9장 프로세스 제어

9.1 프로세스 생성

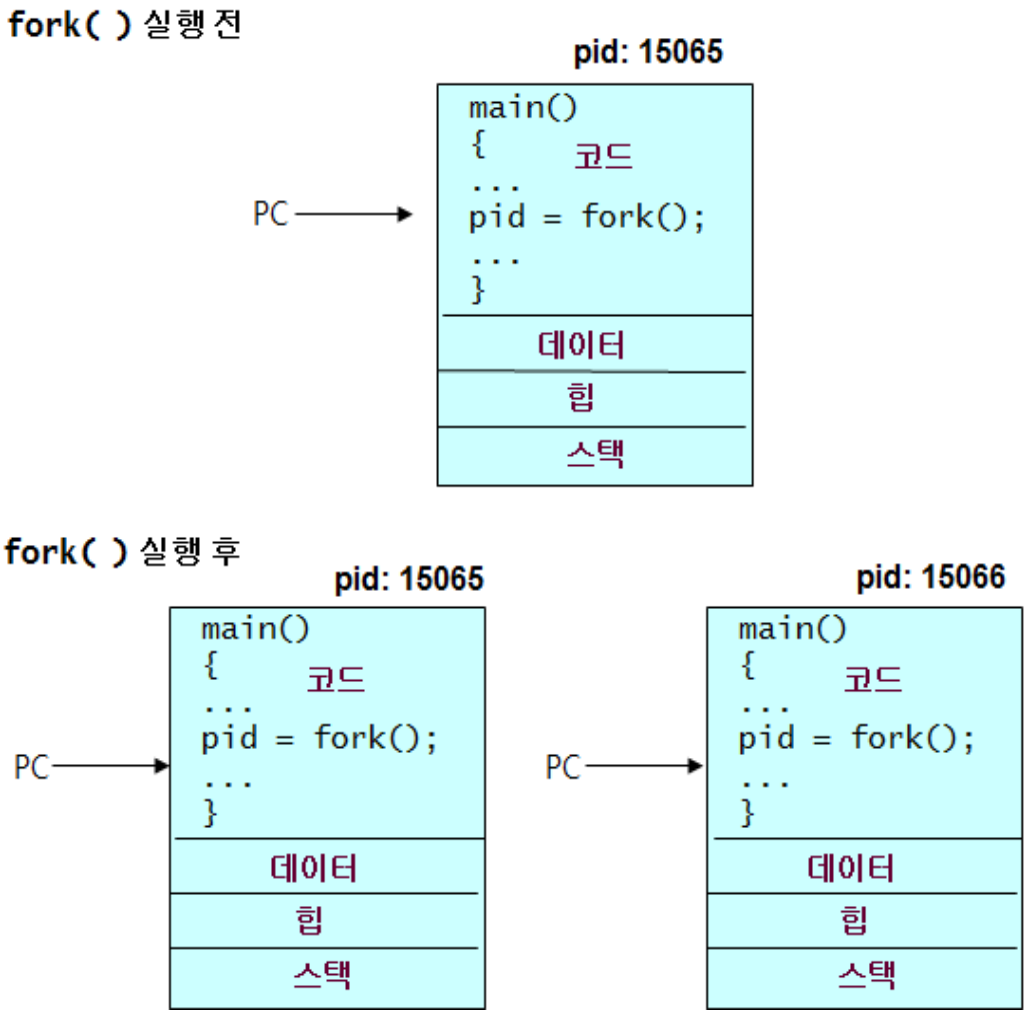
프로세스 생성

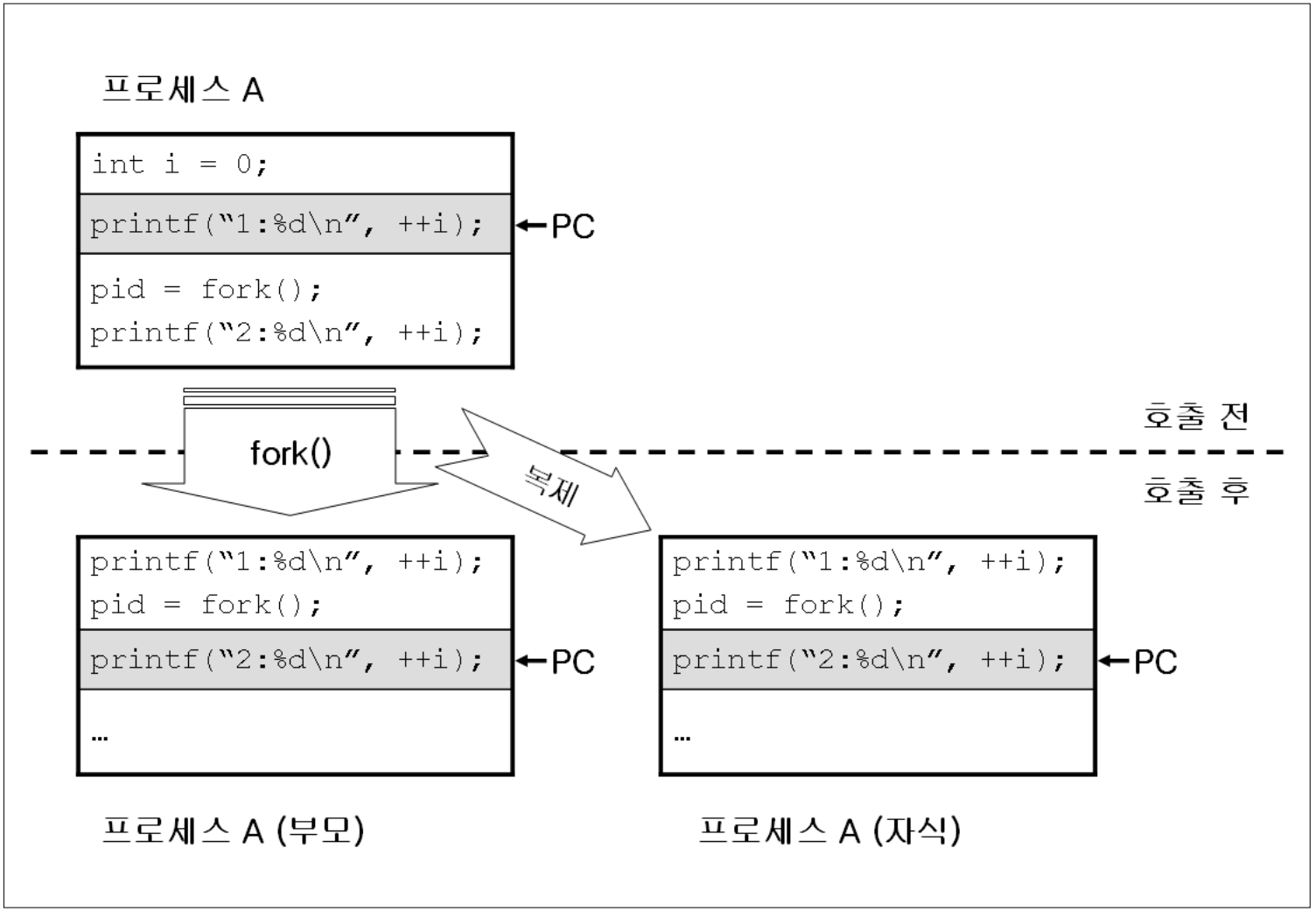
- fork() 시스템 호출
  - 부모 프로세스를 똑같이 복제하여 새로운 자식 프로세스를 생성
  - 자기복제(自己複製)

