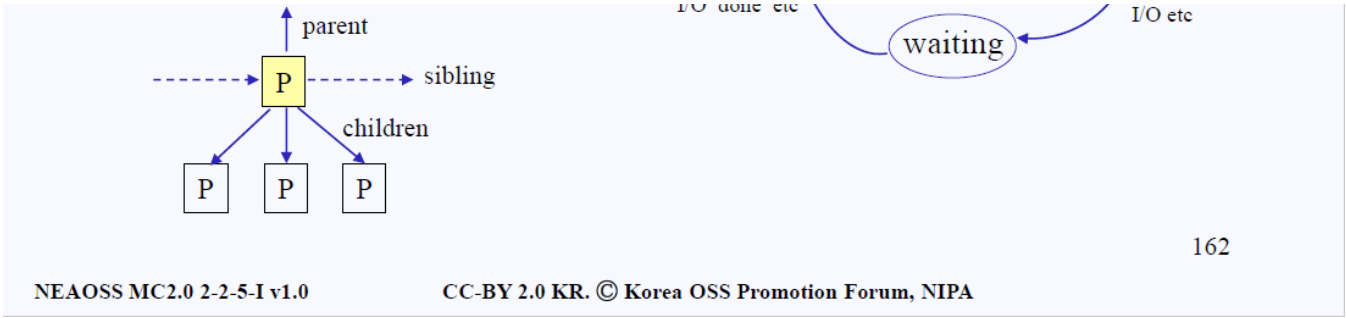
위 그림을 보면 CPU가 여러개 있고 **CPU 마다 PC(Program Counter)**를 가지고 있다. 지금은 PC가 커널을 가리키고 있다. 여기서 커널이 `clone()` 을 통하여 스레드 2개를 만들어줬다고 가정해보자. **CPU #2**에 할당된 것은 프린트 데몬(서버)이고, **#3**에 할당된 것은 **PageFault** 데몬이다. 앞 장에서 다뤘듯 스레드는 위 그림의 우측 노란박스에 해당하는 **Task basic info** 파트만 복사를 하고 나머진 부모 프로세스와 공유를 한다.

Task basic info 안에는 **state vector save area**가 존재하기 때문에각 스레드마다 별도의 **Program Counter**와 **Stack Pointer**를 갖고 있을 수 있는 것이다. 각 스레드가 각자의 **Stack**을 갖고 있기 때문에 개별적으로 커널 내의 다른 함수들을 호출하면서 실행될 수가 있다.

2. Process State

프로세스의 상태에는 **ready, running, waiting**이 존재한다. **running**은 프로세스 입장에서는 최상의 상태이며, running 중 Disk I/O를 요구하는 사건이 발생하면 CPU가 해당 프로세스의 상태를 **waiting**으로 바꾼다. waiting의 경우 시그널의 상황에 따라 2가지의 반응이 있을 수 있는데 이 부분은 중요한 내용은 아니므로 넘어가도록 한다.





I/O가 끝나고 상태는 **wait**에서 **ready**로 넘어가게 된다(CPU는 항상 바쁘다). **I/O**가 끝나고 **ready list**에 참여해서 기다리다 보면 자신의 차례가 올 것이다. 차례가 오는 것을 **Scheduler dispatches**라고 표현한다. dispatch과정을 상세하게 풀어보자면, `context_switch()` 의 동작으로 설명할 수 있다.

첫번째로 현재 사용중이던 프로세스의 `state vector`를 저장한다. 그리고나서 **run**하고 싶은 프로세스의 **state vector**를 **CPU**에 로드한 후, **Program Counter**가 가리키는 곳으로 가는 것이 **dispatch**의 과정이다.

CPU가 주어지면 어느 정도의 시간**(time slice)**만큼만 동작하고, 시간이 끝나면 다시 **ready list**로 가는 구조인데, 일단 프로세스가 성공적으로 `exit()` 했다고 가정해보자. 프로세스가 `exit()` 을 하게 되면 **zombie** 상태가 된다.

좀비 상태라는 것은 `a.out` 도 날라가고 `file` 도 전부 `closed` 되고 메인 메모리도 다 뺐기고, `PCB` 만 남은 상태를 의미한다. 왜 PCB가 남아있는 것일까?

만약 부모 프로세스(parent)가 지금까지 기다리고 있었으면, parent가 깨어나 CPU를 쥐고 실행될 것이다. **이 때 parent는 자신이 잘 동안child가 뭘 했는지, 디스크와 CPU를 얼마나 썼는지, 제대로 끝났는지 등에 대한 내용을 child의 PCB에서 확인한다.**

위와 같은 중요한 정보들이 `PCB` 에 있기 때문에 `PCB` 는 남겨둬야 한다. **따라서 자식 프로세스의 PCB는 부모 프로세스가 말소시키는 것이 맞다. parent가 말소시킬 때까지는 자식은 zombie상태인 것이다. 따라서 최상위 부모 프로세스는 자식 프로세스들이 사용한 모든 자원을 전부 파악할 수 있어야 한다.**

3. Kernel Scheduling (커널 스케쥴링)

- priority++ ← CPU preempted for I/O? (I/O bound)
- priority -- ← CPU preempted while computing? (CPU-bound)

- Base priority (initial value)
  - Standard priority
  - Real-time priority

- Who runs next? ➔ Runnable (ready) process with

AND { remaining timeslice
AND { highest priority

166

리눅스에서는 어떤 프로세스가 다음에 실행될 프로세스일까? 물론 **priority(우선순위)**가 가장 큰 프로세스가 실행될 것이다. 하지만 **time slice**를 가지고 있는지도 반드시 확인해야 한다.

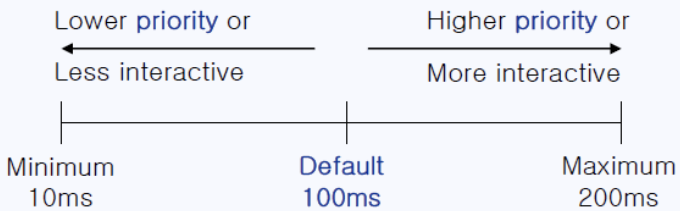
타임 슬라이스(time slice)에 대한 설명은 아래 그림을 보면서 살펴보도록 한다.

Timeslice

- Preempted? ➔ preempted task can use remaining timeslice later
  Task is runnable(ready) iff timeslice remain

> eg Task is given 100 ms. timeslice.
> After 20 m., CPU preempted.
> Use 80 ms. later

