

## 제목 : Xenomai 실습과제



이름 : 이대호

학과: 항공소프트웨어공학과

학번: 202000969

교수명: 심종익교수님

제출날짜: 22.12.09(금)

# 목차

## 1. 필수 과제

1. 2개의 태스크 생성 샘플 프로그램
2. **Wait-and-signal synchronization**
3. **Non-interlocked, one-way data communication**

## 2. 선택 과제

1. 6.4.2 여러 태스크의 대기 -신호(**Wait-Signal**)동기화
2. 6.4.3 크레딧-트래킹(**Credit-Tracking**) 동기화
3. 6.4.4 단일 공유자원 접근 동기화(**카운팅 세마포어**)
4. 6.4.4 단일 공유자원 접근 동기화(**무텍스**)
5. 6.4.5 재귀적 공유자원 접근 동기화
6. 6.4.6 복수 공유자원 접근 동기화

## 1.1 2개의 태스크 생성 샘플 프로그램

```
#include<stdio.h>
#include<sys/mman.h>
#include<native/task.h>

#define STACK_SIZE 0
#define TASKMODE 0
#define T_SIGNAL_PRI 90
#define T_WAIT_PRI 99

RT_TASK T_WAIT_desc, T_SIGNAL_desc;

void TWAITTask()
{
    printf("TWAITTASK 시작!\n");
    while(1)
    {
        printf("TWAIT_TASK : 실행중..\n");
        sleep(2);
    }
}

void TSIGNALTask()
{
    printf("TSIGNAL_TASK 시작!\n");
    while(1)
    {
        printf("TSIGNAL_TASK : 실행중\n");
        sleep(2);
    }
}

int main()
{
    if(rt_task_create(&T_WAIT_desc, "TWAITTask", STACK_SIZE, T_WAIT_PRI, TASKMODE))
    {
        printf("T_WAIT_SIGNAL_TASK 생성에 실패하였습니다.\n");
        return -1;
    }

    printf("T_WAIT_TASK 생성에 성공하였습니다!\n");

    if(rt_task_create(&T_SIGNAL_desc, "TSIGNALTask", STACK_SIZE, T_SIGNAL_PRI, TASKMODE))
    {
        printf("T_SIGNAL_TASK 생성에 실패하였습니다.\n");
        return -1;
    }

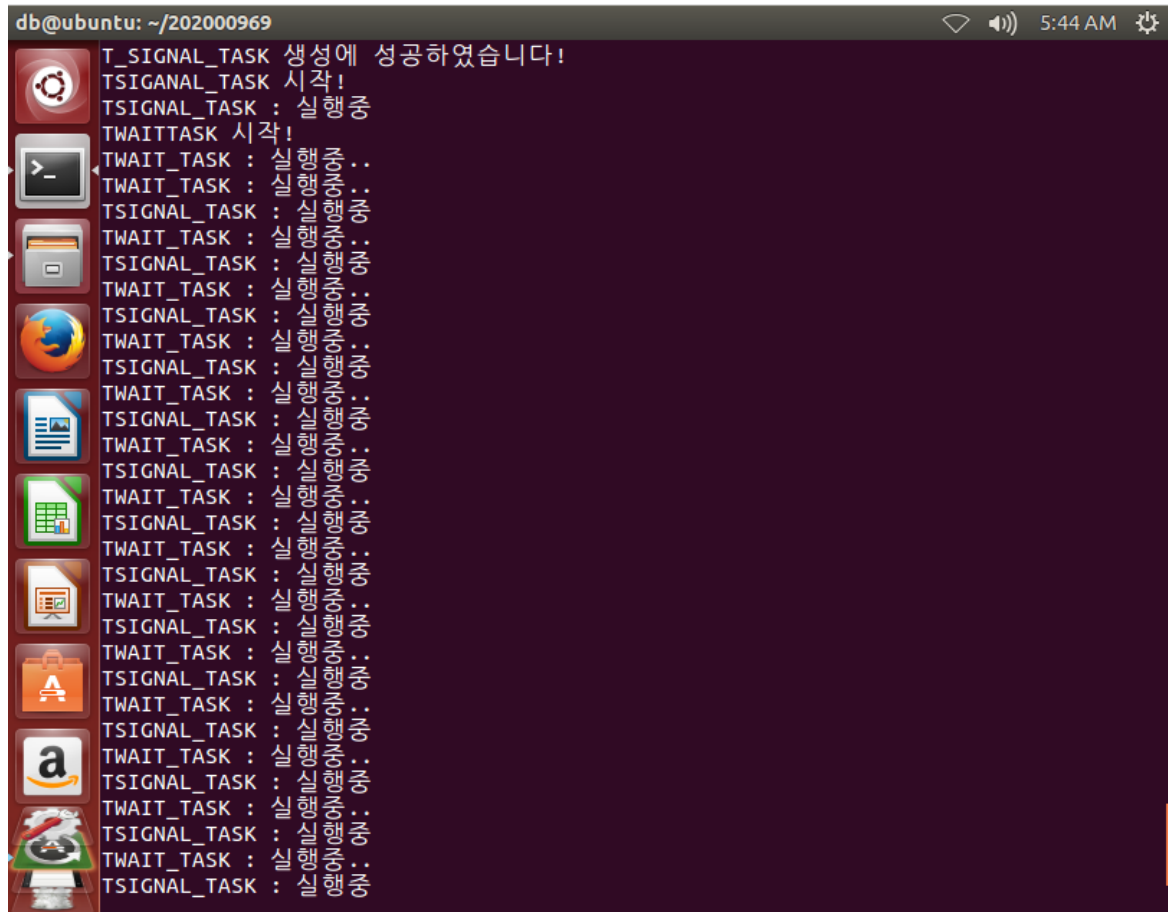
    printf("T_SIGNAL_TASK 생성에 성공하였습니다!\n");

    rt_task_start(&T_SIGNAL_desc, &TSIGNALTask, NULL);
    rt_task_start(&T_WAIT_desc, &TWAITTask, NULL);
    pause();

    return 0;
}
db@ubuntu:~/20200969$
```