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

새로운 자식 프로세스를 생성한다. 자식 프로세스에게는 0을 리턴하고 부모 프로세스에게는 자식 프로세스 ID를 리턴한다.

프로세스 생성


fork( ) 실행 전
pid: 15065
PC
main()
{
코드
...
pid = fork();
...
}
데이터
힙
스택
fork( ) 실행 후
pid: 15065
PC
main()
{
코드
...
pid = fork();
...
}
데이터
힙
스택
pid: 15066
PC
main()
{
코드
...
pid = fork();
...
}
데이터
힙
스택


프로세스 A
int i = 0;
printf("1:%d\n", ++i);
PC
pid = fork();
printf("2:%d\n", ++i);
fork()
복제
호출 전
호출 후
printf("1:%d\n", ++i);
pid = fork();
printf("2:%d\n", ++i);
PC
…
printf("1:%d\n", ++i);
pid = fork();
printf("2:%d\n", ++i);
PC
…
프로세스 A (부모)
프로세스 A (자식)

프로세스 생성

- fork()는 한 번 호출되면 두 번 리턴한다.
  - 자식 프로세스에게는 0을 리턴하고
  - 부모 프로세스에게는 자식 프로세스 ID를 리턴한다.

  - 부모 프로세스와 자식 프로세스는 병행적으로 각각 실행을 계속한다.

프로세스 생성

- 부모(parent) 프로세스와 자식(child) 프로세스
  - fork를 호출하는 쪽을 부모 프로세스라고 하고 새로 생성된 쪽을 자식 프로세스라고 한다.
- 부모 프로세스와 자식 프로세스는 서로 다른 프로세스이다.
  - 프로세스 식별 번호 (PID)가 서로 다르다.
  - 자식 프로세스의 부모 프로세스 식별 번호 (PPID)는 자신을 생성한 부모 프로세스가 된다.
- 자식 프로세스는 부모 프로세스가 fork를 호출하던 시점의 상태를 그대로 물려받는다.
  - 프로그램 코드
  - 프로그램 변수에 저장되어 있는 데이터 값
  - 하드웨어 레지스터의 값
  - 프로그램 스택의 값 등등
- fork 호출 이후에 부모와 자식 프로세스는 자신들의 나머지 프로그램 코드를 수행한다.

fork1.c

```
#include <stdio.h>
#include <unistd.h>
/* 자식 프로세스를 생성한다. */
int main()
{
    int pid;
    printf("[%d] 프로세스 시작 ₩n", getpid());
    pid = fork();
    printf("[%d] 프로세스 : 리턴값 %d₩n", getpid(), pid);
}
```

부모 프로세스와 자식 프로세스 구분

- fork() 호출 후에 리턴값이 다르므로 이 리턴값을 이용하여
- 부모 프로세스와 자식 프로세스를 구별하고
- 서로 다른 일을 하도록 할 수 있다.

```
pid = fork();
if ( pid == 0 )
{
    자식 프로세스의 실행 코드
 }
else
{
    부모 프로세스의 실행 코드
}
```

fork2.c

```
#include <stdlib.h>
#include <stdio.h>
/* 부모 프로세스가 자식 프로세스를 생성하고 서로 다른 메시지를 프린트 */
int main()
{
    int pid;
    pid = fork();
    if (pid ==0) {    // 자식 프로세스
        printf("[Child] : Hello, world pid=%d₩n", getpid());
    }
    else {    // 부모 프로세스
        printf("[Parent] : Hello, world pid=%d₩n", getpid());
    }
}
```

