

url 19 : https://www.youtube.com/watch?v=RdJY64ME8ko&list=PLz—ENLG_8TMdMJlwyqDlpcEOysvNoonf&index=21

url 20 : https://www.youtube.com/watch?v=GLMNFOCiGSU&list=PLz—ENLG_8TMdMJlwyqDlpcEOysvNoonf&index=22

● TCP/UDP

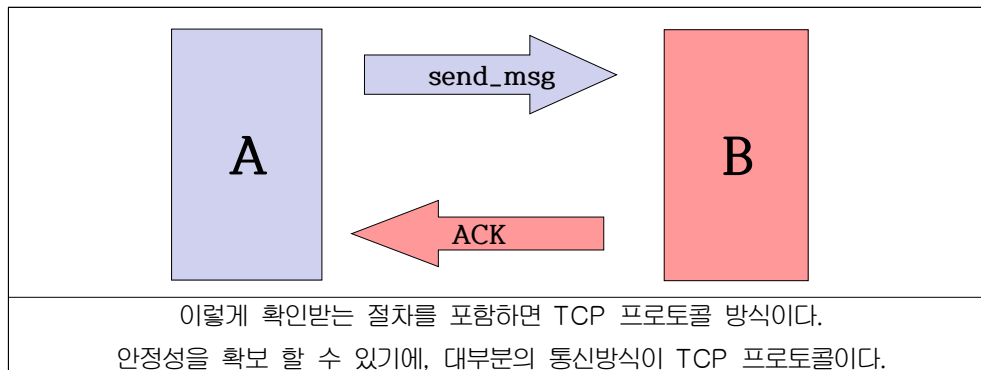
1. 통신의 개념

발신자가 신호를 보내면, 수신자가 알아듣는다.

여기서 신호의 의미를 해석하는 것도 중요하다. 통신에서 수신 받은 신호의 의미를 읽는 약속을 프로토콜이라고 한다.

또한 물리적인 통로가 필요하다.

2. TCP

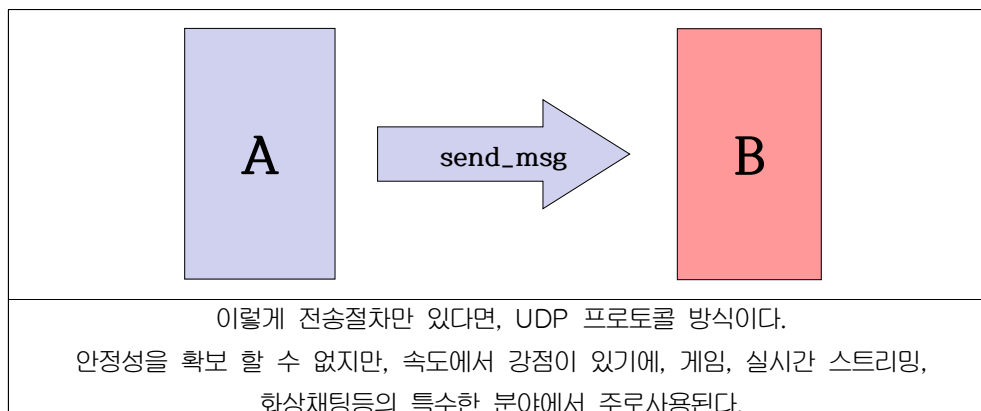


A가 B에게 패킷을 전송한다.

그럼, B는 A에게 정확히 패킷을 받았다고 ACK 신호를 보낸다.

만약, ACK 신호가 오지 않았다면, A는 다시 패킷을 B에게 전송한다.

3. UDP



A는 B에게 패킷을 전송하는 것으로 끝난다.

● 소켓

하나의 추상화 개념이다.

원래는 통신이 이루어지려면, 발신자와 수신자간의 가상의 통로 (세션)가 필요하다.

예를 들어서 발신자는 SYN 신호를 보내고, 확인하고 수신자는 ACK 신호를 보내고, 발신자는 이를받고 등등의 과정이 있지만,

이러한 복잡한 과정을 콘센트구멍에 비유해 하나의 개념으로 추상화하였다.

