시그널 보내기[2]

시그널 보내기: raise(2)

```
#include <signal.h>

int raise(int sig);
```

- 함수를 호출한 프로세스에 시그널 발송

시그널 보내기: abort(3) 종료 시그널 보내기

```
#include <stdlib.h>

void abort(void);
```

- 함수를 호출한 프로세스에 SIGABRT시그널 발송
- SIGABRT 시그널은 프로세스를 비정상적으로 종료시키고 코어덤프 생성

시그널 핸들러 함수[2]

시그널 핸들러 지정: sigset(3)

```
#include <signal.h>

                        시그널 넘버          시그널을 받았을 때, 처리할 함수
void (*sigset(int sig, void (*disp)(int)))(int);
```

- disp : sig로 지정한 시그널을 받았을 때 처리할 방법
  - 시그널 핸들러 함수의 주소값
  - SIG_IGN : 시그널을 무시하도록 지정
  - SIG_DFL : 기본 처리 방법으로 처리하도록 지정

SUN OS 에서 해당하는 이야기. LINUX 에서는 아님.

- sigset함수는 signal함수와 달리 시그널 핸들러가 한 번 호출된 후에 기본동작으로 재설정하지 않고, 시그널 핸들러를 자동으로 재정한다.

[예제 7-4] sigset 함수 사용하기(test1.c)

ex7_4.c

```
...
07  void handler(int signo) {
08      printf("Signal Handler Signal Number : %d\n", signo);
09      psignal(signo, "Received Signal");
10  }
11
12  int main(void) {
13      if (sigset(SIGINT, handler) == SIG_ERR) {
14          perror("sigset");
15          exit(1);
16      }
17
18      printf("Wait 1st Ctrl+C... : SIGINT\n");
19      pause();
20  printf("After 1st Signal Handler\n");
21  printf("Wait 2nd Ctrl+C... : SIGINT\n");
22      pause();
23  printf("After 2nd Signal Handler\n");
24
25      return 0;
26  }
```

시그널 핸들러를 재지정 하지 않아도됨

여기서는 상수(integer) 가 리턴됨.
실제로는 핸들러 구조체를 리턴.

리눅스에서는 sigset() 함수의 리턴값은 sighandler_t 임.

인터럽트 시그널을 받음.

```
# ex7_4.out
Wait 1st Ctrl+C... : SIGINT
^CSignal Handler Signal Number: 2
Received Signal: Interrupt
After 1st Signal Handler
Wait 2nd Ctrl+C... : SIGINT
^CSignal Handler Signal Number: 2
Received Signal: Interrupt
After 2nd Signal Handler
```

시그널 집합

시그널 집합의 개념

거의 모든 시그널 관련 함수가 시그널 집합을 인자로 받게끔 정의되어 있음.

- 시그널을 개별적으로 처리하지 않고 복수의 시그널을 처리하기 위해 도입한 개념
- POSIX에서 도입

시그널 집합의 처리를 위한 구조체

- sigset_t

```
typedef struct {          시그널 집합을 위한 구조체
        unsigned int __sigbits[4];
} sigset_t;
```

이 비트 값을 통해, SUN OS 에서는 어떤 시그널이 발생했는지 알 수 있음.

마스크

- 시그널을 비트 ~~마스트~~로 표현. 각 비트가 특정 시그널과 1:1로 연결
- 비트값이 1이면 해당 시그널이 설정된 것이고, 0이면 시그널 설정 안된 것임

시그널 집합 처리 함수[1]

□ 시그널 집합 비우기 : sigemptyset(3)

```
#include <signal.h>

int sigemptyset(sigset_t *set);
```

- 시그널 집합에서 모든 시그널을 0으로 설정

□ 시그널 집합에 모든 시그널 설정: sigfillset(3)

```
#include <signal.h>

                                시그널의 비트마스크
int sigfillset(sigset_t *set);
```

- 시그널 집합에서 모든 시그널을 1로 설정

□ 시그널 집합에 시그널 설정 추가: sigaddset(3)

```
#include <signal.h>  기본적으로 sigemptyset() 함수와 같이 쓰임.
                     0으로 설정 후, 추가하는 방법.

int sigaddset(sigset_t *set, int signo);
```

- signo로 지정한 시그널을 시그널 집합에 추가

시그널 집합 처리 함수[2]

시그널 집합에서 시그널 설정 삭제: sigdelset(3)

```
#include <signal.h>
int sigdelset(sigset_t *set, int signo);
```