fork3.c: 두 개의 자식 프로세스 생성

```
#include <stdlib.h>
#include <stdio.h>
/* 부모 프로세스가 두 개의 자식 프로세스를 생성한다. */
int main()
{
    int pid1, pid2;
    pid1 = fork();
    if (pid1 == 0) {
        printf("[Child 1] : Hello, world ! pid=%d₩n", getpid());
        exit(0);
    }
    pid2 = fork();
    if (pid2 == 0) {
        printf("[Child 2] : Hello, world ! pid=%d₩n", getpid());
        exit(0);
    }
}
```

```
01 #include <unistd.h>
02 #include <sys/types.h>
03
04 main()
05 {
06     pid_t pid;
07     int i = 0;
08
09     i++;
10     printf("before calling fork(%d)₩n", i);
11
12     pid = fork();
13
14     if(pid == 0)
15         /* 자식 프로세스가 수행할 부분 */
16         printf("child process(%d)₩n", ++i);
17     else if(pid > 0)
18         /* 부모 프로세스가 수행할 부분 */
19         printf("parent process(%d)₩n", --i);
20     else
21   /* fork 호출이 실패할 경우 수행할 부분 */
22         printf("fail to fork₩n");
23}
```

부모와 자식의 실행 순서는?

```
$ ex07-02
before calling fork(1)
parent process(0)
child process(2)
$
```

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int glob = 6; /* external variable in initialized data */
char buf[] = "a write to stdout\n";

int main(void) {
        int var; /* automatic variable on the stack */
        pid_t pid;
        var = 88;
        if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
                perror("write error");
        printf("before fork\n");
        if ( (pid = fork()) < 0)
                perror("fork error");
        else if (pid == 0) { /* child */
                glob++; /* modify variables */
                var++;
        }
        else sleep(2); /* parent */
        printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
        exit(0);
}
```

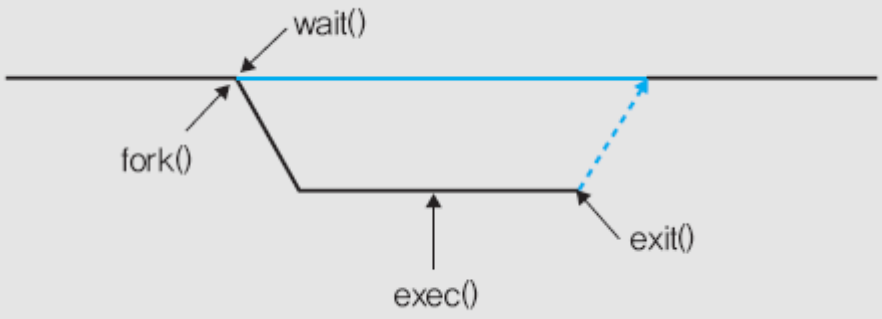
실습

- 파일을 open한 후에 fork를 실행하고, 부모와 자식이 각각 read 명령을 내리면 읽은 내용이 같은가? 즉, 부모와 자식이 offset을 공유하는가?
  - 직접 test 해볼 것!

프로세스 기다리기: wait()

- 자식 프로세스 중의 하나가 끝날 때까지 기다린다.
  - 끝난 자식 프로세스의 종료 코드가 status에 저장되며
  - 끝난 자식 프로세스의 번호를 리턴한다.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *statloc, int options);
```



wait()

자식 프로세스를 가진 부모 프로세스가 wait를 호출하면

- 자식 프로세스가 종료할 때까지 실행이 중단된다. (대기 상태)
- 자식 프로세스가 종료하면 이를 처리한다.
  - wait 호출 이전에 자식이 종료했다면 대기 상태가 되지 않고 처리

waitpid

- wait는 자식 프로세스 중 가장 먼저 종료되는 것을 처리해주나,

  waitpid는 PID로 지정한 자식 프로세스의 종료만 처리해준다.
- WNOHANG 옵션을 사용할 때 종료한 자식 프로세스가 없으면 0을 반환한다. 호출이 실패할 경우 -1을 반환한다. 일반적으로는 0을 사용한다.

wait와 waitpid의 차이점

- wait
  - 부모 프로세스가 특정 자식 프로세스를 기다리지 않는다.

    먼저 종료되는 것을 먼저 처리해준다.
  - 종료되는 자식 프로세스를 wait의 반환 값으로 알 수 있다.
  - 종료하는 자식 프로세스가 있을 때까지 부모 프로세스는 대기 상태가 된다.
- waitpid
  - 부모 프로세스가 PID로 자식 프로세스를 지정하여 기다린다.

    지정하지 않은 자식 프로세스의 종료를 처리해주지 않는다.

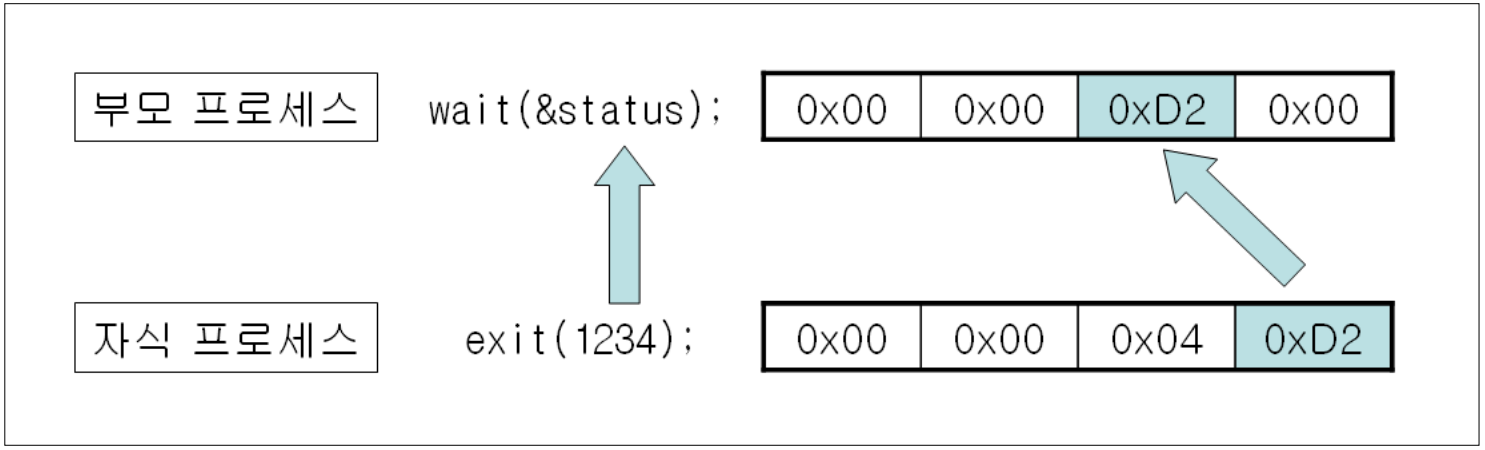
    자식 프로세스의 종료 순서에 상관없이 부모 프로세스가 처리 순서를 결정할 수 있다.
  - 옵션에 따라서 자식 프로세스가 종료할 때까지 대기 상태가 될 수도 있고 아닐 수도 있다.

