제 7 장
시그널 및 그 처리

ACS30021
고급 프로그래밍

나보균 (bkna@kpu.ac.kr)

컴퓨터 공학부
한국산업기술 대학교

학습 목표

- Signal의 개요
- Signal의 생성
  - ✓ kill(), raise(), alarm()
- Signal의 전달 및 블록
  - ✓ 시그널 블록킹 – Signal set과 signal masking
  - ✓ 시그널에 의한 상호배제(mutal exclusion)
- Signal의 처리
  - ✓ 무시, 기본 동작, 지정 동작 수행
  - ✓ sigaction ()
- Signal 기다림
  - ✓ Busy-waiting
  - ✓ Blocked-waiting
- 비동기 시그널 안전성과 프로그래밍
  - ✓ 시그널 + ioctl ()
  - ✓ Async. I/O

Co-operating Processes

- *Independent* process can not affect or be affected by the execution of another process.

- *Co-operating* process can affect or be affected by the execution of another process
- Advantages of process co-operation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Co-operating Processes

- process example

```
/* fork example */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int             glob = 6;
char            buf[] = "a write to stdout\n";

int main(void)
{
  int  var;
  pid_t pid;

  var = 88;
  write(STDOUT_FILENO, buf, sizeof(buf));
  printf("before fork\n");

  if ((pid = fork()) == 0) {            /* child */
    glob++; var++;
  } else
    sleep(2);                           /* parent */

  printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
  exit (0);
}
```

(Source : Adv. programming in the UNIX Env., pgm 8.1)

Process: naturally independent

Co-operating Processes

- Co-operating processes
  - ✓ producer and consumer
  - ✓ shared resource
  - ✓ parallel matrix processing
  - ✓ mobile computing

```
while (1) {   // Producer
     /* produce an item in next produced */
        while (counter == BUFFER_SIZE)  ;     /* do nothing */
        buffer[in] = nextProduced;
        in = (in+1) % BUFFER_SIZE;
        counter ++;
}
```

what's the potential problem?

```
while (1) {   // Consumer
     while (counter == 0)   ;     /* do nothing */
        nextComsumed = buffer[out];
        out = (out+1) % BUFFER_SIZE;
        counter --;
}
```

Co-operating Processes

- Interface for *Cooperating* processes at the same sites
  - ✓ signal
  - ✓ synchronization
  - ✓ pipe, FIFO
  - ✓ System V IPC (InterProcess Communication)
    - ➢ message passing
    - ➢ shared memory
  - ✓ STREAM (socket)
- Interface for *Cooperating* processes at the different sites
  - ✓ socket
  - ✓ RPC
  - ✓ RMI, DCOM, CORBA, **… Hadoop**

시그널의 개념

- 시그널
  - 프로세스에 뭔가 발생했음을 알리는 비동기적 메시지
  - software interrupts
  - Facility for handling exceptional conditions
  - Provides process with a way of handling asynchronous events
  - The kernel may send signals to process or processes may send each other signals with kill( ) system call
- 발생사유
  - 0으로 나누기처럼 프로그램에서 예외적인 상황이 일어나는 경우
  - kill 함수처럼 시그널을 보낼 수 있는 함수를 사용해서 다른 프로세스에 시그널을 보내는 경우
  - 사용자가 Ctrl+C와 같이 인터럽트 키를 입력한 경우

Signal

- 시그널의 종류
  - SIGKILL, SIGINT, SIGBUS, SIGUSR1, …. (최대 64)
  - The *interrupt* signal, SIGINT, is used to stop a command before that command completes (usually produced by ^C).
  - signals for terminating processes
  - signals triggered by a particular physical circumstance
  - signals available for use by the programmer
  - signal generated when a pipe is closed
  - suspending or resuming
- 시그널 생성
  - event 발생 요인에 의해 발생
  - kill(), raise(), alarm()
- 시그널 통지 (Signal notification)
  - sigsend()
  - kill(), raise(), alarm() 등도 통지

Signal

- 시그널 처리방법
  - ✓ 각 시그널에 지정된 기본 동작 수행.
    - ➢ 프로세스 종료/무시/일시중지/재시작
  - ✓ 시그널 무시 (ignore)
    - ➢ 프로세스가 시그널을 전달받았지만 폐기
    - ➢ sigignore ()
  - ✓ 시그널 처리를 위한 함수(시그널 핸들러)를 지정해놓고 시그널을 받으면 해당 함수 호출
  - ✓ 시그널이 발생하지 않도록 블록 처리
    - ➢ 시그널 블록이 해제되면 전달 됨
  - ✓ sigaction (), signal (), sigset () 에 의해 시그널핸들러 설정
  - ✓ sigaction() – 시그널 처리 및 처리과정 제어
  - ✓ 태스크 종료(exit, abort)
  - ✓ 태스크 수행 중지 (stop)
- 기타 시그널 관련 함수
  - ✓ 시그널 블록이나 해제 – sighold(), sigrelse(), sigpromask()
  - ✓ 시그널

시그널의 일생

- 발생 (generated) – 발생 사건의 주체가 생성
- 전달 (delivered) – 목표 프로세스가 시그널에 대응한 행위를 취함
- 펜딩 (pending) – 시그널이 생성되었으나 목표 프로세스에게 아직 전달되지 못한 시점.
- 수용 (catch) – 목표 프로세스에 시그널처리함수 (signal handler)가 실행 ⇔ 무시 (ignore)
- sigaction 함수의 인자가 SIG_DFL이나 SIG_IGN 이면 시그널 무시로 시그널 전달로 볼 수 없음
- 블록 (block) – 프로세스 시그널 마스크에 의해 목표 프로세스에게 시그널이 전달 되지 않음
  - ✓ 펜딩 상태인 시그널이 블록 되었다가 다시 언블록 되면 전달됨

Signal 처리 기본 구조

- 시그널 처리 함수 구현, 시그널 번호 할당, 함수 등록

```
void sig_handler (int signo)
{
  signal (SIGUSR1, sig_handler);          /* reinstall */
  printf("received signal %d\n", signo);  /* handle the signal */
            …..
}

int main ()
{
  signal (SIGUSR1, sig_handler);          /* install the handler */
            ….
  for ( ; ; )
    pause();
}
```

Signal

```
/* signal example */
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

void timeout (int signo)
{
    printf (“Timeout (5 seconds), Abort\n”);
    exit(1);
}

int main ()
{
    int n; char buf[80];
    signal (SIGALRM, timeout);

    while (1) {
                printf (“Enter input (timeout 5 sec.): \n”);
                alarm (5);
                if ((n = read (STDIN_FILENO, buf, 80)) > 0) {
                   alarm (0);
                   // Processing input...
                }
    }
}
```

