

---

# 실시간 시스템 (6주차)

## 5주차 Review

---

- Chapter 6 of An Embedded Software Primer
- Multitasking
- Scheduling
  - Context Switching
- Preemptive & Non-Preemptive Kernel
- Shared Data Problem
  - Reentrant Function

---

# RTOS: SCHEDULING은 실제로 어떻게 구현되는가?

# 스케줄링 Details

---

- Priority-based 스케줄러는
  - ready상태의 task들 중에서
  - 우선 순위가 가장 높은 태스크를 찾아서 실행 기회를 주어야 한다.
- 스케줄러는 ready상태 task를 어떻게 관리하고,
- 우선 순위가 가장 높은 task를 어떻게 빠른 시간 안에 찾아내는 것일까?
- uC/OS-II의 실제 스케줄링 코드를 통해서 알아보자.

# uC/OS-II의 task 우선순위

- uC/OS-II의 기본
  - 각 task의 우선 순위는 unique하다.
  - 가장 높은 우선 순위: 0
  - 가장 낮은 우선 순위: OS\_LOWEST\_PRIO
    - 이 값은 사용자가 지정 가능하다.
  - 가장 낮은 우선 순위는 idle task에게 할당된다.
  - 최대 생성 가능한 task 개수는 64개이다.
    - 따라서, 0부터 63까지가 할당가능한 우선 순위
    - 이 들 중, 8개는 OS service에 사용되므로, 사용자가 사용 가능한 최대 개수는 56개이다.
- uC/OS-II의 스케줄링 소요시간은 constant이다.
  - 다음 번 수행할 task를 선택하는 데 소요되는 시간이,
  - ready 상태의 task 개수에 상관없이 일정하다.
  - 따라서, 실시간 스케줄링에 적합하다.

# uC/OS-II의 ready list

- 실시간 스케줄링을 위해서,
  - ready list를 이용한다.
  - ready list는 ready 상태의 task를 저장하는 곳으로,
  - ready 상태로 전이한 task들을 추가할 수 있고,
  - running상태로 전이하는 task들을 제거할 수 있다.
- Ready list는
  - 추상적으로 생각하면
    - ready 상태의 task들이 일렬로 늘어선 큐의 형태지만
  - “큐” 면 안되는 이유 I
    - FIFO 방식으로 task들의 실행 순서가 정해지지 않기 때문이다.
    - 우선 순위에 따라 정해져야 한다.
  - “큐” 면 안되는 이유 II
    - 우선 순위에 따른 탐색 시간이 길어진다.
    - 큐 안의 task 개수에 따라 탐색 시간이 영향을 받으면 안 된다.

# uC/OS-II의 ready list

---

- Ready list는 다음 두 변수를 사용하여 구현한다.
  - OSRdyGrp: 8-bit 정수 변수
  - OSRdyTbl[8]: 8-bit 정수 8개를 저장하는 배열

# uC/OS-II의 ready list


- OSRdyTbl[8]: 8-bit 정수 8개를 저장하는 배열

OSRdyTbl[0]	7	6	5	4	3	2	1	0
OSRdyTbl[1]	15	14	13	12	11	10	9	8
OSRdyTbl[2]	23	22	21	20	19	18	17	16
OSRdyTbl[3]	31	30	29	28	27	26	25	24
OSRdyTbl[4]	39	38	37	36	35	34	33	32
OSRdyTbl[5]	47	46	45	44	43	42	41	40
OSRdyTbl[6]	55	54	53	52	51	50	49	48
OSRdyTbl[7]	63	62	61	60	59	58	57	56



# uC/OS-II의 ready list

- OSRdyTbl[8]: 8-bit 정수 8개를 저장하는 배열



OSRdyTbl[0]	7	6	5	4	3	2	1	0
OSRdyTbl[1]	15	14	13	12	11	10	9	8
OSRdyTbl[2]	23	22	21	20	19	18	17	16
OSRdyTbl[3]	31	30	29	28	27	26	25	24
OSRdyTbl[4]	39	38	37	36	35	34	33	32
OSRdyTbl[5]	47	46	45	44	43	42	41	40
OSRdyTbl[6]	55	54	53	52	51	50	49	48
OSRdyTbl[7]	63	62	61	60	59	58	57	56

# uC/OS-II의 ready list

- OSRdyTbl[8]: 8-bit 정수 8개를 저장하는 배열

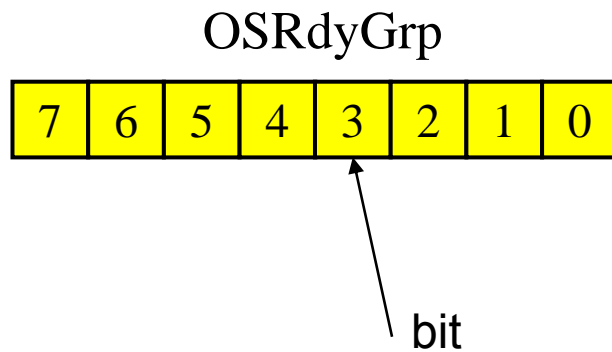
각 숫자는 task의 우선순위  
- 0이 가장 높음  
- 해당 bit가 1이면 ready상태

OSRdyTbl[0]	7	6	5	4	3	2	1	0
OSRdyTbl[1]	15	14	13	12	11	10	9	8
OSRdyTbl[2]	23	22	21	20	19	18	17	16
OSRdyTbl[3]	31	30	29	28	27	26	25	24
OSRdyTbl[4]	39	38	37	36	35	34	33	32
OSRdyTbl[5]	47	46	45	44	43	42	41	40
OSRdyTbl[6]	55	54	53	52	51	50	49	48
OSRdyTbl[7]	63	62	61	60	59	58	57	56

