7. 시그널

- 프로세스에서 발생한 이벤트를 다른 프로세스에 알리는 도구
- S/W 인터럽트라 하며 주로 OS가 일반 프로세스에 보낸다.
- 일반 프로세스에서는 커널의 도움으로 보낸다.

7.1 시그널 처리

- 유닉스 시그널을 소개
- 프로세스에서 시그널 수신을 처리하는 방법

시그널의 종류

- 유닉스의 시그널 이름은 SIG로 시작
  - 30여개의 시그널을 정의
  - 자세한 내용은 signal.h 파일에 명시

시그널	발생 조건
SIGINT	인터럽트키(ctrl-C)를 입력했을 때
SIGKILL	강제 종료 시그널로 프로세스가 이 시그널을 무시/블록할 수 없음
SIGIO	비동기 입출력이 발생했을 때
SIGPIPE	닫힌 파이프나 소켓에 데이터 쓰기/읽기를 시도할 때
SIGCHLD	프로세스가 종료되거나 취소될때 부모 프로세스에 전달
SIGPWR	전원 중단 및 재시작 시에 init 프로세스로 전달
SIGSTOP	키보드에서 중지키(보통 ctrl-z)를 눌렀을 때
SIGSYS	잘못된 시스템 호출 시 발생
SIGURG	(urgent) 대역외 데이터를 수신했을 때
SIGUSR1, 2	사용자 목적으로 사용할 수 있는 시그널

시그널의 종류

시그널	발생 조건
SIGHUP	(hang up) 터미널과 연결이 끊어졌을 때 세션 리더에게 보내짐
SIGQUIT	종료키(보통 ctrl-\)를 눌렀을 때
SIGILL	프로세스가 규칙에 어긋날 명령을 수행하려고 할 때
SIGTRAP	(trap) 프로그램이 디버깅 지점에 도달하면 전달됨
SIGABRT	abort() 함수를 호출할 때 발생
SIGFPE	(floating point error) 숫자를 0으로 나누거나 연산 에러 시 발생
SIGVTALRM	setitimer() 함수에 의한 가상 타이머 시간 만료를 알리는 시그널

- **시그널 발생**

```
int kill(pid_t pid, int sig); //pid인 프로세스에 sig 시그널을 발생
int raise(int sig); // 프로세스 자신에게 시그널을 보낸다.
```

시그널 처리 기본 동작

- 시그널 처리
  - 각 시그널에 따라 처리할 기본 동작이 미리 정해져 있으며 대부분의 시그널 처리 기본동작은 프로세스를 종료시킴
  - 어떤 시그널은 처리 기본동작이 단순히 시그널을 “무시(discard)”하는 것도 있음
  - 시그널 처리의 기본 동작을 프로그래머가 변경할 수 있음
    - 미리 정해진 함수를 수행하도록 할 수 있음
    - 시그널을 블록시킬 수 있음
- 시그널 수신 시 처리할 수 있는 동작
  - 정해진 기본 동작 수행
    - 프로세스 종료 또는 시그널 무시
  - 사용자가 지정한 작업 수행
    - 시그널 핸들러 수행
    - 시그널 무시 : 시그널을 받지 않은 것처럼 아무 영향을 받지 않음
    - 시그널 블록 : 시그널이 도착하면 시그널 대기큐로 보낸다
  - SIGKILL, SIGSTOP은 프로그래머가 변경할 수 없음

시그널 처리 기본 동작

- 시그널 대기큐
  - 시그널을 블록하면 해당 시그널이 프로세스에 도착했을 때 시그널 대기큐에 들어감
  - 프로세스는 하던 동작을 계속하고 나중에 시그널을 처리할 수 있음
  - 블록시킬 시그널은 bitmask 형태의 시그널 블록 마스크에 추가
  - 블록된 시그널은 누적되지 않음
    - SIGINT 시그널이 연속으로 두 개 도착하면 두 번째 시그널은 누적 않됨

