



# 윤성우의 열혈 TCP/IP 소켓 프로그래밍

윤성우 저 열혈강의 TCP/IP 소켓 프로그래밍 개정판

## Chapter 18. 멀티쓰레드 기반의 서버구현



## Chapter 18-1. 스레드의 이론적 이해

윤성우 저 열혈강의 TCP/IP 소켓 프로그래밍 개정판

# 쓰레드의 등장배경

## 프로세스는 부담스럽다.

- 프로세스의 생성에는 많은 리소스가 소모된다.
- 일단 프로세스가 생성되면, 프로세스간의 컨텍스트 스위칭으로 인해서 성능이 저하된다.
- 컨텍스트 스위칭은 프로세스의 정보를 하드디스크에 저장 및 복원하는 일이다.

## 데이터의 교환이 어렵다.

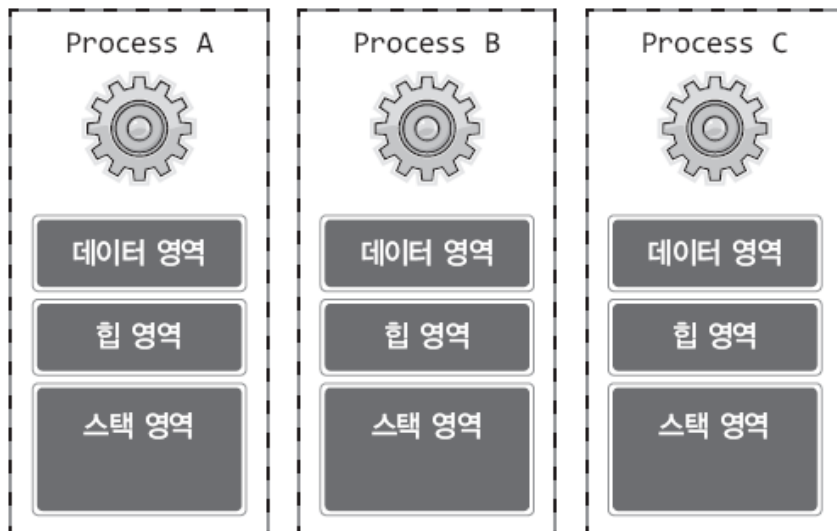
- 프로세스간 메모리가 독립적으로 운영되기 때문에 프로세스간 데이터 공유 불가능!
- 따라서 운영체제가 별도로 제공하는 메모리 공간을 대상으로 별도의 IPC 기법 적용

## 그렇다면 쓰레드는?

- 프로세스보다 가벼운, 경량화된 프로세스이다. 때문에 컨텍스트 스위칭이 빠르다.
- 쓰레드 별로 메모리 공유가 가능하기 때문에 별도의 IPC 기법 불필요
- 프로세스 내에서의 프로그램의 흐름을 추가한다.

