시스템 프로그래밍 (2016)

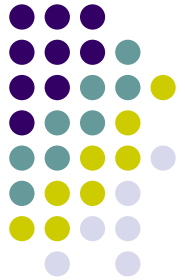
강의 10. 시그널과 비지역성 점프

강의 일정

주	날짜	강의실	날짜	실습실
1	9월 1일(목)	**소개 강의**	9월 6일(화)	**리눅스 개발환경 익히기 (VI, 쉘 기본명령어들)**
2	9월 8일(목)	**정수 표현 방법**	9월 13일(화)	GCC & Make, shell script
3	9월 15일(목)	**추석 휴강**	9월 20일(화)	**GDB 사용하기 1 (소스 수준 디버깅)**
4	9월 22일(목)	**실수 표현 방법**	9월 27일(화)	**Data lab**
5	9월 29일(목)	**어셈1 - 데이터이동**	10월 4일(화)	**어셈1 – move(실습),**
6	10월 6일(목)	**어셈2 – 제어문**	10월 11일(화)	**어셈2- 제어문 (실습)**
7	10월 13일(목)	**어셈3 – 프로시저**	10월 18일(화)	**어셈3-프로시저(실습)**
8	10월 20일(목)	**어셈보충/중간시험**	10월 25일(화)	**GDB 사용하기2 (어셈수준 디버깅)**
9	10월 27일(목)	**보안**(buffer overflow)	11월 1일(화)	Binary bomb 1
10	11월 3일(목)	**프로세스 1**	11월 8일(화)	Binary bomb 2
11	11월 10일(목)	**프로세스 2**	11월 15일(화)	Tiny shell 1
12	11월 17일(목)	**시그널**	11월 22일(화)	Tiny shell 2
13	11월 24일(목)	**동적메모리 1**	11월 29일(화)	Malloc lab1
14	12월 1일(목)	**동적메모리 2**	12월 6일(화)	Malloc lab2
15	12월 8일(목)	**기말시험**	12월 13일(화)	Malloc lab3

오늘의 주제들

1. 시그널의 개념, 송신, 수신
2. 프로세스 그룹과 시그널
3. 시그널 수신
4. 시그널을 block하기
5. 안전한 시그널 처리 1
6. 안전한 시그널 처리 2
7. 비지역성 점프

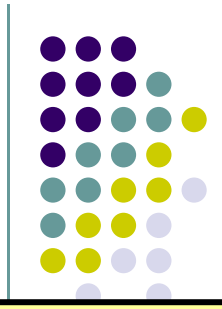


1. 시그널의 개념, 송신, 수신

시그널 – process 수준의 예외적 제어 흐름

- *시그널 signal* 은 어떤 이벤트가 시스템에 발생했다는 것을 프로세스에게 알려주는 짧은 메시지
  - 예외상황과 인터럽트를 커널이 처리하지만, 몇몇 예외는 사용자프로세스에게 신호로 드러나게 해준다. (예: SIGFPE, SIGILL, SIGSEGV)
  - 그외 다른 신호들은 상위수준SW이벤트에 해당함.(Ctrl-C→ SIGINT, 다른프로세스에 종료신호:SIGKILL, 자식프로세스의 종료를 부모프로세스에게: SIGCHLD)
  - 커널이 프로세스에게 보내준다 (간혹 다른 프로세스가 요청하기도 함, 항상 커널이 보내줌)
  - 서로 다른 시그널들은 정수 아이디로 구분한다 (1-30)
  - 시그널에 포함된 유일한 정보는 아이디와 시그널이 도착했다는 사실이다
  - Shell 상에서 kill 명령을 사용해서, 신호를 모두 보낼 수 있음(kill –n signnum pid/jobid)

signal list (그림 8.26)



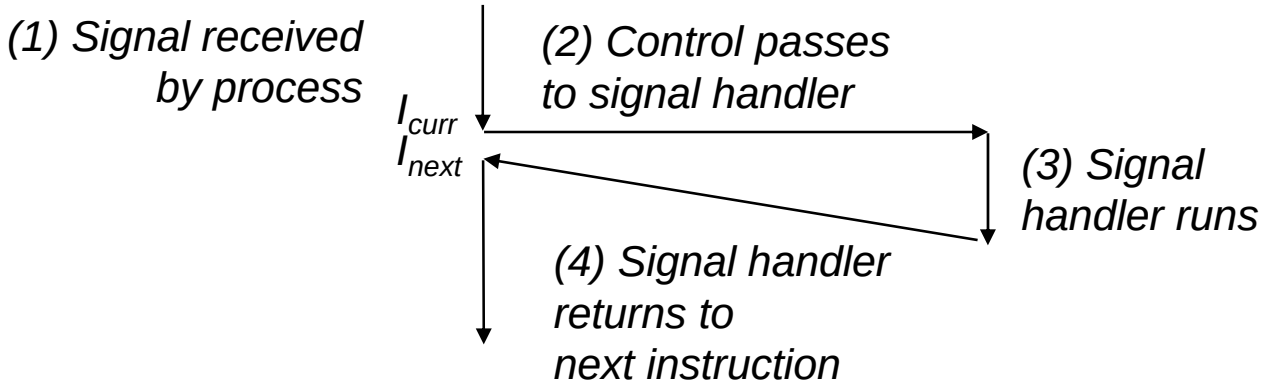
ID	Name	기본 동작	해당 이벤트 (see Figure 8.26)
1	SIGHUP	Terminate	Terminal line hangup
2	SIGINT	Terminate	키보드 인터럽트 (Ctrl-C)
4	SIGILL	Terminate	illegal instruction
8	SIGFPE	Terminate	floating point예외(divide by zero...)
9	SIGKILL	Terminate	프로그램을 종료시킨다 (무시 또는 변경불가)
10,12	SIGUSR1/2	Terminate	사용자 정의 신호1/2
11	SIGSEGV	Terminate & Dump	메모리 참조 오류(세그멘테이션폴트)
14	SIGALRM	Terminate	타이머 시그널Timer signal
17	SIGCHLD	Ignore	자식이 정지 또는 종료함
18	SIGCONT	Ignore	정지된 process 계속 수행 (bg after Ctrl-Z)
19	SIGSTOP	Stop till next SIGCONT	터미널에서 오지 않은 stop signal
20	SIGTSTP	Stop till next SIGCONT	터미널에서 온 stop signal(Ctrl-Z)

시그널의 송신 Sending a signal

- 커널은 목적지 프로세스의 컨텍스트 내 일부 상태를 갱신하는 방법으로 시그널을 목적지 프로세스에 보낸다
- 커널은 다음 두 이유중 하나로 시그널보냄 :
  - 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(a process can send a signal to itself)

시그널의 수신 Receiving a signal

- 목적지 프로세스가 시그널을 받을 때, 어떤 형태로든 반응을 하도록 커널에 의해 요구될 때, 시그널을 받는다고 한다.
- 세가지 반응 :
  1. 무시 Ignore the signal (do nothing)
  2. 목적지 프로세스를 종료 (코드 덤프할 수도).
  3. *시그널 핸들러* 라고 부르는 유저레벨 함수를 실행하여 시그널을 잡는다(catch)
     - 비동기형 인터럽트에 대한 응답으로 호출되는 인터럽트 핸들러 방식과 유사



signal - pending(대기)

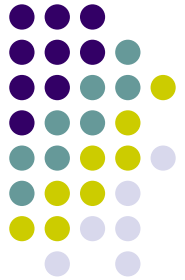
- 전송하였지만, 아직 수신되지 않은 시그널은 “대기하고 있다(pending)”고 한다
  - 특정 시그널에 대해서 at most one 대기 시그널
  - 중요 : 시그널은 큐에 들어가지 않는다
    - 만일 어떤 프로세스가 k타입의 대기 시그널을 가지고 있다면, 다음에 이 프로세스로 전달되는 k타입의 시그널들은 무시된다.
- pending 신호는 기껏해야 한번 수신될 수 있다.
  - 커널은 대기 시그널들을 나타내기 위하여 bit vector 사용.

signal – blocked(블록)

- 프로세스는 특정 시그널의 수신을 블록할 수 있다.(시그널의 거절)(blocked 구조) → interrupt disable과 유사
  - 블록된 시그널들은 전달될 수 있지만, 이 시그널이 풀릴 때(unblocked)까지는 수신될 수 없다.
  - 프로세스에서 블록될 수 없는 유일한 시그널은 SIGKILL이다.