시그널 핸들러

- 시그널 수신 시 호출할 함수(시그널 핸들러)를 등록하는 방법

시그널 집합

- 여러 개의 시그널을 한 번에 표시하기 위해 시그널 집합 타입 sigset_t를 사용
  - sigset_t 변수에는 여러 개의 시그널을 bitmask 형태로 누적하여 표현
- sigset_t의 변수를 다루는 주요 함수

```
#include <signal.h>

int sigemptyset(sigset_t *set);
            // 시그널 집합 set을 비운다.
int sigfillset(sigset_t *set);
            // set에 모든 시그널을 포함시킨다.
int sigaddset(sigset)t *set, int signum);
            // set에 signum 시그널을 추가한다.
int sigdelset(sigset_t *set, int signum);
            // set에서 signum 시그널을 삭제한다.
int sigismember(const sigset_t *set, int signum);
            // set에 sinum 시그널이 포함되어 있는지 확인한다.
```

시그널 집합

- **시그널 집합에 SIGINT 시그널만 남기는 코드**

```
sigset_t signals;
sigemptyset(&signals);
sigaddset(&signals, SIGINT);
```

- **시그널 집합에 SIGINT 시그널만 빠져있는 집합을 만드는 코드**

```
sigset_t signals;
sigfillset(&signals);
sigdelset(&signals, SIGINT);
```

- **시그널 집합에 SIGINT 시그널이 존재하는지 확인하는 함수**

```
sigset_t signals;
sigemptyset(&signals);
sigaddset(&signals, SIGINT);
if (sigismember(&signals, SIGINT))
    prinf(“this is true\n”);

if (sigismember(&signals, SIGPIPE))
    printf(“this is false\n”);
```

시그널 핸들러 설정

- sigaction()
  - 시그널 수신시 호출할 함수(시그널 핸들러)를 등록할 때 사용
    - 인수 signum은 처리할 대상이 되는 시그널
    - 인자 act는 sigaction 타입의 구조체

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

- sigaction 구조체
  - 시그널 핸들러 이름, 시그널 핸들러가 실행되는 도중에 블록시킬 시그널 집합, 각종 환경 설정을 위한 flag를 포함

```
struct sigaction {
    void (*sa_handler) (int);                          // 시그널 핸들러, 일반적으로 사용
    void (*sa_sigaction)(int, siginfo_t*, void *); // 핸들러, 부가적 정보를 이용시 사용
    sigset_t sa_mask; // 시그널 핸들러가 동작하는 동안 블록시킬 시그널 집합
    int sa_flags;  // 환경 설정을 위해 사용
    void (*sa_restore)(void);// 사용 않함
}
```

sa_flags

- SA_RESTART
  - 시그널이 도착하면 프로세스의 현재 작업이 인터럽트되고 등록된 시그널 핸들러가 수행되고 시그널 핸들러가 작업을 마치면 이전의 수행 지점으로 돌아가 수행을 계속
  - 시스템 콜 수행중 시그널이 발생
    - 시스템콜은 -1을 반환하고 errno 변수에 EINTR을 저장
    - 핸들러 처리 후에 이전의 시스템 콜 수행 지점으로 돌아가지 못하고 종료될 수 있음
    - SA_RESTART로 해결
- SA_NOCLDWAIT
  - 좀비 프로세스를 만들지 않도록 하기 위한 플래그
  - SIGCHLD 시그널 처리시에 주로 사용됨
- SA_NODEFER
  - 핸들러 수행 중 도착한 동일한 시그널이 누적되도록 함
  - 도착한 시그널의 수만큼 핸들러를 수행
- SA_SIGINFO
  - sigaction 구조체의 sa_sigaction 인자는 부가 정보를 이용할 때 사용
  - 설정하지 않으면 sa_handler에 명시된 핸들러가 수행
  - 설정하면 sa_sigaction에 명시된 핸들러가 수행됨

sa_flags

- SA_ONESHOT or SA_RESETHAND
  - **시그널 핸들러가 한 번 실행된 후에는 시그널 처리 기본 동작으로 되돌아감**
  - **즉, 설치한 시그널 핸들러는 한 번만 실행되고 이후에 발생하는 시그널은 SIG_DFL(기본 동작 설정)에 의해 수행된다.**

