Semaphores

동기화

- **동기화의 필요성**
  - 멀티태스킹 환경의 병렬성(parallelism) 활용.
    - ➔ 처리율, 응답 시간 등의 성능 향상.
  - 공유 자원(shared resource)
    - 시스템 자원의 대부분이 공유될 수 있음.
    - 동시 접근으로부터 보호되어야 함.
  - 경쟁 조건(race condition)의 발생.
    - 하나 이상의 태스크가 동일한 자원을 접근하기를 원함.

동기화

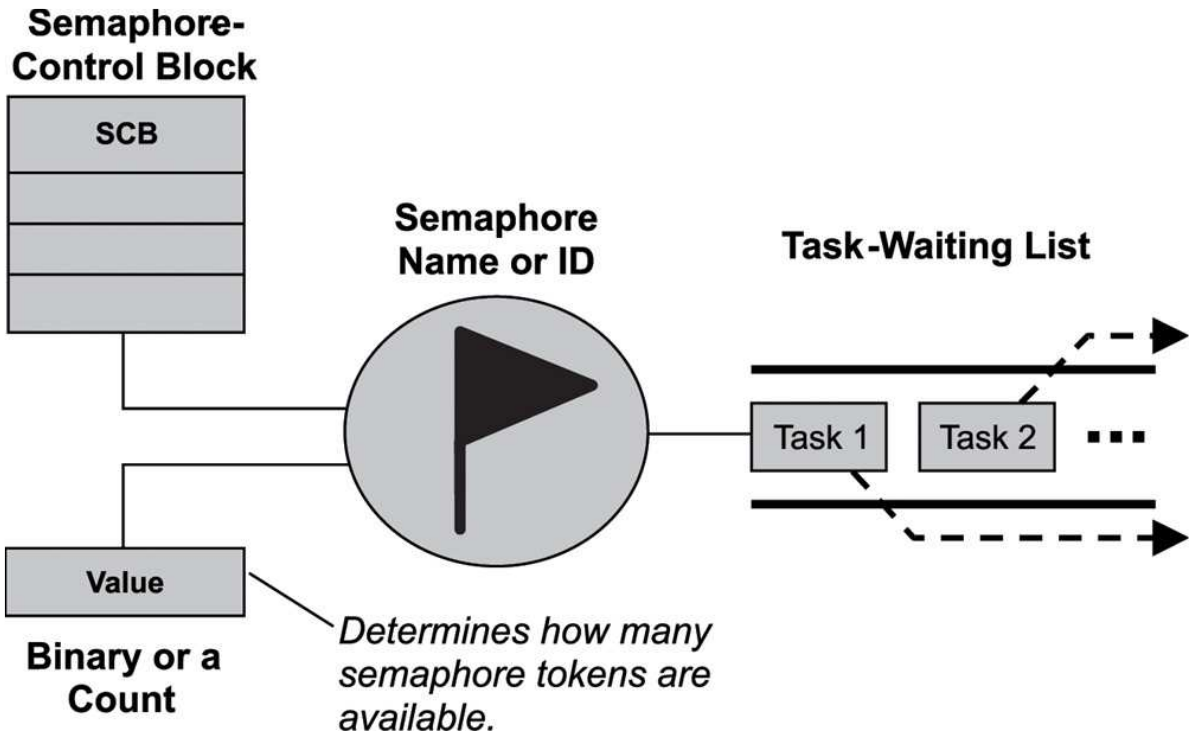
- **동기화 방법**
  - 상호 배제(mutual exclusion) 메커니즘의 사용.
    - 한 시점에 하나의 태스크만 공유 자원을 접근할 수 있음.
  - 임계 구역(critical section)을 정의.
    - 공유 자원을 접근하기 위한 코드의 일부.
  - 대표적인 동기화 방법.
    - 세마포어, 뮤텍스, 파이프 등.

Introduction

- **Multiple concurrent tasks must be able to**
  - Synchronize their execution.
  - Coordinate mutually exclusive access to shared resources.
- **RTOS kernels provide**
  - Semaphore object
  - Associated semaphore management services.
- **Semaphore (semaphore token)**
  - A kernel objects that one or more tasks can acquire or release for the purposes of synchronization or mutual exclusion.

Defining semaphores

- **Data structures on semaphore**
  - Semaphore control block (SCB)
  - A unique ID
  - A value (binary or count)
  - A task-waiting list



