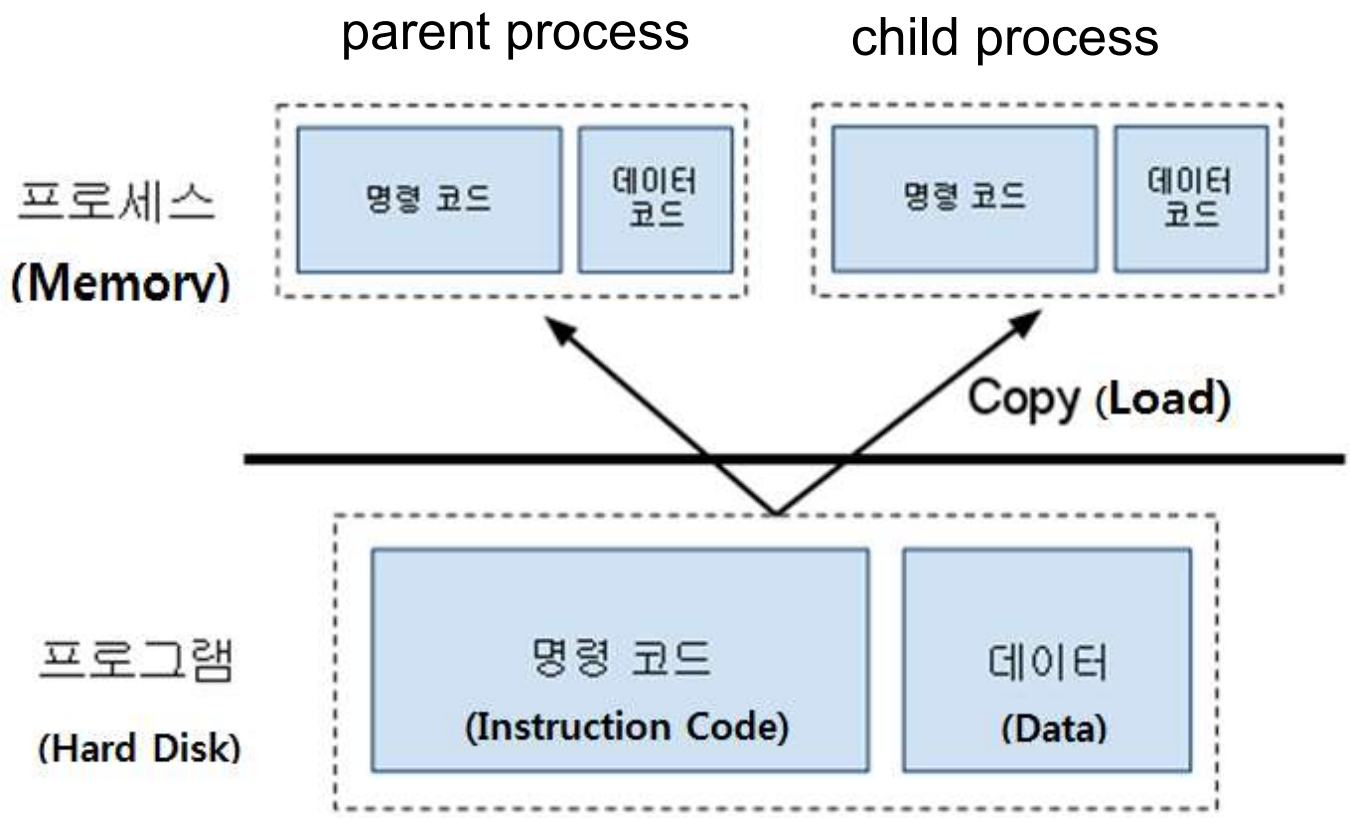
멀티 프로세스 소켓 프로그래밍

뇌를 자극하는 TCP/IP 소켓 프로그래밍

프로세스

- 최소 실행 단위 객체
- 프로그램을 메모리에 복사해서 실행
- 프로세스 → 프로그램의 실행 이미지



프로세스 구분

- 현대의 운영체제는 다수의 프로세스를 동시에 운용
- PID (Process ID)
  - 프로세스를 구분
  - 프로세스의 제어 단위 (PID로 시그널 전송)

```
파일  편집  보기  책갈피  설정  도움말
0 R  1000 24039  6625  0  80   0 -  1390 -      pts/3    00:00:00 ps
yundream@mypc:~$ ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 10:14 ?        00:00:00 /sbin/init
root         2     0  0 10:14 ?        00:00:00 [kthreadd]
root         3     2  0 10:14 ?        00:00:00 [ksoftirqd/0]
root         4     2  0 10:14 ?        00:00:00 [migration/0]
root         5     2  0 10:14 ?        00:00:00 [watchdog/0]
root         9     2  0 10:14 ?        00:00:01 [events/0]
root        11     2  0 10:14 ?        00:00:00 [cpuset]
root        12     2  0 10:14 ?        00:00:00 [khelper]
root        13     2  0 10:14 ?        00:00:00 [netns]
root        14     2  0 10:14 ?        00:00:00 [async/mgr]
root        15     2  0 10:14 ?        00:00:00 [pm]
root        17     2  0 10:14 ?        00:00:00 [sync_supers]
root        18     2  0 10:14 ?        00:00:00 [bdi-default]
root        19     2  0 10:14 ?        00:00:00 [kintegrityd/0]
root        21     2  0 10:14 ?        00:00:00 [kblockd/0]
root        23     2  0 10:14 ?        00:00:06 [kacpid]
```

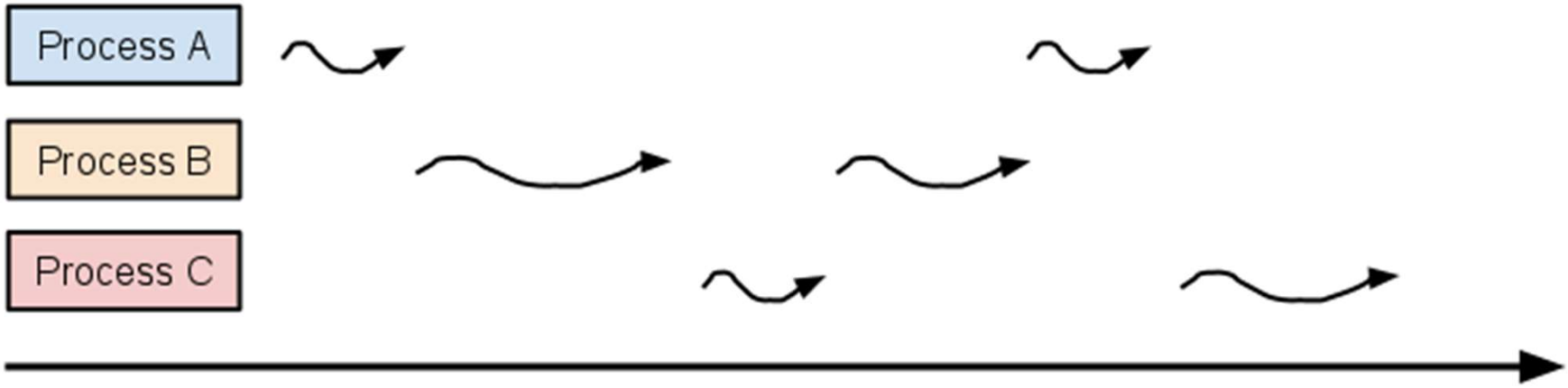
프로세스 제어

- 시그널을 이용한 프로세스 제어
  - 프로세스 중단
  - 중단된 프로세스의 재 가동
  - 프로세스 종료(메모리에서 제거)
- Signal을 이용
  - 터미널 상에서 kill 명령을 이용
  - C언어에서는 signal() 시스템 콜 이용

```
# ps -aux | grep myprog
yundream  25578  0.4  0.1  10040 3164 ?   S   22:00  ./myproc

# kill -9 25578
```

멀티 프로세스

- 동시에 여러 프로세스를 운용
  - 리눅스와 윈도우, 대부분의 운영체제들
  - 프로세스를 스위칭 함으로써 구현
  - 여러 프로세스가 CPU 시간을 나누어 사용(time sharing)