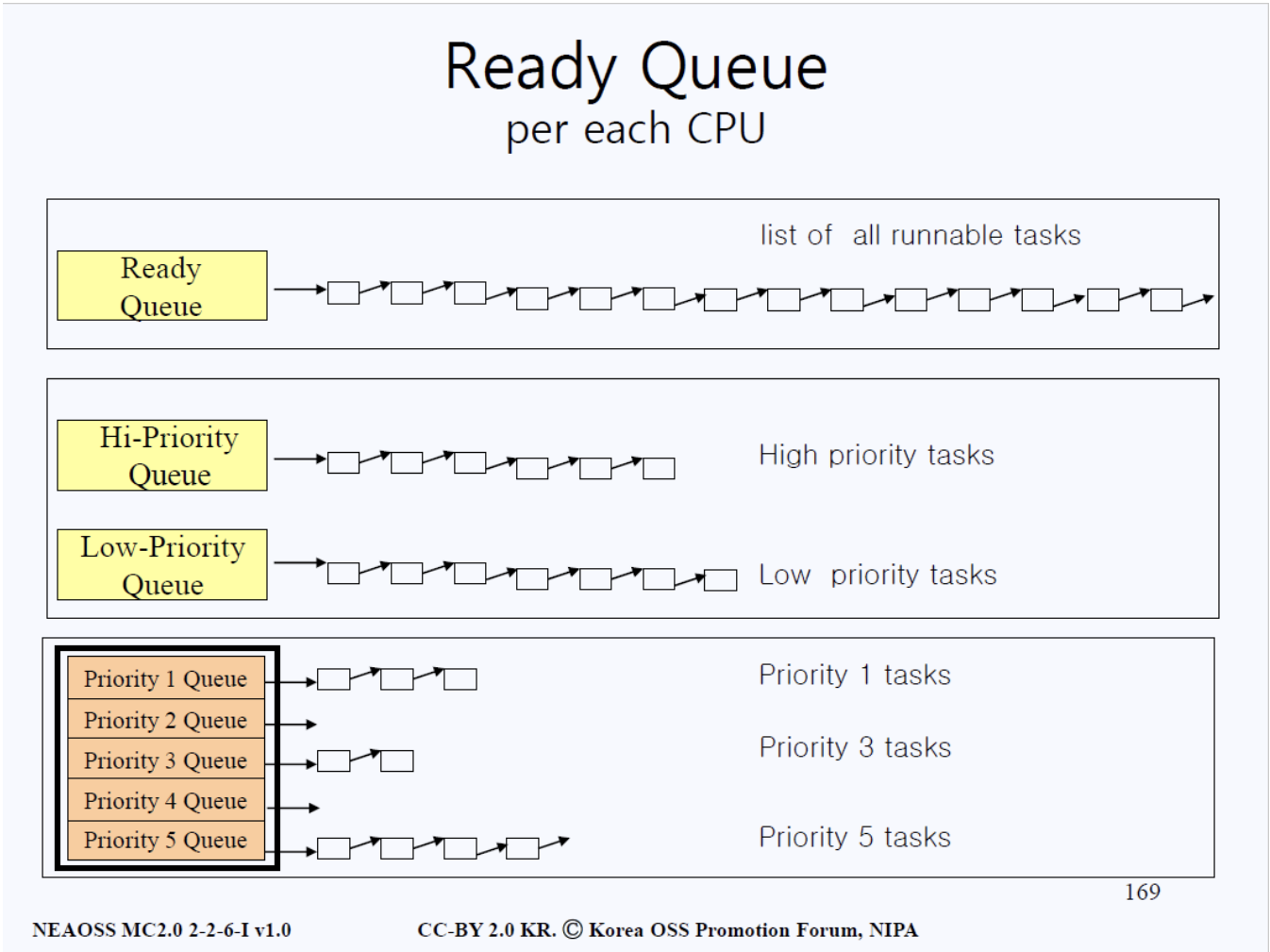
- A task used all the timeslice?
  this task cannot run on CPU
  wait until all other tasks exhausted their timeslices
  recalculate timeslice for this task based on priority



먼저 위쪽 네모박스 안의 내용을 살펴보자. **CPU가 어떤 프로세스에게 할당될 때는 일종의 시간제한**이 있게 된다. 위 예시에서는 100ms라고 되어 있다. 하지만 안타깝게도 이마저도 못쓰게 되는 경우가 발생할 수 있다. 현재 프로세스보다도 더 급한 작업이 요구되었을 때, CPU를 또다시 빼앗길 수 있다.

위 네모박스 안에서는 **20ms를 사용하다가 CPU를 뺏겨버렸고 남은 80ms는 추후에 다시 CPU를 받았을 때 사용**해야 한다. 여기서 **남은 80ms를 remaining timeslice**라고 한다.

즉, 리눅스에서 프로세스가 CPU를 차지하기 위해서는 **우선순위가 높아야 하고 남은 타임슬라이스가 0보다 커야 한다.** 아래 그림을 보면서 스케쥴링이 정확하게 어떻게 이루어지는지 알아보자.



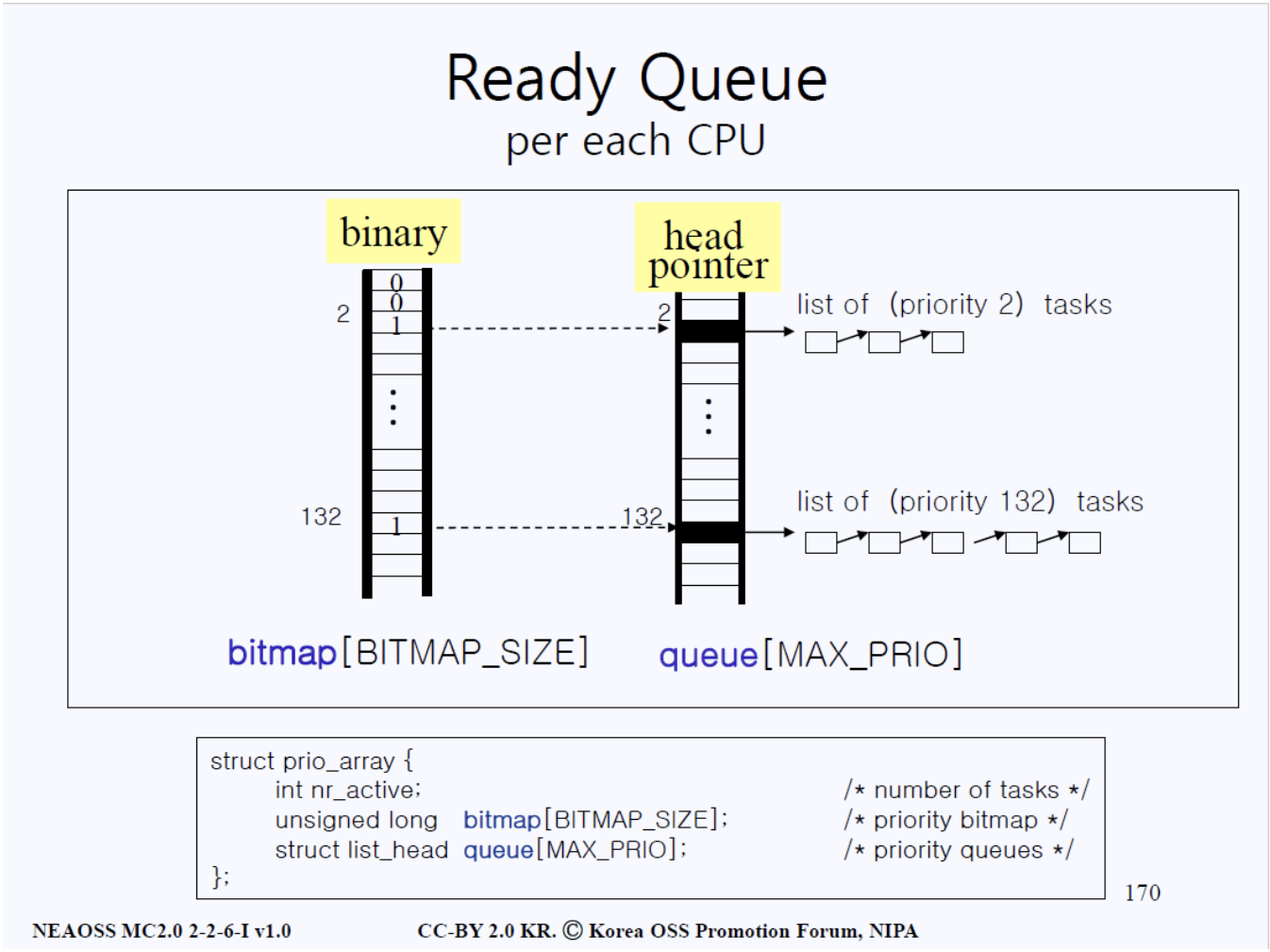
먼저 맨 위의 레디큐**(Ready Queue)**를 살펴보자. 레디큐에는 **PCB**가 여러개 들어 있다**.** 바로 실행**(run)**할 수 있는 작업들이 쭈욱 연결되어 있는 것이다.