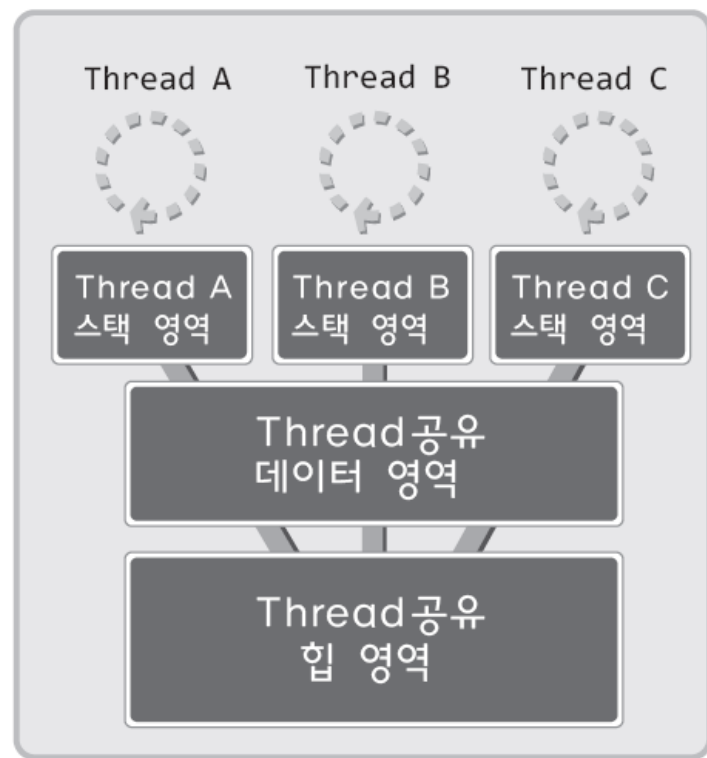


# 쓰레드와 프로세스의 차이점



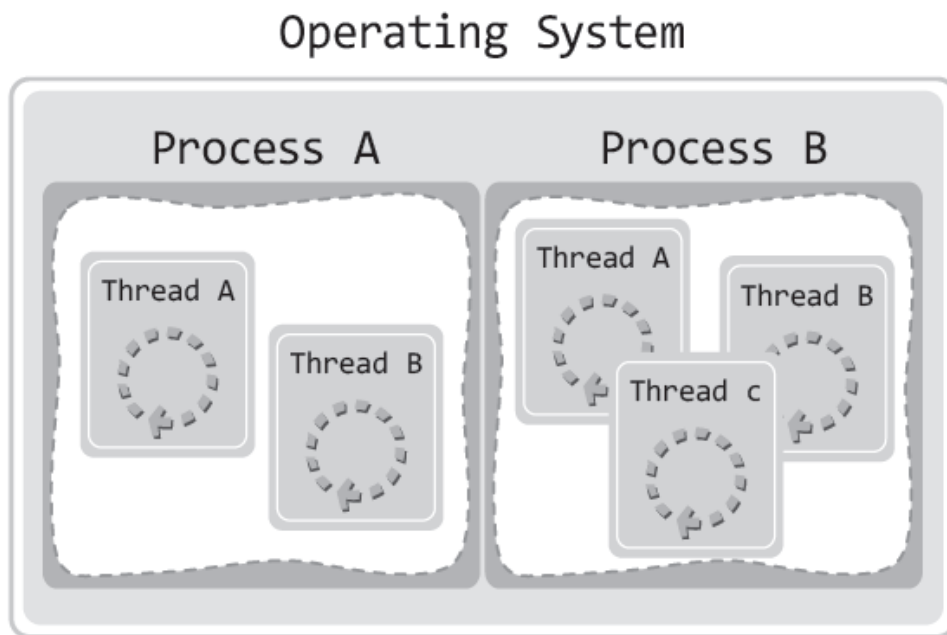
왼쪽 그림에서 보이듯이 프로세스는 서로 완전히 독립적이다. 프로세스는 운영체제 관점에서의 실행흐름을 구성한다.

## 하나의 프로세스

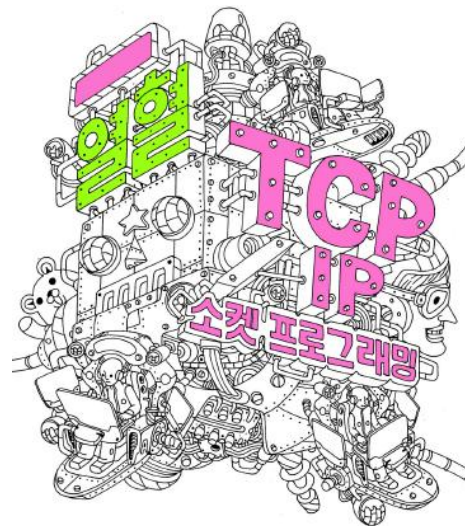


오른쪽 그림에서 보이듯이 **쓰레드는 프로세스 내에서의 실행흐름을 갖는다**. 그리고 데이터 영역과 힙 영역을 공유하기 때문에 컨텍스트 스위칭에 대한 부담이 덜하다. 또한 공유하는 메모리 영역으로 인해서 쓰레드간 데이터 교환이 매우 쉽게 이뤄진다.

# 운영체제와 프로세스, 스레드의 관계



하나의 운영체제 내에서는 둘 이상의 프로세스가 생성되고, 하나의 프로세스 내에서는 둘 이상의 스레드가 생성된다.



## Chapter 18-2. 쓰레드의 생성 및 실행

윤성우 저 열혈강의 TCP/IP 소켓 프로그래밍 개정판

# 쓰레드 생성에 사용되는 함수

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

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

➔ 성공 시 0, 실패 시 0 이외의 값 반환

- thread      생성할 쓰레드의 ID 저장을 위한 변수의 주소 값 전달, 참고로 쓰레드는 프로세스와 마찬가지로 쓰레드의 구분을 위한 ID가 부여된다.
- attr      쓰레드에 부여할 특성 정보의 전달을 위한 매개변수, NULL 전달 시 기본적인 특성의 쓰레드가 생성된다.
- start\_routine      쓰레드의 main 함수 역할을 하는, 별도 실행흐름의 시작이 되는 함수의 주소 값(함수 포인터) 전달.
- arg      세 번째 인자를 통해 등록된 함수가 호출될 때 전달할 인자의 정보를 담고 있는 변수의 주소 값 전달.



# 쓰레드 생성의 예

```
int main(int argc, char *argv[])
{
    pthread_t t_id;
    int thread_param=5;

    if(pthread_create(&t_id, NULL, thread_main, (void*)&thread_param)!=0)
    {
        puts("pthread_create() error");
        return -1;
    };
    sleep(10); puts("end of main");
    return 0;
}
```

예제 **thread1.c**

프로세스의 종료를 막기 위해서  
sleep 함수 호출을 통해서 10초간  
main 함수의 실행을 지연시키고  
있다.

thread\_main 함수가 쓰레드의 main 함수이다. 따라서 이를 가리켜 쓰레드 함수라 한다.

쓰레드가 생성되면 생성된 쓰레드는 쓰레드 함수를 실행한다.

