[TCP/IP] Chapter 10 - 13

[GBC20190027] Network

27기 최하영

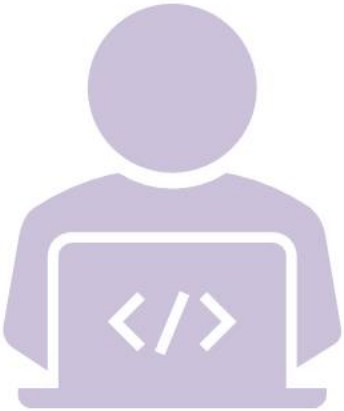
Agenda

Chapter10. 멀티 프로세스 기반의 서버 구현

Chapter11. 프로세스간 통신

Chapter12. IO 멀티플렉싱

Chapter13. 다양한 입출력 함수들



10-1. 프로세스의 이해와 활용

1. 다중 접속 서버의 구현 방법들

- 둘 이상의 클라이언트에게 동시 접속 허용 및 서비스 제공하는 서버

1) 멀티 프로세스 기반 서버
   - 다수의 프로세스를 생성하는 방식으로 서비스 제공

2) 멀티 플렉싱 기반 서버
   - 입출력 대상을 묶어서 관리하는 방식으로 서비스 제공

3) 멀티 쓰레딩 기반 서버
   - 클라이언트의 수만큼 쓰레드를 생성하는 방식으로 서비스 제공

2. 프로세스의 이해

1) 프로세스

- 메모리 공간을 차지한 상태에서 실행중인 프로그램
- 멀티프로세스 운영체제는 둘 이상의 프로세스를 동시에 생성 가능

2) 프로세스 ID
   - 운영체제는 생성되는
모든 프로세스에 ID를 할당함.

```
choehayeong-ui-MacBookPro:~ hayeong$ ps au
USER       PID  %CPU %MEM      VSZ    RSS   TT  STAT STARTED      TIME COMMAND
root     15363   0.3  0.0  4268468   1032 s003  R+    5:24AM   0:00.00 ps au
hayeong  15354   0.1  0.0  4288336   1608 s003  S     5:24AM   0:00.02 -bash
root     15353   0.0  0.1  4304648   5848 s003  Ss    5:24AM   0:00.80 login -pf
hayeong  12224   0.0  0.0  4267908    500 s002  S+    9:30PM   0:00.00 ./server
hayeong  10128   0.0  0.0  4288336    720 s002  Ss    4:12PM   0:00.03 /bin/bash
hayeong   7347   0.0  0.0  4288336    440 s001  Ss+  수 12AM   0:00.02 /bin/bash
choehayeong-ui-MacBookPro:~ hayeong$
```

10-2. 프로세스 & 좀비 프로세스

1. 좀비 프로세스

1) 의미
- 실행이 완료되었음에도, 소멸되지 않은 프로세스
- 소멸되지 않은 → 프로세스가 사용한 리소스가 메모리 공간에 남아 있다는 의미
- main()이 반환되면 프로세스도 소멸되어야 함

2) 생성 원인
- fork 함수의 호출로 생성된 자식 프로세스가 종료되는 상황
  - 인자를 전달하면서 exit을 호출하는 경우
  - main()에서 return문을 실행하면서 값을 반환하는 경우

→ 자식 프로세스가 종료되면서 리턴하는 상태 값이 부모 프로세스에게 전달되지 않으면 해당 프로세스는 좀비가 됨

3) 생성 확인

→ 자식 프로세스의 종료 값을 리턴 받을 부모 프로세스가 소멸되면, 좀비의 상태로 있던 자식 프로세스도 소멸되기 때문에, 부모 프로세스가 소멸되기 전에 좀비의 생성을 확인해야 함

10-2. 프로세스 & 좀비 프로세스

2. 좀비 프로세스의 소멸

1) wait 함수의 사용

```
#include <sys/wait.h>
pid_t wait(int *statloc);
//성공 시 자식 프로세스의 ID, 실패시 -1 반환
```

* WIFEXITED | 자식 프로세스가 정상 종료한 경우 true 리턴
* WEXITSTATUS | 자식 프로세스의 전달 값을 반환

→ 자식 프로세스가 종료되지 않은 상황에서는 반환 X, 블로킹 상태에 놓임

2) waitpid 함수의 사용

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *statloc, int options);
//성공 시 종료된 자식 프로세스의 ID, 실패시 -1 반환
```

→ wait 함수의 블로킹이 문제가 된다면 waitpid 함수의 호출 고려

10-3. 시그널 핸들링

1. 시그널

- 자식 프로세스의 종료라는 상황 발생시, 특정 함수의 호출을 운영체제에게 요구하는 것
  - 특정 상황이 되었을 때 운영체제가 프로세스에게 해당 상황이 발생했음을 알리는 메시지

2. 시그널 등록

- 특정 상황에서 운영체제로부터 프로세스가 시그널을 받기 위해서는 해당 상황에 대해 등록의 과정이 필요!