어떤 운영체제든, 어떤 HW를 사용하든 프로토콜만 지켜서 통신하게 해주는 하나의 추상화 개념이다.

해당 실습은 TCP 프로토콜만을 사용하여 진행한다.

1. TCP 서버

1. 1. 특징

- * 서버소켓은 연결만을 담당
- * 서버와 클라이언트간에 1대1로 연결, 이에 대한 소켓도 따로 필요
- * 해당 소켓은 대체적으로 client의 전송내용을 기다리는 동작을 취함
- * 대체적으로 서버가 열리고 클라이언트가 들어오는 구조로 설계

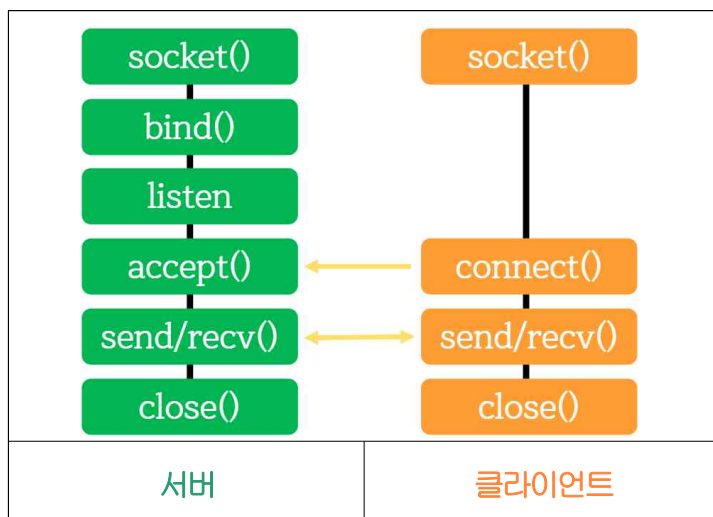
2. TCP 클라이언트

2. 1. 특징

- * 서버로의 접속을 시도하는 소켓이 필요
- * 해당 소켓으로 서버와의 통신까지 진행

3. 통신과정

TCP 서버에 클라이언트가 몇 개가 접속하였든 상관없이, 클라이언트와 1:1 의 연결관계를 맺는다 → 클라이언트 수만큼의 소켓필요
각 동작은 함수단위로 진행된다.



3. 1. 서버의 동작함수

- * `int socket(도메인 서비스 타입, 프로토콜 지정)`

소켓을 생성하는 함수이다. 해당 파라미터에 맞춰서, int 형의 소켓 디스크립터를 생성한다.

C/C++ 의 클래스나 구조체를 지정하는 것과 마찬가지로 틀을 지정한다.

* `int bind`(소켓디스크립터, 소켓에 할당할 주소 정보, 해당주소의 길이)

소켓을 특정 IP주소와 포트번호에 연결한다.

이렇게 되면, 소켓은 클래스나 구조체의 인스턴스와 같은 형태를 지니게 된다.

* `int listen`(소켓디스크립터, 최대 연결요청 개수)

소켓을 수신대기 상태로 설정한다

최대 연결요청개수는 서버에 접속을 시도하는 클라이언트의 수를 나타낸다.

* `int accept`(소켓디스크립터, 연결된 클라이언트의 주소정보, 해당주소의 길이)

수신대기중인 연결요청을 수락하고, 새로운 소켓을 생성하여 클라이언트와 연결한다.

* `ssize_t send`(소켓디스크립터, 전송할 버퍼, 데이터의 길이, 옵션 플래그)

소켓을 통해 데이터를 상대방에게 전송한다. (실습에선 `write` 함수로 진행)

* `ssize_t recv`(소켓디스크립터, 수신할 버퍼, 수신할 데이터 길이, 옵션 플래그)

소켓을 통해 버퍼에 데이터를 수신한다. (실습에선 `read` 함수로 진행)

* `int close`(소켓디스크립터)

소켓을 닫는다.

* `int connect`(소켓디스크립터, 서버의 주소정보, 주소의 길이)

서버에 연결을 요청한다.

* 주소정보

주소정보는 다음과 같은 구조체로 선언된다.