기본적으로 sigfillset() 함수와 같이 쓰임.
1로 설정 후, 삭제하는 방법

- signo로 지정한 시그널을 시그널 집합에서 삭제

시그널 집합에 설정된 시그널 확인: sigismember(3)

```
#include <signal.h>

int sigismember(sigset_t *set, int signo);
```

- signo로 지정한 시그널이 시그널 집합에 포함되어 있는지 확인

[예제 7-5] 시그널 집합 처리 함수 사용하기(test2.c)

ex7_5.c

```
01  #include <signal.h>
02  #include <stdio.h>
03
04  int main(void) {
05      sigset_t st;
06
07      sigemptyset(&st);
08
09      sigaddset(&st, SIGINT);
10      sigaddset(&st, SIGQUIT);
11
12      if (sigismember(&st, SIGINT))
13          printf("SIGINT is setting.\n");
14
15      printf("** Bit Pattern: %x\n",st.__sigbits[0]);
16
17      return 0;
18  }
```

시그널 집합 비우기

시그널 추가

시그널 설정 확인

SUN OS의 경우, LINUX는 다른 이름의 변수를 사용.

6은 2진수로 00000110이므로 오른쪽에서 2번, 3번 비트가 1로 설정
SIGINT는 2번, SIGQUIT는 3번 시그널

```
# ex7_5.out
SIGINT is setting.
** Bit Pattern: 6
```

sigaction 함수의 활용[1]

sigaction 함수

- signal이나 sigset 함수처럼 시그널을 받았을 때 이를 처리하는 함수 지정
- signal, sigset 함수보다 다양하게 시그널 제어 가능

sigaction 구조체

```
struct sigaction {
    int sa_flags;          시그널을 어떻게 처리할 것인가
    union {
        void (*sa_handler)();          시그널 핸들러
        void (*sa_sigaction)(int, siginfo_t *, void *);          시그널을 받은 프로세스 정보 등을 알 수 있음.
    } _funcptr;          좀 더 정교하게 처리할 수 있는 시그널 핸들러 처리 함수
    sigset_t sa_mask;
};          시그널 핸들러가 수행을 마치고 블록된 시그널이 다시 pop 되어 처리됨.
```

- sa_flags : 시그널 전달 방법을 수정할 플래그(다음 쪽 참조)
- sa_handler/sa_sigaction : 시그널 처리를 위한 동작 지정
  - sa_flags에 SA_SIGINFO가 설정되어 있지 않으면 sa_handler에 시그널 처리동작 지정
  - sa_flags에 SA_SIGINFO가 설정되어 있으면 sa_sigaction 멤버 사용
- sa_mask : 시그널 핸들러가 수행되는 동안 블록될 시그널을 지정한 시그널 집합

sigaction 함수의 활용[2]

sa_flags에 지정할 수 있는 값(sys/signal.h) 이것은 외워야 함.

이 두개는 특히!

플래그	설명
SA_ONSTACK (0x00000001)	이 값을 설정하고 시그널을 받으면 시그널을 처리할 프로세스에 sigaltstack 시스템 호출로 생성한 대체 시그널 스택이 있는 경우에만 대체 스택에서 시그널을 처리한다. 그렇지 않으면 시그널은 일반 스택에서 처리된다.
SA_RESETHAND (0x00000002)	이 값을 설정하고 시그널을 받으면 시그널의 기본 처리 방법은 SIG_DFL로 재설정되고 시그널이 처리되는 동안 시그널을 블록하지 않는다.
SA_NODEFER (0x00000010)	이 값을 설정하고 시그널을 받으면 시그널이 처리되는 동안 유닉스 커널에서 해당 시그널을 자동으로 블록하지 못한다.
SA_RESTART (0x00000004)	이 값을 설정하고 시그널을 받으면 시스템은 시그널 핸들러에 의해 중지된 기능을 재시작하게 한다.
SA_SIGINFO (0x00000008)	이 값이 설정되지 않은 상태에서 시그널을 받으면 시그널 번호(sig 인자)만 시그널 핸들러로 전달된다. 만약 이 값을 설정하고 시그널을 받으면 시그널 번호 외에 추가 인자 두 개가 시그널 핸들러로 전달된다. 두 번째 인자가 NULL이 아니면 시그널이 발생한 이유가 저장된 siginfo_t 구조체를 가리킨다. 세 번째 인자는 시그널이 전달될 때 시그널을 받는 프로세스의 상태를 나타내는 ucontext_t 구조체를 가리킨다.
SA_NOCLDWAIT (0x00010000)	이 값이 설정되어 있고 시그널이 SIGCHLD면 시스템은 자식 프로세스가 종료될 때 좀비 프로세스를 만들지 않는다.
SA_NOCLDSTOP (0x00020000)	이 값이 설정되어 있고 시그널이 SIGCHLD면 자식 프로세스가 중지 또는 재시작할 때 부모 프로세스에 SIGCHLD 시그널을 전달하지 않는다.

