

Xilinx Zynq FPGA, TI DSP, MCU 기반의 프로그래밍 및 회로 설계 전문가 과정

강사 – Innova Lee(이상훈)

gcccompil3r@gmail.com

학생 – hoseong Lee(이호성)

hslee00001@naver.com



네트워크

술게임만들기

시그널 , 메모리, 프로세스 관련 함수 재정리.

*Signal ()

void (*signal(int signum, void (*handler)(int)))(int);

int signum : 시그널 번호 , void (*handler)(int) : 시그널을 처리할 핸들러 , void(*) (int) : 이전에 설정된 시그널 핸들러 , 반환

```
// 아래 예제에서는 while() 문을 이용하여 계속 문자열을 출력하면서
// Ctrl-C 로 SIGINT가 발생하면 바로 종료하는 것이 아니라
// 프로그램에서 작성된 함수를 실행한 후에
// 다시 ctrl-c 키가 눌리면 기존의 방법에 따라
// 프로세스가 종료되는 것을 보여준다.

#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void (*old_fun)( int);

void sigint_handler( int signo)
{
    printf( "ctrl-c 키를 누르셨죠!!\n");
    printf( "또 누르시면 종료됩니다.\n");
    signal( SIGINT, old_fun);    // 또는 signal( SIGINT, SIG_DFL);
}

int main( void)
{
    int i;
    old_fun = signal( SIGINT, sigint_handler);
    while(1 )
    {
        printf( "%d\n",i);
        sleep(1);
        i++;
    }
}
```

```
lee@lee-Lenovo-YOGA-720-13IKB:~/my_proj/lhs/lec/c$ ./a.out
0
1
2
3
4
^Cctrl-c 키를 누르셨죠 !!
또 누르시면 종료됩니다 .
5
6
7
8
^C
```

→ SIGINT :control-c 나 delete 키를 입력했을 때, 발생한다.

369 게임에 적용.

프로세스 함수

***execl**

***execv**

***wait - <sys/wait.h>**

자식 프로세스 작업이 끝날 때까지 대기하며, 자식 프로세스가 종료한 상태를 구함.

pid_t wait(int *status); → status – wait 가 복귀될 때 자식프로세스의 상태 정보를 나타낸다.

***waitpid - <sys/wait.h>**

wait()함수처럼 자식 프로세스가 종료될 때까지 대기함. 차이점은 wait() 함수가 자식 프로세스 중 어느 하나라도 종료되면 복귀되지만, 즉 대기에서 풀리지만 waitpid()는 특정 자식 프로세스가 종료될 때까지 대기. = pid_t waitpid(pid_t pid, int *status, int option)

Pid- 부모가 기다리고 싶은 자식 프로세스의 프로세스 id

status – wait 가 복귀할때 자식프로세스의 상태 정보를 나타냄. // wait 와 같은데, 정상적종료라면 하위 8 비트는 0 상위 8 비트는 프로세스가 종료되게 한 exit 함수의 인수가 기록된다. 비정상적 종료시엔 하위 8 비트에 프로세스를 종료시킨 시그널의 번호, 상위 8 비트엔 0 이 저장된다.

Options - <sys/wait.h> 에 정의 된 여러 값을 취할 수 있음.

ex1)

waitpid(pid,null,0) → pid (이자식만) 종료되면 부모가 깨어남.

waitpid(-1,null,0) → 부모 프로세스가 자식프로세스들 중 하나라도 종료되면 부모 프로세스가 깨어나도록 한다. wait 모드

waitpid(pid,NULL,WNOHANG) → 부모 프로세스가 기다리지 않고도 특정 프로세스가 멈출 때의 상태를 알 수 있다. (논블록킹 처리방식)

→ 성공시 자식프로세스의 아이디를 반환 실패시 -1 리턴

Ex2)

