3장. Process
0. 목적
-프로세스의 본질
프로세스를 어떻게 OS에서 지원하는지?
-프로세스의 생성 및 종료 과정
파일, 네트워크 등 액세스 가능하며, 다양한 계산 수행 가능
-프로세스의 경쟁 관계
프로세스 여러 개 있기 때문에, 서로 협력하거나 CPU나 I/O 사용권에 대해 경쟁하는 관계가 되는데, 이를 OS가 잘 제어함 (Process Scheduling)
-IPC의 개념
2가지의 모델 존재
OS에서 IPC를 제공하며, 그 방법으로 pipes와 POSIX가 제공하는 shared Memory 살펴봄
-분산 환경에서 C-S Communication
C-S 모델에서 Communication 존재하는데, 이 C-S에서의 IPC에는 Socket(네트워크를 쉽게 구현하기 위한 API), Remote procedure call(소켓과 같은 네트워크를 이용해 사용자가 더 쉽게 프로그래밍할 수 있도록 도와주는 것)의 형태 존재하며 이에 대해 알아봄

1. Process 개념
Process : 프로그램이 수행하는 것
ex. 사용 설명서를 보고 Device를 보고 수행하는 동작
프로세스는 메모리 상에 로딩되어야 함.

Program(passive) vs. Process(active)
프로그램은 executable file의 일종으로써 Storage에 저장되어 그대로 있는 것
프로그램은 GUI을 통하거나 CLI을 통해 수행 가능함
한 프로그램에 여러 프로세스가 수행할 수 있다. ex. 크롬 여러 번 실행

Process의 여러 가지 구성 요소 -> 7p
-Text Section : prog 코드(명령어라고 생각하면 됨)
-Program Counter (CPU 내 Register 값 - 다음에 실행할 Instruction을 가리킴)
P.C는 CPU 내 존재하지만, 프로세스는 CPU를 사용할 때 다음에 수행해야 할 instruction 번지 수가 얼마인지 독립적으로 갖고 있어야 함(CPU는 여러 proc가 공유하기 때문에)
-Stack
일시적인 Data를 보관하는 장소(함수 호출할 때마다 늘어남, return마다 줄어듦)
ex. 함수 호출 및 return 과정에서 일시적인 저장 장소가 필요한데 이것이 stack에 저장됨
main 또한 함수의 영역이므로, Stack에 저장됨

#데이터 영역이 Data Section과 Heap으로 나누어짐
-Data Section
프로그래밍할 때, 미리 설정해놓은 Data (Data 영역이 고정된 것)
프로그램 내 전역 변수

-Heap(Dynamic Memory 수행 시)
프로그램 도중에 필요에 따라 메모리에 추가 할당하는 Data (Data 영역이 고정되지 않은 것)
Stack이나 Heap은 프로그램 수행 중 size의 변경이 있으므로, 빈 공간을 가운데 두고 존재

Process State
Process는 살아서 움직이므로 상태가 실시간으로 바뀜
-New : 프로세스 새로 생성된 상태
#생성 후
-Running : CPU를 할당받아 동작하는 상태
-Waiting : 다양한 event를 기다리는 상태 ex. I/O 수행, 사용자 입력 등 대기 상태
-Ready : Waiting 하던 것이 event 발생하여 Ready로 가 CPU 할당 대기 상태
#종료 전
-Terminated : 종료 상태
>>생성 후 종료 전까지는 3가지 상태가 반복적으로 process 수행

State Trasition Diagram (of Process state)
-Node처럼 보이는 것은 State이며, 화살표는 다른 State로의 Transition을 의미함.
-schedular dispatch : CPU schedular에 의해 자신이 CPU를 할당 받았다는 의미
*running -> ready 상태 전환 ex. CPU를 계속적으로 할당받는 것이 아니라 Time sharing에 의해 자신이 짧은 시간 CPU를 할당받은 경우 or prog 수행 중 H, S/W INT 오는 경우 (Timer INT 일종)