sigaction 함수의 활용[3]

sigaction 함수

```
#include <signal.h>

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

시그널을 처리할 방법의 구조체 주소

기존의 방법

- sig : 처리할 시그널
- act : 시그널을 처리할 방법을 지정한 구조체 주소
- oact : 기존에 시그널을 처리하던 방법을 저장할 구조체 주소

- 첫번째 인자로 SIGKILL과 SIGSTOP을 제외한 어떤 시그널도 올 수 있음
- 성공하면 0, 실패시 -1

[예제 7-6] sigaction 함수 사용하기(1) test3.c

ex7_6.c

```
...
07  void handler(int signo) {
08      psignal(signo, "Received Signal:");
09      sleep(5);
10      printf("In Signal Handler, After Sleep\n");
11  }
12
13  int main(void) {
14      struct sigaction act;
15
16      sigemptyset(&act.sa_mask);
17      sigaddset(&act.sa_mask, SIGQUIT);
18      act.sa_flags = 0;
19      act.sa_handler = handler;
20      if (sigaction(SIGINT, &act, (struct sigaction *)NULL) < 0) {
21          perror("sigaction");
22          exit(1);
23      }
24
25      fprintf(stderr, "Input SIGINT: ");
26      pause();
27      fprintf(stderr, "After Signal Handler\n");
28
29      return 0;
30  }
```

컨트롤 + C 를 눌러 SIGINT 를 받고, 5초 내로 컨트롤 + \ 를 눌러 SIGQUIT 을 받으면

슬립을 하는 동안에 SIGQUIT 이 블록되어 있음.

함수가 끝까지 다 수행되지 않았기 때문!

컨트롤 + C 를 눌러 SIGINT 를 받고, 5초 내로 컨트롤 + C ...

연속적으로 컨트롤 + C 를 누르면 스택에서 한 번 만을 시그널로 받아들이는 듯.

0 로 초기화

sa_mask 초기화

sigaddset 을 하지 않으면, 핸들러가 각자 수행된다.

SIGQUIT 시그널을 블록시키기 위해 추가

별도의 플래그 없음.

시그널핸들러 지정

시그널 받기 위해 대기(pause함수)

컨트롤 + C 로 인터럽트 시그널을 발생시키면 handler 함수 수행.

```
# ex7_6.out
Input SIGINT: ^CReceived Signal:: Interrupt
^\In Signal Handler, After Sleep
끝(Quit)(코어덤프)
```

SIGQUIT 을 받기 위해서는 컨트롤 + \
SIGINT 를 받기 위해서는 컨트롤 + C

SUN OS 에서는 SIGINT 를 했는데 코어덤프가 발생.

[예제 7-7] sigaction 함수 사용하기(SA_RESETHAND) test4.c

test3 과 거의 다 같음.

```
...
07  void handler(int signo) {
08      psignal(signo, "Received Signal:");
09      sleep(5);
10      printf("In Signal Handler, After Sleep\n");
11  }
12
13  int main(void) {
14      struct sigaction act;
15
16      sigemptyset(&act.sa_mask);
17      sigaddset(&act.sa_mask, SIGQUIT);
18      act.sa_flags = SA_RESETHAND;
19      act.sa_handler = handler;
20      if (sigaction(SIGINT, &act, (struct sigaction *)NULL) < 0) {
21          perror("sigaction");
22          exit(1);
23      }
24
25      fprintf(stderr, "Input SIGINT: ");
26      pause();
27      fprintf(stderr, "After Signal Handler\n");
28
29      return 0;
30  }
```

컨트롤 + C 를 누르고 5초 내로 컨트롤 + C 를 누르면, 두 번째 시그널을 받아들여서 시그널 핸들러를 처리하는 것이 아니라 프로세스를 종료시킴. (기본적인 설정으로 바뀌어서)

시그널 핸들러가 한번 호출된 후에 시그널 처리 방법이 기본처리 방법으로 재설정

SA_RESETHAND 지정

이 부분만 다름.