```
#include <signal.h>
void (*signal(int signo, void (*func)(int)))(int);
// 시그널 발생시 호출되도록 이전에 등록된 함수의 포인터 반환
```

→ 시그널 등록에 필요한 함수
시그널이 등록되면, 함께 등록된 함수의 호출을 통해서 운영체제는 시그널의 발생을 알림

10-3. 시그널 핸들링

3. Sigaction 함수를 이용한 시그널 핸들링

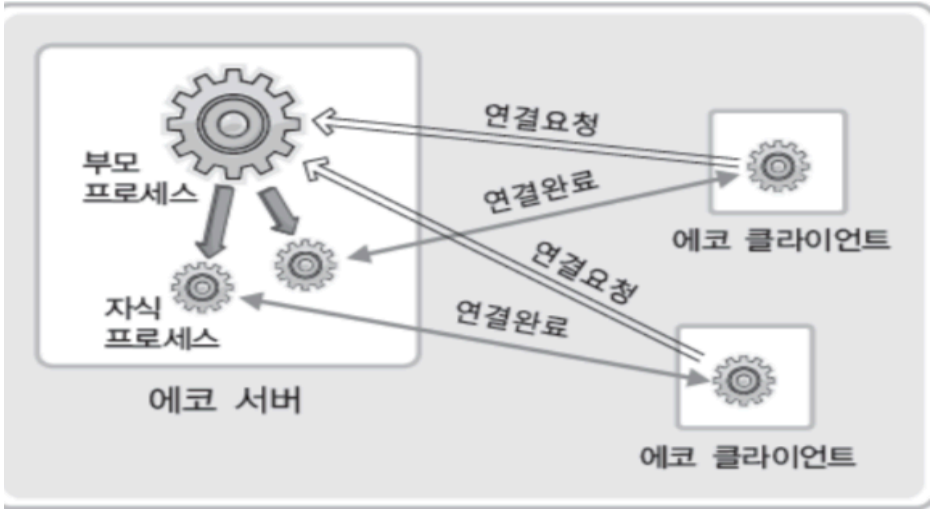
```
#include <signal.h>
int sigaction(int signo, co st struct sigaction *act, struct sigacion *oldact);
```

- signal() 함수를 대체할 수 있고, signal()보다 훨씬 안정적으로 동작
  → signal 함수는 유닉스 계열의 운영체제 별로 동작 방식에 있어서 약간의 차이를 보일 수 있지만, sigaction 함수는 차이를 보이지 않는다.

10-4. 멀티태스킹 기반의 다중접속 서버

1. 프로세스 기반의 다중접속 서버의 구현 모델

- 이전 에코 서버는 한번에 하나의 클라이언트에게만 서비스 제공 가능
  (동시에 둘 이상의 클라이언트에게 서비스를 제공하지 못하는 구조)
  --→ 동시에 둘 이상의 클라이언트에게 서비스를 제공하는 형태로 에코 서버 확장



1) 에코 서버는 accept 함수호출을 통해 연결요청 수락
2) 이때 얻게 되는 소켓의 파일 디스크립터를 자식 프로세스를 생성하여 넘겨줌
3) 자식 프로세스는 전달받은 파일 디스크립터를 바탕으로 서비스 제공

