[Chapter 3]

경영학과/2018100233/김정민

I. Process

A. Program vs. Process (차이점)

1. Program (정적)
   - 디스크에 저장되어 있는 실행 가능한 파일
   - 아직 실행되지 않은 상태
   - 컴파일을 통해 만들어진 바이너리 이미지 (코드 덩어리)
2. Process (동적)
   - 프로그램이 실행된 상태
   - 과거에는 실행과 스케줄링의 기본적인 유닛
     - 이제는 Thread라는 더 기본적인 유닛의 개념이 나왔다.
   - Process ID (PID)를 통해 각 프로세스를 식별 가능하다.
     - Kernel이 관리
   - 디스크에 있던 프로그램이 **커널에 의해** 실행되면, 코드 덩어리가 **메모리에 로드된다**.

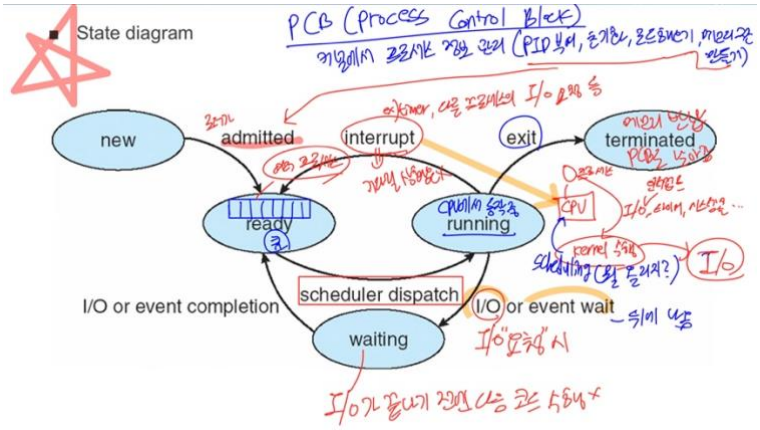
B. Process Concept (프로세스란?)

1. 실행 중인 프로그램의 인스턴스
   - 실행 가능한 파일(프로그램)을 실행하면 로더에 의해 메모리에 프로그램의 복사판이 올라간다..
2. Encapsulation
   - Binary code의 동작(분기, 순차, 점프, 등)을 사용자는 모르고도 프로그램을 실행할 수 있다.
3. Dynamic and active entity
   - 프로그램과 반대되는 개념
   - 프로그램이 저장되어 있는 디스크는 static한 개념

C. Process Address Space (저장공간)

1. 개발자는 피지컬 메모리의 주소는 몰라도 된다.
   - 로지컬(가상) 메모리의 주소만 활용해도 된다.
   - 각 프로세스는 자체의 가상 메모리를 갖게 되고, 각 프로세스가 독립적으로 실행될 수 있다.
2. Stack
   - 함수 호출과 관련된 변수와 함수 호출을 관리하기 위한 메모리 공간
   - 런타임에 함수의 호출과 반환 지원
3. Heap
   - New(), malloc(), 등 동적 할당을 위한 메모리 공간
4. Static data
   - 정적 변수들을 저장해놓는 공간
   - 글로벌 변수, 상수 데이터 등
5. Code
   - 이진수로 컴파일된 코드 덩어리를 저장해놓는 공간