- sigaction **구조체를 이용한 시그널 핸들러 등록 코드 예**
  - SIGINT **시그널이 발생하면** catch_sigint()**를 수행**

```
struct sigaction act;
sigfillset(&masksets);                  // 블록할 시그널 선택
act.sa_handler = catch_sigint;          // 시그널 핸들러 등록
sigfillset(&act.sa_mask);               // 핸들러 수행 중 모든 시그널을 블록
act.sa_flags = 0;                       // 기본 플래그
sigaction(SIGINT, &act, NULL);          // 핸들러를 등록
```

Signal-unsafe 함수

- Signal-unsafe 함수
  - 시그널 핸들러 내에서 호출하면 그 결과가 불확실한 함수
  - malloc(), free(), alarm(), sleep(), fcntl(), fork() 등을 시그널 핸들러 내에서 호출하면 메모리나 타이머 값의 변경으로 핸들러 외부의 작업과 충돌 가능성이 있다.
  - 정적 변수나 전역 변수를 사용하고 있는 경우가 많음
- 시그널 핸들러 내에서 signal-unsafe 함수 호출을 피하는 방법
  - 시그널 핸들러 내에서 특정 플래그만 설정하고 핸들러가 종료된 후 플래그를 확인하고 signal-unsafe 함수를 호출 하는 방식을 사용

signal()

- sigaction() **소개 전의 시스템 콜, 시그널 핸들러를 등록하거나 시그널의 무시를 설정하기 위해 사용**

```
#include <signal.h>
void (*signal(int signum, void (*handler)(int)))(int);
```

  - signum **인자** : **발생하는 시그널**
  - handler **인자의 종류**
    - signal_handler : **시그널 핸들러 함수명**
    - SIG_IGN : **시그널을 무시하도록 설정**
    - SIG_DFL : **기본 동작으로 설정**
  - **시그널 핸들러가 한 번 실행하고 난 후에는 시그널 처리 기본 동작으로 자동 환원**
    - **시그널 핸들러는 1회만 유효**
    - **필요하면 시그널 핸들러를 다시 등록해야 함**
    - **운영체제에 따라서 시그널 핸들러가 계속 유효한 경우도 있음**

7.2 시그널 처리 예

SIGINT 처리 예(catch_signal.c)

- SIGINT 시그널
  - 키보드에서 ctrl-c 를 누르면 발생
  - 기본 동작은 프로세스를 종료
- 예제 프로그램
  - SIGINT 시그널이 발생하면 종료 대신 catch_sigint() 함수를 호출
  - 키보드에서 ctrl-c를 누르면 count 값이 2까지는 키보드 입력을 받고 화면에 “(count=1) CTRL-C pressed!” 를 출력하고 프로그램은 종료 시키지 않는다.
  - 실행 결과

```
$ catch_sigint
^C
(count=0) CTRL-C pressed!
^C
(count=1) CTRL-C pressed!
^C
(count=2) CTRL-C pressed!
```

시스템 콜 수행중의 시그널 처리

- **시스템 콜 수행 중 시그널 발생**
  - **시그널 핸들러 처리 후에 이전의 시스템 콜 수행 지점으로 돌아가지 못하고 종료될 수 있음**
  - read()**에서 에러가 발생한 경우 이것이** EINTR **에러인지 확인하는 예**
    - For **루프를 수정**

```
for(count=0; count<3; count++) {
    int n = read(0, buf, sizeof(buf));
    if (n==-1 && errno == EINTR)
        printf("read 함수 interrupted\n");
    else
        printf("\t** %d byte input\n", n);
```

