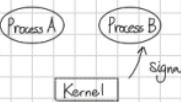




## • Signal Concepts: Sending a Signal

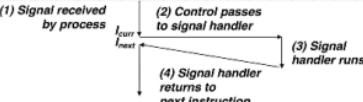
sent (delivered) 된 상태 : Kernel이 process에게 signal을 보내지만, process는 빠져나온 상태  
아직 Action은 하지 않음.

received 된 상태 : process가 signal을 받아 처리(react) 해야 함



## • Signal Concepts: Receiving a Signal

- signal 받았을 때 수행하는 것 X
- 받은 bit vector로 알기.
- 나중에 context switch로 원인 탐지하는데, signal을 채워.
- context switch로 실행될 때, 실행장치에 bit vector를 확인.



react 풀기

Ignore : signal을 무시, 아무것도 하지 X, process action은 강제해제 X

Terminate : process 종료

Catch : signal handler는 user-level function을 설정해, signal을 catch

hardware exception handler는  
내부로, OS가 거둔 거기에서 기록  
exception은 아니.

## • Signal Concepts: Pending & Blocked Signals

\* 여기서는 블록하지 않음. \* 여기서는 블록함.

### 1) Pending

→ sent(발송), received(수신) X

→ message box를 통해 전달되는가?

→ "not queued" → 큐에 되어 있는데요? 생각하지 않아.

증명으로 최대 1회에 Pending 가능 ex) kill 어떤 번호 가능

### 2) Block

→ sent (deliver) 되었지만, 의미로 received 하지 X

↳ unblocked 되기 전까지 receive 하지 않는 것

→ 각 프로세스의 message box에  
pending bit vector와 block bit vector가 있음.  
→ context switch로 2단계로 signal box가 있음.

### ① Pending

signal k7가 deliver되었을 때 kernel의 pending bit 상태의  
k7번 째 bit를 set(설정)

Signal k7가 receive 되었을 때 kernel의 pending bit 상태의  
k7번 째 bit를 clear(제거)

### ② Blocked

sigprocmask function을 사용  
여기 k7번 째 bit를 1로 설정하면 receive X.  
block bit 상태의

pid : process ID  
pgid : process 그룹 ID

getpgid() : 현재 process의  
process 그룹 ID를 return.

setpgid() : process의 그룹을 변경

## Sending Signals with /bin/kill Program

### Sending Signals with /bin/kill Program

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

① forks 16 이후 파일을 통해 → pid 924818  
child process 2개 생성  
Pgrp 924818.

24818, 24819 모두 있음.

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

24821 pid=kill -9 -24817

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

24823 pts/2 00:00:00 ps

```
linux>
```

② /bin/kill -9 -24817 : 24817이라는 process group의 SIGKILL signal을 줌.  
→ 그룹에서, process group의

/bin/kill -9 24818 : 24818에게 process의 SIGKILL signal을 줌.

context switch할 때 kernel pending bit vector를 체크해 kill 가능.

## Sending Signals from the Keyboard

```
bluefish> ./forks 17
Child: pid=28108 pgrp=28107
Parent: pid=28107 pgrp=28107
<types ctrl-z>
Suspended
```

ctrl + z

```
bluefish> ps
PID TTY STAT TIME COMMAND
27699 pts/8 Ss 0:00 -tcsh
28107 pts/8 T 0:01 ./forks 17
28108 pts/8 T 0:01 ./forks 17
28109 pts/8 R+ 0:00 ps w
```

```
bluefish> fg
./forks 17
<types ctrl-c>
bluefish> ps w
PID TTY STAT TIME COMMAND
27699 pts/8 Ss 0:00 -tcsh
28110 pts/8 R+ 0:00 ps w
```

ctrl + C : SIGINT

terminate (종료) ⇒ foreground 3. 화면에 있는 process의 kernel의 SIGINT 신호

ctrl + Z : SIGSTP

stop or suspend (스스) ⇒ foreground 3. 화면에 있는 process의 kernel의 SIGSTP 신호

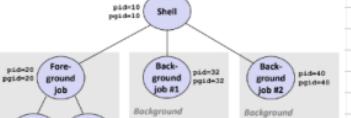
S : sleeping

T : stopped

+ foreground process group

## Sending Signals: Process Groups

당신의 process 그룹에는 여러 개의 process가 존재 가능



getpgrp() : Return process group of current process

setpgid() : Change process group of a process (see text for details)

