7장 프로세스 다루기(1)

- 서론
- 예제 프로그램
- 함수
  - fork
  - exec 계열
  - fork와 exec를 함께 사용하기
  - exit
  - atexit
  - _exit

프로세스를 생성하고 종료하는 시스템 호출/표준 라이브러리 함수

함수	의미
fork	자신과 완전히 동일한 프로세스를 생성한다.
exec 계열	지정한 실행 파일로부터 프로세스를 생성한다.
exit	종료에 따른 상태 값을 부모 프로세스에게 전달하며 프로세스를 종료한다.
atexit	exit로 프로세스를 종료할 때 수행할 함수를 등록한다.
_exit	atexit로 등록한 함수를 호출하지 않고 프로세스를 종료한다.

【예제】 예제 프로그램(1/3)


```
01 #include <unistd.h>
02 #include <stdlib.h>
03 #include <sys/types.h>
04
05 void cleanupaction(void);
06
07 main()
08 {
09     pid_t pid;
10     int i;
11
12     for(i = 0; i < 3; i++)
13    {
14         printf("before fork [%d]₩n", i);
15         sleep(1);
16     }
17
18     pid = fork();
```

09 프로세스 식별 번호를 저장한다

18 자식 프로세스를 생성한다. Pid 에는 사직 프로세스의 식별 번호가 저장된다.

표준입력 스트림

【예제】 예제 프로그램(2/3)


```
19    if(pid > 0) {
20       for( ; i < 7; i++) {
21           printf("parent [%d]₩n", i);
22           sleep(1);
23       }
24
25     atexit(cleanupaction);
26    }
27    else if(pid == 0) {
28       for( ; i < 5; i++) {
29           printf("child [%d]₩n", i);
30           sleep(1);
31           execl("/bin/ls", "ls", "-l", (char *)0);
32       }
33    }
34    else {
35       printf("fail to fork child process₩n");
36    }
```

19 부모 프로세스가 수행하는 부분이다

25 clean-up-action을 등록한다.

27 자식 프로세스가 수행하는 부분이다.

31 “ls”를 실행하여 새로운 프로세스를 생성한다

34 자식 프로세스 생성에 실패했을 때 실행되는 부분이다

표준입력 스트림

【예제】 예제 프로그램(3/3)


```
37      exit(0);
38 } /* end of main */
39
40 void cleanupaction(void)
41 {
42      printf("clean-up-action₩n");
43 }
```

40 exit 호출 시 실행될 함수이다.
(clean-up-action)

표준입력 스트림

실행 결과

```
$ ex07-01
before fork [0]
before fork [1]
before fork [2]
parent [3]
child [3]
parent [4]
-rwxr-xr-x    1 usp      student     14456 Oct 26 00:31 ex07-01
parent [5]
parent [6]
clean-up-action
$
```