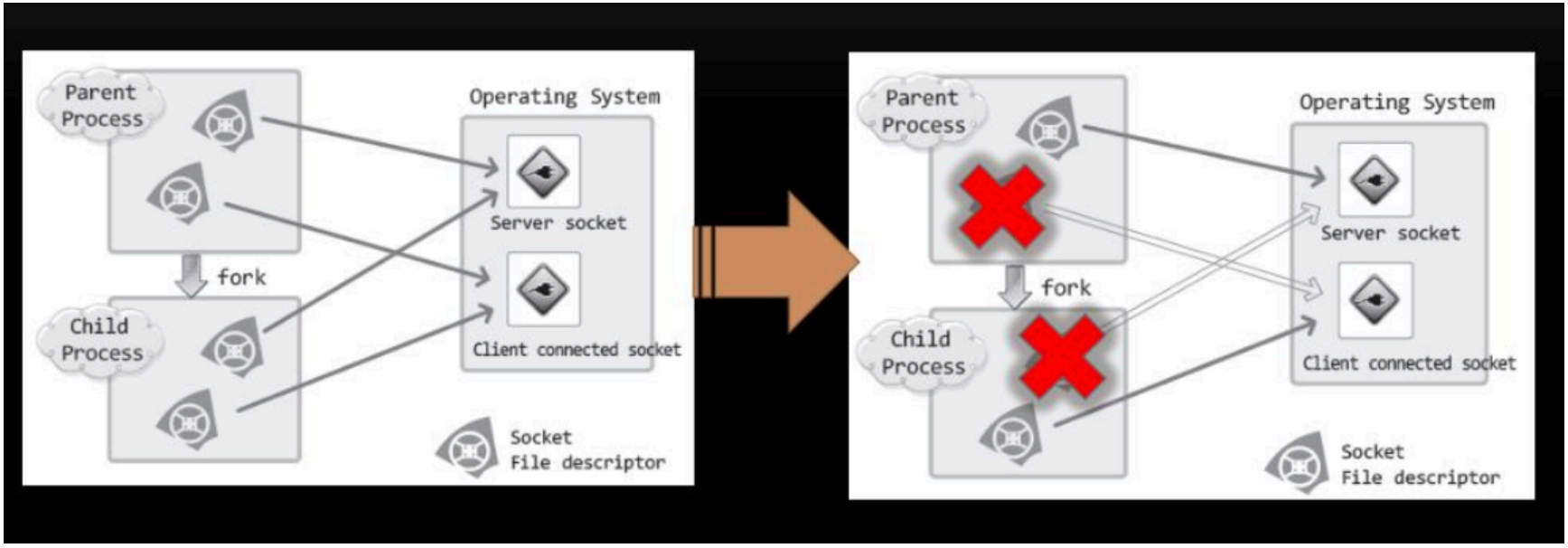
⭐) 연결이 하나 생성될 때마다 프로세스를 생성하여 해당 클라이언트에게 서비스 제공

10-4. 멀티태스킹 기반의 다중접속 서버

2. fork 함수 호출을 통한 디스크립터의 복사

- 프로세스가 복사되는 경우 해당 프로세스에 의해 만들어진 소켓이 복사 (X)
  파일 디스크립터가 복사됨 (O)

→ 하나의 소켓에 두 개의 파일 디스크립터가 존재하는 경우,
두 파일 디스크립터 모두 종료되어야 해당 소켓 소멸, 그래서 fork 함수 호출 후에는
서로에게 상관없는 파일 디스크립터를 종료함



10-5. TCP의 입출력 루틴 분할

1. 입출력 루틴 분할의 의미

- 소켓은 양방향 통신이 가능함

→ 입력을 담당하는 프로세스와 출력을 담당하는 프로세스를 각각 생성할 경우, 입력과 출력을 각각 별도로 진행시키는 것이 가능해진다.

- 입출력 루틴을 분할하면,
  보내고 받는 구조가 아닌 동시 진행이 가능해진다.

11-1. 프로세스간 통신의 기본 개념

1. 프로세스간 통신

- 두 프로세스 사이에서의 데이터 전달을 의미한다.
- 데이터 전달이 가능하려면, 두 프로세스가 공유하는 메모리 존재

2. 프로세스간 통신의 어려움

- 모든 프로세스는 자신만의 메모리 공간 독립적으로 구성
- 프로세스A는 프로세스B의 메모리 공간에 접근 불가능,
  그 반대도 불가능
  → 운영체제가 별도의 메모리 공간을 마련해줘야 프로세스간 통신 가능

11-1. 프로세스간 통신의 기본 개념

3. 파이프 기반의 프로세스간 통신