forkwait.c

```
#include <unistd.h> ...
/* 부모 프로세스가 자식 프로세스를 생성하고 끝나기를 기다린다. */
int main()
{
    int pid, child, status;
    printf("[%d] 부모 프로세스 시작 ₩n", getpid( ));
    pid = fork();
    if (pid == 0) {
        printf("[%d] 자식 프로세스 시작 ₩n", getpid( ));
        exit(1);
    }
    child = wait(&status); // 자식 프로세스가 끝나기를 기다린다.
    printf("[%d] 자식 프로세스 %d 종료 ₩n", getpid(), child);
    printf("₩t종료 코드 %d₩n", status>>8);
}
```

```
[hansung@localhost ~]$ ./a.out
[26413] 부모 프로세스 시작
[26414] 자식 프로세스 시작
[26413] 자식 프로세스 26414 종료
        종료 코드 1
[hansung@localhost ~]$ _
```

- 자식 프로세스가 exit를 호출하면서 지정한 값을 부모 프로세스는 status 변수로 받는다.
- 자식 프로세스가 exit(n); 을 실행했을 때 부모 프로세스에게 전달되는 실제 값은 n의 하위 1바이트 뿐이다.
- 자식 프로세스가 전달한 1바이트 값은 부모 프로세스 쪽의 status 변수의 하위 두 번째 바이트에 저장된다.



waitpid.c

```
1  #include <sys/types.h>
...
7   /* 부모 프로세스가 자식 프로세스를 생성하고 끝나기를 기다린다. */
8   int main()
9   {
10      int pid1, pid2, child, status;
11
12      printf("[%d] 부모 프로세스 시작 ₩n", getpid( ));
13      pid1 = fork();
14      if (pid1 == 0) {
15          printf("[%d] 자식 프로세스[1] 시작 ₩n", getpid( ));
16          sleep(1);
17          printf("[%d] 자식 프로세스[1] 종료 ₩n", getpid( ));
18          exit(1);
19      }
```

waitpid.c

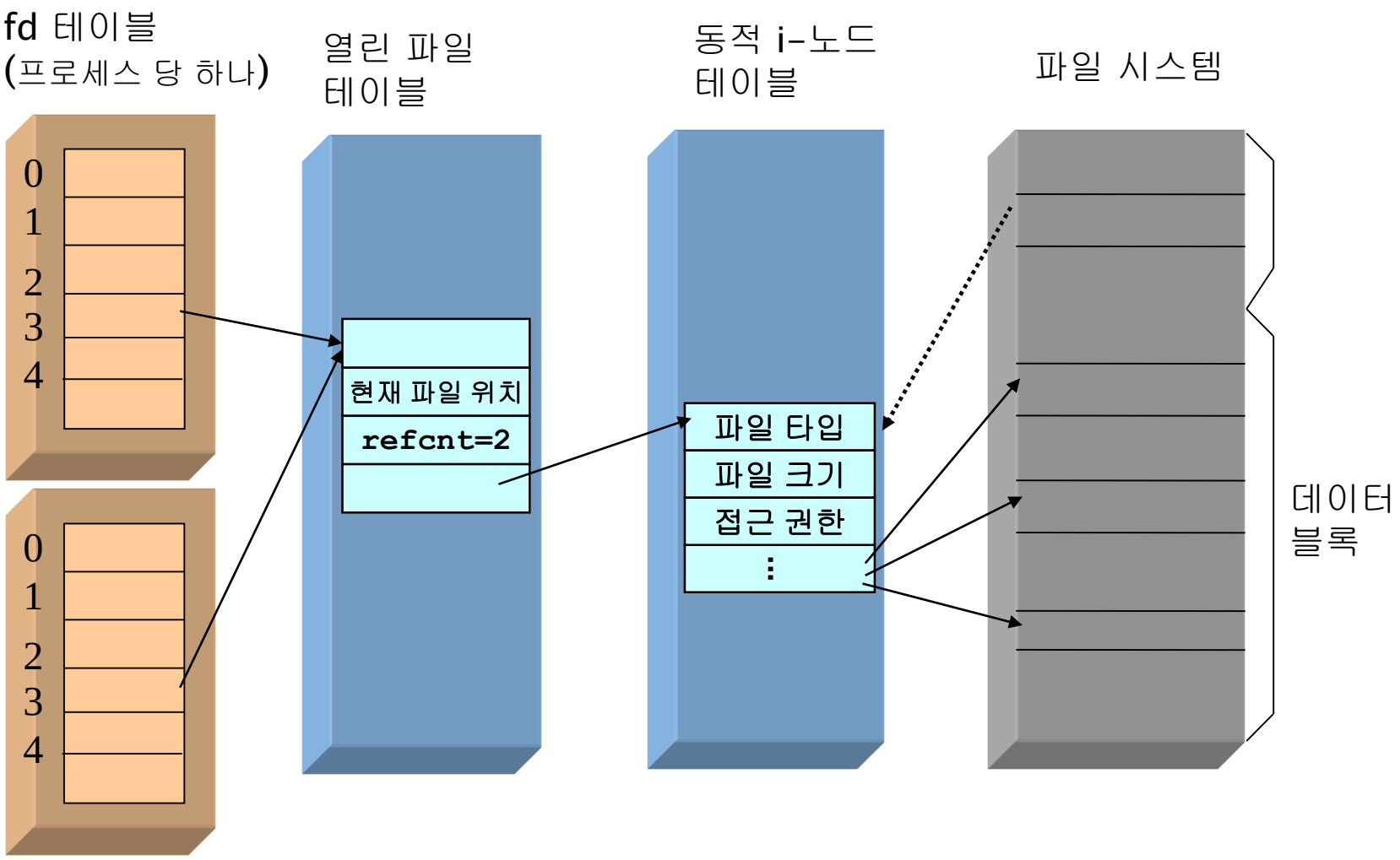
```
20
21          pid2 = fork();
22          if (pid2 == 0) {
23          printf("[%d] 자식 프로세스 #2 시작 ₩n", getpid( ));
24          sleep(2);
25          printf("[%d] 자식 프로세스 #2 종료 ₩n", getpid( ));
26          exit(2);
27          }
28          // 자식 프로세스 #1의 종료를 기다린다.
29          child = waitpid(pid1, &status, 0);
30          printf("[%d] 자식 프로세스 #1 %d 종료 ₩n", getpid( ), child);
31          printf("₩t종료 코드 %d₩n", status>>8);
32        }
```

