chapter08. Signals

objectives

1. Learn the fundamentals of signal handling
2. Experiment with signals for control
3. Explore the POSIX signal facilities
4. Use signal masks and handlers
5. Understand async-signal safety

Signal

a s/w notification to a process of an event.

→ 임의의 process OR OS kernel에게 정보 전달

- the lifetime of a signal : the interval btw its generation and its delivery (to target process)

signal을 target에게 전달

- **pending signal** : generate 되었지만 delivered x → process에게 전달되지 못 한 signal
  - pending list 내에서 유지

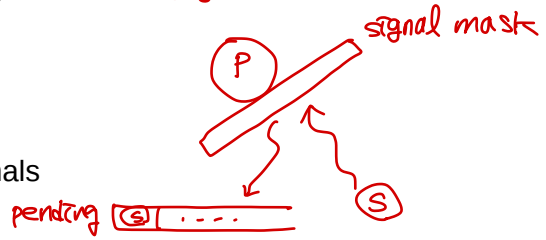
- process : signal이 delivered → signal handler가 실행 → signal 감지

default가 존재함

  - **signal handler**
    - sigaction function → user가 지정한 이름을 가진 signal handler 실행 !
      1. SIG_DFL : default signal handler
      2. SIG_IGN : ignore the signal → 아무 action도 X

handler 대신에 macro 호출

  - **signal mask** : contains a list of currently blocked signals
    - sigprocmask : signal mask control function
    - signal handler = SIG_IGN → program이 ignore하도록 signal mask setting



signal mask 변경 가능

- 모든 signal은 이름이 SIG로 시작

  ex) SIG USR1

Generating

signal 생성 function

```
//1.
//signal을 전송하는 명령어
#include <signal.h>
int kill(pid_t pid, int sig);

//pid : target process ID
// ① 0 -> caller's process group → group 내 모든 process
// ② -1 -> to all processes for which it has permission to send
// ③ 다른 음수 -> to process group with group ID
                                  (= |PID|)
                                  ↳ 있으면 다!
//sig : signal num → SIG 빼고 parameter야!

//return 0 -> successful
//return -1 > unsuccessful

//kill –s signal_name pid ..
//Kill –l [exit_status]
//Kill [-signal_name] pid..
//Kill [-signal_number] pid..
```

가능한 symbolic signal name

* signal num
  - 0 → signal 0
  - 1 → SIGHUP
  - 2 → SIGINT
  - 3 → SIGOUT
  - 6 → SIGABRT
  - 9 → SIGKILL
  - 14 → SIGALRM
  - 15 → SIGTERM

```
//2.
//signal을 자기 자신에게 보내는 함수
#include <signal.h>
int raise(int sig);

//return 0 -> successful
//return other -> unsuccessful
```

```
//3.
//특정 시간 후에 호출한 process에게 SIGALAM signal을 보냄
#include <unistd.h>
unsigned alarm(unsigned seconds);
                ↳ 알람이 설정되면 바로 return

//seconds : 지정한 시간
//         0 -> 이전 alarm request를 cancel
//default action : to terminate the process

//return (남은 초 수) -> successful
//return 0 -> 지정한 alarm이 없는 경우

//ex -> 10초 후에 program 종료
#include <unistd.h>
int main(void){
  alarm(10);
```

```
for(;;);
}
```

→ 알람이 뜰 때까지 (0초 동안 대기) 후 terminate

- **signal sets** : signal set(signal이 여러 개인 data structure)


```
#include <signal.h>

①int sigaddset(sigset_t* set, int signo);
②int sigdelset(sigset_t* set, int signo);
③int sigemptyset(sigset_t* set);
④int sigfillset(sigset_t* set);
  //return 0 -> successful
  //return -1 -> unsuccessful

⑤int sigismember(const sigset_t* set, int signo);
  //return 1 -> member인 경우
  //return 0 -> 나머지
```

add/del 하고자 하는 signal num

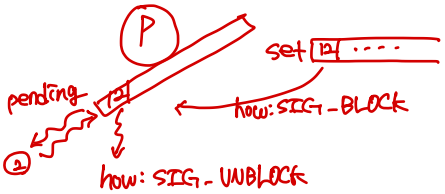
초기화 함수 ⇒ 다 비움

초기화 함수 ⇒ 등록 가능한 것으로 다 채움.