    - **시그널 핸들러가 리턴한 후** read()**는 이전의 읽기작업을 계속하는 것이 아니라 중단되어 다음 문장을 실행**
    - **실행결과**

```
^C
(count=0) CTRL_C pressed!
Read 함수 Interrupted
^C
(count=0) CTRL_C pressed!
Read 함수 Interrupted
. . .
```

시스템 콜 수행중의 시그널 처리(2)

- **시그널 핸들러를 수행 후 시스템 콜의 동작을 계속 수행하도록 설정**
  - sa_flags**에** sa_RESTART**를 설정**
    - act.sa_falgs = 0 **를** act.sa_falgs = SA_RESTART **로 수정**
  - **실행결과**

```
^C
(count=0) CTRL_C pressed!
^C
(count=0) CTRL_C pressed!
^C
(count=0) CTRL_C pressed!
^C
(count=0) CTRL_C pressed!

a
        ** 2 byte input
b
        ** 2 byte input
c
        ** 2 byte input
```

SA_RESTART와 select()

- select() : **입출력 멀티플렉싱을 위해 서버에서 자주 사용**
  - SA_RESTART **플래그래 의해 재시작되지 않는 예외적인 함수**
  - **솔라리스** 7, **리눅스 커널** 2.4.18**에서는 자동으로 재시작 되지 않음**
  - **호환성 있는 코드 작성을 위해서는 재시작 여부를 검사하는 것이 안전**

```
while (1) {
  if (select(maxfdp1, &read_fds, NULL, NULL, NULL) < 0) {
    if (errno == EINTR)
        continue;
    else {
      perror("select fail");
      exit(0);
    }
  else
    break;
}
```

중복 블록된 시그널 처리

- **중복된 시그널은 기록이 남지 않음**
- **시그널이 중복된 사실을 알 필요가 있을 때 처리하는 방법**

```
// 시그널 핸들러, 3초간 sleep 하도록 변경
void catch_sigint(int signum)
{
    static int call_count = 0; // 정적 변수 사용하여
    printf('catch_sigint %d번째 호출 \n", call_count++);
    sleep(3);
    printf("catch_sigint 리턴\n");
    return;
}
```

```
^C
catch_sigint 0 번째 호출
^C ^C
catch_sigint 리턴
catch_sigint 1 번째 호출
catch_sigint 리턴
a
            ** 2 byte input
b
            ** 2 byte input
```

- **연속 3회의 ctrl-c를 입력하는 경우**
  - **시그널 핸들러가 수행되는 동안 두 개의 SIGINT 시그널이 전달됨**
  - **추가로 2번의 시그널 핸들러가 수행되지 않고 1회만 수행됨**
    - **시그널 대기큐에는 같은 시그널이 두 개 이상 쌓이지 않기 때문**
- **중복 도착한 시그널의 수만큼 시그널 핸들러 구동**
  - SA_NODEFER **옵션 사용**

```
act.sa_flags = SA_RESTART | SA_NODEFER
```

SIGALAM에 의한 read()의 timeout 예

- alarm()
  - 임의의 작업을 수행하는 동안 타이머를 구동시키고 지정한 시간이 지나면 SIGALARM 시그널을 발생
  - 프로세스 내에서 한번에 하나만 실행
    - 다른 곳에서 alarm()을 중복하여 호출하면 타임아웃 값은 새롭게 변경

- 예 : alarm_intr.c
  - read() 함수가 기다리는 타임아웃을 5초로 제한하려면 read() 함수 호출 직전에 alarm(5)를 호출

7.3 SIGCHLD와 프로세스의 종료

프로세스의 종료

- 모든 프로세스는 종료할 때 종료 상태를 남기며 자신의 부모 프로세스가 이 상태를 읽어 보도록 할 수 있다.
- 모든 프로세스는 종료 시 자신의 부모 프로세스에게 SIGCHLD 시그널을 보내며 부모 프로세스는 이 시그널을 받아 처리할 수 있다.
- 자식 프로세스에게 어떤 작업을 맡겼을 경우 자식 프로세스가 종료 시 처리하는 방법
- SIGCHLD 시그널 처리 방법

wait()