태스크를 생성하기 위해서 xenomai에서 제공하는 API인 `rt_task_create`에 필요한 파라미터들을 입력해준다. 디스크립터, 함수명, 스택사이즈, 태스크의 우선순위, 태스크모드를 입력해준다. 해당 값들은 `#define` 및 전역변수로 선언을 해주었다. 실행값들이 잘 나오기 위해서 `TWAIT` 및 `TSIGNAL` 함수에 `sleep(2)`를 넣어서 시간지연을 해주어 코드를 확인하기 편하게 해주었다. 메인함수에서는 해당 task들이 생성되었는지에 따라 오류문 혹은 생성이 잘되었다는 print문을 통해 알려주었다. 성공한다면 함수에서 정의한 태스크들의 루틴을 실행해주고,

```
db@ubuntu:~/20200969$ sudo ./create_task
T_WAIT_TASK 생성에 성공하였습니다!
T_SIGNAL_TASK 생성에 성공하였습니다!
TSIGNAL_TASK 시작!
TSIGNAL_TASK : 실행중
TWAITTASK 시작!
TWAIT_TASK : 실행중..
TWAIT_TASK : 실행중..
TSIGNAL_TASK : 실행중
ACdb@ubuntu: ~/20200969$
```



4/27

## 1.2 Wait-Signal-Synchronization(대기 신호 동기화)

2개의 태스크가 데이터 교환 없이 동기화만을 위해 통신을 할 경우가 존재한다. 이럴 경우 2개의 태스크 사이에서 실행권 전달을 제어하기 위해 바이너리 세마포어를 사용할 수 있다. 하지만 xenomai에서는 바이너리 세마포어를 지원하지 않아, 카운트 세마포어의 값을 1로 지정하여 바이너리 세마포어 처럼 사용하도록 한다. 초기 세마포어를 0으로 하여 tWaitTask가 우선순위가 높아 세마포어를 요청하지만, 세마포어가 없어 블록상태로 들어간다. 그 이후 tSignalTask가 실행이 되어 세마포어를 반환하면 블록상태의 tWaittask가 실행된다.

아래의 사진은 대기 신호 동기화를 구현하기 위한 코드이다.

```
#include<stdio.h>
#include<sys/mman.h>
#include<native/task.h>
#include<native/sem.h>

#define STACK_SIZE 0
#define TASKMODE 0
#define T_SIGNAL_PRI 40
#define T_WAIT_PRI 50

#define BINARYSEM 0
#define SEMAPHORE_MODE S_PRIO

RT_TASK T_WAIT_desc, T_SIGNAL_desc;
RT_SEM SEMAPHORE_desc;

int i,j;

void TWAITTask(){
    printf("TWAITSIGNAL 시작!\n");
    while(1){
        for(i=0;i<1000;i++){
            for(j=0;j<1000;j++){
                printf("TWAITTASK : 일하는 중..\nTWAITTask : 현재 세마포어 기다리는중..\n");
                rt_sem_p(&SEMAPHORE_desc, TM_INFINITE);
                printf("TWAITTASK : 세마포어 획득! 계속해서 일하겠습니다.\n");
                for(i=0;i<1000;i++){
                    for(j=0;j<1000;j++){

```



두개의 태스크 생성과 다른 부분은 세마포어를 획득하고 반환 생성하는 과정이 추가되었다. 태스크를 생성하는 과정은 위에 설명했던 것과 동일하다. 추가된 부분 및 실행과정에 대해서 코드 설명을 진행하겠다.

먼저 세마포어 또한 디스크립터를 요구하기 때문에 세마포어 디스크립터를 선언해준다. TWaitTask 및 TSignalTask에서 세마포어를 요하기 때문에 세마포어를 얻는 xenomai api인 rt\_sem\_p()를 사용한다. 사용 방법은 디스크립터 및 태스크가 세마포어를 기다리는 모드를 설정해준다. 내가 선언한 모드는 세마포어를 받을 때 까지 무한정 기다리는 TM\_INFINITE를 선언해주었다. 세마포어를 다 사용 후 반환할 때는 rt\_sem\_p()라는 API를 사용하는데 이 때는 디스크립터를 파라미터로 입력해주면 된다.

메인 함수에는 세마포어의 생성을 해준다. 생성법은 rt\_sem\_create() API를 이용한다. 이 때 필요한 파라미터들은 디스크립터, 세마포어의 이름, 세마포어 생성 갯수, 세마포어의 모드를 입력해준다. 나는 우선순위 기반의 세마포어를 생성하였다. 태스크와 동일하게 생성이 되지 않았다면 오류 메시지를 출력한 이후 프로그램을 종료해주었다. 정상적으로 생성이 되었다면 세마포어가 생성되었다고 출력했다. 태스크가 종료가 된 후, 세마포어를 삭제해주는 API를 사용해주었다. rt\_sem\_delete라는 API이며 파라미터로 디스크립터를 넣어준다.

tSignal과 tWait를 보면 busy-wait를 주기 위해 이중포문을 사용하였으며 tSignal은 세마포어를 반환하고 tWait는 세마포어를 획득 후 반환하는 알고리즘을 가지고 있다. tWait의 우선순위가 tSignal task보다 우선순위가 높다. xenomai에서는 숫자가 높을수록 우선순위가 높기 때문에 tWaitTask의 우선순위를 50, tSignalTask의 우선순위를 40으로 부여하였다. 해당 코드를 실행한 결과는 아래와 같다.

```
db@ubuntu: ~/202000969
TWAITTASK : 세마포어 획득! 계속해서 일하겠습니다.
TWAITTASK : 일하는 중..
TWAITTask : 현재 세마포어 기다리는중..
TSIGNALTask : 세마포어 반환 ! 계속해서 일하겠습니다.
TSIGNALTask : 일하는 중..
TSIGNALTask : 이제 세마포어를 반환합니다.
TWAITTASK : 세마포어 획득! 계속해서 일하겠습니다.
TWAITTASK : 일하는 중..
TWAITTask : 현재 세마포어 기다리는중..
TSIGNALTask : 세마포어 반환 ! 계속해서 일하겠습니다.
TSIGNALTask : 일하는 중..
TSIGNALTask : 이제 세마포어를 반환합니다.
TWAITTASK : 세마포어 획득! 계속해서 일하겠습니다.
TWAITTASK : 일하는 중..
TWAITTask : 현재 세마포어 기다리는중..
TSIGNALTask : 세마포어 반환 ! 계속해서 일하겠습니다.
TSIGNALTask : 일하는 중..
TSIGNALTask : 이제 세마포어를 반환합니다.
TWAITTASK : 세마포어 획득! 계속해서 일하겠습니다.
TWAITTASK : 일하는 중..
TWAITTask : 현재 세마포어 기다리는중..
TSIGNALTask : 세마포어 반환 ! 계속해서 일하겠습니다.
TSIGNALTask : 일하는 중..
TSIGNALTask : 이제 세마포어를 반환합니다.
```

실행결과 위에서 설명한 현상대로 정상적으로 프로그램이 실행된 것을 볼 수 있다.