Process Control Block (PCB)
-Process를 관리하기 위한 Table
OS 제공 하는 서비스 : proc 관리, Memory 관리, file 관리, I/O 관리
Proc 관리를 위해 Process Table(proc 마다 proc정보 저장 위한)이 필요함.
+Memory 관리를 위해 Memory Table 또한, 필요하며 page, seg 단위마다 Table 사용함
*PCB(Process Table) 구성
-process state
-process number ex. 학번, 주민번호
#p.c 등 register값은 CPU에 존재하는 것이지만, 현재 CPU를 할당받은 proc가 이용하게 됨. 이때, CPU는 하나지만, proc는 여러 개가 존재하여 CPU 할당받고 진행하다 중단되는 process 수행 과정에서 자신이 어디까지 수행했는지 보관할 필요가 있다. (CPU에서 보관 X)
따라서 PCB에 이를 저장해야 함
-process counter
-register(p.c 외의 register)
-memory limits : 프로세스마다 사용하는 메모리 영역
-list of open files : File Table(Stable Storage)에 access 하여 open한 file들의 정보

Threads
-한 proc 내 prog을 수행하는 action(자원)을 공유하는 것 ex. 채팅 프로그램(채팅 입/출력)

같은 프로그램을 수행할 때, proc 여러 개를 통해 진행하면(+IPC) 프로세스 자원 소모 및 overhead가 많음 ->한 proc 내 prog이 수행하는 동작을 여러 개 만들어 한 proc에 여러 action이 존재하도록 만든 것 (multiThreads)
*multiThreads 장점
자원 적게 차지
overhead 줄일 수 있음
proc 내 data 통해 Thread들이 R/W함으로써 IPC보다 빠르게(Shared Memory같은 효과)

#동시에 여러 작업을 수행하는 점에서 두 방법은 동일하지만, 각각의 장단이 있으므로 적용하는 시스템에 따라 적합한 동작 방식을 선택하고 적용 필요
+ Multiprocess vs. MultiThreads
a. 멀티 스레드 : 멀티 프로세스보다 적은 메모리 공간을 차지하고 Context Switching이 빠른 장점이 있지만, 동기화 문제와 하나의 스레드 장애로 전체 스레드가 종료될 위험 O
b. 멀티 프로세스 : 하나의 프로세스가 죽더라도 다른 프로세스에 영향을 주지 않아 안정성이 높지만, 멀티 스레드보다 많은 메모리 공간과 CPU 시간 차지

Process Representation in Linux
-리눅스에서 (C 구조)task_struct로 PCB가 표현된다
-PCB 구조체(C 언어)
pid t_pid;
long state;
unsigned int time_slice;
struct task_struct* partent; (어떤 proc는 자신과 동일한 proc 복제 가능 - fork())
struct list_head children;
struct files_struct* files;
struct mm_struct *mm;

2. Process Scheduling(=CPU scheduling)
여러 프로세스가 동작하다 보니, CPU 할당 권한을 가지고 프로세스의 스케줄링 진행하는 것
[proc는 여러 개지만, CPU는 하나] -> 어느 proc를 언제 수행할 것인가?를 결정해줌
CPU 효율을 최대화하는 것

프로세스들의 Scheduling Queues(어떤 자원이 있을 때 이를 기다리는 것)를 유지하는 것
Scheduling queue의 2종류
-Ready queue : CPU 할당을 기다리는 queue
-Wait queue : 어떠한 event를 기다리는 queue(event마다, I/O 마다 Wait queue 존재)

앞선 Queue의 linked list 표현- 14p
 어떠한 proc가 CPU를 기다리는지 표현할 때, queue header의 head가 PCB들의 head 부분과 연결되는 모습

여러 Queue의 종합적인 표현- 15p
Queuing Diagram, O-service, queue-서비스 기다리는 대기열

CPU Switching (proc -> proc)
-CPU가 여러 proc를 scheduling(proc 수행 순서 결정)하는데, 이 스케줄(시간 계획)에 따라 proc의 CPU 사용권을 넘겨주는 동작

context switch
proc0->proc1로 전환하는 시점이 되면, proc0 상태를 PCB에 잘 저장해야 함(context save)
-context : proc의 모든 살아있는 정보, 진행하던 상태 ex. p.c 등의 register 값
-reload(=resume) : PCB에 save한 정보를 다시 CPU에 loading하는 것
돌던 것을 잘 보관하고 돌려고 하는 것을 resume하는 동작

