Chapter 9: Signal (1/5)

김선영
sunyzero@gmail(dot)com
버 전: 2017-06-04

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

Abstract

- UNIX std. Signal
  - signal() vs sigaction()
  - EINTR
  - SIGCHLD
- Session, Process group
- Signal block mask
- async-signal-safe
- alternative stack

Signal handling

- Common Exception handling
- Process's operation from external environment
- Exception of Resource allocation
- Notification of Event
  - Realtime extension

UNIX signal

- UNIX traditional signal
  - 약 30여개의 시그널 지원
- Realtime signal
  - 추가 시그널 (RT~)
  - kill -l 로 리스트를 확인해보자.

Tip!

- kill은 죽이는 기능이 아니다. 시그널을 전송하는 기능이다.
- 초기 유닉스에서 시그널은 죽이는 용도로 사용했기에 잘못 붙여진 이름이다.

Signal

Signal list (kill -l)

```
[sunyzero@localhost work]$ $ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF                                    30) SIGPWR
31) SIGSYS      34) SIGRTMIN                                    SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5                                  SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10                                 SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

유닉스 시그널과

Realtime 시그널을 구분하자.

Behavior (default)

Behavior	Description
Term	기본 행동이 프로세스 종료로 설정 됨
Ign	기본 행동이 시그널이 무시됨
Core	기본 행동이 “프로세스 종료 + 코어 덤프”
Stop	기본 행동이 프로세스의 정지
NoCatch	시그널 처리를 설치할 수 없는 시그널 (기본 행동으로만 작동함)
NoIgn	무시할 수 없는 시그널 (블록킹 불가능 = 시그널 매스크 설치 불가능)

* 붉은색 표시는 꼭 알아두자.

Signal List

Signal (No)	Behavior	Description
SIGHUP	Term	**Hang up,** 프로세스 재설정 요청 **= Session**이 끊긴 경우, 하위 프로세스를 모두 종료시킬 때 사용한다. 예를 들어 **nohup** 명령이 **SIGHUP**를 무시**(Ign bebavior)**하는 기능이다.
SIGINT	Term	**Interrupt,** 키보드로 프로세스 중단 요청 **(CTRL-C)**
SIGQUIT	Core	**Quit,** 키보드로 프로세스 중단+코어 덤프 요청 **(CTRL-□)**
SIGILL	Core	**Illegal,** 잘못된 명령
SIGABRT	Core	**Abort,** 강제 코어 덤프 **(**스택 추적시 사용**)**
SIGFPE	Core	**Float processing error,** 부동 소수점 계산 에러
SIGKILL	Term NoCatch NoIgn	**Kill,** 강제 종료 **(**프로세스 자원을 **OS**가 강제 회수한다.**)**
SIGUSR1	Term	**User defined 1,** 사용자 정의 시그널 **1**
SIGUSR2	Term	**User defined 2,** 사용자 정의 시그널 **2**

* 붉은색 표시는 꼭 알아두자.

Signal List (con't)

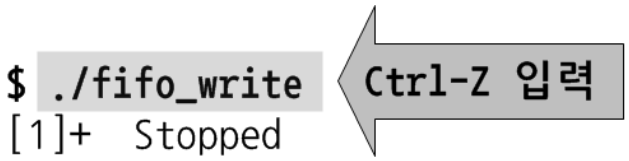
Signal (No)	Behavior	Description
SIGSEGV	Core	**Segment violation,** 메모리 세그먼트 침범
SIGPIPE	Term	**Pipe error,** 파이프 단절
SIGALRM	Term	**Alarm,** 알람 설정
SIGTERM	Term	**Termination,** 프로세스 종료 요청
SIGCHLD	Ign	**Child's event,** 자식 프로세스의 종료/정지
SIGCONT	-	**Continue,** 중단된(Stopped) 프로세스 재개
SIGSTOP	Stop NoCatch NoIgn	프로세스 중단 요청
SIGTSTP	Stop	**Temporary Stop (CTRL-Z)**
SIGTTIN	Stop	**Background**에서 **Control terminal** 읽기
SIGTTOU	Stop	**Background**에서 **Control terminal** 쓰기

* 붉은색 표시는 꼭 알아두자.

Signal List (addition : SUSv2, SUSv3)

Signal (No)	Behavior	Description
SIGBUS	Term	Bus error
SIGPOLL	Term	Pollable event (from SysV UNIX)
SIGPROF	Core	Profiling timeout
SIGSYS	Core	Invalid parameter (system call)
SIGTRAP	Core	Trace/Breakpoint trap
SIGURG	Ign	Urgent (TCP OOB)
SIGVTALRM	Term	Virtual Alarm (4.2 BSD)
SIGXCPU	Core	Exceed CPU time (4.2 BSD)
SIGXFSZ	Core	Exceed File size (4.2 BSD)

SIGTSTP