D. Process State (다이어그램 이해 중요)



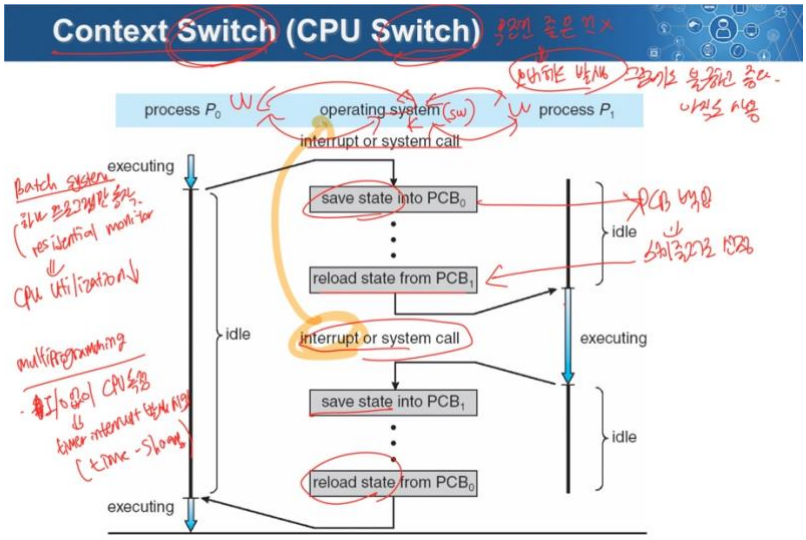
1. new
   - 아직 프로세스가 실행되지 않은 상태
   - 실행하기 위해 대기 중인 상태
2. ready
   - 할 일은 있지만, 스케줄러에 의해 CPU 점유를 위해 대기 중인 상태
   - 레디 큐에 들어가서 대기한다.
   - 스케줄러에 의해 running 상태로 전환될 수 있다.
3. running
   - CPU를 할당받아서 CPU 위에서 동작 중인 상태
   - 커널에 I/O 요청을 하게 되면, waiting 상태로 전환된다.
   - 타이머 인터럽트 등 다른 작업을 기다릴 필요가 없는 상황엔 다시 ready 상태로 돌아가서 레디 큐 상에서 대기한다.
4. waiting
   - 요청한 I/O 작업이 끝나기 전엔 다음 명령어를 수행할 수 없어서 레디 큐에 들어갈 수 없다.
     - I/O가 얼마나 걸릴 지 모르는 것!
   - 그래서 웨이팅 큐가 존재한다.
   - 요청한 I/O 작업이 완료되면 ready 상태로 전환된다.
5. terminated
   - 프로세스가 exit() 되면, terminated 상태가 된다.
   - 이때, PCB의 정보도 삭제되고, 메모리 또한 반납된다.

E. Process Control Block(PCB)

1. 각 프로세스를 관리하기 위한 관한정보를 갖고있다. (매우 중요)
   - 프로세스 스케줄링할 때 필요한 정보들을 제공
2. 포함 정보
   - Process state
   - PC (Program Counter)
   - CPU registers (값)
   - CPU scheduling information
   - Memory-management information (메모리)
   - Accounting information
   - I/O status information (어떤 파일을 열었는지 등)
3. 리눅스에선 doubly linked list로 관리된다.

