



CHUNGNAM NATIONAL UNIVERSITY



시스템 프로그래밍

강의 9. 시그널

교재 8.5~8.6

<http://eslab.cnu.ac.kr>

전달사항

■ 다음주 예습

● 846-851 (9.9~9.9.2)

복습 : fork 와 execve 사용하기: 셸 프로그램

- 셸 *shell* 은 사용자의 명령을 처리해 주는 응용 프로그램이다
 - tsh의 메인 루틴

```
int main()
{
    char cmdline[MAXLINE];

    while (1) {
        /* read */
        printf("> ");
        Fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);

        /* evaluate */
        eval(cmdline);
    }
}
```

- 메인 루프는 사용자 명령 입력 및 명령처리로 구성된다

복습 : 간단한 쉘의 eval 함수

```
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* argv for execve() */
    int bg;               /* should the job run in bg or fg? */
    pid_t pid;            /* process id */

    bg = parseline(cmdline, argv);
    if (!builtin_command(argv)) {
        if ((pid = Fork()) == 0) { /* child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }

        if (!bg) { /* parent waits for fg job to terminate */
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else /* otherwise, don't wait for bg job */
            printf("%d %s", pid, cmdline);
    }
}
```

복습 : 간단한 쉘 구현의 문제점

- 쉘은 포그라운드 작업들을 정확히 기다리고, 제거해준다
- 그러나, 백그라운드 작업들은 어떻게 처리하는가 ?
 - 이들은 종료되면 좀비가 된다
 - 이들은 쉘이 제거하지 않는데, 그 이유는 쉘은 대개 종료되지 않기 때문이다
 - 이 경우, 메모리 누수가 발생하며, 그 결과 커널 메모리 부족현상이 발생한다
 - 최신 **Unix**: 프로세스 메모리 할당량이 있어서 이를 초과하면 새로운 명령을 실행하지 못하게 된다. **fork()**는 -1을 리턴
- 해결책
 - 백그라운드 작업을 제거하기 위해서 **signal** 이라고 하는 방법을 사용한다

오늘의 주제

■ Signals

- 커널 소프트웨어

■ Long jumps

- 응용 프로그램

■ More on signals

시그널 – 상위수준의 예외적 제어 흐름

- 시그널 signal은 어떤 이벤트가 시스템에 발생했다는 것을 프로세스에게 알려주는 짧은 메시지
 - 예외상황과 인터럽트를 커널에서 추상화한 개념
 - 커널이 프로세스에게 보내준다 (간혹 다른 프로세스가 요청하는 수도 있다)
 - 서로 다른 시그널들은 정수 아이디로 구분한다 (1-30)
 - 시그널에 포함된 유일한 정보는 아이디와 시그널이 도착했다는 사실이다

ID	Name	기본 동작	해당 이벤트
2	SIGINT	Terminate	키보드 인터럽트 (ct1-c)
9	SIGKILL	Terminate	프로그램을 종료시킨다 (무시 또는 변경불가)
11	SIGSEGV	Terminate & Dump	세그멘테이션 위반
14	SIGALRM	Terminate	타이머 시그널Timer signal
17	SIGCHLD	Ignore	자식이 정지 또는 종료함

시그널의 개념

■ 시그널의 송신 Sending a signal

- 커널은 목적지 프로세스의 컨텍스트 내 일부 상태를 갱신하는 방법으로 시그널을 목적지 프로세스에 보낸다
- 커널은 다음과 같은 경우 에 시그널을 보낸다 :
 - ▶ 커널이 divide-by-zero (SIGFPE) 나 자식 프로세스의 종료와 같은 (SIGCHLD) 시스템 이벤트를 감지했을 때
 - ▶ 다른 프로세스가 kill 시스템 콜을 호출해서 커널이 목적지 프로세스로 시그널을 보낼 것을 요청했을 때

시그널의 개념(continued)

■ 시그널의 수신 Receiving a signal

- 목적지 프로세스가 시그널을 받을 때, 어떤 형태로든 반응을하도록 커널에 의해 요구될 때, 시그널을 받는다고 한다.
- 세가지 반응 :
 - ▶ 무시 Ignore the signal (do nothing)
 - ▶ 대상 프로세스를 종료 (with optional core dump).
 - ▶ **시그널 핸들러** 라고 부르는 유저레벨 함수를 실행하여 시그널을 잡는다(catch)
 - 비동기형 인터럽트에 대한 응답으로 호출되는 인터럽트 핸들러 방식과 유사
 - 여기서 질문 하나. 인터럽트 발생시에 CPU가 어떻게 동작하는가?