시그널의 개념- pending/blocked bit

- 커널은 각 프로세스 별로 pending과 blocked 비트 벡터 유지
  - 각 process의 context 내에 유지
  - pending – 대기 시그널들을 표시
    - 커널은 타입 k 시그널이 배달되면, pending[k]← 1로 설정
    - 커널은 타입 k시그널이 수신처리되면, pending[k] ← 0
  - blocked – 블록된 시그널들을 표시
    - sigprocmask 함수로 1/0 설정
    - signal mask라고도 한다.
    - 배달은 되지만, 수신은 되지 않음.

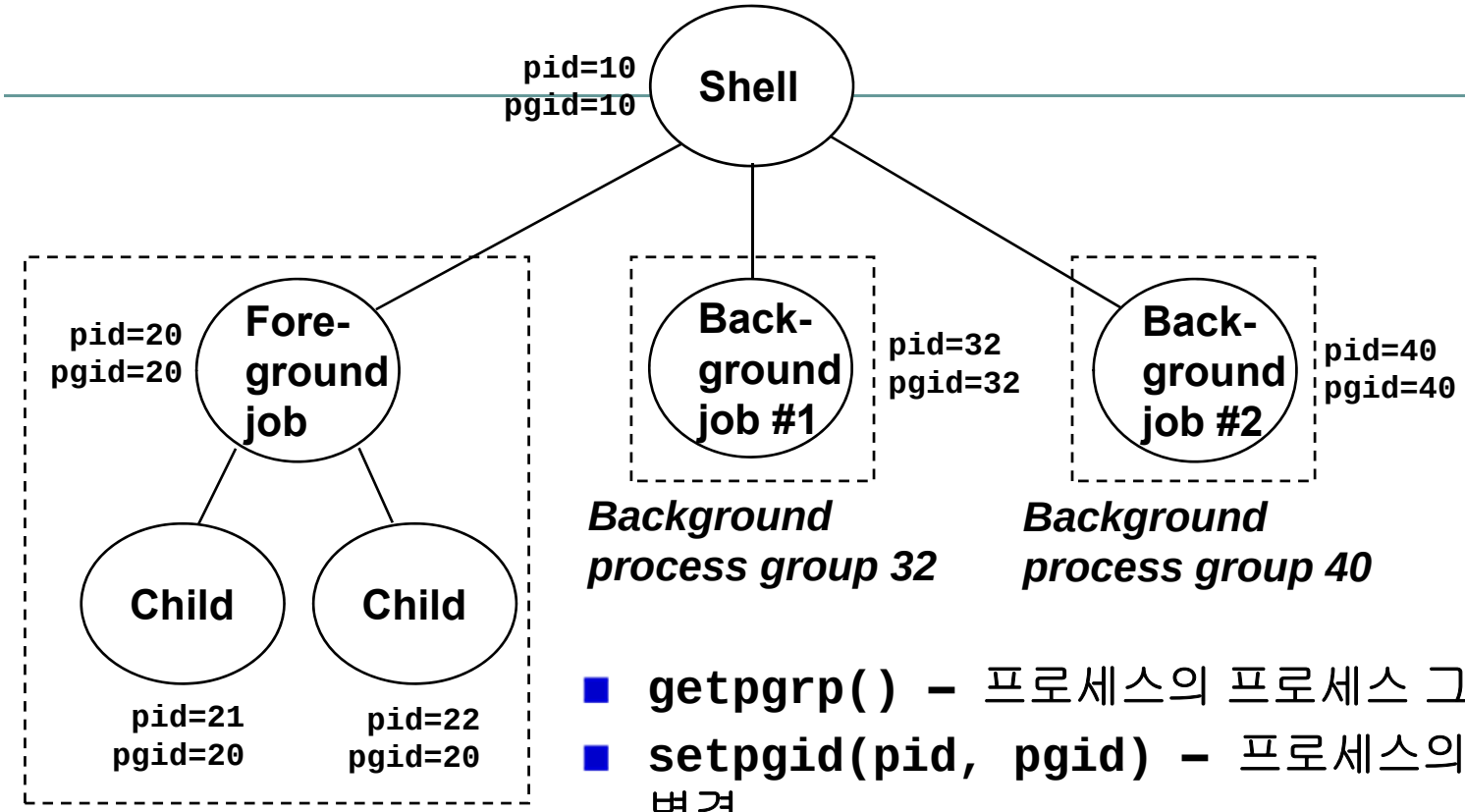


2. 프로세스 그룹과 시그널

프로세스 그룹

- 각 프로세스는 하나의 프로세스 그룹에 속한다
- process group IP (pgid), getpgrp() 호출로 알 수 있다.
- 자식프로세는 부모 프로세스와 동일 그룹에 속함(디폴트)
- 자신이나, 다른 프로세스의 pgid를 setgpid()로 변경가능.

프로세스 그룹



- **getpgrp() –** 프로세스의 프로세스 그룹을 리턴
- **setpgid(pid, pgid) –** 프로세스의 그룹을 변경

  * **pid==0** ➔ 현재프로세스의 **pid**사용

  * **pgid==0** ➔ **pid**로 정해진 프로세스의 **pid**가 **pgid**로 사용됨.

  * **setpgid(0,0)** ➔ **pid=20**프로세스 호출시, 자신의 **pgid**도 **20**이 됨.

kill 명령을 이용한 시그널 보내기

- /bin/kill 시스템 콜은 프로세스 또는 프로세스 그룹에 임의의 시그널보냄

- 예제
  - kill –9 24818
    - SIGKILL 을 process 24818로 보냄
  - kill –9 –24817
    - 프로세스 그룹 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>
```

Getpgrp() & setpgid()

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

main()
{
  printf("getpgrp = %d\n", getpgrp());
  fork();
  printf("getpgrp after fork = %d\n", getpgrp());
  fork();
  printf("getpgrp after fork & fork = %d\n",
getpgrp());
  // setpgid(0,0); 자신의 pid로 group id 설정
  fork();
  printf("getpgrp after fork & fork & fork = %d\n",
getpgrp());
}
```