<pre>struct sockaddr_in { ADDRESS_FAMILY sin_family; USHORT sin_port; IN_ADDR sin_addr; }</pre>	<p><code>sin_family</code> : unsigned short 형으로 주소체계를 지정한다. <code>sin_port</code> : uint16_t 형으로 포트번호를 지정한다. (바이트변환) <code>sin_addr</code> : 구조체 <code>in_addr</code> 으로 ip 주소를 지정한다. (바이트변환)</p>
---	---

(여기서, 바이트 변환은, 리틀엔디안에서 빅엔디안으로의 변환이 필요함을 의미한다.)

● 스레드

운영체제는 저장 HW의 코드를 메모리로 올리고, 운영체제의 스케줄러에 따라서 프로세스들을 정렬해, 조금씩 실행시킨다.

CPU 입장에서 실행되는 가장 최소의 작업단위는 스레드

운영체제 입장에서 실행되는 가장 최소의 작업단위는 프로세스

결국 '프로세스 > 스레드' 라고 볼 수 있다.

프로세스와 스레드의 가장 큰 차이는 다음으로 볼 수 있다.

똑같이 프로세스와 스레드는 운영체제가 관리하지만, 프로세스는 다른 프로세스에게 독립적이고, 스레드는 다른 스레드에게 종속적이다. 운영체제가 스레드를 스케줄러에 올려서 조금씩 실행한다고 했었는데, 조금씩 실행하게 되면, CPU는 작업중에 갑자기 다른 작업을 해야 한다. 이를 context switching 이라고 한다.

프로세스끼리의 context switching은 기존의 작업정보를 어딘가에 저장하고 새로운 작업을 진행하기에 상관없지만, 스레드는 프로세스의 부분집합이기에, 해당 리소스를 스레드끼리 나눠쓴다. 그래서 context switching이 일어나면 기존의 작업정보는 저장되지 않는다 -> 버그 위험성

1. 위험성 예시

```
<function A>

if ~
    count ++;      // -> 스레드1 실행
else              // 갑자기 context switching
    count --;      // -> 스레드2 실행

....

printf("%d", count);    // count 변수가 변하지 않네?
```

count 변수가 전역변수나 static 변수라고 가정해보자. 만약, A 함수를 스레드1,2 로 동작 시켜볼 때, 변수 count에 접근하는 과정에서 context switching 이 발생하여, 제대로 된 접근이 일어나지 않게되면? 해당 변수는 올바른 값을 가지지 못하게 된다.

만약, 왼쪽 예시처럼, if, else 문에서 각각 count 변수를 다르게 접근한다. 이러한 부분은 무조건 하나의 flow로 실행되어야 하는데, context switching 으로 인해, 복수의 flow로 실행되면, count는 올바른 값을 갖지 못한다. 특히나 그런 변수가 프로세스에서 중요한 key 변수일 때는 더더욱

C에서 스레드를 사용하기 위해선 다음과 같은 준비물과 과정이 필요하다.

준비물	과정
라이브러리 : #include <pthread.h> 스레드 객체 생성 : pthread_t 변수이름	변수를 선언했을 경우 * pthread_create 함수를 이용해 스레드를 원하는대로 초기화한다. 해당 함수안에 스레드가 실행시킬 함수와 함수에 필요한 파라미터들을 넣는다 * 해당함수가 끝나면 (스레드 작업종료) pthread_exit 함수를 이용해, 리소스를 반환하고, 반환 값을 전달한다 * 스레드를 기다릴때에는 pthread_join 함수를 이용해, 모든 스레드가 끝날 때까지 기다리고, 스레드가 처리한 작업의 결과를 받는다. * 쓰레드를 소스코드에서 쓰고싶다면, gcc 컴파일러에게 pthread 라이브러리를 이용한다고 지정한다. <div>gcc thread.c -Lpthread -o thread.out</div>