```
#include <unistd.h>
int pipe(int filedes[2]);
```

→ OS는 서로 다른 프로세스가 함께 접근할 수 있는 메모리 공간을 만들고,
파이프 접근에 필요한 파일 디스크립터를 리턴함

4. 프로세스간 양방향 통신 : 잘못된 방식

- 하나의 파이프를 이용하여 양방향 통신을 하는 경우,
data를 읽고 쓰는 타이밍이 매우 중요함
(파이프에 데이터가 들어가면, 임자 없는 데이터가 되는데,
read 함수 호출을 통해 먼저 데이터를 읽어 들이는 프로세스에게 데이터가 전달되므로)

11-1. 프로세스간 통신의 기본 개념

5. 프로세스간 양방향 통신 : 적절한 방식

→ 양방향 통신을 위해서는 두 개의 파이프를 생성해야 한다.
각각이 서로 다른 데이터의 흐름을 당당하게 하면 됨

11-2. 프로세스간 통신의 적용

*** 메시지를 저장하는 형태의 에코 서버**

→ 파이프를 생성하고 자식 프로세스를 생성하여,
자식 프로세스가 파이프로부터 data를 읽어들여 저장하도록 구현

→ accept 함수 호출 후, fork 함수호출을 통해서 파이프의 디스크립터를 복사하고,
이전에 만들어진 자식 프로세스에게 데이터를 전송

12-1. IO 멀티플렉싱 기반의 서버

1. 멀티 프로세스 서버의 단점과 대안

1) 단점

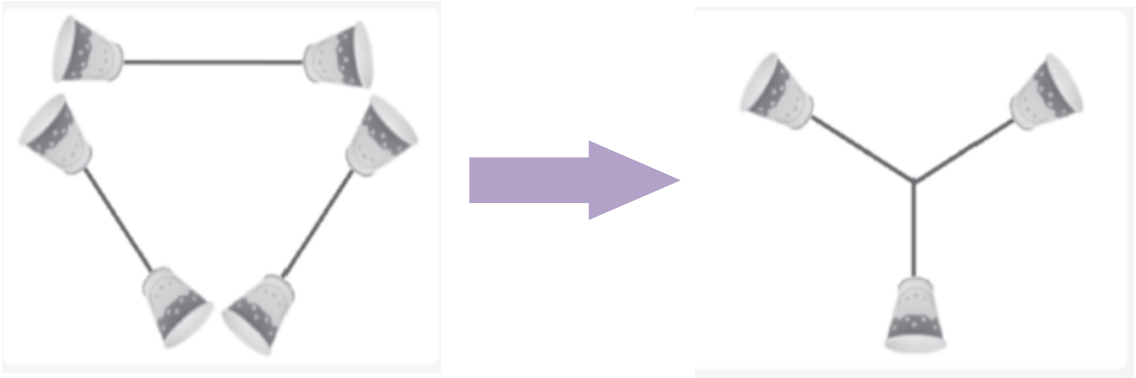
- 프로세스의 빈번한 생성이 성능의 저하로 이어진다
- 멀티 프로세스의 흐름을 고려해야 하므로, 구현이 어렵다
- 프로세스 간의 통신이 있어야 할 때, 서버의 구현이 복잡하다

2) 대안

- 하나의 프로세스가 다수의 클라이언트들에게 서비스를 제공할 수 있게 한다
  → 여러 개의 소켓을 핸들링 할 수 있는 방법 (IO 멀티 플렉싱)

2. 멀티 플렉싱

- 하나의 프로세스가 다수의 소켓을 관리

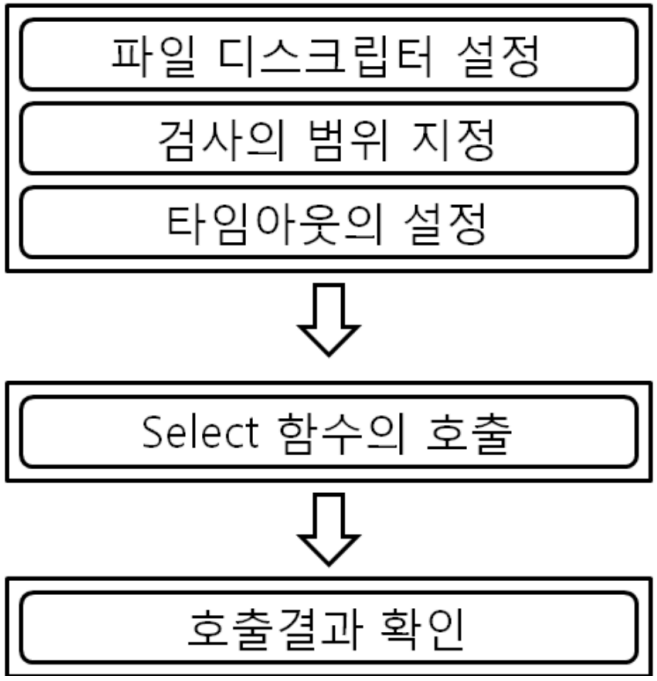


12-2. select 함수의 이해와 서버의 구현

1. Select 함수의 기능과 호출 순서

1) select 함수 이용할 때, 배열에 저장된 파일 디스크립터에서 관찰할 수 있는 항목
- 수신한 데이터를 지니고 있는 소켓이 존재하는가?
- 블로킹이 되지 않고 데이터의 전송이 가능한 소켓은 무엇인가?
- 예외 상황이 발생한 소켓은 무엇인가?

2) 호출 순서



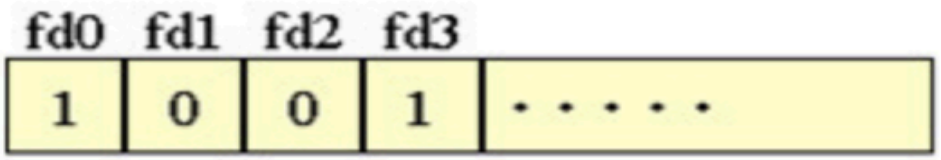
12-2. select 함수의 이해와 서버의 구현

2. 파일 디스크립터의 설정

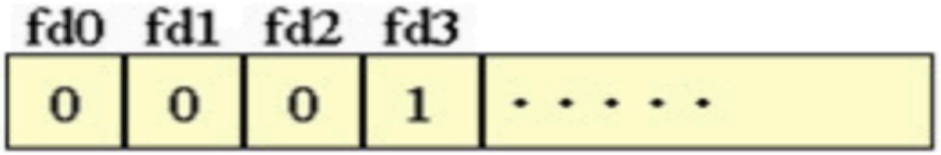
모두 0으로 초기화 → 디스크립터 1을 관찰 대상으로 추가 →
디스크립터 2를 관찰 대상으로 추가 → 디스크립터 2를 관찰 대상에서 제외

12-2. select 함수의 이해와 서버의 구현

3. Select 함수 호출 이후의 결과 확인



select 호출 전 readfds



select 호출 후 readfds

select 함수 호출 이후에는 변화 발생(입력받은 데이터 존재, 출력 가능 데이터) 한 소켓의 디스크립터만 1으로 설정되어있고, 나머지는 모두 0으로 초기화!!!

13-1. send & recv 입출력 함수

1. 리눅스에서의 send & recv