Getpgrp() & setpgid

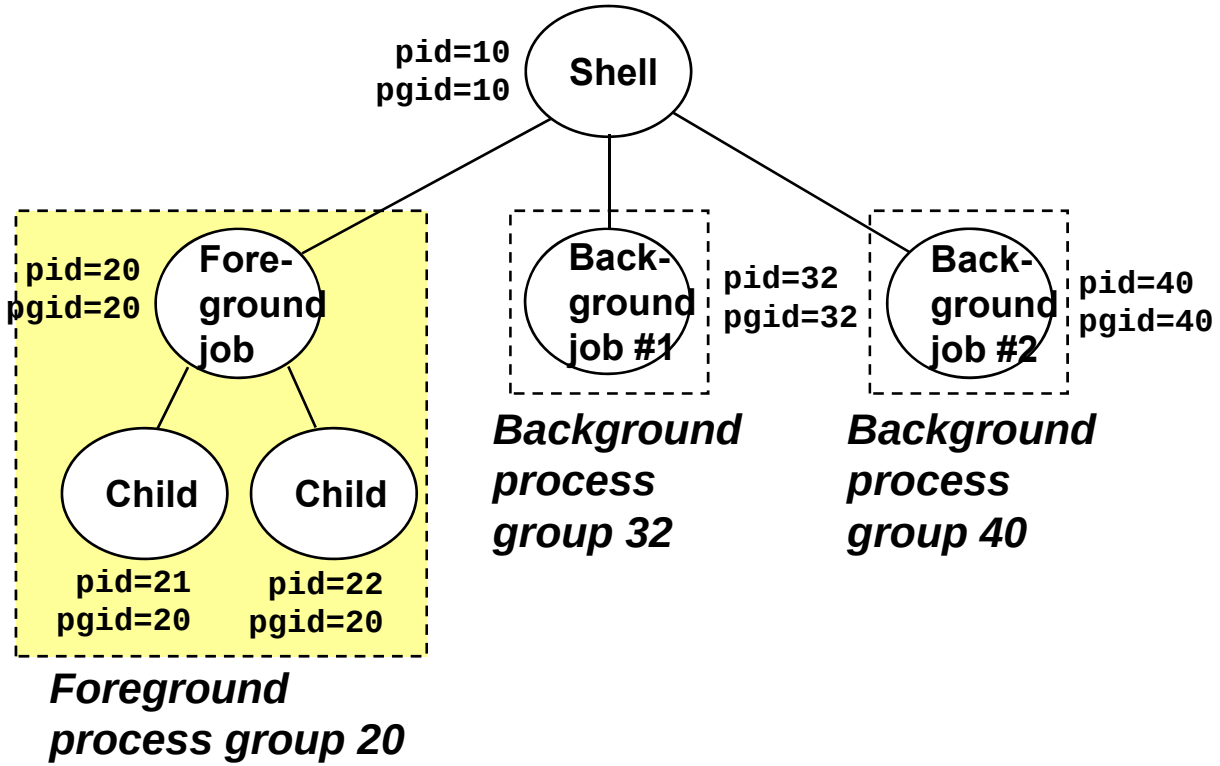

```
#include <stdio.h>
#include <unistd.h>

main()
{
  printf("getpgrp = %d\n", getpgrp());
  pid_t pid = fork();
  //자신의 pid로 group id 설정
  if(pid==0) setpgid(0,0);
  printf("getpgrp after fork = %d\n", getpgrp());
  fork();
  printf("getpgrp after fork & fork = %d\n",
getpgrp());
fork();
  printf("getpgrp after fork & fork & fork = %d\n",
getpgrp());
}
```

```
sun@...rt...  .../Dropbox/assem
getpgrp after ...k & fork & fork = 2448
getpgrp = 2448
getpgrp after fork = 2450
getpgrp after fork & fork = 2450
getpgrp after fork & fork & fork = 2450
getpgrp after fork & fork & fork = 2450
getpgrp = 2448
getpgrp after fork = 2448
getpgrp after fork & fork = 2448
getpgrp after fork & fork & fork = 2448
getpgrp after fork & fork & fork = 2448
getpgrp = 2448
getpgrp after fork = 2448
getpgrp after fork & fork = 2448
getpgrp after fork & fork & fork = 2448
getpgrp after fork & fork & fork = 2448
getpgrp = 2448
getpgrp after fork = 2450
getpgrp after fork & fork = 2450
getpgrp after fork & fork & fork = 2450
getpgrp after fork & fork & fork = 2450
getpgrp = 2448
getpgrp after fork = 2450
--More--
```

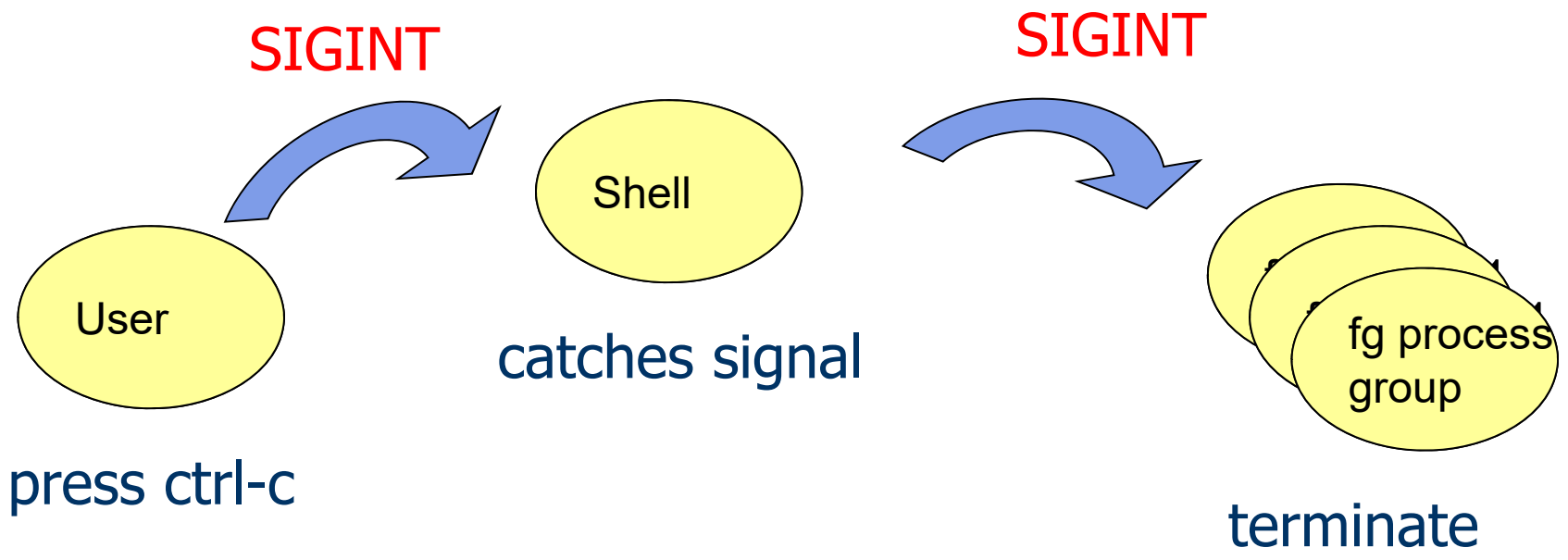
// child subgroup을 만들게 됨.

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

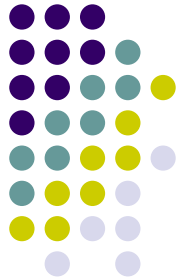


Example of `ctrl-c` and `ctrl-z`

```
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 함수를 이용해서 시그널 보내기