Moblie System에서의 Multitasking
Multiprogramming : Memory에 여러 proc를 CPU에 loading하는 것
Multitasking : time sharing에 의해 짧은 시간 CPU 사용권이 넘어가는, 그러니까 CPU 시간을 쪼개 사용하는 것
*process 구분
-foreground : UI에서 사용자와 인터렉션이 있는 프로세스 ex. 문서 작성기
-background : UI 없이 뒤에서(사용자와 인터렉션 X) 동작하는 프로세스
+안드로이드에서는 중간 단계의 service라는 프로세스를 두었음
service : 원래 background이지만, 제한적으로 사용자가 그 프로세스 수행을 감지할 수 있는 프로세스

3. Operations on Processes
다양한 프로세스의 Operation (생성 - 수행 - 종료, 프로세스의 시작과 끝)

(1) proc 생성
Unix나 Linux 시스템은 Parent - Children Relationship이 있어 fork()로 child 생성 가능
Parent, Children는 모든 것이 같지만, proc identifier(pid)는 다르다.
*fork()사용 이유
Parent 혼자 수행하는 것보다 필요한 만큼 child 생성해 여러 개의 proc를 동시 수행(concurrently, 동시성)하는 것이 성능상 이득임 ex. 웹서버, file 서버에 여러 사용자의 요청

리눅스에서의 Tree of Process
-systemd(시스템 데몬)이라는 태초의 proc가 create에 의해 생성
-이후, 모든 proc는 systemd의 fork()를 통해 tree 구조 형태로 생성
*systemd이 fork()를 통해 생성해도, exec()로 실제 다른 프로그램으로 대체되어 수행됨.
ex. systemd -> logind

->리눅스 시스템은 systemd이 최초 생성되고, fork(), exec()를 통해 다양한 프로그램 생성
exec() : 프로세스 대체(원래 프로세스는 멈추고, 새로운 프로세스로 대체)
fork()와의 차이점 : 새로운 proc를 생성(메모리 할당 필요O)하는 것이 아니라, exec()를 호출한 프로세스에 exec()에 의해 호출된 프로세스를 덮어쓰는(메모리 할당 필요 X) 형식
-proc의 fork() 이후, parent와 child proc 구분 방법 : [pid > 0, = 0]인지 확인하는 것
#C prog으로 fork()하는 과정
-Parent program이 main을 수행하고 fork()하여 child proc 만듦
-if문을 통해 pid를 이용하여, p, c를 구분하고 각각 할 일을 programming
-parent의 경우 child 종료 대기하는 내용 필수

(2) proc 종료
-프로그램 끝에 도달하거나 exit() 등으로 인해 proc 종료
(exit()의 경우, child가 종료되었을 때 wait하는 parent에게 신호로써 이용됨)
-abort()는 child의 여러 가지 문제가 있어 종료할 필요가 있을 때 사용
*Cascading Termination
일부 parent proc가 종료될 때, child proc 종료시키는 OS 존재함
-Parent가 종료됨으로써, 그 밑의 child, grandchild가 연쇄적으로 종료되는 것
-child 비정상 종료로 인해 zombie 상태 되거나 parent 먼저 종료되어 orphan(고아) 상태됨

#안드로이드에는 여러 가지 proc 계층 존재
-Foreground process
-Visible process : 동작 상태를 Foreground를 통해(간접)으로 볼 수 있는 프로세스
-Service process : 원래 Background나 간접적으로 동작 감지 가능 프로세스 ex. 음악
-Background process
-Empty process

#Multiprocess Architecture - 크롬 브라우저 예시
크롬에서 프로세스가 복잡한 일 모두 처리하는 것이 아니라, 다양한 프로세스로 분리됨.
-Browser : UI나 disk와 network I/O 관리
-Renderer : 웹 사이트 오픈 시마다 독립적으로 생성되어, 웹페이지 생성or HTML 처리 등 수행
-Plug-in : 미디어 파일 재생 등의 다양한 용도의 프로그램 수행

4. Interprocess Communication
컴퓨터 상에서 다양한 프로세스가 동작하는데, 이 process 간 통신도 가능하다.
시스템 내의 프로세스는 2가지의 형태 가능
-Indepedent ex. 문서 작성 시, 음악 프로그램 듣는 것
-Cooperating ex. C-S model, Web browser-Renderer의 multiprocess Architecture 등
어느 정도 규모가 있는 Prog의 경우, Interprocess communication 사용 (proc 통신)

#Proc 간 협력해야 하는 이유
-정보 교환
-성능 향상(여러 proc가 동시 수행하면)
-모듈화
(한 proc로 작성하면 전체가 한 덩어리이므로, 프로그램 관리,유지,보수 어려우므로)
-편의성

