지금까지 1대1

이제는 1대다

다중 접속 서버

Chapter 10

멀티프로세스 기반의 서버 구현

다중 접속 서버의 구현 방법들

다중 접속 서버의 정의

- 다중 접속 서버
  - ✓ 둘 이상의 클라이언트에게 **동시에 접속을 허용**
  - ✓ 둘 이상의 클라이언트에게 **동시에 서비스를 제공**하는 서버를 의미함

다중 접속 서버 구현 방법

- 멀티프로세스 기반 서버: 다수의 프로세스를 생성하는 방식
- 멀티플렉싱 기반 서버: 입출력 대상을 묶어서 관리하는 방식
- 멀티쓰레딩 기반 서버: 클라이언트의 수만큼 쓰레드를 생성하는 방식

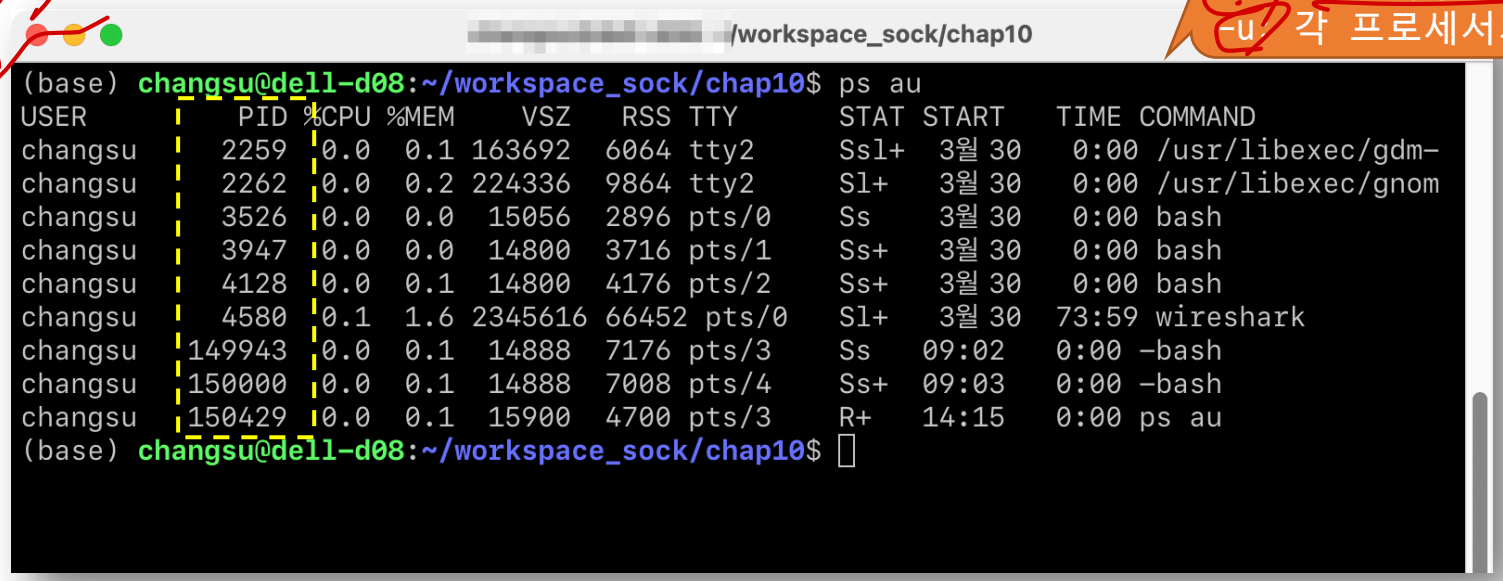
프로세스와 프로세스 ID

- 프로세스란?
  - 간단하게는 실행 중인 프로그램을 의미
  - 실행 중인 프로그램에 관련된 메모리, 리소드 등을 총칭하는 의미
  - 멀티프로세스 운영체제는 둘 이상의 프로세스를 동시에 생성 가능