Signal masks

signal mask 내용 변경 → 일시적인 의도가 많음

⇒ 원래의 signal mask 상태로 돌아갈 수 있어야 함 → output parameter에 저장 (oset)



- signal mask → single thread에서만 사용해야 함
  - pthread_sigmask() → multi thread ~~(ch.12)~~
- SIGSTOP, SIGKILL signal은 signal mask로 block 불가

```
#include <signal.h>

//현재 block되어 있는 signal들을 return
int sigprocmask(int how, const sigset_t *restrict set, sigset_t *restrict oset);

//how : signalmask를 어떻게 수정할건지 정함
//     1) SIG_BLOCK : add 'set' signals
//     2) SIG_UNBLOCK : delete 'set' signals
//     3) SIG_SETMASK : 'set' signals만 block 다시 add

//set : 수정하고자 하는 signal set
//oset : Not NULL -> 현재 저장되어 있는 signal set output parameter
//return 0 -> successful
//return -1 -> unsuccessful
```

```
//ex1 -> SIGINT signal을 signal mask에 추가

sigset_t newsigset;
if((sigemptyset(&newsigset) == -1 || (sigaddset(&newsigset, SIGINT) == -1))
  perror("Failed to initialize the signal set");
else if( sigprocmask(SIG_BLOCK, &newsigset, NULL) == -1)
```

① 초기화 ② SIGINT 추가 ③ signal mask에 추가

```
  perror("Failed to block SIGINT");

//ex2 -> modify -> restore
sigset_t blockmask;
sigset_t oldmask;
… //add signals to blockmask
① modify → blockmask만
if( sigprocmask(SIG_SETMASK, &blockmask, &oldmask) == -1)
  return -1;
…  ② restore → 원상복구
if( sigprocmask(SIG_SETMASK, &oldmask, NULL) == -1)
  return -1;
…
```

Catching and ignoring

```
#include <signal.h>

//signal을 받았을 때 수행할 action을 등록하는 함수
int sigaction(int sig, const struct sigaction *restrict act,
              struct sigaction *restrict oact);

//sig : the signal number for the action -> target signal
//act : 가져올 action
//oact : 이전 action -> output parameter

//return 0 -> successful
//return -1 -> unsuccessful
```

```
//sigaction 구조체

struct sigaction{
  //signal handler
  //SIG_DFL, SIG_IGN or pointer to function -> sig로 설정된 signal이 도착하면 실행
① void (*sa_handler)(int);
  //SIG_DFL : restore the default action for the signal
  //SIG_IGN : handle the signal by ignoring it (throwing it away)

  //additional signals to be blocked during execution of handler
  //mask와 별도로 block할 signal이 있다면 지정
② sigset_t sa_mask;

  //special flags and options -> 없으면 0으로 설정
③ int sa_flags;
  //SA_SIGINFO of the sa_flags is cleared -> sa_handler specifies the action
  //SA_SIGINFO is set -> sa_sigaction specifies a signal-catching function

  //realtime handler(signal handler인데 parameter가 더 많은 버전)
④ void (*sa_sigaction) (int, siginfo_t *, void *);
}
```

POSIX : signal handler는 void type + integer parameter 한개
⇒ 더많이 전달하고 싶다면 sa_sigaction

①, ④ : signal handler가 실행 중일때 mask와 별도로 block할 signal이 있으면 지정

- example

• example 1

```
//setting the signal handler for SIGINT to mysighand

struct sigaction newact;
newact.sa_handler = mysighand;
newact.sa_flags = 0;

if((sigemptyset(&newact.sa_mask) == -1) ||( sigaction(SIGINT, &newact, NULL) == -1))
  perror("Failed to install SIGINT signal handler");
```

• example 2

```
//ignoring SIGINT if the default action is in effect for this signal

struct sigaction act;

if(sigaction(SIGINT, NULL, &act) == -1) //새로운 action 등록x -> 이전 action return
  perror("Failed to get old handler for SIGINT");

else if(act.sa_handler == SIG_DFL){ //default인지 확인
  act.sa_handler = SIG_IGN;
  if(sigaction(SIGINT, &act, NULL) == -1) //SIGINT를 ignore로 변경
    perror("Failed to ignore SIGINT");
}
```

- **program 8.5** : 반복문 돌면서 sin(x) (x= 0~1)의 평균을 계산
  - asynchronous한 event(ex. interrupt signal)을 받으면 종료
  - **doneflag** → critical section으로 선언되어야 함
    - signal handler가 중간에 실행되어 program의 jump가 이루어지기 때문임
    - main, signal handler 둘 다 참조하는 변수