Foreground process group 20

## Sending Signals with kill function.

```

void fork12()
{
    pid_t pid[N];
    int i;
    int child_status;

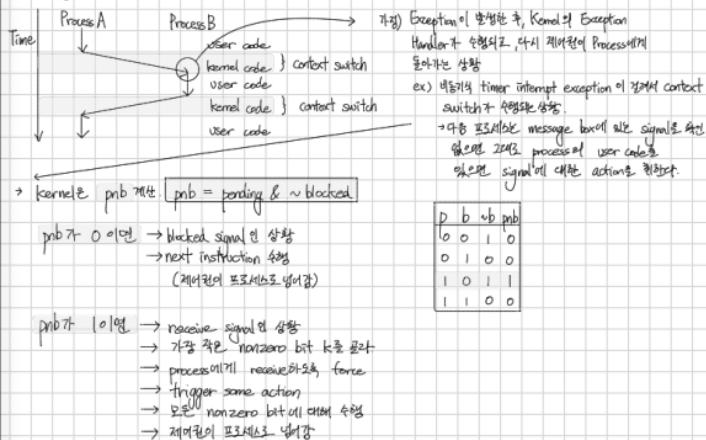
    for (i = 0; i < N; i++) {
        if ((pid[i] = fork()) == 0) {
            /* Child: Infinite Loop */
            while(1)
                child는 여기로 도달 X, 계속 while문 수행중
        }
    }

    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }
    // kill()에 else3
    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);
    }
}

```

*forks.c*

## Receiving Signals



### → Signal의 case: default action

- 1) Process terminate      ex) SIGINT (CTRL+C), SIGKILL, SIGSEGV, SIGALRM, SIGTERM
  - 2) Process stop      ex) SIGSTP (CTRL+Z), SIGSTOP  
↳ SIGCONT signal 때문에 restart하게 된 경우 stop.
  - 3) Ignore      ex) SIGCHLD  
↳ wait() 이후 블아웃하고 원래대로.
- 여기? 종료하거나 멈출수 없잖아 → 무시하는 대신 OS이 Reaping 요청

## Signal Handler Installation

```
handler_t *signal(int signum, handler_t *handler)
```

등록부정 ① SIG\_IGN : signum이 발생하는 signal을 무시 ex) signal(SIGINT, SIG\_IGN)

② SIG\_DFL : signum이 발생하는 signal의 Default Action은 2013.4.28

③ 2.21 : User-Level Signal Handler의 주요 특징:

- signum이 발생하는 시그널을 받았을 때 호출
- 다른 handler를 두는 것을 '설치(installing)'한다고 정한다.
- signal handler의 수행이 끝나면, 제어권이 다시 프로세스에게 돌아감

signal이 의해 interrupt 되기 이전의 프로세스의 Control flow이거나  
다음 명령에 해당하는 명령의 수행

- Signal Handler는 새로운 프로세스인가?  
No, 그냥 subroutine

```
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");
}
```

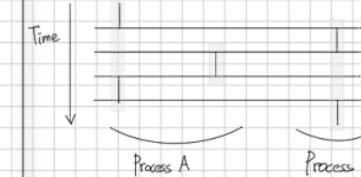
```
/* Wait for the receipt of a signal */
pause();
```

```
return 0;
}
```

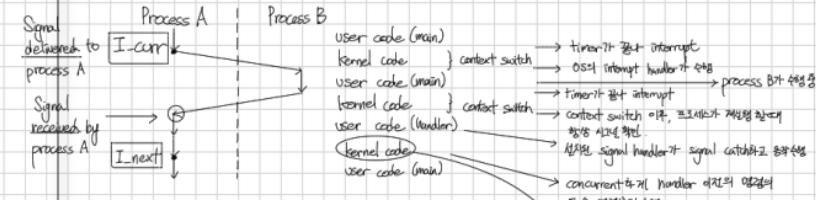
## Signal Handlers as Concurrent Flows

Process A      Process A      Process B  
while()      handler() {  
;      ...  
}

signal handlers separate logical flow (process A ~ X)  
main program A concurrent B ~ Y