이런 구조에서의 **문제점은 멀티 프로세서 시스템이 되서 CPU의 갯수가 증가하게 되면 연결된 머신의 수에 따라 레디큐에 연결되는 PCB도 기하급수적으로 많아질 것**이라는 데 있다. 500개의 프로세스가 돌고 있다고 가정한다면, `context_switch()` 를 실행할 때마다 이 500개에 해당하는 레디큐의 내용을 전부 뒤져서 우선순위가 높은 프로세스를 골라내야 하는데, 탐색 작업이 상당히 비효율적(시간이 오래걸림)이라는 문제가 생긴다.

그래서 이번에는 **위 그림의 중간에 있는 구조(큐를 두 개로 나눈 구조)로 설계**를 해 보았다. 하나의 레디큐로 두는 것은 탐색하는데 비효율적이라고 생각으로부터 나온 설계다. **높은 우선순위와 낮은 우선순위를 가지는 두 개의 큐를 제작**했다. 하나의 큐 보다는 확실히 효율적이겠지만, 이마저도 만족스럽지 않기에 **위 그림의 맨 아래와 같은 설계(여러 개의 큐를 두는 구조)**를 해봤다. **좀 더 분류를 세분화해서 여러 개의 큐**를 만들었다. 각 큐는 `PCB` 를 가리키는 포인터로 이루어져 있다.

하지만 주황색으로 구현된 큐를 보니 중간 중간 비어 있는 큐(2번과 4번 큐)도 보인다. 이러한 큐까지 탐색할 필요는 없으니 좀 더 효율적으로 탐색을 해볼 수 있지 않을까? 라는 생각을 할 수 있다.

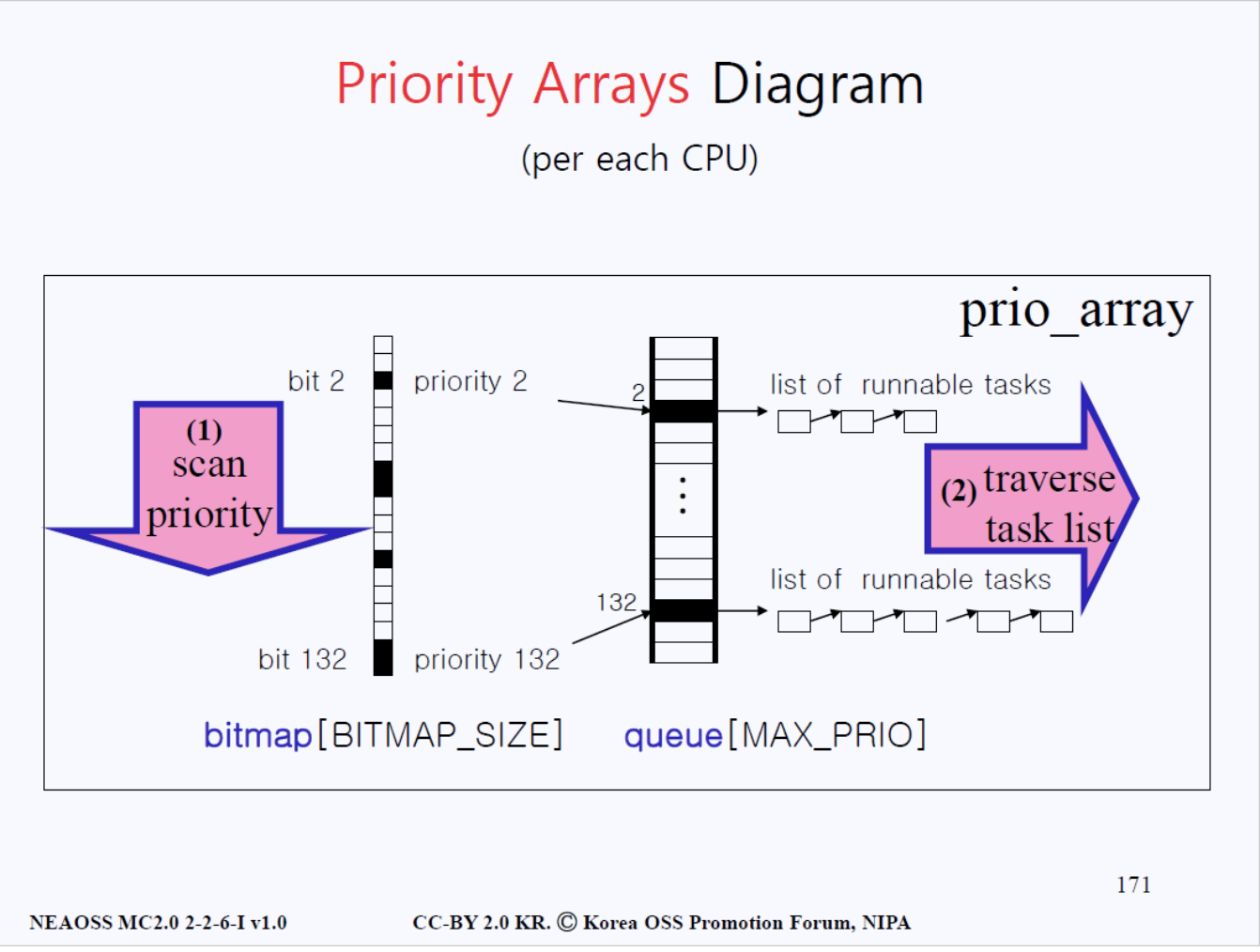
따라서 아래 그림과 같이 **해당 우선순위에 해당하는 큐가 비었는지 안 비었는지를 체크할 수 있는 비트 배열**을하나 더 두게 된다.



위 그림의 좌측에는 바이너리 배열이 하나 있는데, **0은 해당 인덱스의 포인터를 따라가면 해당 인덱스의 큐가 비어 있다**는 뜻이고 **1이라면 해당 인덱스의 큐에 내용물이 있다**는 뜻이다. **유닉스에서는 이 바이너리 배열을 비트맵**이라 부른다.

