Chapter 9: Signal (2/5)

김선영
sunyzero@gmail(dot)com
버 전: 2017-06-06

가메 출판사 http://www.kame.co.kr
저자홈페이지 http://sunyzero.tistory.com

SIGCHLD / Zombie process

defunct process의 배경적 이해, 개념

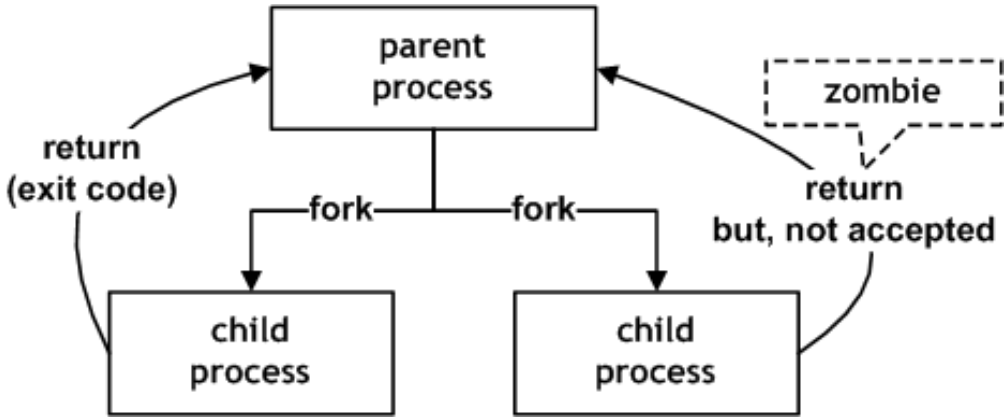
fork : Create process

- UNIX-like에서 모든 프로세스는 fork를 통해 만들어진다.
  - (예외: init 프로세스 or 조상 프로세스)
- fork시 프로세스는 복제되어 2개가 된다.
  - original : parent process라고 부른다.
  - replicated : child process라고 부른다.

zombie process

zombie process와 프로세스 종료코드

- 자식 프로세스는 종료하면서 부모 프로세스에게 보고하는 체계가 있다.
  - zombie는 종료하면서 보고하는 체계가 비정상으로 작동하지 않은 경우에 보인다.



zombie의 발생은 부모를 잘못 만난 탓이다.

종료된 child 프로세스는 아주 짧은 시간동안 zombie 프로세스가 되지만
parent 프로세스의 waitpid로 인해 곧 해제된다.
하지만 waitpid로 처리되지 않는다면 pending된 상태가 되고
이런 zombie 프로세스는 비로소 유저에게 계속 보이게 된다.

zombie process (con't)

- zombie process가 생기는 이유는?
  - process가 차지하는 메모리 공간은?
    - process image (실행 코드 부분)
    - meta data
  - 프로세스가 종료되면 process image는 즉시 해제되지만...
    - meta data가 아직 남아있다면 정상적인 상황인가?
  - 예를 들어 file도 마찬가지다.
    - file에는 meta data + data 부분이 존재하는데..
    - = data 부분만 삭제되고 meta data는 남아있다면 journaling에서는 어떻게 처리되는가를 생각해보자.

*** zombie process는 defunct process라고도 한다.**

shell vs child process (con't)

- 컴파일, 실행해보면 child process의 리턴값을 알 수 있다

```
$ cat helloworld.c
#include <stdio.h>
#include <unistd.h>
int main() {
    dprintf(STDOUT_FILENO, "Hello world\n"); // POSIX.1-2008
    return 234;
}
$ make helloworld
cc     helloworld.c    -o helloworld
$ ./helloworld
Hello world
$ echo $?
```

return 2340 이 되면?

무슨 일이 생기는가?

return type

- main()의 리턴값은 부모에게 전달된다.
  - shell에서 실행된 명령어의 리턴은 누가 읽는가?
    - e.g.) bash prompt에서 ls라고 명령하면...
      - 부모 프로세스 = bash shell, 자식 프로세스 = ls

```
$ ls -al
... 생략 ...
$ echo $?
0
```

```
$ ./helloworld
Hello world
$ echo $?
0
```

helloworld.c 에서
main의 리턴값을
1로 변경하면?

zombie를 없애려면?