- **부모 프로세스가 자식의 종료 시점을 알거나 종료 상태 값을 알기 위해 wait()나 waitpid() 함수를 이용**
  - **부모 프로세스가 자식 프로세스가 종료된 것을 확인하고 어떤 작업을 처리할 경우의 코드**

```
pid_t pid;
int stat;
// 부모 프로세스의 일 처리
pid = wait(&stat);   // 자식 프로세스가 종료될 때까지 이곳에서 블록됨
// 자식 프로세스 종료 후에 처리할 일
```

    - **종료상태 값은 stat 인자로 리턴**
  - **자식 프로세스가 종료된 후 부모 프로세스가 종료 상태를 읽어가지 않으면 자식 프로세스는 기본적으로 좀비(zombie) 상태가 됨**
    - **좀비 상태는 사용자 영역 메모리는 모두 free 되었지만 커널이 관리하는 메모리는 아직 남아 있는 상태 (메모리 낭비)**
    - **좀비 상태를 만든 이유는 나중에 자식 프로세스의 종료 상태를 읽기 위함**
  - **프로세스가 자주 생성, 종료되는 프로그램에서는 좀비 프로세스가 남지 않도록 해야 함**
    - **SIGCHLD(기본 처리 동작은 무시) 핸들러 처리 또는 wait() 함수를 이용하여 자식 프로세스의 종료를 파악**

기본동작으로서 SIGCHLD를 무시하는 경우

- 자식 프로세스의 종료 상태를 읽을 필요 없는 경우
  - SIGCHLD 시그널을 무시
- 자식 프로세스의 종료 시점 파악이 필요한 경우
  - wait()의 호출 시점에 따라 다른 처리방법이 필요
    - wait()를 호출한 이후 자식 프로세스가 종료한 경우 wait()가 리턴
      - 다수의 자식 프로세스가 있는 경우 wait()를 여러 번 호출할 수 있고 자식 프로세스가 종료될 때마다 wait()가 리턴
    - 부모 프로세스가 wait()를 호출하기 전에 자식 프로세스가 종료하면 자식프로세스는 좀비가 됨
      - 부모 프로세스는 필요 시 나중에 wait()를 호출하여 자식의 종료상태를 읽음
      - 부모가 wait()를 호출하지 않고 종료하면 시스템의 init() 프로세스가 좀비 프로세스에 대해 wait()를 대신 호출(좀비 상태를 제거)

SIG_IGN으로 SIGCHLD를 무시하는 경우

- SIGCHLD를 무시하는 것과 거의 같은 동작을 수행
- 차이점
  - 부모 프로세스가 wait()를 호출하기 전에 자식 프로세스가 종료한 경우에도 좀비가 되지 않음
    - 종료된 자식 프로세스의 상태는 얻지 못함
  - wait() 호출시 자식 프로세스가 없으면 -1을 리턴
- 공통점
  - 차이점 이외의 경우 기본 동작과 같음
    - wait()를 호출한 이후 자식 프로세스가 종료한 경우 wait()가 리턴
    - 부모 프로세스가 wait()를 호출하기 전에 자식 프로세스가 종료하면 자식프로세스는 좀비가 됨

시그널 핸들러를 등록한 경우

- **자식 프로세스가 종료된 시점에서 종료 상태를 읽기 위해** SIGCHLD **시그널 핸들러를 등록**
  - **시그널 핸들러 내에** wait() **함수로 자식의 종료 상태를 얻음**
  - **시그널 핸들러가 실행되었다고 자식의 좀비 상태가 끝나는 것이 아니라** wait()**가 실행되어야 좀비 프로세스가 사라짐**