Defining semaphores

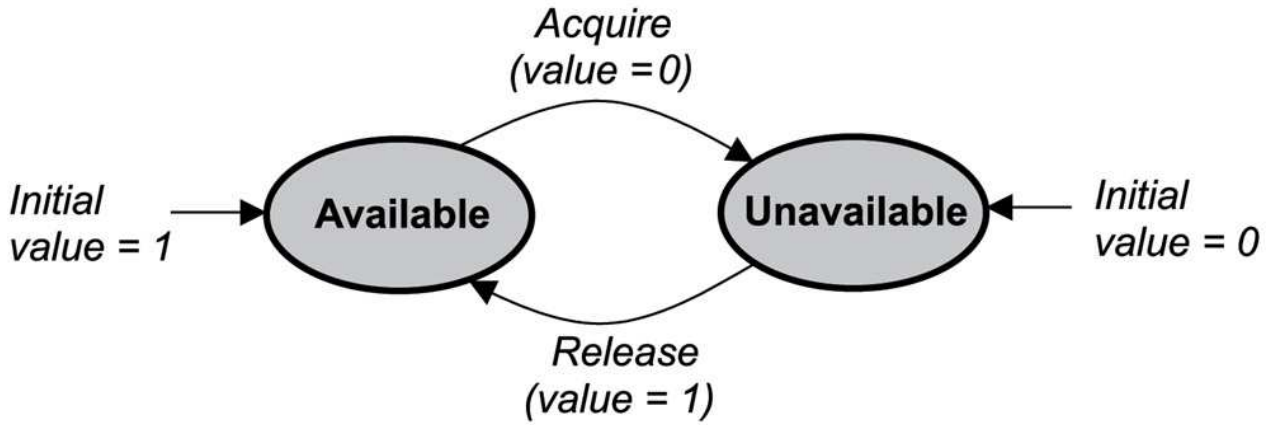
- **Operations on semaphore**
  - Kernel tracks the number of times that a semaphore has been acquired or released by maintaining a token count.
    - When a task acquires the semaphore, token count is decremented.
    - When a task releases the semaphore, token count is incremented.
  - If the token count reaches 0, a requesting task blocks.
    - Task-waiting list tracks all tasks blocked while waiting on an unavailable semaphore.
      - with FIFO or highest priority first order.
    - When semaphore becomes available, the first task in the task-waiting list acquires it.

Defining semaphores

- **Kernel support many different types of semaphores**
  - Binary semaphore
  - Counting semaphore
  - Mutual-exclusion semaphore (mutex)

Binary semaphores

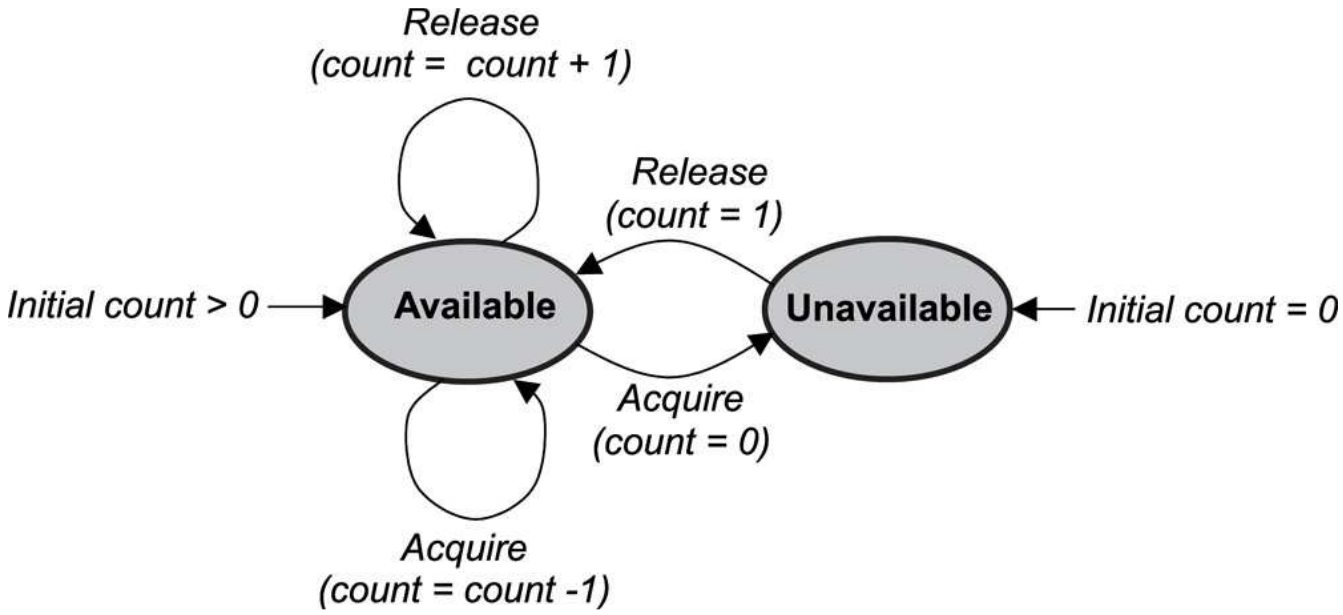
- **A binary semaphore can have a value of either 0 or 1.**
  - 0, unavailable (or empty)
  - 1, available (or full)



- **Binary semaphores are global resources.**
  - Even the task did not initially acquire semaphore can release it.