- 1. parent process에서 wait 계열 함수를 호출한다.
  - wait 계열 함수 목록 : wait, waitpid, waitid
- 2. SIGCHLD 시그널 핸들러를 SIG_IGN 으로 변경한다.
- 3. Child process를 orphan process로 만든다.
  - 이 경우에 PID 1이 step parent가 되어 zombie가 되지 않도록 처리한다.

* 위에서 가장 추천하는 방법은?

SIGCHLD : Process

Process hierarchy

- init
  - original parent process - ancestor
    - PID 1번을 사용한다.
  - 최근의 리눅스는 systemd로 대체되었다.
- parent와 child process의 관계
  - parent는 child의 보고를 받아야 하며...
  - parent는 child를 signal로 제어할 수 있다.
    - 시그널 전파 : process group을 이용하여...
    - 시그널 수신 : child의 상태 감지

SIGCHLD

- 수신 조건
  - child가 종료(TERM)된 경우
  - child가 정지(STOP)된 경우
    - sigaction: SA_NOCLDSTOP가 지정되면?
- daemon
  - child의 종료 상태만 관심을 가지는 대표적인 예
  - daemon 프로세스를 만드는 법은?
    - 세션, 제어터미널의 개념이 먼저!

sig_nocldstop.c (1/2)

```
void sa_handler_chld(int signum);
int main() {
    int     ret;
    struct sigaction sa_chld;
    memset(&sa_chld, 0, sizeof(struct sigaction));
    sa_chld.sa_handler = sa_handler_chld;
    sigfillset(&sa_chld.sa_mask);   /* 시그널 블록 매스크를 모두 채운다. */
    sa_chld.sa_flags = SA_NOCLDSTOP;
    sigaction(SIGCHLD, &sa_chld, NULL);
    printf("[MAIN] SIGNAL Handler installed\n");
    switch((ret = fork())) {
        case 0: /* child process */
            pause();
            exit(EXIT_SUCCESS);
        case -1: /* error */
            break;
        default:
            printf("- Child pid = %d\n", ret);
            break;
    }
```

sig_nocldstop.c (2/2)

```
    while (1) {
        pause();
        printf("[MAIN] Recv SIGNAL...\n");
    }
    return EXIT_SUCCESS;
}
void sa_handler_chld(int signum) {
    pid_t   pid_child;
    int     status;
    printf("[SIGNAL] RECV SIGCHLD signal\n");
    while (1) {
        if ((pid_child = waitpid(-1, &status, WNOHANG)) > 0) {
            printf("\t- Exit child PID(%d) \n", pid_child);
        } else {
            break; /* 좀비 프로세스가 더이상 존재하지 않는다. */
        }
    } /* loop: while */
}
```

sig_nocldstop: exec.

```
$ ./sig_nocldstop
[MAIN] SIGNAL Handler installed
[MAIN] pid is 4403
- Child pid = 4404

[2]+  Stopped                 ./sig_nocldstop

$ fg
./sig_nocldstop
```

Ctrl-Z 입력

SA_NOCLDSTOP을 제거한 뒤에 실행

```
$ ./sig_cldstop
[MAIN] SIGNAL Handler installed
[MAIN] pid is 4429
- Child pid = 4430

[2]+  Stopped                 ./sig_nocldstop

$ fg
./sig_nocldstop
[SIGNAL] RECV SIGCHLD signal
[MAIN] Recv SIGNAL...
```