프로세스의 종료 처리

- 멀티프로세스로 서버를 구현하는 경우
  - 여러 개의 자식 프로세스들이 작업을 나누어 수행하고 부모 프로세스는 이들을 관리
  - 부모 프로세스는 자식 프로세스의 종료 시점을 확인하기 위해 wait()를 호출
  - 자식 프로세스의 종료 상태를 얻지 않아도 되면 SIGCHLD 핸들러로 SIG_IGN으로 등록하고 wait()가 -1을 리턴할 때까지 검사
    - wait()가 -1을 리턴하는 것은 더 이상 자식 프로세스가 없음을 의미
- wait_test.c 는 프로세스의 종료 조건들을 시험하는 프로그램
  - 5개의 프로세스를 생성시키는 코드
    - 0, 1, 2 프로세스는 즉시 종료(부모 프로세스가 wait 호출 전에 종료)
    - 3, 4 프로세스는 sleep 후 종료(wait 호출 후에 종료)

```
for(i=0; i<5; i++) {
  if (fork() == 0) {
    if (i>2) sleep(6);
      printf("(%d번 Child), PID=%d, PPID=%d Exited\n", I, getpid(), getppid());
    exit(13);            // 종료 상태 값 : 13(임의 값을 배정)
  }
}
```

프로세스의 종료 처리

- **시그널 핸들러를 등록하는 코드**

```
struct sigaction sact;
sact.sa_flags = 0;
sigemptyset(&sact.sa_mask);
sigaddset(&sact.sa_mask, SIGCHLD);
sact.sa_handler = SIG_DFL;              // 기본 동작(시그널 무시)
//sact.sa_handler = SIG_IGN;            // 시그널 무시를 등록
//sact.sa_handler = catch_sigchld;      // 시그널 핸들러 등록
Sigaction(SIGCHLD, &sact, NULL);
```

- wait() **함수가** -1**을 리턴할 때** errno**의 값을 확인하는 코드**

```
for( ; ; ) {
  chstat = -1;
  n = wait(&chstat);
  printf("# wait = %d(child stat=%d)\n", n, chstat);
  if (n == -1) {
    if (errno = ECHILD) {
      perror("기다릴 자식프로세스가 존재하지 않음");
      break;  }
    else if (errno == EINTR) {
      perror("wait 시스템 콜이 인터럽트 됨");
      continue;  }
  }
}
```

wait_test.c의 수행결과(SIG_DFL)

- (1번 Child),PID=22047,PPID=22045 Exited
- (2번 Child),PID=22048,PPID=22045 Exited
- (0번 Child),PID=22046,PPID=22045 Exited
- ----------
- PID TTY TIME CMD
- 22045 pts/0 00:00:00 wait_test_DFL
- 22046 pts/0 00:00:00 wait_test_DFL <defunct>
- 22047 pts/0 00:00:00 wait_test_DFL <defunct>
- 22048 pts/0 00:00:00 wait_test_DFL <defunct>
- 22049 pts/0 00:00:00 wait_test_DFL
- 22050 pts/0 00:00:00 wait_test_DFL
- 22051 pts/0 00:00:00 ps
- ----------
- #(Parent) wait **호출함**
- # wait = 22046(child stat=3328)
- # wait = 22047(child stat=3328)
- # wait = 22048(child stat=3328)
- (3번 Child),PID=22049,PPID=22045 Exited
- (4번 Child),PID=22050,PPID=22045 Exited
- # wait = 22049(child stat=3328)
- # wait = 22050(child stat=3328)
- # wait = -1(child stat=-1)
- **기다릴 자식프로세스가 존재하지 않음**: No child processes
- #(Parent) **종료함**

wait_test.c의 수행결과(catch_sjgchld)