```
$ ./fifo_write
[1]+  Stopped                 ./fifo_write

$ jobs
[1]+  Stopped                 ./fifo_write

$ kill -SIGCONT %1 ; jobs
[1]+  Running                 ./fifo_write &
```

* <CTRL-Z>는 쉘에 의해 SIGTSTP를 발생시킨다.

Signal : Generated from the terminal

- 키보드 입력으로부터 발생하는 시그널을 기억해두자.
  - CTRL-C : SIGINT
  - CTRL-\ : SIGQUIT
  - CTRL-Z : SIGTSTP
- 각 시그널들이 하는 일은 무엇인가?

kill

- Signal을 Process에 전달

```
int kill(pid_t pid, int sig);
```

  - pid

Positive	지정한 PID를 가지는 프로세스에게 시그널 전달
0	동일 프로세스 그룹내 모든 프로세스에게 전달
-1	모든 프로세스에게 시그널 전달 (init제외, 접근 권한이 허락하는 경우만 전달)
<-1	프로세스 그룹에 시그널 전달 (시그널 전파용)

  - sig : 양수(시그널), 0(시그널 사용하지 않음, 에러 검사만...)

✓ POSIX에서 kill()은 죽이는게 아니라 시그널을 전송하는 기능이다.

✓ 간혹 kill명령이나 kill()함수를 죽이는 기능으로 알고 있다면 다시 공부하자.

signal : func.

sigaction	시그널 핸들러를 설치한다. 기존 시그널 핸들러의 백업.
sigemptyset	시그널 세트(sigset_t)를 모두 비운다.
sigfillset	시그널 세트를 모두 채운다.
sigaddset	시그널 세트에 특정 시그널 번호를 추가한다.
sigdelset	시그널 세트에서 특정 시그널 번호를 제거한다.
sigismember	시그널 세트에 해당 시그널 번호가 채워져 있는지 확인.
sigpending	지연된 시그널이 있는지 확인한다.
sigprocmask	시그널 블록 매스크를 조작한다.(시그널 블록 매스크의 읽기 및 저장)
sigsuspend	임시로 시그널 매스크를 대체하고 시그널 수신 대기.
strsignal **psignal** **psiginfo**	시그널 번호 및 **siginfo_t** 구조체로부터 시그널 정보를 알려준다.

signal() vs sigaction()

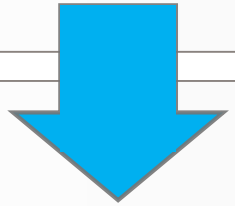
- signal()
  - varies across UNIX versions.
  - old fashioned
- sigaction()
  - from IEEE std 1003.1 standard
  - instead of signal()
    * provide common semantic to various UNIX system.

*** signal() 함수는 왠만하면 지양하자.**

signal() vs sigaction()

```
void chk_sigusr(int signum)
{
    …/* handler */;
}
signal(SIGUSR1, chk_sigusr);
```

옛날 방식이므로, 지양하자.