#OS에서 IPC에 대한 API 제공하며, 2가지 모델 존재(설명 뒤에)
-Shared-Memory model
-Message-Passing model

Proc 간 Interface하는 것을 몇 가지 모델로 나눌 수 있음
Producer(데이터 생성)-Consumer(데이터 소비) Problem ex. Printer 작업
이 개념 기반으로 IPC 두 가지 모델 접근 가능(여러 proc가 p-c 입장에서 접근한다는 것)
공유하는 공간이 필요하다는 것(우리가 제시한 예시에서는 M.P든 S.M이든 공유 공간 존재)
그런 것 없이 직접 proc 간 직접 메시지 전달하는 경우라면, Producer-Consumer 문제 X
-이용 buffer 종류
unbounded-buffer : 무한정 큰 버퍼, bounded-buffer : 제약이 있는 버퍼

#IPC(Inter Process Communication) 하는 방법에는 2가지 존재
5. Shared-Memory 시스템 내 IPC
공유 메모리를 통해 두 가지 이상의 process가 메모리의 W/R를 하면서 통신을 진행함.
ex. 게시판, 벽보나 BlackBoard [R-W 방식]
#주의할 점 : 동기화 (여러 proc가 data에 대해 access할 때, 충돌 가능하므로) ex. 예약

Shared-Memory의 C언어 코드 표현
-item이라는 구조체 사용
(buffer size - 1)만큼 data를 저장하는 공간
-in(생산자가 생산한 data를 write하는 P.ter), out(소비자가 data read하는 P.ter)변수 이용
-buffer 배열 이용
-modular 연산 및 while 문

6. Message-Passing 시스템 내 IPC
OS가 제공하는 기능을 통해 프로세스 간 직접 Message를 주고 받음
-(커널 영역에 존재하는 msg queue 이용) 실제 송신(send)할 msg queue의 주소를 약속하고 이를 다른 proc에서 가져감(receive)으로써 수신함. [send - receive 방식]
-다른 방법들(다른 buffer 활용)
*OS는 보내고 받는 프로그래밍할 수 있도록 IPC API 제공 (send(msg), receive(msg))

※Message-Passing에서 proc P와 Q가 communication할 때, 필요한 조건 및 옵션
(1)communication link(전화할 때, 전화 연결 과정) 필요
(2)data send/receive(실제 통화)

@구현 옵션
-connection 연결 방법
-하나의 link의 2개 이상 process 연결 가능 유무
-상호작용하는 process들의 소통 짝마다 존재 가능한 link 개수
-link의 수용 용량
-link가 수용가능한 msg의 크기의 고정 유무
-link가 단방향(unidirectional)인지 양방향(bidirectional)인지

※communication link 구현
(1) 물리적 구현 (OS에서 알아서 해줌)
-Shared memory
-Hardware bus
-Network
(2) 논리적 구현
-Direct or indirect
-Synchronous or Asynchronous
-Automatic or Explicit buffering

Direct Communication
당사자로부터 or 목적지로 직접 msg 주고 받는 Communication
-send(P, msg) : P에게 msg를 전달해라(P:목적지)
-receive(Q, msg) : Q로부터 msg를 받아라
D.C의 여러 가지 옵션(programmer가 목적에 맞게 선택해서 적용)

Indirect Communication ex. 메일 박스, 게시판
제 3의 공간에 data를 주고, 받는 사람도 그 공간으로부터 data 읽어오는 communication
-각 mailbox는 고유 ID 보유
-proc는 오직 mailbox를 통해 소통 가능
I.C의 여러 가지 옵션(programmer가 목적에 맞게 선택해서 적용)
*Operation
mailbox 생성, mailbox로부터 msg 송수신, mailbox 삭제
-send(A, msg) : A에게 msg를 전달해라(A: mailbox)
-receive(A, msg)
@여러 사용자들이 mailbox를 사용할 때 충돌 문제 발생 가능(-> 동기화 필요성)

Synchronization

(1) Blocking vs. Non-blocking 구분 (완성 시까지 대기 유무)