쓰레드 함수의 실행이 완료되면 쓰레드는 종료된다.

```
void* thread_main(void *arg)
{
    int i;
    int cnt=*((int*)arg);
    for(i=0; i<cnt; i++)
    {
        sleep(1); puts("running thread");
    }
    return NULL;
}
```

```
root@my_linux:/tcpip# gcc thread1.c -o tr1 -lpthread
root@my_linux:/tcpip# ./tr1
running thread
running thread
running thread
running thread
running thread
end of main
```

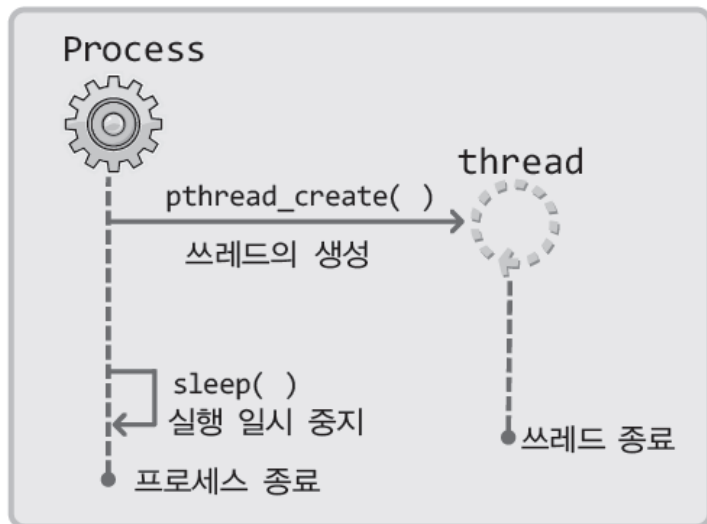
-lpthread 옵션 추가하여 쓰레드 라이브러리 링크 별도 지시

실행 결과



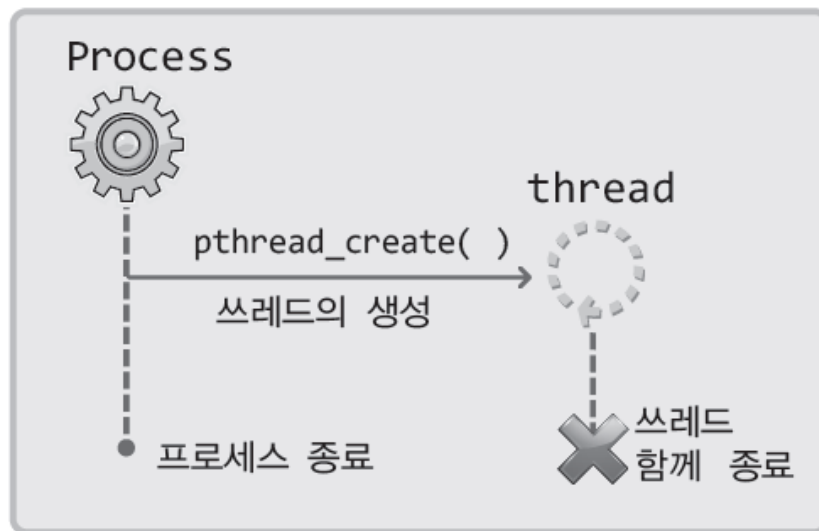
# 프로세스의 종료와 쓰레드

## 쓰레드의 생성



sleep 함수의 호출을 통해서 프로그램의 흐름을 관리하는 데는 한계가 있다.

## 쓰레드의 소멸



프로세스가 종료되면, 해당 프로세스 내에서 생성된 쓰레드도 함께 소멸된다.

# 쓰레드의 종료를 대기

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

```
int pthread_join(pthread_t thread, void **status);
```

➔ 성공 시 0, 실패 시 0 이외의 값 반환

- thread 이 매개변수에 전달되는 ID의 쓰레드가 종료될 때까지 함수는 반환하지 않는다.
- status 쓰레드의 main 함수가 반환하는 값이 저장될 포인터 변수의 주소 값을 전달한다.

첫 번째 인자로 전달되는 ID의 쓰레드가 종료될 때까지, 이 함수를 호출한 프로세스(또는 쓰레드)를 대기상태에 둔다.

# pthread\_join 함수의 호출 예

```
int main(int argc, char *argv[])
{
    pthread_t t_id;
    int thread_param=5;
    void * thr_ret;

    if(pthread_create(&t_id, NULL, thread_main, (void*)&thread_param)!=0)
    {
        puts("pthread_create() error");
        return -1;
    };

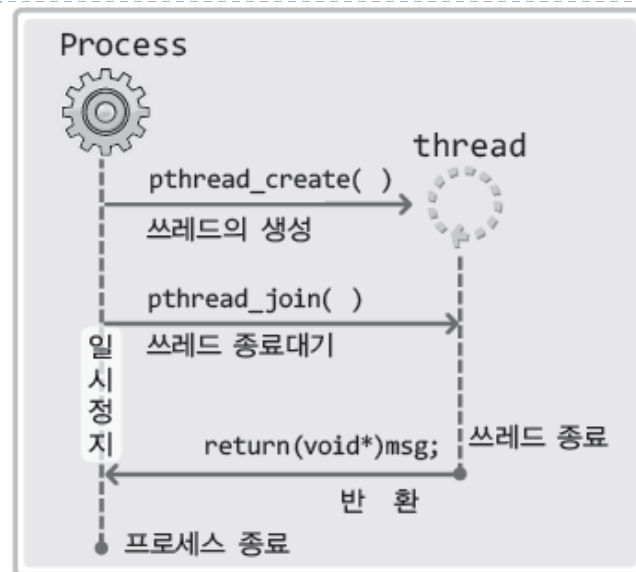
    if(pthread_join(t_id, &thr_ret)!=0)
    {
        puts("pthread_join() error");
        return -1;
    };

    printf("Thread return message: %s \n", (char*)thr_ret);
    free(thr_ret);
    return 0;
}
```