```
void chk_sigusr(int signum)
{
    …/* handler */;
}
struct sigaction sa_usr1;
memset(&sa_usr1, 0, sizeof(struct sigaction));
sa_usr1.sa_handler = chk_sigusr;
sigfillset(&sa_usr1.sa_mask);
sigaction(SIGUSR1, &sa_usr1, NULL);
```

이렇게 **sigaction**을 쓰는 방식으로 고쳐보자.

sigaction

```
int  sigaction(int  sig,
        const struct sigaction  *restrict act,
        struct sigaction  *restrict oact);
```

- sig : signal no.
- act, oact : new sigaction, old sigaction(backup)
- struct sigaction

시그널 조작 구조체를 살펴보자

```
struct  sigaction {
    void  (*sa_handler)(int);
    void  (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t  sa_mask;
    int  sa_flags;
}
```

sigaction.sa_handler

- .sa_handler
  - func. ptr : signal handler function.
    - void (*)(int)
  - SIG_DFL : default action.
  - SIG_IGN : ignore the signal.

sigaction.sa_mask

- .sa_mask
  - sigset_t 타입
  - signal handler가 실행하는 동안 대체할 블록 매스크
    - 블록 매스크에 지정된 시그널은 블록되어 시그널 핸들러 이후에 실행된다. 즉 지연되어 처리된다.(=pending signal 효과와 같다)
  - 조작
    - sigfillset, sigemptyset
    - sigaddset, sigdelset

sigaction.sa_flag

Flag	Description
SA_RESTART	시스템 콜 함수가 시그널 핸들러로 인해 중단된 경우 시그널 핸들러 종료 후 자동으로 재시작 **(no EINTR)** 그러나 타임 아웃이 존재하는 시스템 콜 함수는 여전히 **EINTR**을 발생시킨다.
SA_NOCLDSTOP	**SIGCHLD** 시그널 핸들러에 지정 자식 프로세스의 정지**(STOP)**에 대해 핸들러를 작동시키지 않는다.
SA_ONESHOT SA_RESETHAND	시그널 처리기를 일회용으로 만든다. **(**일회용 처리 후 **SIG_DFL**로 자동 복귀**)**
SA_NOMASK SA_NODEFFER	시그널 핸들러가 같은 시그널의 중복 수신을 허용한다. **(= SA_NOMASK)**
SA_SIGINFO	시그널 핸들러가 추가적인 시그널 정보를 저장하도록 지시. **(**주로 리얼타임 확장에서 사용함. 리얼타임 확장 부분 참조**)** 이 경우에는 핸들러 구조체의 **sa_sigaction** 멤버를 사용하며**, siginfo_t**를 주로 사용하여 정보를 얻어온다.
SA_ONSTACK	시그널 핸들러가 **sigaltstack**으로 대체된 스택 공간을 사용하도록 한다.

Signal : obsolete func.

- signal, sighold, sigignore, sigpause, sigrelse, sigset
  - 구식 함수 :
    - 차후 SUS 표준에서 삭제될 가능성이 있다.
    - c.f. gettimeofday, gethostbyname ...
    - 구식 함수에 대한 내용은 표준의 Rationale 부분을 살펴보자.

sig_basic.c (1/2)

```
void sa_handler_usr(int signum);
int  main() {
    struct sigaction sa_usr1;
    struct sigaction sa_usr2;
    memset(&sa_usr1, 0, sizeof(struct sigaction));
    sa_usr1.sa_handler = sa_handler_usr; /* 시그널 핸들러 함수 설정 */
    sigfillset(&sa_usr1.sa_mask); /* 시그널 블록 매스크를 모두 채운다. */

    memset(&sa_usr2, 0, sizeof(struct sigaction));
    sa_usr2.sa_handler = sa_handler_usr; /* 시그널 핸들러 함수 설정 */
    sigemptyset(&sa_usr2.sa_mask);       /* 시그널 블록 매스크를 모두 비운다. */

    sigaction(SIGUSR1, &sa_usr1, NULL); /* SIGUSR1 핸들러를 설치한다. */
    sigaction(SIGUSR2, &sa_usr2, NULL); /* SIGUSR2 핸들러를 설치한다. */
```

sig_basic.c (2/2)

```
    for(;;) {
        pause(); /* 시그널을 받을 때까지 블록된다. */
        printf("[MAIN] Recv SIGNAL...\n");
    }
    return EXIT_SUCCESS;
}
void sa_handler_usr(int signum) {
    int i;
    for (i=0; i<10; i++) {
        printf("\tSignal(%s):%d sec\n",
            signum == SIGUSR1 ? "USR1":"USR2",i);
        sleep(1);
    }
}
```

원래는 printf 계열 함수는
signal handler에서 사용하면
좋지 않다.
write() 함수로 교체하자!

run : sig_basic

```
$ ./sig_basic
[MAIN] SIGNAL-Handler installed, pid(37450)
         Signal(USR1):0 sec.
         Signal(USR1):1 sec.
         Signal(USR1):2 sec.
         Signal(USR1):3 sec.
         Signal(USR1):4 sec.
         Signal(USR1):5 sec.
         Signal(USR1):6 sec.
...생략...
```

첫 번째 터미널
sig_basic을 실행시켜놓자.