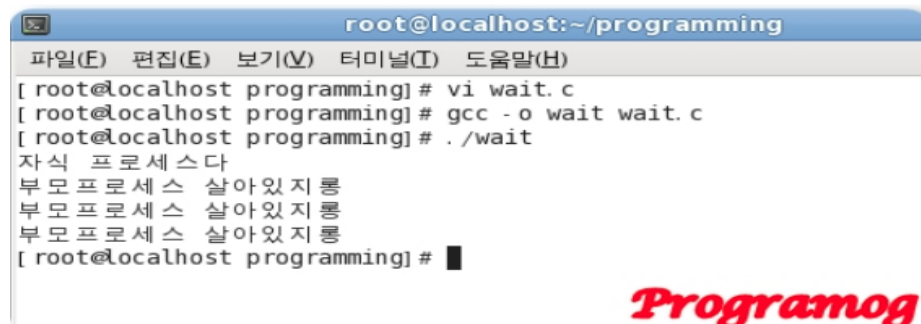
```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    int pid;

    if ((pid = fork()) == -1)
        puts("fork 함수 에러!! \a\n");
    else if (pid != 0)
    {
        while (waitpid(pid, NULL, WNOHANG) == 0)
        {
            puts("부모프로세스 살아있지롱");
            sleep(1);
        }
    }
    else
    {
        puts("자식 프로세스다");
        sleep(3);
        exit(0);
    }
    return 0;
}
```

GCC
Programog

위 - 예제 / 아래 - 실행 결과



```
root@localhost:~/programming
파일(F) 편집(E) 보기(V) 터미널(T) 도움말(H)
[root@localhost programming]# vi wait.c
[root@localhost programming]# gcc -o wait wait.c
[root@localhost programming]# ./wait
자식 프로세스다
부모프로세스 살아있지롱
부모프로세스 살아있지롱
부모프로세스 살아있지롱
[root@localhost programming]#
```

Programog

waitpid 함수 == 0 , 지정한 자식 프로세스가 종료되지 않은 상태에서는 무조건 0 을 리턴한다. (>0 논블록킹 처리방식)
sleep 함수는 해당 프로세스를 몇 턴동안 기다리게 하는 함수이다.

→ wait 함수가 특정 프로세스를 기다린다면, sleep 함수는 몇턴동안 기다리게 한다는 것.

메모리 함수

*memset

void *memset(void *buffer, int c, size_t n); 메모리를 설정하는 함수

입력 매개 변수 리스트

buffer 버퍼, c 설정할 값, n 설정할 바이트 수

반환 값 buffer

배열이나 구조체 등의 메모리의 모든 내용을 0 으로 설정할 때 많이 사용함.

* signal 과 fork 함수

```
#include<stdio.h>
#include<stdlib.h>
#include<signal.h>
#include<unistd.h>
#include<sys/types.h>

#define MAXBUF 8162

void handler(int sig)
{
    pid_t pid;

    if((pid = waitpid( -1, NULL, 0))< 0)
    {
        //waitpid 함수는 블록된 특정 프로세스가 종료할때 까지 현재 프로세스를 대기 시킨다.
        // -1 인자 값으로 임의의 프로세스가 종료할 때까지 부모 프로세스는 대기한다.
        // waitpid 함수를 통해 부모프로세스는 종료되는 자식프로세스의 pid를 확인한다.

        printf("waitpid error\n");
        exit(0);
    }
    printf("Handler reaped child %d\n", (int)pid);
    sleep(2);
}

int main()
{
    int i, n;
    char buf[MAXBUF];

    if(signal(SIGCHLD, handler)==SIG_ERR)
    {
        printf("signal error");
        exit(0);
    }

    for( i=0; i<3; i++){
        if( fork() == 0) {
            printf("Hello from child %d\n",(int)getpid());
            sleep(1);
            exit(0);
        }
    }

    if(( n= read(STDIN_FILENO, buf, sizeof(buf))) <0)
    {
        printf("read error");
        exit(0);
    }

    printf("Parent processing input\n");
    while(1){}

    exit(0);
}
```

프로세스 – 독립적
Thread – 종속적

fork 함수는 자원효율성의 몇가지 문제점을 가지고 있다. 프로세스는 기본적으로 code, data, stack, fill I/O, 그리고 signal table 의 5 가지 요소로 구성 된다. fork()를 이용해서 새로운 프로세스를 생성하게 되면, 이러한 5 가지 구성요소가 모두 복사된다.

fork 의 이러한 방식은 상당히 효율이 떨어지는 측면이 있다. 어떤 프로그램을 병렬로 실행시킨다고 했을 때, 실제 우리가 병렬로 실행되기를 원하는 영역은 코드의 일부분이지 프로그램 전체는 아니기 때문이다.

```
// ....
pid =fork();
if(pid>0)
{
    //실제는 이 부분의 코드만 병렬로 실행되면 된다.
    //fork()는 다른 모든영역의 코드가 복사되어 버린다.
}
```

게다가 전혀 다른 프로세스를 생성시킴으로써, 프로세스간 통신이라는 상당히 복잡한 문제까지 해결해야 한다. 병렬로 작동하는 프로그램은 특성상 데이터를 공유하거나 서로 통신을 해야하는 경우가 많다. 그런데 프로세스는 서로 독립된 객체이므로 일반적인 방법으로는 데이터를 공유할 수가 없다. 이러한 프로세스간 데이터 통신을 위해서 리눅스는 IPC(:12)라는 설비를 제공한다.