프로세스를 복제하여 완전히 동일한 프로세스를 생성한다

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

pid_t fork(void);
```

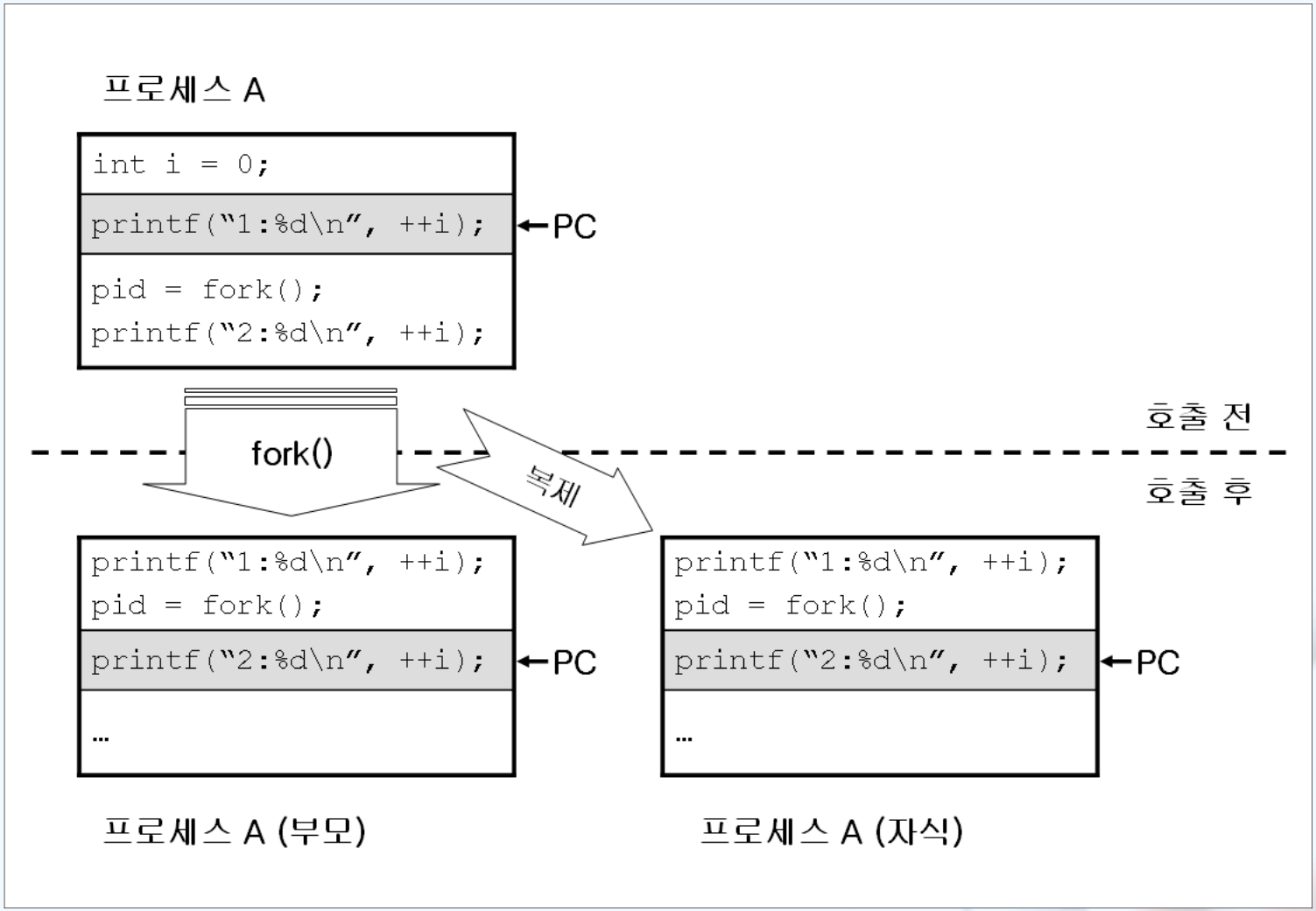
반환값	fork 호출이 성공하여 자식 프로세스가 만들어지면 부모 프로세스에서는 자식 프로세스의 프로세스 ID가 반환되고 자식 프로세스에서는 0을 반환한다. fork 호출이 실패하여 자식 프로세스가 만들어지지 않으면 부모 프로세스에서는 -1이 반환된다.

- **프로세스는 실행 파일로 존재하는 프로그램으로부터 생성되는 것이 일반적이다.그러나, fork를 사용하면 실행 중인 프로세스를 복제하여 새로운 프로세스를 생성할 수 있다.**

부모 (parent) 프로세스와 자식 (child) 프로세스

- fork를 호출하여 새로운 프로세스를 생성할 때, 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.

fork를 사용한 프로세스 생성



fork를 호출하는 프로그램의 구조

- fork를 호출하는 시점을 기준으로, fork를 호출한 이후에 부모 프로세스가 할 일과 자식 프로세스가 할 일을 구분한다.
  - fork의 반환 값으로 부모 프로세스와 자식 프로세스를 구분한다.

```
pid = fork();    /* fork 호출이 성공하면 자식 프로세스가 생성된다. */

if(pid == 0)
    /* 자식 프로세스가 수행할 부분 */
    …;
else if(pid > 0)
    /* 부모 프로세스가 수행할 부분 */
    …;
else
    /* fork 호출이 실패할 경우 수행할 부분 */
    …;
```

【예제 7-2】 ex07-02.c


```
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}
```

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

- 경로 이름 또는 파일 이름으로 지정한 실행 파일을 실행하여 프로세스 생성한다.

```
#include <unistd.h>

extern char **environ;

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

path	실행 파일의 경로로 상대 경로와 절대 경로 모두 사용할 수 있다.
file	경로 이름이 아닌 실행 파일의 이름이다.
반환값	호출이 성공하면 호출하는 프로세스에서는 반환 값을 받을 수 없다. 만약 함수 호출 후 -1이 반환되면 이는 함수 호출이 실패했음을 의미한다.