      ☆ if-while문 내에서 doneflag을 읽을 때 signal handler가 작동된다면

      → 읽기 중단 후 수정 시작하여야 함

      ⇒ multi thread가 동시에 같은 변수로 작업하는 효과

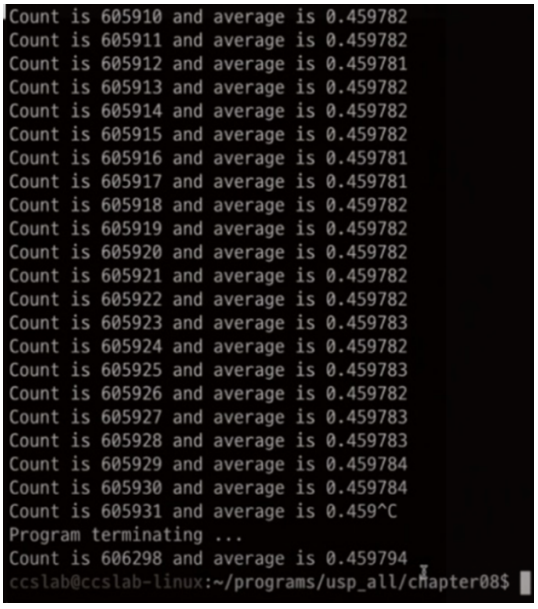
      → 충돌! ☆

      (예제는 single proces, thread이지만 critical section 필요)

    - sig_atomic_t : 자동적으로 critical section이 되는 integer type
    - volatile : complier가 register에서 load하지 않고 memory를 읽음

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

static volatile sig_atomic_t doneflag = 0;
```

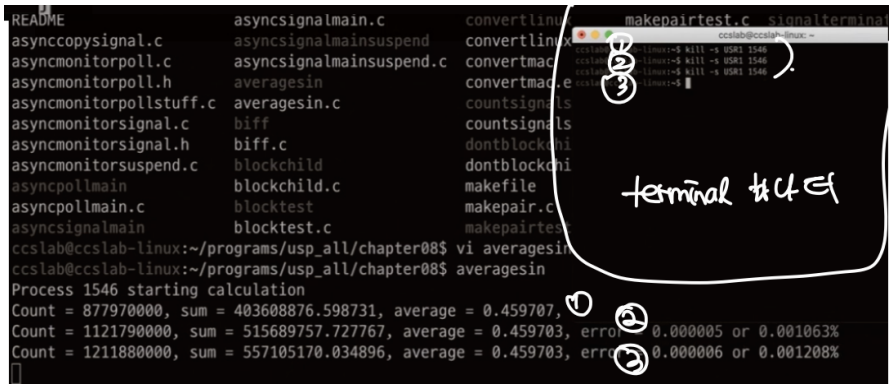
```c
/* ARGSUSED */
static void setdoneflag(int signo) {
   doneflag = 1;
}

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

   act.sa_handler = setdoneflag;        /* set up signal handler */
   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;
}
```



- **program 8.6** : 반복 10,000번째마다 결과값 포함한 string 생성

- buf에 중간 계산 결과 저장
- buf 읽어서 화면에 출력
- critical section : result() 내에서 buf 쓰는 일, main에서 buf를 access하는 것을 분리 필요!
  - pending signal 활용 : signal 잠시 pending 했다가 main 끝나면 가져옴

→ signal이 string을 수정하는 동안 결과값을 가져오는 것을 막음




```
#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 → 100 byte

static char buf[BUFSIZE]; → buffer
static int buflen = 0;  → 실제 길이

/* ARGSUSED */
static void handler(int signo) {          /* handler outputs result string */
   int savederrno;

   savederrno = errno; → errno을 main과 겹치지 않도록 저장
   write(STDOUT_FILENO, buf, buflen);
   errno = savederrno;
}  → async safe function

static void results(int count, double sum) {       /* set up result string */
   double average;  → 10000번째마다 호출해서 buf에 쓰는 역할
   double calculated;
   double err;
   double errpercent;
   sigset_t oset;
   sigset_t 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)
```

① sigemptyset
② sigaddset
③ sigprocmask
SIGUSR1
P
Critical section
→ signal mask 설정

```
      snprintf(buf, BUFSIZE, "No values calculated 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");
}

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;
}
```

Waiting

- **pause**

```
#include <unistd.h>

//내가 원하는 signal이 나오면 signal handler 전달 -> 그때까지 기다림
int pause(void);