하지만, IPC → 어려움 , thread → 쉽게 사용가능

Thread 는 새로운 프로세스를 생성시키지 않고, 특정 문맥(코드) 만을 병렬로 실행할 수 있도록 허용한다.
또한 같은 프로세스이기 때문에, 데이터를 공유하기 쉽다

Thread vs Process 차이점

- 프로세스는 독립적이다. 쓰레드는 프로세스의 서브셋이다.
- 프로세스는 각각 독립적인 자원을 가진다. 쓰레드는 stat, memory 기타 다른 자원들을 공유한다.
- 프로세스는 자신만의 주소영역을 가진다. 쓰레드는 주소영역을 공유한다.
- 프로세스는 IPC(:12)를 이용해서만 통신이 가능하다.
- 일반적으로 쓰레드의 문맥교환(context switching)는 프로세스의 문맥교환보다 빠르다.

Multi Thread 프로그램의 단점

모든 도구가 그러하듯이 Multi Thread 프로그램이라고 해서 장점만 가진 것은 아니다. Multi Thread 프로그램은 Multi Process 프로그래밍 방식에 비해서 다음과 같은 단점을 가진다.

- 하나의 쓰레드에서 발생된 문제가 전체 프로세스에 영향을 미친다.

멀티 프로세스의 경우에는 프로세스하나가 문제가 생기더라도 단일 프로세스로 문제가 제한된다. 그러나 멀티쓰레드 프로그램의 경우 하나의 쓰레드에 생긴 문제가 다른 쓰레드에까지 영향을 줄 수 있다. 예를 들어 쓰레드 하나가 다른 프로세스의 메모리 영역을 침범할 경우 프로세스 자체가 죽어버림으로써, 프로세스에 생성된 다른 모든 쓰레드도 프로세스와 함께 죽어버리게 된다. - 이 문제는 해결 가능하지만 여기에서는 다루지 않도록 하겠다. 시그널(:12)을 잘 활용하면 된다. 관심있으면 한번 고민해 보기 바란다. -

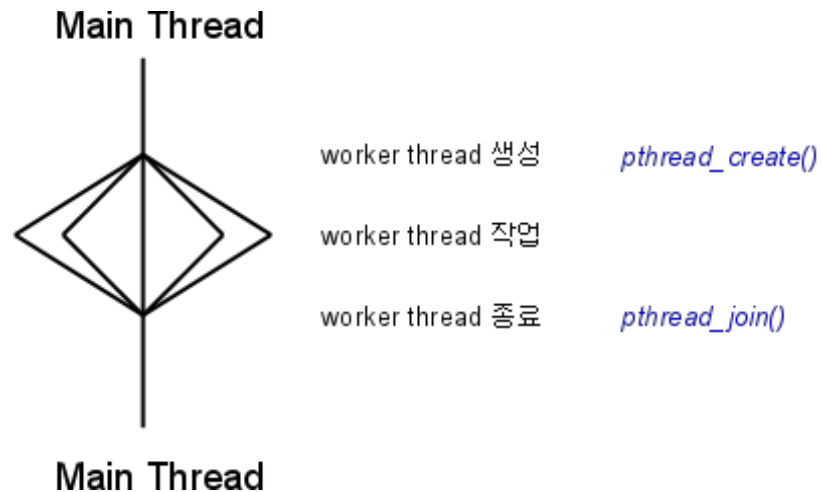
- 디버깅이 어렵다. 문맥이 서로 교환되므로 추적하기가 까다롭다.

이러한 단점이 있음에도 불구하고 멀티쓰레딩 프로그래밍 기법을 선호하고 있다.