2. 실습

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

int a;

void * thread_DO(void * th_id)
{
    printf("arg : %d\n",(int)th_id);

    while(1)
    {
        printf("thread%d : a[%d]\n",(int)th_id,a++);
        sleep(2);
    }

    return NULL;
}

int main()
{
    pthread_t thread1 , thread2; // pthread_t는 리눅스에서 unsigned long
    int th1_id = 77;
    int th2_id = 88;

    if( !pthread_create(&thread1, NULL, thread_DO,(void*)th1_id)) // 스레드 생성 성공
    // 함수의 매개변수 (스레드 포인터, 스레드 특성을 지정하는 구조체 포인터)
    // 특성은 스레드의 스택 크기, 우선순위 등의 특성을 설정 -> NULL로 전
    // 나머지는 스레드에게 할당된 함수에 들어갈 파라미터
        printf("created thread1 successfully!\n");

    else
        printf("failed to create thread1\n");
    // pthread_create 함수는 성공하면 0을 반환, 아니면 오류코드를 반환

    if(!pthread_create(&thread2, NULL, thread_DO, (void*)th2_id))
        printf("created thread2 successfully!\n");
    else
        printf("failed to create thread2\n");

    while(1)
    {
        printf("main thread loop!\n");
        sleep(1);
        // 메인코드도 어떻게 보면 또다른 스레드여서, 메인문이 리턴
        -> 원하는 동작 실행x
    }

    if( !pthread_join(thread1, NULL)) // 스레드가 종료될 때까지 호출
    대기시키는 함수
    // 함수의 매개변수 (종료를 기다릴 대상 스레드, 스레드가 반환한
    // NULL 일때는 반환값을 무시
        printf("thread1 has been terminated successfully!\n");

    if(!pthread_join(thread2,NULL))
        printf("thread2 has been terminated successfully!\n");

    return 0;
}
```

thread.c 소스코드

```
root@LAPTOP-BQ4AK11N:/project/24-Cstudy20# ./thread.out
created thread1 successfully!
arg : 77
thread77 : a[0]
arg : 88
thread88 : a[1]
created thread2 successfully!
main thread loop!
main thread loop!
thread77 : a[2]
main thread loop!
thread88 : a[3]
main thread loop!
thread77 : a[4]
thread88 : a[5]
main thread loop!
main thread loop!
```

실행결과

해당 스레드의 동작함수는 전역변수 a를 가진다.
그리고, 스레드 2개에 프로세스의 메인문을 도는 원본까
지 합하여 총 3개의 스레드라고 봐야한다.

결과를 보면, 3개의 스레드가 불규칙하게 실행되는 것을
볼 수 있다.

2. mutex

그래서 이런 부분을 critical Area 라고 칭한다. 이러한 부분에서 context switching이 일어나지 않게 할 수는 없을까?
쓰레드를 사용할 수 있게 하는 라이브러리에선 mutex라는 객체를 제공한다.

이 객체는 원하는 critical Area에서 context switching이 일어나지 않게 lock을 걸어준다.
물론 그 영역을 벗어나면, switching 이 일어나게 unlock을 해주는 작업도 필요하다.

역시나 mutex를 사용하기 위해선 준비물과 과정이 필요하다.

준비물	과정
라이브러리 : #include <pthread.h> 뮤텍스 객체 생성 : pthread_mutex_t 변수이름	변수를 선언했을 경우 * pthread_mutex_init 함수로 뮤텍스 객체를 초기화 * 해당 뮤텍스를 쓸 critical 영역에 pthread_mutex_lock 함수로 context 스위칭을 방지하고 pthread_mutex_unlock 함수로 context 스위칭을 다시 열어준다 * 뮤텍스의 리소스를 반환하기 위해, pthread_mutex_destroy 함수를 사용한다. * 역시나 gcc 컴파일러에게 pthread 라이브러리를 이용한다고 지정한다. <div>gcc thread.c -Lpthread -o thread.out</div>

2. 1. 실습

이전 실습에서 뮤텍스를 추가하여 결과를 살펴보았다.
a 변수에 접근할 때, 뮤텍스를 걸어두어서, 스레드들이 동시에 a 변수에 접근하지 못하게 하였다.

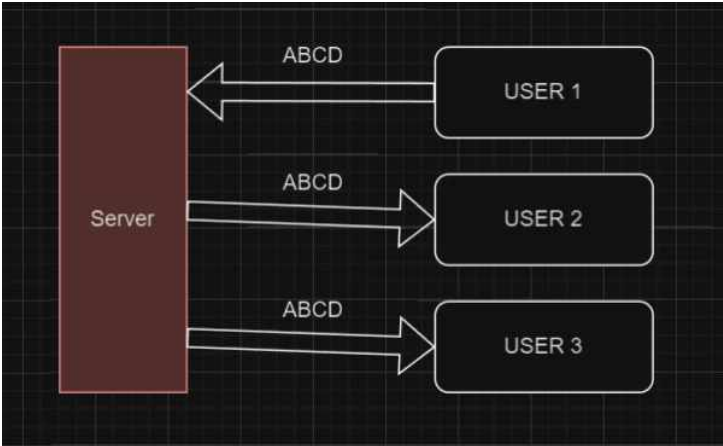
```
1 #include <pthread.h>
2 #include <stdio.h>
3
4 int a;
5 pthread_mutex_t mutex; // 뮤텍스 선언
6
7 void * thread_DO(void * th_id)
8 {
9     printf("arg : %d\n", (int)th_id);
10
11     while(1)
12     {
13         pthread_mutex_lock(&mutex); //뮤텍스로 context switching 방지
14
15         printf("thread%d : a[%d]\n", (int)th_id, a++);
16
17         pthread_mutex_unlock(&mutex); // lock 해제
18
19         sleep(2);
20     }
21
22     return NULL;
23 }
24
25 int main()
26 {
27     pthread_t thread1, thread2;
28     int th1_id = 77;
29     int th2_id = 88;
30
31     pthread_mutex_init(&mutex, NULL); // 뮤텍스 초기화하기
32     // 뮤텍스 초기화함수 (뮤텍스 주소값, 옵션)
33     // NULL로 하면, fast 옵션으로 실행
34
35     if (!pthread_create(&thread1, NULL, thread_DO, (void*)th1_id))
36         printf("created thread1 successfully!\n");
37
38     else
39         printf("failed to create thread1\n");
40
41     if (!pthread_create(&thread2, NULL, thread_DO, (void*)th2_id))
42         printf("created thread2 successfully!\n");
43
44     else
45         printf("failed to create thread2\n");
46
47     while(1)
48     {
49         printf("main thread loop!\n");
50         sleep(1);
51     }
```