- 프로세스 ID (PID)
  - 운영체제는 생성되는 모든 프로세스에 ID를 할당함

$ps

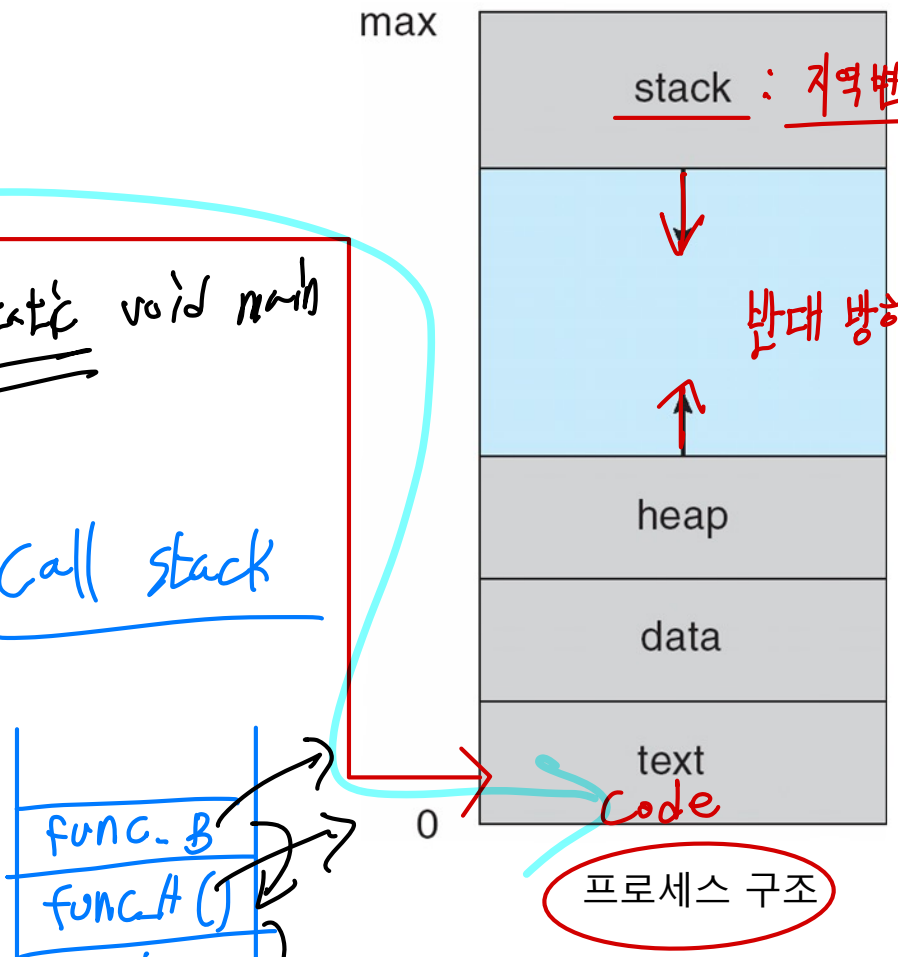
ps: process status
-a: 다른 사용자들의 프로세스 보여줌
-u: 각 프로세서의 사용자 이름과 시작 시간