Signal

- 시그널 관련 시스템 호출

System Call	Description
`kill( )`	Send a signal to a process.
`sigaction( )`	Change the action associated with a signal.
`signal( )`	Similar to `sigaction( )`.
`sigpending( )`	Check whether there are pending signals.
`sigprocmask( )`	Modify the set of blocked signals.
`sigsuspend( )`	Wait for a signal.
`rt_sigaction( )`	Change the action associated with a real-time signal.
`rt_sigpending( )`	Check whether there are pending real-time signals.
`rt_sigprocmask( )`	Modify the set of blocked real-time signals.
`rt_sigqueueinfo( )`	Send a real-time signal to a process.
`rt_sigsuspend( )`	Wait for a real-time signal.
`rt_sigtimedwait( )`	Similar to `rt_sigsuspend( )`.

Signal

- 시그널의 예
  - SIGINT – DEL or Ctrl-C
  - SIGILL
  - SIGTERM
- Signals are used to notify a process of a particular event
  - simple method for transmitting software interrupts to UNIX processes
- 시그널 이름
  - <signal.h>에 정의
- 정상 종료 와 비정상 종료
  - 대부분의 시그널은 그 시그널이 수신되면 정상종료 (normal termination) 발생
  - SIGABRT, SIGBUS, SIGSEGV, SIGQUIT, SIGILL, SIGTRAP, SIGSYS, SIGXCPU, SIGXFSZ, SIGFPE 시그널은 비정상 종료 (abnormal termination)을 발생시킴
  - abort() – SIGABRT 발생, core dump → 디버깅에 사용

시그널의 종류

시그널	번호	기본 처리	발생 요건
SIGHUP	1	종료	행업으로 터미널과 연결이 끊어졌을 때 발생
SIGINT	2	종료	인터럽트로 사용자가 Ctrl+C를 입력하면 발생
SIGQUIT	3	코어 덤프	종료 신호로 사용자가 Ctrl+\를 입력하면 발생
SIGILL	4	코어 덤프	잘못된 명령 사용
SIGTRAP	5	코어 덤프	추적(trace)이나 중단점(breakpoint)에서 트랩 발생
SIGABRT	6	코어 덤프	abort 함수에 의해 발생
SIGEMT	7	코어 덤프	에뮬레이터 트랩으로 하드웨어에 문제가 있을 경우 발생
SIGFPE	8	코어 덤프	산술 연산 오류로 발생
SIGKILL	9	종료	강제 종료로 발생
SIGBUS	10	코어 덤프	버스 오류로 발생
SIGSEGV	11	코어 덤프	세그먼테이션 폴트로 발생
SIGSYS	12	코어 덤프	잘못된 시스템 호출로 발생
SIGPIPE	13	종료	잘못된 파이프 처리로 발생
SIGALRM	14	종료	알람에 의해 발생
SIGTERM	15	종료	소프트웨어적 종료로 발생
SIGUSR1	16	종료	사용자 정의 시그널 1

이외에도 시그널의 종류는 다양함 (표 7-7 참조)

Signal의 생성

- 터미널에서 특수키를 누르는 경우
- 하드웨어의 오류
  - ✓ 0으로 나눈 경우, 잘못된 메모리를 참조하는 경우 등
- kill 함수의 호출
  - ✓ 특정 프로세스나 프로세스 그룹에게 원하는 시그널을 발생/전달한다.
  - ✓ 대상 프로세스에 대한 권한이 있어야 한다.
- kill 명령의 실행
  - ✓ 내부적으로 kill 함수를 호출한다.
- 소프트웨어적 조건
  - ✓ 네트워크에서의 데이터 오류 (SIGURG), 파이프 작업에서의 오류 (SIGPIPE), 알람 시계의 종료 (SIGALRM) 등

시그널 발생 함수

- kill () – 특정 프로세스에 시그널 발생
- wait ()
- alarm () – 일정 시간 후 시그널 발생
- raise () – 자기 자신의 프로세스에게 시그널 발생

시그널 발생: kill(2)

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

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

- pid가 0보단 큰 수 : pid로 **지정한 프로세스에** 시그널 발송
- pid가 0 : 특별한 프로세스를 제외하고 프로세스 그룹ID가 시그널을 보내는 프로세스의 프로세스 **그룹ID**와 같은 모든 프로세스에게 시그널 발송
- pid가 -1 : 시그널을 보낸는 프로세스의 유효 사용자ID가 root가 아니면, 특별한 프로세스를 제외하고 프로세스의 실제 사용자ID가 시그널을 보내는 프로세스의 **유효 사용자ID**와 같은 모든 프로세스에 시그널 발송
- pid가 -1이 아닌 음수 : 프로세스 그룹ID가 pid의 절대값인 프로세스 그룹에 속하고 시그널을 보낼 권한을 가지고 있는 모든 프로세스에 시그널 발송

kill 함수 사용하기

```
01  #include <sys/types.h>
02  #include <unistd.h>
03  #include <signal.h>
04  #include <stdio.h>
05
06  int main(void) {
07      printf("Before SIGCONT Signal to parent.\n");
08
09      kill(getppid(), SIGCONT);
10
11      printf("Before SIGQUIT Signal to me.\n");
12
13      kill(getpid(), SIGQUIT);
14
15      printf("After SIGQUIT Signal.\n");
16
17      return 0;
18  }
```

SIGQUIT의 기본동작은 코어덤프

```
# ex7_1.out
Before SIGCONT Signal to parent.
Before SIGQUIT Signal to me.
끝(Quit)(코어 덤프)
```

시그널 발생

- raise(2)

```
#include <signal.h>
int raise(int sig);
```

  - ✓ 함수를 호출한 프로세스에 시그널 발송

- abort(3)

```
#include <stdlib.h>
void abort(void);
```

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

Signal의 전달(delivery)

- 시그널의 전달
  - ✓ 발생된 시그널이 수신되어 정해진 방법대로 처리되는 것
- 시그널의 지연 (pending)
  - ✓ 발생된 시그널이 전달되지 못한 상태
- 시그널의 블록 (block)
  - ✓ 블록이 해제되거나 무시하도록 변경될 때가지 시그널 전달이 지연된 상태로 존재
- 시그널의 무시 (ignore)
  - ✓ 시그널이 전달되었으나 접수한 프로세스가 시그널을 폐기하는 것