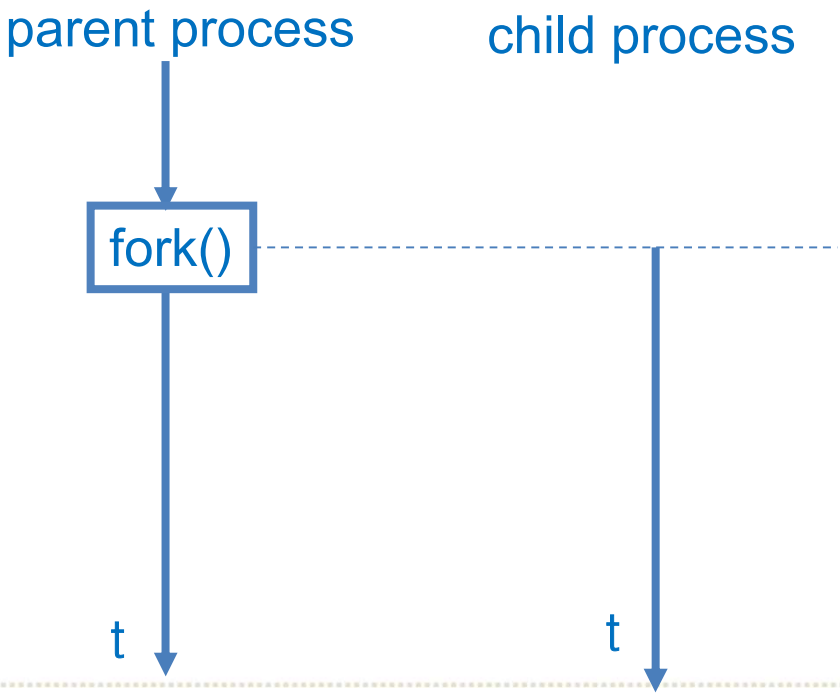


Linux fork()를 이용한 프로세스 복사

- fork 함수를 이용해서 프로세스를 복사

```
pid_t = fork();
```

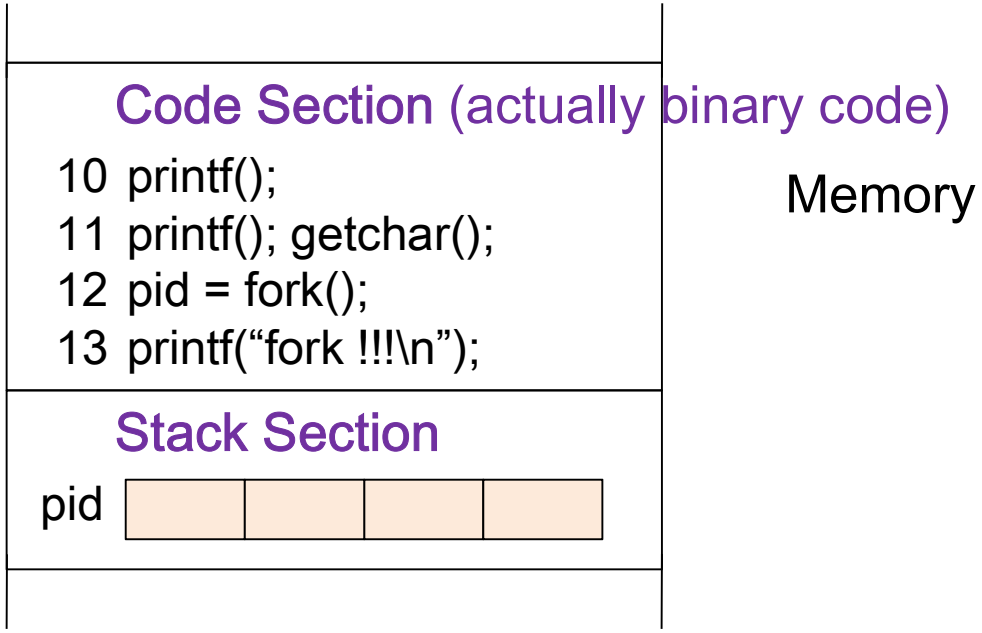
- 부모 프로세스 : fork 함수를 호출한 프로세스
- 자식 프로세스 : fork 함수로 복사된 프로세스



Linux fork()를 이용한 프로세스 복사

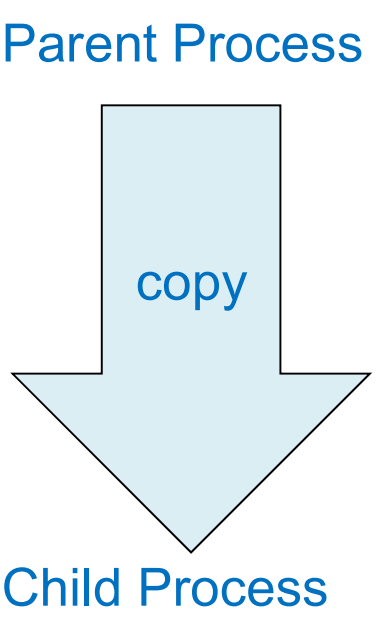
- before line12 pid=fork();

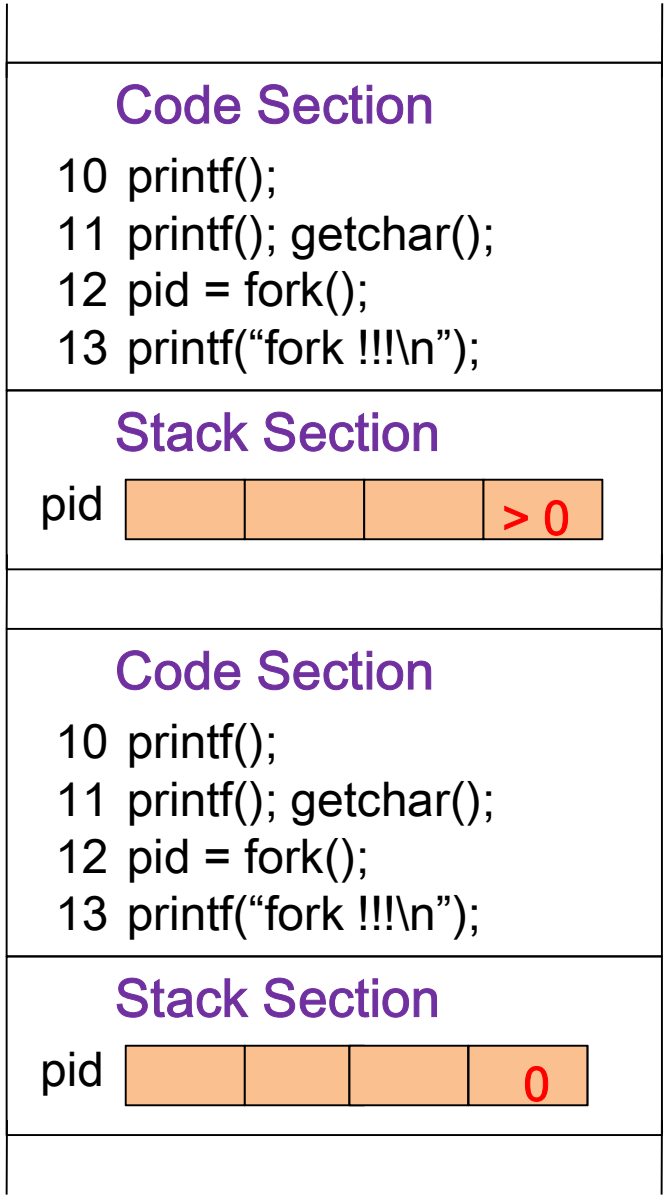
Parent Process



Linux fork()를 이용한 프로세스 복사

- after line12 pid=fork();





Memory

Linux fork()를 이용한 프로세스 복사

- 리턴 값으로 자식 프로세스와 부모 프로세스의 실행 코드를 결정
  - 자식 프로세스 : pid_t == 0
  - 부모 프로세스 : pid > 0

```
pid_t pid = fork();
if (pid > 0)
{
  // 부모 프로세스가 수행할 코드
}
else if (pid == 0)
{
  // 자식 프로세스가 수행할 코드
}
```