프로세스 개념

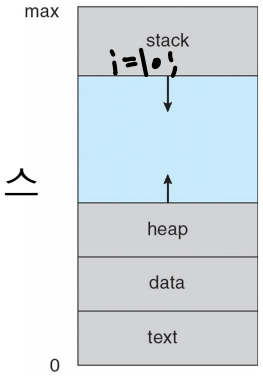
- 프로세스
  - 실행 중인 프로그램

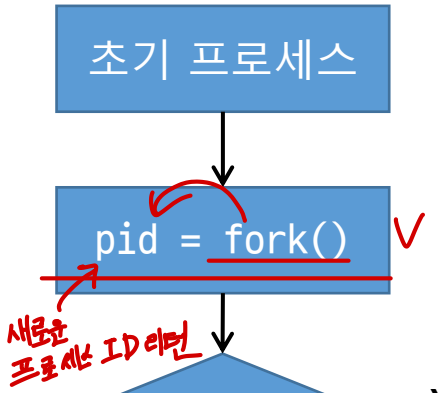
- 프로세스 구성 요소
  - **Text section (Code section)**
    - 프로그램 소스 코드
  - **Data section**
    - 전역 변수나 정적 변수를 저장
  - **Heap section**
    - 동적 메모리 할당 영역
    - malloc()
  - **Stack section**
    - 임시 데이터 저장
    - 지역 변수들, 함수 파라미터, 리턴 주소



프로세스 생성: 복사본 생성 fork()

초기 프로세스

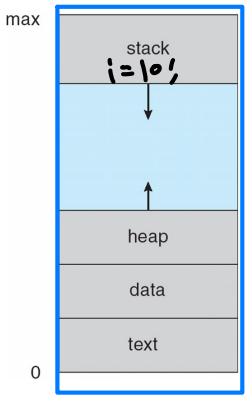





```
include <unistd.h>

pid_t fork(void);
-  실패: -1 리턴
-  성공: 새로운 PID를 리턴
-  이미 실행중인 프로세스를 복사함
-  메모리 영역까지 동일하게 복사(코드 포함)
```

- pid=0을 리턴(자식프로세스)
- 초기 프로세스의 메모리 영역을 복사
- 별도의 메모리 공간을 가짐

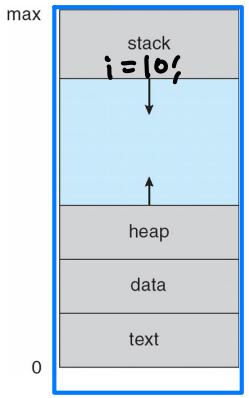
fork()함수 이후부터
독립적으로 실행



부모 프로세스

fork()함수를 호출한 프로세스

새
프로세스



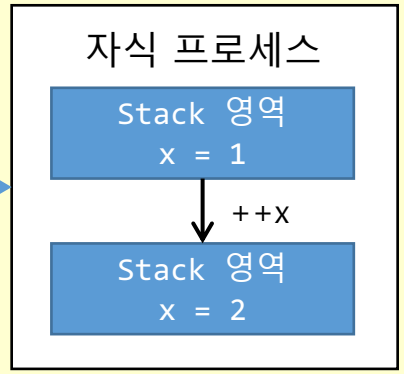
자식 프로세스

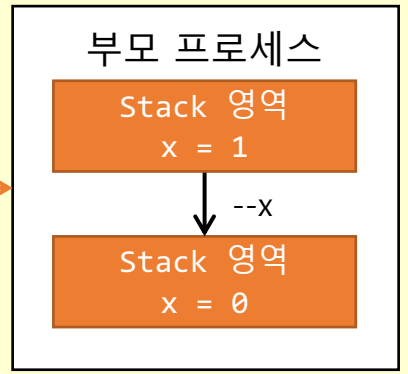
fork()함수 호출을 통해 생성(복제)된 프로세스

fork 예제 (forktest1.c)

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

void forkexample()
{
    int x = 1;
    pid_t pid;

    pid = fork();
    if(pid == 0)
        printf("Child has x= %d\n", ++x);
    else
        printf("Parent has x= %d, Child pid= %d\n", --x, pid);
}

int main()
{
    forkexample();
    return 0;
}
```

fork() 함수 호출
- 메모리 영역까지 동일하게 복사된 프로세스가 생성
- Text section (코드영역)도 동일하게 복사



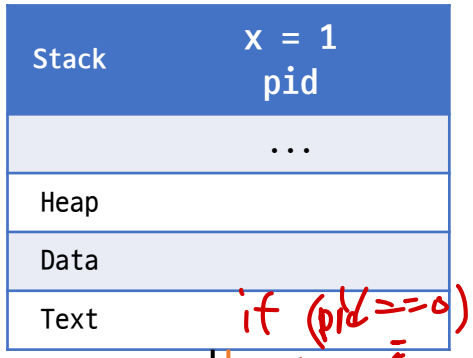