## 예제 thread2.c

```
root@my_linux:/tcip# gcc thread2.c -o tr2 -lpthread
root@my_linux:/tcip# ./tr2
running thread
running thread
running thread
running thread
running thread
Thread return message: Hello, I'am thread~
```

## 실행 결과



```
void* thread_main(void *arg)
{
    int i;
    int cnt=*((int*)arg);
    char * msg=(char *)malloc(sizeof(char)*50);
    strcpy(msg, "Hello, I'am thread~ \n");

    for(i=0; i<cnt; i++)
    {
        sleep(1); puts("running thread");
    }

    return (void*)msg;
}
```

# 임계영역 내에서 호출이 가능한 함수

## 쓰레드에 안전한 함수, 쓰레드에 불안정한 함수

- 둘 이상의 쓰레드가 동시에 호출하면 문제를 일으키는 함수를 가리켜 **쓰레드에 불안정한 함수 (Thread-safe function)**라 한다.
- 둘 이상의 쓰레드가 동시에 호출을 해도 문제를 일으키지 않는 함수를 가리켜 **쓰레드에 안전한 함수 (Thread-unsafe function)**라 한다.

## 쓰레드에 안전한 함수의 예

```
struct hostent * gethostbyname(const char * hostname); 불안전

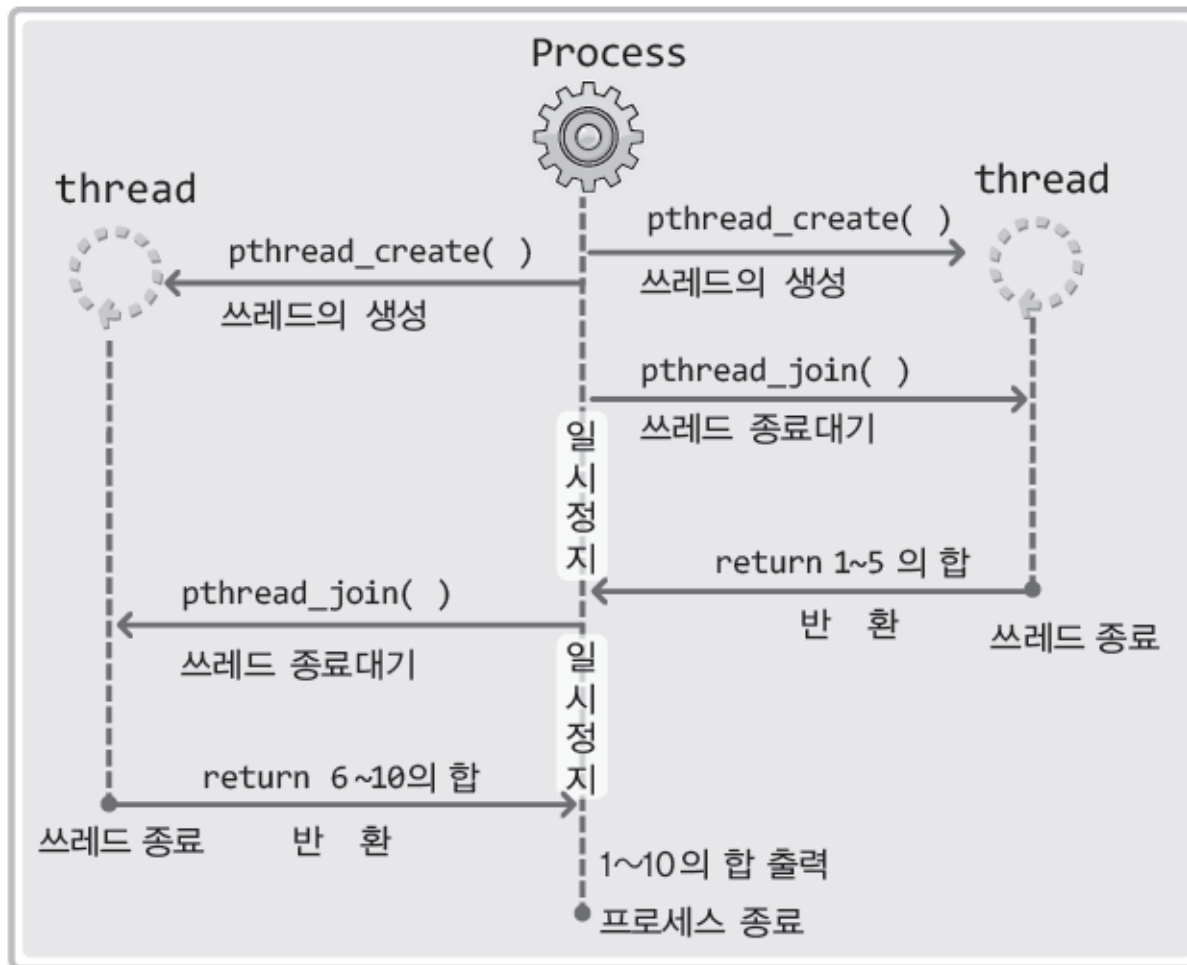
struct hostent *gethostbyname_r(
    const char *name, struct hostent *result, char *buffer, intbuflen, int *h_errnop); 안전
```

헤더파일 선언 이전에 매크로 `_REENTRANT`를 정의하면, 쓰레드에 불안정한 함수의 호출문을 쓰레드에 안전한 함수의 호출문으로 자동 변경 컴파일 된다.

```
root@my_linux:/tcpip# gcc -D_REENTRANT mythread.c -o mthread -lpthread
```