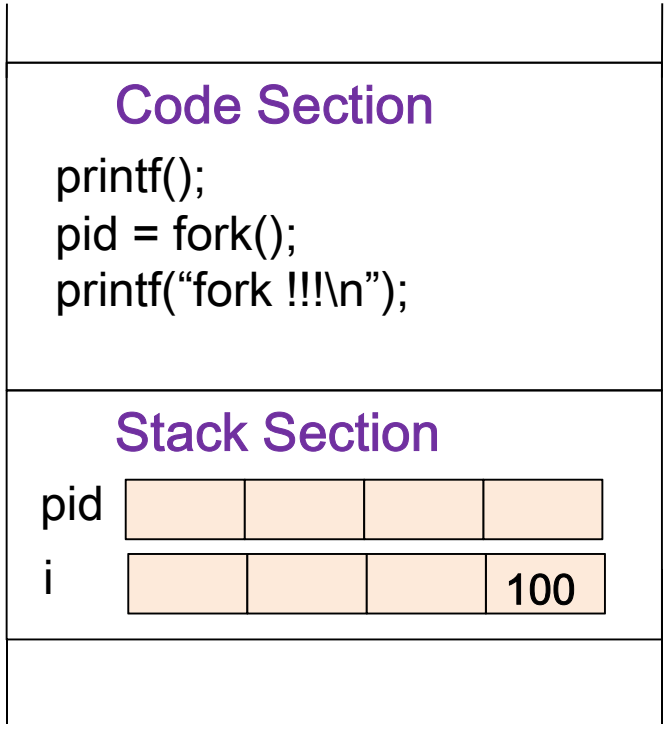
Lab. fork_test.c

```c
pid_t pid;
int i = 100;

printf("Program Start!!\n");

printf("hit a key "); getchar(); printf("\n");
pid = fork();
printf("fork !!!\n");
if (pid < 0)
{
        printf("fork failure\n");
        return 1;
}

if (pid > 0)
{
        printf("I'm parent Process %d\n", getpid());
        while(1)
        {
                printf("P : %d\n", i);
                i++;
                sleep(1);
        }
}
else if (pid == 0)
{
        printf("I'm Child Process %d\n", getpid());
        while(1)
        {
                printf("C : %d\n", i);
                i+=2;
                sleep(1);
        }
}
```

Lab. fork_test.c

- before pid=fork();

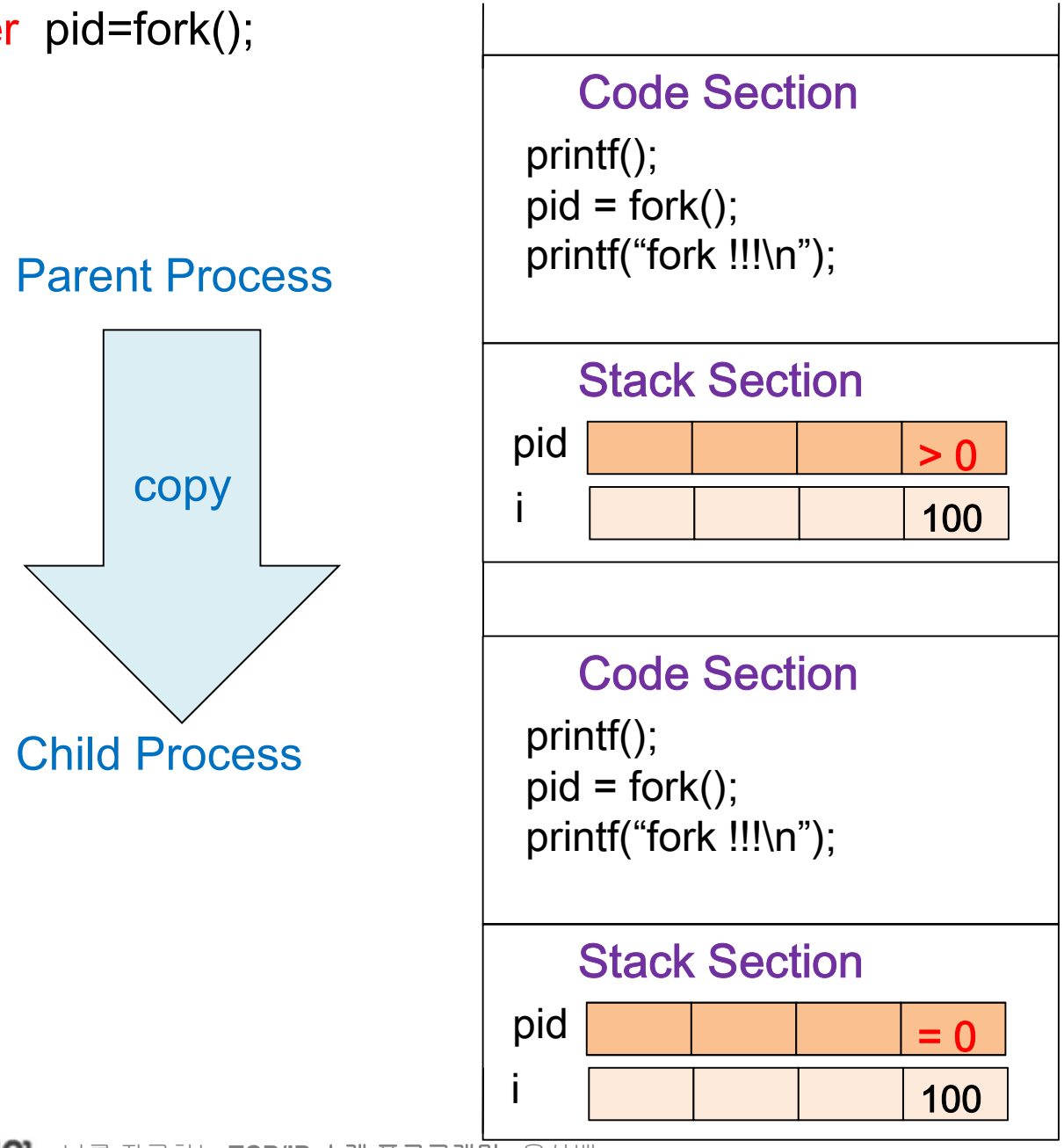
Parent Process



Memory

Lab. fork_test.c

- after pid=fork();