	Blocking	Non-blocking
정의	송수신 시, Send-Receive 완성될 때까지 기다리는 것	송수신 시, Send-Receive 완성되지 않아도 반환하는 것
종류	-Blocking Send 메시지를 받을 때까지 송신자는 block됨 -Blocking Receive 메시지가 올 때까지 수신자는 blocked됨	-Non-blocking Send 일단 메시지를 보냄 -Non-blocking Receive 메시지가 없어도 일단 Return함

엄밀히 말하면, 위의 Blocking, Non-blocking은 synchronous이다.

asynchronous(알림 기능)는 자신이 Receive 했다면 바로 return이 되고, data가 도착 시에 signal을 통해 알려줘, data를 받을 수 있는 것

<일반적으로, Blocking은 synchronous 고려하고, Non-blocking은 asynchronous 고려함>

그러니까, blocking도 asynchronous 사용 가능한데, 일반적인 고려 상 synchronous 사용

Buffering

양 단(CPU - I/O)의 시간/속도/처리 단위 차이가 많이 존재하는데, 이를 완충(화)해주는 것

-Zero capacity : 버퍼 없는 것

-Bounded capacity : 버퍼 크기 한계 존재하는 것

-Unbounded capacity : 버퍼 크기 한계 없는 것(다 쓰면 앞자리로 다시 되돌아감)

7. IPC 시스템 사례 -실제 OS에서 어떤 IPC 메커니즘을 제공하는 API [System call 등 API]

(1) POSIX의 Shared-Memory

-shm_fd = shm_open(name, O CREAT | O RDWR, 0666);

shm_open이라는 API를 통해 shared Memory 생성

-ftruncate(shm_fd, 4096);

ftruncate를 통해 shared Memory size 설정

-ptr = mmap(0, SIZE, PROT WHIE, MAP SHARED, shm_fd, 0);

mmap을 통해 shm_fd(file descrpiter, file open시 그 파일을 가르키는 인덱스)를 메모리 주소 공간으로 매핑시켜 줌 (proc가 memory 주소 공간을 사용하기 때문에..)

(2) Mach(카네기 멜론에서 만든 OS 커널)에서 제공하는 IPC API

-mach_msg()를 통해 data 주고 받음

-mach_port_allocate를 통해 data 주고받기 위한 communication port 설정

-> Client에서 사용할 port, Server에서 사용할 port 설정 필요

Client에서 msg 보내고 Server에서 msg 받음

(3) Windows에서 제공하는 IPC API

LPC(advanced Local Procedure call - function call을 의미함) 제공

-범위 : 한 컴퓨터 내 서로 다른 proc라도, 다른 proc가 제공하는 함수 호출 가능

(function call은 원래, 한 proc 안에서 함수 호출이 가능하지만..)
※Communication 작동 선행 조건
-Client가 subsyetm의 connection port 열고 이를 통해 connection 요청
(S는 C가 Connection할 수 있는 handle 제공해야 하며 C는 이를 통해 연결 요청 가능)
-Server가 두 개의 private communication port 만들고 이를 통해 실제 data 주고받음

#function call이 proc 안에서 작동하는 이유
prog 모듈성, 유연성 향상을 위해 규모 있는 prog 작성 시, main에다 prog 다 넣지 않고 function, 기능별로 다 분리함 (따라서 func tion call 시, 한 process 내 호출하게 됨)
*RPC(remote procedure call) : 원격지의 컴퓨터에 대한 function call

Pipes
-한 proc가 다른 proc에게 data를 줄 때, 물이 흘러가듯이 data를 전달할 수 있는 것
a. Ordinary pipes는 parent-child 관계가 존재해야만 가능함
#Ordinary pipe 통신 과정
(1)Pipe를 P가 먼저 생성
(2)P가 C를 fork() [C는 생성되면서 P가 생성했던 Pipe를 그대로 물려받음]
(3)P는 생성했던 Pipe를 통해 C와 통신 가능
@Pipe 실제 사용 과정
pipe를 만들 수 있는 pipe()라는 sys.call(API) 제공됨
이 pipe()는 fd(file descriptor)를 매개변수를 필요로 하며, 이는 int[] 형태임
(fd를 먼저 int배열 형태로 선언해 놓고 매개변수로 활용)
일반적으로, fd[0]에서 read하고, fd[1]에서 write함 : read(fd[0], -), write(fd[1], -)

b. Named pipes는 P-C 관계없이도 만들 수 있는 Pipe
FIFO 특성(Queue처럼)이 있으며, 한 컴퓨터에 있는 일반적 proc가 data 주고받을 수 있음
file처럼 pipe에 이름 존재하여, p-c관계 없이 이름을 가지고 open을 함