Counting semaphores

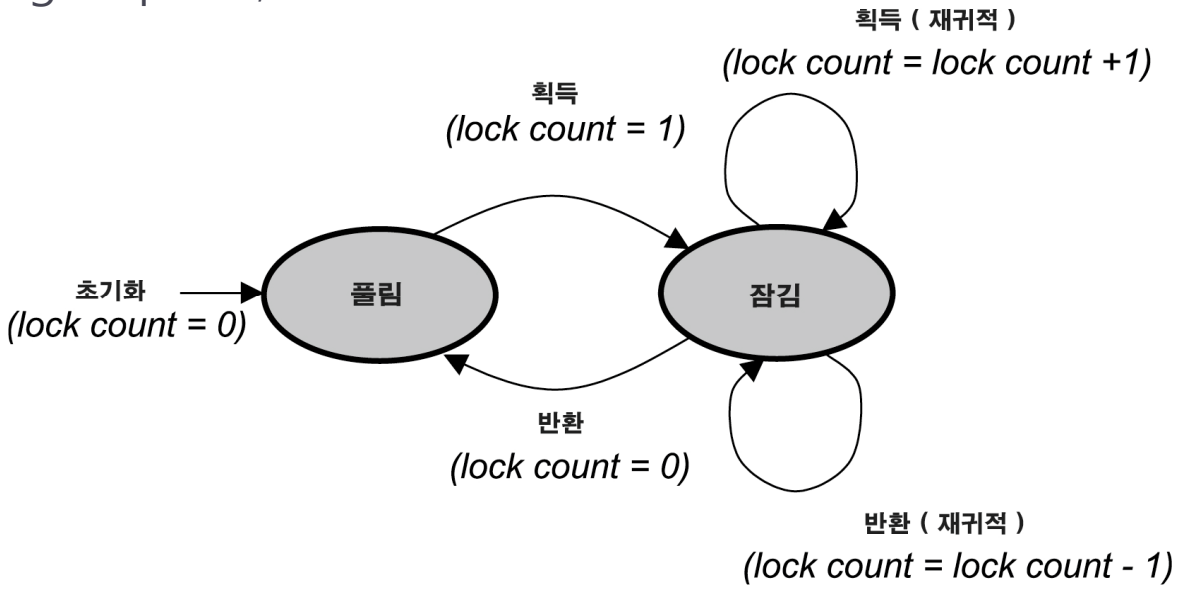
- **A counting semaphore uses a count to allow it to be acquired or released multiple times.**
  - If the initial count is
    - 0, is created in unavailable state.
    - > 0, is created in the available state.



- **Counting semaphores are also global resources.**

Mutual exclusion semaphores

- **A mutual exclusion(mutex) semaphore is a special binary semaphore.**
- **The states of a mutex are**
  - 0, unlocked
  - 1, locked
- **Operations on mutex**
  - A mutex is initially created in the unlocked state.
  - After being acquired, the mutex moves to the locked state.



Mutual exclusion semaphores

- **A mutual exclusion semaphore can support**
  - Ownership
  - Recursive locking
  - Task deletion safety
  - Priority inversion avoidance

Mutual exclusion semaphores

- **Ownership**
  - Ownership of a mutex is gained when a task locks the mutex.
  - Conversely, a task loses ownership of the mutex when unlocks it.

- **Compare with binary semaphore!**
  - Even a task that did not originally acquire the binary semaphore can release it.

Mutual exclusion semaphores

- **Recursive locking**
  - Task that already owns the mutex can acquire it multiple times.
  - The mutex with recursive locking is called a recursive mutex.

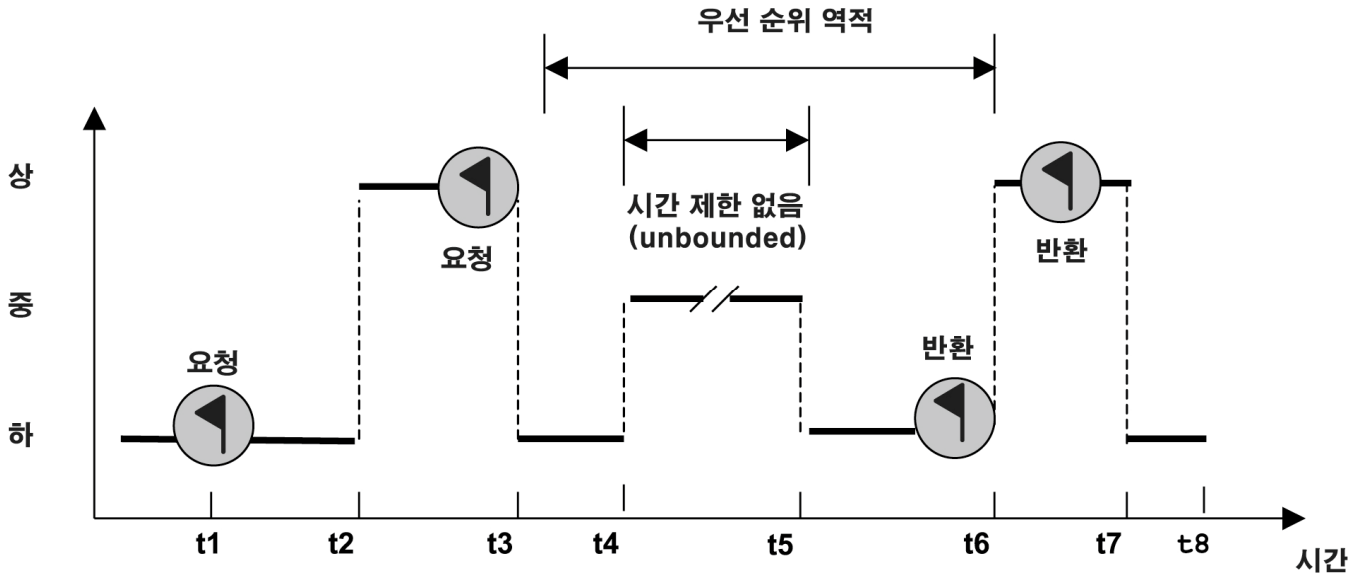
- **Count in mutex vs. count in counting semaphore**
  - The count in mutex
    - the number of times that task owning the mutex has locked or unlocked it.
  - The count in counting semaphore
    - the number of tokens that have been acquired or released by tasks.