//signal handler return 후 pause return

//return 항상 -1
```

```
//Exercise 8.21
static volatile sig_atomic_t sigreceived = 0;

while(sigreceived == 0)
  //이 사이에 해당 signal 도착하면 오류 발생 가능 -> thread suspend 됨
  //도착한 signal은 pause에 영향을 미치지 않고 pause는 또 같은 signal이 도착할 때까지 대기
  pause();
```

sigprocmask() 활용 ⇒ block

sigprocmask() ⇒ unblock

내가 원하는 signal 도착할 때까지 pause

이것도 같은 문제 발생 (not atomic)

→ unblocking a signal + starting pause at once 필요!

⇒ sigsuspend 사용!

- **sigsuspend**

```
#include <signal.h>

//target signal을 signal mask로 막아 놓은 뒤 signal이 process에 도착할 때까지 suspend
int sigsuspend(const sigset_t* sigmask);

//sigmask : to unblcok the signal the program is looking for
//          sigsuspend가 호출되기 이전에 mask 상태를 저장해놨다가 return 후 다시 reset
//return 항상 -1
```

sigmask로 !

+) target signal이 도착해야만 깨어남 → 이의 사용전에 검사할 필요 X

```
//wrong example
Sigfillset(&sigmost);
Sigdelset(&sigmost, signum);
Sigsuspend(&sigmost);

//signal이 code 시작 전에 도착하면
//process는 다른 signum을 가진 signal이 생성되지 않는 이상 deadlock에 걸림
```

⇒ sigprocmask로 target을 막아놔야 수행이 가능함

```
//correct example 1
//제한적인 사용 -> process가 suspend 될 때 오로지 signum signal만 nonblock
static volatile sig_atomic_t sigreceived = 0;
sigset_t maskall, maskmost, maskold;
int signum = SIGUSR1;
sigfillset(&maskall);
sigfillset(&maskmost);
sigdelset(&maskmost, signum);
sigprocmask(SIG_SETMASK, &maskall, &maskold);

//target signal이 도착해야만 깨어남
if(sigreceived == 0)
  sigsuspend(&maskmost);
sigprocmask(SIG_SETMASK, &maskold, NULL);
```

⇒ target signal

⇒ 전체 signal (현재 상태가 X)

⇒ target만 제거

① mask 설정

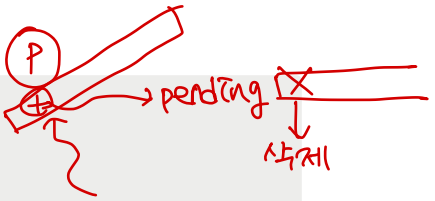
② sigsuspend : target만 pend대기에서 제외

③ mask 원상복구

```
//correct example 2
//다른 signal 도착 가능
//maskblocked : code 시작할 때의 block 되어 있던 signal 중 signum 제외하고 포함
static volatile sig_atmoic_t sigreceived = 0;
sigset_t maskblocked, maskold, maskunblocked;
int signum = SIGUSR1; → target signal

sigprocmask(SIG_SETMASK, NULL, &maskblocked); ⇒ 현재 mask 상태를 blocked에 저장
sigprocmask(SIG_SETMASK, NULL, &maskunblocked); ⇒ 〃 unblocked 〃
sigaddset(&maskblocked, signum); ⇒ target 추가
sigdelset(&maskunblocked, signum); ⇒ target 제외
sigprocmask(SIG_BLOCK, &maskblocked, &maskold);

while(sigreceived == 0)
  sigsuspend(&maskunblocked); → sigsuspend가 return될 때 원래의 signal mask 상태가 X
sigprocmask(SIG_SETMASK, &maskold, NULL); ⇒ 원래 mask 상태로 되돌림.
```

→ target signal이 도착할 때까지 도착하는 signal 모두 test + process suspend

- **sigwait**



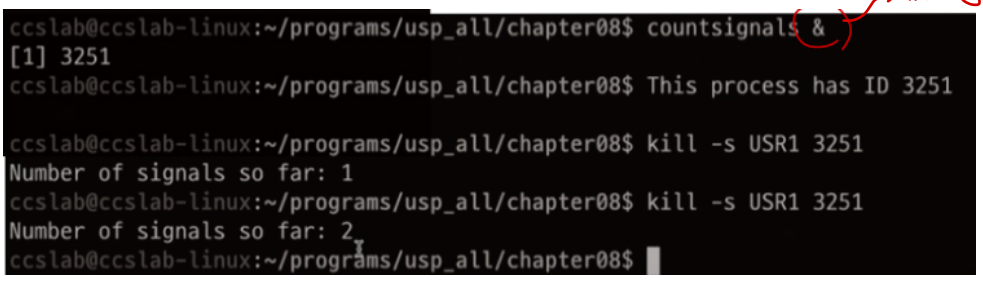