시그널의 개념(continued)

■ 시그널 관련용어

- 전송하였지만, 아직 수신되지 않은 시그널은 “대기하고 있다(pending)”고 한다
 - ▶ 어느 특정 타입의 시그널에 대해서 최대 한 개의 대기 시그널이 존재할 수 있다.
 - ▶ 중요 : 시그널은 큐에 들어가지 않는다
 - 만일 어떤 프로세스가 k타입의 대기 시그널을 가지고 있다면, 다음에 이 프로세스로 전달되는 k타입의 시그널들은 무시된다.
- 프로세스는 특정 시그널의 수신을 블록할 수 있다.(시그널의 거절)
 - ▶ 블록된 시그널들은 전달될 수 있지만, 이 시그널이 풀릴 때까지는 수신될 수 없다.
 - ▶ 프로세스에서 블록될 수 없는 유일한 시그널은 SIGKILL이다.
- 대기하는 시그널은 최대 한번만 수신할 수 있다
 - ▶ 커널은 대기 시그널들을 나타내기 위하여 비트 벡터를 사용한다.

시그널의 개념- 구현

■ 커널은 각 프로세스의 컨텍스트(?)에 pending 과 blocked 비트 벡터를 가지고 있다.

● **pending** – 대기 시그널들을 표시

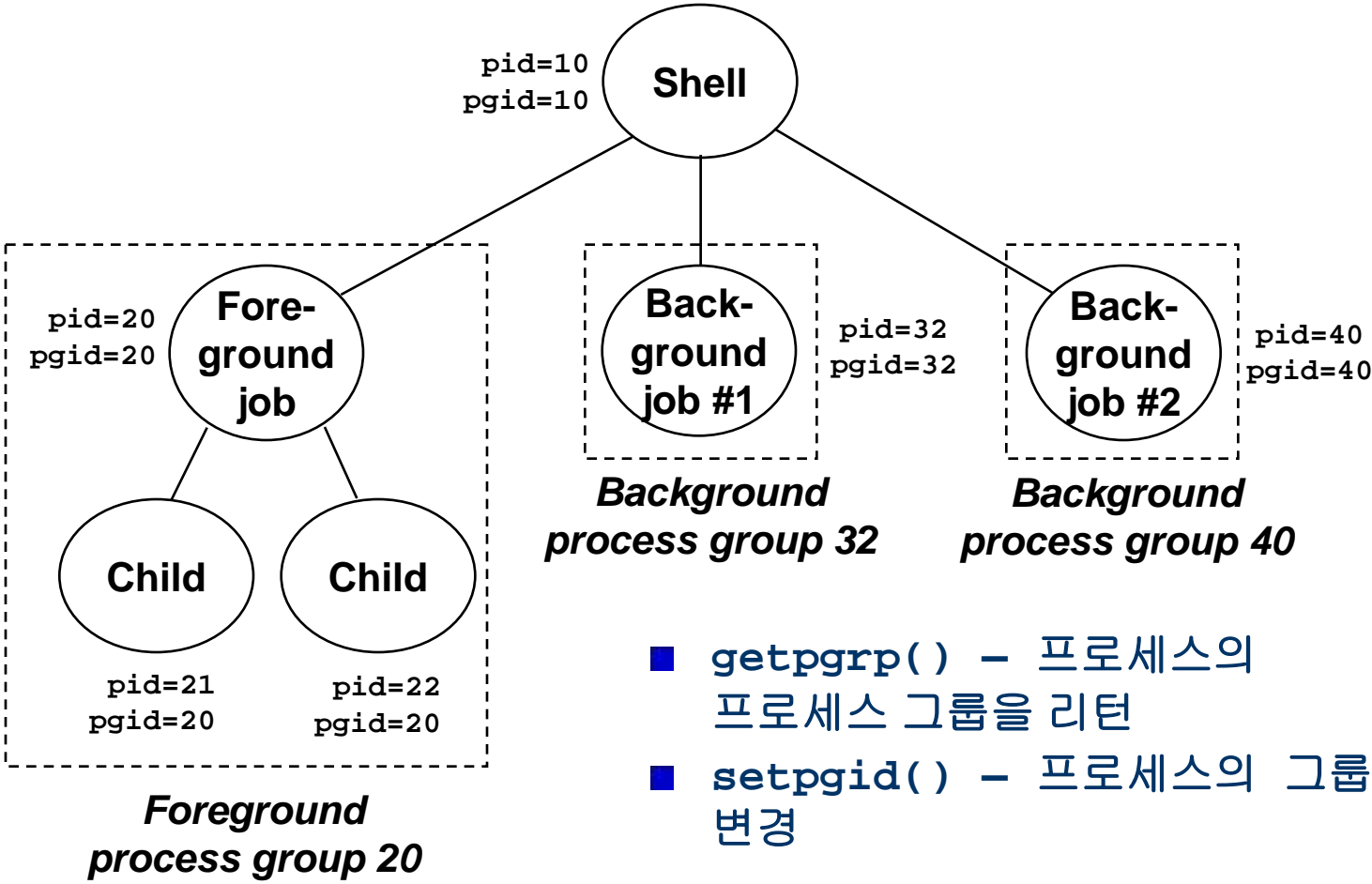
- ▶ 커널은 타입 k 시그널이 도착할 때마다 pending 값의 k 번째 비트를 1로 설정
- ▶ 커널은 타입 k 시그널을 수신할 때마다, pending 값의 k 번째 비트를 0으로 설정