## Another View of Signal Handler as Concurrent Flows

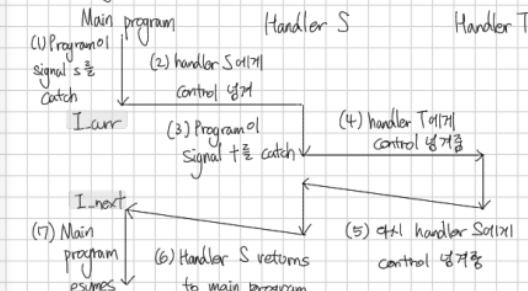


☞ Process A의 내용은 6줄: Signal of signum은 2013.4.28. Signal Handler install 2013.4.28

At sign 2013.4.28 10:00, I start! Timer interrupt off after Context Switch 2. Execute 8:00

?

## Nested Signal Handlers



## Blocking & Unblocking Signals

- Implicit blocking mechanism
  - + Kernel이 처리하지 않는 Action은 signal을 차단
  - 보통 Type의 pending signal은 SFT blocking
  - 어떤 시그널은 해당하는 'Pending & Blocking Bit' Vector에 있나?
  - ? ev process가 차단 SIGINT로 대처할 경우, 수행결과는?  
2 process에 동일한 SIGINT를 deliver하는 경우, block

- Explicit Blocking & Unblocking Mechanism
  - > Kernel이 직접 programmer가 표기해줘야 함
  - > sigprocmask, function을 이용
  - > API: Kernel 5. Implicit Blocking Mechanism // 2.9.10 of Kernel 4.18  
+ 보조함수

int sigemptyset (sigset_t *set)	sigemptyset	: create empty set 인자: set을 empty set으로 만듬.
int sigfillset (sigset_t *set)	sigfillset	: add every signal number to set 인자: set을 모든 signal로 초기화
int sigaddset (sigset_t *set, int signum)	sigaddset	: add signal number to set 인자: set을 모든 signal로 초기화
int sigdelset (sigset_t *set, int signum)	sigdelset	: delete signal number from set 인자: set에서 해당되는 signal 삭제

sigset\_t mask, prev\_mask;

Sigemptyset(&mask); → 일단 Empty Set을 만들어  
Sigaddset(&mask, SIGINT); → set에 SIGINT 추가 (아직 차단되지 않은 거임!!)

/\* Block SIGINT and save previous blocked set \*/  
Sigprocmask(SIG\_BLOCK, &mask, &prev\_mask); → prev\_mask에 기존 디폴트 값을 저장하고  
| 영구적으로 mask를 set으로 설정하여

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

/\* Restore previous blocked set, unblocking SIGINT \*/  
Sigprocmask(SIG\_SETMASK, &prev\_mask, NULL); → 기존의 set을 prev\_mask로 복구한다.

int sigprocmask (int how, const sigset\_t \*set, sigset\_t \*oldset);

how: signal mask를 어떻게 변경할지 지정  
 ↗ SIG\_BLOCK : 현재 마스크에 set의 signal을 추가(차단할 시그널 추가)  
 ↗ SIG\_UNBLOCK : 현재 마스크에서 set의 signal들을 제거(차단 해제)  
 ↗ SIG\_SETMASK : 현재 마스크를 set으로 완전 교체.

set: 적용할 signal set

oldset: 현재 마스크를 저장할 변수 (NULL이면 저장X)

0329

## Safe Signal Handling

→ Signal Handler는 concurrent flow (main program과), 전역 변수 공유

→ guideline

G1: Keep your handlers as simple as possible.

G2: handlers의 entry(44)와 exit(45)에서 errno는 save and restore(예전 저장했던가 반환)?