II. Scheduling

A. Context Switch (CPU Switch)

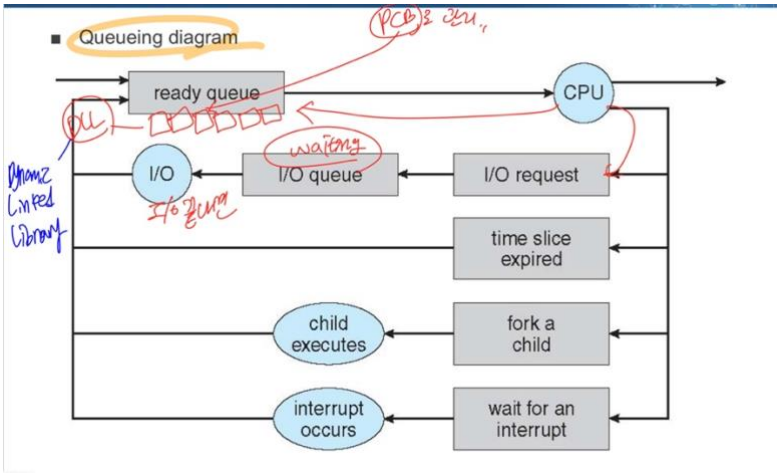


1. Batch system
   - 하나의 프로그램만 동작한다.
   - I/O 요청이 발생하면 그동안 CPU는 유휴상태에 들어가기 때문에 CPU 활용률이 낮다.
2. multiprogramming
   - I/O가 없으면 CPU를 한 프로세스가 계속 독점할 수 있다.
   - 타이머 인터럽트를 통해 time-sharing
3. Switch 과정
   - CPU 상에서 연산을 수행하다가 인터럽트나 시스템 콜에 의해 다른 프로세스가 CPU를 점유하게 될 때
   - 기존 P0의 PCB 상태를 모두 백업한다.
   - 그리고 스케줄러에 의해 CPU 점유를 허락받은 P1의 PCB 상태를 불러온다.
   - 해당 과정이 계속 반복된다.
   - 계속해서 커널의 개입이 있기 때문에 무조건 좋은 건 아니다.
     - 커널 오버헤드가 발생
     - 하지만 그래도 좋기 때문에 아직도 사용한다.
4. 스위치 과정에서 커널의 오버헤드가 발생 (할 일이 늘어난다.)
   - 레지스터와 메모리 매핑 정보를 저장해서 다음에 다시 실행할 때 작업을 이어서 수행될 수 있게 해준다.
   - 캐시 메모리를 비우고 새로운 프로세스에 필요한 캐시를 불러온다.
   - 프로세스 상태, 스케줄링 정보 등 다양한 데이터들을 갱신해야 한다.
5. 스위치 오버헤드는 하드웨어 의존성이 높다.
   - CPU 제조사들은 오버헤드를 낮추기 위해 노력한다.
6. 일반적으로 초당 수백, 수천 회의 스위치가 일어난다.
   - I/O가 많을수록 스위치는 증가한다.

B. Schedulers

1. Long-term(job) scheduler
   - 가상 메모리가 없던 시절
   - Disk to memory
   - 어떤 프로세스를 레디 큐에 올릴 것인지 결정하는 스케줄러
2. Medium-term scheduler (swapper)
   - 가상 메모리가 없던 시절
   - 실행 중인 프로세스들이 메모리 오버플로우를 발생시켜서(동적 메모리 할당 등) 프로세스를 메모리에서 디스크로 내려야 하는 경우에 어떤 프로세스를 내릴 것인지 결정하는 스케줄러

3. Short-term(CPU) scheduler
   - Ready queue to CPU
   - 레디 큐에 대기 중인 프로세스 중 어떤 프로세스를 CPU에 올릴 것인지 결정하는 스케줄러
4. Queueing diagram

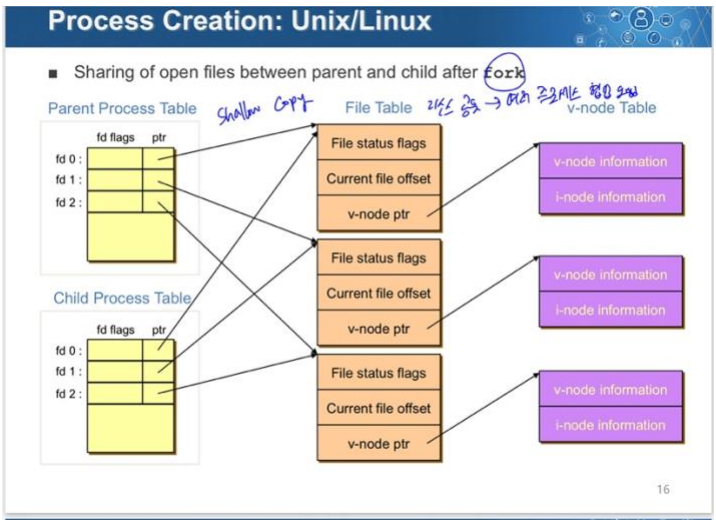


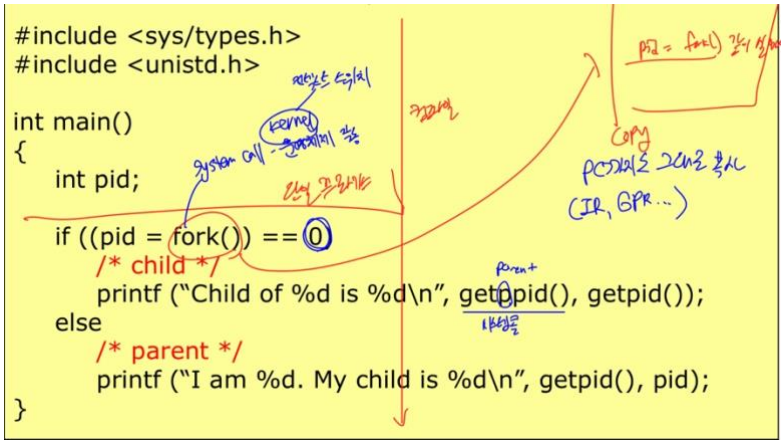
5. 레디 큐, 웨이팅 큐 등 각종 큐들은 linked list로 관리된다.
   - I/O 웨이팅 큐의 경우, I/O 장치에 따라서 다른 웨이팅 큐가 존재한다.
   - 각 큐의 헤더에서 head와 tail 포인터를 가지고 있어서 처음과 끝을 알 수 있다.
   - 임베디드 OS에서는 이런 복잡한 구조를 사용할 수 없다.