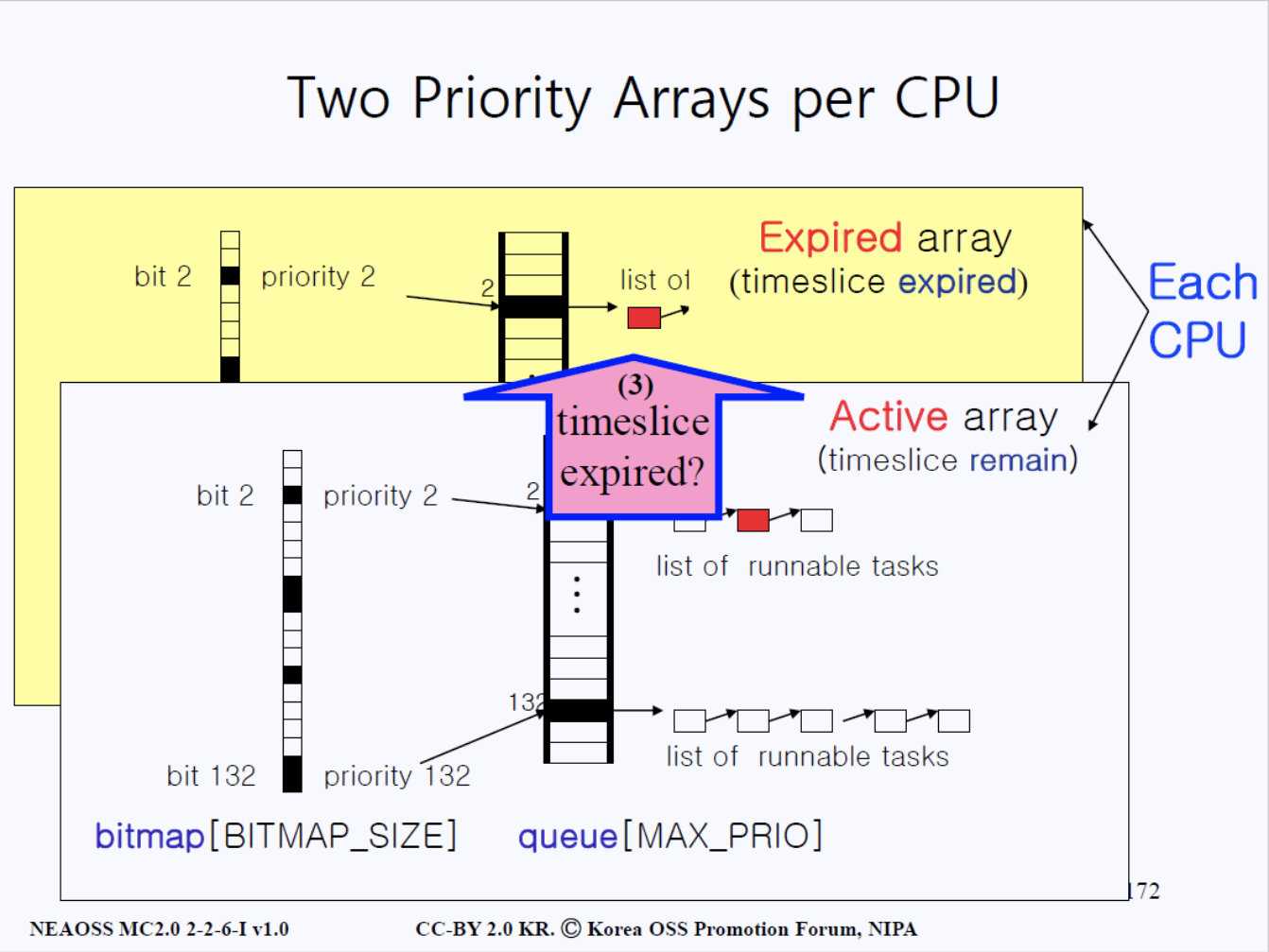
이렇게 비트로된 배열을 사용함으로써 **탐색 속도가 향상**될 수 있다. 이 비트맵의 인덱스가 얼마나 존재하는지가 시스템에서 다루는 난이도가 얼마나 세분화되어 있는지를 나타낸다. 132개의 인덱스가 존재한다고 가정한다면, 우선순위가 132개로 나뉘어진다는 뜻이다. **우측에는 큐로 이루어진 배열**이 그 구현체다. **해당 큐의 각 내용물은 PCB**로 이루어져 있다.

위 그림의 맨 밑을 보면 **구조체가 하나 존재**한다. 그 구조체 **안에는 비트맵과 큐 배열**이 존재한다. 멀티 프로세서 시스템에서 CPU가 10개 있다고 한다면, 이 구조체가 **각 CPU마다 존재**한다고 생각하면 된다. 비트맵과 큐 배열을 함께 포함하고 있는 구조체가 바로 **priority array (우선순위 배열)**이다. 아래 그림을 살펴보자.