# 워커(Worker) 쓰레드 모델



쓰레드에게 일을 시키고  
그 결과를 취합하는 형  
태의 쓰레드 구성 모델

# 워커(Worker) 쓰레드 모델의 예

```
int sum=0;

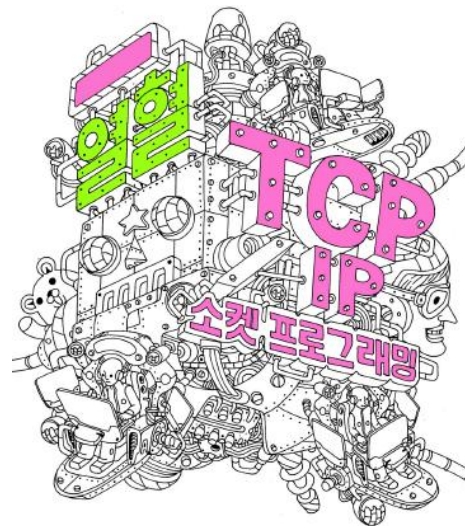
int main(int argc, char *argv[])
{
    pthread_t id_t1, id_t2;
    int range1[]={1, 5};
    int range2[]={6, 10};
    pthread_create(&id_t1, NULL, thread_summation, (void *)range1);
    pthread_create(&id_t2, NULL, thread_summation, (void *)range2);
    pthread_join(id_t1, NULL);
    pthread_join(id_t2, NULL);
    printf("result: %d \n", sum);
    return 0;
}
```

```
void * thread_summation(void * arg)
{
    int start=((int*)arg)[0];
    int end=((int*)arg)[1];
    while(start<=end)
    {
        sum+=start;
        start++;
    }
    return NULL;
}
```

```
root@my_linux:/tcpip# gcc thread3.c -D_REENTRANT -o tr3 -lpthread
root@my_linux:/tcpip# ./tr3
result: 55
```

실행 결과

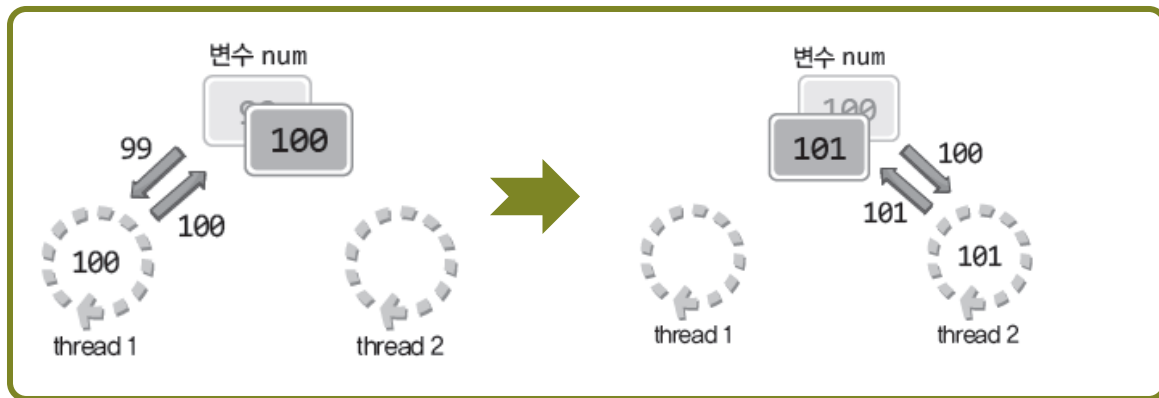
위의 실행결과에는 이상이 없지만, 둘 이상의 쓰레드가 전역변수 sum에 동시에 접근하기 때문에 문제의 발생소지를 지니고 있는 상황이다.