쓰레드의 생성과 종료

멀티 쓰레드 프로그램이라고 하더라도, 처음 시작되었을 때는 `main()`에서 시작되는 단일 쓰레드 상태로 작동이 된다. 이 상태에서 `pthread_create(3)` 함수를 호출함으로써, 새로운 쓰레드를 생성할 수 있다. **pthread_create** 를 이용해서 생성된 새로운 쓰레드를 worker 쓰레드라고 하자.

멀티 쓰레드 프로그램은 다음과 같은 흐름을 가진다.2



생성된 worker thread 는 언젠가 종료가 될 것이다. Master Thread (이하 부모 쓰레드)는 `pthread_join()`을 이용해서 worker thread 들의 종료를 기다린다. `pthread_join()`은 종료된 worker thread 의 자원을 정리하는 일을 한다. `fork()`를 이용한 멀티 프로세스 프로그램에서, 부모 프로세스가 `wait()`를 이용해서 자식 프로세스를 기다리는 것과 같은 이유라고 보면 된다.

*pthread_create : 쓰레드 생성

pthread_create(3)함수를 이용하면 새로운 쓰레드를 생성할 수 있다. 이 함수는 다음과 같이 사용할 수 있다

```
#include <pthread.h>

int pthread_create(pthread_t * thread, pthread_attr_t *attr,
    void * (*start_routine)(void *), void * arg);
```

1. **thread** : 쓰레드가 성공적으로 생성되었을 때, 넘겨주는 쓰레드 식별 번호.
2. **attr** : 쓰레드의 특성을 설정하기 위해서 사용한다. NULL(:12)일 경우 기본 특성
3. **start_routine** : 쓰레드가 수행할 함수로 함수포인터(:12)를 넘겨준다.
4. **arg** : 쓰레드 함수 start_routine 를 실행시킬 때, 넘겨줄 인자

이 함수는 성공적으로 수행되었다면, 0 을 리턴한다. 그렇지 않을 경우 1 을 리턴한다.

*pthread_join : 쓰레드 정리

쓰레드가 실행시키는 것은 함수 이다. 그러므로 return 이나 exit(0)등을 이용해서 쓰레드를 종료시킬 수 있게 된다. 그러나 쓰레드 함수가 종료되었다고 해서 곧바로 쓰레드의 모든자원이 종료되지 않는다. fork()기반의 멀티프로세스 프로그램에서 종료된 자식프로세스를 정리하기 위해서 wait()로 기다리듯이, 종료된 쓰레드를 기다려서 정리를 해주어야만 한다. 그렇지 않을 경우 쓰레드의 자원이 되돌려지지 않아서 메모리 누수현상이 발생하게 된다.

pthread_create()로 생성시킨 쓰레드는 pthread_join()을 통해서 기다리면 된다. pthread_join 함수는 다음과 같이 사용할 수 있다.

```
#include <pthread.h>

int pthread_join(pthread_t th, void **thread_return);

|
```

1. **th**: pthread_create 에 의해서 생성된, 식별번호 **th** 를 가진 쓰레드를 기다리겠다는 얘기다.
2. **thread_return**: 식별번호 **th** 인 쓰레드의 종료시 리턴값이다.

쓰레드 생성예제

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

// 쓰레드 함수
void *t_function(void *data)
{
    int id;
    int i = 0;
    id = *((int *)data);

    while(1)
    {
        printf("%d : %d\n", id, i);
        i++;
        sleep(1);
    }
}

int main()
{
    pthread_t p_thread[2];
    int thr_id;
    int status;
    int a = 1;
    int b = 2;

    // 쓰레드 생성 아규먼트로 1 을 넘긴다.
    thr_id = pthread_create(&p_thread[0], NULL, t_function, (void *)&a);
    if (thr_id < 0)
    {
        perror("thread create error : ");
        exit(0);
    }

    // 쓰레드 생성 아규먼트로 2 를 넘긴다.
    thr_id = pthread_create(&p_thread[1], NULL, t_function, (void *)&b);
    if (thr_id < 0)
    {
        perror("thread create error : ");
        exit(0);
    }