Mutual exclusion semaphores

- **Task deletion safety**
  - Premature task deletion is avoided.
  - While a task owns the mutex, the task cannot be deleted by other tasks.

Mutual exclusion semaphores

- **Priority inversion avoidance**
  - Priority Inheritance protocol



Typical semaphore operations

- **Typical operations with semaphores**
  - Creating and deleting semaphores.
  - Acquiring and releasing semaphores.
  - Clearing a semaphore's task-waiting list.
  - Getting semaphore information.

Typical semaphore use

- **Semaphores are useful for**
  - the synchronized execution of multiple tasks
  - coordinating access to a shared resources.

- **Typical uses**
  - Wait-and-signal synchronization
  - Single shared-resource-access synchronization
  - Recursive shared-resource-access synchronization
  - Multiple shared-resource-access synchronization
  - …

Wait-and-signal synchronization

- **Two tasks can communicate for the purpose of synchronization.**
  - Binary semaphore is initially 0.
  - tWaitTask has higher priority and runs first, and blocked.
  - Lower priority tSignalTask has a chance to run and release semaphore.
  - tWaitTask is unblocked and preempts tSignalTask.

Wait-and-signal synchronization

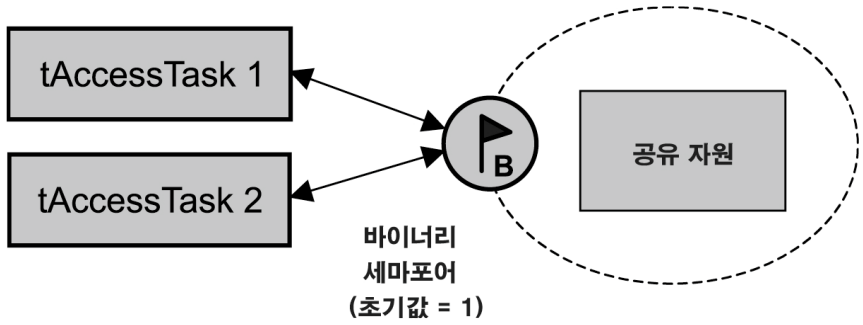
- **Pseudo code**

```
tWaitTask ( )                    ← higher priority
{
        :
        acquire binary semaphore token
        :
}

tSignalTask ()                   ← lower priority
{
        :
        release binary semaphore token
        :
}
```

Single shared-resource-access synchronization

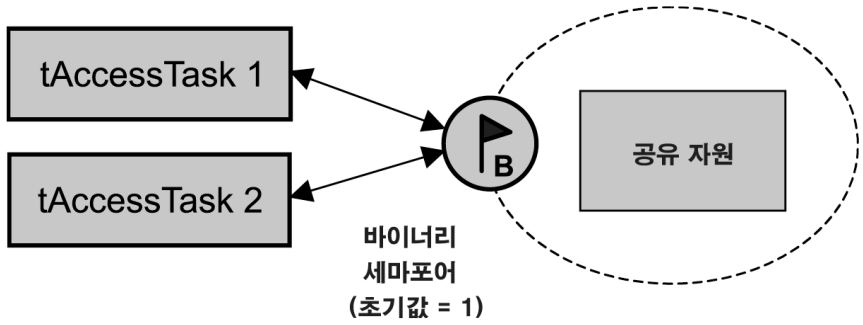
- **Mutually exclusive access to a shared resource**
  - A binary semaphore is initially 1.
  - To use the shared resource, task needs to acquire the binary semaphore.
  - Since binary semaphore is global, may cause problem.
  - ➔ use Mutex with ownership concept.



Single shared-resource-access synchronization

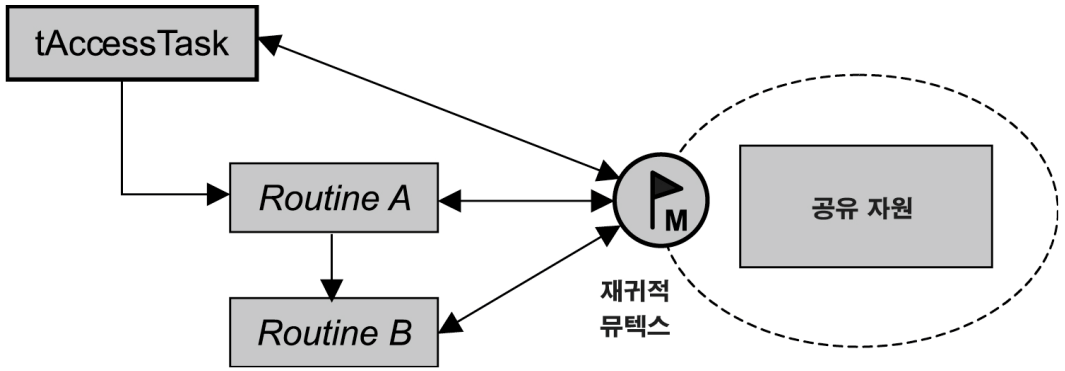
- **Pseudo code**

```
tAccessTask ( )
{
        :
        acquire binary semaphore token
        read or write to shared resource
        release binary semaphore token
        :
}
```



Recursive shared-resource-access synchronization

- **Tasks to access a shared resource recursively.**
  - "tAcessTask ➔ Routine A ➔ Routine B"
  - Assume that all three need access to the same shared resource.
  - Tasks would end up blocking, causing a "deadlock".
  - Use "recursive mutex"
    - After tAccessTask locks the mutex, the task owns it.
    - Additional attempts for locking the mutex from the task or routines will succeed.