- (0번 Child),PID=22085,PPID=22084 Exited
- ###(Parent) catch SIGCHLD
- (1번 Child),PID=22086,PPID=22084 Exited
- ###(Parent) catch SIGCHLD
- (2번 Child),PID=22087,PPID=22084 Exited
- ###(Parent) catch SIGCHLD
- ----------
- PID TTY TIME CMD
- 22084 pts/0 00:00:00 wait_test_sigch
- 22085 pts/0 00:00:00 wait_test_sigch <defunct>
- 22086 pts/0 00:00:00 wait_test_sigch <defunct>
- 22087 pts/0 00:00:00 wait_test_sigch <defunct>
- 22088 pts/0 00:00:00 wait_test_sigch
- 22089 pts/0 00:00:00 wait_test_sigch
- 22090 pts/0 00:00:00 ps
- ###(Parent) catch SIGCHLD
- ----------
- #(Parent) wait **호출함**
- # wait = 22085(child stat=3328)
- # wait = 22086(child stat=3328)
- # wait = 22087(child stat=3328)
- (3번 Child),PID=22088,PPID=22084 Exited
- (4번 Child),PID=22089,PPID=22084 Exited
- ###(Parent) catch SIGCHLD
- # wait = 22088(child stat=3328)
- # wait = 22089(child stat=3328)
- # wait = -1(child stat=-1)
- **기다릴 자식프로세스가 존재하지 않음**: No child processes
- #(Parent) **종료함**

wait_test.c의 수행결과(SIG_IGN)

- (0번 Child),PID=22064,PPID=22063 Exited
- (1번 Child),PID=22065,PPID=22063 Exited
- (2번 Child),PID=22066,PPID=22063 Exited
- ----------
- PID TTY TIME CMD
- 22063 pts/0 00:00:00 wait_test_IGN
- 22067 pts/0 00:00:00 wait_test_IGN
- 22068 pts/0 00:00:00 wait_test_IGN
- 22069 pts/0 00:00:00 ps
- ----------
- #(Parent) wait **호출함**
- (3번 Child),PID=22067,PPID=22063 Exited
- # wait = 22067(child stat=3328)
- (4번 Child),PID=22068,PPID=22063 Exited
- # wait = 22068(child stat=3328)
- # wait = -1(child stat=-1)
- **기다릴 자식프로세스가 존재하지 않음**: No child processes
- #(Parent) **종료함**

프로세스의 종료 처리

- 기본 동작(시그널 무시) 실행 결과
  - 5개의 자식 프로세스 중 부모가 wait()를 호출하기 전에 종료한 0,1,2 번 프로세스는 좀비가 됨
  - 이 내용은 system("ps –a")로 확인 가능
  - 나중에 부모 프로세스가 wait()를 호출하여 좀비를 제거하고 종료 상태를 얻음
- 시그널 무시(SIG_IGN)를 등록한 경우
  - 좀비 프로세스가 없음
  - 자식 프로세스가 좀비 상태가 되지 않으므로 부모 프로세스가 wait()를 호출해도 종료 상태를 얻을 수 없음
  - LINUX의 동작과 다른 시스템도 있음
  - 솔라리스의 경우 wait() 함수는 모든 자식 프로세스가 종료할 때까지 wait()가 블록상태가 됨
  - 리눅스 에서 모든 자식이 리턴할 때까지 기다리는 코드

```
while (!(wait() == - && errno==ECHILD));
```

프로세스의 종료 처리

- SIGCHLD 시그널 핸들러를 등록한 경우
  - SIGCHLD에 대한 catch_sigchld()가 수행
  - catch_sigchld() 내에서 wait()를 호출하지 않으므로 종료한 자식 프로세스는 좀비 프로세스가 됨
  - 아직 종료하지 않은 자식 프로세스는 wait()로 대기하고 이 프로세스가 종료되면 부모 프로세스에게 SIGCHLD 시그널 전송
  - wait() 시스템 콜을 호출중인 부모 프로세스의 블록 상태
    - 일반적으로 시그널이 발생하면 시스템 콜은 인터럽트됨
    - SIGCHLD 시그널이 발생하면 대기중인 wait()는 인터럽트 되면서 catch_sigchld()가 호출됨
    - errno가 EINTR을 확인하는 코드에서 break를 사용했다면 wait() 호출 후 한번만 자식 프로세스를 기다리게 됨
    - 호환성을 위해서 break가 아닌 continue를 사용하는 것이 좋음
- 시스템 콜 인터럽트 경우
  - read()와 같은 시스템 콜은 인터럽트 됨
  - 시그널 처리가 SIG_D리 또는 SIG_IGN으로 설정되어 있으면 영향을 받지 않음