## Chapter 18-3. 스레드의 문제점과 임계영역

윤성우 저 열혈강의 TCP/IP 소켓 프로그래밍 개정판

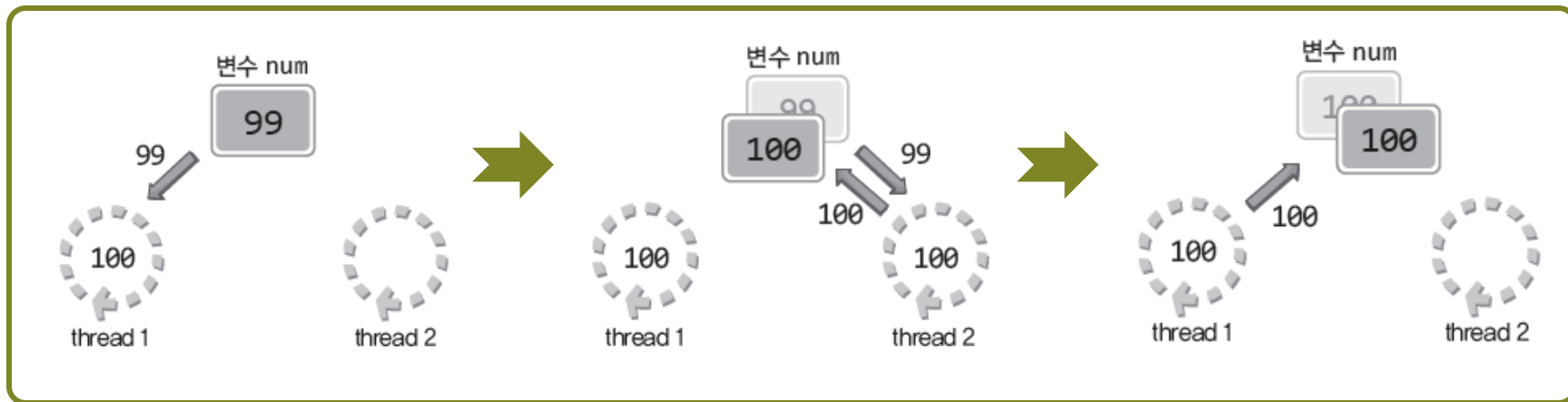
# 둘 이상의 스레드 동시접근의 문제점



정상적인 접근의 예

순차적으로 변수 num에 접근하면 문제가 발생하지 않는다.

잘못된 접근의 예



thread1과 thread2가 각각 1씩 증가시켰는데, 변수 num의 값은 1만 증가하였다.



# 임계영역은 어디?

## 두 개의 쓰레드 함수

```
void * thread_inc(void * arg)
{
    int i;
    for(i=0; i<500000000; i++)
        num+=1;    // 임계영역
    return NULL;
}

void * thread_des(void * arg)
{
    int i;
    for(i=0; i<500000000; i++)
        num-=1;    // 임계영역
    return NULL;
}
```

**임계영역**은 둘 이상의 쓰레드가 동시에 실행하면 문제를 일으키는 영역이다. 왼쪽에서 보이는 바와 같이, 서로 다른 문장임에도 불구하고 동시에 실행이 되는 상황에서도 문제는 발생할 수 있기 때문에 임계영역은 다양하게 구성이 된다.



## Chapter 18-4. 쓰레드의 동기화

윤성우 저 열혈강의 TCP/IP 소켓 프로그래밍 개정판

# 동기화의 두 가지 측면과 동기화 기법

## 동기화가 필요한 대표적인 상황

- 동일한 메모리 영역으로의 동시접근이 발생하는 상황
- 동일한 메모리 영역에 접근하는 스레드의 실행순서를 지정해야 하는 상황

즉, 동기화를 통해서 동시접근을 막을 수 있고, 게다가 접근의 순서를 지정하는 것도 가능하다.

## 동기화 기법

- 뮉텍스(Mutex) 기반 동기화
- 세마포어(Semaphore) 기반 동기화

동기화는 운영체제가 제공하는 기능이기 때문에 운영체제에 따라서 제공되는 기법 및 적용의 방법에 차이가 있다.

# 무텍스 기반의 동기화

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

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

→ 성공 시 0, 실패 시 0 이외의 값 반환

**무텍스의 생성과 소멸**

- mutex    무텍스 생성시에는 무텍스의 참조 값 저장을 위한 변수의 주소 값 전달, 그리고 무텍스 소멸 시에는 소멸하고자 하는 무텍스의 참조 값을 저장하고 있는 변수의 주소 값 전달.
- attr    생성하는 무텍스의 특성정보를 담고 있는 변수의 주소 값 전달, 별도의 특성을 지정하지 않을 경우에는 NULL 전달.

**무텍스의 획득과 반환**

```
pthread_mutex_lock(&mutex);
// 임계영역의 시작
// . . . . .
// 임계영역의 끝
pthread_mutex_unlock(&mutex);
```

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

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

→ 성공 시 0, 실패 시 0 이외의 값 반환

**무텍스 기반 동기화의 기본구성**



# mutex 기반의 동기화의 예

## 예제 mutex.c

```
int main(int argc, char *argv[])
{
    pthread_t thread_id[NUM_THREAD];
    int i;
    pthread_mutex_init(&mutex, NULL);
    for(i=0; i<NUM_THREAD; i++)
    {
        if(i%2)
            pthread_create(&(thread_id[i]), NULL, thread_inc, NULL);
        else
            pthread_create(&(thread_id[i]), NULL, thread_des, NULL);
    }
    for(i=0; i<NUM_THREAD; i++)
        pthread_join(thread_id[i], NULL);

    printf("result: %lld \n", num);
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

```
root@my_linux:/tcip# gcc mutex.c -D_REENTRANT -o mutex -lpthread
root@my_linux:/tcip# ./mutex
result: 0
```

## 실행 결과

```
void * thread_inc(void * arg)
{
    int i;
    pthread_mutex_lock(&mutex);
    for(i=0; i<50000000; i++)
        num+=1;
    pthread_mutex_unlock(&mutex);
    return NULL;
}

void * thread_des(void * arg)
{
    int i;
    for(i=0; i<50000000; i++)
    {
        pthread_mutex_lock(&mutex);
        num-=1;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

mutex의 lock과 unlock의 함수호출 횟수는 최소화 하는게 성능에 유리하다.

# 세마포어(Semaphore)

## 세마포어의 생성과 소멸

```
#include <semaphore.h>
```

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

```
int sem_destroy(sem_t *sem);
```

→ 성공 시 0, 실패 시 0 이외의 값 반환

- sem 세마포어 생성시에는 세마포어의 참조 값 저장을 위한 변수의 주소 값 전달, 그리고 세마포어 소멸 시에는 소멸하고자 하는 세마포어의 참조 값을 저장하고 있는 변수의 주소 값 전달.
- pshared 0 이외의 값 전달 시, 둘 이상의 프로세스에 의해 접근 가능한 세마포어 생성, 0 전달 시 하나의 프로세스 내에서만 접근 가능한 세마포어 생성, 우리는 하나의 프로세스 내에 존재하는 스레드의 동기화가 목적이므로 0을 전달한다.
- value 생성되는 세마포어의 초기 값 지정.