# uC/OS-II의 ready list

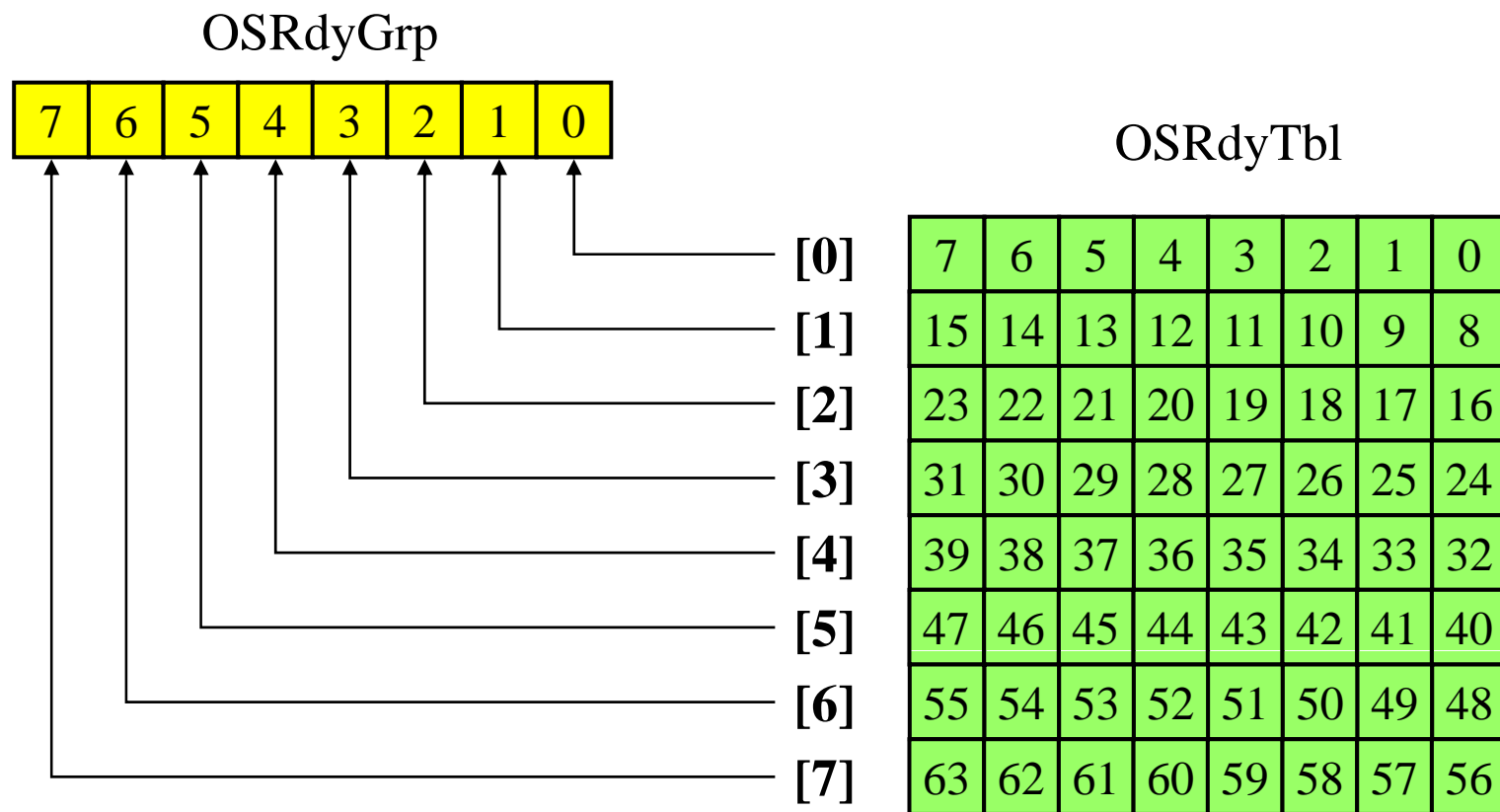
---

- OSRdyGrp: 8-bit 정수



# uC/OS-II의 ready list

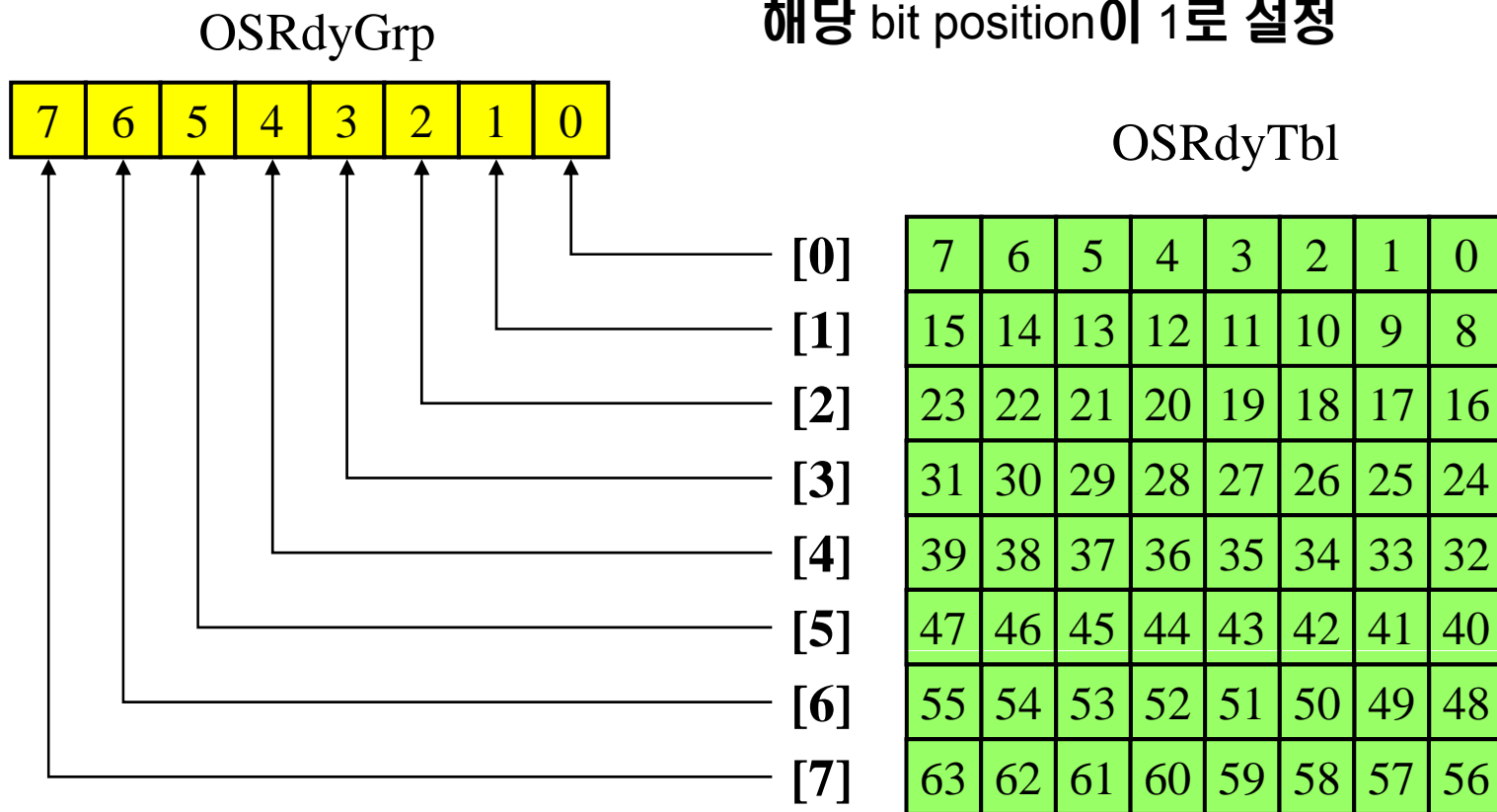
- OSRdyGrp: 8-bit 정수



# uC/OS-II의 ready list

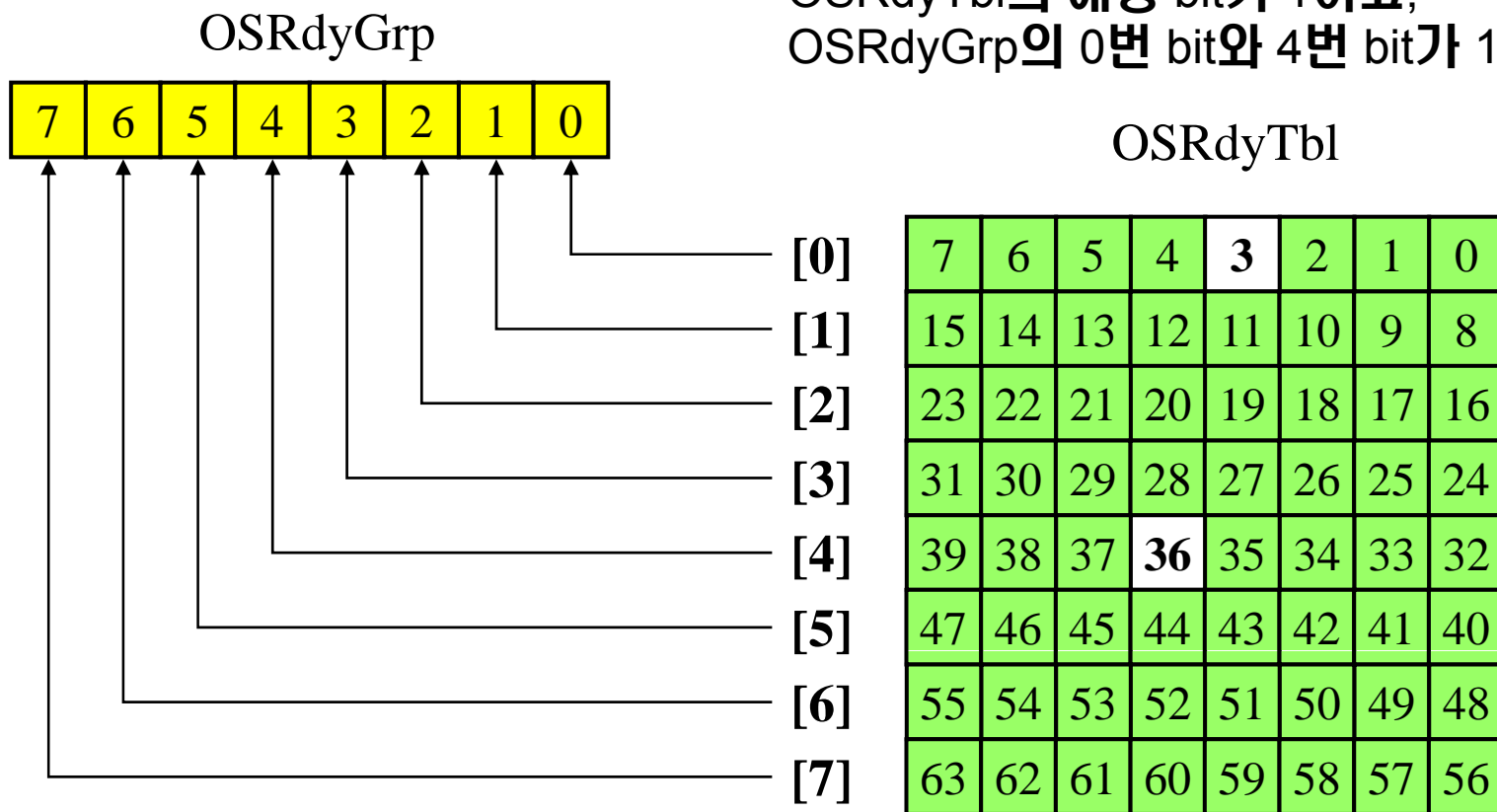
- OSRdyGrp: 8-bit 정수

해당하는 그룹에 1이 하나라도 있으면,  
해당 bit position이 1로 설정



# uC/OS-II의 ready list

- OSRdyGrp: 8-bit 정수 예를 들어,  
우선 순위 3 task와 36 task가 ready 상태이면,  
OSRdyTbl의 해당 bit가 1이고,  
OSRdyGrp의 0번 bit와 4번 bit가 1이 됨.



# uC/OS-II의 ready list

- 잠깐 정리,
  - OSRdyTbl[]은
    - 각 task의 고유 우선순위에 해당하는 bit을 가지고 있고,
    - 해당 task가 ready 상태가 되면 해당 bit을 1로 설정
  - OSRdyGrp은
    - 우선순위를 8개씩 묶어서, group을 만들고,
    - 각 bit는 group에 해당
    - group에 속하는 task 들 중에 하나라도 ready상태이면 해당 bit는 1로 설정
  - Ready 상태로 전이하는 task가 있으면 해당 우선순위의 bit 값을 OSRdyTbl[]과 OSRdyGrp에서 찾아서 1로 설정
  - Ready상태에서 Running 상태로 전이하는 task가 있으면 해당 우선순위의 bit값을 OSRdyTbl[]과 OSRdyGrp에서 찾아서 0으로 설정

# uC/OS-II의 ready list

- Ready 상태로 전이하는 task가 있으면
  - 해당 우선순위의 bit 값을 OSRdyTbl[]과 OSRdyGrp에서 찾아서 1로 설정
  - 이를 위해, 보조 테이블인 OSMaPtbl[]을 사용
  - OSMaPtbl: OS에서 사용하는 mapping table
    - Mapping: task의 우선순위를 OSRdyGrp/OSRdyTbl[]로 매핑

OSMaPtbl[]:  
8 bit 정수 배열

인덱스	값
0	0000_0001
1	0000_0010
2	0000_0100
3	0000_1000
4	0001_0000
5	0010_0000
6	0100_0000
7	1000_0000



# uC/OS-II의 ready list

- Ready 상태로 전이하는 task가 있으면
  - 해당 우선순위의 bit 값을 OSRdyTbl[]과 OSRdyGrp에서 찾아서 1로 설정
  - 이를 위해, 보조 테이블인 OSMaPtbl[]을 사용
  - OSMaPtbl: OS에서 사용하는 mapping table
    - Mapping: task의 우선순위를 OSRdyGrp/OSRdyTbl[]로 매핑

OSMaPtbl[]:  
8 bit 정수 배열

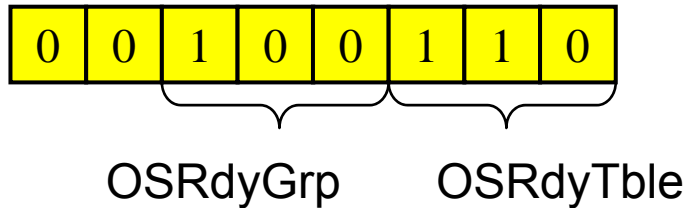
인덱스	값
0	0000_0001
1	0000_0010
2	0000_0100
3	0000_1000
4	0001_0000
5	0010_0000
6	0100_0000
7	1000_0000

# Ready상태 설정 예/1

- 우선순위 38의 task가 ready상태가 되면 다음 코드를 실행하여, 해당 bit를 1로 설정

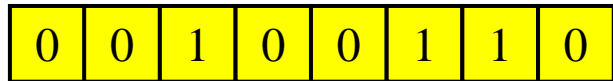
```
OSRdyGrp      |= OSMapTbl[prio >> 3];  
OSRdyTbl[prio >> 3] |= OSMapTbl[prio & 0x07];
```

38은 이진수로



# Ready상태 설정 예/2

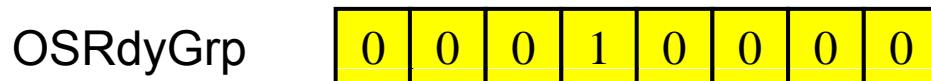
38은 이진수로



OSRdyGrp      OSRdyTble

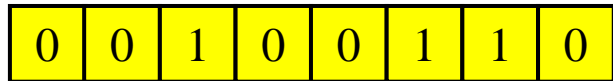
<b>OSRdyGrp</b>	<b><math>\mid = \text{OSMapTbl}[\text{prio} \gg 3];</math></b>
<b>OSRdyTbl[prio &gt;&gt; 3]</b>	<b><math>\mid = \text{OSMapTbl}[\text{prio} \&amp; 0x07];</math></b>

1.  $\text{prio} \gg 3 = 0000\_0100 = 4$
2.  $\text{OSMapTbl}[4] = 0001\_0000$
3. OSRdyGrp (bit-wise-OR) 0001\_0000



# Ready상태 설정 예/3

38은 이진수로



OSRdyGrp      OSRdyTbl

```
OSRdyGrp                    |= OSMapTbl[prio >> 3];  
OSRdyTbl[prio >> 3]      |= OSMapTbl[prio & 0x07];
```

1. prio (bit-wise-AND) 0x07 = 0000\_0110 = 6
2. OSMapTbl[6] = 0100\_0000
3. prio >> 3 = 0000\_0100 = 4
4. OSRdyTbl[4] (bit-wise-OR) 0100\_0000

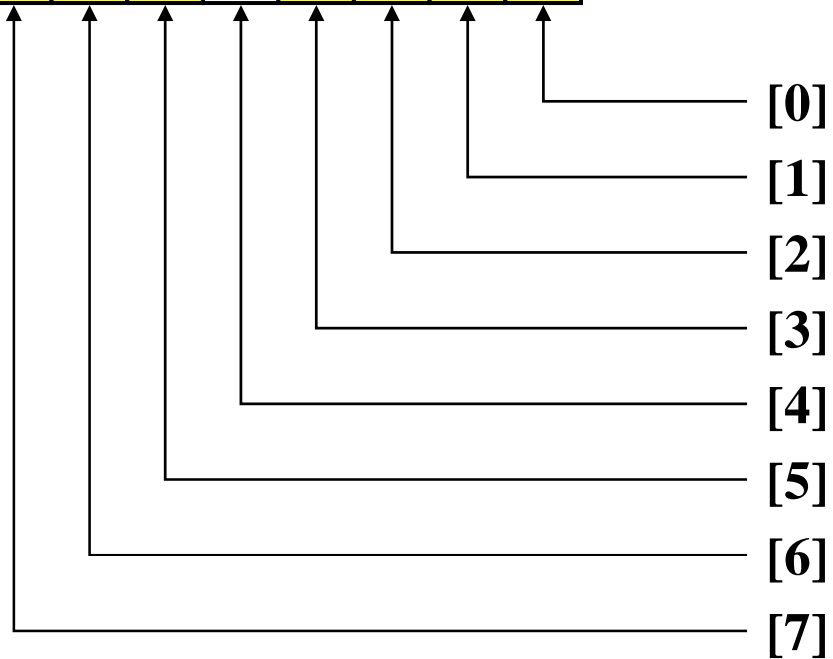
# Ready상태 설정 예/4

38은 이진수로

0	0	1	0	0	1	1	0
---	---	---	---	---	---	---	---

OSRdyGrp

0	0	0	1	0	0	0	0
---	---	---	---	---	---	---	---



OSRdyTbl[8]