- 경로 이름 또는 파일 이름으로 지정한 실행 파일을 실행하여 프로세스 생성한다.

```
#include <unistd.h>

extern char **environ;

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

arg	path나 file로 지정한 실행 파일을 실행할 때 필요한 명령어 라인의 옵션과 인자이다. 한 개 이상을 지정할 수 있으며 마지막 인자는 반드시 NULL 포인터로 지정해야 한다.
argv	arg와 같은 의미를 가지나 문자형 포인터의 배열로 형태가 다르다. 배열의 마지막은 NULL 문자열로 끝나야 한다.

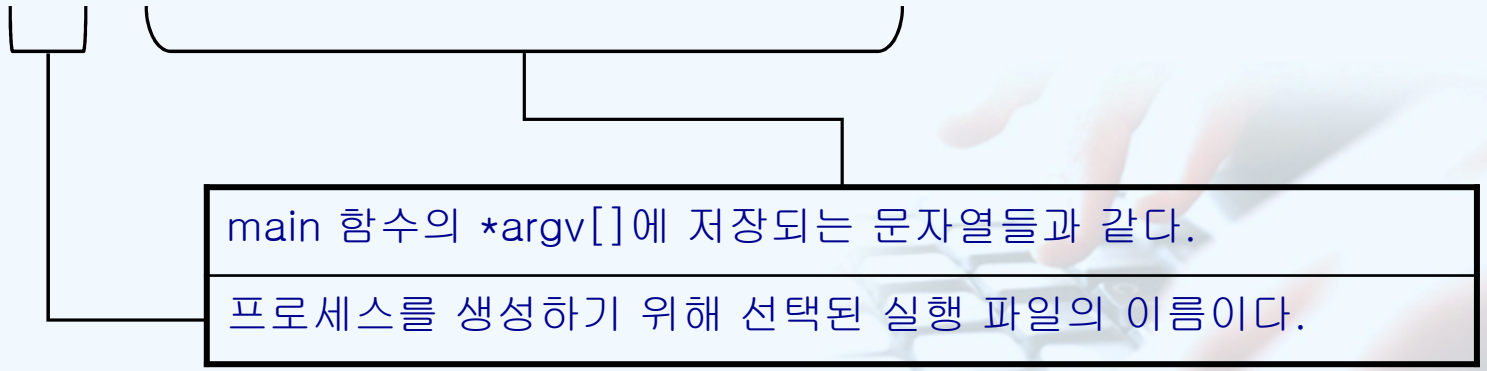
- **exec 계열의 함수는 지정한 실행 파일로부터 프로세스를 생성한다.**

  fork는 실행 중인 프로세스로부터 새로운 프로세스를 생성한다

- **exec 계열의 함수의 사용 예**

  셸 프롬프트 상에서 "ls"를 실행하는 것과 비교

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



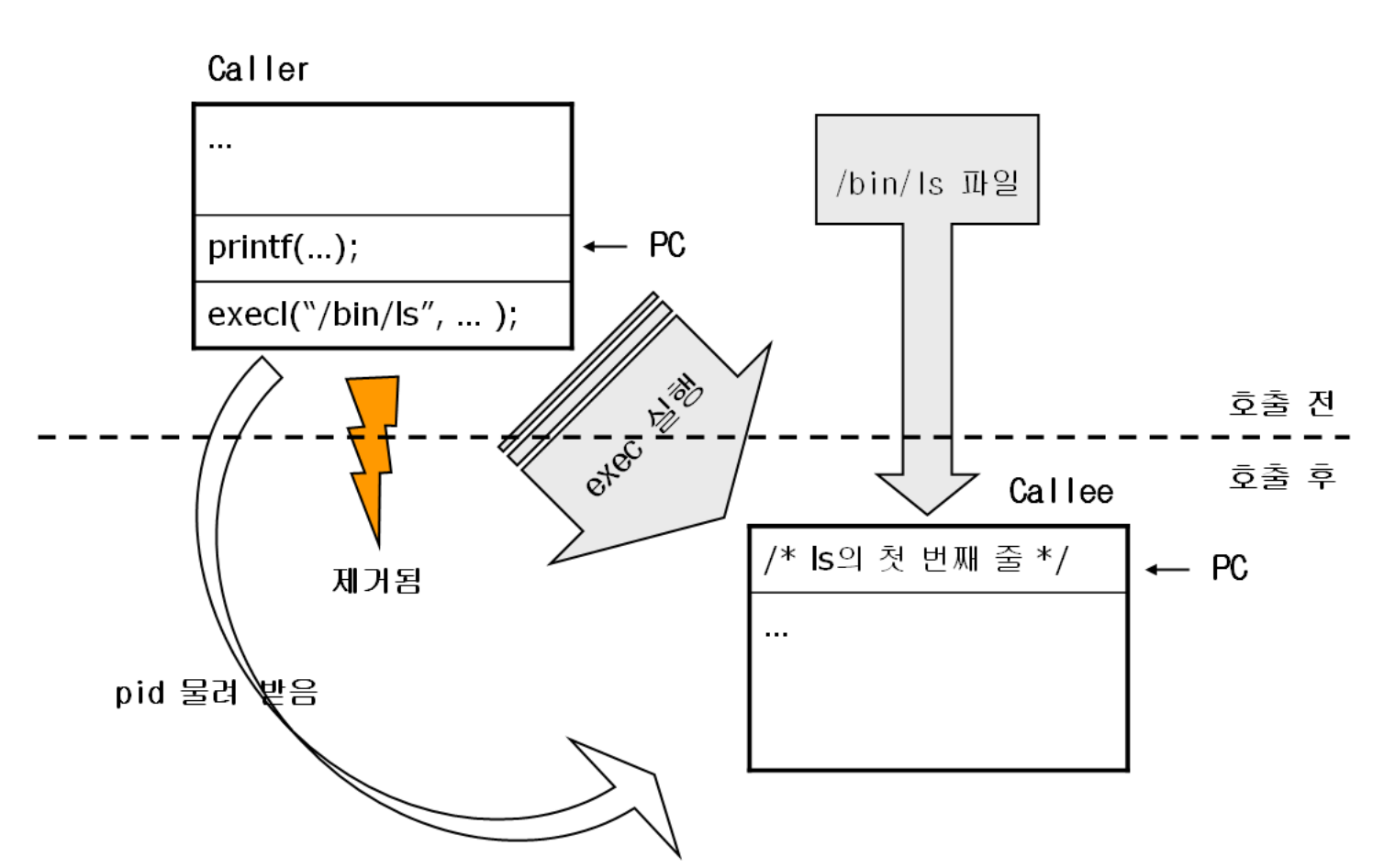
호출 프로세스와 피호출 프로세스

- 호출 프로세스 (caller process)
  exec를 실행하는 프로세스
- 피호출 프로세스 (callee process)
  exec에 의해 생성되는 프로세스

exec를 성공적으로 호출한 결과

- 호출 프로세스는 종료된다.
- 호출 프로세스가 메모리 영역을 피호출 프로세스가 차지한다.
- 호출 프로세스의 PID를 피호출 프로세스가 물려받는다.

exec 계열을 사용한 프로세스 생성



exec 계열 함수의 구분

- 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.

【예제 7-3】 ex07-03.c