```
void forknkill()
{
    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);
    }
}
```



3. 시그널 수신

시그널의 수신

- 커널이 예외처리 핸들러에서 돌아오고 있고, 제어권을 프로세스 p로 넘겨줄 준비가 되었다고 가정해보자.
- 커널은 pnb = pending & ~blocked 을 계산
  - 프로세스 p의 블록되지 않은 시그널들을 표시
- If (pnb == 0)
  - 프로세스 p의 논리적인 제어흐름상의 다음 인스트럭션으로 제어권을 이동.
- Else
  - pnb 에서 0이 아닌 가장 작은 k번째 중요한 비트를 선택하고 프로세스 p가 시그널 k를 수신하도록 한다.
  - 시그널을 수신하면, 프로세스 p는 다른 일들을 수행한다
  - pnb의 모든 영이 아닌 비트 k들에 대해 위 과정을 반복
  - 제어권을 프로세스 p의 논리적 제어흐름 상의 인스트럭션으로 넘겨줌.

기본동작(Default Actions)

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

시그널 핸들러의 설치

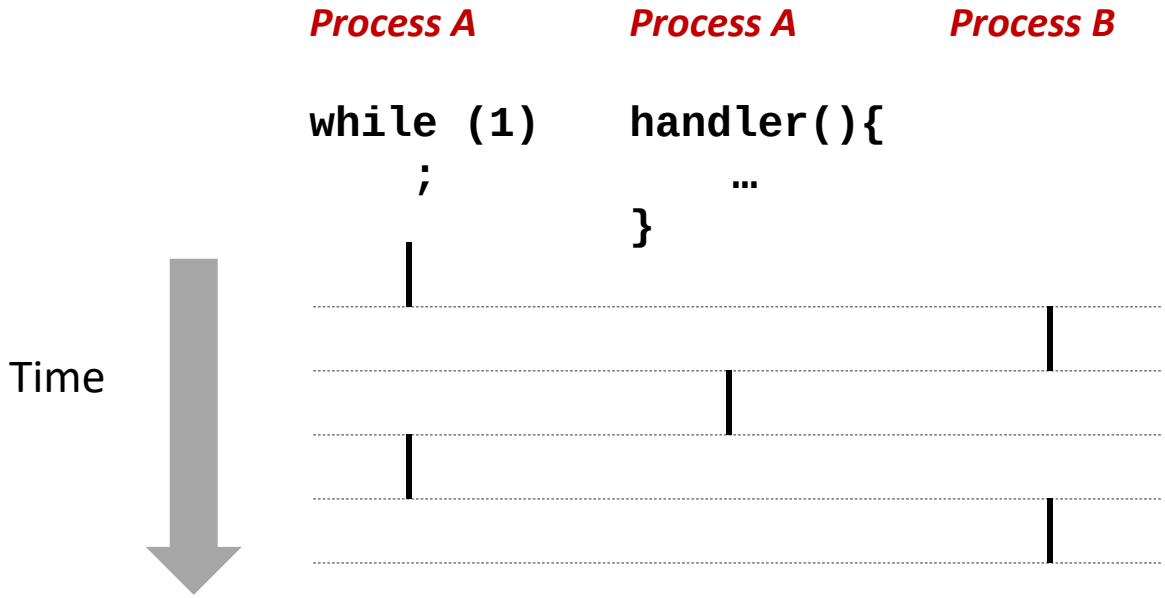
- `signal` 함수는 `signum` 시그널의 수신과 관련된 기본 동작을 수정한다
  - `handler_t *signal(int signum, handler_t *handler)`
- 여러가지 `handler` 값
  - SIG_IGN: `signum` 타입 시그널을 무시
  - SIG_DFL: 시그널 타입 `signum` 의 기본 동작으로 복귀
  - 그 외의 경우, handler는 *signal handler의 주소가 된다*
    - Called when process receives signal of type `signum`
    - Referred to as "*installing*" the handler.
    - 이때 실행되는 핸들러는 시그널을 "붙잡는다*catching*" 또는 "처리한다" 라고 부른다
    - 핸들러가 리턴문을 만나면, 제어권은 시그널에 의해 중단되었던 프로세스의 다음 명령으로 돌아간다.

signal handler 예제

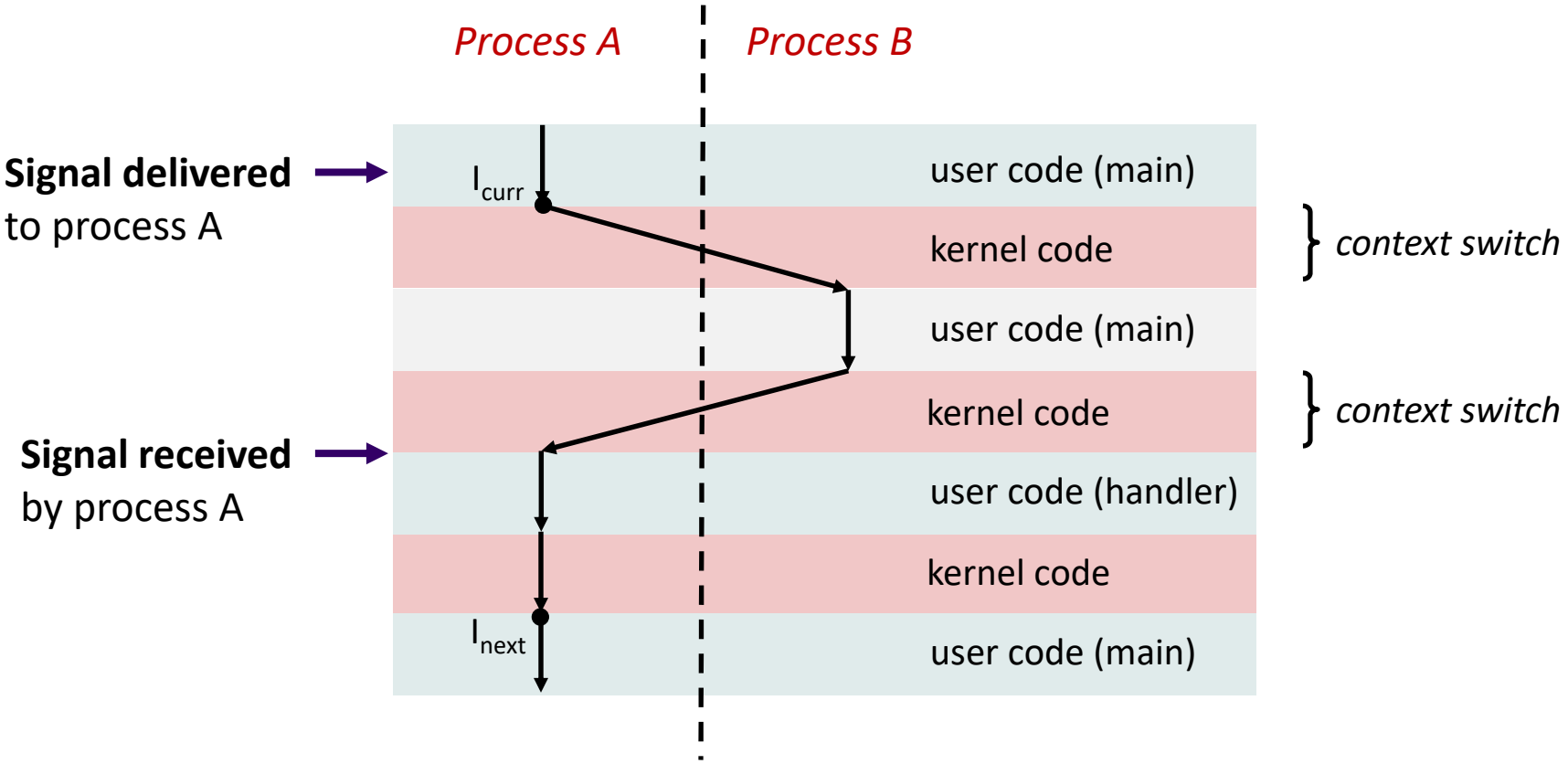
```
void sigint_handler(int sig) /* SIGINT handler */
{
    printf("So you think you can stop the bomb with ctrl-c, do you?₩n");
    sleep(2);
    printf("Well...");
    fflush(stdout);
    sleep(1);
    printf("OK. :-)₩n");
    exit(0);
}
int main()
{
    /* Install the SIGINT handler */
    if (signal(SIGINT, sigint_handler) == SIG_ERR)
        unix_error("signal error");