시그널 핸들러 함수[1]

- 시그널 핸들러
  - ✓ 시그널을 받았을 때 이를 처리하기 위해 지정된 함수
  - ✓ 프로세스를 종료하기 전에 처리할 것이 있거나, 특정 시그널에 대해 종료하고 싶지 않을 경우 지정
  - ✓ signal (), sigset ()은 단순히 시그널 핸들러만 지정
  - ✓ sigaction ()는 시그널핸들러 지정과 플래그 설정을 통해 시그널 처리과정을 제어 및 핸들러 수행 중 다른 시그널을 블록 가능
- 시그널 핸들러 지정: signal(3)

```
#include <signal.h>

void (*signal(int sig, void (*disp)(int)))(int);
```

  - ✓ sig: SIGKILL, SIGSTOP 제외한 시그널 지정
  - ✓ disp: sig로 지정한 시그널을 받았을 때 처리할 방법
    - ➢ 시그널 핸들러 함수명
    - ➢ SIG_IGN : 시그널을 무시하도록 지정
    - ➢ SIG_DFL : 기본 처리 방법으로 처리하도록 지정
  - ✓ signal함수는 시그널이 들어올 때마다 시그널 핸들러를 호출하려면 매번 시그널 핸들러를 **재지정**해야함.

signal 함수 사용하기

```
...
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      void (*hand)(int);
14
15      hand = signal(SIGINT, handler);
16      if (hand == SIG_ERR) {
17          perror("signal");
18          exit(1);
19      }
20
21      printf("Wait 1st Ctrl+C... : SIGINT\n");
22      pause();
23      printf("After 1st Signal Handler\n");
24      printf("Wait 2nd Ctrl+C... : SIGINT\n");
25      pause();
26      printf("After 2nd Signal Handler\n");27
28      return 0;
29  }
```

```
# ex7_2.out
Wait 1st Ctrl+C... : SIGINT
^CSignal Handler Signal Number : 2
Received Signal: Interrupt
After 1st Signal Handler
Wait 2nd Ctrl+C... : SIGINT
^C#
```

두번째 Ctrl+C는 처리못함

시그널 핸들러 재지정하기

```
...
07  void handler(int signo) {
08       void (*hand)(int);
09       hand = signal(SIGINT, handler);
10       if (hand == SIG_ERR) {
11            perror("signal");
12            exit(1);
13       }
14
15       printf("Signal Handler Signal Number: %d\n", signo);
16       psignal(signo, "Received Signal");
17  }
...
```

시그널 핸들러 재지정

두번째 Ctrl+C도 처리

```
# ex7_3.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 핸들러 함수

- 시그널 핸들러 지정: sigset(3)

```
#include <signal.h>
void (*sigset(int sig, void (*disp)(int)))(int);
```

  - ✓ 복수의 시그널을 처리 가능
  - ✓ sig: SIGKILL, SIGSTOP 제외한 시그널 지정
  - ✓ disp : sig로 지정한 시그널을 받았을 때 처리할 방법
    - ➢ 시그널 핸들러 함수명
    - ➢ SIG_IGN : 시그널을 무시하도록 지정
    - ➢ SIG_DFL : 기본 처리 방법으로 처리하도록 지정
  - ✓ sigset함수는 signal함수와 달리 시그널 핸들러가 한 번 호출된 후에 기본동작으로 재설정하지 않고, 시그널 핸들러를 **자동으로 재지정**

sigset 함수 사용하기

```
...
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 재지정하지 않아도 됨

```
# 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
  - ✓ 시그널을 개별적으로 처리하지 않고 **복수의 시그널을 처리** 가능
  - ✓ POSIX에서 도입
- 시그널 집합의 처리를 위한 구조체
  - ✓ sigset_t

```
typedef struct {
   unsigned int __sigbits[4];
} sigset_t;
```

  - ✓ 시그널을 비트 마스크로 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>
int sigfillset(sigset_t *set);
```

  - ✓ 시그널 집합에서 모든 시그널을 1로 설정

- 시그널 집합에 시그널 설정 추가: sigaddset(3)

```
#include <signal.h>
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);
```

  - ✓ 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

Signal set 관련 함수

- 시그널 집합 (signal set)
  - ✓ 설정된 시그널들의 리스트를 지정
- 시그널 집합 관련 함수

```
#include <signal.h>

/* 초기화 */
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);

/* 조작 */
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);

/* 검사 */
int sigismember (const sigset_t *set, int signo);
```

signal Set의 사용 예

```
#include <signal.h>

sigset_t mask1, mask2;
.
.
.

/* 빈 집합을 생성한다. */
sigemptyset (&mask1);

/* 시그널을 추가한다. */
sigaddset (&mask1, SIGINT);
sigaddset (&mask1, SIGQUIT);

/* 완전히 차있는 집합을 생성한다. */
sigfillset (&mask2);

/* 시그널을 제거한다 */
sigdelset (&mask2, SIGCHLD);
...
```

시그널 집합 처리 함수 사용하기

```
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

6은 2진수로 000001100이므로 오른쪽에서 2번, 3번 비트가 1로 설정
SIGINT는 2번, SIGQUIT는 3번 시그널

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

시그널 마스크 (signal mask)

```
#include <signal.h>
int sigprocmask (int how, const sigset_t *set, sigset_t *oset);
```

- signal set을 사용해 한번에 여러 시그널 블록/해제
- 시그널 마스크 함수
  - ✓ 특정 시그널에 대한 동작 상태(설정/해제)를 수정
  - ✓ 시그널 집합 블록과 해제
    - ➢ 예외: SIGSTOP, SIGKILL 블록 불가
- signal blocking:
  - ✓ signals will not be handled until the process has completed its delicate operations
- pthread 사용 시 sigprocmask () 대신 pthread_sigmask ()

sigprocmask() 매개변수

- **how:** sigprocmask의 특정 행동을 지정
  - ✓ **SIG_SETMASK:** block out the signals in the second parameter **signal set** (모든 시그널을 **"blocked"** 상태로 설정)
  - ✓ **SIG_UNBLOCK:** remove the signal block
  - ✓ **SIG_BLOCK:** add the signals in **set** to the current signal set (추가되는 시그널집합만 "blocked" 상태로 설정)
- set :
  - ✓ 블록하거나 해제할 시그널 집합 주소
- oset :
  - ✓ NULL 또는 이전 설정값을 저장한 시그널 집합주소
  - ✓ 설정 전의 시그널 세트를 저장하는 용도

sigprocmask 함수 사용하기