fork() 후에 파일 공유

- 자식은 부모의 fd 테이블을 복사한다.
  - 부모와 자식이 같은 파일 디스크립터를 공유
  - 같은 파일 오프셋을 공유
  - 부모와 자식으로부터 출력이 서로 섞이게 됨

- 자식에게 상속되지 않는 성질
  - fork()의 반환값
  - 프로세스 ID
  - 파일 잠금
  - 설정된 알람과 시그널

fork()


fd 테이블
(프로세스 당 하나)
열린 파일
테이블
동적 i-노드
테이블
파일 시스템
0
1
2
3
4
0
1
2
3
4
현재 파일 위치
refcnt=2
파일 타입
파일 크기
접근 권한
데이터
블록

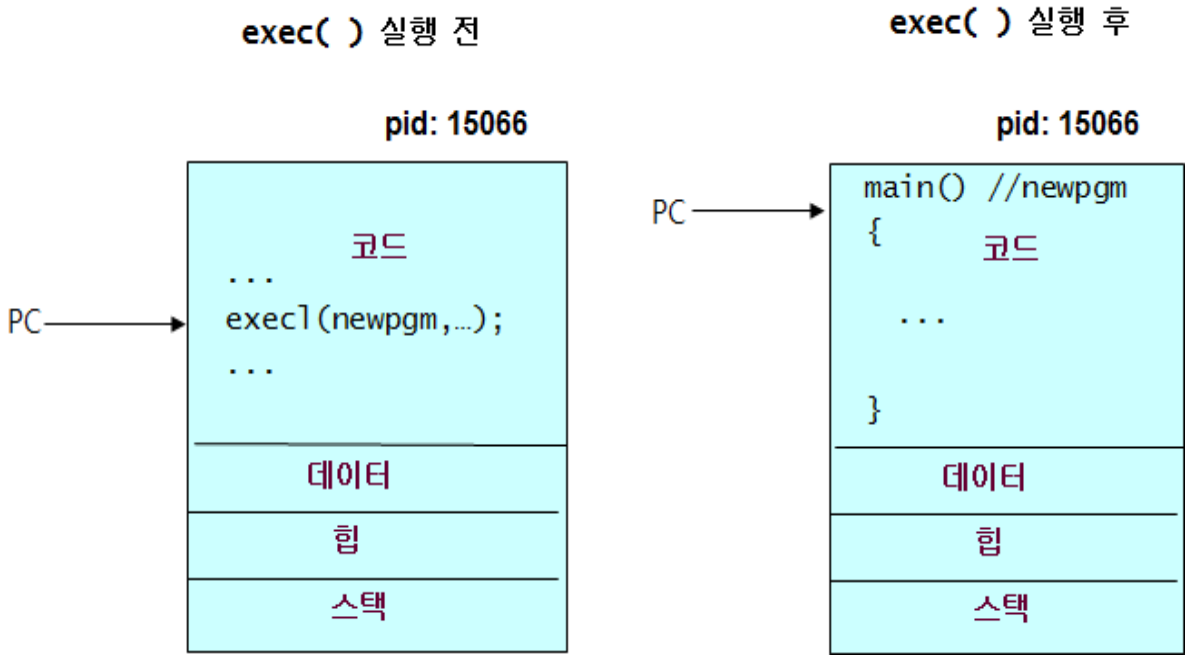
9.2 프로그램 실행

프로그램 실행

- fork() 후
  - 자식 프로세스는 부모 프로세스와 똑같은 코드 실행
- 자식 프로세스에게 새 프로그램 실행
  - exec() 시스템 호출 사용
  - 프로세스 내의 프로그램을 새 프로그램으로 대치
- 보통 fork() 후에 exec( )

프로그램 실행: exec()

- 프로세스가 exec() 호출을 하면,
  - 그 프로세스 내의 프로그램은 완전히 새로운 프로그램으로 대치
  - 자기대치(自己代置)
  - 새 프로그램의 main()부터 실행이 시작된다.