G3: Protect access to shared data structures by temporarily blocking all signals

G4: 전역 변수를 사용해고자 한다면 volatile \_atomic\_t를 compiler가 register에 저장하는 걸 방지시켜야.

G5: 전역 변수가 flag라는 사용자자료형이라면 volatile sig\_atomic\_t를 사용

→ only read or written (Ex) flag = 1 (o) flag++ (x)  
여기서 중요한 점은?

## Async-Signal-Safety

→ reentrant or non-reentrant

→ \_exit, write, wait, waitpid, sleep, kill

→ printf, sprintf, malloc, exit (x)

## Safely Generating Formatted Output

Set reentrant?

ssize\_t sio\_puts (char s[] )  
ssize\_t sio\_pvt (long v)  
void sio\_error (char s[]) → pt message & exit

```
void sio_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);
}
```

```
ssize_t sio_puts (char s[])
{
    return write (STDOUT_FILENO, s, sio_strlen(s));
}
```

```
void sio_error (char s[])
{
    Sio_puts(s);
    _exit(1);
}
```

03/25

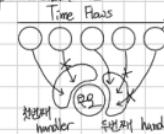
## Incorrect Signal Handling Example fork() child init.

```
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;
    count = N; // handler를 설치할 때
    Signal(SIGCHLD, child_handler); // SIGCHLD는 Parent
    for (i = 0; i < N; i++) {
        if ((pid[i] = Fork()) == 0) { // N개의 프로세스를 만듭니다.
            Sleep(1); //睡眠
            exit(0); /* Child exits */ // SIGCHLD를 발생합니다.
        }
    }
    while (ccount > 0) /* Parent spins */
    ;
}
```

```
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;
}
```

- 5개의 child process 생성
- 각 child의 초기화 대기한 후 종료
- exit()와 kernel의 SIGCHLD는 signal type
- Handler는 각각의 child를 reaping할 때
- reaping에 count를 decrement
- 근데 결과 보면 2번!
- whaleshark> /fork 14**
- Handler reaped child 23240
- Handler reaped child 23241
- Async-Signal-Safe한 함수도 사용하고
- errno는 빠지으면 안됩니다.
- 정역변수로 handler에서만 접근하는지 ...?
- \* Implicit Signal Blocking Mechanism



Action of handling child: Signal에 대해서 pending signal은 큐에 넣습니다.  
SIGCHLD는 catch한 signal handler가 수행합니다.  
하지만 그 상태에 도달한 SIGCHLD는 다시해서 또 처리됩니다.

→ wait()는  
calling process가 fork된 child가 더 이상 없는데  
wait()를 부르면 -1을 return, errno=ECHILD

## Synchronizing Flags to Avoid Races

```
int main(int argc, char **argv)
{
    int pid;
    sigset(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);
}

void handler(int sig)
{
    int olderrno = errno;
    sigset(SIGCHLD, handler);
    pid_t pid;

    if (olderrno != ECHILD)
        Sio_error("waitpid error");
    errno = olderrno;
}
```

Job queue는 queue가 대로 처리된 것으로 queue의 마지막에 signal을 받지 않겠다.

모든 signal을 블록. →进程세스가 생성되면 linked list에 저장?

child가 먼저들면 문제.

```
int main(int argc, char **argv)
{
    int pid;
    sigset(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    Sigfillset(&mask_all); → 전체.
    Sigemptyset(&mask_one); → SIGCHLD가 1, SIGCHLD를 블록하지 않겠다!
    Sigaddset(&mask_one, SIGCHLD);
    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        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);
}
```

04/08

## Explicitly Waiting for Signals

```
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)
{ }
```

```
int main(int argc, char **argv) {
    sigset(SIG_BLOCK, 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);
```

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

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

### Solution: `sigsuspend`

`sigsuspend(const sigset_t *mask)`

SIG\_SETMASK

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

Similar to a shell waiting for a foreground job to terminate.

```
int main(int argc, char **argv) {
    sigset(SIG_BLOCK, 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);
    
```

- 1) OSS setSIG (unblock)
- 2) pause
- 3)

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

04/08 8:57

waitforsignal.c

04/08

sigsuspend.c