7	6	5	4	3	2	1	0
15	14	13	12	11	10	9	8
23	22	21	20	19	18	17	16
31	30	29	28	27	26	25	24
39	38	37	36	35	34	33	32
47	46	45	44	43	42	41	40
55	54	53	52	51	50	49	48
63	62	61	60	59	58	57	56

# Ready상태 해제 예

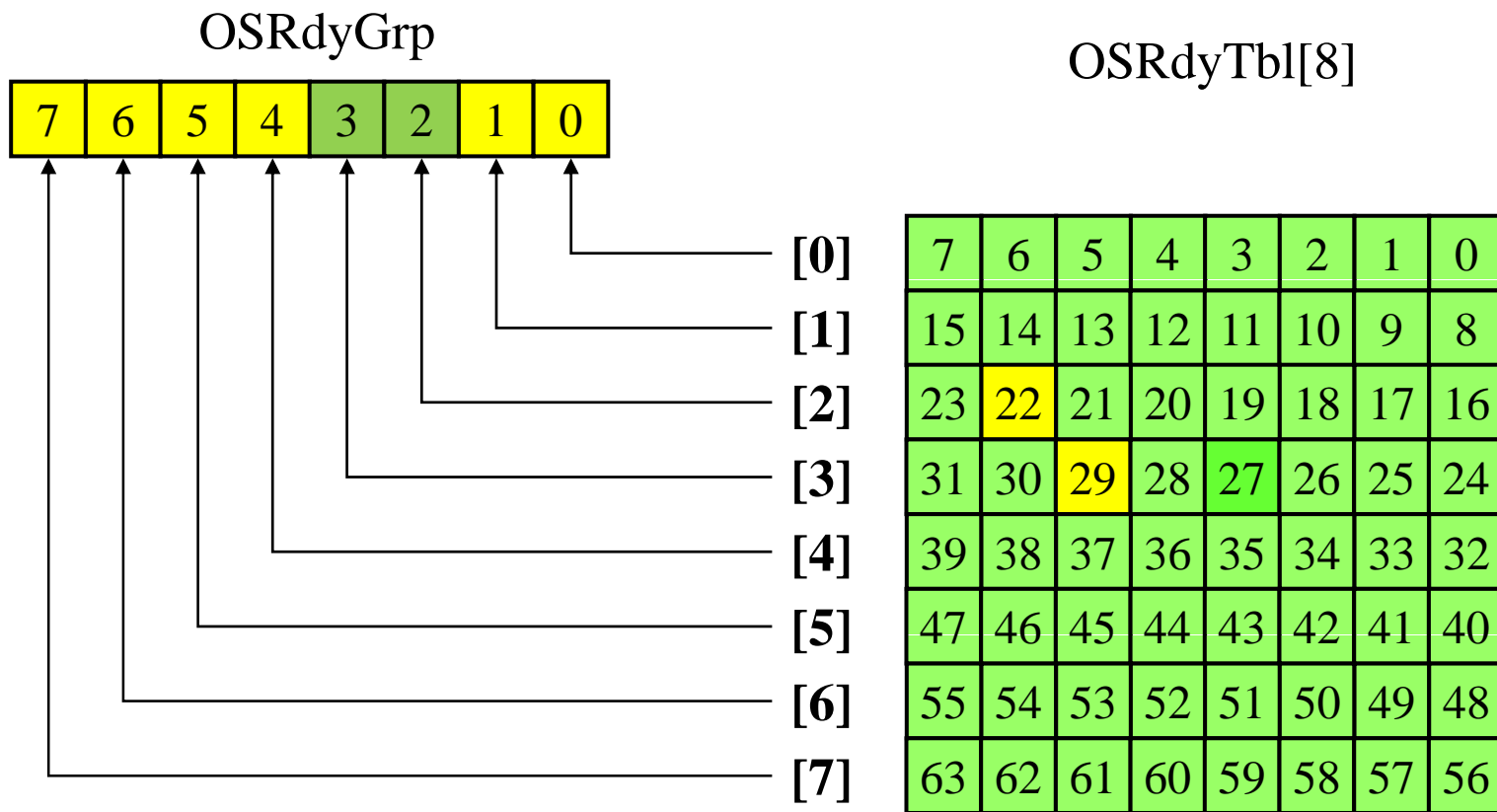
- 우선순위 38의 task가 ready상태에서 running상태로 전이하면 다음 코드를 실행하여, 해당 bit를 0으로 설정

```
OSRdyTbl[prio >> 3]    &= ~OSMapTbl[prio & 0x07];  
OSRdyGrp                &= ~OSMapTbl[prio >> 3];
```

- 아이디어
  - 1로 설정할 때와 마찬가지로, index를 찾고,
  - 값을 설정할 때는 현재 값 (당연히 ready이므로 1)을 negation하여 (0으로 바꾸고),
  - AND 함. (따라서, 0과 AND하므로 해당 bit가 0으로 전환)

# Ready list에서 최우선순위 찾기

우선순위 22와 29를 가지는 task들이 ready상태인 경우,  
running상태로 천이할 task 선정한다면, 당연히 22가 선정되어야 함.



# Ready list에서 최우선순위 찾기

이러한 scheduling 선정과정을 빠르게 진행하기 위해,  
아래와 같은 OSUnMapTbl[]을 이용함.

\*\* OSMapTbl[]과 OSUnMapTbl[]은 OS 초기화할 때 구성됨.

```
INT8U const OSUnMapTbl[] = {
    0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x00 to 0x0F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x10 to 0x1F */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x20 to 0x2F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x30 to 0x3F */
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x40 to 0x4F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x50 to 0x5F */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x60 to 0x6F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x70 to 0x7F */
    7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x80 to 0x8F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x90 to 0x9F */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0xA0 to 0xAF */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0xB0 to 0xBF */
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0xC0 to 0xCF */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0xD0 to 0xDF */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0xE0 to 0xEF */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0 /* 0xF0 to 0xFF */
};
```



# Ready list에서 최우선순위 찾기

- 최우선 순위를 찾아내는 방법

```
y = OSUnMapTbl[OSRdyGrp];  
x = OSUnMapTbl[OSRdyTbl[y]];  
prio = (y << 3) + x;
```

Determine Y position in **OSRdyTbl[]**

Determine X position in **OSRdyTbl[Y]**

- 동작 방식

- OSRdyGrp과 OSRdyTbl의 현재 값을 이용해서 계산하면, 최우선 순위를 구할 수 있음.
- 이러한 계산 소요시간은, ready상태인 task의 개수와는 상관없음.
- 따라서, constant scheduling time을 구현할 수 있음.

# Ready list에서 최우선순위 찾기

- 최우선 순위를 찾아내는 방법

```
y = OSUnMapTbl[OSRdyGrp];  
x = OSUnMapTbl[OSRdyTbl[y]];  
prio = (y << 3) + x;
```

Determine Y position in **OSRdyTbl[]**

Determine X position in **OSRdyTbl[Y]**

- 우선순위 22와 29가 ready상태라고 하면,
  - OSRdyGrp의 값은 0000\_1100 (=12)임.
  - OSUnMapTbl[12]의 값은 2임.
  - 따라서 y값은 2가 됨.
  - OSRdyTbl[2]의 값은 0100\_0000 (= 64) 임.
  - OSUnMapTbl[64]의 값은 6임.
  - 따라서, x값은 6이 됨.
  - 결론적으로  $prio = (2 << 3) + 6 = (2 * 8) + 6 = 22$
  - 즉, 최우선 순위는 22임.

# OS Scheduling 정리

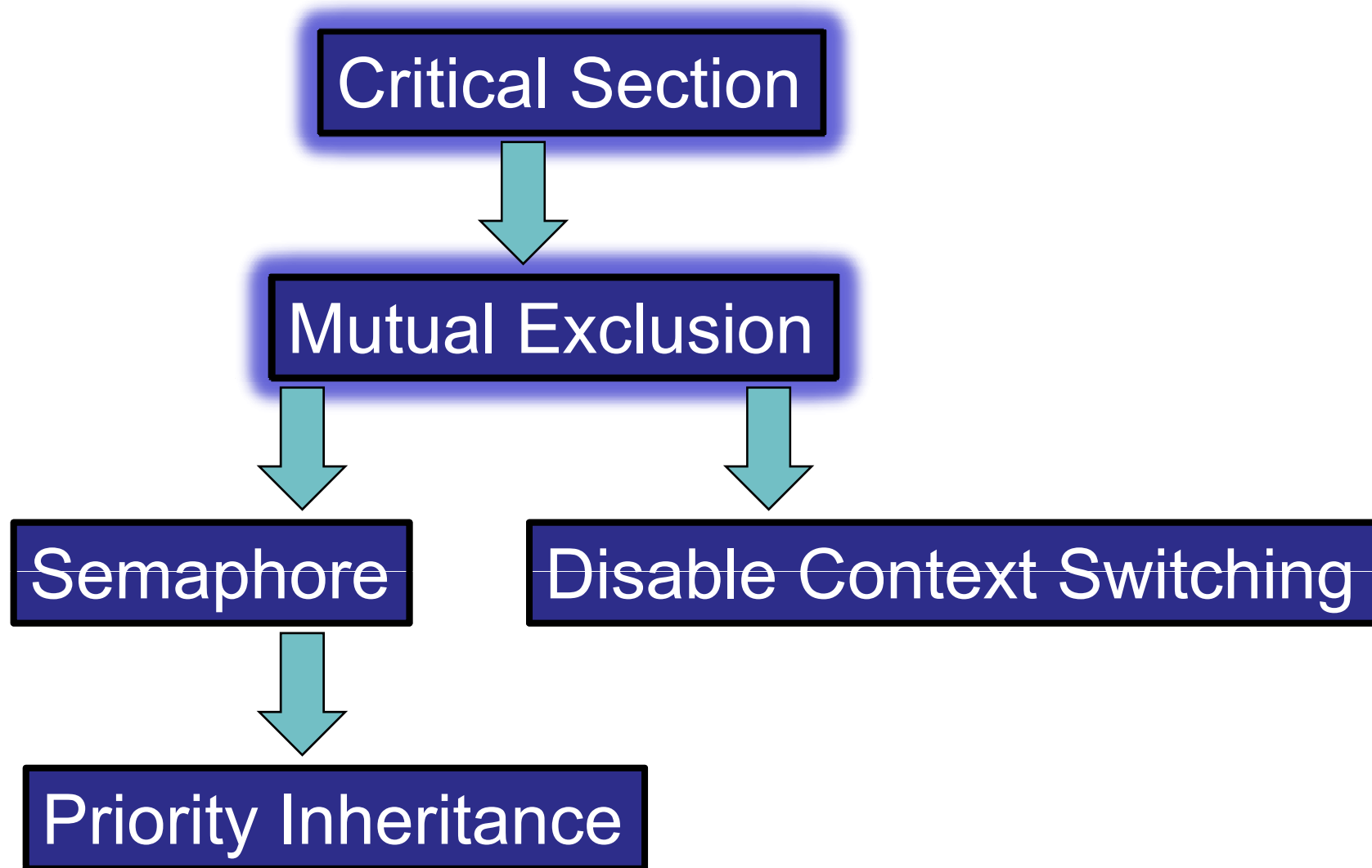
- RTOS에서는 task를 스케줄링할 때,
  - ready-queue 혹은 ready-list에서
  - ready상태의 task들 중 하나를 선택한다.
- Ready queue/list라는 용어 때문에, 추상적으로 큐 형식의 대기열을 상상할 수 있지만,
- 실시간 스케줄링의 요구사항을 만족시키기 위해서는
- 큐 형식이 아닌, table mapping방식을 사용해야 한다.
  - 즉, ready상태의 task 개수에 상관없이 항상 일정한 시간이 소요되는 스케줄링 구현이 필요
  - 특히, 스케줄링은 tick마다 실행되므로, “짧은 시간에 일정하게” 라는 요구 조건을 맞춰야 한다.