```
Parent has x = 0, Child pid=150472
Child has x = 2
```

fork() 함수 호출 이후,
부모 프로세스와 자식 프로세스는
독립된 메모리 공간을 가짐

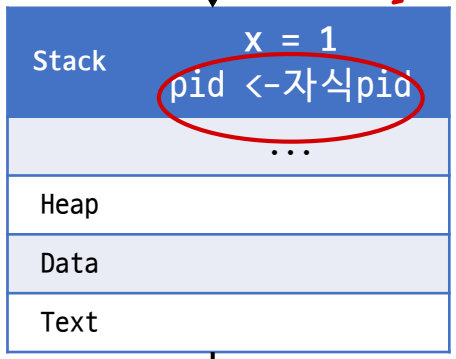
프로세스 복제 과정

초기 프로세스

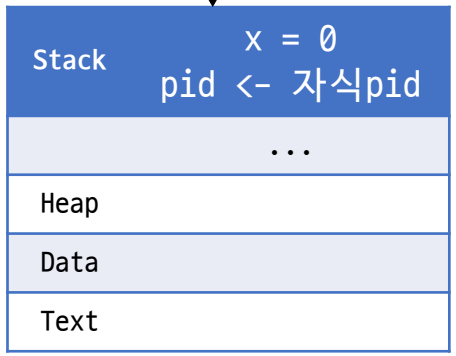



```
int x = 1;
pid_t pid;

pid = fork();
if(pid == 0)
    printf("Child has x= %d\n", ++x);
else
    printf("Parent has x= %d, Child pid= %d\n", --x, pid);
```

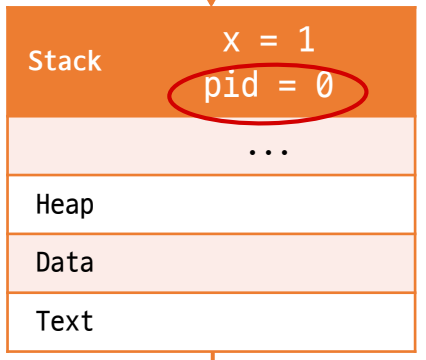
fork()호출



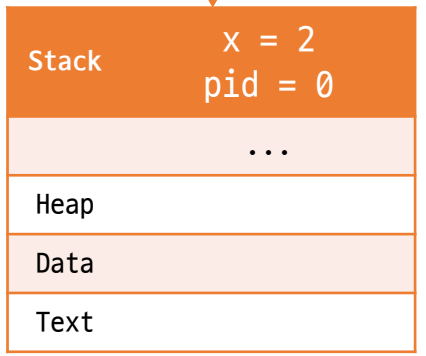
--x



프로세스 복제



++x



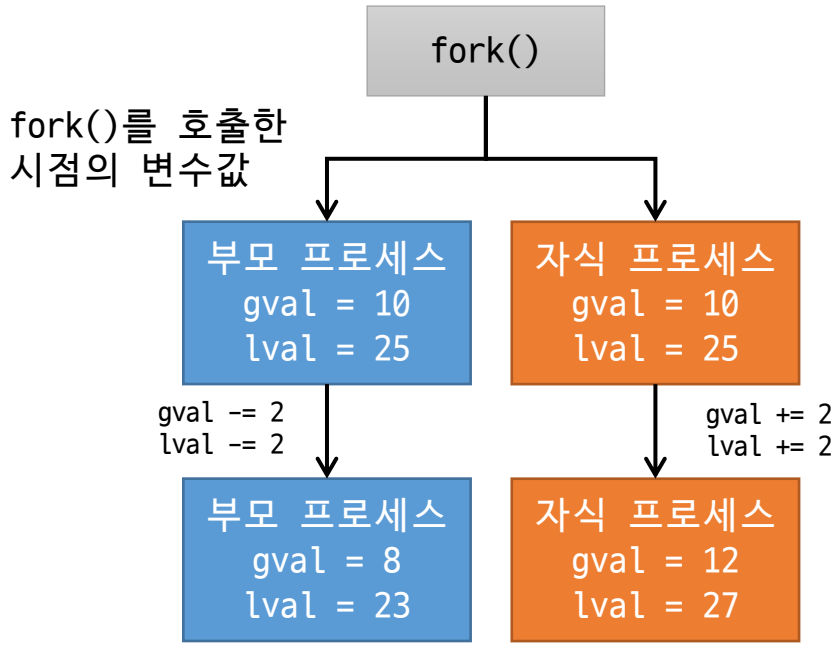
자식 프로세스

부모 프로세스

- fork()함수를 호출한 프로세스

fork.c