프로그램 실행: exec()

- exec() 호출이 성공하면 리턴할 곳이 없어진다.
- 성공한 exec() 호출은 절대 리턴하지 않는다.

```
#include <unistd.h>
int execl(char* path, char* arg0, char* arg1, ... , char* argn, NULL)
int execv(char* path, char* argv[ ])
int execlp(char* file, char* arg0, char* arg1, ... , char* argn, NULL)
int execvp(char* file, char* argv[ ])
```

호출한 프로세스의 코드, 데이터, 힙, 스택 등을 path가 나타내는 새로운 프로그램으로 대치한 후 새 프로그램을 실행한다.

성공한 exec( ) 호출은 리턴하지 않으며 실패하면 -1을 리턴한다.

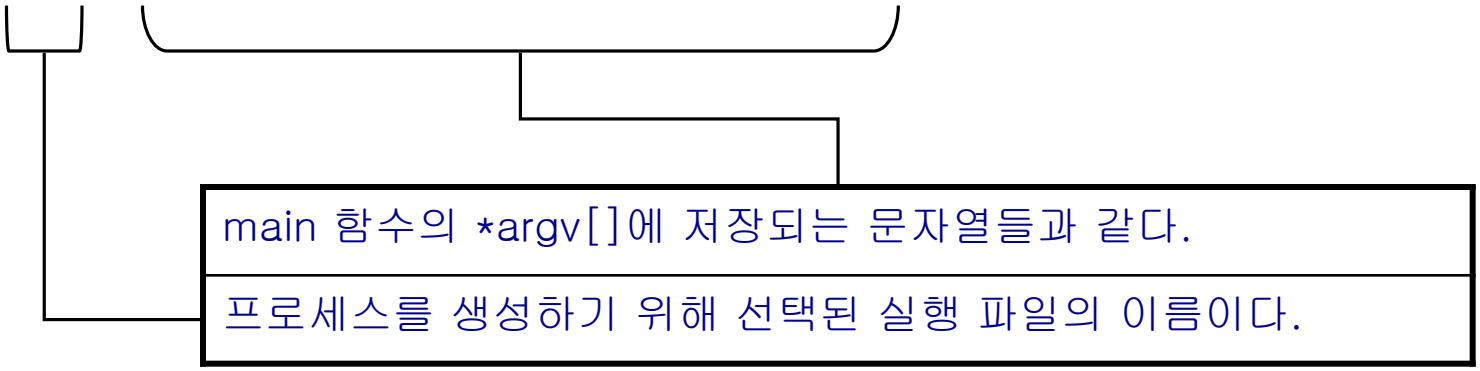
fork/exec

- 보통 fork() 호출 후에 exec() 호출
  - 새로 실행할 프로그램에 대한 정보를 arguments로 전달한다
    - .
- exec() 호출이 성공하면
  - 자식 프로세스는 새로운 프로그램을 실행하게 되고
  - 부모는 계속해서 다음 코드를 실행하게 된다.

```
if ((pid = fork()) == 0 ){
    exec( arguments );
    exit(1);
}
// 부모 계속 실행
```

fork/exec

```
$ ls -l apple/
...
execlp("ls", "ls", "-l", "apple/", (char *)0);
```



fork/exec

함수 이름에 p가 있고 없고의 차이

- p가 없으면 경로(path)로 실행 파일을 지정한다.
- p가 있으면 실행 파일의 이름만 지정한다.

- 경로를 지정하는 경우 (p가 없을 경우)
  - 지정한 (상대/절대)경로에서 해당 파일을 찾는다.

- 파일의 이름만 지정하는 경우 (p가 있을 경우)
  - 쉘 환경 변수 PATH에서 지정한 디렉터리를 차례대로 검색하여 찾는다.
    - 예) $ printenv PATH ← 환경 변수 PATH의 값을 출력한다.
    - 또는 $ echo $PATH

fork/exec

```
01 #include <unistd.h>
02
03 main()
04 {
05    printf("before executing ls -l₩n");
06    execl("/bin/ls", "ls", "-l", (char *)0);
07    printf("after executing ls -l₩n");
08 }
```

```
$ ex07-03
before executing ls -l
-rwxr-xr-x    1 usp      student     13707 Oct 24 21:57 ex07-03
$
```

fork/exec

```
01 #include <stdio.h>
02
03 main()
04 {
05     char *arg[] = {"ls", "-l", (char *)0};
06     printf("before executing ls -l₩n");
07     execv("/bin/ls", arg);
// exec가 성공하면 아래 부분은 절대 실행되지 않음
08     printf("after executing ls -l₩n");
09 }
```

```
$ ex07-04
before executing ls -l
-rwxr-xr-x     1 usp       student      13707 Oct 24 21:57 ex07-04
$
```

fork/exec

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

main()
{
    pid_t pid;

    printf("hello!₩n");

    pid = fork();

   if(pid > 0) { /* parent process */
        printf("parent₩n");
        sleep(1);
    }
    else if(pid == 0) { /* child process */
        printf("child₩n");
        execl("/bin/ls", "ls", "-l", (char *)0);
        printf("fail to execute ls₩n");
   }
    else
        printf("parent : fail to fork₩n");

    printf("bye!₩n");
}
```

fork/exec

```
$ ex07-07
hello!
parent
child
-rwxr-xr-x    1 usp      student     13856 Oct 25 15:56 ex07-07
bye!
$
```

execute1.c