### 1.3 Non-interlock, One-way data communication

```
#include <stdio.h>
#include<stdlib.h>
#include <string.h>
#include <native/task.h>
#include <native/queue.h>

#define TASK_MODE 0
#define SINKPR 99
#define SOURCEPR 50
#define QPIPESIZE 30
#define Q_LIMIT 5
#define Q_MODE Q_FIFO
#define STACK_SIZE 0

RT_TASK SINK_desc,SOURCE_desc;
RT_QUEUE QUEUE_desc;

void SOURCET()
{
    int length,i=0;
    void *letter;
    const char *mes[]={ "Hi\n", "RTOS\n"};
    printf("메세지를 보냅니다!\n");
    while(1){
        i%=2;
        printf("SOURCE : 실행중!\n");
        length=strlen(mes[i])+1;
        if(!(letter=rt_queue_alloc(&QUEUE_desc,length))){
            printf("큐 할당 에러\n");
            exit(0);
        }

        strcpy(letter,mes[i++]);
        rt_queue_send(&QUEUE_desc,letter,length,Q_NORMAL);
        printf("SOURCE : 전송완료!\n");
        sleep(2);
    }
}
```

인터락 없는 단방향 데이터 통신은 송신 태스크(소스), 메시지 큐, 수신 태스크(싱크)로 구성된다. 인터락이 없기 때문에 송신 및 수신 태스크의 동작은 동기화 되지 않는다. 소스는 단순히 메시지를 보낼 뿐 싱크 태스크로 부터 확인 응답을 요하지 않는다. 내 코드에서는 수신 태스크의 우선순위를 송신 태스크보다 높은 우선 순위를 부여하였다. 그 결과 수신 태스크가 먼저 실행돼 빈 메시지큐에서 블록된다. 송신 태스크가 메시지를 보내면 수신 태스크가 메시지를 받아서 실행을 재개한다. Void SOURCET()함수는 송신 태스크이며 보낼 메시지를 문자형 배열에 입력을 하여 메시지를 보낼 큐를 할당, 메시지를 복사하여 큐를 통해 메시지를 보낸다. 큐

는 Normal mode 즉, FIFO방식으로 하였다. 그 이후 전송이 완료되었다면 메시지를 전송완료 했다는 출력문을 출력한다.

```
void SINKT()
{
    ssize_t length;
    void *letter;
    printf("SINKT : 실행시작!\n");
    while(1){
        if(rt_queue_bind(&QUEUE_desc,"QUEUE",TM_INFINITE)){
            printf("바인드 오류발생\n");
            exit(0);
        }

        while(length=rt_queue_receive(&QUEUE_desc,&letter,TM_INFINITE)>0){
            printf("SINKT 메시지를 받았습니다!\n");
            printf("message : %s\n",(const char*)letter);
            rt_queue_free(&QUEUE_desc,letter);
            sleep(2);
        }
        rt_queue_unbind(&QUEUE_desc);
        if(length != -EIDRM){
            printf("메세지 길이 오류가 발생하였습니다.\n");
            exit(0);
        }
    }
}

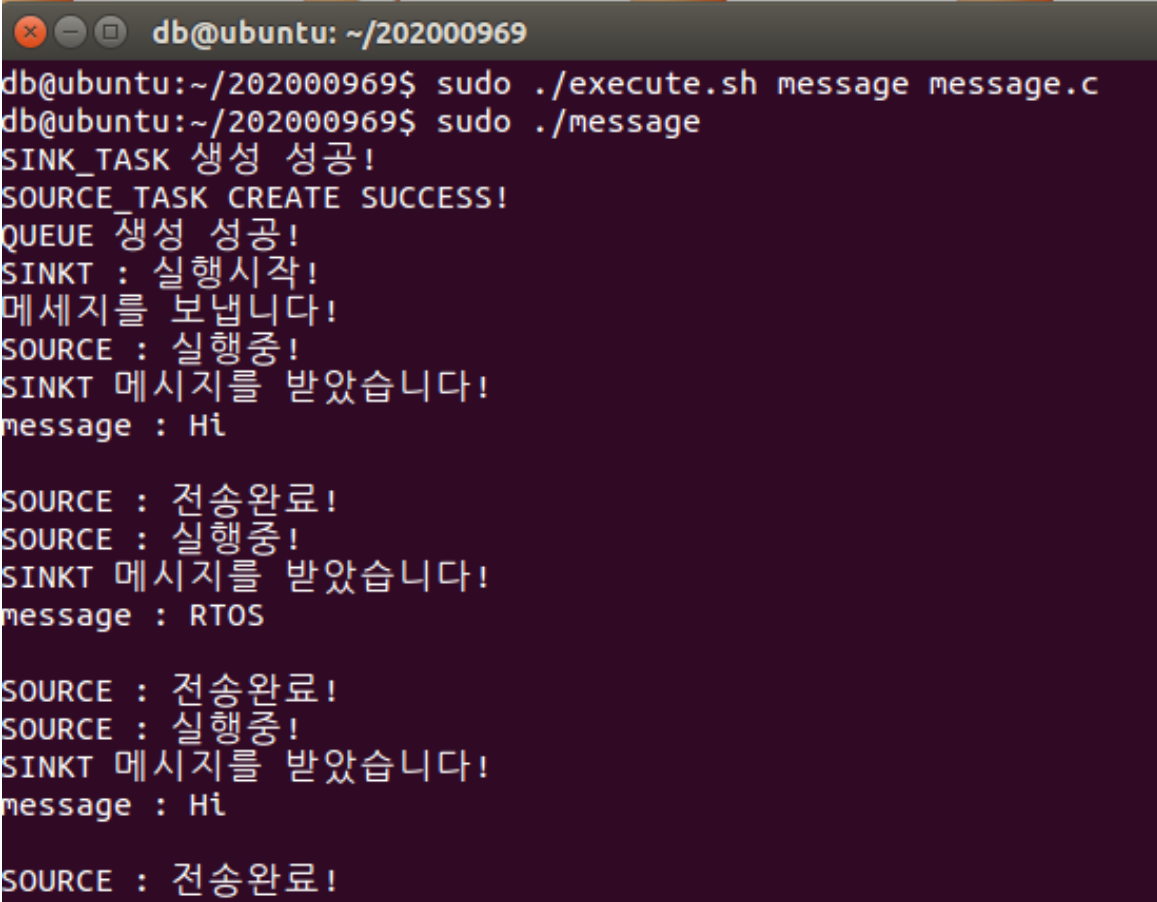
int main()
{
    if(rt_task_create(&SINK_desc,"SINKT",STACK_SIZE,SINKPR,TASK_MODE)!=0){
        printf("SINK_TASK 생성에 실패하였습니다.\n");
        return -1;
    }
    printf("SINK_TASK 생성 성공!\n");
    if(rt_task_create(&SOURCE_desc,"SOURCET",STACK_SIZE,SOURCEPR,TASK_MODE)!=0){
        printf("SOURCE_TASK 생성에 실패하였습니다.\n");
        return -1;
    }
    printf("SOURCE_TASK CREATE SUCCESS!\n");
    if(rt_queue_create(&QUEUE_desc,"QUEUE",QPIPE_SIZE,Q_LIMIT,Q_MODE)){
        printf("QUEUE 생성에 실패하였습니다.\n");
        return -1;
    }
    printf("QUEUE 생성 성공!\n");
    rt_task_start(&SINK_desc,&SINKT,NULL);
    rt_task_start(&SOURCE_desc,&SOURCET,NULL);
    pause();
    return 0;
}
db@ubuntu:~/20200969$
```