```c
#include <stdio.h>
#include <unistd.h>
int gval=10;

int main(int argc, char *argv[])
{
    pid_t pid;
    int lval=20;
    lval+=5;

    pid = fork();
    if(pid==0) // if Child Process
        gval += 2, lval += 2;
    else       // if Parent Process
        gval -= 2, lval -= 2;

    if(pid==0)
        printf("Child Proc: [%d, %d] \n", gval, lval);
    else
        printf("Parent Proc: [%d, %d] \n", gval, lval);

    return 0;
}
```



부모, 자식 프로세스의
전역 변수, 지역 변수 모두
각각 다른 값을 가짐

```
Parent Proc: [8, 23]
Child Proc: [12, 27]
```

fork.c 실행 과정

```c
#include <stdio.h>
#include <unistd.h>
int gval=10;

int main(int argc, char *argv[])
{
    pid_t pid;
    int lval=20;
    lval+=5;

    pid=fork();
```

pid = 자식프로세스

부모 프로세스

```c
  if(pid==0) // if Child Process
    gval+=2, lval+=2;
  else           // if Parent Process
    gval-=2, lval-=2;

  if(pid==0)
    printf("Child Proc: [%d, %d]\n", gval, lval);
  else
    printf("Parent Proc: [%d, %d]\n", gval, lval);
  return 0;
}
```

pid = 0

자식 프로세스

```c
  if(pid==0) // if Child Process
    gval+=2, lval+=2;
  else           // if Parent Process
    gval-=2, lval-=2;

  if(pid==0)
    printf("Child Proc: [%d, %d] \n", gval, lval);
  else
    printf("Parent Proc: [%d, %d] \n", gval, lval);
  return 0;
}
```

좀비 프로세스의 이해

- 좀비 프로세스란?
  - 실행이 완료되었음에도 소멸되지 않은 프로세스
  - 프로세스도 main 함수가 반환되면 소멸되어야 함
  - 소멸되지 않았다는 것은 프로세스가 사용한 리소스가 메모리 공간에 여전히 존재함을 의미

return 0

- 자식 프로세스가 종료되는 상황 2가지

```
• 인자를 전달하면서 exit()를 호출하는 경우: exit(1)
• main 함수에서 return 문을 실행하면서 값을 반환하는 경우: return 0
```

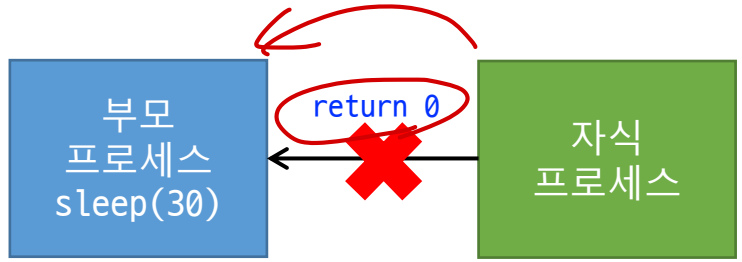
- 좀비 프로세스의 생성 원인
  - 자식 프로세스가 종료되면서 반환하는 상태 값이 부모 프로세스에게 전달되지 않으면
    - 해당 프로세스는 소멸되지 않고 좀비가 됨

zombie.c

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

int main(int argc, char *argv[])
{
    pid_t pid=fork();
    if(pid==0) // if Child Process
    {
        puts("Hi I am a child process");
    }
    else
    {
        printf("Child Process ID: %d \n", pid);
        sleep(30); // Sleep 30 sec.
    }

    if(pid==0)
        puts("End child process");
    else
        puts("End parent process");

    return 0;
}
```