Recursive shared-resource-access synchronization

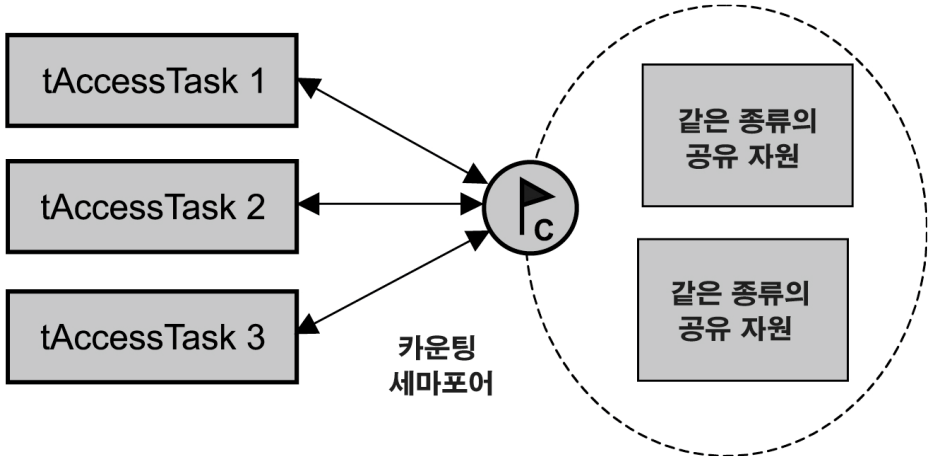
KWANGWOON UNIVERSITY

- **Pseudo code**

```
tAccessTask ( )
{
        :
        acquire mutex
        access shared resource
        call Routine A
        release mutex
        :
}
Routine A ( )
{
        :
        acquire mutex
        access shared resource
        call Routine B
        release mutex
        :
}
Routine B ( )
{
        :
        acquire mutex
        access shared resource
        release mutex
        :
}
```

Multiple shared-resource-access synchronization

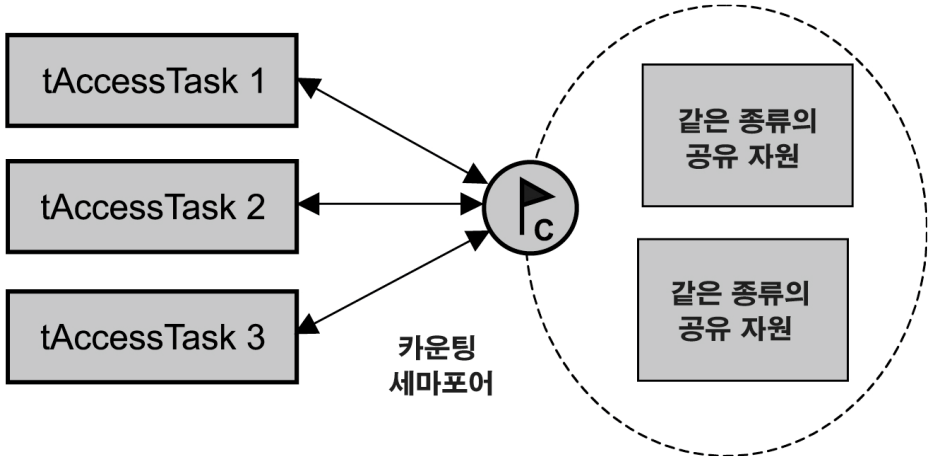
- **Counting semaphore for multiple equivalent shared resources**
  - The count is initially set to the number of shared resources.
  - Assume that the count is 2.
  - tAccessTask1 and tAccessTask2 acquire a token successfully.
  - tAccessTask3 blocks until the other task releases a token.



Multiple shared-resource-access synchronization

- **Pseudo code**

```
tAccessTask ( )
{
        :
        acquire a counting semaphore token
        read or write to shared resource
        release a counting semaphore token
        :
}
```



Multiple shared-resource-access synchronization

KWANGWOON UNIVERSITY

- **If a task release a semaphore that it did not originally acquire.**
  - ➔ use "mutex".
  - Instead of the previous code, use the following.

```
tAccessTask ( )
{
                :
                acquire first mutex in non-blocking way
                                if not successful then acquire 2nd mutex in a blocking way
                read or write to shared resource
                release the acquired mutex
                :
}
```

Case study: uCOS-II

Semaphores

동기화 메커니즘

- **인터럽트 잠금**
  - OSEnterCritical() / OSExitCritical()
- **선점금지**
  - OSSchedLock() / OSSchedUnlock()
- **기타**
  - 세마포어
  - 뮤텍스
  - 이벤트 플래그