Void SINKT()는 함수는 수신 태스크이다. 우선 순위가 높기 때문에 메시지가 올 때까지 대기 를 하였다가 메시지가 도착하면 메시지를 받는다(출력한다). 오류가 발생 시 오류 메시지를 출 력하고, 반복문을 통해 송신 태스크에서 보낸 메시지를 반복적으로 출력한다.

메인 함수에서는 앞서 설명한 프로그램 처럼 태스크를 생성한 후, 메시지를 보낼 큐를 생성 하고 태스크를 실행한다.



정상적으로 실행되었다면 아래 사진처럼 결과가 출력된다.



```
db@ubuntu: ~/202000969
db@ubuntu:~/202000969$ sudo ./execute.sh message message.c
db@ubuntu:~/202000969$ sudo ./message
SINK_TASK 생성 성공!
SOURCE_TASK CREATE SUCCESS!
QUEUE 생성 성공!
SINKT : 실행시작!
메세지를 보냅니다!
SOURCE : 실행중!
SINKT 메시지를 받았습니다!
message : Hi

SOURCE : 전송완료!
SOURCE : 실행중!
SINKT 메시지를 받았습니다!
message : RTOS

SOURCE : 전송완료!
SOURCE : 실행중!
SINKT 메시지를 받았습니다!
message : Hi

SOURCE : 전송완료!
```

## 2.1 6.4.2 여러 태스크의 대기-신호(Wait-signal) 동기화

```
db@ubuntu:~/202000969$ cat flush.c
#include<stdio.h>
#include<sys/mman.h>
#include<native/task.h>
#include<native/sem.h>

#define STACK_SIZE 0
#define TASKMODE 0
#define T_SIGNAL_PRI 40
#define T_WAIT_PRI1 99
#define T_WAIT_PRI2 98
#define T_WAIT_PRI3 97

#define BINARYSEM 0
#define SEMAPHORE_MODE S_PRIO

RT_TASK T_WAIT_desc1, T_WAIT_desc2, T_WAIT_desc3, T_SIGNAL_desc;
RT_SEM SEMAPHORE_desc;
int i,j;

void TWAITTask1(){
    printf("TWAITTASK1 시작!\n");
    while(1)
    {
        printf("TWAITTASK1 : 현재 세마포어 기다리는중..\n");
        sleep(2);
        rt_sem_p(&SEMAPHORE_desc, TM_INFINITE);
        printf("TWAITTASK1 : 세마포어 획득!\n일 끝났습니다!\n");
        sleep(5);
    }
}

void TWAITTask2(){
    printf("TWAITTASK2 시작!\n");
    while(1)
    {
        printf("TWAITTASK2 : 현재 세마포어 기다리는중..\n");
        sleep(2);
        rt_sem_p(&SEMAPHORE_desc, TM_INFINITE);
        printf("TWAITTASK2 : 세마포어 획득!\n일 끝났습니다!\n");
        sleep(5);
    }
}
```

```

void TWAITTask3(){
    printf("TWAITTASK3 시작!\n");
    while(1)
    {
        printf("TWAITTASK3 : 현재 세마포어 기다리는중..\n");
        sleep(2);
        rt_sem_p(&SEMAPHORE_desc, TM_INFINITE);
        printf("TWAITTASK3 : 세마포어 획득!\n일 끝냈습니다!\n");
    }
}

void T SIGNALTask(){
    printf("T SIGNALTask 시작!\n");
    while(1){
        printf("T SIGNALTask : 일하는 중..\nT SIGNALTask : 이제 세마포어를 반환합니다.\n");
        sleep(2);
        rt_sem_broadcast(&SEMAPHORE_desc);
        printf("T SIGNALTask : 세마포어 반환 !\n");
    }
}

int main(){

    if(rt_task_create(&T_WAIT_desc1, "TWAITTask1", STACK_SIZE, T_WAIT_PRI1, TASKMODE)){
        printf("TWAITTASK1이 생성되지 않았습니다.\n");
        return -1;
    }
    printf("TWAITTASK1이 생성되었습니다!\n");

    if(rt_task_create(&T_WAIT_desc2, "TWAITTask2", STACK_SIZE, T_WAIT_PRI2, TASKMODE)){
        printf("TWAITTASK2가 생성되지 않았습니다.\n");
        return -1;
    }
    printf("TWAITTASK2가 생성되었습니다!\n");

    if(rt_task_create(&T_WAIT_desc3, "TWAITTask3", STACK_SIZE, T_WAIT_PRI3, TASKMODE)){
        printf("TWAITTASK3이 생성되지 않았습니다.\n");
        return -1;
    }
    printf("TWAITTASK3이 생성되었습니다!\n");

    if(rt_task_create(&T_SIGNAL_desc, "T SIGNALTask", STACK_SIZE, T_SIGNAL_PRI, TASKMODE)){
        printf("T SIGNALTASK가 생성되지 않았습니다.\n");
        return -1;
    }
    printf("T SIGNALTASK가 생성되었습니다!\n");

    if(rt_sem_create(&SEMAPHORE_desc, "SEMAPHORE", BINARYSEM, SEMAPHORE_MODE)){
        printf("세마포어가 생성되지 않았습니다.\n");
        return -1;
    }
    printf("세마포어가 생성되었습니다!\n");

    rt_task_start(&T_WAIT_desc1, &TWAITTask1, NULL);
    rt_task_start(&T_WAIT_desc2, &TWAITTask2, NULL);
    rt_task_start(&T_WAIT_desc3, &TWAITTask3, NULL);
    rt_task_start(&T_SIGNAL_desc, &T SIGNALTask, NULL);
    pause();
    rt_sem_delete(&SEMAPHORE_desc);
    return 0;
}

```

TWAITTask1, TWAITTask2, TWAITTask3에게 순서대로 높은 우선 순위를 부여한다. 하지만 세마포어를 부여받지 못하여 세마포어를 받을 때까지 무한정 대기를 한다. 순서대로 루틴을 실행하며, 우선순위가 제일 낮은 T SIGNALTask 함수가 rt\_sem\_broadcast() API를 사용하여 루틴이 종료되면, 블록상태였던 TWAITTask1,2,3은 블록상태에서 벗어나며 자신의 일을 수행한다. 세마포어를 얻던 얻지 못하던 루틴을 실행한다. 해당 코드에서는 세마포어를 얻었고 일을 수행한다고 하였으나, 해당 사항은 예시로 출력하였다.