```
# ex7_7.out
Input SIGINT: ^CReceived Signal:: Interrupt
^CIn Signal Handler, After Sleep
#
```

sigaction 함수의 활용[4]

- sa_flags에 SA_SIGINFO 플래그를 지정하면 시그널 발생원인을 알 수 있다.
  - 시그널 핸들러의 형식

```
void handler (int sig, siginfo_t *sip, ucontext_t *ucp);
```

  - sip : 시그널이 발생한 원인을 담은 siginfo_t 구조체 포인터
  - ucp : 시그널을 받는 프로세스의 내부상태를 나타내는 구조체 포인터

  - siginfo_t 구조체

```
typedef struct {
    int si_signo;
    int si_errno;
    int si_code;
    union sigval si_value;
    union {
        …
    }__data;
} siginfo_t;
```

si_signo : 시그널 번호
si_errno : 0 또는 오류번호
si_code : 시그널 발생 원인 코드
__data : 시그널의 종류에 따라 값 저장

sigaction 함수의 활용[5]

시그널 발생 원인 코드

[표 7-8] 사용자 프로세스가 시그널을 생성했을 때 si_code 값

코드	값	의미
SI_USER	0	kill(2), sigsend(2), raise(3), abort(3) 함수가 시그널을 보냄
SI_LWP	-1	_lwp_kill(2) 함수가 시그널을 보냄
SI_QUEUE	-2	sigqueue(3) 함수가 시그널을 보냄
SI_TIMER	-3	timer_settime(3) 함수가 생성한 타이머가 만료되어 시그널을 보냄
SI_ASYNCIO	-4	비동기 입출력 요청이 완료되어 시그널을 보냄
SI_MESGQ	-5	빈 메시지 큐에 메시지가 도착했음을 알리는 시그널을 보냄

시그널 발생 원인 출력: psiginfo(3)

```
#include <siginfo.h>
void psiginfo(siginfo_t *pinfo, char *s);
```

- **pinfo : 시그널 발생원인 정보를 저장한 구조체, s: 출력할 문자열**

[예제 7-8] 시그널 발생원인 검색하기 – test5,c

ex7_8.c

```
...
08  void handler(int signo, siginfo_t *sf, ucontext_t *uc) {
09      psiginfo(sf, "Received Signal:");
10      printf("si_code : %d\n°±, sf->si_code);
11  }
12
13  int main(void) {
14    struct sigaction act;
15
16    act.sa_flags = SA_SIGINFO;
17    act.sa_sigaction = (void (*)(int, siginfo_t *, void *))handler;
18    sigemptyset(&act.sa_mask);
19    if (sigaction(SIGUSR1, &act, (struct sigaction *)NULL) < 0) {
20        perror("sigaction");
21        exit(1);
22    }
23
24    pause();
25
26    return 0;
27  }
```

오류 메시지 출력

SA_SIGINFO 플래그 설정

모두 0으로 만듬. 별도의 시그널을 설정하지 않음.

sigaction 함수 설정

유저가 정의한 시그널
SIGUSR1, SIGUSR2

./test5 &: 백그라운드에서 프로세스 실행.

kill 명령어나 함수는 시그널을 다른 프로세스에게 보낼 때도 사용.

```
# ex7_8.out&
[1]     2515
# kill -USR1 2515
# Received Signal: : User Signal 1 ( from process  1579 )
si_code : 0
```

백그라운드 프로세스를 사용한 이유는 이 프로세스의 ID 를 알기 위해

SIGUSR1 시그널 보내기

2515 프로세스에게 시그널 보냄.

원인은 0. kill 이라는 명령어가 시그널을 보냈음으로

background process: 이 프로세스가 끝나지 않은 상태에서 다음 프로세스를 처리할 수 있음.
$ test & (백그라운드에서 수행되는 프로세스)
$
$
ex) Daemon 프로그램 (백그라운드 프로세스로 실행됨.)
foreground process: 우리가 지금까지 배운 방식. 현재 프로세스가 끝나야 다음 프로세스를 실행시킬 수 있음.
$ test
$ test

foreground 로 다시 돌려놓으려면
$ fg

background 프로세스를 종료하려면
컨트롤 + z
$ ps -ef | grep s16010980
$ kill -9 프로세스 넘버

foreground 를 background 로 바꾸고싶다면,
컨트롤 + z
$ bg