```
#include <signal.h>

//1. sigmask에 있는 signal이 도착하면 pending 될 때까지 block
//2. pending signals에서 해당 signal을 remove, signo return
//3. unblock

int sigwait(const sigset_t *restrict sigmask, int *restrict signo);
//signo : pending signal에서 지워진 signal 수는 signo pointer에 저장
                                    ↳output parameter

//return 0 -> successful
//return -1 -> unsuccessful
```

  - suspend()와 다른 점
    1. sigmask :
       a. sigsuspend : target signal을 제외한 signal mask를 사용
          ↔ sigwait : target signal을 포함한 signal mask를 사용
          ⇒ signal mask를 수정하지 않음.
       b. sigprog로 target signal을 block하고 실행하는 것은 동일
          i. target이 pending 되어야 실행 중단 x → pending 상태에서 먼저 거름

- Program 8.11 : SIGUSR1이 전달된 횟수를 count with sigwait()
  ⇒ signal이 항상 block되어 있기 때문에 signal handler 필요 X
  ⇒ process 전달 필요 X

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

int main(void) {
   int signalcount = 0;
```

```c
    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);
    }
}
```



Program control

Errors and Async-signal safety

1. signal에 의해 interrupt된 POSIX function이 재시작될 수 있는지
   a. interrupted function → return -1 (with errno set to EINTR)
      ⇒ program : 이 error를 조절 혹은 다시 호출해야 함

But 어떤 함수가 interrupt 될 수 있는지 찾는 것이 항상 가능한 일은 x

2. When signal handlers call *nonreentrant* functions
   a. *nonreentrant* : 함수 종료 되기 전에 다시 호출되는 것이 문제가 될 때
   b. async-signal safe : signal handler 내부 수행 중에 다른 곳에서 또 호출하더라도 문제 x
      i. POSIX : async-signal safe function list 존재

3. The handling of error that use errno

a. signal handler가 error handling interrupt 하지 않도록 주의 필요

b. signal handler : errno를 바꿀 수 있는 함수를 호출하는 경우 저장하고 restore해야 함

Useful rules for signal handling

1. 의심스러운 경우 →
   a. 프로그램 내에서 library 호출을 명시적으로 다시 시작
   b. restart library 사용
2. signal handler에서 사용되는 각각의 library function을 async-signal safe인지 확인
3. external variable을 수정하는 signal handler와
   해당 variable에 접근하는 다른 program 사이에 potential interaction이 있는지 유념
   → critical section인지 확인
4. 적절한 때에 errno를 저장하고 restore

⇒ context을 위해 그동 건너뛰는 system call 구현

Directly error handling → sigsetjmp, siglongjmp

- program : error를 handler하기 위해 signal들을 사용 ex. 실행 구문 jump
  - 오랫 동안 block되었던 긴 계산을 중단하는 동안 프로그램 종료를 막으려면
    1. ctrl-c : restart → 여러 layer로 쌓인 함수를 통해 return해야 해서 복잡함
    2. sigsetjmp, siglongjmp : 원하는 지점으로 바로 back