---

# **RTOS: CRITICAL SECTION**

# RTOS: Critical Section

---



# Critical Section(Region)

---

- 다른 task에 의해서 중단되어선 안 되는 코드 블록
- 이 블록의 코드를 시작하게 되면, 코드 블록의 종료까지 Context Switch 에 의한 다른 task의 수행이 없어야 함
- Requirements of Possible Solutions
  - Mutual Exclusion: only one process
  - Progress: able to decide who will enter
  - Bounded Waiting: able to enter in some time
- Solution e.g.
  - Semaphore

# Mutual Exclusion

- 하나의 task가 공유자원을 사용하고 있는 동안 다른 task가 이 자원을 사용하지 못하도록 보장
- Mutual Exclusion을 보장하기 위한 방법
  - 공유자원을 사용하는 동안 **인터럽트를 금지시킴?**

```
Disable interrupts;  
    Access(read/write) the shared resource;  
Enable interrupts;
```

- 매우 간단
- 인터럽트 금지 시간이 길어지면 문제 발생(**인터럽트를 잊어버림**)
  - for-loop, printf 금지
- 되도록 빠른 시간 안에 다시 인터럽트를 enable해야 함

# Mutual Exclusion(2)

## 공유자원을 사용하는 동안 Scheduling을 금지

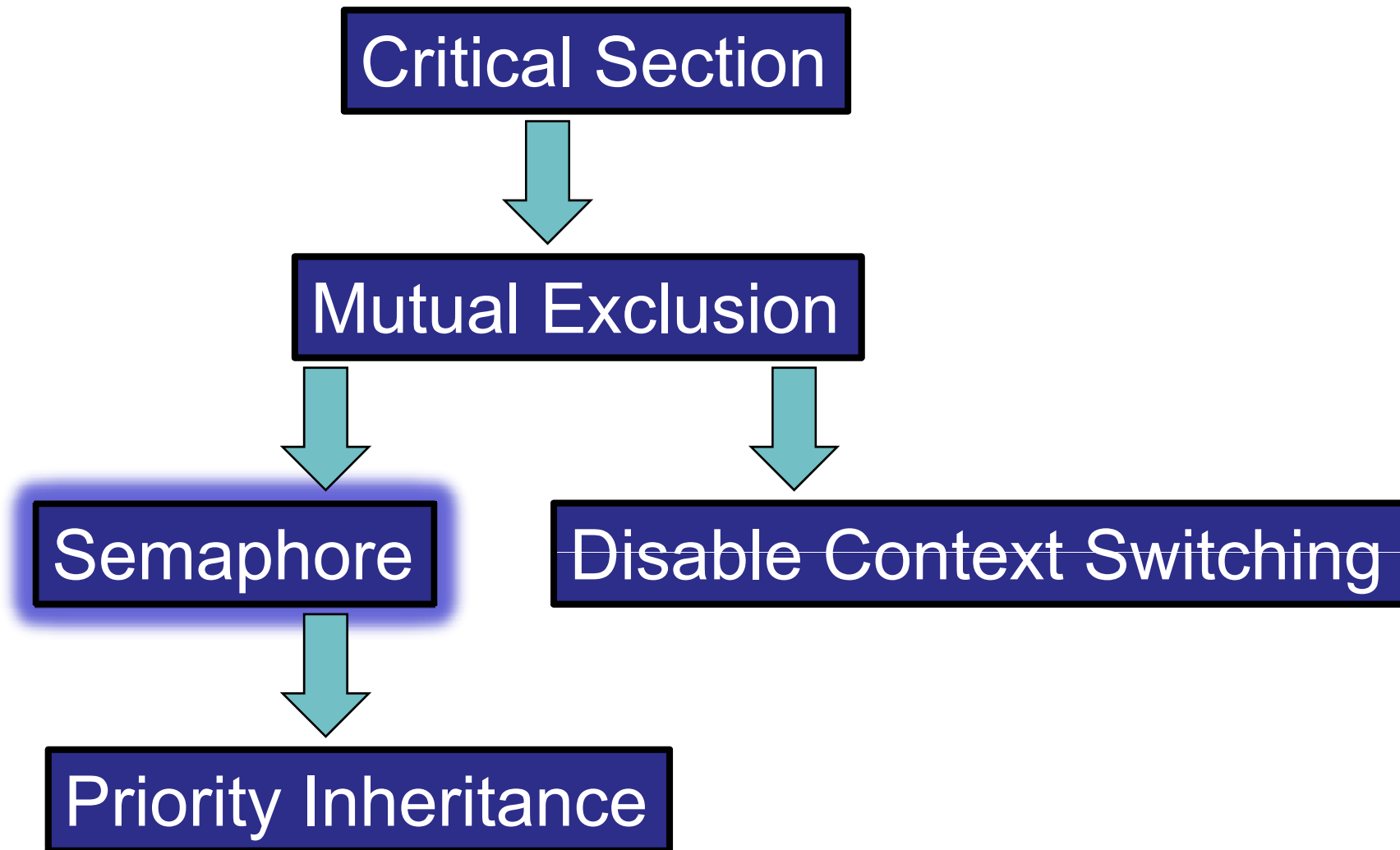
```
Disable scheduling;  
Access(read/write) the shared resource;  
Enable scheduling;
```

- Scheduling만을 금지 하였기 때문에 **인터럽트가 발생할 수 있음**
- **ISR에서 공유자원 접근 시** Mutual Exclusion을 보장할 수 없음
- 따라서 task간에 공유되는 자원에 대해서만 사용됨
- 이 방법을 많이 사용하게 될 경우에, 높은 priority task가 CPU를 점유할 수 있는 시점이 지연되기 때문에 **deterministic의 특성이 파괴되는 문제가 발생 (이 문제는 언제든지 발생가능!!!)**
- Semaphore를 사용한다
  - 가장 좋은 방법
  - **공유 자원의 access time이 짧은 경우 위의 두 방법이 더 효과적**



# RTOS: Critical Section

---



# Semaphores

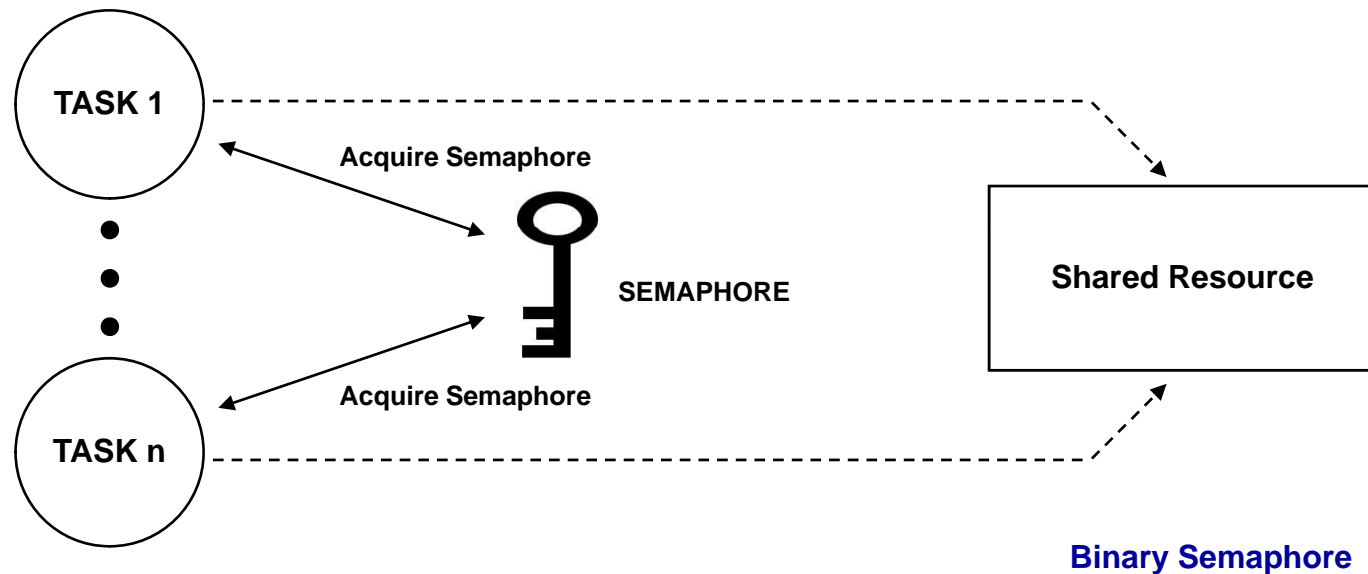
---

- task간 shared data problem 해결을 위해 OS에서 제공하는 장치
- 가장 간단한 형태는 binary semaphore
- OS 마다 비슷한 종류의 함수 제공
  - TakeSemaphore(); semTake(), pend()
  - ReleaseSemaphore(); semGive(), post()
- TakeSemaphore() 호출 후, semaphore가 없다면 block될 수 있음.
  - 이러한 block 기능이 shared data problem에 대한 해결책임.

# Semaphore

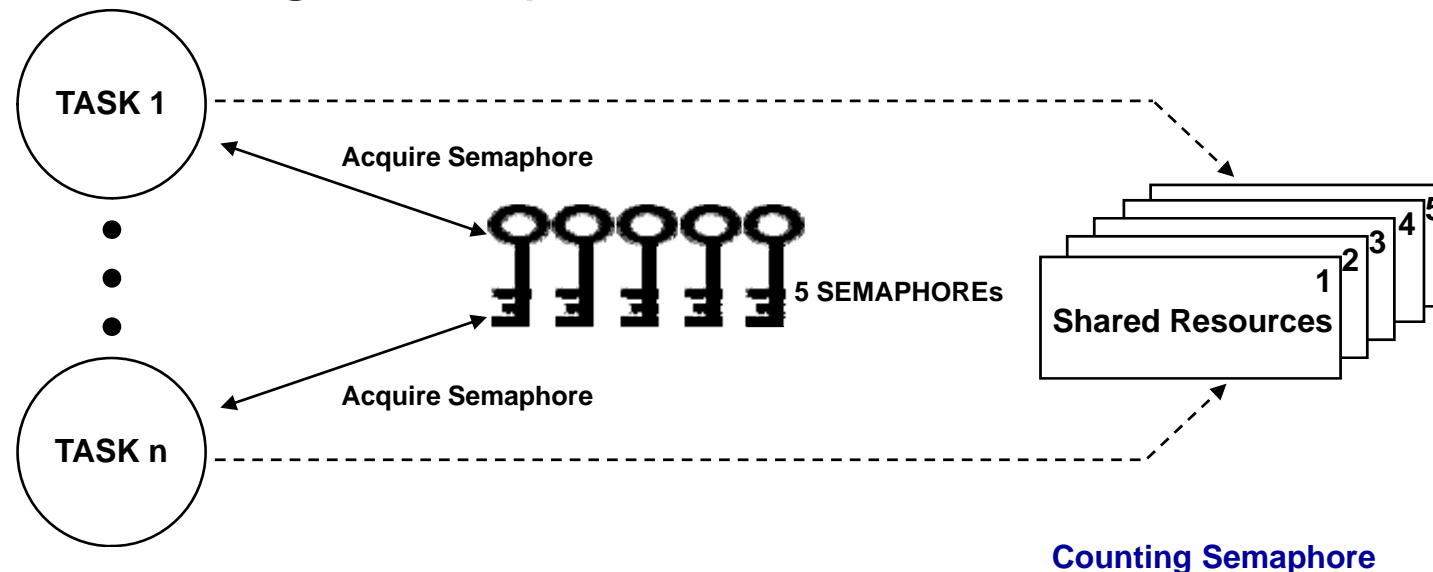
## □ Semaphore

- ❖ 1960년대 중반 Edgser Dijkstra에 의해서 고안
- ❖ 대부분의 RTOS에서 지원
- ❖ 공유자원을 액세스하기 위해서는 **key**가 필요
- ❖ Binary Semaphore