동기화 구현

- uC/OS-II는 커널 내부의 동기화 메커니즘으로 대부분 “인터럽트 잠금” 기능을 이용

- 응용 프로그램은 선점금지, 인터럽트 잠금, 세마포어, 이벤트 등을 활용하여 동기화 구현

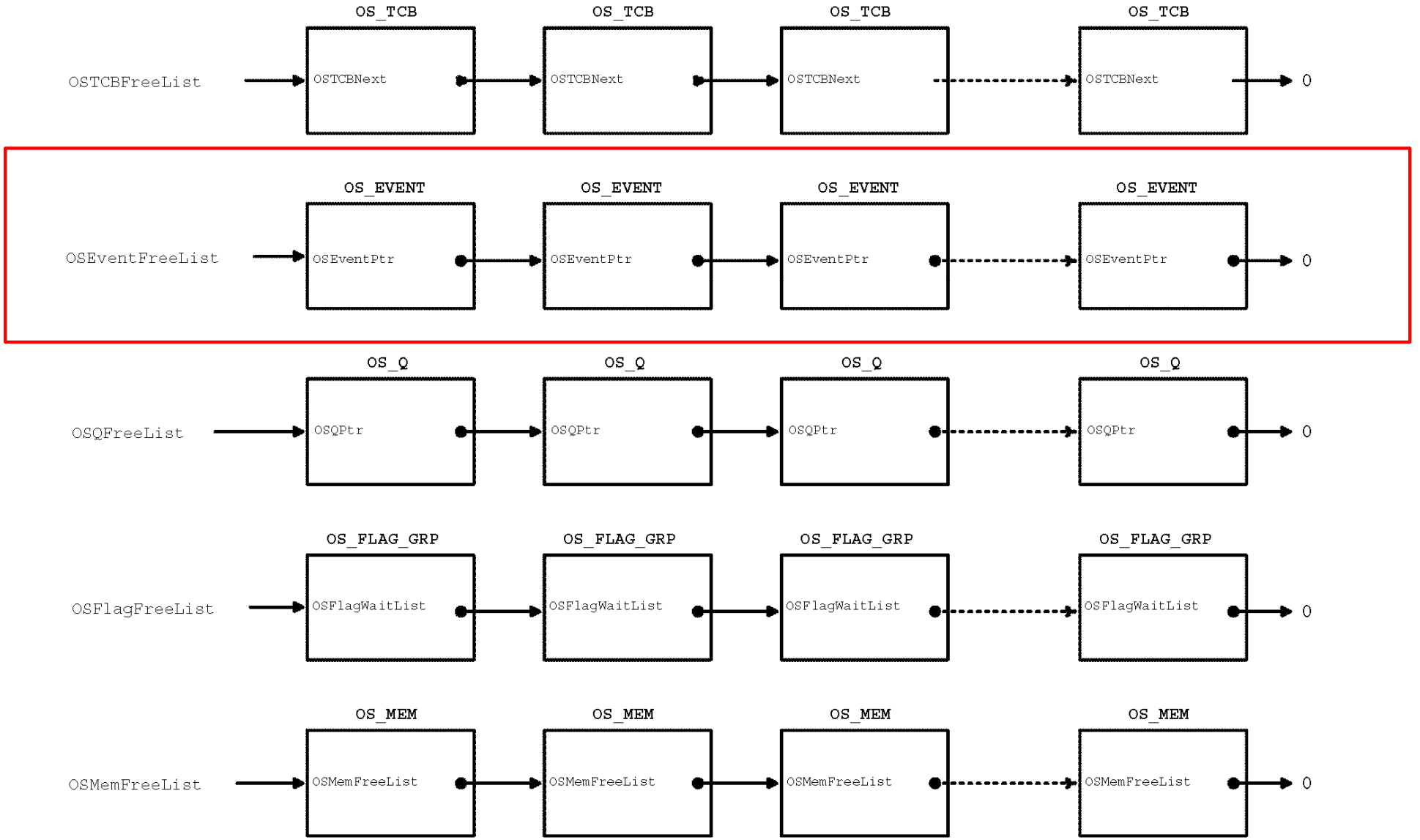
세마포어 개요

- **uC/OS-II 세마포어**
  - 바이너리 / 카운팅 세마포어 (카운트는 최대 65535까지 허용)
  - 세마포어 소유자에 대한 체크가 없음.

  - API
    - OSSemCreate() // 세마포어 생성 및 초기화
    - OSSemDel() // 세마포어 삭제
    - OSSemPend() // 세마포어 획득
    - OSSemPost() // 세마포어 반환
    - OSSemAccept() // 세마포어 획득 시도
    - OSSemQuery() // 세마포어 정보 획득