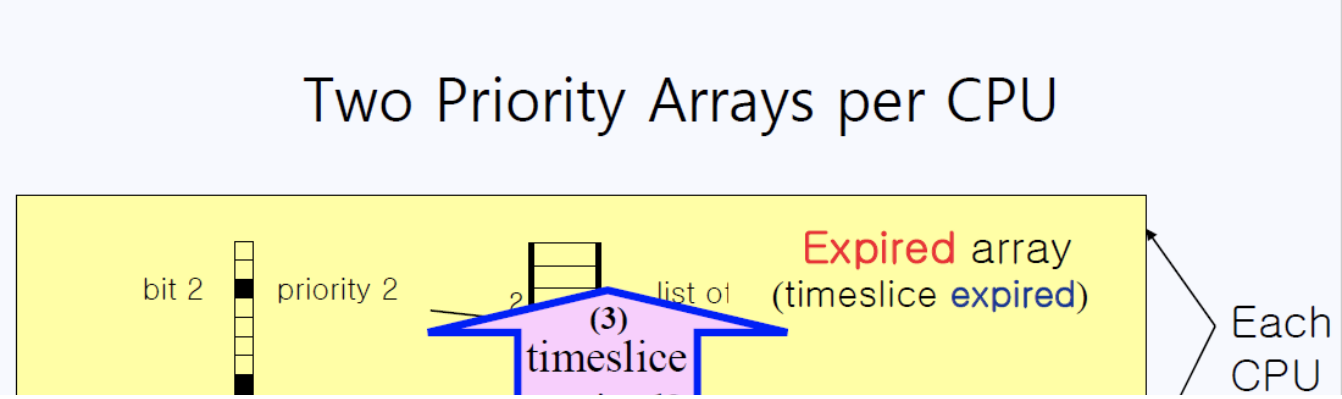


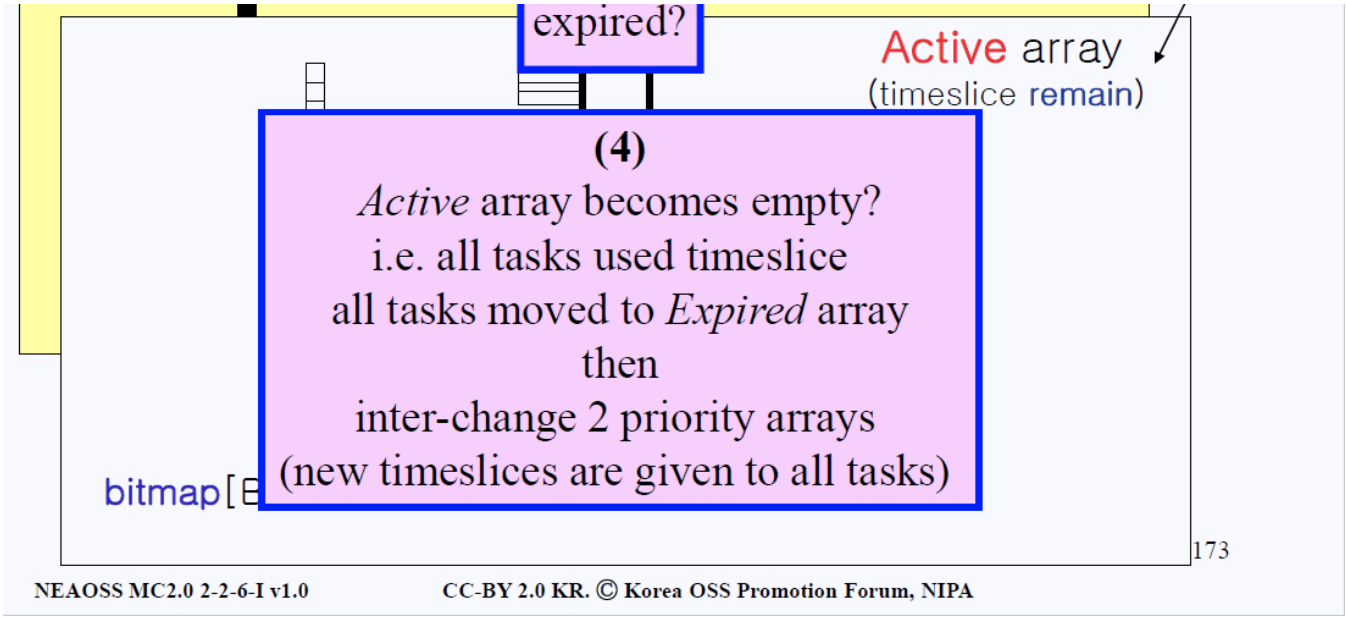
CPU 스케쥴러가 `context_switch()` 가 일어날 때마다, 레디큐에서 우선순위가 가장 높은 프로세스를 뽑아야 하는데 이때 해야 할 작업은 **비트맵을 스캔하고 0이 아닌 항목이 있다**

면, 해당 난이도의 큐를 찾아내서 큐의 작업내역을 순회하며 실행하는 것이다. 0이 아닌 비트맵이 있다면 포인터를 따라가서 해당 큐의 작업 내역을 실행하면 된다. 아래 그림을 살펴보자.

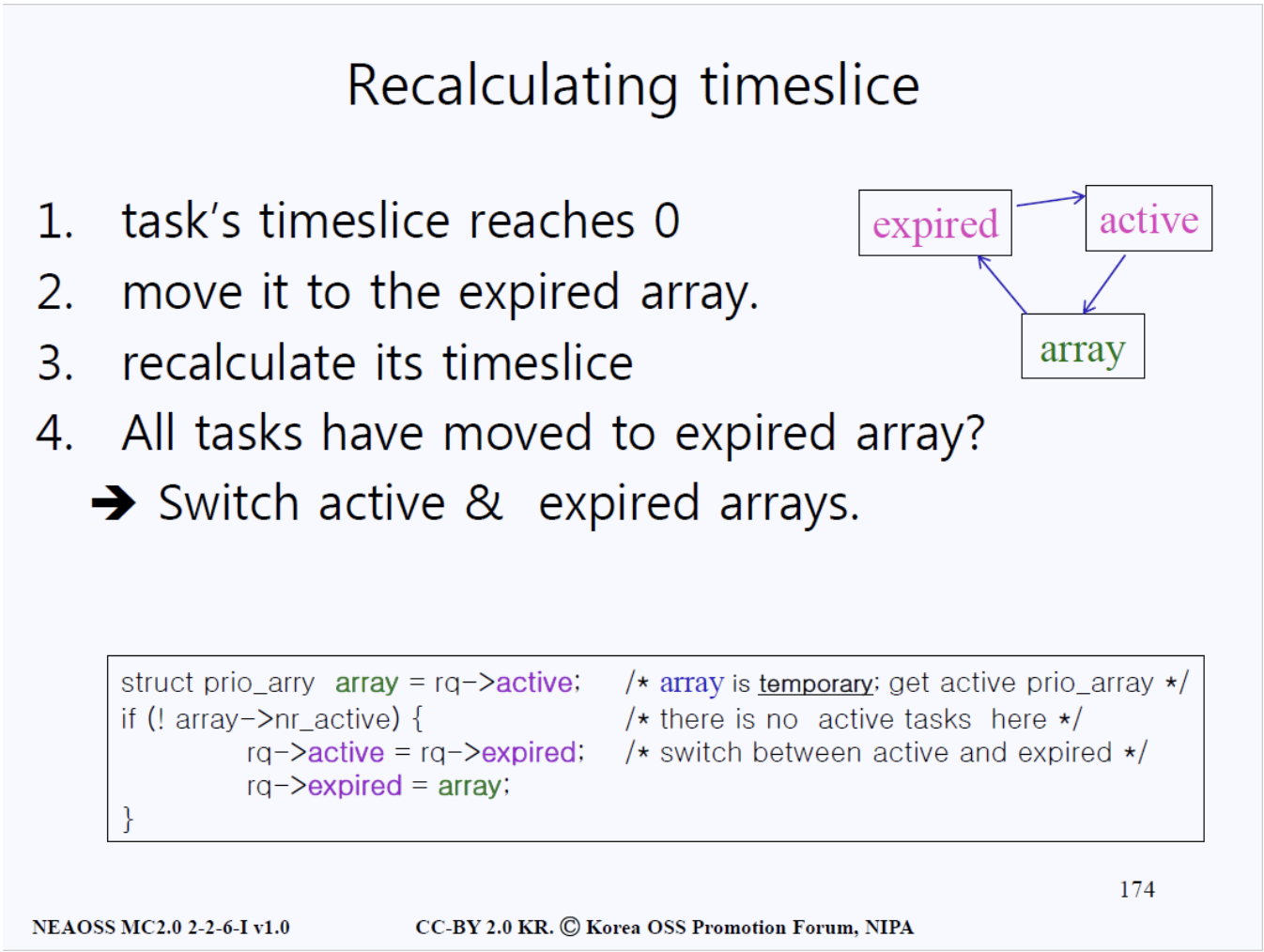


레디 큐라는 것은 **ready to run a cpu**를 의미한다. 즉 바로 실행될 수 있는 프로세스들을 모아둔 큐이다. 하지만 만약 레디 큐에 있던 프로세스가 자신에게 할당된 타임 슬라이스를 다 썼다면어떻게 해야할까? 타임 슬라이스를 다 사용한 프로세스는 위의 그림에서 나타난노란색 영역**(Expired array)**으로 빠지게 된다. 다음 타임 슬라이스를 배정받기 전까지는 레디 큐에 있을 자격이 없기 때문에 **Expired array**에서 대기하게 된다. 이 두개의 **array**는 각 **CPU**마다 존재한다.