```
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 }
```

06 인자의 나열이 끝났음을 의미한다.

07 exec 호출이 성공하면 실행되지 않는다. (될수가 없다.)

표준입력 스트림

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

【예제 7-4】 ex07-04.c

```
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);
08     printf("after executing ls -l₩n");
09 }
```

5~7

05 이름에 'l'이 포함된 경우와 'v'가 포함된 경우의 차이점이다

표준입력 스트림

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

fork와 exec의 비교 (1)

	fork	exec 계열
프로세스의 원본	부모 프로세스를 복제하여 새로운 프로세스를 생성한다.	지정한 프로그램(파일)을 실행하여 프로세스를 생성한다.
셸 명령줄의 프로그램 인자	새롭게 지정할 수 없고 부모 프로세스의 것을 그대로 사용한다.	필요할 경우 적용할 수 있다.
부모(또는 호출) 프로세스의 상태	자식 프로세스를 생성한 후에도 자신의 나머지 코드를 실행한다.	호출이 성공할 경우 호출(Caller) 프로세스는 종료된다.
자식(또는 피호출) 프로세스의 메모리 상의 위치	부모 프로세스와 다른 곳에 위치한다.	호출 프로세스가 있던 자리를 피호출 프로세스가 물려받는다.
프로세스 생성 후 자식(또는 피호출) 프로세스의 프로그램 코드의 시작 지점	fork 호출 이후부터 수행된다.	프로그램의 처음부터 수행된다.
프로세스 식별 번호 (PID)	자식 프로세스는 새로운 식별 번호를 할당받는다.	호출 프로세스의 식별 번호를 피호출 프로세스가 물려받는다.
프로세스의 원본인 파일에 대한 권한	부모 프로세스를 복제하므로 상관없다.	실행 파일에 대한 실행 권한이 필요하다.

fork와 exec의 비교 (2)

- Fork
  - 자신과 동일한 자식 프로세스만 생성할 수 있다.
  - 다른 종류의 프로세스를 생성할 수 없다.
  - 자식 프로세스를 생성하더라도 자신은 종료되지 않는다.
- exec
  - 자신과 다른 종류의 프로세스를 생성할 수 있다.
  - 새로운 프로세스를 생성하면 자신은 종료된다.

fork와 exec를 함께 사용하기

- fork를 호출하여 자식 프로세스를 생성 한 후에 자식 프로세스가 exec를 호출하여 새로운 프로세스를 생성한다.
- 결과적으로, 부모 프로세스는 종류가 다른 자식 프로세스를 생성하고 자신 역시 나머지 작업을 계속할 수 있다.

【예제 7-6】 ex07-06.c