III. Operations on Processes

A. UNIX/LINUX

1. fork()
   - Process creation
   - 프로세스 새로 생성 (PCB 생성 및 초기화, 메모리 공간 생성 및 초기화)
   - 자식 프로세스의 경우, 부모의 모든 PCB 및 메모리 공간을 shallow copy 형태로 복사한다.
     - 오픈한 파일, I/O 디바이스, PC 등까지 포함
   - 부모 프로세스는 Fork() 끝나면 자식의 PID 반환, 자식 프로세스는 0을 반환
     - 부모, 자식을 판별할 수 있다. (코드가 똑같으니 구분 필요)
   - PCB를 레디 큐에 넣는다.

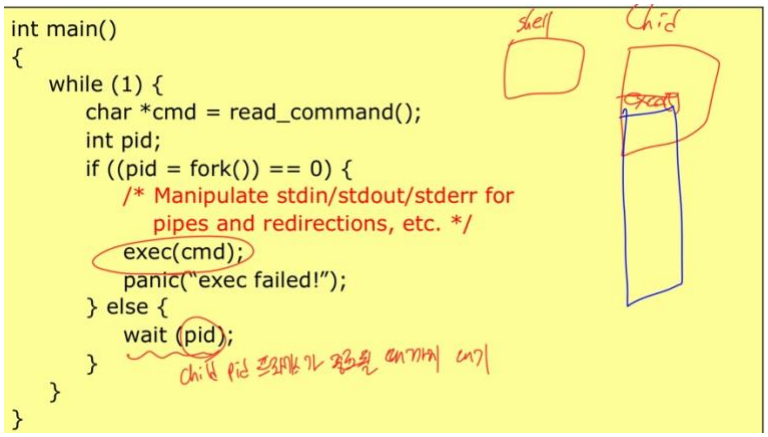




- 부모 프로세스는 계속 새로운 요청을 기다린다.
- Ex. Netflix, 웹서버
  - 서버 프로그램이 존재하고, 요청이 들어올 때마다 fork()를 통해 자식 프로세스를 생성
- 부모와 자식 프로세스는 스케줄러에 따라 수행 순서가 달라질 수 있다.
  - 부모 프로세스도 fork()가 완료되기 전까지는 레디 큐에 들어간다.
- 소켓을 연결해서 클라이언트 하나당 자식 프로세스를 생성한다.
- 부모와 자식 간에 협업할 때 유용하다.
- 싱글 프로세스의 경우에는 요청 처리하는 중에 다른 클라이언트한테 요청을 받아도 처리할 수가 없다.

2. exec()

- fork()와 같이 수행된다.
  - Fork() -> 자식 생성 -> 즉시 Exec()
- 프로그램(.exe)을 프로세스의 메모리 공간에 로드한다.
- 새로운 프로그램을 위해 하드웨어 정보와 아규먼트들을 초기화한다.
- PCB를 레디 큐에 넣는다.
- 부모는 새로운 요청을 기다리고 자식이 끝날 때까지 기다리는 반면에, 자식은 Exec()을 수행해서 새로운 프로그램을 실행한다.



B. Windows(무근본)

1. CreateProcess()

- fork()와 exec()을 합쳐서 하나의 명령어로 만든 것
- 따라서 윈도우 운영체제는 운영체제 단에서 부모 자식 관계를 관리하지 않는다.