```
$ kill -USR1 $(pgrep -f ./sig_basic)
$ kill -USR1 $(pgrep -f ./sig_basic)
$ kill -USR1 $(pgrep -f ./sig_basic)
$ kill -USR1 $(pgrep -f ./sig_basic)
$ kill -USR1 $(pgrep -f ./sig_basic)
$
```

kill로 USR1 시그널을 5번 보냈다.

run : sig_basic

```
$ ./sig_basic
[MAIN] SIGNAL-Handler installed, pid(37450)
        Signal(USR1):0 sec.
        Signal(USR1):1 sec.
        Signal(USR1):2 sec.
        Signal(USR1):3 sec.
        Signal(USR1):4 sec.
        Signal(USR1):5 sec.
        Signal(USR1):6 sec.
        Signal(USR1):7 sec.
        Signal(USR1):8 sec.
        Signal(USR1):9 sec.
        Signal(USR1):0 sec.
        Signal(USR1):1 sec.
        Signal(USR1):2 sec.
        Signal(USR1):3 sec.
        Signal(USR1):4 sec.
        Signal(USR1):5 sec.
        Signal(USR1):6 sec.
        Signal(USR1):7 sec.
```

시그널을 5번 보냈어도
실제로는 2번만 실행된다.
왜 그럴까?

Tip!

- fork와 signal handler
  - signal handler는 child에게 상속된다. (also sigmask)
    - * 그러나 blocked signal은 상속되지 않는다.
  - child의 signal handler는 parent에서 설정해주는 것이 좋다.
    - why?

SA_NODEFER flag

```
void sa_handler_usr(int signum);
int  main() {
    struct sigaction sa_usr1;
    struct sigaction sa_usr2;
    memset(&sa_usr1, 0, sizeof(struct sigaction));
    sa_usr1.sa_handler = sa_handler_usr;
    sigfillset(&sa_usr1.sa_mask);

    memset(&sa_usr2, 0, sizeof(struct sigaction));
    sa_usr2.sa_handler = sa_handler_usr;
    sigemptyset(&sa_usr2.sa_mask);
    sa_usr2.sa_flags = SA_NODEFER;  /* SIGUSR2에 SA_NODEFER 플래그 적용 */

    sigaction(SIGUSR1, &sa_usr1, NULL); /* SIGUSR1 핸들러를 설치한다. */
    sigaction(SIGUSR2, &sa_usr2, NULL); /* SIGUSR2 핸들러를 설치한다. */
```

SA_SIGINFO

- siginfo_t 타입 사용 : 코드 9.3