```
...
05  int main(void) {
06      sigset_t new;
07
08      sigemptyset(&new);
09      sigaddset(&new, SIGINT);
10      sigaddset(&new, SIGQUIT);
11      sigprocmask(SIG_BLOCK, &new, (sigset_t *)NULL);
12
13      printf("Blocking Signals : SIGINT, SIGQUIT\n");
14      printf("Send SIGQUIT\n");
15      kill(getpid(), SIGQUIT);
16
17      printf("UnBlocking Signals\n");
18      sigprocmask(SIG_UNBLOCK, &new, (sigset_t *)NULL);
19
20      return 0;
21  }
```

시그널 집합에 SIGINT, SIGQUIT 설정

시그널 집합 블록설정

SIGQUIT 시그널 보내기

시그널 집합 블록 해제

```
# ex7_12.out
Blocking Signals : SIGINT, SIGQUIT
Send SIGQUIT
UnBlocking Signals
끝(Quit)(코어 덤프)
```

블록해제 후 시그널을 받아 종료

Signal 처리: sigaction ()

- sigaction 함수
  - ✓ signal이나 sigset 처럼 시그널을 받았을 때 이를 처리하는 함수 지정
    - ➢ 시그널 처리 과정을 제어 가능
  - ✓ signal, sigset 함수보다 다양하게 시그널 제어 가능
- sigaction 구조체

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

  - ✓ sa_flags : 시그널 전달 방법을 수정할 플래그
  - ✓ 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)

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.

Signal 처리의 가능한 경우

- 시그널을 받은 프로세스의 행동
- default action (SIG_DFL)
  - ✓ 특별한 처리 방법을 선택하지 않은 경우
  - ✓ 대부분 시그널의 기본 처리 방법은 프로세스를 종료 시킴
- ignore the signal (SIG_IGN)
  - ✓ SIGKILL과 SIGSTOP 시그널을 제외한 모든 시그널을 무시할 수 있다.
  - ✓ 하드웨어 오류에 의해 발생한 시그널에 대해서는 주의해야 한다.
- catch the signal
  - ✓ 시그널이 발생하면 미리 등록된 함수(signal handler)가 수행
  - ✓ SIGKILL and SIGSTOP 은 전달과 무시 불가(cannot be caught or ignored)

signal ():
- **ISO C 표준 함수**
- 단일 쓰레드 프로그램에서조차 신뢰 불가

❖ 그러므로 시그널 핸들러를 등록하기 위해서는 반드시 **sigaction ()** 사용!

시그널 처리 과정

```
static void handler (int signo) {
   ...
}

static void func (int .... )                /* 시그널의 전달 여부 결정 */
{
   ...
   sigemptyset ( );
   sigaddset ()
   sigprocmask ()
}

int main ()
{
     struct sigaction act;
     ...
     act.sa_handler = handler;              /* 시그널 처리 함수 결정 */
     sigaction (.. , &act, NULL);           /* 시그널 수용 설정 */
     ...
     func ( );                              /* 시그널 전달 조건 설정 */
}
```

SIGINT의 포착(1)

```
/* sigex -- sigaction이 어떻게 동작하는지 보인다 */

#include <signal.h>

int main()
{
  static struct sigaction act;

  /* catchint를 선언한다. 후에 핸들러로 사용된다. */
  void catchint (int);

  /* SIGINT를 수신했을 때 취해질 행동을 지정한다. */
  act.sa_handler = catchint;

  /* 완전히 찬 마스크를 하나 생성한다. */
  sigfillset(&(act.sa_mask));

  /* sigaction 호출전에는, SIGINT가 프로세스를 종료시킨다(디폴트 행동) */
  sigaction(SIGINT, &act, NULL);
```

SIGINT의 포착(2)

```
  /* SIGINT를 수신하면 제어가 catchint로 전달될 것이다 */

  printf ("sleep call #1\n");
   sleep (1);
   printf ("sleep call #2\n");
   sleep (1);
   printf ("sleep call #3\n");
   sleep (1);
   printf ("sleep call #4\n");
   sleep (1);

  printf ("Exiting\n");
   exit (0);
}

/* SIGINT를 처리하는 간단한 함수 */
void catchint (int signo)
{
  printf ("\nCATCHINT: signo=%d\n", signo);

  printf("CATCHINT: returning\n\n");
}
```

SIGINT의 포착(3)

- 실행 예

(Ctrl-C 를 누르지 않았을 때)

```
% sigex
sleep call #1
sleep call #2
sleep call #3
sleep call #4
Exiting
```

- 실행 예

(Ctrl-C 를 눌렀을 때)

```
% sigex
sleep call #1
<interrupt>

CATCHING: signo=2
CATCHINT: returning

sleep call #2
sleep call #3
sleep call #4
Exiting
```

SIGINT의 무시

- SIGINT 의 무시

```
act.sa_handler = SIG_IGN
```

- 인터럽트 키의 enable

```
act.sa_handler = SIG_DFL;
sigaction(SIGINT, &act, NULL);
```

- 여러 개의 시그널을 동시에 무시

```
act.sa_handler = SIG_IGN;
sigaction(SIGINT, &act, NULL);
sigaction(SIGQUIT, &act, NULL);
```

이전의 행동을 복원하기

```
#include <signal.h>

static struct sigaction act, oact;

/* SIGTERM을 위한 과거의 행동을 남겨둔다. */
sigaction(SIGTERM, NULL, &oact);

/* SIGTERM을 위한 새로운 행동을 지정한다. */
act.sa_handler = SIG_IGN;
sigaction(SIGTERM, &act, NULL);

/* 여기서 무언가 작업을 수행한다 ... */

/* 이제 과거의 행동을 복원한다. */
sigaction(SIGTERM, &oact, NULL);
```

sigaction 함수 사용하기(1)

```
...
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  }
```

sa_mask 초기화

SIGQUIT 시그널을 블록시키기 위해 추가

시그널핸들러 지정

시그널 받기 위해 대기(pause함수)

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

sigaction 함수 사용하기(SA_RESETHAND)

```
...
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  }
```

시그널 핸들러가 한번 호출된 후에 시그널 처리방법이 기본처리방법으로 재설정

SA_RESETHAND 지정