해당 코드의 실행결과는 아래와 같다.

```
db@ubuntu:~/202000969$ sudo ./flush
TWAITTASK1이 생성되었습니다!
TWAITTASK2가 생성되었습니다!
TWAITTASK3이 생성되었습니다!
TSIGNALTASK가 생성되었습니다!
세마포어가 생성되었습니다!
TWAITTASK1 시작!
TWAITTASK1 : 현재 세마포어 기다리는중..
TWAITTASK2 시작!
TWAITTASK2 : 현재 세마포어 기다리는중..
TWAITTASK3 시작!
TWAITTASK3 : 현재 세마포어 기다리는중..
TSIGNALTask 시작!
TSIGNALTask : 일하는 중..
TSIGNALTask : 이제 세마포어를 반환합니다.
TWAITTASK1 : 세마포어 획득!
일 끝냈습니다!
TWAITTASK2 : 세마포어 획득!
일 끝냈습니다!
TWAITTASK3 : 세마포어 획득!
일 끝냈습니다!
```

실행결과를 보면 우선순위에 따라 TWAITTASK1, 2, 3순으로 실행이 되었으며 블록상태로 대기하다, TSIGNALTask가 broadcast를 해준 이후 블록상태에서 벗어난 것을 볼 수 있다.

### 2.2 6.4.3 크레딧-트래킹(Credit-Tracking) 동기화

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/mman.h>
#include<native/task.h>
#include<native/sem.h>

#define STACK_SIZE 0
#define TASKMODE 0
#define Sem_CT 0
#define Sem_MODE S_PRIO
#define tSTPRIO 99
#define tWTPRIO 90

RT_TASK ts_desc,tw_desc;
RT_SEM sem_desc;

int count=0;

void ts()
{
    printf("tSignal 실행!\n");
    while (1)
    {
        printf("작업 실행중...\n");
        rt_sem_v(&sem_desc);
        count++;
        printf("세마포어 반납 : %d\n",count);
        sleep(1);
    }
}
```

```

void tW()
{
    printf("tWait 실행!\n");
    while (1)
    {
        printf("세미포어를 획득하고싶습니다!\n");
        rt_sem_p(&sem_desc, TM_INFINITE);
        count--;
        printf("세마포어 획득 : %d\n", count);
        printf("작업을 실행합니다...\n");
        rt_sem_v(&sem_desc);
        count++;
        printf("tw세마포어 반납 : %d\n", count);
        sleep(5);
    }
}

int main()
{
    if(rt_task_create(&tS_desc, "tS", STACK_SIZE, tSTPRIO, TASKMODE))
    {
        printf("tSignalTask가 생성되지 않았습니다.\n");
        exit(0);
    }

    printf("tSignalTask 생성완료!\n");

    if(rt_task_create(&tW_desc, "tW", STACK_SIZE, tSTPRIO, TASKMODE))
    {
        printf("tWaitTask가 생성되지 않았습니다.\n");
        exit(0);
    }

    printf("tWaitTask 생성완료\n");

    if(rt_sem_create(&sem_desc, "semaphore", Sem_CT, Sem_MODE))
    {
        printf("세마포어가 생성되지 않았습니다.\n");
        exit(0);
    }

    printf("세마포어 생성완료\n");

    rt_task_start(&tS_desc, &tS, NULL);
    rt_task_start(&tW_desc, &tW, NULL);
    pause();
    rt_sem_delete(&sem_desc);

    return 0;
}

```



세마포어를 통해 신호를 보내는 태스크의 실행 주기가 신호를 받는 태스크의 수행 주기보다 빠른 경우 신호의 발생 횟수를 세마포어에 기록할 수 있는 방법이 필요할 때 사용한다. 주로 인터럽트가 발생할 때 카운팅 세마포어를 증가, 그 수만큼 release하는 방법인데 sleep함수를 통해서 인터럽트 대신 구현을 하였다. 이것을 구현하기 위해 tW는 tS보다 낮은 우선순위를 가지며 tS가 release할 때까지 블록상태에 놓인다.

tS함수는 release만 해주고 tW함수에서 세마포어를 얻기 원하며, sleep(5)초를 주어 시간차를 두었다. tS는 자신의 주기로 실행을 하고 있고 sleep을 통해 인터럽트와 비슷한 환경을 만들어 계속 release를 하도록 하였다. tW는 세마포어를 얻고 반환하는 과정을 반복, tS는 세마포어를 반환하는 과정이 자신의 주기를 실행하는 것이다. 그러면 실행결과를 보도록 하겠다.