세마포어는 세마포어 카운트 값을 통해서 임계영역에 동시접근 가능한 스레드의 수를 제한할 수 있다.

세마포어 카운트가 0이면 진입불가, 0보다 크면 진입가능

```
#include <semaphore.h>
```

```
int sem_post(sem_t *sem);
```

```
int sem_wait(sem_t *sem);
```

→ 성공 시 0, 실패 시 0 이외의 값 반환

- sem 세마포어의 참조 값을 저장하고 있는 변수의 주소 값 전달, sem\_post에 전달되면 세마포어의 값은 하나 증가, sem\_wait에 전달되면 세마포어의 값은 하나 감소.

## 세마포어의 획득과 반환


```
sem_wait(&sem); // 세마포어 값을 0으로...
// 임계영역의 시작
// . . . . .
// 임계영역의 끝
sem_post(&sem); // 세마포어 값을 1로...
```

세마포어 동기화의 기본구성

# 세마포어 기반 동기화의 예

```
void * read(void * arg)
{
    int i;
    for(i=0; i<5; i++)
    {
        fputs("Input num: ", stdout);
        sem_wait(&sem_two);
        scanf("%d", &num);
        sem_post(&sem_one);
    }
    return NULL;
}

void * accu(void * arg)
{
    int sum=0, i;
    for(i=0; i<5; i++)
    {
        sem_wait(&sem_one);
        sum+=num;
        sem_post(&sem_two);
    }
    printf("Result: %d \n", sum);
    return NULL;
}
```



```
int main(int argc, char *argv[])
{
    pthread_t id_t1, id_t2;
    sem_init(&sem_one, 0, 0);
    sem_init(&sem_two, 0, 1);

    pthread_create(&id_t1, NULL, read, NULL);
    pthread_create(&id_t2, NULL, accu, NULL);

    pthread_join(id_t1, NULL);
    pthread_join(id_t2, NULL);

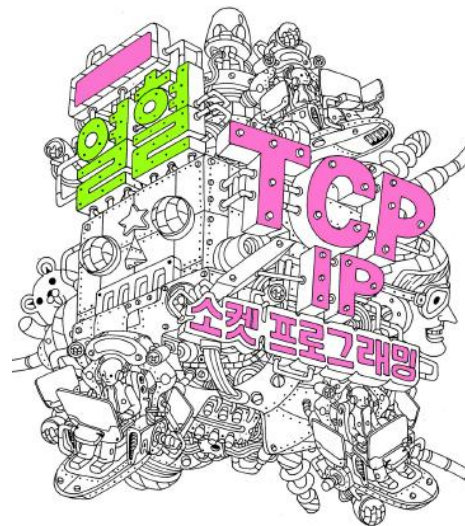
    sem_destroy(&sem_one);
    sem_destroy(&sem_two);
    return 0;
}
```

sema\_two의 초기 값이 1이다!  
따라서 read 함수 내에 있는  
scanf 함수가 먼저 호출된다.

실행 결과

```
root@my_linux:/tcip# gcc semaphore.c -D_REENTRANT -o sema -lpthread
root@my_linux:/tcip# ./sema
Input num: 1
Input num: 2
Input num: 3
Input num: 4
Input num: 5
Result: 15
```

두 개의 세마포어를 이용해서 접  
근 순서를 동기화하고 있다.



## Chapter 18-5. 쓰레드의 소멸과 쓰레드 기반 서버의 구현

윤성우 저 열혈강의 TCP/IP 소켓 프로그래밍 개정판



# 쓰레드의 소멸

## 쓰레드의 소멸을 위해 필요한 것!

- pthread\_join 함수의 호출
- pthread\_detach 함수의 호출

쓰레드 함수가 반환을 해도 자동 소멸되지 않는다. 위의 함수 중 하나를 호출해서 쓰레드의 소멸을 도와야 한다.

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

```
int pthread_detach(pthread_t thread);
```

➔ 성공 시 0, 실패 시 0 이외의 값 반환

- thread 종료와 동시에 소멸시킬 쓰레드의 ID 정보 전달.

pthread\_join 함수의 호출은 블로킹 상태에 놓이게 되니 pthread\_detach 함수를 호출해서 쓰레드의 소멸을 도와야 한다.



# 멀티쓰레드 기반의 다중접속 서버의 구현

## 예제 chat\_server.c의 일부

```
while(1)
{
    clnt_adr_sz=sizeof(clnt_adr);
    clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr,&clnt_adr_sz);
    pthread_mutex_lock(&mutex);
    clnt_socks[clnt_cnt++]=clnt_sock;
    pthread_mutex_unlock(&mutex);

    pthread_create(&t_id, NULL, handle_clnt, (void*)&clnt_sock);
    pthread_detach(t_id);
    printf("Connected client IP: %s \n", inet_ntoa(clnt_adr.sin_addr));
}
```

쓰레드 함수호출이 완료  
되면 자동으로 쓰레드가  
소멸될 수 있도록  
**pthread\_detach** 함수를  
호출하고 있다.