    /* Wait for the receipt of a signal */
    pause();
    return 0;
}
```

시그널 핸들러는 concurrent하게 수행

- 시그널 핸들러는 별도의 논리흐름(process는 아님)으로 main()과 동시에 (concurrently) 수행됨.



Concurrent 흐름 – 시그널 핸들러에 대한 다른 관점



Handler 수행 중에 다른 signal도 착하면?

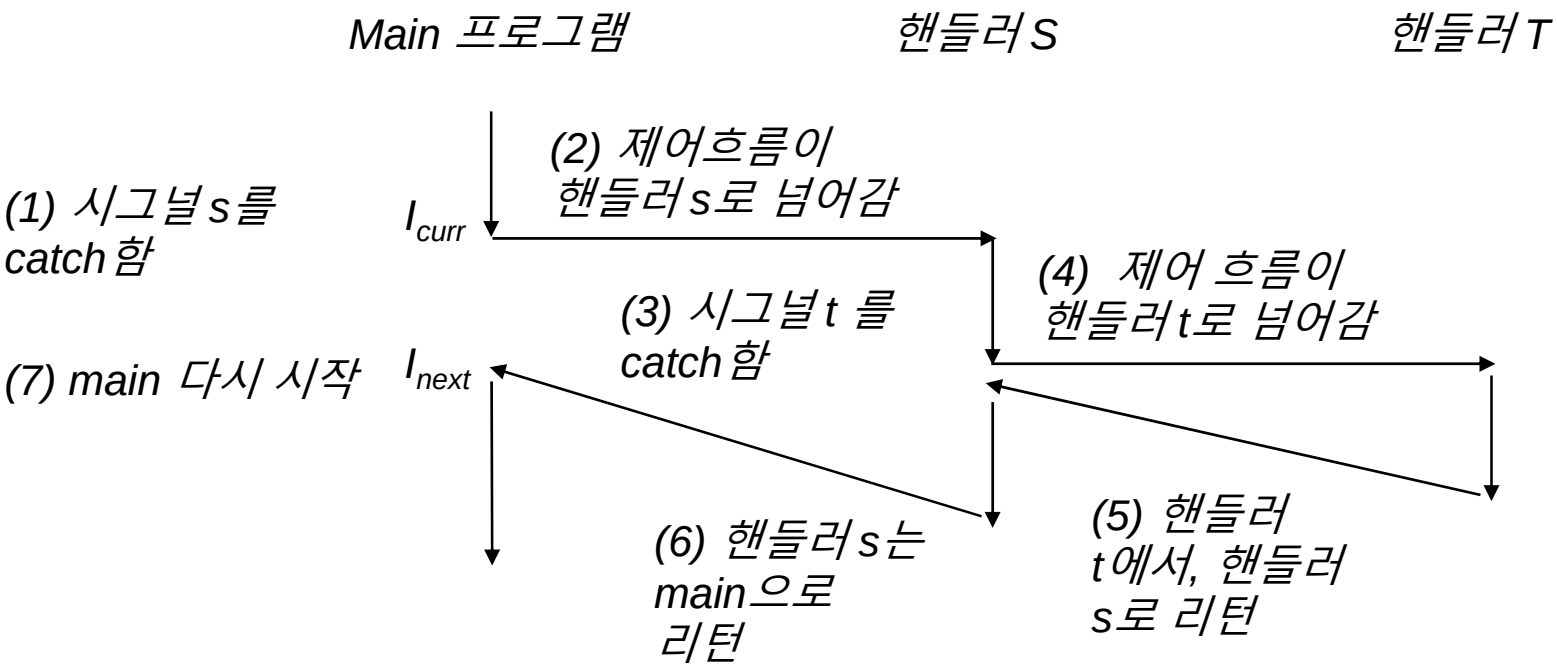
SIGINT 핸들러 수행 중에,
SIGFPE가 생기면,
그 시그널처리함.
동일시그널이 아니면,
pending되거나, 무시되지
않음.

```
#include <stdio.h>   // ddiv.c
#include <signal.h>
void handler(int sig){
                printf("div by zero\n");
                int a = 50 / 0;
}
main(){
signal(SIGINT, handler);

printf("process - %d\n",getpid());
fflush(stdout);
while(1);

printf("process after div .by 0 - %d\n",getpid());
fflush(stdout);


}
```

```
f836                    handlerinterno.c.txt  signal2
sun@sun-VirtualBox:~/Dropbox/assem/9thwk$ ./ddiv
process - 2570
^Cdiv by zero
부동 소수점 예외 (core dumped)
sun@sun-VirtualBox:~/Dropbox/assem/9thwk$
```

중첩된 시그널 핸들러

- 핸들러는 다른 핸들러에 의해 중단 가능



4. 시그널을 BLOCK 하기

시그널의 block, unblock

- 암묵적인 block메커니즘
  - 커널은 현재 처리중인 pending 시그널은 block함
  - 예: SIGINT 핸들러는 다른 SIGINT 시그널로 중단되지 않음.
- 명시적인 block, unblock 메커니즘
  - `sigprocmask` 함수 사용
  - `sigprocmask(int how, const sigset_t *set, sigset_t *oldset)`
  - 호출한 `thread`의 `signal mask`를 꺼내오거나 변경함
  - `signal mask`란 현재 호출자에 대해서 `block`된 시그널들의 집합.
  - `oldset != NULL`, 이전 `blocked signals set` → `*oldset`에 저장
  - `set == NULL, signal mask`는 변경 없음.
  - `how: SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK`
  - `SIG_BLOCK:  current_set UNION set → new set`
  - `SIG_BLOCK:  current_set MINUS set → new set`
  - `SIG_BLOCK:  set → new set`

시그널의 block, unblock

- 지원 함수들
  - `sigemptyset(sigset_t *set)` – 새로 빈 set 구성
  - `sigfillset(sigset_t *set)` – 모든 시그널을 포함
  - `sigadd/delset(signset_t *set, int signum)` - set 에서 특정 signum 시그널을 추가/삭제함.
  - `sigismember(sigset_t *set, int signum)` – signum 이 set의 구성원 여부 리턴.

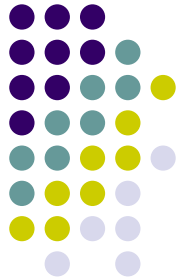
임시로 시그널 block하기

```
sigset_t mask, prev_mask;

Sigemptyset(&mask);
Sigaddset(&mask, SIGINT);

/* Block SIGINT and save previous blocked set */
Sigprocmask(SIG_BLOCK, &mask, &prev_mask);

  ⋮/* Code region that will not be interrupted by SIGINT */

/* Restore previous blocked set, unblocking SIGINT */
Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
```

signal(__ , SIG_IGN)과 blocking과 차이?

- signal(SIGALARM, SIG_IGN)는 alarm을 무시함.
- blocking(sigprocmask()사용)은 시그널을 대기시킴. queue에 들어감.
- blocked 시그널은 잃어버리지 않으나, ignored signal은 잃어버림.

http://stackoverflow.com/questions/5730975/difference-in-handling-of-signals-in-unix



5. 안전한 시그널 처리 1

안전한 시그널 처리 (Safe Signal Handling)

- 핸들러는 main과 함께 동시 수행되면서, 같은 전역 자료 구조를 공유하므로, 문제소지
  - 공유 자료 구조가 오염될 수 있다.

- 이런 문제를 회피할 가이드라인이 필요함.

안전한 핸들러 작성 가이드라인

- G0: 가능한 단순하게 작성
  - e.g., 전역 플래그 설정하고 바로 리턴...
- G1: 핸들러 내에서는 async-signal-safe (비동기시그널에 안전한)함수만 부른다.(그림 8.33)
  - `printf, sprintf, malloc, exit` 은 안전하지 않다
  - 대신 write() 사용.
- G2: `errno` 를 저장및복구하라
  - 다른 핸들러들 그 핸들러 수행에 의한 errno로 사용되지않게...
- G3: 공유자료 구조 보호를 위해, 잠시 시그널들을 block한다
  - 가능한 오염 방지
- G4: 전역변수를 volatile로 선언함.
  - 항상 해당 변수를 메모리에서 읽어오도록 함. 컴파일러의 최적화 방지. 보통은 레지스터에 저장하게 됨. handler가 수정하는 값을 읽을 수 없게 됨.
- G5: 전역 플래그는 `volatile sig_atomic_t flag;` 로선언
  - *flag*: 읽기나 쓰기 (e.g. flag = 1, flag++은 해당 안됨.)
  - 이렇게 선언된 플래그는 다른 전역변수처럼 보호되지않아도 됨.

안전한 출력 함수

- csapp.c에 있는 reentrant SIO (Safe I/O library) 사용 권장.
  - `ssize_t sio_puts(char s[]) /* Put string */`
  - `ssize_t sio_putl(long v)   /* Put long */`
  - `void sio_error(char s[])   /* Put msg & exit */`

```
void sigint_handler(int sig) /* Safe SIGINT handler */
{
    Sio_puts("So you think you can stop the bomb with ctrl-c, do you?₩n");
    sleep(2);
    Sio_puts("Well...");
    sleep(1);
    Sio_puts("OK. :-)₩n");
    _exit(0);
}
```

sigintsafe.c

안전한 시그널 처리

```
int ccount = 0;
void child_handler(int sig) {
    int olderrno = errno;
    pid_t pid;
    if ((pid = wait(NULL)) < 0)
        Sio_error("wait error");
    ccount--;
    Sio_puts("Handler reaped child ");
    Sio_putl((long)pid);
    Sio_puts(" ₩n");
    sleep(1);
    errno = olderrno;
}

void fork14() {
    pid_t pid[N];
    int i;
    ccount = N;
    Signal(SIGCHLD, child_handler);

    for (i = 0; i < N; i++) {
        if ((pid[i] = Fork()) == 0) {
            Sleep(1);
            exit(0);  /* Child exits */
        }
    }
    while (ccount > 0) /* Parent spins */
        ;
}
```

forks.c

- pending시그널은 대기 큐에 들어가지않음
  - 각 비트별 1비트 뿐
  - …기껏해야 대기 시그널은 하나 밖에(유형별로)
- 이벤트 계수에 시그널을 사용할 수 없음.
  - 예:종료된 자식의 수를 세거나.... ??