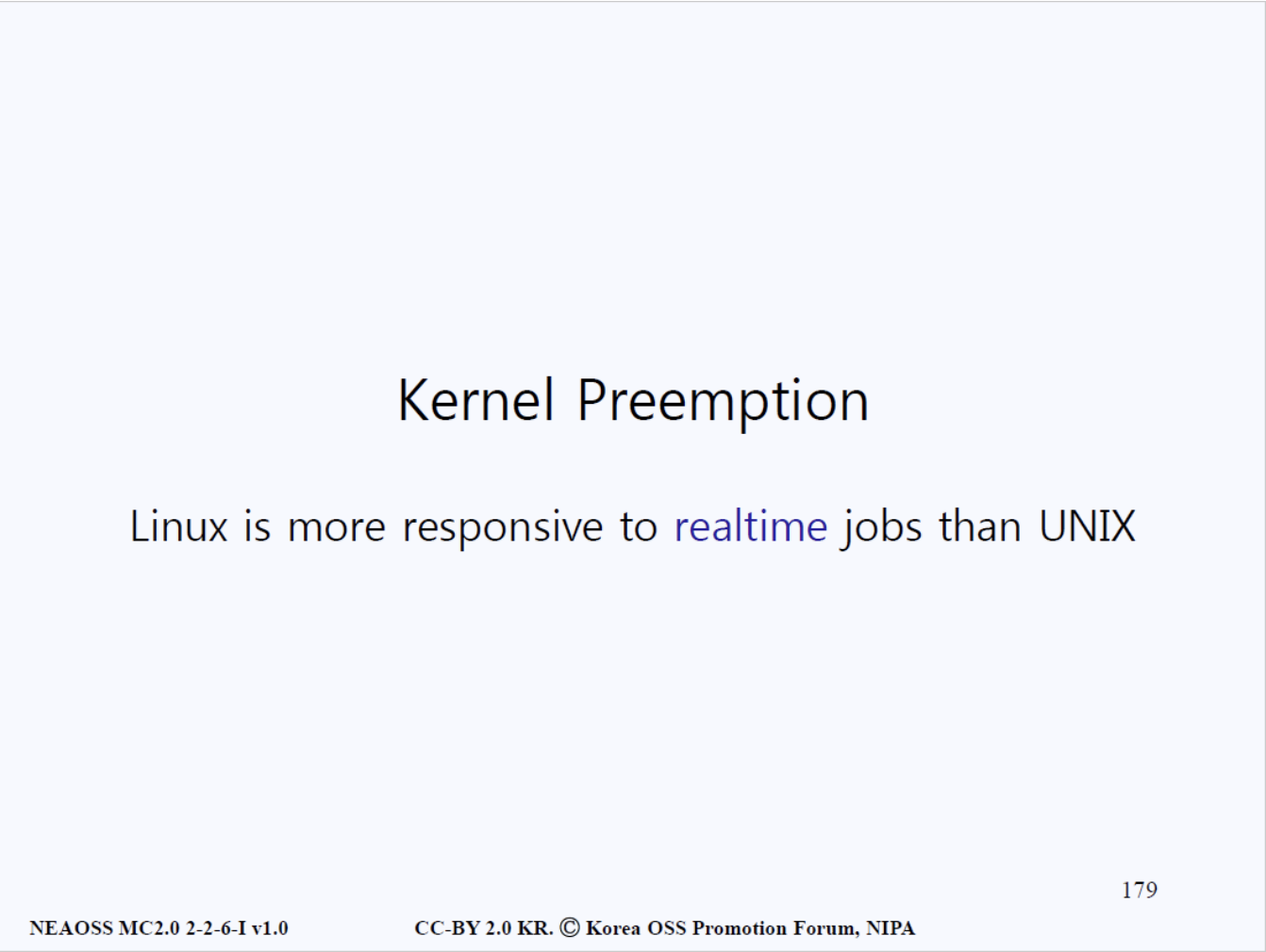




좀 더 정확하게 설명을 하자면, **Active** 영역에 있는 작업들이 모두 처리된 후 **Expired** 영역으로 가게 된 프로세스들은 일률적으로 **(한 번에)** 타임 슬라이스를 배정받게 된다. 그 뒤 **Expired** 영역이 **Active** 영역으로, 기존의 **Active** 영역은 **Expired** 영역으로 변환**(interchange)**된다**.** 이 변환 과정은 단순히 서로가 가리키는 포인터가 바뀌는 작업을 의미하는데 구현 코드는 아래 그림의 맨 아래 네모박스에 나와있다.



4. Kernel Preemption



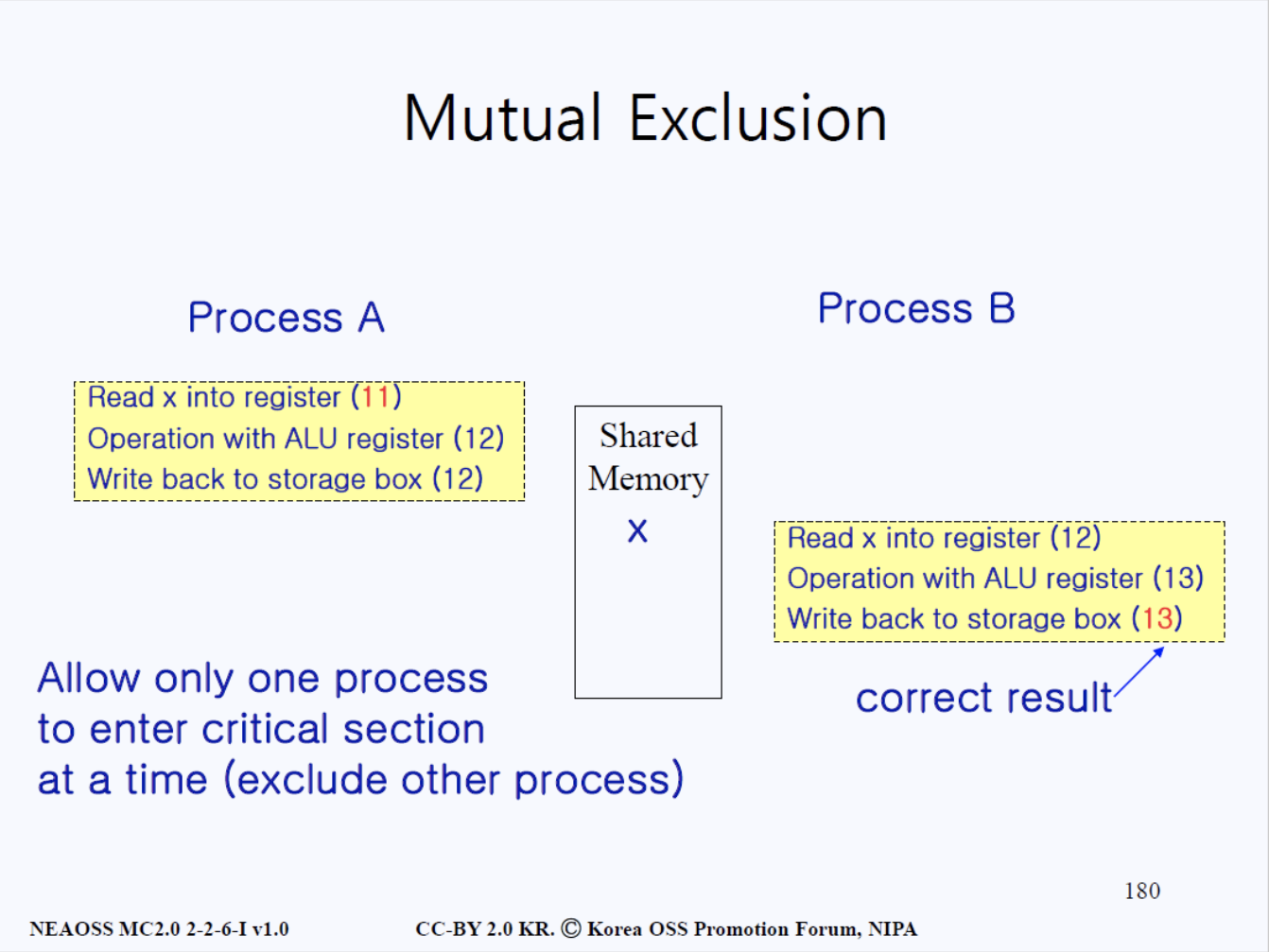
4.1 Mutual Exlusion — 상호 배제

컴퓨터 시스템을 얘기할 때 가장 중요한 파트 중 하나가 **상호 배제 문제**이다. 시스템이 정상적으로 작동하기 위해서는 이 상호 배제 개념은 반드시 필요하다.