```
db@ubuntu:~/202000969$ sudo ./credit
tSignalTask 생성완료!
tWaitTask 생성완료
세마포어 생성완료
tSignal 실행!
작업 실행중...
세마포어 반납 : 1
tWait 실행!
세마포어를 획득하고싶습니다!
세마포어 획득 : 0
작업을 실행합니다...
tw세마포어 반납 : 1
작업 실행중...
세마포어 반납 : 2
작업 실행중...
세마포어 반납 : 3
작업 실행중...
세마포어 반납 : 4
작업 실행중...
세마포어 반납 : 5
세마포어를 획득하고싶습니다!
세마포어 획득 : 4
작업을 실행합니다...
tw세마포어 반납 : 5
작업 실행중...
세마포어 반납 : 6
작업 실행중...
세마포어 반납 : 7
작업 실행중...
세마포어 반납 : 8
작업 실행중...
세마포어 반납 : 9
```

실행결과를 보면 tS함수가 실행되기 전까지 블록상태에 있다가 tS함수가 반환할 때 까지 기다리며 시간차를 통해 반납하는 세마포어의 수는 계속 증가하는 것을 볼 수 있다.

#### 2.3 6.4.4 단일 공유자원 접근 동기화(카운팅 세마포어)

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/mman.h>
#include<native/task.h>
#include<native/sem.h>

#define STACK_SIZE 0
#define TASKMODE 0
#define SEMAPHORE_CT 1
#define SEMAPHORE_MODE S_PRIO
#define tAC1PRIO 99
#define tAC2PRIO 99

RT_TASK tAC1_desc,tAC2_desc;
RT_SEM Semaphore_desc;

int resource=100;

void tAC1()
{
    printf("tAC1 실행\n");
    while (1)
    {
        printf("tAC1 공유자원 접근\n세마포어 대기중입니다.\n");
        rt_sem_p(&Semaphore_desc,TM_INFINITE);
        printf("tAC1 : 세마포어 획득!\n");
        resource-=10;
        printf("tAC1 공유자원 접근 성공! 공유자원 %d\n",resource);
        rt_sem_v(&Semaphore_desc);
        printf("tAC1 세마포어를 반납하였습니다.\n");
        sleep(2);
    }
}
```

```

void tAC2()
{
    printf("tAC2 실행\n");
    while (1)
    {
        printf("tAC2 공유자원 접근\n세마포어 대기중입니다.\n");
        rt_sem_p(&Semaphore_desc, TM_INFINITE);
        printf("tAC2 : 세마포어 획득!\n");
        resource+=110;
        printf("tAC2 공유자원 접근 성공! 공유자원 %d\n",resource);
        rt_sem_v(&Semaphore_desc);
        printf("tAC2 세마포어를 반납하였습니다.\n");
        sleep(2);
    }
}

int main()
{
    if(rt_task_create(&tAC1_desc,"tAC1",STACK_SIZE,tAC1PRIO,TASKMODE))
    {
        printf("tAC1이 생성되지 않았습니다.\n");
        exit(0);
    }

    printf("tAC1 생성완료!\n");

    if(rt_task_create(&tAC2_desc,"tAC2",STACK_SIZE,tAC2PRIO,TASKMODE))
    {
        printf("tAC2가 생성되지 않았습니다.\n");
        exit(0);
    }

    printf("tAC2 생성완료!\n");

    if(rt_sem_create(&Semaphore_desc,"semaphore",SEMAPHORE_CT,SEMAPHORE_MODE)){
        printf("세마포어가 생성되지 않았습니다.\n");
        exit(0);
    }

    printf("세마포어 생성 완료!\n");

    rt_task_start(&tAC1_desc,&tAC1,NULL);
    rt_task_start(&tAC2_desc,&tAC2,NULL);
    pause();
    rt_sem_delete(&Semaphore_desc);

    return 0;
}

```

단일 공유자원에 접근하려는 같은 우선순위를 가진 태스크 2개가 있다. 단일 공유자원은 전역 변수로 선언해주었다. tAC1, tAC2는 공유자원에 접근한다는 것으로 각 10 및 110을 빼주고 더해주었다. 바이너리 세마포어를 지원하지 않아, 하나의 세마포어를 생성하였다. 메인함수는 다른 함수와 동일하게 태스크 및 세마포어 생성 태스크 실행 및 세마포어 삭제를 하였다. 확실한 방법은 뮤텍스를 사용하여서 소유권을 주는 것이다. 세마포어를 줄 경우 다른 곳에서 반환을 하면 동기화에 문제가 발생할 수 있다. 이번 코드 결과에서는 순서대로 잘 들어갔다. 이제 코드의 결과를 보겠다.

```
db@ubuntu:~/202000969$ sudo ./danil
tAC1 생성완료!
tAC2 생성완료!
세마포어 생성 완료!
tAC1 실행
tAC1 공유자원 접근
세마포어 대기중입니다.
tAC1 : 세마포어 획득!
tAC1 공유자원 접근 성공! 공유자원 90
tAC1 세마포어를 반납하였습니다.
tAC2 실행
tAC2 공유자원 접근
세마포어 대기중입니다.
tAC2 : 세마포어 획득!
tAC2 공유자원 접근 성공! 공유자원 200
tAC2 세마포어를 반납하였습니다.
tAC1 공유자원 접근
세마포어 대기중입니다.
tAC2 공유자원 접근
세마포어 대기중입니다.
tAC1 : 세마포어 획득!
tAC1 공유자원 접근 성공! 공유자원 190
tAC1 세마포어를 반납하였습니다.
tAC2 : 세마포어 획득!
tAC2 공유자원 접근 성공! 공유자원 300
tAC2 세마포어를 반납하였습니다.
```

순서대로 세마포어 획득 및 반납을 통해 동기화에 문제가 발생하지 않았다. 공유자원에 접근을 하여 각각의 연산을 잘 수행한 것을 볼 수 있다.

#### 2.4.6.4.4 단일 공유자원 접근 동기화(무텍스)

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/mman.h>
#include<native/task.h>
#include<native/mutex.h>

#define STACK_SIZE 0
#define TASKMODE 0
#define tAC1PRI 99
#define tAC2PRI 99

RT_TASK t1_desc,t2_desc;
RT_MUTEX owner;

int source =100;

void tAC1()
{
    printf("tAC1 실행\n");
    while(1)
    {
        printf("tAC1 공유자원 접근\n");
        rt_mutex_acquire(&owner,TM_INFINITE);
        printf("tAC1 : 무텍스 획득!\n");
        printf("tAC1 사용후 공유자원의 값 : %d\n",source);
        source-=10;
        rt_mutex_release(&owner);
        printf("tAC1 무텍스 반환..\n");
        sleep(2);
    }
}

void tAC2()
{
    printf("tAC2 실행\n");
    while(1)
    {
        printf("tAC2 공유자원 접근\n");
        rt_mutex_acquire(&owner,TM_INFINITE);
        printf("tAC2 :무텍스 획득!\n");
        printf("tAC2 사용후 공유자원의 값 : %d\n",source);
        source-=30;
        rt_mutex_release(&owner);
        printf("tAC2 무텍스 반환..\n");
        sleep(5);
    }
}
```

```

int main()
{
    if(rt_task_create(&t1_desc, "tAC1", STACK_SIZE, tAC1PRI, TASKMODE))
    {
        printf("tAC1이 생성되지 않았습니다.\n");
        exit(0);
    }

    printf("tAC1 생성완료!\n");

    if(rt_task_create(&t2_desc, "tAC2", STACK_SIZE, tAC2PRI, TASKMODE))
    {
        printf("tAC2가 생성되지 않았습니다.\n");
        exit(0);
    }

    printf("tAC2 생성완료!\n");

    if(rt_mutex_create(&owner, "key"))
    {
        printf("뮤텍스가 생성되지 않았습니다.\n");
        exit(0);
    }

    printf("뮤텍스가 생성 완료!\n");

    rt_task_start(&t1_desc, &tAC1, NULL);
    rt_task_start(&t2_desc, &tAC2, NULL);
    pause();
    rt_mutex_delete(&owner);

    return 0;
}

```

동기화를 확실히 하기 위해서는 소유권을 부여하는 뮤텍스를 사용하는 것이 좋다. 뮤텍스의 개수는 공유자원의 개수만큼 부여한다. 카운팅 세마포어와 동일하게 전역변수로 공유자원을 선언하고 tAC1, tAC2는 순서대로 -10, -30의 연산을 진행한다. 뮤텍스를 반환하기 전까지, 다른 태스크들은 공유자원에 진입할 수 없다, 그러므로 세마포어를 사용했을 때 발생할 수 있는 동기화의 문제를 해결할 수 있다, 그러면 프로그램의 실행결과를 보자.