Ctrl-Z 입력

wait, waitpid, waitid

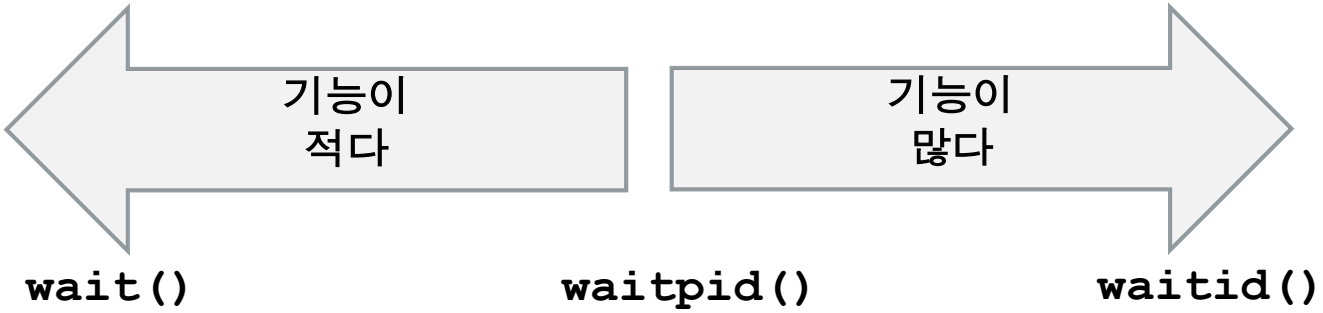
```
pid_t  wait(int  *status);
pid_t  waitpid(pid_t  pid,  int  *status,  int  options);
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

- pid
  - -1 : 자식 프로세스 중 아무나 (WAIT_ANY)
  - positive : PID 지정
  - 0 : 동일 프로세스 그룹의 속한 자식 프로세스(WAIT_MYPGRP)
  - neg < -1 : 절대값과 같은 pid의 자식 프로세스
    - 특정 프로세스 그룹 지정 (fork-exec의 경우를 생각해보자)

wait, waitpid, waitid

```
pid_t  wait(int  *status);
pid_t  waitpid(pid_t  pid,  int  *status,  int  options);
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

- 기능



waitpid: options

WNOHANG	준비된 자식 프로세스가 없다면 기다리지 않고 즉시 반환함 (넌블럭킹)
WUNTRACED	종료되거나 정지된 자식 프로세스에 대해서 모두 보고한다.

wait, waitpid : macro for status

정상 종료인 경우	WIFEXITED(status)	자식 프로세스가 종료한 경우라면 non-zero를 리턴.
	WEXITSTATUS(status)	WIFEXITED에서 non-zero가 리턴된 경우에만 사용한다. 자식 프로세스의 종료값을 리턴한다.
시그널로 종료된 경우	WIFSIGNALED(status)	자식 프로세스가 시그널로 종료되었다면 non-zero 임
	WTERMSIG(status)	WIFSIGNALED에서 non-zero가 리턴된 경우에만 사용한다. 자식 프로세스가 수신한 종료 시그널 번호를 반환한다.
정지된 경우	WIFSTOPPED(status)	자식 프로세스가 정지되었다면 non-zero 임 = 이 매크로는 waitpid(2)에 WUNTRACED 옵션을 사용한 경우임
	WSTOPSIG(status)	WIFSTOPPED에서가 non-zero가 리턴된 경우에만 사용한다. 자식 프로세스가 수신한 정지 시그널 번호를 반환한다.

waitpid : test macro

sig_nocldstop.c (adv)

```
void sa_handler_chld(int signum) {
    pid_t   pid_child;
    int     status;
    while (1) {
        if ((pid_child = waitpid(-1, &status, WNOHANG)) > 0) {
            printf("\t- child pid(%d)\n", pid_child);
            if (WIFEXITED(status)) {
                printf(">> WEXITSTATUS(%d)\n", WEXITSTATUS(status));
            } else if (WIFSIGNALED(status)) {
                printf(">> WTERMSIG(%d)\n", WTERMSIG(status));
            } else {
                /* STOPED */
            }
        } else {
            break; /* zombie process no longer exists */
        }
    }
}
```

waitid

```
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

- 확장된 시그널 처리를 위한 함수
  - wait, waitpid보다 더 세밀한 처리가 가능하다.
- idtype_t idtype, id_t id

idtype	P_PID	두번째 인수인 id에 지정한 PID를 가지는 child process의 종료를 기다린다.
	P_PGID	두번째 인수인 id에 지정한 PGID에 속한 child process의 종료를 기다린다.
	P_ALL	child process 중에 아무나 종료를 기다린다. 두번째 인수인 id는 무시된다.

waitid (con't)