# Semaphore(2)

## – Counting Semaphore



## – 반환된 Semaphore를 획득하기 위한 방법

- priority based - RTOS
- FIFO based – general-purpose OS

# Semaphores 응용

```
1 void Task1 (void)
2 {
3     .
4     .
5     .
6     vCountErrors (9);
7     .
8     .
9     .
10 }
11
12 void Task2 (void)
13 {
14     .
15     .
16     .
17     vCountErrors (11);
18     .
19     .
20     .
21 }
22
23 static int cErrors;
24 static NU_SEMAPHORE semErrors;
25
26 void vCountErrors (int cNewErrors)
27 {
28     NU_Obtain_Semaphore (&semErrors, NU_SUSPEND);
29     cErrors += cNewErrors;
30     NU_Release_Semaphore (&semErrors);
31 }
```

Nucleus의 semaphore

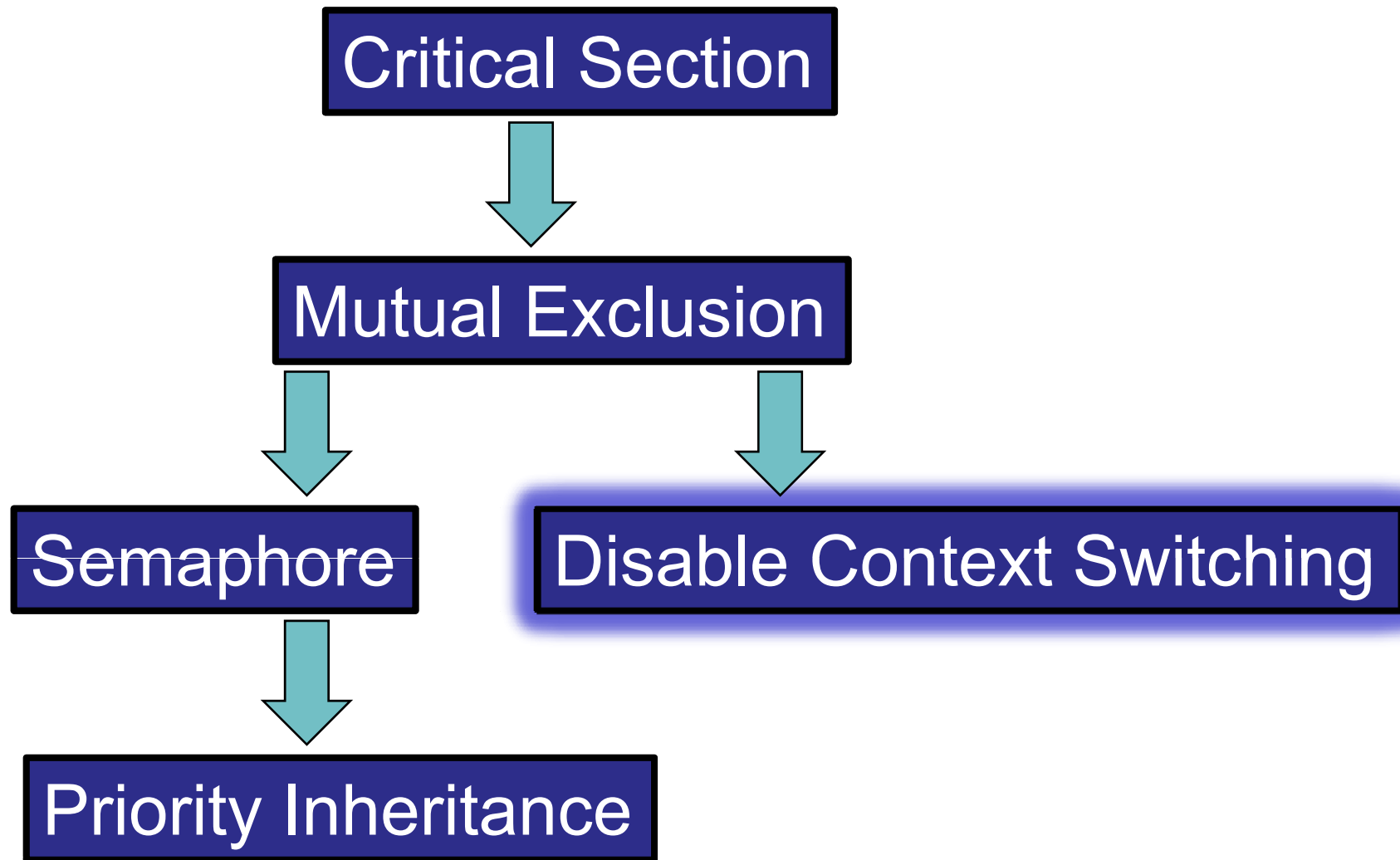
# Semaphore Variants

---

- Counting semaphore
  - take semaphore: 숫자 감소
  - release semaphore: 숫자 증가
  - Binary semaphore는 counting semaphore의 일종
- Resource semaphore
  - semaphore를 take한 task에 의해서만 release가 가능
- Mutex semaphore
  - Priority inversion 문제를 해결

# RTOS: Critical Section

---



# Shared Data Problem 해결책

---

- Disable/Enable Interrupt
- Semaphore
- 새로운 방법
  - disable task switching
    - task간 shared data problem 문제일 경우



# Shared Data Problem 해결책 비교

---

- Disable/Enable Interrupt
  - 가장 효과적
  - 인터럽트와 태스크간 shared data problem 해결책
  - task switching도 막을 수 있음.
  - 장점
    - 빠르게 처리: one assembly inst. for disable/enable

# Shared Data Problem 해결책 비교

---

- Semaphore
  - data shared problem에 연관된 task들만 관련
  - interrupt routine에는 아무 영향없음
  - 단점
    - 인터럽트와 task간에는 사용하지 않는 것이 좋음

# Shared Data Problem 해결책 비교

---

- task switching 방지
  - 앞의 두 방법과 비교하여 중간 정도
  - interrupt와 task간 data shared problem에는 사용될 수 없음.
  - 다른 모든 task를 중지시키므로 위험할 가능성

---

# **RTOS: SEMAPHORE는 실제로 어떻게 구현되는가?**

# Semaphore Details

---

- uC/OS-II의 semaphore 구현 코드를 통해서 알아보자.
- uC/OS-II에서는 다음을 지원한다.
  - Semaphore: counting semaphore
  - Mutex: priority inheritance 지원
  - Message box
  - Message queue

# Semaphore Details

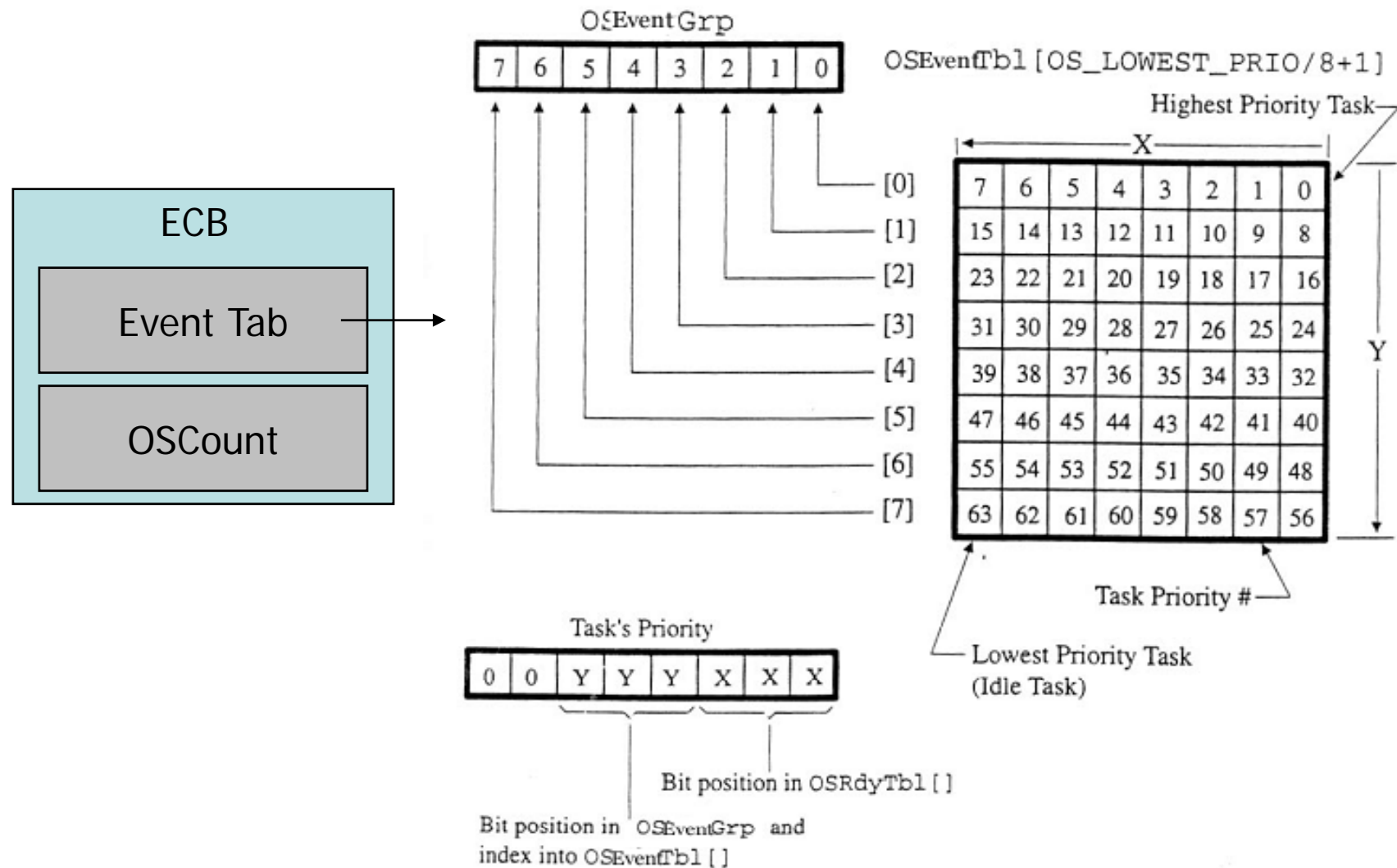
- 이러한 semaphore 등을 구현하기 위해서

- Event Control Block (ECB)을 사용
- ECB는 C로 구현된 하나의 구조체

```
struct OS_EVENT
{
    INT8U   OSEventType;
    INT16U  OSEventCnt;
    void    *OSEventPtr;
    INT8U   OSEventGrp;
    INT8U   OSEventTbl[];
}
```

- uC/OS-II에서는 semaphore등을 event라고 지칭함.
  - OSEventType: semaphore, mutex, msg box/queue를 지정
  - OSEventCnt: semaphore 구현에 사용
  - OSEventPtr: msg box/queue 구현에 사용
  - OSEventGrp/OSEventTbl: waiting list 구현에 사용

# OSEventGrp과 OSEventTbl의 구현



# Semaphore Details

---

- Task가 semaphore를 take할 때 ,
  - $\text{cnt} > 0$ 이면 , cnt값을 1 감소시키고 진행
  - $\text{cnt} == 0$  이면 ,
    - OSEventGrp와 OSEventTbl[ ]의 해당 위치 bit을 1로 setting
    - TCB를 wait 상태로 바꾸고, context switch out
    - mutex일 경우, 현재 semaphore를 소유하고 있는 task의 우선순위가 자신보다 낮다면, 자신과 동일하게 상향 조정
- Task가 semaphore를 give할 때 ,
  - cnt값을 1 증가시키고,
  - OSEventGrp값이 0이 아니면 ,
  - OSEventGrp과 OSEventTbl[ ]을 이용해서
  - 최우선순위를 찾아내서, 해당 task를
  - ready상태로 전환



# Semaphore의 또다른 용도

---

- task간 동기화를 위한 목적으로 사용
  - 두 개 이상의 task가 서로 간에 작업진척을 맞추어야 할 경우
  - 사용 예.
    - Task A는 자신의 일을 처리한 후 Task B가 나머지 일을 처리할 때까지 기다리기 위해, TakeSemaphore()를 호출
    - Task B는 작업 종료 후 Task A에게 알리기 위해 ReleaseSemaphore()를 호출

# Task Synchronization

---

- Task A는 변수 N을 1부터 2000까지 1씩 증가
- Task B는 변수 N이 증가할 때 마다 출력
- Task A와 B간의 작업 동기화를 맞추려면?