```
db@ubuntu:~/202000969$ sudo ./danilmu
tAC1 생성완료!
tAC2 생성완료!
뮤텍스가 생성 완료!
tAC1 실행
tAC1 공유자원 접근
tAC1뮤텍스 획득 기다리는 중...
tAC1 : 뮤텍스 획득!
tAC1 사용후 공유자원의 값 : 100
사용시작!
tAC1 뮤텍스 반환..!
tAC2 실행
tAC2 공유자원 접근
tAC2뮤텍스 획득 기다리는 중...
tAC2 :뮤텍스 획득!
tAC2 사용후 공유자원의 값 : 90
사용시작!
tAC2 뮤텍스 반환..!
tAC1 공유자원 접근
tAC1뮤텍스 획득 기다리는 중...
tAC1 : 뮤텍스 획득!
tAC1 사용후 공유자원의 값 : 60
사용시작!
tAC1 뮤텍스 반환..!
tAC1 공유자원 접근
tAC1뮤텍스 획득 기다리는 중...
tAC1 : 뮤텍스 획득!
tAC1 사용후 공유자원의 값 : 50
사용시작!
tAC1 뮤텍스 반환..!
```

뮤텍스 소유권을 잘 확하기 위해서 tAC1은 sleep(2)를 tAC2는 sleep(5)를 주었다. 태스크의 실행 사이클이 차이가 나면서, 공유자원을 들어가기 위해 뮤텍스를 얻고 반환해야 들어갈 수 있는 것을 볼 수 있다.

#### 2.5. 6.4.5 재귀적 공유자원 접근 동기화

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/mman.h>
#include<native/task.h>
#include<native/mutex.h>

#define STACK_SIZE 0
#define TASKMODE 0
#define tA_PRI 80
#define Sem_MODE SEMAPHORE_PRIO

RT_TASK tA_desc;
RT_MUTEX owner;

int resource=100;

int routine_a()
{
    printf("RA 루틴 접근 성공! 공유자원이 필요하다.\n");
    printf("RA 뮉텍스 획득 준비중...\n");
    rt_mutex_acquire(&owner,TM_INFINITE);
    printf("RA 뮉텍스 획득! 루틴 시작..\n공유자원 접근하겠다!\n");
    printf("현재 공유자원의 값 : %d\n",resource);
    resource+=20;
    printf("루틴 B 호출\n");
    routine_b();
    printf("RA 뮉텍스 반환..!\n");
    rt_mutex_release(&owner);
    printf("뮉텍스 반환~ RA 종료합니다~\n");

    return 0;
}
```

```

int routine_b()
{
    printf("RB 루틴 접근 성공! 공유자원이 필요하다.\n");
    printf("RB 뮅텍스 획득 준비중...\n");
    rt_mutex_acquire(&owner, TM_INFINITE);
    printf("RB 뮅텍스 획득! 루틴 시작..\n공유자원 접근하겠다!\n");
    printf("현재 공유자원의 값 : %d\n", resource);
    resource+=20;
    printf("RB 뮅텍스 반환..!\n");
    rt_mutex_release(&owner);
    printf("뮅텍스 반환~ RB 종료합니다~\n");

    return 0;
}

void tA()
{
    printf("tA 시작!\n");
    printf("tA 뮅텍스 획득 준비중..\n");
    rt_mutex_acquire(&owner, TM_INFINITE);
    printf("tA 뮅텍스 획득! 루틴 시작..\n공유자원 접근하겠다!\n");
    printf("현재 공유자원의 값 : %d\n", resource);
    resource+=20;
    printf("루틴 A 호출\n");
    routine_a();
    printf("tA 뮅텍스 반환..!\n");
    rt_mutex_release(&owner);
    printf("뮅텍스 반환~ 프로그램 종료하겠습니다\n");
}

int main()
{
    if(rt_task_create(&tA_desc, "tA", STACK_SIZE, tA_PRI, TASKMODE))
    {
        printf("tA가 생성되지 않았습니다.\n");
        return -1;
    }
    printf("tA가 생성되었습니다!\n");

    if(rt_mutex_create(&owner, "KEY"))
    {
        printf("뮅텍스가 생성되지 않았습니다.\n");
        return -1;
    }
    printf("뮅텍스가 생성되었습니다!\n");

    rt_task_start(&tA_desc, &tA, NULL);
    pause();
    rt_mutex_delete(&owner);

    return 0;
}

```

---

tA안에 routine\_a함수를 선언하였고 routin\_a함수에는 routine\_b를 선언해주었다. 모든 루틴들이 공유자원에 접근하기 위해 뮤텁스를 요구하며 데드락 상태에 빠지지 않게 뮤텁스를 부여한다. 단, 뮤텁스를 받은 만큼 반환을 해야한다. 공유자원에 접근 했는지, 즉 뮤텁스가 정상적으로 부여되었는지 확인하기 위해 모든 함수에 연산을 넣었고, 한번만 출력하기 위해서 반복문을 사용하지 않았다. 이 코드의 실행결과를 아래에서 보자.

```
db@ubuntu:~/202000969$ sudo ./recursive
[sudo] password for db:
tA가 생성되었습니다!
뮤텁스가 생성되었습니다!
tA 시작!
tA 뮤텁스 획득 준비중..
tA 뮤텁스 획득! 루틴 시작..
공유자원 접근하겠다!
현재 공유자원의 값 : 100
루틴 A 호출
RA 루틴 접근 성공! 공유자원이 필요하다.
RA 뮤텁스 획득 준비중...
RA 뮤텁스 획득! 루틴 시작..
공유자원 접근하겠다!
현재 공유자원의 값 : 120
루틴 B 호출
RB 루틴 접근 성공! 공유자원이 필요하다.
RB 뮤텁스 획득 준비중...
RB 뮤텁스 획득! 루틴 시작..
공유자원 접근하겠다!
현재 공유자원의 값 : 140
RB 뮤텁스 반환..!
뮤텁스 반환~ RB 종료합니다~
RA 뮤텁스 반환..!
뮤텁스 반환~ RA 종료합니다~
tA 뮤텁스 반환..!
뮤텁스 반환~ 프로그램 종료하겠습니다
```

공유자원의 값이 정상적으로 접근하였고 데드락에 빠지지 않도록 뮤텁스를 부여한 것을 볼 수 있다.