부모 프로세스의 종료를 일부러 늦춤

자식 프로세스의 종료(리턴) 값을 받을 부모 프로세스가 소멸되면, 좀비 상태의 자식 프로세스도 소멸됨
- 부모 프로세스가 소멸되기 전에 좀비 상태 확인




```
$ ./zombie
Child Process ID: 150526
Hi I am a child process
End child process
End parent process
```

Zombie 프로세스 확인

- 부모 프로세스가 종료되지 않은 시점
  - 자식 프로세스(pid: 150526)가 zombie가 됨

```
$ ps au
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
changsu     2259  0.0  0.1 163692  6064 tty2     Ssl+  3월30   0:00 /usr/libexec/gdm-way
changsu     4128  0.0  0.1  14800  4176 pts/2    Ss+   3월30   0:00 bash
changsu   150000  0.0  0.1  14888  7008 pts/4    Ss    09:03   0:00 -bash
changsu   150525  0.0  0.0   2772  1024 pts/3    S+    15:36   0:00 ./zombie
changsu   150526  0.0  0.0      0     0 pts/3    Z+    15:36   0:00 [zombie] <defunct>
changsu   150527  0.0  0.1  15900  4556 pts/4    R+    15:36   0:00 ps au
```

  - defunct 프로세스: 실행은 완료했지만, 부모 프로세스에게 완료 상태를 전달하지 못한 프로세스

- 부모 프로세스가 종료된 시점: [zombie] <defunct> 없어짐

```
$ ps au
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
changsu     2259  0.0  0.1 163692  6064 tty2     Ssl+  3월30   0:00 /usr/libexec/gdm-way
changsu     4128  0.0  0.1  14800  4176 pts/2    Ss+   3월30   0:00 bash
changsu   150000  0.0  0.1  14888  7008 pts/4    Ss    09:03   0:00 -bash
changsu   150530  0.0  0.1  15900  4556 pts/4    R+    15:45   0:00 ps au
```

좀비 프로세스의 소멸 방법 #1: wait()함수

- wait() 함수
  - 자식 프로세스가 종료될 때까지 대기
  - 자식 프로세스가 종료한 상태를 가져옴

자바는 join()

  - wait() 함수 호출 시, 이미 종료된 프로세스가 있는 경우
    - 자식 프로세스가 종료되면서 전달한 값이 status 변수에 저장
    - status 변수
      - exit함수의 인자값
      - main함수의 return값 저장

exit(3);
return -1;

```
#include <sys/wait.h>

pid_t wait(int *status);
-> 성공 시 종료된 자식 프로세스의 ID, 실패 시 -1 반환
```

좀비 프로세스의 소멸 방법 #1: wait()함수

- 자식 프로세스 종료 상태 분석 매크로 함수
  - int WIFEXITED (int status)
    - 자식 프로세스의 종료 여부 확인 (wait if exited)
    - 주로 조건문에 사용

```
#include <sys/wait.h>

int WIFEXITED(int status);
-> 자식 프로세스가 정상 종료한 경우 non-zero 반환
```

  - int WEXITSTATUS (int status)
    - 자식 프로세스의 전달 값을 반환 (종료 시 리턴값 확인) (wait exit status)

```
#include <sys/wait.h>

int WEXITSTATUS(int status);
-> exit() 함수의 인자값, return 값이 반환
```