# Task Synchronization

```
int N = 0;

void taskA(void) /* task A */
{
    int i;
    for (i = 1; i <= 2000; i++)
        N++;
}

void taskB(void) /* taskB */
{
    int i;
    for (i = 1; i <= 2000; i++)
        printf("N is %d\n", N);
}
```

**두 개의 task를 동시에 돌리면**

- N이 1씩 증가되지 않으며**
- 같은 값의 N이 여러 번 중복해서 출력**

# Task Synchronization

```
int N = 0;

void taskA(void) /* task A */
{
    int i;
    for (i = 1; i <= 2000; i++)
    {
        N++;
        Sleep(1);
    }
}

void taskB(void) /* taskB */
{
    int i;
    for (i = 1; i <= 2000; i++)
    {
        printf("N is %d\n", N);
        Sleep(1);
    }
}
```

**두 개의 task를 동시에 돌리면**

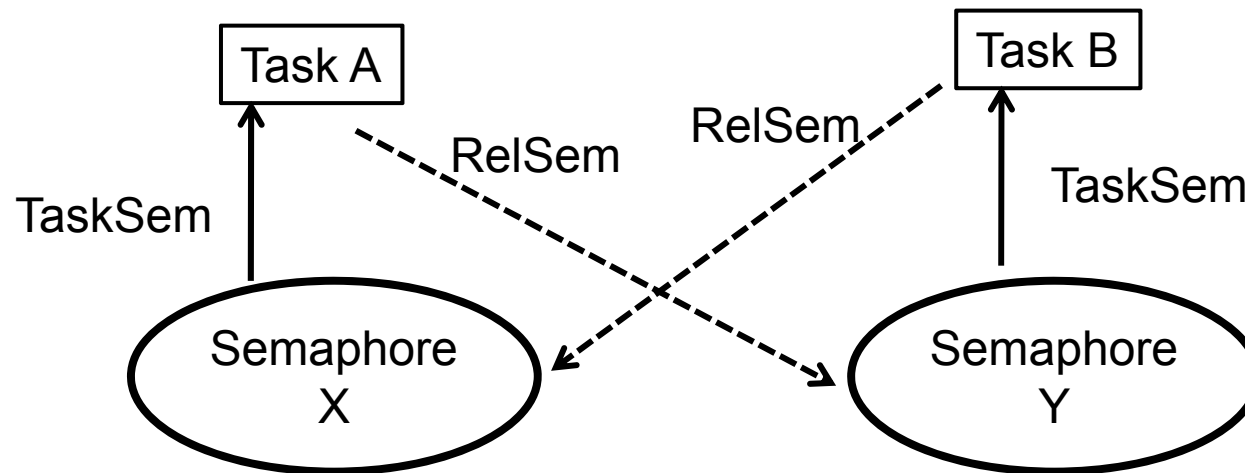
**-N이 1씩 증가하며,  
-중복없이 출력**

**이 방식의 문제점은?**

**- Sleep()으로 인한 낭비시간 발생**

# Task Synchronization

## Semaphore를 사용하는 방법



# Task Synchronization

```
int  N = 0;

/*  semaphore X count = 0  */
/*  semaphore Y count = 1  */

void taskA(void) /* taskA */
{
    int  i;
    for (i = 1; i <= 2000; i++) {
        Take semaphoreX; /* attempt to get semaphore X */
        N++;
        Give semaphoreY; /* release semaphore Y */
    }
}

void taskB(void) /* taskB */
{
    int  i;
    for (i = 1; i <= 2000; i++) {
        Take semaphoreY; /* attempt to get semaphore Y */
        printf("N is %d\n", N);
        Give semaphoreX; /* release semaphore X */
    }
}
```

두 개의 semaphore를 이용

# Task Synchronization

```
int  N = 0;

/*  semaphore X count = 1  */

void taskA(void) /* taskA */
{
    int  i;
    for (i = 1; i <= 2000; i++) {
        Take semaphoreX;
        N++;
        Give semaphoreX;
    }
}

void taskB(void) /* taskB */
{
    int  i;
    for (i = 1; i <= 2000; i++) {
        Take semaphoreX; /* attempt to get semaphore Y */
        printf("N is %d\n", N);
        Give semaphoreX; /* release semaphore X */
    }
}
```

**한 개의 semaphore를 이용할 때의 문제점은?**

**두 조건이 만족해야만 정상작동**

**-두 task가 동시에 시작해야 한다.**

**- 동일 N값 중복 출력**

**- 초기 N값 출력안됨**

**-두 task가 같은 priority를 가져야 한다.**

**- priority-sema의 경우, 높은 우선순위만 계속 수행**

# Semaphore 사용시 문제점

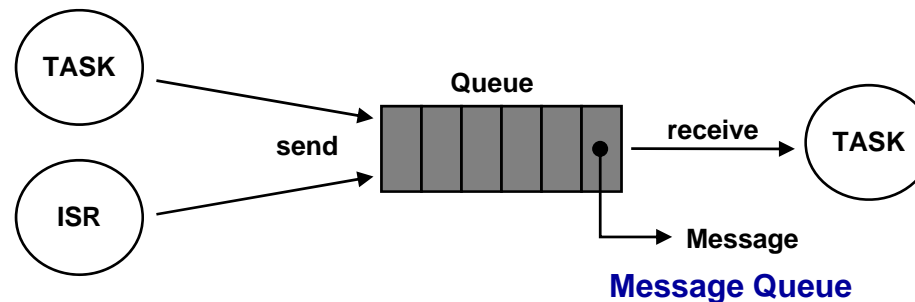
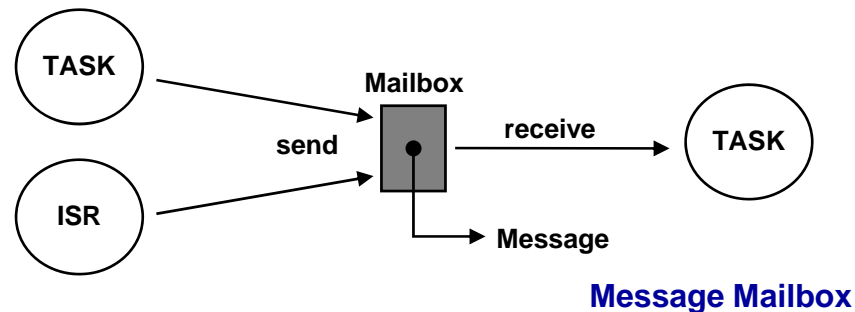
---

- 간단한 프로그램상의 문제점
  - TakeSemaphore 생략
  - ReleaseSemaphore 생략
  - 다른 semaphore를 TakeSemaphore
  - 너무 긴 시간동안의 TakeSemaphore
    - 특히, interrupt와 semaphore 공유시 주의
      - Interrupt 내에서는 Semaphore를 사용하지 말아야!!



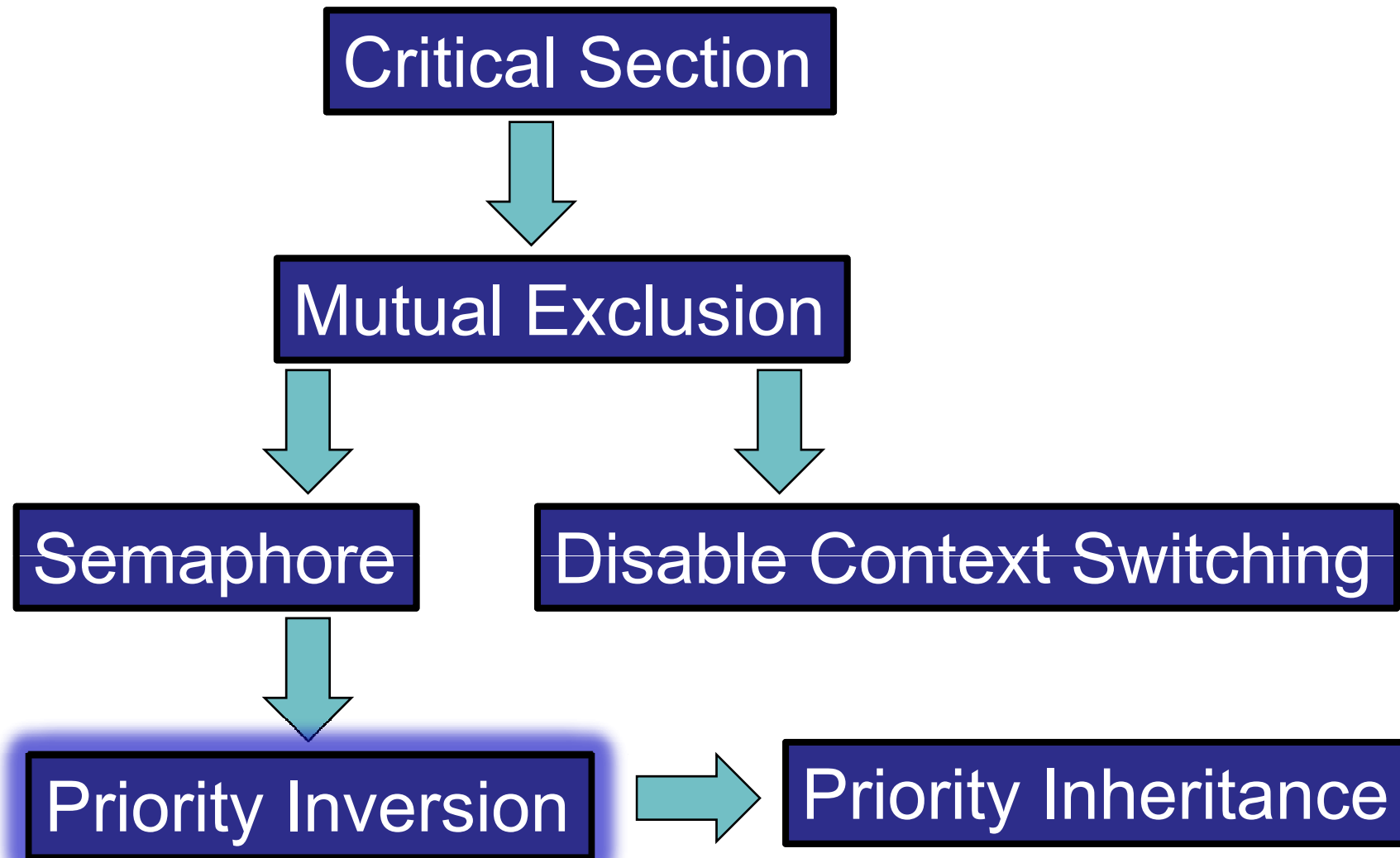
# Inter-task Communication

- global variable을 사용해야 하는 경우
  - Linked list, Circular queue ..
  - Mutual Exclusion이 보장되어야 함
- message passing
  - Message Mailbox
    - 하나의 메시지
  - Message Queue
    - 복수 개의 메시지