```
01 #include <unistd.h>
02 #include <sys/types.h>
03
04 main()
05 {
06     pid_t pid;
07
08     printf("hello!₩n");
09
10     pid = fork();
11
12    if(pid > 0) { /* parent process */
13         printf("parent₩n");
14         sleep(1);
15     }
16     else if(pid == 0) { /* child process */
17         printf("child₩n");
18        execl("/bin/ls", "ls", "-l", (char *)0);
19        printf("fail to execute ls₩n");
20    }
21     else
22         printf("parent : fail to fork₩n");
23
24     printf("bye!₩n");
25 }
```

12~14

12 부모 프로세스는 자식 프로세스의 생존 여부에 상관없이 자신의 나머지 일을 수행한다

16~19

16 자식 프로세스가 exec를 호출하여 새로운 프로세스를 생성한다. 새로운 프로세스를 생성하고 자식 프로세스는 종료된다.

표준입력 스트림

실행 결과

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

- 프로세스를 종료하면서 부모 프로세스에게 종료와 관련된 상태 값을 넘겨준다.

```
#include <stdlib.h>

void exit(int status);
```

status	부모 프로세스에게 전달되는 상태 값으로 0~255(1바이트)의 값이 사용된다.
반환값	없음

- exit는 프로세스를 의도적으로 종료시킨다.
  이외에 프로세스가 종료하는 경우는
  - 더 이상 수행할 문장이 없거나
  - main 함수 내에서 return문을 수행할 때이다.
- status의 값은 0~255 사이의 값으로 각각에 대한 정해진 의미가 없다.
  - 프로그램 작성자가 임의로 정해서 사용한다.

프로세스가 exit를 호출하여 종료할 때 수행되는 함수들을 등록한다.

```
#include <stdlib.h>

int atexit(void (*function)(void));
```

function	atexit로 등록할 함수의 이름이다.
반환값	호출이 성공하면 0을 반환하고, 실패하면 0이 아닌 값을 반환한다.

- **function**
  - **함수의 이름으로 함수는 void function(void); 형으로 정의되어야 한다.**
- **종료 시 마무리 작업 (clean-up-action)**
  - **프로세스가 종료할 때 깔끔한 마무리를 위해 수행해야 하는 작업들**
  - **최대 32개까지 등록할 수 있다. (실제 실행 순서는 등록 순서의 역순)**

【예제 7-8】 ex07-08.c


```
01 #include <unistd.h>
02 #include <stdlib.h>
03
04 void func1(void);
05 void func2(void);
06
07 main()
08 {
09     printf("hello!₩n");
10     atexit(func1);
11     atexit(func2);
12
13     printf("bye!₩n");
14     exit(0);
15 }
```

```
16 void func1(void)
17 {
18     printf("func1₩n");
19 }
20
21 void func2(void)
22 {
23     printf("func2₩n");
24 }
```

```
$ ex07-08
hello!
bye!
func2
func1
$
```

exit 함수와 같지만 clean-up-action을 수행하지 않는다

```
#include <unistd.h>

void _exit(int status);
```

status	부모 프로세스에게 전달되는 상태 값으로 0~255(1바이트)의 값이 사용된다.
반환값	없음

- **_exit는 atexit로 clean-up-action에 해당하는 함수들을 등록해 놓았더라도 종료 할 때 이를 수행하지 않는다.**

【예제 7-9】 ex07-09.c


```
01 #include <unistd.h>
02 #include <stdlib.h>
03
04 void func1(void);
05 void func2(void);
06
07 main()
08 {
09     printf("hello!₩n");
10     atexit(func1);
11     atexit(func2);
12
13     printf("bye!₩n");
14     _exit(0);
15 }
```

```
$ ex07-09
hello!
bye!
$
```

```
16 void func1(void)
17 {
18     printf("func1₩n");
19 }
20
21 void func2(void)
22 {
23     printf("func2₩n");
24 }
```