## 2.6 6.4.6 복수 공유자원 접근 동기화

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/mman.h>
#include<native/task.h>
#include<native/sem.h>

#define STACK_SIZE 0
#define TASKMODE 0
#define SEMAPHORE_CT 2
#define SEMAPHORE_MODE S_PRIO
#define tAC1PRI 99
#define tAC2PRI 99
#define tAC3PRI 99

int source1 =100;
int source2=100;

RT_TASK tAC1_desc,tAC2_desc,tAC3_desc;
RT_SEM Sem_desc;
|
void tAC1()
{
    printf("tAC1 실행\n");
    while(1)
    {
        printf("tAC1 공유자원 접근\n세마포어 대기중입니다.\n");
        rt_sem_p(&Sem_desc,TM_INFINITE);
        printf("tAC1 : 세마포어 획득!\n");
        printf("사용할 공유자원 = 공유자원 1 공유자원의 값 : %d\n사용시작!\n",source1);
        source1-=10;
        printf("공유자원 접근! 현재 공유자원 1의 값 : %d\n",source1);
        rt_sem_v(&Sem_desc);
        printf("공유자원 사용 완료! tAC1 세마포어 반납!\n");
        sleep(2);
    }
}

void tAC2()
{
    printf("tAC2 실행\n");
    while(1)
    {
        printf("tAC2 공유자원 접근\n세마포어 대기중입니다.\n");
        rt_sem_p(&Sem_desc,TM_INFINITE);
        printf("tAC2 : 세마포어 획득!\n");
        printf("사용할 공유자원 = 공유자원 2 공유자원의 값 : %d\n사용시작!\n",source2);
        source2-=10;
        printf("공유자원 접근! 현재 공유자원 2의 값 : %d\n",source2);
        rt_sem_v(&Sem_desc);
        printf("공유자원 사용 완료! tAC2 세마포어 반납!\n");
        sleep(2);
    }
}

void tAC3()
{
    printf("tAC3 실행\n");
    while(1)
    {
        printf("tAC3 공유자원 접근\n세마포어 대기중입니다.\n");
        rt_sem_p(&Sem_desc,TM_INFINITE);
        printf("tAC3 : 세마포어 획득!\n");
        printf("사용할 공유자원 = 공유자원 1 공유자원의 값 : %d\n사용시작!\n",source1);
        source1+=10;
        printf("공유자원 접근! 현재 공유자원 1의 값 : %d\n",source1);
        rt_sem_v(&Sem_desc);
        printf("공유자원 사용 완료! tAC3 세마포어 반납!\n");
        sleep(2);
    }
}
```

```

int main()
{
    if(rt_task_create(&tAC1_desc, "tAC1", STACK_SIZE, tAC1PRI, TASKMODE))
    {
        printf("tAC1이 생성되지 않았습니다.\n");
        exit(0);
    }

    printf("tAC1 생성완료!\n");

    if(rt_task_create(&tAC2_desc, "tAC2", STACK_SIZE, tAC2PRI, TASKMODE))
    {
        printf("tAC2가 생성되지 않았습니다.\n");
        exit(0);
    }

    printf("tAC2 생성완료!\n");

    if(rt_task_create(&tAC3_desc, "tAC3", STACK_SIZE, tAC3PRI, TASKMODE))
    {
        printf("tAC3이 생성되지 않았습니다.\n");
        exit(0);
    }

    printf("tAC3 생성완료!\n");

    if(rt_sem_create(&Sem_desc, "semaphore", SEMAPHORE_CT, SEMAPHORE_MODE)){
        printf("세마포어가 생성되지 않았습니다.\n");
        exit(0);
    }

    printf("세마포어 생성 완료!\n");

    rt_task_start(&tAC1_desc, &tAC1, NULL);
    rt_task_start(&tAC2_desc, &tAC2, NULL);
    rt_task_start(&tAC3_desc, &tAC3, NULL);
    pause();
    rt_sem_delete(&Sem_desc);

    return 0;
}

```

단일 공유자원과 다르게 같은 공유자원이 여러 개 있는 것을 복수 공유 자원이라고 한다. 그래서 똑같은 값 100으로 초기화된 두개의 공유자원을 선언하였다. tAC1, tAC3은 공유자원1에 접근하도록 하였고, tAC2는 공유자원2에 접근하도록 하였다. 원래 실제 프로그램 같으면 동기화의 문제가 발생할 것이다. 하지만 실습환경에서는 세마포어 반환이 부여된 태스크들에서 일어남으로 정상적으로 동기화가 되고 프로그램이 실행되는 것을 볼 수 있다, 실행 결과를 보도록 하자.



```
tAC1 생성완료!
tAC2 생성완료!
tAC3 생성완료!
세마포어 생성 완료!
tAC1 실행
tAC1 공유자원 접근
세마포어 대기중입니다.
tAC1 : 세마포어 획득!
사용할 공유자원 = 공유자원 1 공유자원의 값 : 100
사용시작!
공유자원 접근! 현재 공유자원 1의 값 : 90
공유자원 사용 완료! tAC1 세마포어 반납!
tAC2 실행
tAC2 공유자원 접근
세마포어 대기중입니다.
tAC2 : 세마포어 획득!
사용할 공유자원 = 공유자원 2 공유자원의 값 : 100
사용시작!
공유자원 접근! 현재 공유자원 2의 값 : 90
공유자원 사용 완료! tAC2 세마포어 반납!
tAC3 실행
tAC3 공유자원 접근
세마포어 대기중입니다.
tAC3 : 세마포어 획득!
사용할 공유자원 = 공유자원 1 공유자원의 값 : 90
사용시작!
공유자원 접근! 현재 공유자원 1의 값 : 100
공유자원 사용 완료! tAC3 세마포어 반납!
tAC1 공유자원 접근
세마포어 대기중입니다.
tAC2 공유자원 접근
세마포어 대기중입니다.
tAC3 공유자원 접근
세마포어 대기중입니다.
```

실행결과를 보았을 때 정상적으로 태스크를 생성하였고, 공유자원에 정상적으로 접근한 것을 볼 수 있다. 외부의 영향이 없어 정상적으로 공유자원의 값이 들어갔지만 외부 프로그램이 있을 때에는 뮤텍스를 사용하는 것이 좋다. 뮤텍스 선언은 공유자원의 개수만큼 선언해주면 된다.