```
51
52
53     if( !pthread_join(thread1, NULL))
54         printf("thread1 has been terminated successfully!\n");
55
56     if(!pthread_join(thread2, NULL))
57         printf("thread2 has been terminated successfully!\n");
58
59     pthread_mutex_destroy(&mutex); // 무텍스 삭제
60     return 0;
61
62 }
```

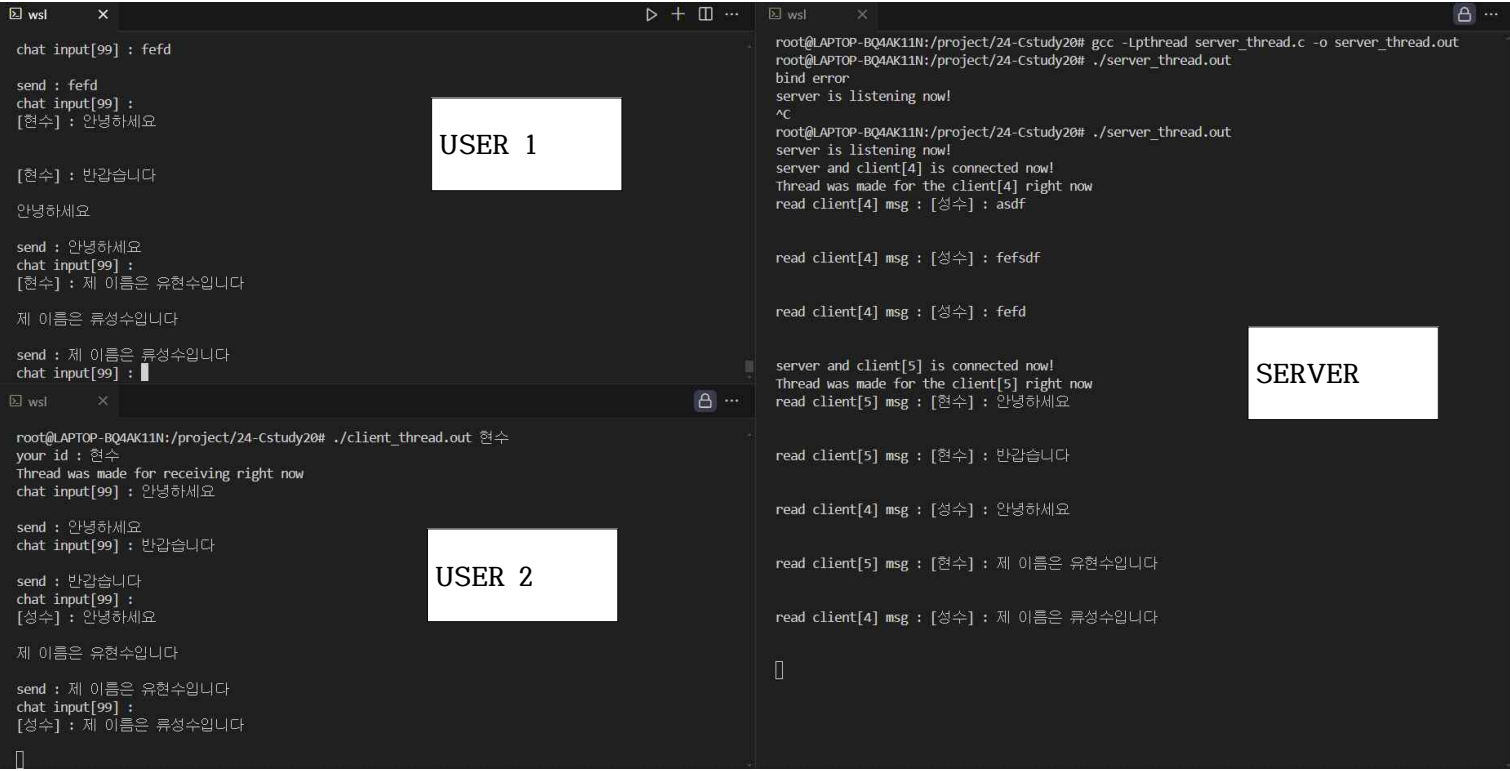
결과를 살펴보면, 이전과 다르게 a 변수가 두 번 연속으로 사용되지 않는 것을 확인할 수 있다.