```c
#include <sys/socket.h>
ssize_t send(int sockfd, const void *buf, size_t nbytes, int flags);

#include <sys/socket.h>
ssize_t recv(int sockfd, void *buf, size_t nbytes, int flags);
```

→ 윈도우 기반 예제에서 사용해 온 것과 동일!

13-1. send & recv 입출력 함수

옵션(Option)	의 미	send	recv
MSG_OOB	긴급 데이터(Out-of-band data)의 전송을 위한 옵션.	●	●
MSG_PEEK	입력버퍼에 수신된 데이터의 존재유무 확인을 위한 옵션.		●
MSG_DONTROUTE	데이터 전송과정에서 라우팅(Routing) 테이블을 참조하지 않을 것을 요구하는 옵션, 따라서 로컬(Local) 네트워크상에서 목적지를 찾을 때 사용되는 옵션.	●	
MSG_DONTWAIT	입출력 함수 호출과정에서 블로킹 되지 않을 것을 요구하기 위한 옵션, 즉, 넌-블로킹(Non-blocking) IO의 요구에 사용되는 옵션.	●	●
MSG_WAITALL	요청한 바이트 수에 해당하는 데이터가 전부 수신될 때까지, 호출된 함수가 반환되는 것을 막기 위한 옵션		●

13-1. send & recv 입출력 함수

2. Urgent mode의 동작원리

- MSG_OOB가 설정된 데이터 전달되면, 운영체제는 시그널을 발생시켜 메시지의 긴급 처리가 필요한 상황임을 프로세스에게 알린다. (데이터를 수신하는 대상에게 데이터의 처리 독촉)

URG = 1 //긴급 메시지가 존재하는 패킷
URG Pointer = 3 // 긴급 메시지가 설정된 위치 정보

13-2. readv & writev 입출력 함수

1. readv & writev 함수의 사용

- 데이터를 모아서 전송하고, 모아서 수신하는 기능의 함수

→ writev 함수를 사용하면 여러 버퍼에 나뉘어 저장되어 있는 데이터를 한번에 전송할 수 있고, readv 함수를 사용하면 데이터를 여러 버퍼에 나눠서 수신할 수 있다.

```
#include <sys/uio.h>
ssize_t writev(int filedes, const struct iovec *iov, int iovcnt);

#include <sys/uio.h>
ssize_t readv(int filedes, const struct iovec *iov, int iovcnt);
```

→ Wirtev 함수를 이용하면, 함수의 호출 횟수를 줄일 수 있으며, 하나의 패킷으로 구성되어 전송될 확률이 높아지고, 전송 속도의 향상으로 이어질 수 있음!!!