```
whaleshark> ./forks 14
Handler reaped child 23240
Handler reaped child 23241
```

안전한 시그널 처리

- 모든 종료 자식 프로세스를 기다려야....
  - 모든 종료 자식 수확을 위해 loop에 wait() 사용

```
void child_handler2(int sig)
{
    int olderrno = errno;
    pid_t pid;
    while ((pid = wait(NULL)) > 0) {
        ccount--;
        Sio_puts("Handler reaped child ");
        Sio_putl((long)pid);
        Sio_puts(" ₩n");
    }
    if (errno != ECHILD)
        Sio_error("wait error");
    errno = olderrno;
}
```

```
whaleshark> ./forks 15
Handler reaped child 23246
Handler reaped child 23247
Handler reaped child 23248
Handler reaped child 23249
Handler reaped child 23250
whaleshark>
```

이식성있는(portable) 시그널처리

- UNIX버전에 따라 시그널 처리 방식 차이
  - 어떤 버전은 시그널 catch후에 디폴트 동작이 되거나
  - 어떤 인터럽트된 시스템콜은 리턴시, errno == EINTR
  - 어떤 시스템은 처리 중인 유형의 시그널 block 불가
- 해결책: `sigaction`

```
handler_t *Signal(int signum, handler_t *handler)
{
    struct sigaction action, old_action;

    action.sa_handler = handler;
    sigemptyset(&action.sa_mask); /* Block sigs of type being handled */
    action.sa_flags = SA_RESTART; /* Restart syscalls if possible */

    if (sigaction(signum, &action, &old_action) < 0)
        unix_error("Signal error");
    return (old_action.sa_handler);
}
```

csapp.c

6. 안전한 시그널 처리 2 - 레이스 조건

레이스 조건, 레이스 해저드

- 레이스조건, 레이스해저드란 전자회로, 소프트웨어나 기타 시스템에서 제어 가능하지 않은 이벤트들의 순서나 타이밍에 따라 출력값이 달라지는 것을 의미함.
- 이벤트가 제 순서로 일어나지 않을 때 버그가 됨.
- 이 용어는 출력에 영향을 미치는 두 시그널이 경쟁한 결과로 출력이 결정된다고 해서...

레이스(race) 문제 회피 위한 동기화된 흐름

- 부모가 자식보다 먼저 실행된다는 가정때문에, 미묘한 동기화 오류를 갖는 단순 쉘

```
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;

    Sigfillset(&mask_all);
    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        if ((pid = Fork()) == 0) { /* Child */
            Execve("/bin/date", argv, NULL);
        }
        Sigprocmask(SIG_BLOCK, &mask_all, &prev_all); /* Parent */
        addjob(pid);  /* Add the child to the job list */
        Sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
    exit(0);
}
```

procmask1.c

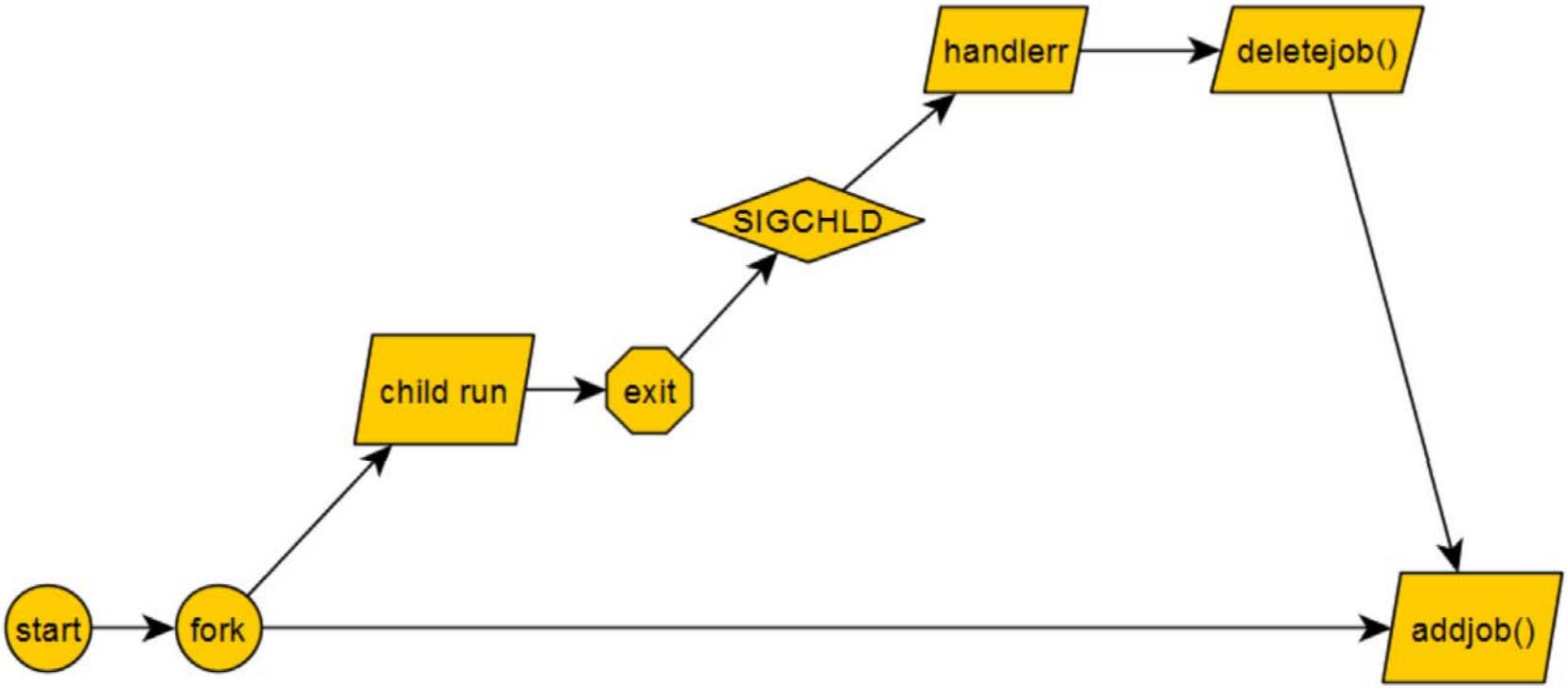
레이스(race) 문제 회피 위한 동기화된 흐름

- SIGCHLD 핸들러(단순 쉘 용)

```
void handler(int sig)
{
    int olderrno = errno;
    sigset_t mask_all, prev_all;
    pid_t pid;

    Sigfillset(&mask_all);
    while ((pid = waitpid(-1, NULL, 0)) > 0) { /* Reap child */
        Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
        deletejob(pid); /* Delete the child from the job list */
        Sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
    if (errno != ECHILD)
        Sio_error("waitpid error");
    errno = olderrno;
}
```

procmask1.c

race 조건이 되는 sequence



race조건없는 main()

fork()이전 부터
block SIGCHLD

```
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, mask_one, prev_one;

    Sigfillset(&mask_all);
    Sigemptyset(&mask_one);
    Sigaddset(&mask_one, SIGCHLD);
    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) { kj1
        Sigprocmask(SIG_BLOCK, &mask_one, &prev_one); /* Block SIGCHLD */
        if ((pid = Fork()) == 0) { /* Child process */
            Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
            Execve("/bin/date", argv, NULL);
        }
        Sigprocmask(SIG_BLOCK, &mask_all, NULL); /* Parent process */
            addjob(pid);  /* Add the child to the job list */
        Sigprocmask(SIG_SETMASK, &prev_one, NULL);  /* Unblock SIGCHLD */
    }
    exit(0);
}
```

procmask2.c

슬라이드 48

kj1 kyoungson jhang, 2016-11-11

명시적으로 시그널을 기다림

- while(1)으로 명시적으로 SIGCHLD를 기다리는 프로그램을 위한 핸들러 두개

```
volatile sig_atomic_t pid;

void sigchld_handler(int s)
{
    int olderrno = errno;
    pid = Waitpid(-1, NULL, 0); /* Main is waiting for nonzero pid */
    errno = olderrno;
}

void sigint_handler(int s)
{
}
```

waitforsignal.c

Linux 쉘에서 fork통한 만든 자식 프로세스, foreground job이 만들어지고,
종료를 기다렸다가, 다음 명령어 받아들이려면, SIGCHLD기다려야 함.
그런 상황에 적합한 예제임. 그런데...

명시적으로 SIGCHLD기다림

foreground 작업이 종료되기를 기다리는 쉘과 유사함.