```
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

- siginfo_t infop
  - waitid 호출 성공시 infop(siginfo_t 구조체)에 설정되는 값

si_pid	child process의 pid
si_uid	child process의 RUID
si_signo	SIGCHLD로 리턴된다.
si_code	다음 코드 중에 하나로 설정된다. CLD_EXITED (정상 종료), CLD_KILLED(시그널에 의한 종료), CLD_DUMPED(시그널에 의한 종료로 코어가 덤프됨), CLD_TRAPPED(트랩됨), CLD_STOPPED(정지됨), CLD_CONTINUED(재개됨)
si_status	si_code에 부가 정보 상태값. (예를 들어 si_code가 CLD_EXITED이면 이 값은 자식 프로세스의 종료값)

waitid (con't)

- options

WNOHANG	준비된 자식 프로세스가 없다면 기다리지 않고 즉시 반환함(넌블럭킹)
WNOWAIT	자식 프로세스의 종료 상태를 읽고 남겨두어 다시 읽을 수 있게 한다.
WSTOPPED	정지된 자식 프로세스에 대해서 보고한다.
WCONTINUED	재개된 자식 프로세스에 대해서 보고한다.
WEXITED	정상 종료된 자식 프로세스에 대해 보고한다.

  - OR결합이 가능하다.
    - WEXITED | WSTOPPED | WCONTINUED

sig_waitid.c (1/4)

```
void sa_handler_chld(int signum);
int main() {
    int     ret;
    struct sigaction sa_chld;
    memset(&sa_chld, 0, sizeof(struct sigaction));
    sa_chld.sa_handler = sa_handler_chld;
    sigfillset(&sa_chld.sa_mask);   /* fill signal block mask. */
    sigaction(SIGCHLD, &sa_chld, NULL);
    printf("[MAIN] SIGNAL Handler installed, pid(%d)\n", getpid());
    switch((ret = fork())) {
        case 0:
            pause();
            exit(EXIT_SUCCESS);
        case -1:
            break;
        default:
            printf("- Child pid = %d\n", ret);
            break;
    } /* end : switch */
```

sig_waitid.c (2/4)

```
    while (1) {
        pause();
        printf("[MAIN] Recv SIGNAL...\n");
    } /* end : while */
    return EXIT_SUCCESS;
} /* end : main */
```

sig_waitid.c (3/4)

```
void sa_handler_chld(int signum) {
    printf("[SIGNAL] RECV SIGCHLD signal\n");
    int     optflags = WNOHANG|WEXITED|WSTOPPED|WCONTINUED;
    siginfo_t wsiginfo = {.si_pid = 0};
    char    *str_status;
    while (1) {
        if (waitid(P_ALL, 0, &wsiginfo, optflags) == 0 && wsiginfo.si_pid != 0) {
            switch (wsiginfo.si_code) {  /* si_code에 따라서 분기 */
                case CLD_EXITED:   /* 정상 종료된 경우 */
                    str_status = "Exited";
                    break;
                case CLD_KILLED:   /* 시그널에 의해 종료된 경우 */
                    str_status = "Killed";
                    break;
                case CLD_DUMPED:  /* 시그널에 의해 종료되면서 코어 덤프한 경우 */
                    str_status = "Dumped";
                    break;
                case CLD_STOPPED: /* 정지된 경우 */
                    str_status = "Stopped";
                    break;
```

sig_waitid.c (4/4)

```
                case CLD_CONTINUED: /* 재개된 경우 */
                    str_status = "Continued";
                    break;
                default:
                    str_status = "si_code";
                    break;
            } /* end : switch */
            printf("child pid(%d) %s(%d)\n",
                    wsiginfo.si_pid, str_status, wsiginfo.si_status);
        } else {
            break;
        }
    } /* end : while */
} /* end : sa_handler_chld */
```

Run

```
$ ./sig_waitid
[MAIN] SIGNAL Handler installed, pid(41941)
- Child pid = 41942
[SIGNAL] RECV SIGCHLD signal
child pid(41942) Stopped(19)
[MAIN] Recv SIGNAL...
[SIGNAL] RECV SIGCHLD signal
child pid(41942) Continued(18)
[MAIN] Recv SIGNAL...
[SIGNAL] RECV SIGCHLD signal
child pid(41942) Killed(15)
[MAIN] Recv SIGNAL...
```