Lab. fork_test.c

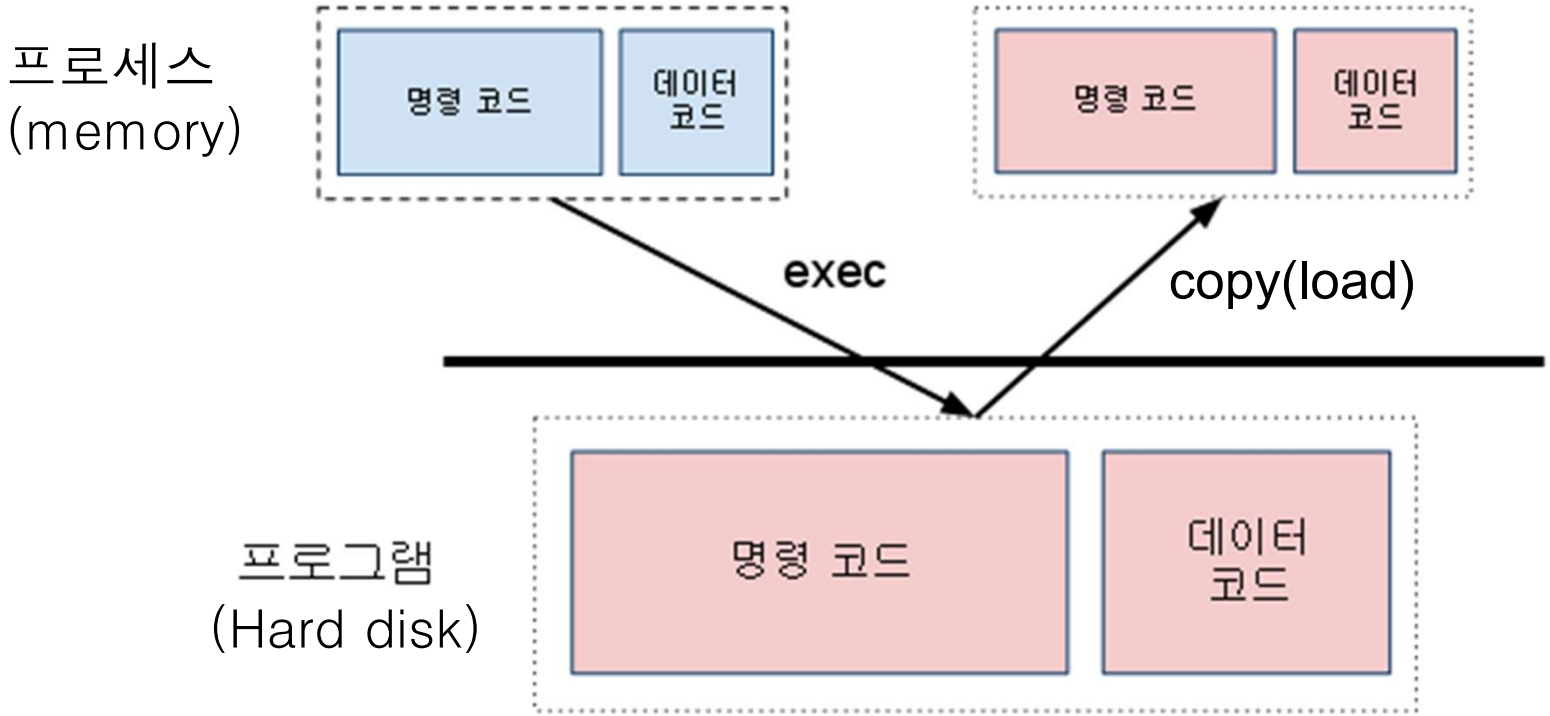
Lab

```
osnw00000000@osnw00000000-osnw: ~/week08
osnw00000000@osnw00000000-osnw:~/week08$ ./fork_test
Program Start!!
hit a key

fork !!!
I'm parent Process 88523
P : 100
fork !!!
I'm Child Process 88540
C : 100
P : 101
C : 102
P : 102
C : 104
P : 103
C : 106
P : 104
C : 108
P : 105
C : 110
P : 106
C : 112
P : 107
C : 114
P : 108
C : 116
```

```
osnw00000000@osnw00000000-osnw: ~/week08
osnw00000000@osnw00000000-osnw:~/week08$ ps -ef | grep fork_test
osnw000+   88523   88320  0 01:30 pts/0    00:00:00 ./fork_test
osnw000+   88539   88377  9 01:30 pts/1    00:00:00 grep --color=auto fork_test
osnw00000000@osnw00000000-osnw:~/week08$ ps -ef | grep fork_test
osnw000+   88523   88320  0 01:30 pts/0    00:00:00 ./fork_test
osnw000+   88540   88523  0 01:31 pts/0    00:00:00 ./fork_test
osnw000+   88542   88377  0 01:31 pts/1    00:00:00 grep --color=auto fork_test
osnw00000000@osnw00000000-osnw:~/week08$
```

Linux exec()를 이용한 프로세스 생성

- fork : 프로세스를 생성하나, 새로운 프로세스가 아닌 자기 프로세스를 복사함
- exec관련 계열 함수
  - 프로세스를 생성하지는 않고, 원래 프로세스 이미지를 새로운 프로세스 이미지로 덮어씀



Lab. command.c, exec_test.c

Lab

```c
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
        printf("-- START -- %d\n", getpid());
        printf("hit a key "); getchar(); printf("\n");
        execl("./exec test","exec test", NULL);
        printf("-- END - NOT EXECUTE --\n");
        return 1;
}
```

command.c

```c
#include <stdio.h>

int main(int argc, char **argv)
{
        getc(stdin);
        return 0;
}
```

exec_test.c

edina@hpubuntu: ~/osnw2020/lab04

```
(base) edina@hpubuntu:~/osnw2020/lab04$ ./command
-- START -- 6142
hit a key
```

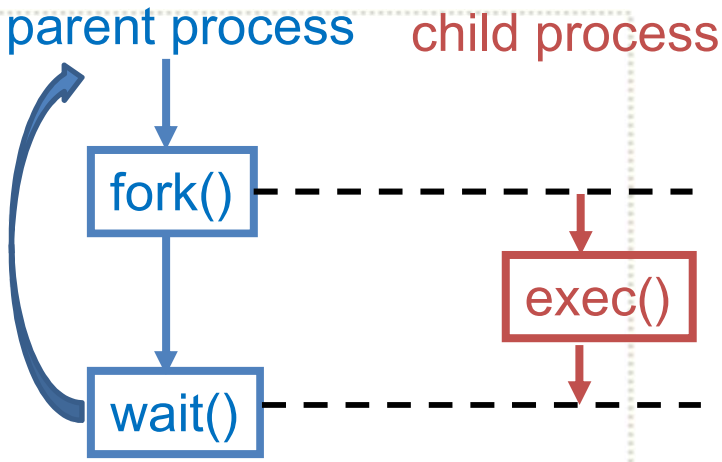
edina@hpubuntu: ~/osnw2020/lab04

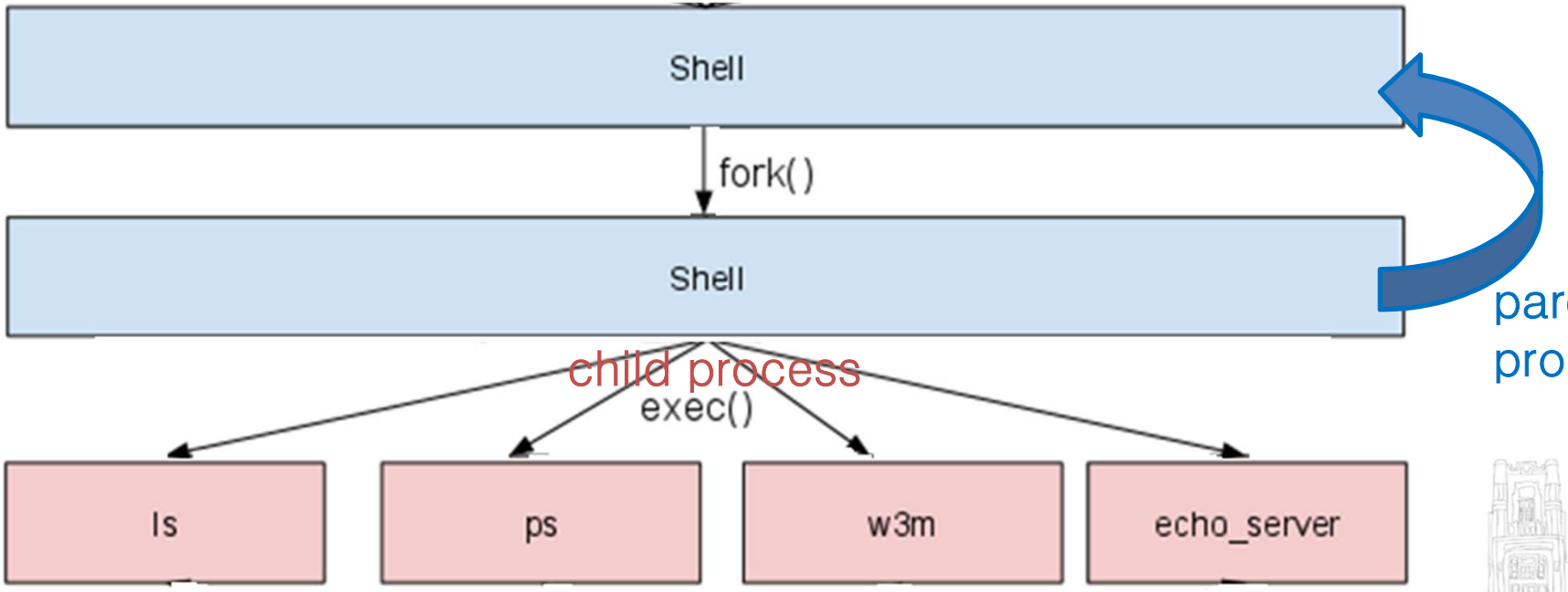
```
(base) edina@hpubuntu:~/osnw2020/lab04$ ps -ef | grep command
edina     6142  5185  0 16:13 pts/3    00:00:00 ./command
edina     6144  5282  0 16:13 pts/4    00:00:00 grep --color=auto command
(base) edina@hpubuntu:~/osnw2020/lab04$ ps -ef | grep command
edina     6146  5282  0 16:13 pts/4    00:00:00 grep --color=auto command
(base) edina@hpubuntu:~/osnw2020/lab04$ ps -ef | grep exec test
edina     6142  5185  0 16:13 pts/3    00:00:00 exec test
edina     6152  5282  0 16:13 pts/4    00:00:00 grep --color=auto exec_test
(base) edina@hpubuntu:~/osnw2020/lab04$
```

fork & exec를 이용한 새로운 프로그램 실행

Lab

- fork 함수로 프로세스 복사
- exec 함수로 프로그램 실행
- ex) shell 프로그램
  - fork로 shell의 자식 프로세스 생성
  - 자식 프로세스에서 **exec()**로 명령을 실행