C. Process Termination

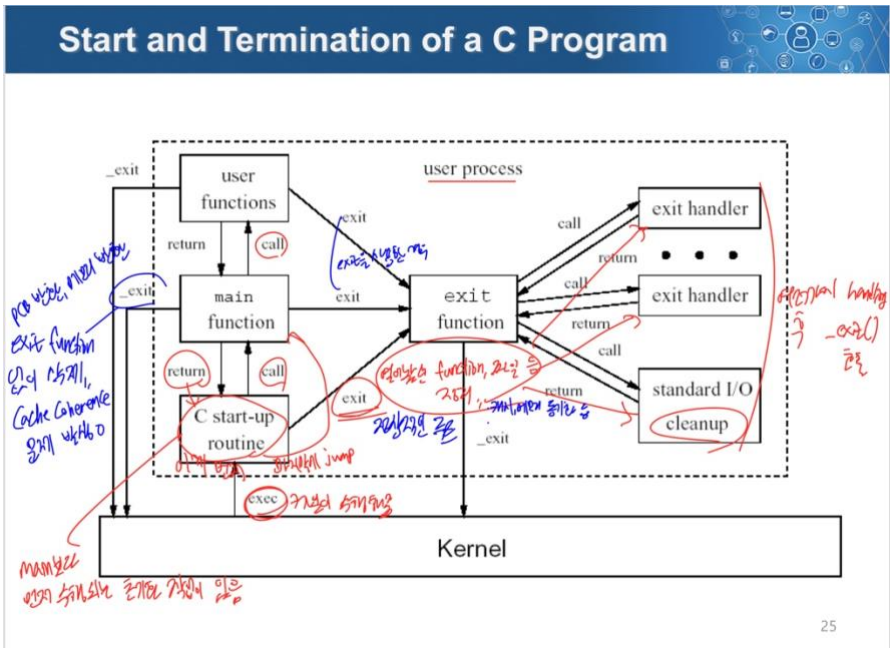
1. Normal Termination

- 메인 함수 리턴 (C언어 기준 시작점)
- exit() 호출
  - 사후처리 후 _exit() 호출

- _exit() 호출
  - 프로세스 종료에 필요한 사후처리를 하지 않는다.
  - 에러코드 알려주기 및 그냥 종료

2. Abnormal termination
   - abort() 호출
   - 특정 시그널을 통해 오류를 감지하고 정상적이지 않게 종료
   - 오류 보고 (리턴 값과 파라미터를 갖지 않음)
3. Wait for a child process
   - wait() 호출
     - 자식 프로세스가 종료될 때까지 기다린다. (동기화)
   - 기다리지 않을 시 문제점
     - 만약 부모가 기다리지 않는다면, 자식은 zombie가 된다.
     - 부모가 Wait을 호출하지 않고 종료된 경우, 자식은 orphan이 된다.

D. C 프로그램에서의 전체적인 관점



1. 시작점은 main 함수가 아닌, C start-up routine
   - Main 함수에 앞서 수행되는 초기화 작업
2. exec()은 커널이 수행한다.
3. 정상적인 exit() 함수
   - 각종 Exit handler를 수행한 후에 _exit()를 호출해서 프로세스 종료
4. Exit() 없이 _exit()로 프로세스가 종료되는 경우
   - PCB, 메모리 반환 등 사후 처리 없이 프로세스 종료
   - Cache Coherence 문제 발생

IV. Multiprocess

A. Application programs

1. 구글 크롬 브라우저
   - 세 가지 유형의 프로세스를 이용한 멀티프로세스
     - Browser process
       - 유저 인터페이스, 디스크 및 네트워크 I/O 관리
     - Renderer process

        - ◆ 웹 페이지 렌더링 및 HTML, 자바스크립트 작업 처리
        - ◆ 각 탭마다 새로운 렌더러 프로세스를 나타낸다.
        - ◆ 디스크와 네트워크 I/O를 최소화하는 샌드박스 안에서 실행 -> 보안 문제를 최소화
    - ■ Plug-in process
        - ◆ 각종 추가 기능 및 확장 프로그램을 지원하는 프로세스
        - ◆ VPN, ad-block 등 크롬 프로세스에서 fork()해서 멀티프로세스 구현

B. Mobile systems

1. 초기 버전의 IOS
    - 한 번에 하나의 프로세스만 실행, 다른 프로세스는 중지된다.
    - 스크린 공간, 유저 인터페이스의 제한
        - ■ Single foreground process
            - ◆ 유저 인터페이스로 통제되는 하나의 프로세스만 표시된다.
        - ■ Multiple background processes
            - ◆ 화면엔 나오지 않지만 메모리에서 실행된다.
            - ◆ 여전히 제약이 존재한다.
        - ■ 이러한 제한에는 한 번에 실행되는 작업 하나, 이벤트 수신 및 오디오 재생과 같은 특정한 장기 실행 작업들도 포함된다.