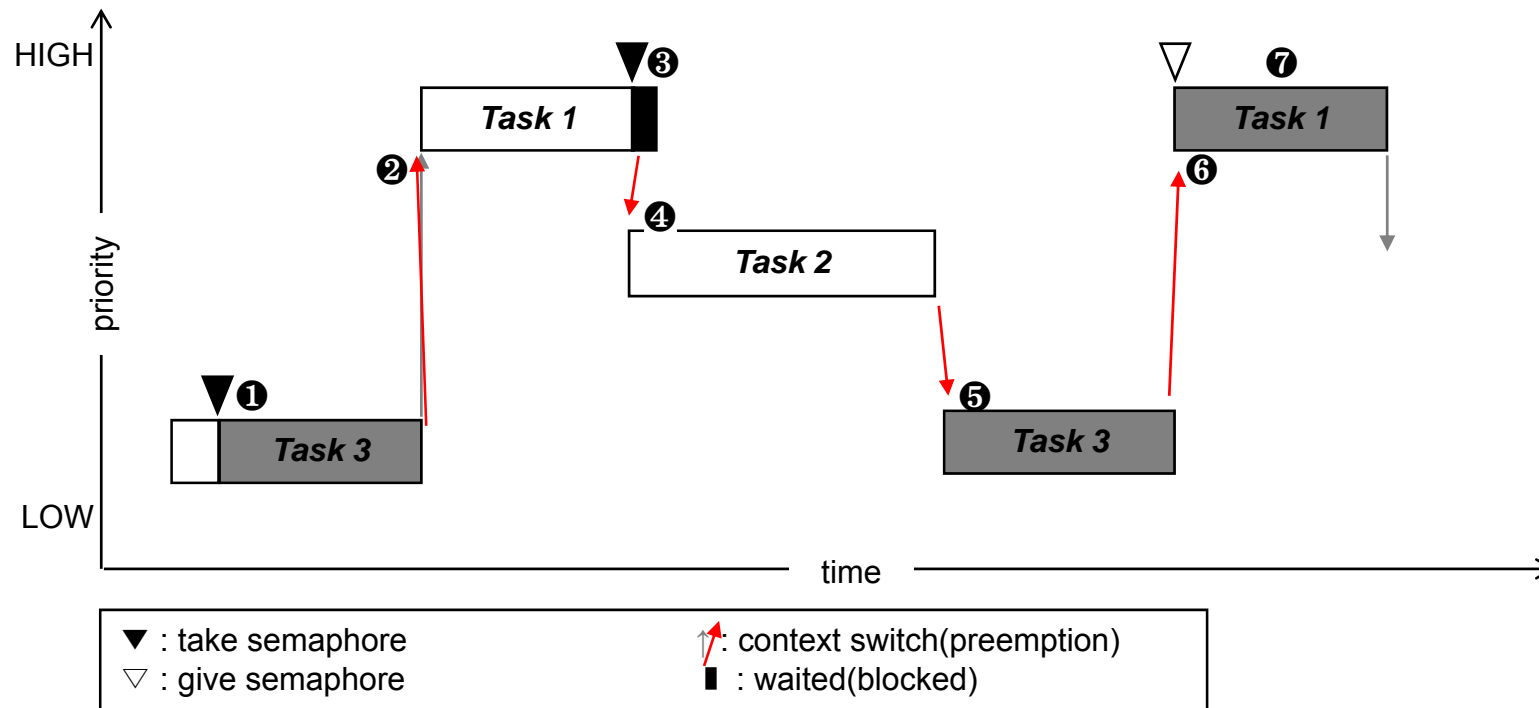
# RTOS: Critical Section

---

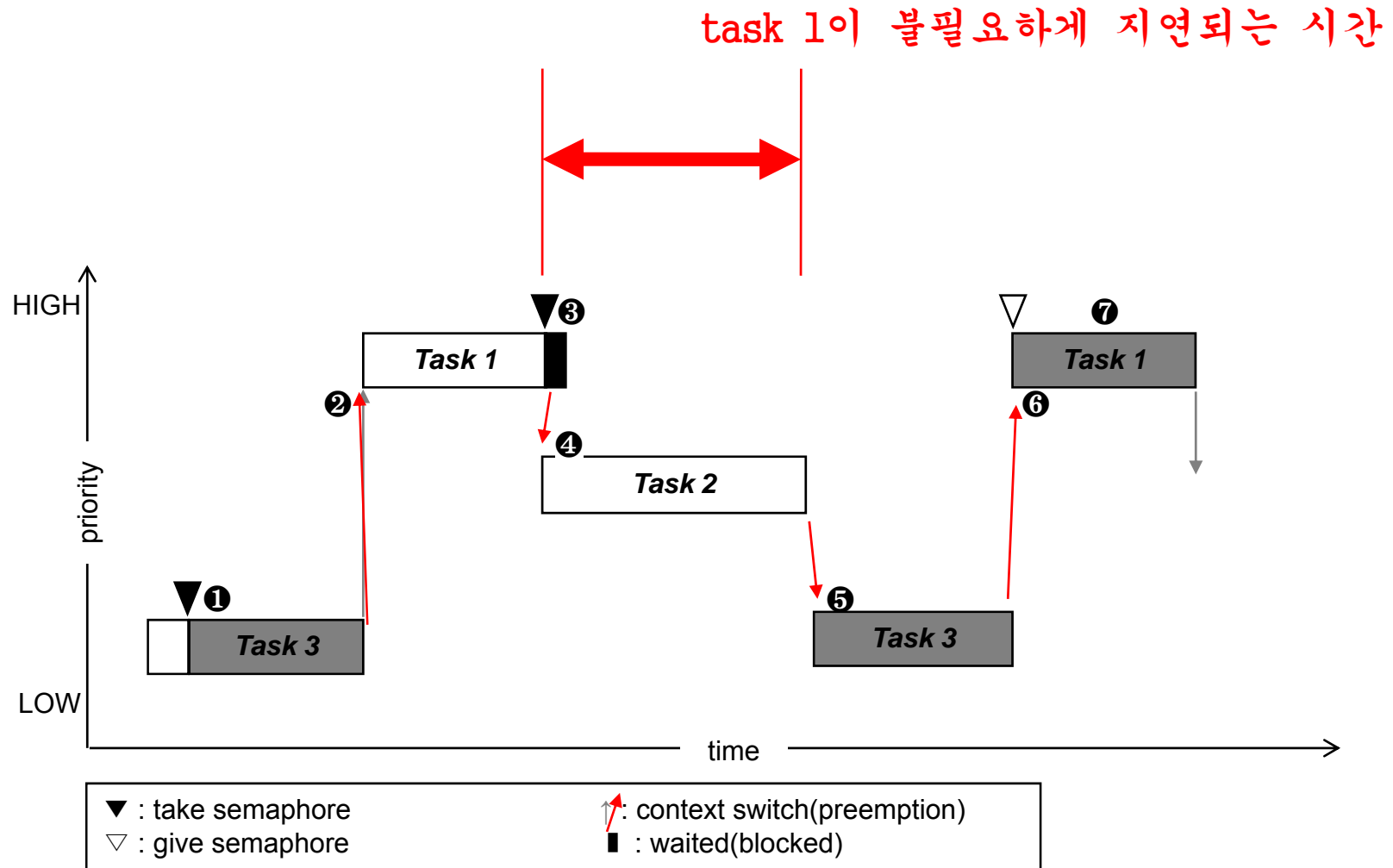


# Semaphore의 문제점:

## Priority Inversion



# Priority Inversion



## Mars Pathfinder

- Launched 1996
- Lander plus Sojourner Rover
- Successful mission: landed July 1997
- See
  - <http://mars.jpl.nasa.gov/missions/past/pathfinder.html>
  - [http://research.microsoft.com/~mbj/Mars\\_Pathfinder/Authoritative\\_Account.html](http://research.microsoft.com/~mbj/Mars_Pathfinder/Authoritative_Account.html)



Dr. Hugh Melvin, Dept. of IT, NUI,G

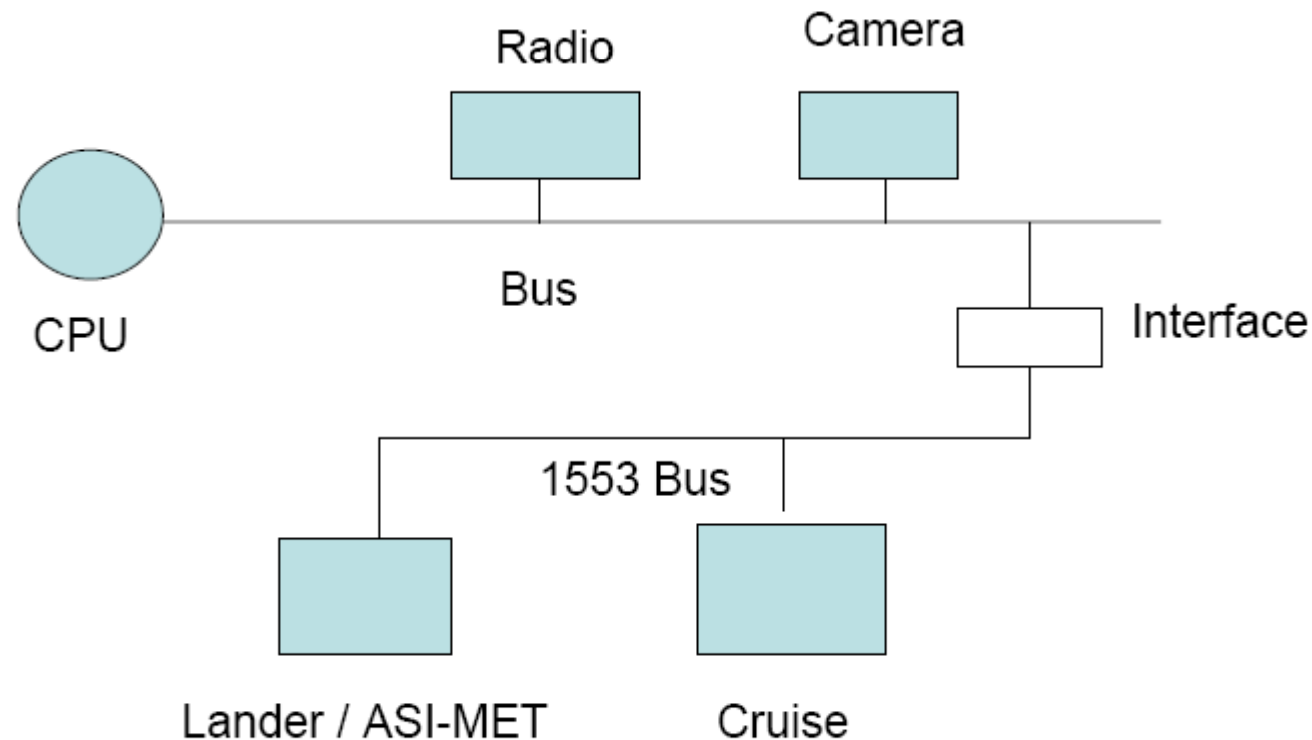
# Priority Inversion

---

- VxWorks RTOS engine on PathFinder
  - See <http://www.windriver.com/portal/server.pt>
- Widely Used RTOS
  - See articles
  - POSIX.4 Compliant
    - Boeing 787
    - Router/Switches
    - Mars Pathfinder!
- Pathfinder suffered repeated resets on the surface