Lab. myshell.c

```c
#define MAX_LINE 256
#define PROMPT   "> "
#define chop(str) str[strlen(str) -1] = 0x00;

int main(int argc, char **argv)
{
        char buf[MAX_LINE];
        int proc_status;
        pid_t pid;
        printf("My Shell Ver 1.0\n");
        while(1)
        {
                printf("%s", PROMPT);
                memset(buf, 0x00, MAX_LINE);
                fgets(buf, MAX_LINE - 1, stdin);
                if (strncmp(buf, "quit\n", 5) == 0)
                {
                        break;
                }
                chop(buf);
                pid = fork();
                if(pid == 0)
                {
                        if(execl(buf, buf, NULL) == -1)
                        {
                                printf("Execl failure\n");
                                exit(0);
                        }
                }
                if (pid > 0)
                {
                        printf("Child wait\n");
                        wait(&proc_status);
                        printf("Child exit\n");
                }
        }
        return 0;
}
```

Lab. myshell.c

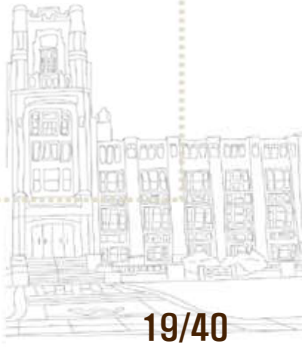
Lab

osnw00000000@osnw00000000-osnw: ~/week08

```
osnw00000000@osnw00000000-osnw:~/week08$ ./myshell
My Shell Ver 1.0
> /bin/ls
Child wait
cal_linux_cli        child_wait    daemonOSNW.c        exec_test    myshell.c
cal_linux_cli.c      child_wait.c  echo_client         exec_test.c
cal_linux_debug      command       echo_client.c       fork_test
cal_linux_server     command.c     echo_server_fork    fork_test.c
cal_linux_server.c   daemonOSNW    echo_server_fork.c  myshell
Child exit
> /bin/date
Child wait
Wed Oct 25 01:49:34 UTC 2023
Child exit
> quit
osnw00000000@osnw00000000-osnw:~/week08$
```

init 프로세스

- 프로세스는 부모 → 자식의 계층적 구조를 가짐
- 모든 프로세스의 부모 프로세스 : **PID 1** 프로세스
- 모든 프로세스는 이 프로세스에서 fork&exec로 생성
- 고아 프로세스를 관리



고아 프로세스

- 프로세스의 세계에서는 자식 프로세스가 먼저 종료되는 것을 정상으로 간주
  - 일반적으로 어떤 프로세스가 죽게 되면, 죽기 이전에 자신이 생성시킨 모든 자식 프로세스를 종료시킴

- 고아프로세스
  - 부모 프로세스가 먼저 종료된 프로세스
  - 고아 프로세스는 **init** 프로세스에 의해서 관리됨
  - 고의적 고아 프로세스 예 : 데몬 프로세스

데몬(daemon) 프로세스

- 데몬(daemon) 프로세스
  - 사용자가 인식하지 못하게 백그라운드로 동작하는 프로세스
  - 사용자 및 다른 프로세스들로부터의 영향을 받지 않아야 하는 프로세스
  - 웹서버, FTP서버, DNS 서버 등과 같은 **서버프로그램들**이 데몬 프로세스로 동작함(윈도우 서비스 프로세스와 유사)

  - 데몬 프로세스 조건
    1. 고아 프로세스여야 함, 즉 부모 프로세스를 종료시킴
    2. 표준 입력, 표준 출력, 표준 에러를 닫는다
    3. 제어 터미널(controlling terminal)을 가지지 않는다
       - 사용자가 터미널 escape 문자(대표적으로 ctrl-c) 프로세스 종료 및 suspend 가능함
       - 원격 터미널 연결이 끊어지는 경우, 원치 않게 프로세스가 종료 될수 있음

데몬(daemon) 프로세스 작성 예제

```
int main()
{
    pid_t pid;

    if (( pid = fork()) < 0)    exit(0);

    // 부모프로세스를 종료한다.
    else if(pid != 0)  exit(0);

    // 데몬 프로그램은 상호대화할 일이 없으므로 표준입/출/에러를 닫는다.
    close(0);    close(1);    close(2);

    // 세션을 생성한다.
    setsid();
    /* 사용자환경에서 독립된 자신의 환경을 만든다. 기존의 환경이 리셋되면서 터미널이 사라진다.
       또한 새로운 터미널을 지정하지 않았기 때문에, 이 프로세스는 결과적으로 터미널을 가지지 않게 된다. */

    // 데몬 프로그램이 실행할 코드를 작성한다.
    while(1)
    {
     }
}
```

Lab. daemonOSNW.c

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
  int pid;
  int i = 0;
  printf("process start %d\n", getpid());
  pid = fork();
  if (pid > 0)
  {
    printf(" parent process id(%d)\n",
     getpid());
    exit(0);
  }
else if (pid == 0)
  {
    sleep(1);
    printf("child process pid(%d) : ppid(%d)\n",
     getpid(), getppid());
    close(0);    close(1);    close(2);
    setsid();
    printf("I'm daemon\n");
    i = 1000;
    while(1)
    {
      printf("child : %d\n", i);
      i++;
      sleep(2);
    }
  }
  return 1;
}
```

데몬 프로그램이 실행할 코드

Lab. daemonOSNW.c

Lab

osnw00000000@osnw00000000-osnw: ~/week08

```
osnw00000000@osnw00000000-osnw:~/week08$ ./daemonOSNW
process start  88607
parent process pid(88607
child process pid 88608) of parent pid(88607)
osnw00000000@osnw00000000-osnw:~/week08$
```

osnw00000000@osnw00000000-osnw: ~/week08