  - uC/OS-II에서 세마포어는 EventControlBlock 구조체를 이용하여 구현됨

OSInit() review



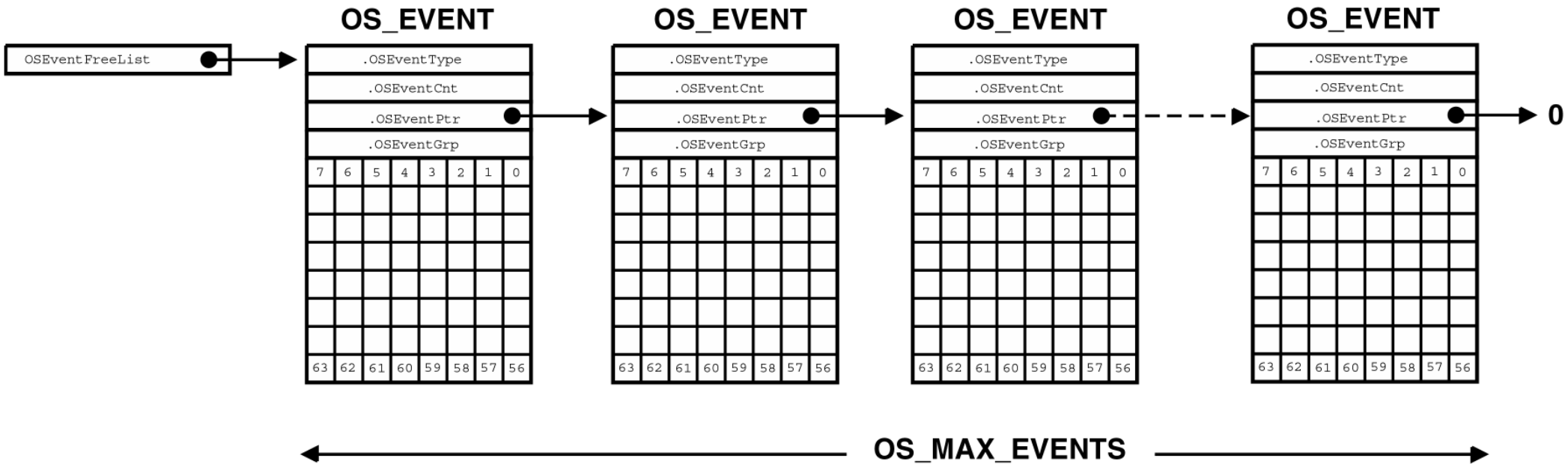
세마포어 구조체

- **uC/OS-II에서 세마포어는 EventControlBlock 구조체를 이용하여 구현됨**

```
typedef struct {
    INT8U   OSEventType;           /* Event type                          */
    INT8U   OSEventGrp;            /* Group for wait list                 */
    INT16U  OSEventCnt;            /* Count (when event is a semaphore)*/
    void   *OSEventPtr;            /* Ptr to message or queue structure*/
    INT8U   OSEventTbl[OS_EVENT_TBL_SIZE]; /* Wait list for event      */
} OS_EVENT;
```

  - OSEventType
    - 세마포어의 경우, OS_EVENT_TYPE_SEM 으로 지정
  - OSEventPtr
    - 세마포어의 경우, 사용하지 않음
  - OSEventTbl[], OSEventGrp
    - 준비 리스트와 동일한 방법으로 세마포어 대기 리스트 표현
  - OSEventCnt
    - 세마포어 카운트 저장

OS_EVENT 구조체


OSEventFreeList
OS_EVENT
.OSEventType
.OSEventCnt
.OSEventPtr
.OSEventGrp
7 6 5 4 3 2 1 0
63 62 61 60 59 58 57 56
OS_EVENT
OS_EVENT
OS_EVENT
0
OS_MAX_EVENTS

Creating a Semaphore, OSSemCreate()

- **OS_SEM.C / Kernel**

- **OS_EVENT *OSSemCreate (INT16U cnt)**
  - cnt : 세마포어의 초기 카운트

  - 세마포어를 생성하고, cnt로 초기화
  - 생성된 세마포어 반환

Acquire a Semaphore, OSSemPend()

- **OS_SEM.C / Kernel**

- **void OSSemPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)**
  - pevent : 대상 세마포어
  - timeout : 획득 실패 시, timeout tick 수
  - err : 세마포어 연산의 결과값

  - 세마포어를 획득한다.
  - 획득 실패 시, timeout ticks 만큼, 세마포어 대기 리스트에서 블록 상태로 대기
  - timeout이 0으로 호출되면, 획득 실패 시, 획득 성공할 때까지 무한정 대기
  - ISR은 호출할 수 없음

Release a Semaphore, OSSemPost()

- **OS_SEM.C / Kernel**

- **INT8U OSSemPost (OS_EVENT *pevent)**
  - pevent : 대상 세마포어

  - 세마포어를 반환한다.
  - 대기 중인 태스크가 있으면, 최상위 우선순위 태스크를 깨움.
  - 대기 중인 태스크가 없으면, 세마포어 count 증가.