```
root@LAPTOP-BQ4AK11N:/project/24-Cstudy20# ./thread2.out
created thread1 successfully!
arg : 77
thread77 : a[0]
created thread2 successfully!
main thread loop!
arg : 88
thread88 : a[1]
thread77 : a[2]
main thread loop!
thread88 : a[3]
thread77 : a[4]
main thread loop!
thread88 : a[5]
thread77 : a[6]
thread88 : a[7]
main thread loop!
thread77 : a[8]
thread88 : a[9]
main thread loop!
thread77 : a[10]
thread88 : a[11]
main thread has escaped loop! with counter : 5
thread77 : a[12]
thread88 : a[13]
thread77 : a[14]
```

● 소켓프로그래밍 실습

설계	
	<p>서버</p> <ol style="list-style-type: none">무한루프로 Listen하며, 클라이언트 기다리기클라이언트가 Connect을 하면, Accept 하고, 1대1 소켓생성해당 클라이언트와의 통신은 스레드를 생성해서 처리사용자가 날린 msg를 다른 사용자들에게 뿌리기
	<p>클라이언트</p> <ol style="list-style-type: none">터미널에서 id를 함께 입력하여 실행서버에 Connect 연결요청연결되면 id와 함께 특정 패턴으로 msg 보내기서버의 msg를 받을 함수를 스레드로 돌려서 대화창에 뿌리기

1. 결과



잘 작동하는 것을 확인하였다.

소스코드 url : <https://github.com/supergravityy/-git>