2. Android
    - Foreground와 background 프로세스를 모두 실행할 수 있다.
    - 제한이 적다.
    - 백그라운드 프로세스는 작업을 수행하기 위해 서비스를 사용한다.
    - 서비스는 백그라운드 프로세스가 중단됐더라도 계속 수행된다.
    - 서비스는 유저 인터페이스가 없고, 메모리 사용량이 적다.

V. Inter-Process Communication (IPC)

A. Communication models
    - 프로세스 간의 정보 전달
    - Memory protection : 다른 프로세스의 메모리에 직접적으로 접근할 수 없다.
    - 그럼 어떻게?

1. Message passing
    - 메시지 큐 활용
    - OS에서 양방향 버퍼를 직접 관리
    - 커널이 관리하기 때문에 안전한 방법
    - But 커널의 오버헤드 증가

2. Shared memory
    - 프로세스 A, B가 공용 메모리를 할당받아서 같이 사용한다.
    - OS의 역할은 공용 메모리를 할당해주는 것으로 한정 (오버헤드 적음)
    - 메모리에 데이터가 쓰여있는지 등에 대한 메모리 관리는 프로세스에서 수행 (개발자의 몫)
    - OS는 메모리가 어떻게 사용되고 있는지 관여하지 않기 때문에 위험한 방법
    - 메모리가 제대로 반환되지 않으면 메모리 누수 발생 -> 동기화 문제 발생

B. Cooperating processes

1. Bounded buffer problem (producer-consumer problem)
    - 두 프로세스 간의 동기화를 위해 버퍼 동기화 코드가 필요하다.
    - 이게 잘못되면 메모리 누수 발생할 수 있다.

C. UNIX/LINUX에서의 IPC를 수행하기 위한 여러 방법

1. Pipes
2. FIFOs
3. Message queue
4. Shared memory
5. Sockets
   - 소켓은 원격 컴퓨터 간의 통신(네트워크 프로그래밍)을 지원하는 것이지만, 소켓을 통해서 IPC(프로세스 간 통신)를 구현할 수 있다.
   - 개발자 입장에선 익숙한 방법

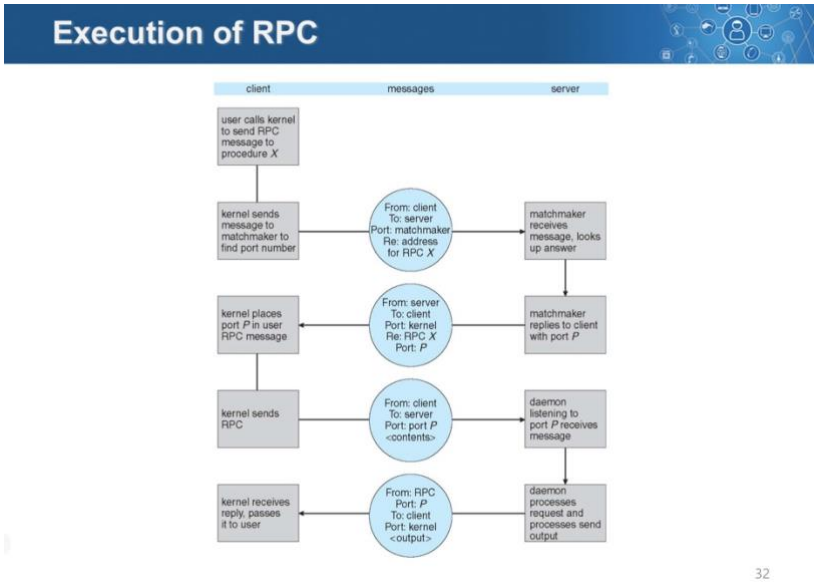
VI. Client-Server Communication

A. Sockets

1. 클라이언트와 서버 간의 데이터 통신을 가능하게 한다.
2. 클라이언트 소켓과 서버 소켓의 연결 설정이 필요하다.

B. Remote Procedure Call (RPC)

1. 소켓을 활용
2. 클라이언트는 함수 수행에 필요한 정보만 전달하고, 원격의 서버에서 해당 함수를 수행해서 리턴 값을 클라이언트로 보내준다. (로컬 함수를 실행하는 것처럼 보임)



C. Remote Method Invocation (RMI in JAVA)

1. 자바에서 사용되는 클라이언트-서버 통신 방법
2. 클라이언트와 서버 모두 자바로 작성된다.
3. 클라이언트에서 서버로 함수 수행에 필요한 데이터를 전송하는 과정