Try to acquire a Semaphore, OSSemAccept()

- **OS_SEM.C / Kernel**
- **INT16U OSSemAccept (OS_EVENT *pevent)**
  - pevent : 대상 세마포어
  - 세마포어를 획득을 시도한다.
  - 획득 실패하는 경우, 대기하지 않고, 바로 return.
    - return 값이 1이상이면, 태스크가 세마포어 획득했음을 의미
    - return 값이 0이면, 태스크가 세마포어 획득 실패했음을 의미

Get the status of a Semaphore, OSSemQuery()

KWANGWOON UNIVERSITY

- **OS_SEM.C / Kernel**

- **INT8U OSSemQuery (OS_EVENT *pevent, OS_SEM_DATA *pdata)**

  - pevent : 대상 세마포어
  - pdata : 세마포어 상태 정보를 복사할 메모리 공간

  - OS_EVENT 구조체에서 세마포어에 해당하는 정보만 pdata로 복사
    - 대기 리스트
    - 세마포어 카운트

Deleting a Semaphore, OSSemDel()

- **OS_SEM.C / Kernel**

- **OS_EVENT *OSSemDel (OS_EVENT *pevent, INT8U opt, INT8U *err)**
  - pevent : 대상 세마포어
  - opt : 옵션
  - err : 수행 결과

  - 세마포어를 삭제
  - option이 OS_DEL_NO_PEND이면, 대기 태스크가 없는 경우에만 세마포어 삭제
  - option이 OS_DEL_ALWAYS이면, 대기 태스크가 있는 경우에도 세마포어 삭제.
    - 대기 태스크들은 모두 wakeup 됨

Example codes

Semaphores

문제 1

- **다음에 기술된 대로 프로그램을 작성하고 실행 결과를 관찰한다.**
  - TicksPerSec를 50으로 수정 (OS_CFG.H)
  - 우선순위 1-5인 태스크 5개를 생성한다.
    - main에서 모든 태스크 생성
  - Main()
    - 난수표 초기화
    - 태스크 생성
  - 각 태스크
    - result.txt 파일을 append 모드로 연다.
    - current clock tick, 자신의 priority, "file open" 메시지를 파일에 출력한다
    - 임의의 클록 수 (1-3) 만큼 sleep (난수 발생 함수 rand() 이용)
    - current clock tick, 자신의 priority, "file close" 메시지를 파일에 출력한다
    - 파일 닫기
    - 임의의 클록 수 (1-3) 만큼 sleep (난수 발생 함수 rand() 이용)
    - 1-6 반복

문제1 - 코드

```
#include "includes.h"
#include <time.h>

#define  TASK_STK_SIZE                512
#define  N_TASKS                       5

OS_STK    TaskStk[N_TASKS][TASK_STK_SIZE];                    // (1)

void  Task(void *data);
void  CreateTasks(void);

int  main (void)
{
    OSInit();                                                 // (2)

    srand(time(NULL));                                        // (3)

    CreateTasks();                                            // (4)

    OSStart();                                                // (5)
    return 0;
}
```

문제1 - 코드

```
void  CreateTasks (void)
{
    INT8U  i;

    for (i = 0; i < N_TASKS; i++) {
            OSTaskCreate(Task, (void *) 0, &TaskStk[i][TASK_STK_SIZE - 1], (INT8U) (i + 1));
                                                                          // (6)
    }
}
```

문제1 - 코드

```
void  Task (void *pdata)
{
    FILE *out;
    INT8U sleep;

    for (;;) {
            out = fopen("result.txt", "a");
                                                                    // (7)
            fprintf(out, "%4u: Task %u, file open₩n", OSTimeGet(), OSTCBCur->OSTCBPrio);
                                                                    // (8)
            fflush(out);
            sleep = (rand() % 3) + 1;
            OSTimeDly(sleep);                                       // (9)
            fprintf(out, "%4u: Task %u, file close₩n", OSTimeGet(), OSTCBCur->OSTCBPrio);
                                                                    //(10)
            fflush(out);
            fclose(out);                                            // (11)
            sleep = (rand() % 3) + 1;
            OSTimeDly(sleep);                                       // (12)
    }
}
```

문제2

- **문제 1 내용을 다음과 같이 수정한다.**
  - 한번에 하나의 태스크만 파일을 접근할 수 있도록 함
    - file open 이후, 다른 태스크는 file open 불가.
    - file close 이후, 다른 태스크가 file open 수행 가능

문제2 - 코드

```
#include "includes.h"
#include <time.h>

#define  TASK_STK_SIZE                512
#define  N_TASKS                       5

OS_STK    TaskStk[N_TASKS][TASK_STK_SIZE];
OS_EVENT *sem;                                                    // (1)

void  Task(void *data);
void  CreateTasks(void);

int  main (void)
{
    OSInit();

    srand(time(NULL));
    sem = OSSemCreate(1);                                         // (2)

    CreateTasks();

    OSStart();
    return 0;
}
```

문제2 - 코드

```
void  CreateTasks (void)
{
    INT8U  i;

    for (i = 0; i < N_TASKS; i++) {
            OSTaskCreate(Task, (void *) 0, &TaskStk[i][TASK_STK_SIZE - 1], (INT8U) (i + 1));
                                                                    // (3)
    }
}
```

문제2 - 코드

```
void  Task (void *pdata)
{
    FILE *out;
    INT8U sleep;
    INT8U err;

    for (;;) {
            OSSemPend(sem, 0, &err);                                    // (3)
            out = fopen("result.txt", "a");
            fprintf(out, "%4u: Task %u, file open₩n", OSTimeGet(), OSTCBCur->OSTCBPrio);
            sleep = (rand() % 3) + 1;
            OSTimeDly(sleep);
            fprintf(out, "%4u: Task %u, file close₩n", OSTimeGet(), OSTCBCur->OSTCBPrio);
            fclose(out);
            OSSemPost(sem);                                             // (4)
            sleep = (rand() % 3) + 1;
            OSTimeDly(sleep);
    }
}
```