위의 반복문에서 보이듯이 클라이언트와 연결되면, 쓰레드를 생성하면서 해당 쓰레드에 소켓을 전달한다. 그래서 쓰레드가 클라이언트에게 서비스를 제공하는 구조로 서버를 디자인한다.

# 멀티쓰레드 기반의 다중접속 서버의 구현

```
void * handle_clnt(void * arg)
{
    int clnt_sock=*((int*)arg);
    int str_len=0, i;
    char msg[BUF_SIZE];
    while((str_len=read(clnt_sock, msg, sizeof(msg)))!=0)
        send_msg(msg, str_len);
    pthread_mutex_lock(&mutx);
    for(i=0; i<clnt_cnt; i++) // remove disconnected client
    {
        if(clnt_sock==clnt_socks[i])
        {
            while(i++<clnt_cnt-1)
                clnt_socks[i]=clnt_socks[i+1];
            break;
        }
    }
    clnt_cnt--;
    pthread_mutex_unlock(&mutx);
    close(clnt_sock);
    return NULL;
}
```

예제 `chat_server.c`의 일부

하나의 뮤텍스를 대상으로 두 영역에서 동기화를 진행하고 있다

수신된 메시지를 모든 클라이언트에게 전송하는 코드이다. 소켓정보를 참조하는 코드가 동기화되어 있음에 주목하자!

소켓정보를 참조하는 동안 소켓의 추가 및 삭제(종료)를 막겠다는 의도이다.

```
void send_msg(char * msg, int len) // send to all
{
    int i;
    pthread_mutex_lock(&mutx);
    for(i=0; i<clnt_cnt; i++)
        write(clnt_socks[i], msg, len);
    pthread_mutex_unlock(&mutx);
}
```

# 쓰레드 기반의 채팅 클라이언트

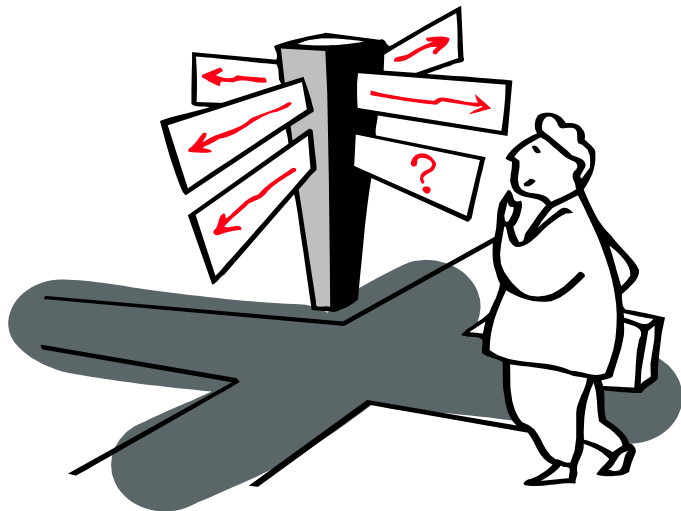
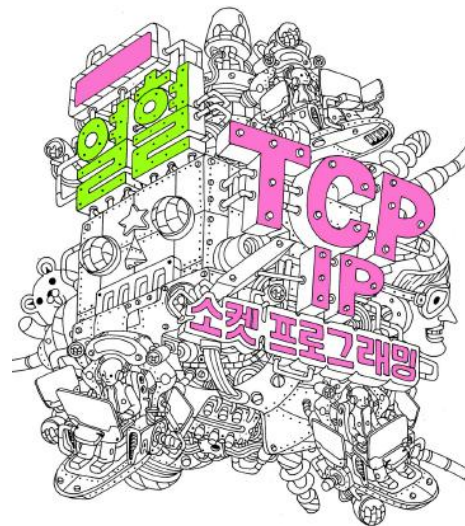
```
if(connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr))==-1)
    error_handling("connect() error");

pthread_create(&snd_thread, NULL, send_msg, (void*)&sock);
pthread_create(&rcv_thread, NULL, recv_msg, (void*)&sock);
pthread_join(snd_thread, &thread_return);
pthread_join(rcv_thread, &thread_return);
close(sock);
```

```
void * recv_msg(void * arg)    // read thread main
{
    int sock=*((int*)arg);
    char name_msg[NAME_SIZE+BUF_SIZE];
    int str_len;
    while(1)
    {
        str_len=read(sock, name_msg, NAME_SIZE+BUF_SIZE-1);
        if(str_len==-1)
            return (void*)-1;
        name_msg[str_len]=0;
        fputs(name_msg, stdout);
    }
    return NULL;
}
```

데이터의 송신과 수신에 각각  
쓰레드를 할당하는 형태로 구현  
되었다.

```
void * send_msg(void * arg)    // send thread main
{
    int sock=*((int*)arg);
    char name_msg[NAME_SIZE+BUF_SIZE];
    while(1)
    {
        fgets(msg, BUF_SIZE, stdin);
        if(!strcmp(msg, "q\n")||!strcmp(msg, "Q\n"))
        {
            close(sock);
            exit(0);
        }
        sprintf(name_msg,"%s %s", name, msg);
        write(sock, name_msg, strlen(name_msg));
    }
    return NULL;
}
```



Chapter 18이 끝났습니다. 질문 있으신지요?