- **sigsetjmp, siglongjmp**

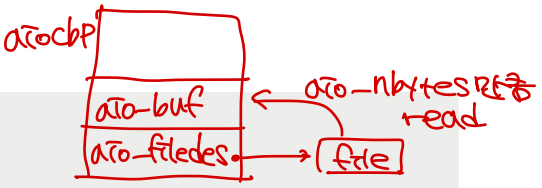
```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);
//provides a marker similar to a statement label
                    jump와 back
//env : marker로 jump하는데 필요한 정보를 초기화 →jump해서 env 시점으로 왔을 때, 상태 context 저장해놓는 변수
//save mask : 0이 아니면 현재 상태의 signal mask를 env buf에 저장

//return 0 : sigsetjmp를 호출하면서 다음에 jump할 위치를 설정해놨을 때
//return val value : jump하고 돌아왔을 때 siglonjmp()의 val

void siglongjmp(sigjmp_buf env, int val);
//jump back to the sigsetjmp point with the same sigjmp_buf variable

//val : sigsetjmp() return 값
```

```c
//example - Program 8.12
//ctrl-c이 입력되었을 때 어떻게 SIGINT handler가 메인 loop로 돌아가도록 set up 하는지?
//1. sigsetjmp -> siglongjmp 실행
//2. sigaction은 sigsetjmp가 딱 한 번만 실행되기 전에 appear
//3. flag -> jmpok : siglongjmp가 먼저 실행되는 것을 막는 flag

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

static sigjmp_buf jmpbuf;
static volatile sig_atomic_t jumpok = 0;

/* ARGSUSED */
static void chandler(int signo) {
   if (jumpok == 0) return;
   siglongjmp(jmpbuf, 1);
}

int main(void)  {
   struct sigaction act;

   act.sa_flags = 0;
   act.sa_handler = chandler;
   if ((sigemptyset(&act.sa_mask) == -1) || (sigaction(SIGINT, &act, NULL) == -1)) {
      perror("Failed to set up SIGINT handler");
      return 1;
   }
                                                  /* stuff goes here */
   fprintf(stderr, "This is process %ld\n", (long)getpid());
   if (sigsetjmp(jmpbuf, 1))
      fprintf(stderr, "Returned to main loop due to ^c\n");
   jumpok = 1;
   for ( ; ; )
      ;                                    /* main loop goes here */
}
```

```
ccslab@ccslab-linux:~/programs/usp_all/chapter08$ sigjmp
This is process 3850
^CReturned to main loop due to ^c
```

Programming with asynchronous I/O

- **asynchronous I/O (AIO)**
  - 요청을 시작하고 계속 실행하여 I/O 작업을 프로그램 실행과 asynchronously 처리
  - **POSIX:AIO** → aio_read(), aio_write(), aio_return(), aio_error()




```
#include <aio.h>

int aio_read(struct aiocb* aiocbp);
//a process queues a request for reading
//aiocbp->aio_buf에서 aiocbp->aio_fildes인 file로부터 aiocbp->aio_nbytes만큼 read

//return 0 -> successful
//return -1 -> unsuccessful

int aio_write(struct aiocb* aiocbp);
//'' for writing

//return 0 -> successful
//return -1 -> unsuccessful
```

```
struct aiocb_structure{
①int aio_fildes;
②volatile void *aio_buf;
③size_t aio_nbytes;

④off_t aio_offset; // the starting position for the I/O

⑤int aio_reqprio; //Lowers the priority of the request (요청우선순위 변경)

⑥struct sigevent aio_sigevent;
  //Specifies how the calling process is notified of the completion
  //aio_sigevent.sigev_notify = SIGEV_NONE -> the OS : not generate a signal
  //                          = SIGEV_SIGNAL
  //                -> the OS generates a signal specified in aio_sigevent.sigev_signo

⑦int aio_lio_opcode; // Used by the lio_listio function to submit multiple I/O requests
}
```

process에게 완료 사실 알리는 법
① 함수호출
② signal 등록 → signal handler

① 함수호출 방법

```
#include <aio.h>

ssize_t aio_return(struct aiocb* aiocbp);
//Returns the status of a completed underlying I/O operation

//return operation이 끝날 때 읽거나 쓰인 byte 수

int aio_error(const struct aiocb* aiocbp);
//Monitor the progress of the asynchronous I/O operation
요청된 I/O 진행 상황 파악 가능
//return 0 -> successful
//return EINPROGRESS -> 아직 실행 중
//return the err code -> fail
```

완료된 I/O return value

```
#include <aio.h>

int aio_suspend(const struct aiocb * const list[], int nent,
                const struct timespec* timeout);
//parameter로 지정한 AIO 완료될 때까지 suspend

//list[] : 요청한 구조체 arr
//nent : 요청한 구조체 개수
```

```
//timeout
// -> not NULL : AIO 끝남과 상관 없이 time_out이 만료되면 그냥 return
// -> NULL : AIO가 한번이라도 끝나면 return , aio_error()도 더이상 EINPROGRESS return x

//return 0 -> successful
//return -1 -> unsuccessful
```

```
#include <aio.h>
int aio_cancel(int fildes, struct aiocb* aiocbp);
//filedes인 file에 대해 요청했던 AIO를 취소하는 함수

//aiocbp : 지정한 I/O만 cancel하고 싶을 때 사용
// ->NULL : fildes에 있는 모든 pending request들을 cancel

//return AIO_CANCELED -> successful
//return AIO_NOTCANCELED -> 요청한 AIO 중 하나라도 cancel 되지 않았음
//return AIO_ALLDONE -> 요청한 AIO 모두 cancel 됨
//return -1 (errno) -> fail
```

→ asynchronous에서만 가능