설명을 진행하기 전, 먼저 `x++` 이라는 연산이 정확하게 어떻게 이루어져 있는지부터 이해하고 가자. 우리가 보통 프로그래밍 언어를 사용할 때 `x++` 과 같은 단항연산자를 사용하면, 하나의 명령만으로 덧셈이 정상적으로 이루어지는 것 같지만 실제로 동작하는 기계 입장에서 이 단항연산 과정은 3단계로 나누어진다. 그 과정은 아래와 같다.

1. `x` 를 저장소로부터 읽어서 `CPU 레지스터` 로 읽어들인다.
2. `CPU` 안에서 `ALU` 연산을 진행한다.
3. `CPU` 로부터 나온 결과를 다시 저장소에 쓴다.

먼저 우리가 흔히 사용하는 **X와 같은 변수 또는 데이터는 항상 저장소(storage)에 존재**한다. CPU 안에는 저장시킬 수 있는 용량이 얼마 없고 또 비싸기 때문에 CPU 안에 많은 변수와 데이터를 저장할 수는 없다. 때문에 우리가 흔히 알고 있는 컴퓨터 시스템에서 **데이터는 저장소에 저장되고 연산은 CPU에서 이루어지게 되는 것**이다. 이 개념을 먼저 숙지한 후 아래의 그림과 함께 설명을 보자.



위 그림을 보면 두 개의 프로세스가 존재하고, 이 두 프로세스는 **한 개의 변수 X를 공유**하고 있다. 프로세스 A가 먼저 `x`에 대한 `x++`를 연산한 후 메모리에 저장하고, 그 다음 우측의 프로세스 B가 변수 `x`를 레지스터로 읽어들여 연산처리를 한 후 디스크에 저장한다. 이렇게만 한다면, `x`라는 변수는 정상적으로 `11 -> 12 -> 13` 순으로 디스크에 저장될 것이다.

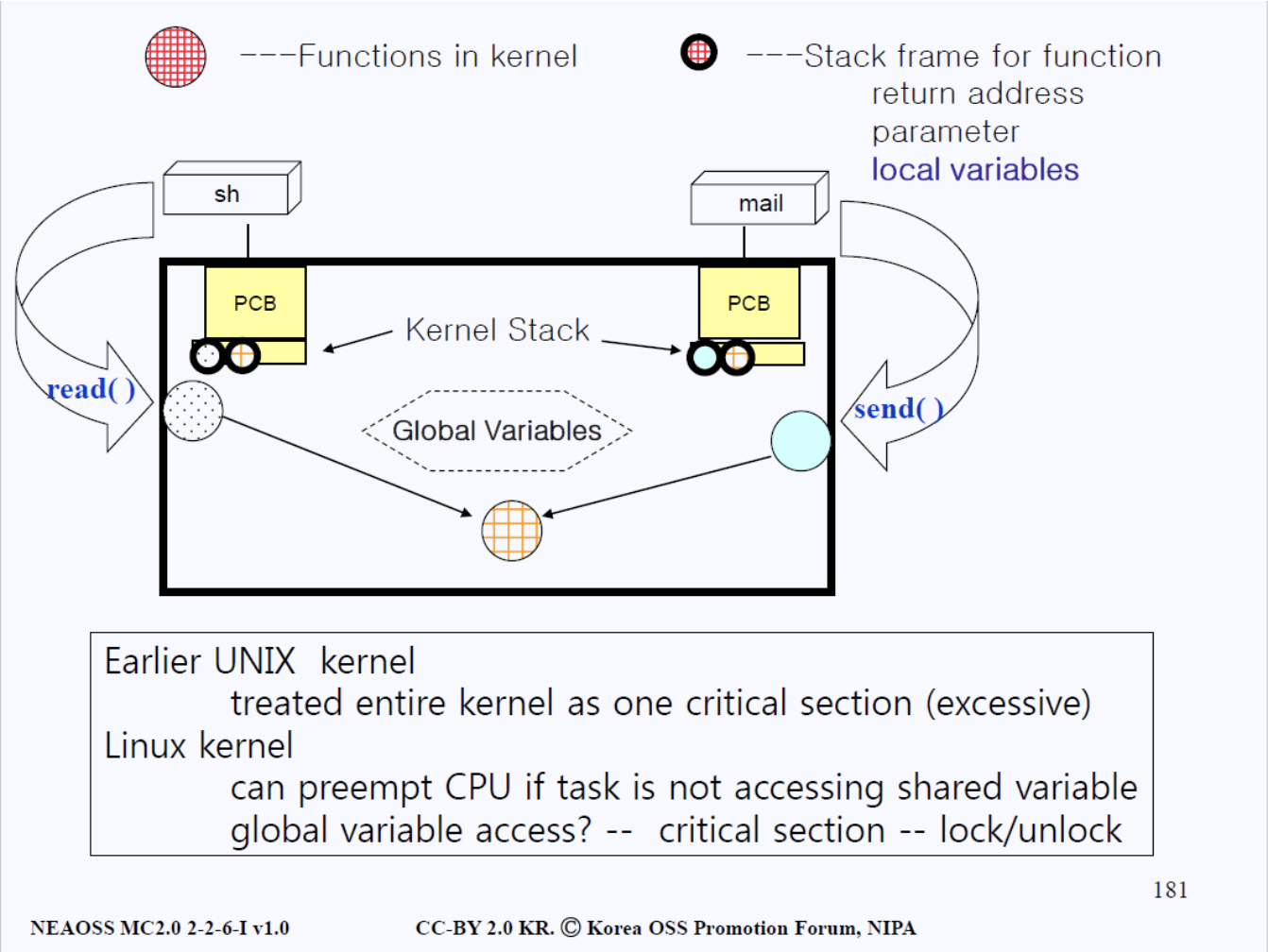
그런데 만약 이 두 프로세스가 `x`를 동시에 읽어들여 연산할 경우에는 문제가 발생하게 된다. A가 `x`를 읽은 후에 `++` 연산을 진행하는 도중 B가 `x`를 읽어들였다고 가정해보자. A는 아직 더한 값을 디스크에 쓰지 않았기 때문에 아직 `x`는 초기값 `11` 그대로다. 따라서 B가 읽어들인 `x` 값은 `11`이다.

여기서 A가 값을 12 로 증가시킨 후 디스크에 12 를 기록했다고 해도, 결국 B또한 12 로 증가시킨 후에 기록하기 때문에 덧셈은 한 번밖에 일어나지 않는다. **두 번의 덧셈연산이 제대로 동작하지 않은 것이다.** 이처럼 프로세스 간 **공유된 변수를 접근하는 부분을 Critical Section**이라 부른다.