    // 쓰레드 종료를 기다린다.
    pthread_join(p_thread[0], (void **)&status);
    pthread_join(p_thread[1], (void **)&status);

    return 0;
}
```

아주 전형적인 프로그램이긴 하지만 **pthread_join** 부분에 문제가 있다. **pthread_join** 은 쓰레드가 종료될 때까지 블럭되기 때문이다. 이래서는 쓰레드를 두개이상 생성시키지 못할 것이다. 그렇다고 pthread_join 을 이용하지 않는다면, 메모리 누수가 생기게 되니, 생략할 수도 없는 노릇이다.

자식쓰레드를 부모쓰레드로 부터 분리하기

→ join 의 사용으로 발생하는 문제점 : 쓰레드가 종료될때 까지 블럭됨. 허나 사용하지않는다면 메모리 낭비가 됨.

pthread_detach 를 이용해서, 자식 쓰레드를 부모쓰레드와 완전히 분리해 버리는 방법이다. 이 경우 자식 쓰레드가 종료되면, 모든 자원이 즉시 반환된다. 반면, 자식 쓰레드의 종료상태를 알 수 없다는 문제가 발생한다. 대개의 경우 자식 쓰레드의 종료상태가 중요한 문제가 되지는 않을 것이다.

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

// 쓰레드 함수
// 1초를 기다린후 아규먼트^2 을 리턴한다.
void *t_function(void *data)
{
    char a[100000];
    int num = *((int *)data);
    printf("Thread Start\n");
    sleep(5);
    printf("Thread end\n");
}

int main()
{
    pthread_t p_thread;
    int thr_id;
    int status;
    int a = 100;

    printf("Before Thread\n");
    thr_id = pthread_create(&p_thread, NULL, t_function, (void *)&a);
    if (thr_id < 0)
    {
        perror("thread create error : ");
        exit(0);
    }

    // 식별번호 p_thread 를 가지는 쓰레드를 detach
    // 시켜준다.
    pthread_detach(p_thread);
    pause();
    return 0;
}
```

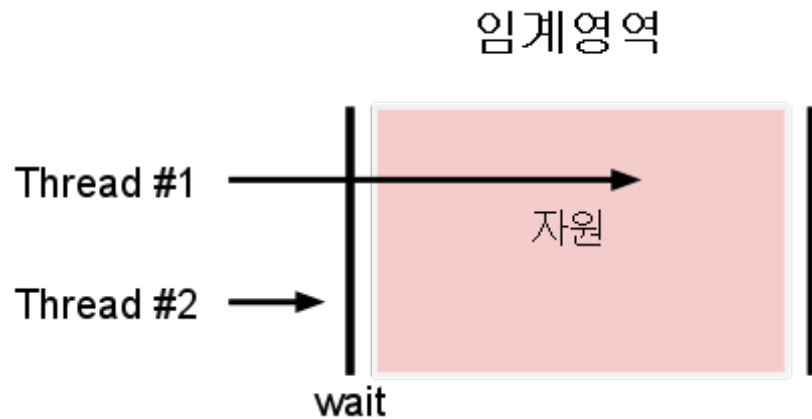
접근제어

동기화 문제는 현실세계에서도 자주 발생한다. 화장실을 생각하면 된다. 화장실은 공유자원이며, 여러명이 사용한다. 누군가 화장실을 사용하고 있다면, 다른 사람은 화장실을 사용하면 안된다. 이 문제를 우리는 **접근을 제어**하는 방식으로 해결한다. 문을 걸어 잠궈서 한번에 한사람만 화장실에 들어가도록 하는 방법이다. 매우 이해하기 쉬운 방식이다.

다중쓰레드 프로그램에서도 마찬가지로 **접근제어**를 이용해서 이 문제를 해결한다. 이를 위해서 pthread는 mutex(:12)라는 **잠금 메커니즘**을 제공한다.

mutex 잠금

동시에 여러개의 쓰레드가 하나의 자원에 접근하려고 할때 발생하는 문제를 pthread는 **임계영역**을 두는 것으로 해결하고 있다. **임계영역** 안에는 접근하고자 하는 **자원**이 놓여있고, 오직 하나의 쓰레드만 임계영역안으로 진입할 수 있도록 제한한다. pthread는 이를 위해서 **mutex**를 제공한다. mutex는 그 자체가 가지는 **잠금**의 특성 때문에 **mutex 잠금**이라고 불리워지기도 한다.



위 그림은 mutex가 작동하는 방식을 보여준다. thread 1이 자원에 접근하면 mutex 잠금을 얻게 된다. 이 잠금은 단지 하나만 존재하기 때문에 thread 2는 잠금을 얻지 못하고 임계영역 밖에서 대기하게 된다. thread 1이 자원을 모두 사용하고 임계영역을 벗어나면 thread 2는 잠금을 얻게 되고 임계영역에 진입해서 자원을 사용할 수 있게 된다.

mutex 의 사용

mutex 를 사용하기 위해서는 다음의 4 가지 함수가 필요하다.