```
# 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

시그널 발생원인 검색하기

```
...
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

sigaction 함수 설정

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

SIGUSR1 시그널 보내기

시그널 생성 – kill ()

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

- 다른 프로세스에게 signal을 전달 가능
- `pid` 값을 갖는 프로세스에 시그널 `sig`를 보냄
  - ✓ 예: `kill(7421, SIGTERM);`
- pid 값에 따른 차이
  - ✓ 0 : 같은 프로세스 그룹에 속하는 모든 프로세스에 시그널이 전송
  - ✓ -1 (euid가 root가 아닐 때): 보내는 프로세스의 euid와 같은 real uid를 갖는 모든 프로세스에 시그널이 전송
  - ✓ -1 (euid가 root일 때): 특수한 시스템 프로세스를 제외한 모든 프로세스에 전송
  - ✓ <0 (not -1): pgid가 pid의 절대값과 같은 모든 프로세스에 전송

시그널 생성 – kill ()

```
/* synchro  -- kill의 예 */

#include <unistd.h>
#include <signal.h>

int ntimes = 0;

main()
{
 pid_t pid, ppid;
 void p_action (int), c_action (int);
 static struct sigaction pact, cact;

 /* 부모를 위해 SIGUSR1 행동을 지정 */
 pact.sa_handler = p_action;
 sigaction (SIGUSR1, &pact, NULL);

 switch (pid = fork()){
  case -1:                /* 오류 */
     perror ("synchro");
     exit (1);
  case 0:                 /* 자식 */
```

```
     /* 자식을 위해 행동을 지정 */
     cact.sa_handler = c_action;
     sigaction (SIGUSR1, &cact, NULL);

     /* 부모의 프로세스 식별번호를 얻음. */
     ppid = getppid();

     for (;;) {
       sleep (1);
       kill (ppid, SIGUSR1);
       pause();
     }
     /* 결코 퇴장(exit) 않음. */
  default:                /*  부모 */
       for(;;) {
          pause();
          sleep (1);
          kill (pid, SIGUSR1);
       }
       /* 결코 퇴장(exit) 않음 */
  }
}
```

시그널 생성 – kill ()

```
void p_action (int sig)
{
  printf ("Parent caught signal
   #%d\n", ++ntimes);
}

void c_action (int sig)
{
  printf ("Child caught signal
   #%d\n", ++ntimes);
}
```

❖ 실행 예:

```
% synchro
Parent caught signal #1
Child caught signal #1
Parent caught signal #2
Child caught signal #2
<interrupt>
%
```

시그널 생성 – raise ()와 alarm ()

- 자신에게 signal 전달

- `raise:` 수행 프로세스에게 시그널을 보냄

```
#include <signal.h>
int raise(int sig);
```

알람 시그널

- 알람 시그널
  - ✓ 일정한 시간이 지난 후에 자동으로 시그널이 발생하도록 하는 시그널
  - ✓ 일정 시간 후에 한 번 발생시키거나, 일정 간격을 두고 주기적으로 발송 가능

- 알람 시그널 생성: alarm(2)

```
#include <unistd.h>

unsigned int alarm(unsigned int sec);
```

  - ✓ sec : 알람이 발생시킬 때까지 남은 시간(초 단위)
  - ✓ 일정 시간이 지나면 SIGALRM 시그널 발생
  - ✓ 프로세스별로 알람시계가 하나 밖에 없으므로 알람은 **하나만 설정 가능**

시그널 생성 – raise ()와 alarm ()

```
#include <stdio.h>
#include <signal.h>

#define TIMEOUT    5      /* 초단위 */
#define MAXTRIES   5
#define LINESIZE   100
#define CTRL_G   '\007' /* ASCII 벨 */
#define TRUE       1
#define FALSE      0

/* 타임아웃이 발생했는지 알아보기 위해 사용
   한다. */
static int timed_out;

/* 입력줄을 보관한다. */
static char answer[LINESIZE];

char *quickreply (char *prompt)
{
 void catch (int);
 int ntries;
 static struct sigaction act, oact;
```

```
    /* SIGALRM을 포착하고, 과거 행동을 저
       장한다. */
    act.sa_handler = catch;
    sigaction (SIGALRM, &act, &oact);

    for  (ntries=0;  ntries<MAXTRIES;
      ntries++) {
      timed_out = FALSE;
      printf ("\n%s > ", prompt);

      /* 알람 시계를 설정 */
      alarm (TIMEOUT);

      /* 입력줄을 가져온다. */
      gets (answer);

      /* 알람을 끈다. */
      alarm (0);

      /* timed_out이 참이면, 응답이 없는
      경우이다. */
      if (!timed_out)
        break;
    }
```

시그널 생성 – raise ()와 alarm ()

```
  /* 과거 행동을 복원한다. */
  sigaction (SIGALRM, &oact, NULL);

  /* 적절한 값을 복귀한다. */
  return (ntries == MAXTRIES ? ((char
   *)0) : answer);
}

/* SIGALRM을 받으면 수행한다. */
void catch (int sig)
{
     /* 타임 아웃 플래그를 설정한다. */
     timed_out = TRUE;

     /* 벨을 울린다. */
     putchar (CTRL_G);
}
```

alarm 함수 사용하기

```
01  #include <unistd.h>
02  #include <signal.h>
03  #include <siginfo.h>
04  #include <stdio.h>
05
06  void handler(int signo) {
07      psignal(signo, "Received Signal");
08  }
09
10  int main(void) {
11      sigset(SIGALRM, handler);
12
13      alarm(2);
14      printf("Wait...\n");
15      sleep(3);
16
17      return 0;
18  }
```

2초 설정

```
# ex7_9.out
Wait...
Received Signal: Alarm Clock
```

기타 시그널 처리 함수 – 시그널 대기

- Signal 기다림
  - ✓ Busy-waiting
  - ✓ Blocked-waiting

- Busy-waiting
  - ✓ CPU를 계속 사용 특정 사건 발생 여부 확인
  - ✓ 반복문 안에서 어떤 변수 값을 계속 확인

- Blocked-waiting
  - ✓ 기다리고 있는 사건이 발생하기까지 프로세스 block 상태 유지
  - ✓ 프로세스가 blocked된 동안 자원을 다른 프로세스가 사용
  - ✓ pause(), sigsuspend(), sigwait()

시그널 대기 – pause () 시스템 호출

- 시그널 대기 : pause(3)

```
#include <signal.h>
int pause(void);
```

  - ✓ 임의의 시그널이 도착할 때까지 수행이 일시 중단됨
    - ➢ 시그널 처리 함수를 호출
    - ➢ 프로세스 종료 시그널 발생

- 시그널 대기 : sigpause(3)