```
osnw00000000@osnw00000000-osnw:~/week08$ ps -ef | grep OSNW
osnw000+   88607   88320  2 01:57 pts/0    00:00:00 ./daemonOSNW
osnw000+   88608   88607  0 01:57 ?        00:00:00 ./daemonOSNW
osnw000+   88610   88377  0 01:57 pts/1    00:00:00 grep --color=auto OSNW
osnw00000000@osnw00000000-osnw:~/week08$ ps -ef | grep OSNW
osnw000+   88608       1  0 01:57 ?        00:00:00 ./daemonOSNW
osnw000+   88612   88377  0 01:57 pts/1    00:00:00 grep --color=auto OSNW
osnw00000000@osnw00000000-osnw:~/week08$
```

프로세스 기다리기

- 자식 프로세스의 종료 시점과 부모 프로세스를 동기화
- 자식 프로세스의 종료 상태를 알 수 있다.

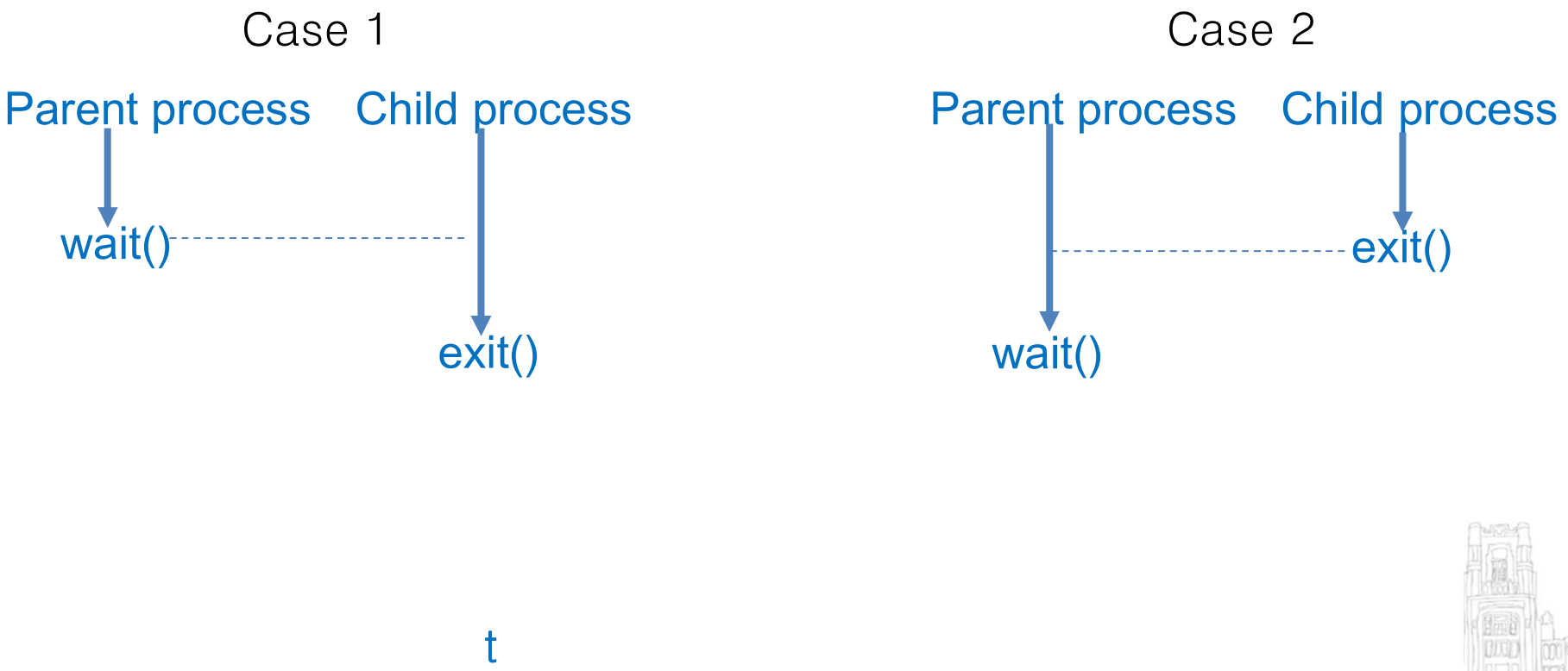
pid_t wait(int *status);

suspends the calling process
until one of its immediate children terminates or
until a child that is being traced stops because it has received a signal

if the child process terminated due to an exit() call,
the low order 8 bits of status will be 0 and
the high order 8 bits of the argument that the child process passed to exit

프로세스 기다리기

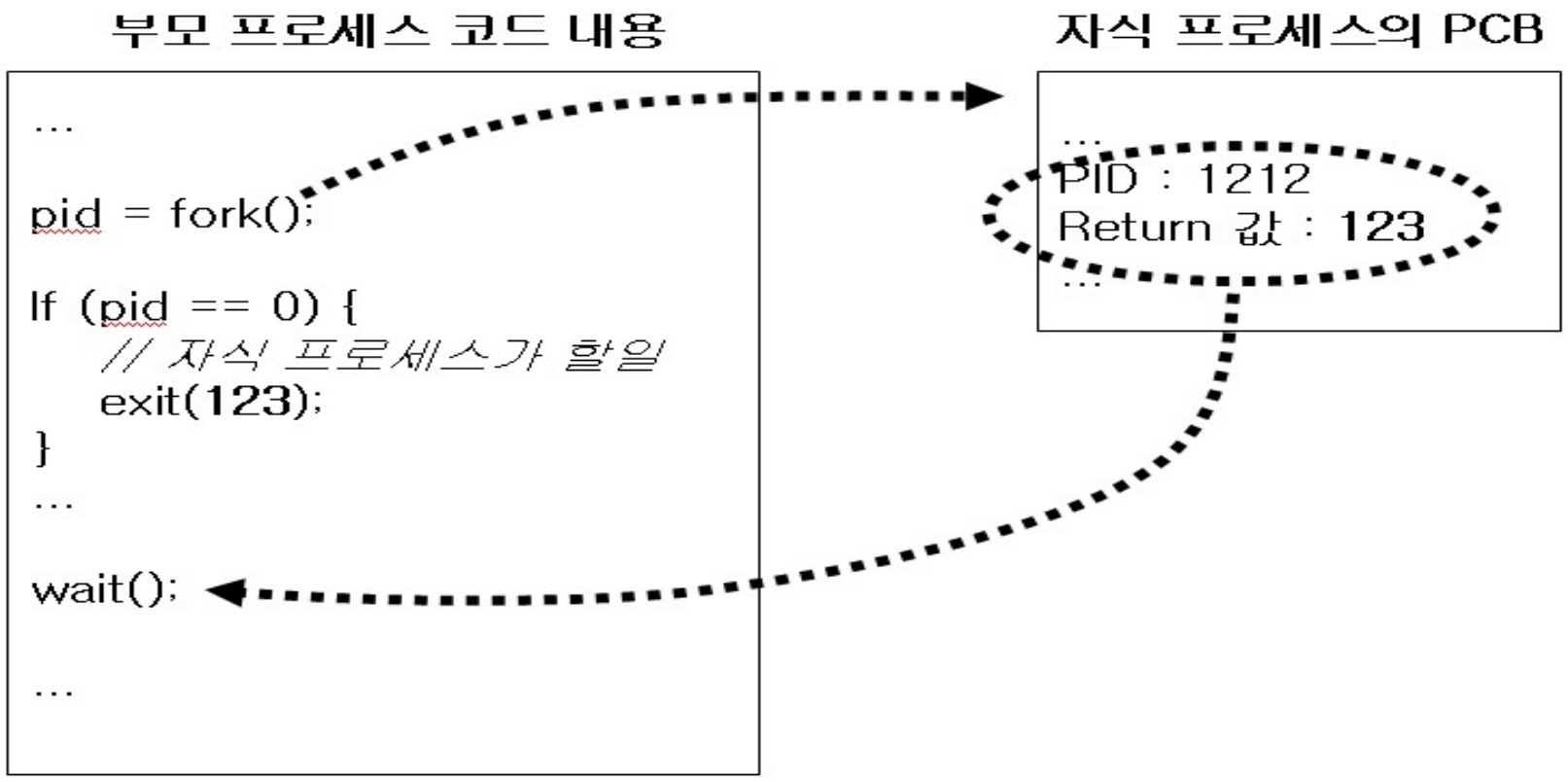
- 자식 프로세스가 종료하면, 종료 상태 값을 가지고 대기 한다.
  - exit를 호출 해도, 프로세스가 메모리 상에서 삭제되지 않는다.
- 부모 프로세스가 wait 함수를 호출해야. 이로서 모든 자원이 해제된다.
  - wait로 정리되기 전까지 프로세스는 좀비(Zombie)상태로 남는다.



프로세스 기다리기

- 좀비(Zombie) 프로세스
  - exit()실행으로 종료되었으나, OS에 의하여 관리되고 있는 상태의 프로세스
  - 종료된 자식 프로세스의 상태(정보)는 wait()함수를 통하여 얻어옴

  - 좀비 프로세스를 이용한 종료된 자식 프로세스의 정보 얻기



프로세스 기다리기

```
  pid = fork();

  if (pid > 0) {
        printf("부모 프로세스 pid(%d)\n", getpid());
        printf("자식 프로세스 종료를 기다림\n");
        pid = wait(&pstatus);
        printf("============================\n");
        printf("종료된 자식 프로세스 : %d\n", pid);
        printf("종료 값              : %d\n", pstatus/256);
} else if (pid == 0) {
        sleep(2);
        printf("I'm Zombie %d\n", getpid());
        exit(100);
}
```

Lab. child_wait.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    int pid;
    int pstatus;

    printf("process start %d\n", getpid());
    pid = fork();

    if (pid > 0) {
        printf("parent process pid(%d)\n", getpid());
        printf("wait for child(%d) to die\n", pid);
        printf("hit a key"); getchar(); printf("\n");
        pid = wait(&pstatus);
        printf("============================\n");
        printf("terminated child process : %d\n", pid);
        printf("return value from child  : %d\n",
                                            pstatus/256);
    } else if (pid == 0) {
        sleep(2);
        printf("I'm Zombie %d\n", getpid());
        exit(100);
    }

    return 1;
}
```