```
int main(int argc, char **argv) {
    sigset_t mask, prev;
    Signal(SIGCHLD, sigchld_handler);
    Signal(SIGINT, sigint_handler);
    Sigemptyset(&mask);
    Sigaddset(&mask, SIGCHLD);

    while (1) {
            Sigprocmask(SIG_BLOCK, &mask, &prev); /* Block SIGCHLD */
            if (Fork() == 0) /* Child */
          exit(0);
            /* Parent */
            pid = 0;
            Sigprocmask(SIG_SETMASK, &prev, NULL); /* Unblock SIGCHLD */

            /* Wait for SIGCHLD to be received (wasteful!) */
            while (!pid)
          ;
            /* Do some work after receiving SIGCHLD */
        printf(".");
    }
    exit(0);
}
```

waitforsignal.c

명시적으로 SIGCHLD기다림

- 앞의 예는 정확하지만, 시간 낭비가 심함
- 다른 방법은?:

```
while (!pid)  /* Race! */
    pause();
```

```
while (!pid) /* Too slow! */
    sleep(1);
```

- race: SIGCHLD가 if(!pid) 후, pause() 전에 도착시 → pid!=0되었지만,pause()에 들어가고, 더이상 시그널이 생성안되면, 계속 pause

- Solution: `sigsuspend`

sigsuspend()로 시그널대기하기

- `int sigsuspend(const sigset_t *mask)`

- 아래와 같은 세 동작의 atomic 버전(끊김없이 한 번에 수행됨)에 해당함:

```
sigprocmask(SIG_BLOCK, &mask, &prev);
pause();
sigprocmask(SIG_SETMASK, &prev, NULL);
```

sigsupend()는 현재 시그널 매스크 집합을 mask로 잠시 교체하고,
다른신호들을 기다리는 동작과 (신호 도착) 후에 다시 이전 set(prev)로
시그널 매스크를 교체/복구하는 것임. 세동작이 원소동작으로 이루어지게..

sigsuspend()로 시그널대기하기

```c
int main(int argc, char **argv) {
    sigset_t mask, prev;
    Signal(SIGCHLD, sigchld_handler);
    Signal(SIGINT, sigint_handler);
    Sigemptyset(&mask);
    Sigaddset(&mask, SIGCHLD);

    while (1) {
        Sigprocmask(SIG_BLOCK, &mask, &prev); /* Block SIGCHLD */
        if (Fork() == 0) /* Child */
            exit(0);

       /* Wait for SIGCHLD to be received */
        pid = 0;
        while (!pid)
            Sigsuspend(&prev);

       /* Optionally unblock SIGCHLD */
        Sigprocmask(SIG_SETMASK, &prev, NULL);
        /* Do some work after receiving SIGCHLD */
        printf(".");
    }
    exit(0);
}
```

sigsuspend.c

SIGFPE 무시할 수 있나?

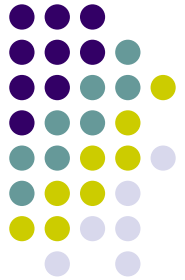
SIG_IGN
으로는
무시할수없다.

```
#include <stdio.h>   // div.c
#include <signal.h>
Void handler(int sig){
                printf("div by zero\n");
//              signal(SIGFPE, SIG_IGN);
}
main()
{   //signal(SIGFPE, handler);
signal(SIGFPE, SIG_IGN);

printf("process - %d\n",getpid());
fflush(stdout);
int a = 50  / 0;
printf("process after div .by 0 - %d\n",getpid());
fflush(stdout);
}
```

```
sun@sun-VirtualBox:~/Dropbox/assem/9thwk$ gcc -o div div.c
div.c: In function 'main':
div.c:17:13: warning: division by zero [-Wdiv-by-zero]
sun@sun-VirtualBox:~/Dropbox/assem/9thwk$ ./div
process - 2533
부동 소수점 예외 (core dumped)
sun@sun-VirtualBox:~/Dropbox/assem/9th
```

```
process - 2567
div by zero
div by zero
div by zero
div by zero
div by zero
div by zero
div by zero
div by zero
div by zero
div by zero
div by zero
div by zero
div by zero
div by zero
div by zero
div by zero
div by zero
div by zero
div by zero
div by zero
--More--
```

Handler
사용하면,
계속 시그널이
날라옴.
진행할수없음



7. 비지역성 점프

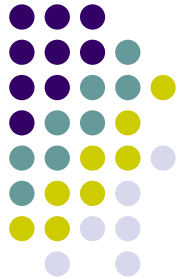
비지역성 점프: setjmp/longjmp

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

setjmp()

#include <setjmp.h>

int setjmp(jmp_buf *env*);



- *setjmp*() 호출시 *longjmp()*에 사용될 env저장
- 직접 호출 시에, *setjmp*()는 0 리턴
- *longjmp()*에 대한 호출 결과인 경우에는...

  setjmp()는 0이 아닌 값을 리턴 함.

* Signal mask를 저장하지는 않음 → sigsetjmp() 참고

longjmp

- void longjmp(jmp_buf j, int i)
  - 동작: *longjmp*() 는 같은 thread에 속한 setjmp()의 최신 호출에 의해 저장된 환경(jmp_buf)로부터 환경을 복구한다. 그 효과는 해당 위치로 jump하는 것을 포함함.
  - *longjmp*()가 완료되면, 마치 *setjmp()* 가 해당 값(i)를 return한 것처럼 행동한다.
  - 그래서, *longjmp*() 함수 인자 i를 0으로 해서는 안됨. 0으로 해도 setjmp()가 1을 리턴하도록 되어야 함.
  - 한번 호출되고, 리턴하지않음.(setjmp()호출위치로 jump만 함)
- longjmp 의 구현:
  - 점프 버퍼 j 로부터 레지스터 컨텍스트를 복원한다
  - %rax (리턴값) 를 i로 설정한다
  - 점프 버퍼 j에 저장된 PC가 가리키는 위치로 이동한다

setjmp/longjmp 예제1

```
#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)
}
```

setjmp/longjmp 예제 2

- 목표: 깊게 중첩된 함수에서 바로 리턴하기