```
#include <stdio.h>
/* 자식 프로세스를 생성하여 echo 명령어를 실행한다. */
int main( )
{
    printf("부모 프로세스 시작₩n");
    if (fork( ) == 0) {
        execl("/bin/echo", "echo", "hello", NULL);
        fprintf(stderr,"첫 번째 실패");
        exit(1);
    }
    printf("부모 프로세스 끝₩n");
}
```

```
[hansung@localhost ~]$ ./a.out
부모 프로세스 시작
부모 프로세스 끝
[hansung@localhost ~]$ hello
```

execute2.c

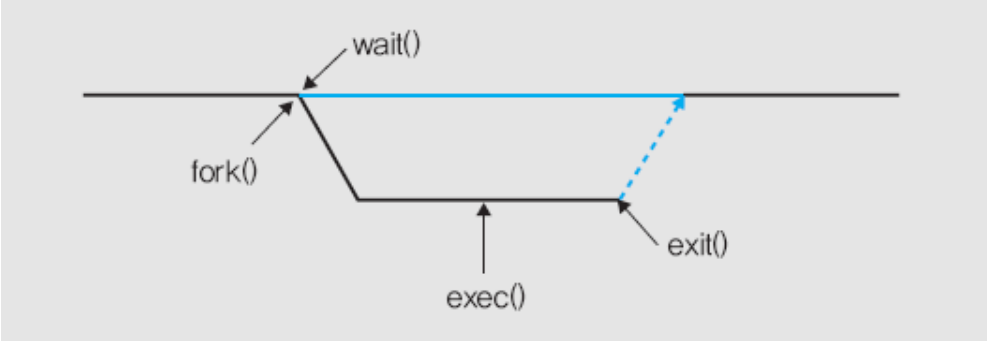
```
#include <stdio.h> …
/* 세 개의 자식 프로세스를 생성하여 각각
    다른 명령어를 실행한다. */
int main( )
{
   printf("부모 프로세스 시작\n");
   if (fork( ) == 0) {
      execl("/bin/echo", "echo", "hello", NULL);
      fprintf(stderr,"첫 번째 실패");
      exit(1);
   }
   if (fork( ) == 0) {
      execl("/bin/date", "date", NULL);
      fprintf(stderr,"두 번째 실패");
      exit(2);
}
   if (fork( ) == 0) {
      execl("/bin/ls","ls", "-l", NULL);
      fprintf(stderr,"세 번째 실패");
      exit(3);
   }
   printf("부모 프로세스 끝\n");
}
```

```
[hansung@localhost ~]$ ./a.out
부모 프로세스 시작
부모 프로세스 끝
[hansung@localhost ~]$ Tue Oct 15 09:43:10 PDT 2013
total 52
-rwxrwxr-x. 1 hansung hansung 5552 Oct 15 09:43 a.out
drwxr-xr-x. 2 hansung hansung 4096 Oct 15 06:19 Desktop
drwxr-xr-x. 2 hansung hansung 4096 Oct 15 06:19 Documents
drwxr-xr-x. 2 hansung hansung 4096 Oct 15 06:19 Downloads
drwxr-xr-x. 2 hansung hansung 4096 Oct 15 06:19 Music
drwxr-xr-x. 2 hansung hansung 4096 Oct 15 06:19 Pictures
drwxr-xr-x. 2 hansung hansung 4096 Oct 15 06:19 Public
drwxr-xr-x. 2 hansung hansung 4096 Oct 15 06:19 Templates
-rw-rw-r--. 1 hansung hansung  338 Oct 15 09:40 test2.c
-rw-rw-r--. 1 hansung hansung  618 Oct 15 09:43 test3.c
-rw-rw-r--. 1 hansung hansung  681 Oct 15 09:33 test.c
drwxr-xr-x. 2 hansung hansung 4096 Oct 15 06:19 Videos
hello

[hansung@localhost ~]$
```

execute3.c

```
#include <stdio.h> ...
/* 명령줄 인수로 받은 명령을 실행시킨다. */
int main(int argc, char *argv[])
{
    int child, pid, status;
    pid = fork( );
    if (pid == 0) { // 자식 프로세스
        execvp(argv[1], &argv[1]);
        fprintf(stderr, "%s:실행 불가₩n",argv[1]);
    } else { // 부모 프로세스
        child = wait(&status);
        printf("[%d] 자식 프로세스 %d 종료 ₩n", getpid(), pid);
        printf("₩t종료 코드 %d ₩n", status>>8);
    }
}
```



```
[hansung@localhost ~]$ ./a.out ls
a.out    Documents  Music     Public     test2.c  test4.c  test.c
Desktop  Downloads  Pictures  Templates  test3.c  test5.c  Videos
[26448] 자식 프로세스 26449 종료
        종료 코드 0
[hansung@localhost ~]$
```

```
01 #include <unistd.h>
02 #include <sys/types.h>
03
04 main()
05 {
06     pid_t pid1, pid2;
07     int status;
08
09     pid1 = pid2 = -1;
10
11     pid1 = fork();
12     if(pid1 > 0)
13         pid2 = fork();
14
15     if(pid1 > 0 && pid2 > 0)
16     {
17         waitpid(pid2, &status, 0);
18         printf("parent: child2 - exit(%d)₩n", status);
19         waitpid(pid1, &status, 0);
20         printf("parent: child1 - exit(%d)₩n", status);
21     }
```

```
22      else if(pid1 == 0 && pid2 == -1)
23      {
24          sleep(1);
25          exit(1);
26      }
27      else if(pid1 > 0 && pid2 == 0)
28      {
29          sleep(2);
30          exit(2);
31      }
32      else
33          printf("fail to forkWn");
34  }
```