좀비 프로세스의 소멸 방법 #1: wait.c

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

int main(int argc, char *argv[])
{
    int status;
    pid_t pid = fork();

    if(pid==0)
    {
        printf("Child process #1 is terminated(3)\n");
        return 3;
    }
    else
    {
        printf("Child PID: %d \n", pid);
        pid = fork();
        if(pid==0)
        {
            printf("Child process #2 is terminated(7)\n");
            exit(7);
        }
        else
        {
            printf("Child PID: %d \n", pid);
            wait(&status);
            if(WIFEXITED(status))
                printf("Child #1 sent: %d \n", WEXITSTATUS(status));

            wait(&status);
            if(WIFEXITED(status))
                printf("Child #2 sent: %d \n", WEXITSTATUS(status));
            sleep(30); // Sleep 30 sec.
        }
    }
    return 0;
}
```

첫 번째 자식 프로세스 생성

자식 프로세스 #1 생성 및 종료

두 번째 자식 프로세스 생성

자식 프로세스 #2 생성 및 종료

부모 프로세스 실행 영역

2개의 자식 프로세스 생성 -> 2번의 wait() 함수 호출

wait 함수의 경우, 자식 프로세스가 종료되지 않은 상황에서는 반환하지 않고 **블로킹 상태**에 놓임

```
$ ./wait
Child PID: 150694
Child process #1 is terminated(3)
Child PID: 150695
Child process #2 is terminated(7)
Child #1 sent: 3
Child #2 sent: 7
```

좀비 프로세스의 소멸 방법 #2: waitpid()함수

- waitpid() 함수

```
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *statloc, int options);
-> 성공 시 종료된 자식 프로세스의 ID, 실패 시 -1 반환
```

WNOHANG

- pid: 종료를 기다릴 자식 프로세스의 ID
  - -1을 전달하면, wait() 함수와 마찬가지로 임의의 자식 프로세스가 종료되기를 기다림
- statloc: wait() 함수의 매개변수(status)와 동일한 의미
- options
  - WNOHANG: Non-blocking (Wait No Hang의 의미), 종료된 자식 프로세스가 존재하지 않아도 블록킹 상태에 있지 않음 (0을 반환하면서 waitpid()함수를 빠져 나옴)

- wait() 함수
  - ✓ 호출된 시점에 종료된 프로세스가 없는 경우, 임의의 자식 프로세스가 종료될 때까지 blocking 상태에 놓임

- waitpid() 함수
  - ✓ Blocking 상태에 놓이지 않게 할 수 있는 장점이 있음
  - ✓ WNOHANG 옵션 사용 (wait no hang)

좀비 프로세스의 소멸 방법 #2: waitpid.c

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main(int argc, char *argv[])
{
    int status;
    pid_t pid=fork();
    int cid = 0;

    if(pid==0)
    {
        sleep(15);
        return 24;
    }
    else
    {
        printf("Child pid= %d\n", pid);
        while((cid = waitpid(-1, &status, WNOHANG))==0)
        {
            sleep(3);
            puts("sleep 3sec.");
        }
        printf("exit while: cid= %d\n", cid);
        if(WIFEXITED(status))
            printf("Child sent %d \n", WEXITSTATUS(status));
    }
    return 0;
}
```

자식 프로세스의 종료를 지연 시킴

종료된 자식 프로세스가 없는 경우에 수행됨

```
$ ./waitpid
Child pid= 150719
sleep 3sec.
sleep 3sec.
sleep 3sec.
sleep 3sec.
sleep 3sec.
exit while: cid= 150719
Child sent 24
```

5회 반복

waitpid(-1, &status, WNOHANG):
- 첫 번째 인자: -1
- 임의의 프로세스가 종료 되는 것을 기다림
- 세 번째 인자(WNOHANG): 종료된 자식 프로세스가 없으면 0을 반환하고 함수를 빠져나옴