크리티컬 섹션에는 하나의 프로세스만 접근해야만 한다. 그래야 위와 같은 오류가 발생하지 않는다. **이러한 원칙이 바로 상호 배제(Mutual Exclusion)의 원칙**이다. 크리티컬 세션은 하나의 프로세스만 접근이 가능하다. 또 그래야만 한다.

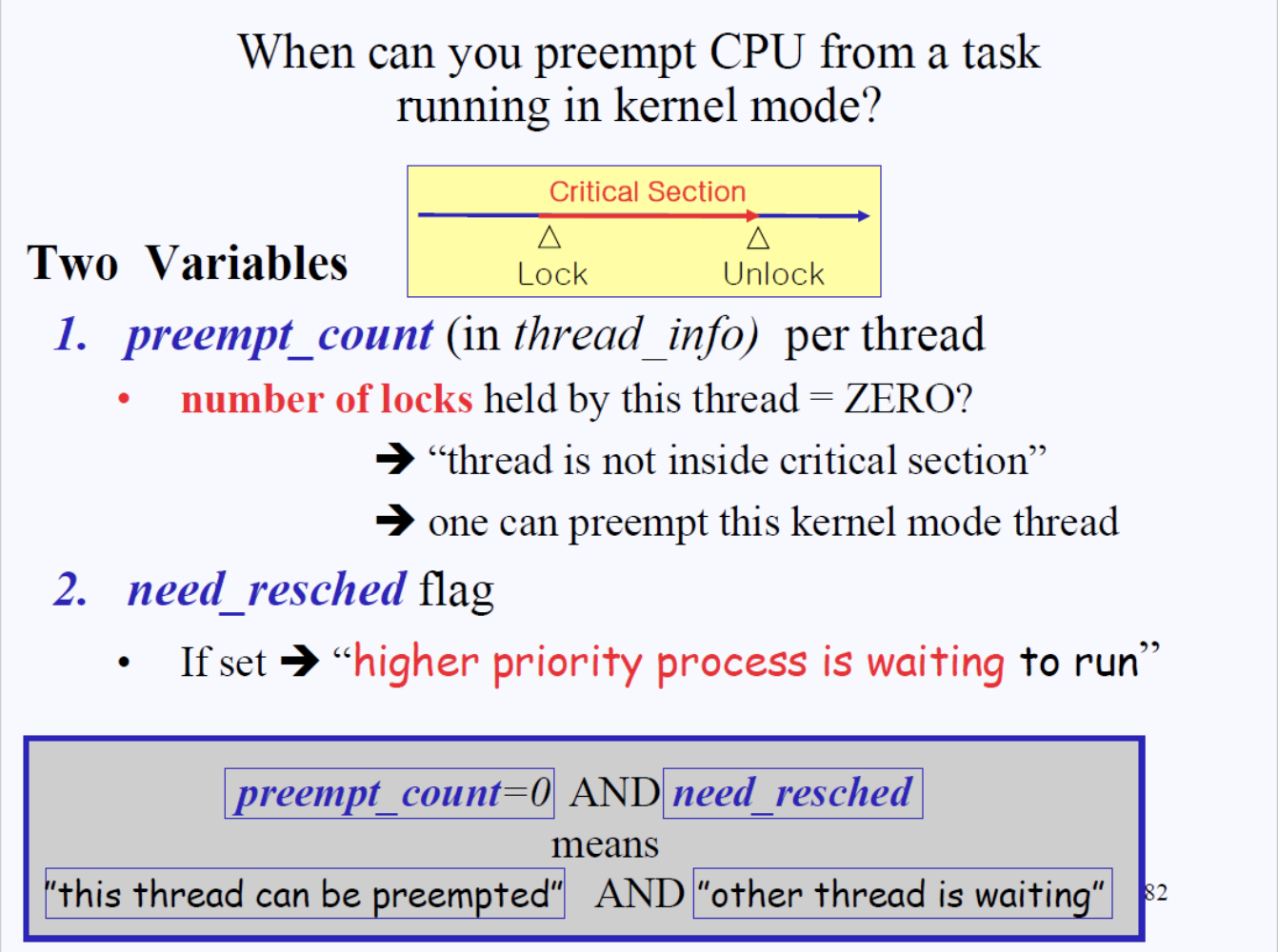
유닉스는 지난 40년간 이러한 문제를 어떻게 해결했을까? 그 방법은 매우 단순하게도, **커널모드인 경우에는 CPU를 뺏지 않고 유저모드일 경우에만 CPU를 뺏는다. 커널에 있을 때는 CPU preemption을 고려하지 않아도 된다.**

하지만 이러한 설계에는 문제가 있다. **중요한 작업이 도중에 발생했다고 해도, Kernel 모드이기 때문에 CPU를 다른 곳에 할당하지 못한다면 리얼타임시스템(real-time system)과 같이 빠른 처리와 빠른 전환이 어려워진다.** 커널이 작업중임에도 CPU를 가져올 수 있어야 진정한 리얼타임시스템이 가능하다. 이 부분을 어떻게 해결할지가 아래 그림에 나와있다.



위 그림의 좌측 상단을 보면, 쉘(sh) 이 read() 시스템 콜을 호출하고 있다. 함수 안에 변수들이 쓰이면서 스택에 이러한 변수들이 담기게 된다. 우측의 mail 프로그램에서도 send() 시스템 콜을 호출하고, 이 호출 또한 지역변수를 자신만의 스택에 담는다. **그러나 둘은 같은 공용 변수에 접근하고 있다. 어떻게 하면 두 프로세스가 원활하게 동작이 가능할까?**

리눅스에서는 **공용 변수에 접근할 때만 따로 lock을 건다.** 접근이 끝났다면 unlock을 한다. **즉, lock이 되어 있다면 커널모드이건 아니건 CPU를 뺐는 일은 발생하지 않는다.** 그러나 **unlock이라면 크리티컬 세션이 아닌 것이므로 커널 모드임에도 CPU를 다른 프로세스에게 할당하는 것이 가능하다.** 리얼타임시스템을 고려한 리눅스에서의 설계다.



위의 preempt_count 가 바로 lock의 갯수 이다. CPU를 뺏으러 왔을 때, **preempt_count가 0이면 공용 변수에 접근하는 프로세스가 하나도 없다는 것이기 때문에 CPU를 뺏을 수 있다.** need_resched 의 경우는 리얼타임 시스템이 CPU를 다른 프로세스에 할당해주기 위해 왔는데, preempt_count 가 0 이 아니여서 뺐을 수는 없으니 "지금 다른 우선순위 높은 프로세스가 CPU를 기다리고 있어!"라는 표시를 해주는 용도로 사용한다. 따라

서 **preempt_count**가 **0**으로 되고 **need_resched** 플래그가 세트되어 있다면, 현재 공용변수에 접근하고 있는 것이 없으니 **CPU**를 다른 프로세스에게 할당해도 좋다는 의미가 된다.

5. 마치며

프로세스 스케쥴링을 할 때 우선순위와 타임슬라이스, 그리고 크리티컬 영역까지 고려한다는 점을 알게 되었다. 커널을 설계하는 개발자들이 얼마나 치밀하고 효율적으로 설계하기 위해 노력했는지를 간접적으로나마 알 수 있었다. 커널과 같은 하나의 훌륭하고 완벽한 프로그램을 만드는 그 날까지 노력해야겠다.