#시스템 내 Memory 및 함수 호출 정리(교수님 판서)

	Memory 관점	function call 관점
Thread	global 변수 저장하는 메모리로 공유할 수 있음	O (가능)
Process	메모리는 프로세스 안에서 사용	O (가능)
One System	Shared Memory 사용	local procedure call(LPC)
Multicomputer	Distributed Shared Memory 사용	Remote procedure call(RPC)

8. Client-Server 시스템 내 Communication
분산 환경에서 여러 process가 동작하는 모델이 있다. -> 대표적 C-S model
여러 Client가 서버에게 요청하고 응답받는 시스템에서 IPC가 어떻게 이용되는가?
물론, P2P, pub/sub 모델도 존재하며 IPC 이용됨

#Socket, RPC : 원격지에 있는 컴퓨터 간 통신을 통해 사용할 수 있는 것

Socket :(Nt 환경) AP 프로그래머가 TCP/IP를 쉽게 이용 가능하도록 제공하는 네트워크 API
-IP 주소 : 컴퓨터 식별
-Port Number : 서비스 및 proc 식별 ex. 161.25.19.8(IP) : 1625(Port)
-loopback 주소(lo) : 한 대의 컴퓨터에서 자체적으로 인터넷 테스트할 때 사용하는 주소
*Socket(TCP, UDP 다 가능)을 이용한 쉬운 API 제공되기도 함 ex. Java, Unix 등
TCP(연결지향성, 연결 후 통화 지속할 때 유리) - UDP(비연결성, 단발성 msg 전송 시 유리)
@Java Socket 예시 – DateServer, DateClient
(1) DateServer가 소켓(IP주소, Port Num)을 먼저 생성
(2) Client 쪽에서 connection 요청이 오는 것을 accept 하고자 대기함.
(3) 연결 설정 후에는 Send(), Receive(), Write(), Read() API를 통해 data 주고받음
Client 코드 내, 소켓 생성 및 connection 요청 API(Java에서는 한 번에 지원함)
- Socket sock = new Socket("127.0.0.1", 6013);

RPC : 서로 다른 컴퓨터의 proc끼리 function을 호출할 수 있는 것
RPC의 경우, 서로 원격지에 있으므로, 어떤 function을 어떤 매개변수와 call 하는지 통신(Socket interface, 네트워크 상 통신)을 통해 data를 전달할 필요 O
@RPC가 일종의 M/W인 이유
C에서 S의 function을 call 시, C와 S는 AP prog이라 네트워크 통해 주고받아야 하므로
-Stubs : RPC 동안 매개변수를 변환에 사용되는 코드(f.c 과정 도와주는 매개체)
※Stubs이 function call 매개하는 과정
C가 C의 Stub에 대해 function call 시, 이것이 복잡한 네트워크를 통해 C가 어떤 함수에 대해 어떤 매개변수를 사용했는지 msg를 통해 S의 Stub로 보내고, S의 Stub는 실제 function call 진행 (C, S 모두에게 존재)
-marshalling : C가 부른 함수 및 매개변수들을 C의 Stub가 msg 형태로 바꾸어 보내는 것
-matchmaker
C가 S(원격지)의 함수 호출 시, 어느 컴퓨터의 어느 proc가 자신이 호출하고자 하는 원격지 함수를 가지고 있는지 찾아주는 역할
(일종의 디렉터리 서버, 누가 어떤 RPC을 제공하는지에 대한 정보 제공)

-MIDL(Microsoft Interface Definition Language) : Windows의 stub prog 표준
OS마다 제공하는 RPC M/W가 있고, M/W마다 Stub쪽 prog의 표준이 정의되어 있음
-XDR(External Data Representation)
컴퓨터마다 데이터 포맷(Big-endian, little-endian 등)이 다르므로, RPC은 이런 것들을 서로 변환하여야 할 필요 존재하며, 이를 변환해주는 것
a.Big-endian : 큰 단위가 앞에 나오는 것
b.little-endian : 작은 단위가 앞에 나오는 것

<u>M/W : 미들웨어는 서로 다른 애플리케이션이 서로 통신하는 데 사용되는 소프트웨어 (범용)</u>