```
#include <signal.h>
int sigpause(int sig);
```

  - ✓ sig : 시그널이 올 때까지 대기할 시그널

시그널 대기 – pause () 시스템 호출

```
/* tml -- tell-me-later 프로그램 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#define TRUE       1
#define FALSE      0
#define BELLS      "\007\007\007"
   /*ASCII bells */

int alarm_flag = FALSE;

/* SIGALRM을 처리할 루틴 */
void setflag (int sig)
{
 alarm_flag = TRUE;
}

main (int argc, char **argv)
{
 int nsecs, j;
 pid_t pid;
 static struct sigaction act;
```

```
if ( argc<=2 ) {
   fprintf (stderr, "Usage:  tml #minutes
   message\n");
   exit (1);
 }
 if ((nsecs=atoi(argv[1])*60) <= 0) {
   fprintf  (stderr,  "  tml  :  invalid
   time\n");
   exit (2);
 }
 /* 백그라운드 프로세스를 생성하기 위해 fork한
   다. */
 switch (pid = fork()){
   case -1:                  /* 오류 */
         perror ("tml");
         exit (1);
   case 0:                   /* 자식 */
         break;
   default:                  /* 부모 */
         printf  ("tml  processid  %d\n",
   pid);
         exit (0);
 }
```

시그널 대기 – pause () 시스템 호출

```
/* 알람을 위한 행동을 지정한다. */
 act.sa_handler = setflag;
 sigaction (SIGALRM, &act, NULL);

 /* 얼람 시계를 켠다. */
 alarm (nsecs);

 /* 시그널이 올 때까지 중단(pause) ... */
 pause();

 /* 만일 시그널이 SIGALRM이면 메시지를 프
   린트하라. */
 if (alarm_flag == TRUE) {
   printf (BELLS);
   for (j = 2; j < argc; j++)
       printf ("%s", argv[j]);
   printf ("\n");
 }

 exit (0);
}
```

- 실행 예

```
% tml 10 time to go home
```

시그널 대기 – sigsuspend ()

- pause()의 사용상 문제점
  - ✓ pause() 사용 전에 시그널이 전달 가능
- sigsuspend ()
  - ✓ 동시에 시그널의 언블록과 pause () 호출 실행
  - ✓ 함수가 반환되기 전에 시그널 마스크를 함수 호출 전의 상태로 설정

```
#include <signal.h>
int sigsuspend (const sigset_t *sigmask);
```

suspend.c

```
#include <errno.h>
#include <signal.h>
#include <unistd.h>

static int isinitialized = 0;
static struct sigaction oact;
static int signum = 0;
static volatile sig_atomic_t sigreceived = 0;

/* ARGSUSED */
static void catcher (int signo)
{
    sigreceived = 1;
}

int initsuspend (int signo)
{
   struct sigaction act;
   if (isinitialized)
      return 0;
   act.sa_handler = catcher;
   act.sa_flags = 0;
```

```
if ((sigfillset(&act.sa_mask) == -1) ||
       (sigaction(signo, &act, &oact) == -1))
      return -1;
   signum = signo;
   isinitialized = 1;
   return 0;
}

int restore (void)
{
   if (!isinitialized)
      return 0;
   if (sigaction(signum, &oact, NULL) == -1)
      return -1;
   isinitialized = 0;
   return 0;
}

int simplesuspend (void)
{
   sigset_t maskblocked, maskold, maskunblocked;
```

suspend.c

```
  if (!isinitialized) {
     errno = EINVAL;
     return -1;
  }

  if ((sigprocmask (SIG_SETMASK, NULL, &maskblocked) == -1) ||
      (sigaddset (&maskblocked, signum) == -1) ||
      (sigprocmask (SIG_SETMASK, NULL, &maskunblocked) == -1) ||
      (sigdelset (&maskunblocked, signum) == -1) ||
      (sigprocmask (SIG_SETMASK, &maskblocked, &maskold) == -1))
      return -1;
  while (sigreceived == 0) sigsuspend (&maskunblocked);
  sigreceived = 0;
  return sigprocmask (SIG_SETMASK, &maskold, NULL);
}
```

suspendtest.c

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

int initsuspend (int signo);
int restore (void);
int simplesuspend (void);

int main(void)
{
   fprintf (stderr, "This is process %ld₩n",
                (long)getpid());
   for ( ; ; ) {
      if (initsuspend (SIGUSR1)) {
         perror("Failed to setup handler for SIGUSR1");
         return 1;
      }
      fprintf (stderr, "Waiting for signal₩n");
      if (simplesuspend ()) {
         perror ("Failed to suspend for signal");
         return 1;
      }
      fprintf (stderr, "Got signal₩n");
      if (restore ()) {
         perror ("Failed to restore original handler");
         return 1;
      }
   }
   return 1;
}
```

시그널 대기 – sigwait ()

```
#include <signal.h>
int sigwait (const sigset_t *sigmask, int *signo);
```

- *sigmask 설정 시그널 중 어느 한 시그널이 펜딩될 때까지 블록
- 첫째 매개변수인 시그널 세트에 포함된 시그널에 의해서만 sigwait()가 실행 완료
- signo 매개변수 값으로 그 시그널 번호를 반환
- 프로세스 시그널 마스크 변경 않음
- 시그널은 sigwait() 호출 전에 블록되어 있어야 함

시그널 대기 – sigwait ()

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

int main(void)
{
   int signalcount = 0;
   int signo;
   int signum = SIGUSR1;
   sigset_t sigset;

   if ((sigemptyset (&sigset) == -1) || (sigaddset (&sigset, signum) == -1) ||
       (sigprocmask (SIG_BLOCK, &sigset, NULL) == -1))
      perror ("Failed to block signals before sigwait");
   fprintf (stderr, "This process has ID %ld₩n", (long)getpid ());
   for ( ; ; ) {
      if (sigwait (&sigset, &signo) == -1) {
         perror ("Failed to wait using sigwait");
         return 1;
      }
      signalcount++;
      fprintf (stderr, "Number of signals so far: %d₩n", signalcount);
   }
}
```

기타 시그널 처리 함수[4]

- 시그널 보내기: sigsend(2)