Lab. child_wait.c

Lab

osnw00000000@osnw00000000-osnw: ~/week08

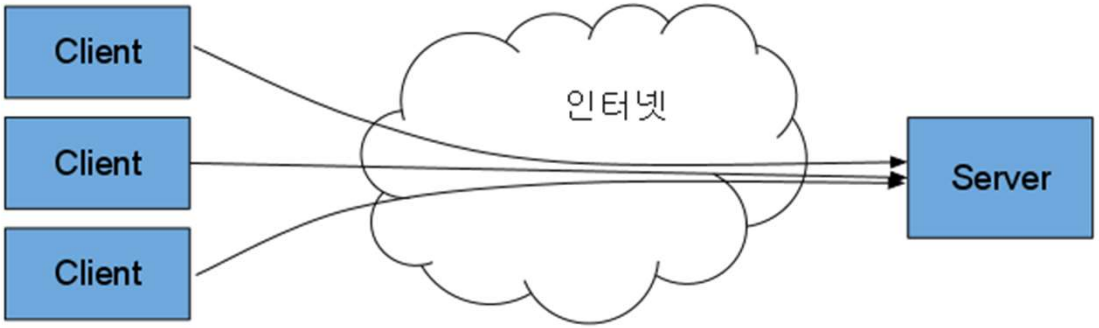
```
osnw00000000@osnw00000000-osnw:~/week08$ ./child_wait
process start 88626
parent process pid(88626)
wait for child(88627) to die
hit a keyI'm Zombie 88627


========================
terminated child process : 88627
return value from child  : 100
osnw00000000@osnw00000000-osnw:~/week08$
```

osnw00000000@osnw00000000-osnw: ~/week08

```
osnw00000000@osnw00000000-osnw:~/week08$ ps -ef | grep child_wait
osnw000+   88626   88320  0 02:03 pts/0    00:00:00 ./child_wait
osnw000+   88627   88626  0 02:03 pts/0    00:00:00 [child_wait] <defunct>
osnw000+   88630   88377  0 02:04 pts/1    00:00:00 grep --color=auto child_wait
osnw00000000@osnw00000000-osnw:~/week08$
osnw00000000@osnw00000000-osnw:~/week08$ ps -ef | grep child_wait
osnw000+   88632   88377  0 02:04 pts/1    00:00:00 grep --color=auto child_wait
osnw00000000@osnw00000000-osnw:~/week08$ ps -ef | grep 88627
osnw000+   88634   88377  0 02:05 pts/1    00:00:00 grep --color=auto 88627
osnw00000000@osnw00000000-osnw:~/week08$
```

멀티 프로세스와 소켓 프로그래밍

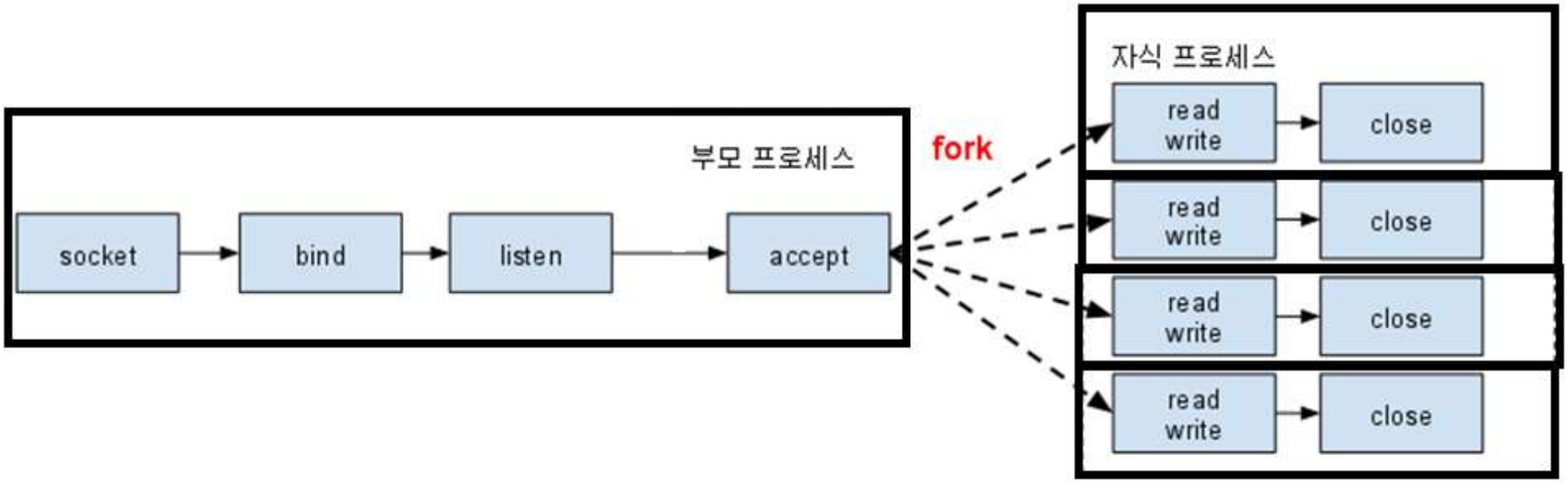
- 서버 프로그램은 다수의 클라이언트를 동시에 처리할 수 있어야 한다.



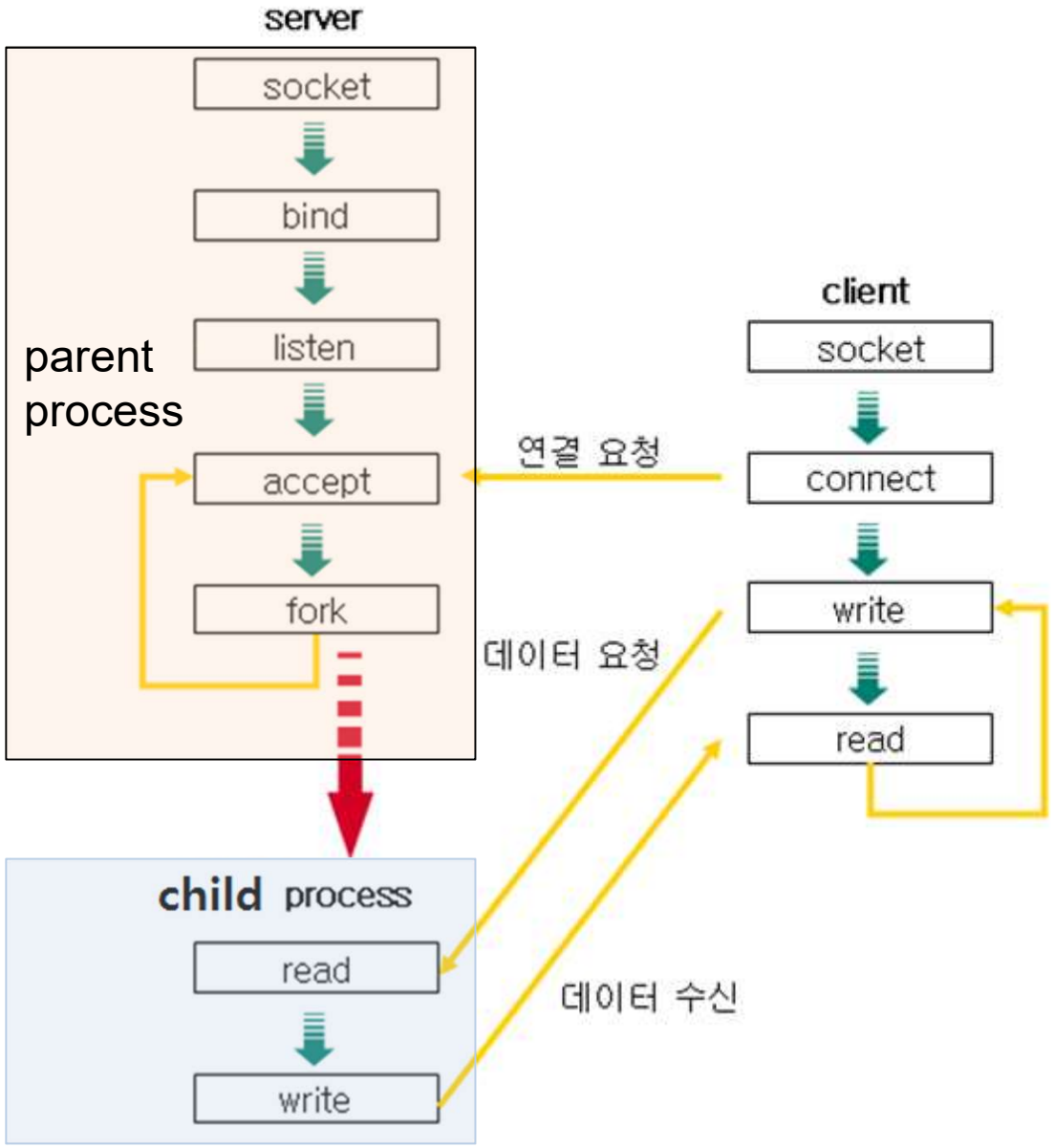
- 대표적인 다중 클라이언트 처리 기술(다중 접속처리 서버 기술)
  - 멀티 프로세스**(Multi-process)**
  - 멀티 쓰레드**(Multi-thread)**
  - 입출력다중화**(I/O Multiplexing)**