Dr. Hugh Melvin, Dept. of IT, NUI,G

## Pathfinder Architecture



Dr. Hugh Melvin, Dept. of IT, NUI,G

# Priority Inversion

---

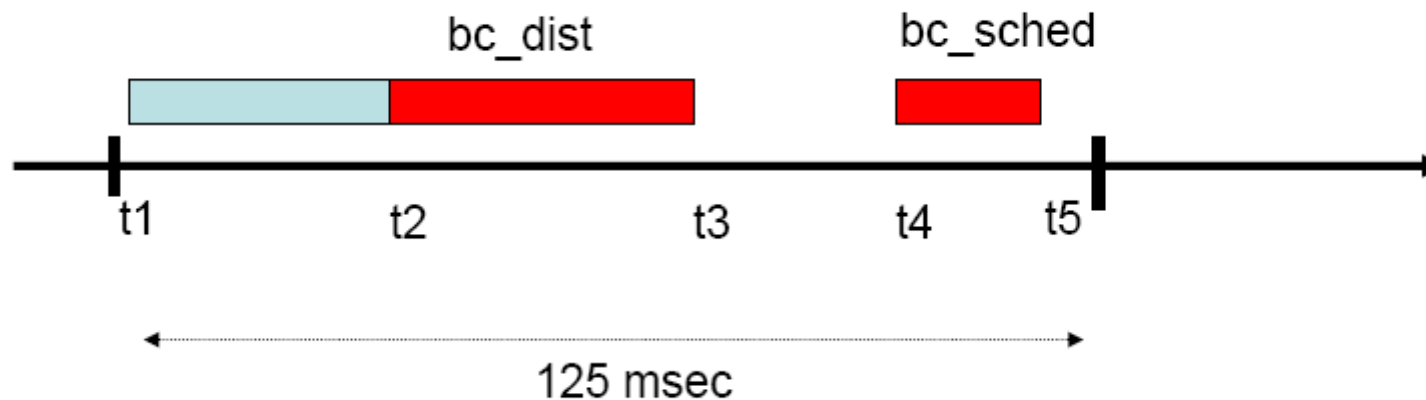
- 1553 Bus controlled by 2 main tasks
  - Bus Scheduler: `bc_sched` task
    - Highest priority
  - Data Collection : `bc_dist`
    - 3<sup>rd</sup> Highest priority
  - Other device tasks
    - Medium priority
  - ASI-MET : Meteorological Data Task
    - Lower priority
- Cycle time 125 msec
  - System designed to reset if `bc_dist` has not completed when `bc_sched` released at  $t_4$  = hard deadline

Dr. Hugh Melvin, Dept. of IT, NUI,G



# Priority Inversion & Priority Inheritance

- 1553 Bus controlled by 2 main tasks
  - t1: Bus hardware tasks run
  - t2 Hardware tasks complete
    - bc\_dist runs and distributes data
  - t3: bc\_dist complete
  - t4: bc\_sched runs setting transactions for next cycle
  - Some slack in cycle



Dr. Hugh Melvin, Dept. of IT, NUI, G

## Problem

- ASI-MET task and `bc_dist` shared a resource
  - IPC mechanism to transfer data
- Fault Sequence
  - ASI-MET acquired semaphore
  - Medium priority tasks preempted ASI-MET
  - `bc_dist` released and preempts medium priority tasks
  - Attempts to lock semaphore but already locked → blocks
  - Medium priority tasks resume
  - At `t4`, `bc_sched` released .. Determines that `bc_dist` has not completed → Forces system reset
  - → **Classic Priority Inversion Problem**

Dr. Hugh Melvin, Dept. of IT, NUI,G

# Priority Inversion & Priority Inheritance

---



By **Kaiwan N Billimoria** on Fri, 2004-02-13 02:00. [Linux Journal](#)

Could OSS have saved the '97 Mars Pathfinder mission some headaches?

---

I recently read some interesting e-mails discussing the 1997 Mars Pathfinder mission. Yes, it's an old topic now; nonetheless, these e-mails are interesting reads.

**The first message**, written by Mike Jones, describes the so-called software glitch that occurred on the Mars Pathfinder vehicle while it was on the surface of Mars and how the problem was solved. **An authoritative follow-up** message was penned by Glenn E Reeves, the software team leader from JPL for the Mars Pathfinder mission.

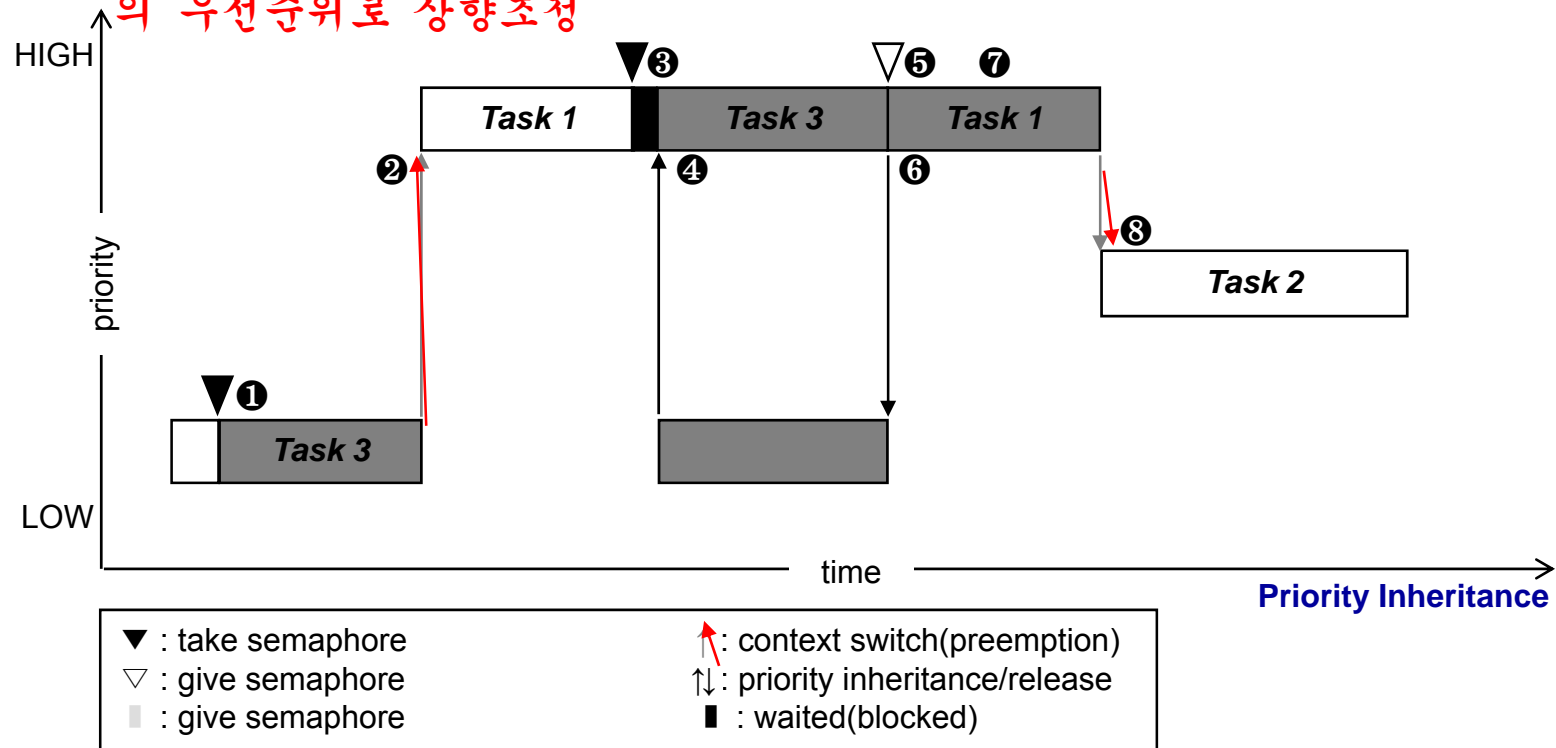
To briefly summarize, a software priority inversion problem caused constant resets of the spacecraft's software. The problem finally was tracked down and solved by JPL's engineers with support from the software provider, Wind River. Yes, it ran VxWorks.

At first glance, this dialogue is merely interesting; I think every hardware and software engineer/tinkerer should read them. On deeper reflection, however, I was struck by something more. Although I assume it was not their intention, the authors quite clearly demonstrate how open-source software (OSS) and the OSS development model would have helped this project enormously, not only in finding the bug but, in all probability, preventing the bug in the first place. The extracts from these e-mails and my comments below should make more sense to you after you've read the original postings. In his well thought-out reply, Glenn makes the following statements:

# 해결책: Priority Inheritance

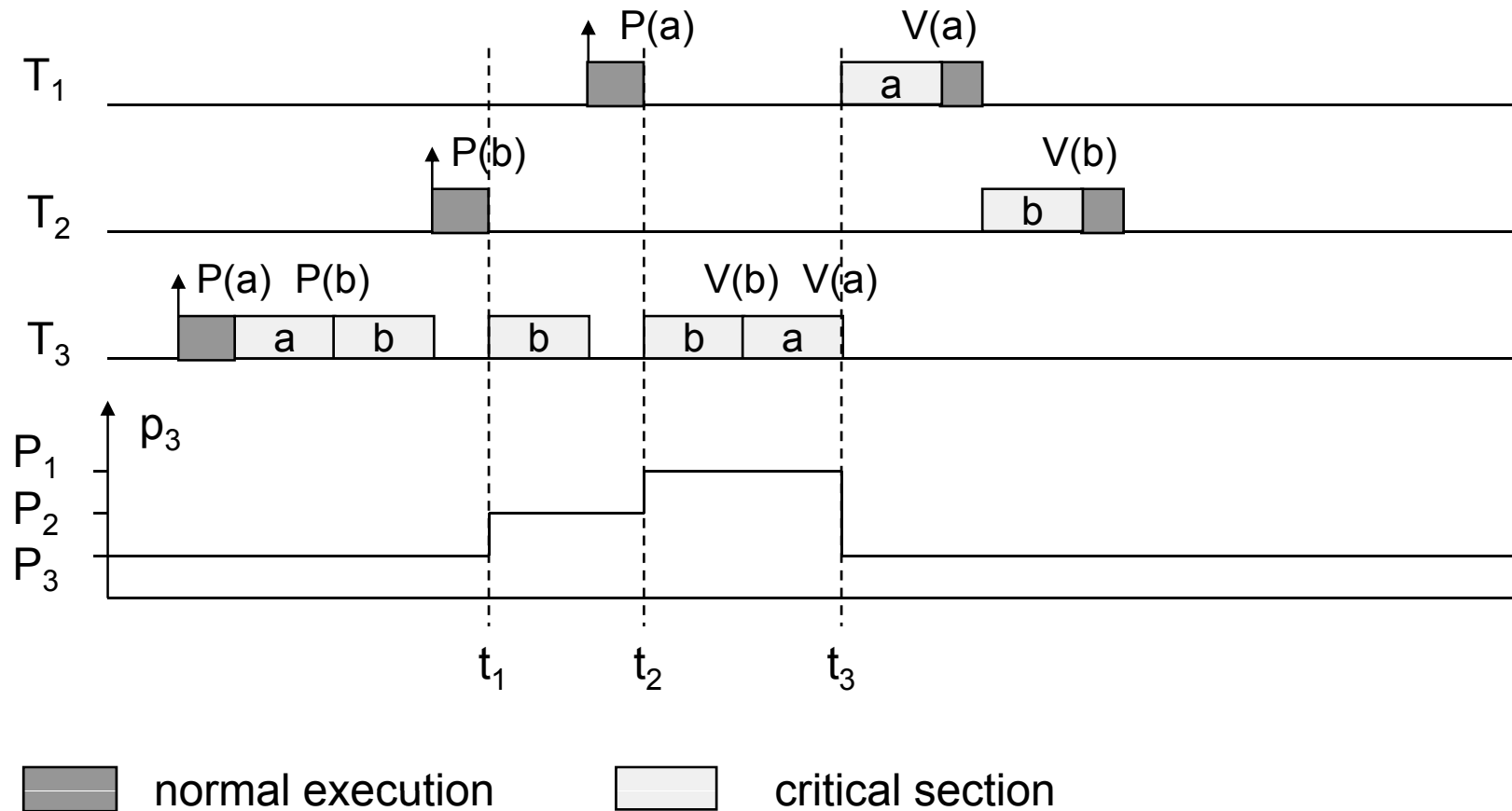
- Priority Inheritance

- 낮은 우선순위 태스크 A가 semaphore를 잡고 있고, 그 semaphore를 높은 우선순위 태스크 B가 waiting하면, 태스크 A의 우선순위를 태스크 B와 같도록 일시적으로 조정
- semaphore를 가진 태스크의 우선순위를 그 semaphore를 waiting하는 태스크의 우선순위로 상향조정



# Priority Inheritance Example

두 개 이상의 semaphore에 의한 priority inheritance



# Priority Inheritance Example

T1은 T2를 기다리고, T2는 T3를 기다리므로,  
T3의 우선순위를 T1급으로 상향 조정한다.