```
#include <signal.h>
int sigsend(idtype_t idtype, id_t id, int sig);
```

  - ✓ idtype : id에 지정한 값의 종류
  - ✓ id : 시그널을 받을 프로세스나 프로세스 그룹
  - ✓ sig : 보내려는 시그널

값	의미
P_PID	프로세스 ID가 id인 프로세스에 시그널을 보낸다.
P_PGID	프로세스 그룹 ID가 id인 모든 프로세스에 시그널을 보낸다.
P_SID	세션 ID가 id인 모든 프로세스에 시그널을 보낸다.
P_TASKID	태스크 ID가 id인 모든 프로세스에 시그널을 보낸다.
P_UID	유효 사용자 ID(EUID)가 id인 모든 프로세스에 시그널을 보낸다.
P_GID	유효 그룹 ID(EGID)가 id인 모든 프로세스에 시그널을 보낸다.
P_PROJID	프로젝트 ID가 id인 모든 프로세스에 시그널을 보낸다.
P_CID	스케줄러 클래스 ID가 id인 모든 프로세스에 시그널을 보낸다.
P_CTID	프로세스 콘트랙트 ID가 id인 모든 프로세스에 시그널을 보낸다.
P_ALL	id를 무시하고 모든 프로세스에 시그널을 보낸다.
P_MYID	함수를 호출하는 자신에게 시그널을 보낸다.

기타 시그널 처리 함수[5]

- 시그널 무시처리 : sigignore(3)

```
#include <signal.h>
int sigignore(int sig);
```

  - ✓ sig : 무시할 시그널 번호
  - ✓ 인자로 지정한 시그널의 처리방법을 SIG_IGN으로 설정

기타 시그널 처리 함수[2]

- 시그널 블록킹과 해제

```
#include <signal.h>

int sighold(int sig);
int sigrelse(int sig);
```

  - ✓ 인자로 받은 시그널을 시그널 마스크에 추가하거나 해제

시그널 블록함수 사용하기

```
...
07  void handler(int signo) {
08      char *s;
09
10      s = strsignal(signo);
11      printf("Received Signal : %s\n", s);
12  }
13
14  int main(void) {
15      if (sigset(SIGINT, handler) == SIG_ERR) {
16          perror("sigset");
17          exit(1);
18      }
19
20      sighold(SIGINT);
21
22      pause();
23
24      return 0;
25  }
```

시그널 이름 리턴

시그널 핸들러 설정

SIGINT 블록설정

SIGINT 시그널을 안받는다

```
# ex7_11.out
^C^C^C^C^C
```

실습:

- 본 슬라이드 40-41 쪽 프로그램을 수정
- 기존 CTRL-C 시그널에 대한 핸들러 함수 실행에 부가하여
- SIGALRM 에도 핸들러 함수 실행
- alarm ()를 사용 시그널 생성

상호배제 – 시그널에 의한 critical section 보호

- 예:

```
sigset_t set1;
…
/* 시그널 집합을 완전히 채운다. */
sigfillset(&set1);

/* 차단(block)을 설정한다. */
sigprocmask(SIG_SETMASK, &set1, NULL);


/* 극도로 중요한 코드를 수행: 상호배제 (mutex) 요구 영역 */


/* 시그널 차단을 제거한다 */
sigprocmask(SIG_UNBLOCK, &set1, NULL);
```

Signal 차단(blocking)

```
/* signal blocking -- sigprocmask의 사
   용예를 보인다. */

#include <signal.h>

main()
{
 sigset_t set1, set2;

 /* 시그널 집합을 완전히 채운다. */
 sigfillset (&set1);

 /* SIGINT와 SIGQUIT를 포함하지 않는 시그
   널 집합을 생성한다. */
 sigfillset (&set2);
 sigdelset (&set2, SIGINT);
 sigdelset (&set2, SIGQUIT);

 /* 중대하지 않은 코드를 수행 ... */
```

```
/* 차단을 설정 */
 sigprocmask(SIG_SETMASK, &set1, NULL);

 /* 극도로 중대한 코드를 수행 */

 /* 하나의 차단을 제거 */
 sigprocmask(SIG_UNBLOCK, &set2, NULL);

 /* 덜 중대한 코드를 수행 ... */
 /* (SIGINT와 SIGQUIT 만을 봉쇄) */

 /* 모든 시그널 차단을 제거 */
 sigprocmask(SIG_UNBLOCK, &set1, NULL);
}
```

시그널과 그 처리 – 크리티컬섹션

```
#include <math.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

static volatile sig_atomic_t doneflag = 0;

/* ARGSUSED */
static void setdoneflag(int signo)
{
   doneflag = 1;  // Critical section
}

int main (void)
{
   struct sigaction act;
   int count = 0;
   double sum = 0;
   double x;

   act.sa_handler = setdoneflag;
   act.sa_flags = 0;
   if ((sigemptyset(&act.sa_mask) == -1) ||
       (sigaction(SIGINT, &act, NULL) == -1)) {
```

```
   perror("Failed to set SIGINT handler");
      return 1;
   }
   while (!doneflag) {
      x = (rand() + 0.5)/(RAND_MAX + 1.0);
      sum += sin(x);
      count++;
      printf("Count is %d and average is %f\n",
                  count, sum/count);
   }

   printf("Program terminating ...\n");
   if (count == 0)
      printf("No values calculated yet\n");
   else
      printf("Count is %d and average is %f\n",
                  count, sum/count);
   return 0;
}
```

시그널과 그 처리

- main()안 while(!doneflag) 명령을 실행하기 전에 SIGINT 시그널이 발생하면?
- while (!doneflag) 반복문에서 doneflag의 값을 확인하는 중 SIGINT 시그널이 발생하면?

- volatile 지시자 – 컴파일러에게 변수가 프로그램 수행 중에 비동기적으로 변경될 수 있음을 알림

volatile을 선언한 변수를 사용하여 컴파일러의 최적화를 못하게 하는 역할:

```
static int foo;
void bar (void) {
   foo = 0;
   while (foo != 255);  // foo의 값의 초기값이 0 이후, 값이 변하지 않기 때문에 while의 조건은 항상 true
}
```

따라서 컴파일러는 다음과 같이 최적화:

```
void bar_optimized (void) {
   foo = 0;
   while (true);   // 이렇게 되면 while의 무한 루프에 빠지게 된다.
}
```

이런 최적화를 방지하기 위해 다음과 같이 volatile을 사용:

```
static volatile int foo;
void bar (void) {
   foo = 0;
   while (foo != 255);
}
```

- 개발자가 의도한 대로, 그리고 눈에 보이는 대로 기계어 코드가 생성
- 이 프로그램만으로는 무한루프라고 생각할 수 있지만, 만약 foo가 하드웨어 장치의 레지스터라면 하드웨어에 의해 값이 변경 가능
- 하드웨어 값을 폴링(poll)할 때 사용

시그널과 크리티컬 섹션