```
void sa_sigaction_usr(int signum, siginfo_t *si, void *sv);
...생략...
sa_usr1.sa_sigaction = sa_sigaction_usr;
sa_usr1.sa_flags = SA_SIGINFO;
...생략...
void sa_sigaction_usr(int signum, siginfo_t *si, void *sv) {
    int i;
    printf("\t (signo:%d) (UID:%d) (PID:%d)\n",
        si->si_signo, si->si_uid, si->si_pid);
    for (i=0; i<10; i++) {
        printf("\tSignal(%s):%d sec.\n",
            signum == SIGUSR1 ? "USR1":"USR2", i);
        sleep(1);
    }
}
```

Signal & EINTR

EINTR

ERRORS

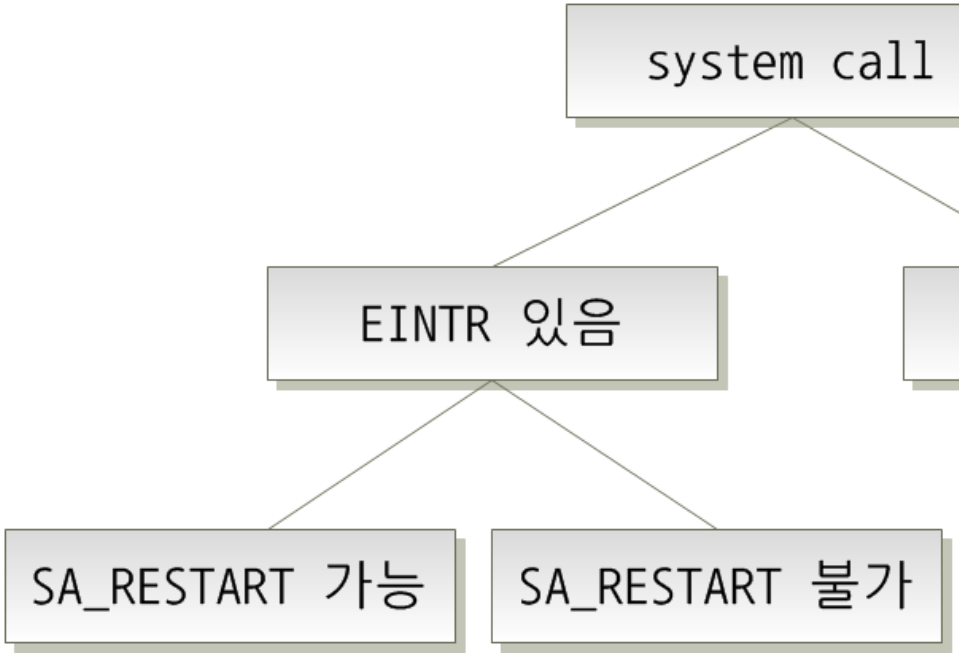
EAGAIN The file descriptor fd refers to a file other than a socket and has been marked nonblocking (O_NONBLOCK), and the write would block.

EBADF fd is not a valid file descriptor or is not open for writing.

EINTR The call was interrupted by a signal before any data was written; see signal(7).

man 페이지의 EINTR 에러에 대한 설명이 있는 경우에는 이에 대한 처리가 필요하다.

EINTR: func.



open(2) 처럼
대상에 따라 EINTR이
발생할 수도 있는 경우가 있다.

SA_RESTART

SA_RESTART 설정시 자동 재시작 함수	I/O 관련	read, readv, write, writev, ioctl open(pipe를 열기 위해 블록된 경우만
	프로세스	wait, waitpid
	소켓	accept, connect, recv, recvfrom, recvmsg, send, sendto, sendmsg
	파일 락	flock, fcntl(F_SETLKW 사용시)
	세마포어	sem_wait, sem_timedwait
SA_RESTART 설정 무시하고 EINTR로 리턴하는 함수	소켓	accept, connect, recv, recvfrom, recvmsg, send, sendto, sendmsg (**setsockopt로 소켓 타임아웃을 설정한 경우만 해당**)
	시그널	pause, sigsuspend, sigtimedwait, sigwaitinfo
	멀티플렉싱	epoll_wait, epoll_pwait, poll, ppoll, select, pselect
	슬립	clock_nanosleep, nanosleep, usleep, sleep (시그널을 받을 경우 무조건 성공으로 리턴. EINTR 아님)

* **EINTR이 있는 함수의 특징들은 무엇인가?**

Linux man페이지 signal(7)을 참조

Temp. Inf. Loop

```
for (; ; ) { /* 무한 루프에 문제가 있다. */
    if ((ret_recv = recv(fd, buf, SZ_BUF_RECV, 0)) == -1) {
        switch(errno) {       /* error */
            case EINTR:
            case EAGAIN:
                continue;    /* 재시도 */
            default:
                /* 그 외의 에러 처리 */
                break;
        }
    } else {
        break; /* 성공: for 루프를 빠져나간다. */
    }
} /* loop: for */
```

이런 코드는 극혐이다!
= 잠정적인 무한 루프를 만들 수 있는 코드이다.

Practice

- 기본 핸들러의 다음 의미는?
  - Term, Ign, Core, Stop, NoCatch, NoIgn
- signal() 함수와 sigaction() 함수 중 obsolete func.은?
- EINTR이 시그널 핸들러에 미치는 영향은?