● **blocked** – 블록된 시그널들을 표시

- ▶ sigprocmask 함수를 사용하여 응용 프로그램이 1 또는 0으로 설정

프로세스 그룹

■ 각 프로세스는 하나의 프로세스 그룹에 속한다



kill 명령을 이용한 시그널 보내기

kill 시스템 콜은 프로세스 또는 프로세스 그룹에 임의의 시그널을 보낸다

예제

● **kill -9 24818**

▶ SIGKILL 을 process 24818로 보냄

● **kill -9 -24817**

▶ 프로세스 그룹 아이디 24817의 각 프로세스에 SIGKILL을 보냄

```
linux> ./forks 16
linux> Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817
```

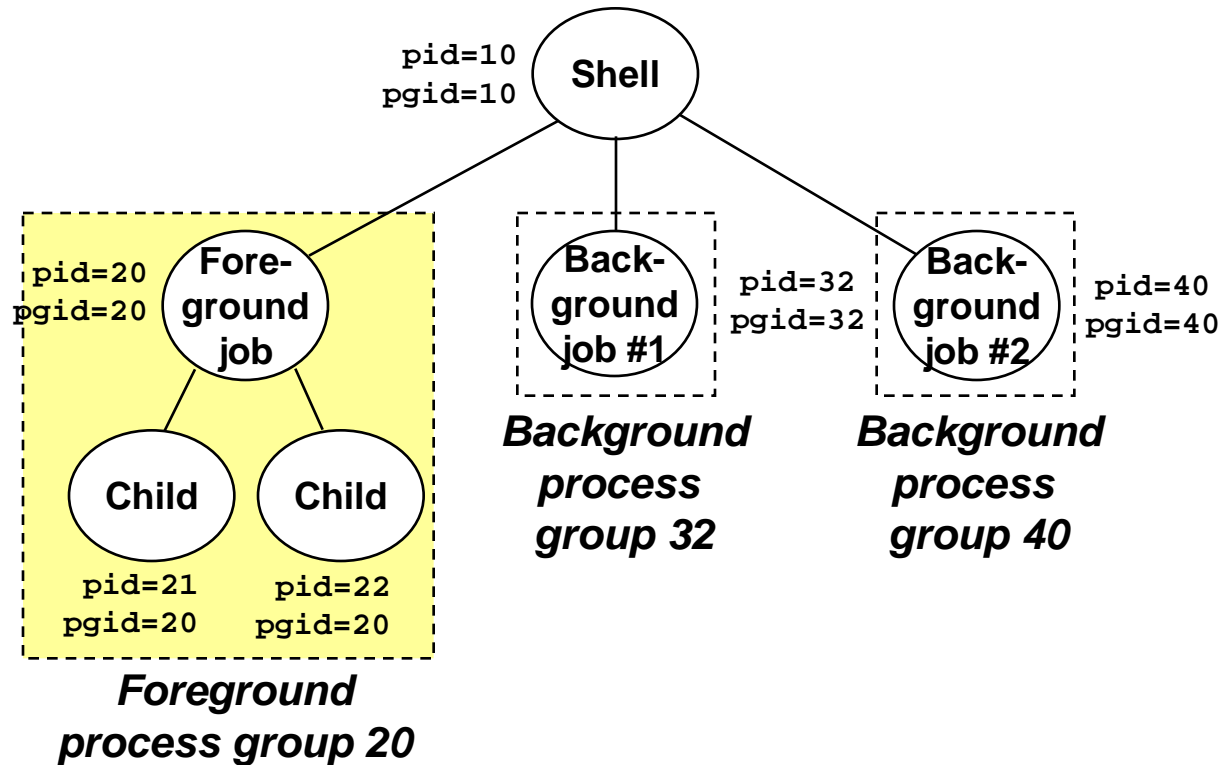
```
linux> ps
  PID TTY          TIME CMD
24788 pts/2        00:00:00 tcsh
24818 pts/2        00:00:02 forks
24819 pts/2        00:00:02 forks
24820 pts/2        00:00:00 ps
```

```
linux> kill -9 -24817
```

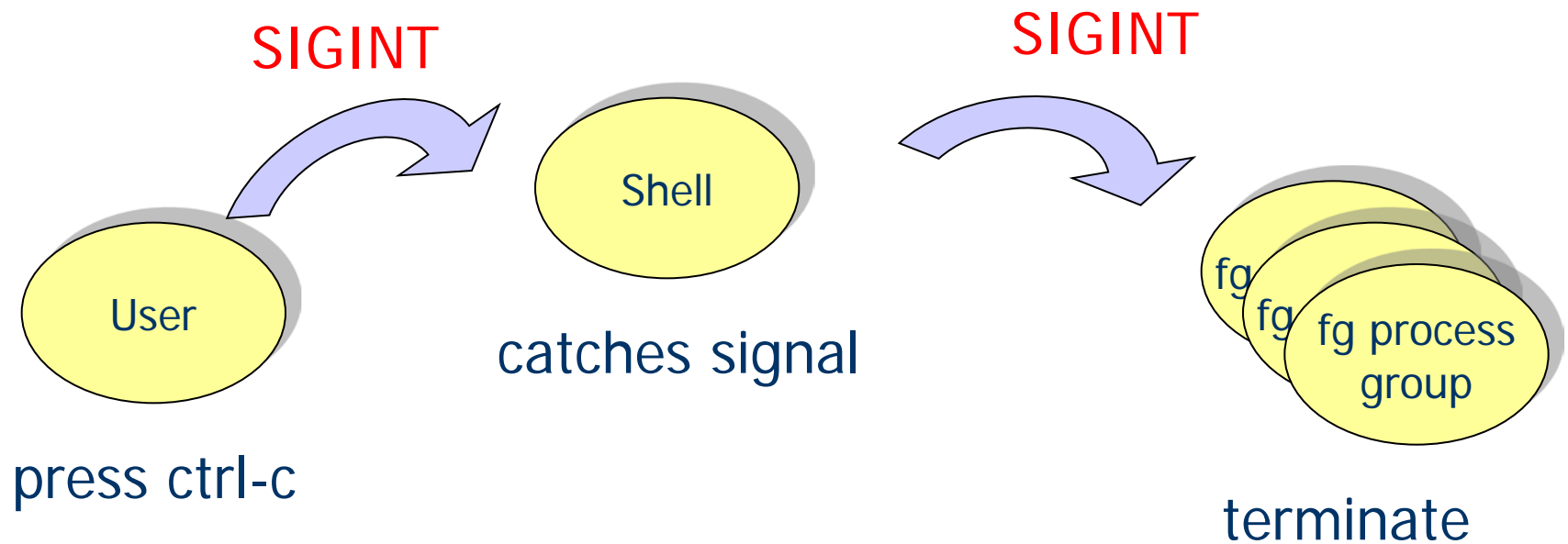
```
linux> ps
  PID TTY          TIME CMD
24788 pts/2        00:00:00 tcsh
24823 pts/2        00:00:00 ps
linux>
```