```
#include <errno.h>
#include <limits.h>
#include <math.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define BUFSIZE 100

static char buf[BUFSIZE];
static int buflen = 0;

/* ARGS USED */
static void handler(int signo)
{
   int savederrno;


   savederrno = errno;
   write(STDOUT_FILENO, buf, buflen);
   errno = savederrno;
}
```

```
static void results (int count, double sum) {
   double average;
   double calculated;
   double err;
   double errpercent;
   sigset_t oset;, sigset;

   if ((sigemptyset(&sigset) == -1) ||
       (sigaddset(&sigset, SIGUSR1) == -1) ||
       (sigprocmask (SIG_BLOCK, &sigset, &oset) == -1) )
      perror("Failed to block signal in results");
   if (count == 0)  snprintf(buf, BUFSIZE, "No values calc yet₩n");
   else {
      calculated = 1.0 - cos(1.0);
      average = sum/count;
      err = average - calculated;
      errpercent = 100.0 * err / calculated;
      snprintf (buf, BUFSIZE,
          "Count = %d, sum = %f, average = %f, error = %f or %f%%₩n",
           count, sum, average, err, errpercent);
   }
   buflen = strlen(buf);
   if (sigprocmask (SIG_SETMASK, &oset, NULL) == -1)
      perror("Failed to unblock signal in results");
}
```

시그널과 크리티컬 섹션

```
int main(void) {
   int count = 0;
   double sum = 0;
   double x;
   struct sigaction act;

   act.sa_handler = handler;
   act.sa_flags = 0;
   if ((sigemptyset (&act.sa_mask) == -1) ||
        (sigaction (SIGUSR1, &act, NULL) == -1) ) {
      perror("Failed to set SIGUSR1 signal handler");
      return 1;
   }
   fprintf (stderr, "Process %ld starting calculation₩n", (long) getpid ());
   for ( ; ; ) {
      if ((count % 10000) == 0)
         results (count, sum);
      x = (rand() + 0.5)/(RAND_MAX + 1.0);
      sum += sin(x);
      count++;
      if (count == INT_MAX)
         break;
   }
   results (count, sum);
   handler(0);        /* call handler directly to write out the results */
   return 0;
}
```

우아한 퇴장(a graceful exit)

- 프로그램 수행 중 임시로 만든 작업파일의 삭제

```
/* 프로그램으로부터 우아하게 퇴장(exit)한다. */
#include <stdio.h>
#include <stdlib.h>

void g_exit(int s){
 unlink ("tempfile"); // remove a file
 fprintf(stderr, "Interrupted-exiting\n"); // 비동기 시그널 안전성 문제 발생 가능
 exit (1);
}
. . .
extern void g_exit(int);
. . .
static struct sigaction act;
act.sa_handler = g_exit;
sigaction(SIGINT, &act, NULL);
```

비동기 시그널 안전성 (Async-signal safety)

- 시그널 핸들러 내부에서 fprintf()나 strlen()를 사용하지 않나?

```
void catch_ctrlc (int signo)
{
   char handmsg[] = "I found CTRL-C\n";
   int msglen = sizeof (handmsg);

   write (STDERR_FILENO, handmsg, msglen);
 }

int main ()
{
   . . .
   struct sigaction act;
   act.sa_handler = catch_ctrlc;  /* 새로운 핸들러 설정 */
   act.sa_flags = 0;                    /* 옵션을 설정 않음 */
   if ((sigemptyset (&act.sa_mask) == -1) || (sigaction (SIGINT, &act, NULL) == -1))
      perror ("Failed to set SIGINT  to handle CTRL-C");
   . . .
}
```

비동기 시그널 안전성

- 시그널 안전성은 main()와 시그널 핸들러 모두가 비동기 시그널 안전성을 갖지 않는 함수를 사용할 때 발생

- 시그널 핸들러 함수에서 주의

Signal과 시스템 호출

- 시스템 호출 수행 도중 시그널이 도착 하면
  - ✓ 대부분의 경우 시스템 호출이 종료할 때까지 아무런 영향을 미치지 않음
  - ✓ 인터럽트 되는 시스템 호출
    - ➢ 느린 장치에 대한 read, write, open
    - ➢ -1을 return, errno에 EINTR을 저장

```
if( write(tfd, buf, size) < 0) {
  if( errno == EINTR ) {
     warn("Write interrupted");
     . . .
  }
}
```

- UNIX signals cannot normally be stacked
  - ✓ 어느 한 순간에 한 프로세스에 대해 해결되지 않은 시그널은 각 유형별로 하나 밖에 없다

sigsetjmp와 siglongjmp에 의한 프로그램 제어

```
#include <setjmp.h>
/* 프로그램내의 위치를 저장한다. */
int sigsetjmp(sigjmp_buf env, int savemask);

/* 저장된 위치로 되돌아 간다. */
void siglongjmp(sigjmp_buf env, int val);
```

- sigsetjmp: 프로그램의 **스택환경을 저장**함으로써 현재 프로그램의 위치와 시그널 마스크를 저장.
  - ✓ return value of sigsetjmp
    - non zero (`val` of siglongjmp) – if sigsetjmp has been called from siglongjmp
    - 0 – if it is called as the next sequential instruction

- siglongjmp는 저장된 위치로 제어를 다시 전달
  - ✓ a kind of long-distance, non-local goto. never returns.

sigsetjmp와 siglongjmp

```
/* sigsetjmp와 siglongjmp의 사용 예 */
#include <sys/types.h>
#include <signal.h>
#include <setjmp.h>
#include <stdio.h>

sigjmp_buf position;

main()
{
  static struct sigaction act;
  void goback (void);
  ...
  /* 현재 위치를 저장한다. */
  if (sigsetjmp (position, 1) == 0) {
   act.sa_handler = goback;
   sigaction (SIGINT, &act, NULL);
  }

  domenu();
  ...
  .
}
```

```
void goback (void)
{
  fprintf (stderr, "\nInterrupted\n");

  /* 저장된 위치로 되돌아간다. */
  siglongjmp(position, 1);
}
```

비동기 I/O 프로그래밍

- read(), write()를 호출할 때 프로세스는 I/O가 완료될 때까지 블록

- 시그널 + ioctl ()
  - ✓ SIGPOLL, SIGIO를 사용

- Async. I/O
  - ✓ aio_read ()
  - ✓ aio_write ()
  - ✓ aio_return ()
  - ✓ aio_error ()
  - ✓ aio_cancel ()