```
$ ex08-03
parent: child2 - exit(512)
parent: child1 - exit(256)
$
```

```
01 #include <unistd.h>
02 #include <sys/types.h>
03 #include <sys/wait.h>
04
05 main()
06 {
07     pid_t pid;
08     int status = 0;
09
10     if((pid = fork()) > 0)
11     {
12         while(!waitpid(pid, &status, WNOHANG))
13         {
14             printf("parent: %d₩n", status++);
15             sleep(1);
16         }
17         printf("parent: child - exit(%d)₩n", status);
18     }
```

```
19      else if(pid == 0)
20      {
21          sleep(5);
22          printf("bye!₩n");
23          exit(0);
24      }
25      else
26          printf("fail to fork₩n");
27  }
```

```
$ ex08-04
parent: 0
parent: 1
parent: 2
parent: 3
parent: 4
bye!
parent: child - exit(0)
$
```

좀비 프로세스와 고아 프로세스

- 좀비 프로세스 (zombie process)
  - 부모 프로세스가 wait를 수행하지 않고 있는 상태에서 자식이 종료
  - 자식 프로세스의 종료를 부모 프로세스가 처리해주지 않으면 자식 프로세스는 좀비 프로세스가 된다.
  - 좀비 프로세스는 CPU, Memory 등의 자원을 사용하지 않으나, 커널의 작업 리스트에는 존재한다.
  - 좀비 프로세스는 누가 처리하나???
  - 좀비 발생을 방지하려면?
- 고아 프로세스 (orphan process)
  - 하나 이상의 자식 프로세스가 수행되고 있는 상태에서 부모가 먼저 종료
  - 부모 프로세스가 수행 중인 자식 프로세스를 기다리지 않고 먼저 종료
- init 프로세스
  - 좀비와 고아 프로세스의 관리는 결국 시스템의 init 프로세스로 넘겨진다.
  - init 프로세스가 새로운 부모가 된다.

```
/* zombie.c */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(int argc, char **argv)
{
  pid_t pid;
  int data=10;
  pid=fork();
  if(pid<0)
        printf("fork 실패 프로세스 id : %d ₩n", pid);
  printf("fork 성공 프로세스 id : %d ₩n", pid);
  if(pid==0) /* 자식 프로세스라면 */
        data+=10;
  else        /* 부모 프로세스라면 */
  {
        data-=10;
        sleep(20); /* 20초 동안 정지 상태에 들어간다 */
  }
  printf("data : %d ₩n", data);
  return 0;
}
```

```
jychang@cse-stu:~$ ps -elf | grep jychang
4 S root      5787  2904  0  80   0 -  2040 -      04:52 ?        00:00:00 sshd: jychang [priv]
5 S jychang   5789  5787  0  80   0 -  2040 -      04:52 ?        00:00:00 sshd: jychang@pts/0
0 S jychang   5790  5789  0  80   0 -  1815 -      04:52 pts/0    00:00:00 -bash
0 S jychang   5810  5790  0  80   0 -   403 -      04:53 pts/0    00:00:00 a.out
1 Z jychang   5811  5810  0  80   0 -     0 -      04:53 pts/0    00:00:00 [a.out] <defunct>
0 R jychang   5816  5790  0  80   0 -  1224 -      04:53 pts/0    00:00:00 ps -elf
0 S jychang   5817  5790  0  80   0 -  1099 -      04:53 pts/0    00:00:00 grep jychang
jychang@cse-stu:~$ data : 0
```

9.3 입출력 재지정

입출력 재지정

- 명령어의 표준 출력이 파일에 저장

  $ 명령어 > 파일

- 출력 재지정 기능 구현
  - 파일 디스크립터 fd를 표준출력(1)에 dup2()

```
fd = open(argv[1], O_CREAT|O_TRUNC|O_WRONLY, 0600);
dup2(fd, 1);
```

```
#include <unistd.h>
int dup(int oldfd);
oldfd에 대한 복제본인 새로운 파일 디스크립터를 생성하여 반환한다.
int dup2(int oldfd, int newfd);
oldfd을 newfd에 복제하고 복제된 새로운 파일 디스크립터를 반환한다.
```

redirect1.c

```
1 #include <stdio.h>
2 #include <fcntl.h>
3 …
4 /* 표준 출력을 파일에 재지정하는 프로그램 */
5 int main(int argc, char* argv[])
6 {
7       int fd, status;
8       fd = open(argv[1], O_CREAT|O_TRUNC|O_WRONLY, 0600);
9       dup2(fd, 1); /* 파일을 표준출력에 복제 */
10      close(fd);
11      printf("Hello stdout !₩n");
12      fprintf(stderr,"Hello stderr !₩n");
13 }
```

redirect2.c

```
1 #include <stdio.h>
2 #include <fcntl.h>
3
4 /* 자식 프로세스의 표준 출력을 파일
     에 재지정한다. */
5 int main(int argc, char* argv[])
6 {
7     int child, pid, fd, status;
8
9     pid = fork( );
10    if (pid == 0) {
11    fd = open(argv[1],O_CREAT|
            O_TRUNC| O_WRONLY, 0600);
12    dup2(fd, 1);   // 파일을 표준출력에 복제
13    close(fd);
14    execvp(argv[2], &argv[2]);
15    fprintf(stderr, "%s:실행 불가\n",argv[1]);
16    } else {
17        child = wait(&status);
18        printf("[%d] 자식 프로세스 %d 종료 \n",
                getpid(), child);
19    }
20 }
```

실행

```
$ a.out out wc you.txt
[2134] 자식프로세스 2133종료

$ cat out
   25   68   213 you.txt
```