키보드로부터 시그널 보내기

- 키보드로 ctrl-c (ctrl-z)를 누르면 SIGINT (SIGTSTP) 시그널이 포그라운드 프로세스 그룹의 모든 작업으로 전송된다.
 - **SIGINT** – 기본동작은 각 프로세스를 모두 종료시킨다
 - **SIGTSTP** – 기본동작은 각 프로세스를 정지시킨다



키보드에서 CTRL-C 의 처리



Example of ctrl-c and ctrl-z

```
linux> ./forks 17
Child: pid=24868 pgrp=24867
Parent: pid=24867 pgrp=24867
<typed ctrl-z>
Suspended
linux> ps a
  PID TTY          STAT       TIME COMMAND
 24788 pts/2        S           0:00 -usr/local/bin/tcsh -i
 24867 pts/2        T           0:01 ./forks 17
 24868 pts/2        T           0:01 ./forks 17
 24869 pts/2        R           0:00 ps a
bass> fg
./forks 17
<typed ctrl-c>
linux> ps a
  PID TTY          STAT       TIME COMMAND
 24788 pts/2        S           0:00 -usr/local/bin/tcsh -i
 24870 pts/2        R           0:00 ps a
```


kill 시스템을 이용해서 시그널 보내기

```
void fork12()
{
    pid_t pid[N];
    int i, child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            while(1); /* Child infinite loop */

    /* Parent terminates the child processes */
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    /* Parent reaps terminated children */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

시그널 받기

- 커널이 예외처리 핸들러에서 돌아오고 있고, 제어권을 프로세스 p 로 넘겨줄 준비가 되었다고 가정해보자.
- 커널은 $pnb = pending \ \& \ \sim blocked$ 을 계산
 - 프로세스 p 의 블록되지 않은 시그널들을 표시
- If ($pnb == 0$)
 - 프로세스 p 의 논리적인 제어흐름상의 다음 인스트럭션으로 제어권을 이동.
- Else
 - pnb 에서 0이 아닌 k 번째 비트를 선택하고 프로세스 p 가 시그널 k 를 수신하도록 한다.
 - 시그널을 수신하면, 프로세스 p 는 시그널 처리 작업을 수행
 - pnb 의 모든 영이 아닌 비트 k 들에 대해 위 과정을 반복
 - 제어권을 프로세스 p 의 논리적 제어흐름 상의 인스트럭션으로 넘겨줌.

기본동작(Default Actions)

- 각 시그널 타입은 사전에 정의된 기본동작을 가진다:
 - 프로세스가 종료한다
 - 프로세스가 종료하고 **core**파일을 덤프
 - 프로세스가 **SIGCONT** 시그널에 의해 실행이 재개될 때까지 정지
 - 프로세스는 이 시그널을 무시
- 기본 동작은 *signal()* 함수를 이용해서 변경이 가능하다
 - **SIGSTOP** 과 **SIGKILL**은 예외

시그널 핸들러의 설치

- `signal` 함수는 `signum` 시그널의 수신과 관련된 기본 동작을 수정한다
 - `handler_t *signal(int signum, handler_t *handler)`
- 여러가지 `handler` 값
 - **SIG_IGN**: `signum` 타입 시그널을 무시
 - **SIG_DFL**: 시그널 타입 `signum`의 기본 동작으로 복귀
 - 그 외의 경우, `handler`는 ***signal handler의 주소가 된다***
 - ▶ 프로세스가 `signum` 시그널을 수신할 때 실행
 - ▶ 핸들러를 설치하는 기능을 수행
 - ▶ 이때 실행되는 핸들러는 시그널을 “붙잡는다 ***catching***” 또는 “***처리한다***”라고 부른다
 - ▶ 핸들러가 리턴문을 만나면, 제어권은 시그널에 의해 중단되었던 프로세스의 다음 명령으로 돌아간다.

시그널 핸들러 예제