멀티 프로세스와 소켓 프로그래밍

- fork 함수를 이용해서 클라이언트와 통신하는 코드를 분리
- **accept() 호출 후 fork() 호출**



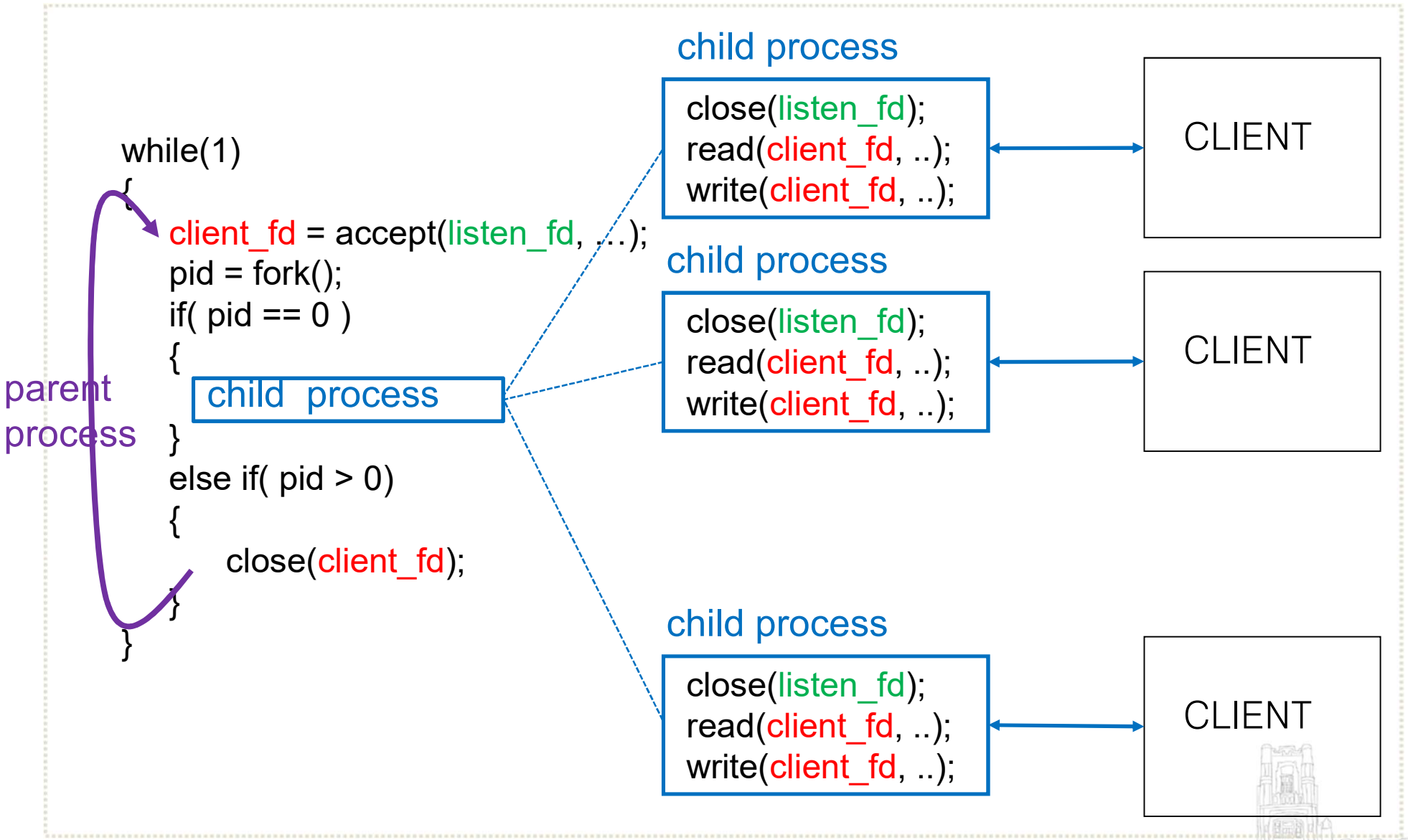
멀티 프로세스와 소켓 프로그래밍



fork() 수행후 소켓 지정 번호 복사

- fork() 수행하여 자식 프로세스 만들 때, 부모 프로세스의 **자원**이 복사 된다.
  - 코드, 데이터
  - 소켓을 포함한 모든 열린 파일들(즉 파일 지정번호)
  - 시그널
- accept() 에서 "듣기(서버) 소켓"과 "연결(클라이언트) 소켓" 분리
- fork() 수행 후 “듣기 소켓”과 “연결 소켓” 자원이 자식 프로세스에 복사
- 부모 프로세스는 “듣기 소켓”으로 accept 호출
  - “연결소켓”은 사용하지 않으므로 닫음
- 자식 프로세스는 “연결 소켓”으로 클라이언트와 통신
  - “듣기소켓”은 사용하지 않으므로 닫음

멀티 프로세스와 소켓 프로그래밍



Lab. echo_server_fork.c

Lab

edina@hpubuntu: ~/osnw2020/lab04

```
(base) edina@hpubuntu:~/osnw2020/lab04$ ./echo_server_fork
Read Data 127.0.0.1(53405) : osnw2020
Read Data 127.0.0.1(53917) : 32181234
Read Data 127.0.0.1(54429) : hong
```

edina@hpubuntu: ~/osnw2020/lab04

```
(base) edina@hpubuntu:~/osnw2020/lab04$ ./echo_client
osnw2020
read : osnw2020
(base) edina@hpubuntu:~/osnw2020/lab04$
```

edina@hpubuntu: ~/osnw2020/lab04

```
(base) edina@hpubuntu:~/osnw2020/lab04$ ./echo_client
32181234
read : 32181234
(base) edina@hpubuntu:~/osnw2020/lab04$
```

edina@hpubuntu: ~/osnw2020/lab04

```
(base) edina@hpubuntu:~/osnw2020/lab04$ ./echo_client
hong
read : hong
(base) edina@hpubuntu:~/osnw2020/lab04$
```

멀티 프로세스 기술의 장점

1. 단순한 프로그램 흐름
    - accept 후 fork 함수 호출
2. 오랜 시간 검증된 기술
    - 유닉스는 멀티 프로세스 기반으로 시작.
    - 멀티 스레드 기술은 비교적 최근에 도입
3. 안정적인 동작
    - 독립된 프로세스로 작동
    - 프로세스의 잘못된 작동이 다른 프로세스에 영향을 미치지 않음

멀티 프로세스 기술의 단점

1. 프로세스 복사에 따른 성능 문제
    - 프로세스가 새로 생성되기 때문에 많은 CPU/Memory 비용이 소모
    - 코드가 중복
    - 연결과 종료가 빈번한 서비스에서 연결 지연이 생길 수 있음.
2. 프로세스간 정보 교환이 어렵다.
    - 독립된 프로세스로 작동.
    - 프로세스간 통신에 **IPC**를 이용해야 함.

Thank You !

뇌를 자극하는 TCP/IP 소켓 프로그래밍