```
/* Deeply nested function foo */
void foo(void)
{
    if (error1)
            longjmp(buf, 1);
    bar();
}

void bar(void)
{
    if (error2)
        longjmp(buf, 2);
}
```

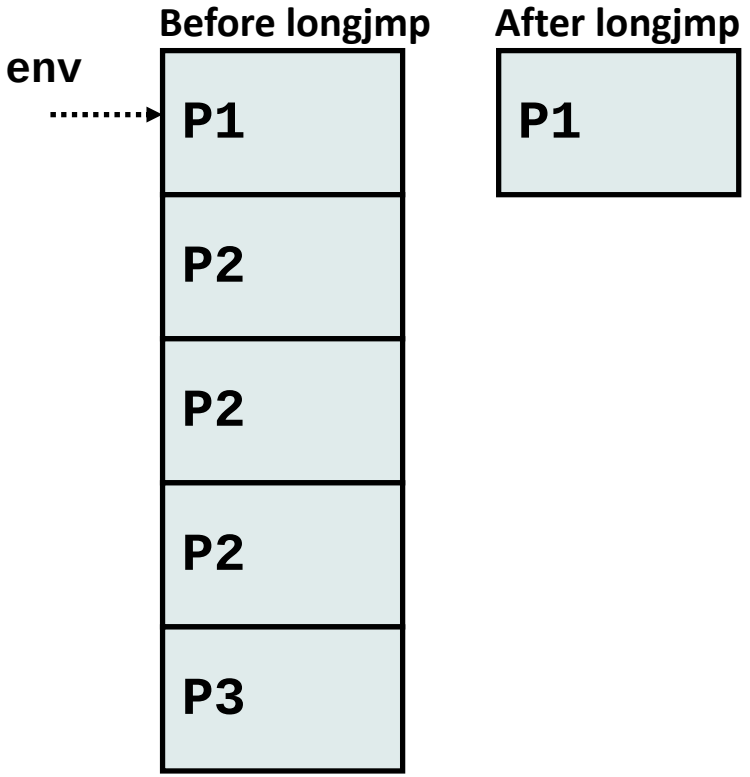
setjmp/longjmp 예제2(계속)

```
jmp_buf buf;

int error1 = 0;
int error2 = 1;

void foo(void), bar(void);

int main()
{
    switch(setjmp(buf)) {
    case 0:
        foo();
        break;
    case 1:
        printf("Detected an error1 condition in foo₩n");
        break;
    case 2:
        printf("Detected an error2 condition in foo₩n");
        break;
    default:
        printf("Unknown error condition in foo₩n");
    }
    exit(0);
}
```

비지역성 점프의 한계

- 스택 내에서 동작
  - 호출된 함수의 환경으로 롱점프함.

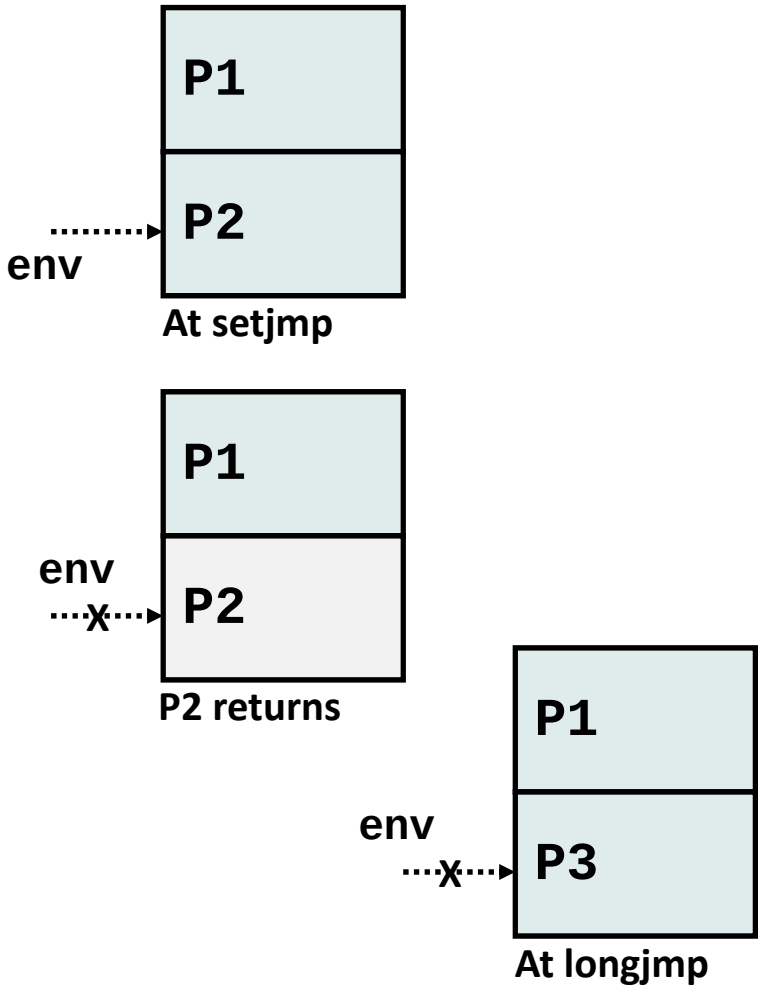
```
jmp_buf env;

P1()
{
  if (setjmp(env)) {
    /* Long Jump to here */
  } else {
    P2();
  }
}

P2()
{  . . . P2(); . . . P3(); }

P3()
{
  longjmp(env, 1);
}
```



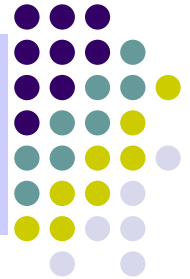
비지역성 점프의 한계

- 스택 내에서 동작
  - 호출된 함수의 환경으로 롱점프함.

```
jmp_buf env;

P1()
{
  P2(); P3();
}

P2()
{
   if (setjmp(env)) {
    /* Long Jump to here */
  }
}

P3()
{
  longjmp(env, 1);
}
```



```
#include <setjmp.h>
int sigsetjmp(sigjmp_buf  env, int  savemask);
```



- *savemask* 인자가 0이 아니면
  - *sigsetjmp*() 는 호출 thread의 signal mask도 같이 저장
- 직접 호출시, *sigsetjmp*()는 0 리턴
- *siglongjmp()*의 호출에 의한 것이면, 0 아닌 값 리턴

```
#include <setjmp.h>
void siglongjmp(sigjmp_buf  env, int  val);
```

- *siglongjmp*() 함수는 longjmp()와 동시하나, 저장된 signal mask도 복구함.
- *siglongjmp*() 호출 완료되면, 프로그램 실행은 마치 해당 *sigsetjmp()* 에서 바로 리턴되고, 그 리턴 값이 val인 것 처럼 동작.
- *siglongjmp*() 에서, 인자 val이 0이어도, *sigsetjmp()* 는 1을 리턴함.

모든 기술을 모아서...
ctrl-c를 누르면 재시작하도록...

```c
#include "csapp.h"

sigjmp_buf buf;

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

int main()
{
    if (!sigsetjmp(buf, 1)) {
        Signal(SIGINT, handler);
            Sio_puts("starting₩n");
    }
    else
        Sio_puts("restarting₩n");

    while(1) {
            Sleep(1);
            Sio_puts("processing...₩n");
    }
    exit(0); /* Control never reaches here */
}
```

restart.c

```
greatwhite> ./restart
starting
processing...
processing...
processing...
restarting
processing...   <-- Ctrl-c
processing...
restarting
processing...   <-- Ctrl-c
processing...
processing...
```

점검문제: 주관식

1. longjmp()가 jump하는 위치는?
2. signal() 함수로, SIGINT를 무시하려면, 호출을 어떻게 해야 하나?
3. 무시할 수 없는 시그널에는 어떤 시그널이 있나?
4. 시그널을 무시하는 것과, block하는 것에는 어떤 효과의 차이가 있나?
5. signal 핸들러와 관련해서 race 조건이 발생하는 이유는 무엇인가?
6. 비동기 시그널에 안전한 함수는 어떤 특징을 갖는가? (두가지)

Summary

- 시그널은 프로세스 수준의 예외처리 방법을 제공한다
  - 사용자 프로그램에서 생성할 수도 있다. 커널이 보낸다.
  - 핸들러로 처리할 수 있다. 무시하거나, block할 수도 있다. 어떤 것들을 무시할 수 없다. ignore와 block은 다르다.
- 단점
  - 오버헤드가 크다.
    - >10,000 클럭 사이클 이상. 예외적인 조건에만 사용하는 게...
  - 큐를 사용하지 않는다. pending 비트는 종류별 한 비트
- 비지역성 점프는 프로세스 내에서 예외적인 제어흐름을 제공한다
  - 스택을 사용하는 한도 내에서

실습과 다음 주 준비

- 실습: Tiny Shell 2

- 다음주 강의 동영상:동적 메모리1
- 예습 질문, 점검문제풀이는 개인과제로 올림.