```
void int_handler(int sig)
{
    printf("Process %d received signal %d\n",
           getpid(), sig);
    exit(0);
}

void fork13()
{
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler);

    . . .
}
```

```
linux> ./forks 13
Killing process 24973
Killing process 24974
Killing process 24975
Killing process 24976
Killing process 24977
Process 24977 received signal 2
Child 24977 terminated with exit status 0
Process 24976 received signal 2
Child 24976 terminated with exit status 0
Process 24975 received signal 2
Child 24975 terminated with exit status 0
Process 24974 received signal 2
Child 24974 terminated with exit status 0
Process 24973 received signal 2
Child 24973 terminated with exit status 0
linux>
```

Normal exit ?

시그널 핸들러의 이상동작

```
int ccount = 0;
void child_handler(int sig)
{
    int child_status;
    pid_t pid = wait(&child_status);
    ccount--;
    printf("Received signal %d from process %d\n",
           sig, pid);
    sleep(2);
}

void fork14()
{
    pid_t pid[N];
    int i, child_status;
    ccount = N;
    signal(SIGCHLD, child_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            sleep(1);
            /* Child: Exit */
            exit(0);
        }
    while (ccount > 0)
        pause(); /* Suspend until signal occurs */
}
```

대기시그널들은 큐에 들어가지 않는다

- 각 시그널의 대기여부를 표시하는데 하나의 비트만 할당되어있다
- 여러 프로세스가 이 시그널을 보내는 경우에는 더 큰 문제

Can you see the problem ?

큐를 사용하지 않는 문제점의 해결

■ 모든 종료된 작업들에 대해서 반드시 체크해야 한다

● 대개 `wait`를 반복해서 적용

모든
자식프로세스를
기다림

자식이 종료된 것이
없으면, 리턴값 0을
가지고 즉시 리턴

```
void child_handler2(int sig)
{
    int child_status;
    pid_t pid;
    while ((pid = waitpid(-1, &child_status, WNOHANG)) > 0) {
        ccount--;
        printf("Received signal %d from process %d\n", sig, pid);
    }
}

void fork15()
{
    . . .
    signal(SIGCHLD, child_handler2);
    . . .
}
```

Demo : sigtest4

연습문제 1. 시그널 핸들러

■ 이 프로그램의 출력을 쓰시오.

```
1  pid_t pid;
2  int counter = 2;
3
4  void handler1(int sig) {
5      counter = counter - 1;
6      printf("%d", counter);
7      fflush(stdout);
8      exit(0);
9  }
10
11 int main() {
12     signal(SIGUSR1, handler1);
13
14     printf("%d", counter);
15     fflush(stdout);
16
17     if ((pid = fork()) == 0) {
18         while(1) {};
19     }
20     kill(pid, SIGUSR1);
21     waitpid(-1, NULL, 0);
22     counter = counter + 1;
23     printf("%d", counter);
24     exit(0);
25 }
```


외부에서 생성된 이벤트를 처리하는 프로그램 (ctrl-c)

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

void handler(int sig) {
    printf("You think hitting ctrl-c will stop the bomb?\n");
    sleep(2);
    printf("Well...");
    fflush(stdout);
    sleep(1);
    printf("OK\n");
    exit(0);
}

main() {
    signal(SIGINT, handler); /* installs ctrl-c handler */
    while(1) {
    }
}
```