프로세스의 종료 처리

- SA_NOCLDWAIT 옵션
  - SIGCHLD 시그널에 대해 SIG_DFL이나 핸들러 함수를 설정하면 wait() 호출 이전에 종료한 자식 프로세스는 좀비가 됨
  - SIG_IGN을 등록했을 때에는 좀비 프로세스가 만들어지지 않음
  - sigaction 구조체의 sa_flags에 SA_NOCLDWAIT 플래그를 설정하여 좀비 프로세스 생성을 막을 수도 있음

```
sact.sa_flags = SA_NOCLDWAIT;
sigemptyset(&sact.sa_mask);
Sigaddset(&sact.sa_mask, SIGCHLD);
```

- waitpid()
  - 특정 PID를 갖는 자식 프로세스의 종료만을 기다리는 함수
  - waitpid()를 호출하였는데 앞으로 종료할 자식 프로세스가 없다면 영원히 블록됨
  - 블록 현상을 막기 위해 flags에 WNOHANG 옵션을 사용할 수 있다
  - WNOHAND 옵션을 사용할 경우
    - 실행중인 자식 프로세스가 있을 경우에만 블록되며 없으면 블록되지 않음

7.4 시그널 블록 마스크의 조작

특정 시그널을 선택적으로 블록시키기 위한 함수

-

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

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigpending(sigset_t *set);
int sigsuspend(const sigset_t *mask);
```

sigprocmask()

- int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);

- **시그널 블록 마스크 내용을 변경시킬 때 사용**
  - sigset_t
    - **현재 시그널 블록 마스크에 추가되거나 삭제될 시그널 집합**
  - how : **인자** set**에 대해 처리할 동작**
    - SIG_BLOCK : **현재의 시그널 블록 마스크에** set**을 추가**
    - SIG_UNBLOCK : **현재의 시그널 블록 마스크에** set**을 제거**
    - SIG_SETMASK : **새로운 시그널 블록 마스크를 설정**
  - old_set
    - **이전 시그널 블록 마스크를 저장**
    - **나중에 원래의 시그널 집합으로 환원할 때 사용**

sigprocmask()

- **원래의 시그널 집합으로 환원시키는 예**

```
sigset_t new;
sigset_t old;

sigemptyset(&new);
sigaddset(&new, SIGINT);
sigaddset(&new, SIGUSR1);

sigprocmask(SIG_BLOCK, &new, &old);

…

// 원래 시그널 마스크 값으로 환원
Sigprocmask(SIG_SETMASK, &old, NULL);
```

sigpending()

- int sigpending(sigset_t *set);
- **시그널이 블록 상태에 있는 경우** pending **시그널이라 함**
- sigpending() **함수는** pending **되어 있는 시그널을 알아내는데 사용**
- **사용 예**

```
sigset_t psets;

sigemptyset(&psets);

sigpending(&psets);

if (sigismember(&psets, SIGINT)) // pending 시그널 중에 SININT 가 있는지 확인
  printf("SIGINT is pending\n");
```

sigsuspend()

- int sigsuspend(const sigset_t *mask);

- sigsuspend() **함수를 사용하여** pending**된 시그널을** unblock**하고 프로세스에게 보낼 수 있음**
- **현재 프로세스에 도착하지 않은 시그널에 대해** sigsuspend() **함수를 호출하면 해당 시그널이 도착할 때까지 블록 상태가 됨**