- mutex 잠금객체를 만드는 함수 - pthread_mutex_init
- mutex 잠금을 얻는 함수 - pthread_mutex_lock
- mutex 잠금을 되돌려주는 함수 - pthread_mutex_unlock
- mutex 잠금객체를 제거하는 함수

pthread_mutex_init

mutex 를 사용하기 위해서는 먼저 pthread_mutex_init() 함수를 이용해서, mutex 잠금 객체를 만들어줘야 한다.
이 함수는 두개의 인자를 필요로 한다.

1.mutex : mutex 잠금객체

2.mutex_attr : mutex 는 **fast**, recursive, **error checking** 의 3 종류가 있다. 이 값을 이용해서 mutex 타입을 결정할 수 있다. NULL 일 경우 기본값이 **fast** 가 설정된다.

- fast : 하나의 스레드가 하나의 잠금만을 얻을 수 있는 일반적인 형태
- recursive : 잠금을 얻은 스레드가 다시 잠금을 얻을 수 있다. 이 경우 잠금에 대한 카운드가 증가하게 된다.

mutex_attr 을 위해서 다음의 상수값이 예약되어 있다.

- fast : PTHREAD_MUTEX_INITIALIZER
- recursive : PTHREAD_RECURSIVE_MUTEX_INITIALIZER
- error checking : PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP

pthread_mutex_lock

mutex 잠금을 얻기 위한 함수다.

mutex 잠금을 얻는다라는 표현보다는 **mutex 잠금을 요청한다**라는 표현이 더 정확할 것 같다. 만약 mutex 잠금을 선점한 스레드가 있다면, 선점한 스레드가 mutex 잠금을 되돌려주기 전까지 이 코드에서 대기하게 된다.

때때로 잠금을 얻을 수 있는지만 체크하고 대기(블록)되지 않은 상태로 다음 코드로 넘어가야할 필요가 있을 수 있을 것이다. 이 경우에는 아래의 함수를 사용하면 된다.

pthread_mutex_unlock

mutex 잠금을 되돌려주는 함수다.

mutex 잠금 예제

count 프로그램을 예제로 할 것이다. 임계영역안에서 보호되어야할 자원은 **count** 이고, 여러개의 스레드가 count 에 접근해서 **+1** 을 시도하려고 한다. 이때 제대로된 count 를 위해서는 한번에 하나의 스레드만이 counting 을 하도록 해야할 것이다. mutex 를 이용해서 임계영역을 보호하도록 할 것이다.

ex

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

int ncount;    // 스레드간 공유되는 자원
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // 스레드 초기화

// 스레드 함수 1
void* do_loop(void *data)
{
    int i;

    pthread_mutex_lock(&mutex); // 잠금을 생성한다.
    for (i = 0; i < 10; i++)
    {
        printf("loop1 : %d", ncount);
        ncount++;
        sleep(1);
    }
    pthread_mutex_unlock(&mutex); // 잠금을 해제한다.
}

// 스레드 함수 2
void* do_loop2(void *data)
{
    int i;

    // 잠금을 얻으려고 하지만 do_loop 에서 이미 잠금을
    // 얻었음으로 잠금이 해제될때까지 기다린다.
    pthread_mutex_lock(&mutex); // 잠금을 생성한다.e
    for (i = 0; i < 10; i++)
    {
        printf("loop2 : %d", ncount);
        ncount++;
        sleep(1);
    }
    pthread_mutex_unlock(&mutex); // 잠금을 해제한다.
}

int main()
{
    int thr_id;
    pthread_t p_thread[2];
    int status;
    int a = 1;

    ncount = 0;
    thr_id = pthread_create(&p_thread[0], NULL, do_loop, (void *)&a);
    sleep(1);
    thr_id = pthread_create(&p_thread[1], NULL, do_loop2, (void *)&a);

    pthread_join(p_thread[0], (void *)&status);
    pthread_join(p_thread[1], (void *)&status);

    status = pthread_mutex_destroy(&mutex);
    printf("code = %d", status);
    printf("programing is end");
    return 0;
}
```