내부에서 발생한 이벤트를 처리하는 프로그램

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

int beeps = 0;

/* SIGALRM handler */
void handler(int sig) {
    printf("BEEP\n");
    fflush(stdout);

    if (++beeps < 5)
        alarm(1);
    else {
        printf("BOOM!\n");
        exit(0);
    }
}
```

```
main() {
    signal(SIGALRM, handler);
    alarm(1); /* send SIGALRM in
               1 second */

    while (1) {
        /* handler returns here */
    }
}
```

```
linux> a.out
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
bass>
```

비지역성 점프: `set jmp/long jmp`

- 제어를 임의의 위치로 이동할 수 있는 유저레벨의 강력한(그러나 위험한) 기법
 - 프로시저 콜/리턴 메커니즘을 효과적으로 벗어날 수 있는 방법
 - 에러 복원과 시그널 처리에 유용함
 - **깊이 연계된 함수** 콜로부터 즉각적인 리턴을 해야할 때
- `int set jmp(jmp_buf j)`
 - **long jmp** 실행 전에 먼저 호출되어야 한다
 - 다음에 나올 **long jmp** 호출시에 이용될 리턴 위치를 표시한다
 - 한번 호출하고, 한번 이상 리턴된다
- 구현방법:
 - 현재 시점의 레지스터들, 스택 포인터, **PC값**을 **jmp** 버퍼에 저장하는 방법으로 나중에 돌아올 프로그램상의 위치를 기억시킨다
 - **처음 호출할 때는 0을 리턴한다**

set jmp/long jmp (cont)

■ `void longjmp(jmp_buf j, int i)`

- **Meaning:**

- ▶ set jmp 점프 버퍼 j 가 기억하고 있는 리턴 구조로부터 리턴을 실행한다
- ▶ 최초 실행시에는 0을 리턴

- set jmp 후에 호출

- 한번 호출하면 리턴되지 않는다

■ longjmp 의 구현:

- 점프 버퍼 j 로부터 레지스터 컨텍스트를 복원한다
- %eax (리턴값) 를 i로 설정한다
- 점프 버퍼 j에 저장된 PC가 가리키는 위치로 이동한다

setjmp/longjmp Example

```
#include <setjmp.h>
jmp_buf buf;

main() {
    if (setjmp(buf) != 0)
        printf("back in main due to an error\n");
    else
        printf("first time through\n");
    p1(); /* p1 calls p2, which calls p3 */
}
...
p3() {
    <error checking code>
    if (error)
        longjmp(buf, 1)
}
```

배운 지식 총 적용 : ctrl-c 발생시 재시동하는 프로그램의 작성

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

sigjmp_buf buf;

void handler(int sig) {
    siglongjmp(buf, 1);
}

main() {
    signal(SIGINT, handler);

    if (!sigsetjmp(buf, 1))
        printf("starting\n");
    else
        printf("restarting\n");
}
```

```
while(1) {
    sleep(1);
    printf("processing...\n");
}
```

```
bass> a.out
starting
processing...
processing...
restarting ← Ctrl-c
processing...
processing...
restarting ← Ctrl-c
processing...
```

Summary

- 시그널은 프로세스 수준의 예외처리 방법을 제공한다
 - 유저 프로그램에서 만들수 있다
 - 시그널 핸들러를 정의할 수 있다
- 단점
 - 오버헤드가 크다
 - ▶ >10,000 clock cycles
 - ▶ 예외적인 경우에만 사용하는 것이 좋다
 - 시그널 큐가 없다
 - ▶ 해당 대기 시그널 타입에 대해 1비트만 사용
- 비지역성 점프는 프로세스 내에서 예외적인 제어흐름을 제공한다
 - 스택 기법을 이용