Chap. 8 (Part 2)

Exceptional Control Flow: Signals and Nonlocal Jumps

ECF Exists at All Levels of a System

- **Exceptions**
  - Hardware and operating system kernel software
- **Process Context Switch**
  - Hardware timer and kernel software
- **Signals**
  - Kernel software
- **Nonlocal jumps**
  - Application code

Previous Lecture

This Lecture

Today

- **Multitasking, shells**
- Signals
- Nonlocal jumps

The World of Multitasking

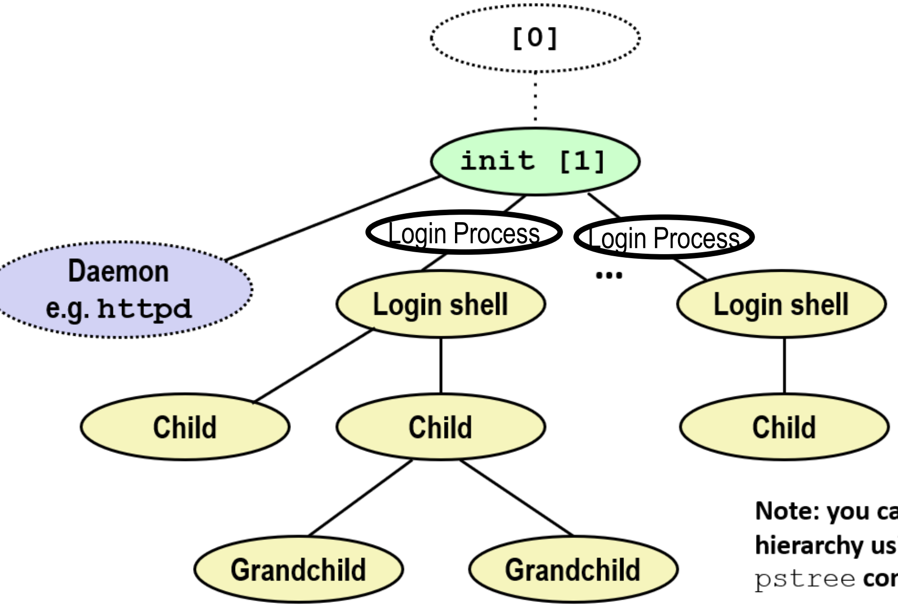
- **System runs many processes concurrently**
- **Process: executing program**
  - State includes memory image + register values + program counter
- **Regularly switches from one process to another**
  - Suspend process when it needs I/O resource or timer event occurs
  - Resume process when I/O available or given scheduling priority
- **Appears to user(s) as if all processes executing simultaneously**
  - Even though most systems can only execute one process at a time
  - Except possibly with lower performance than if running alone

Programmer’s Model of Multitasking

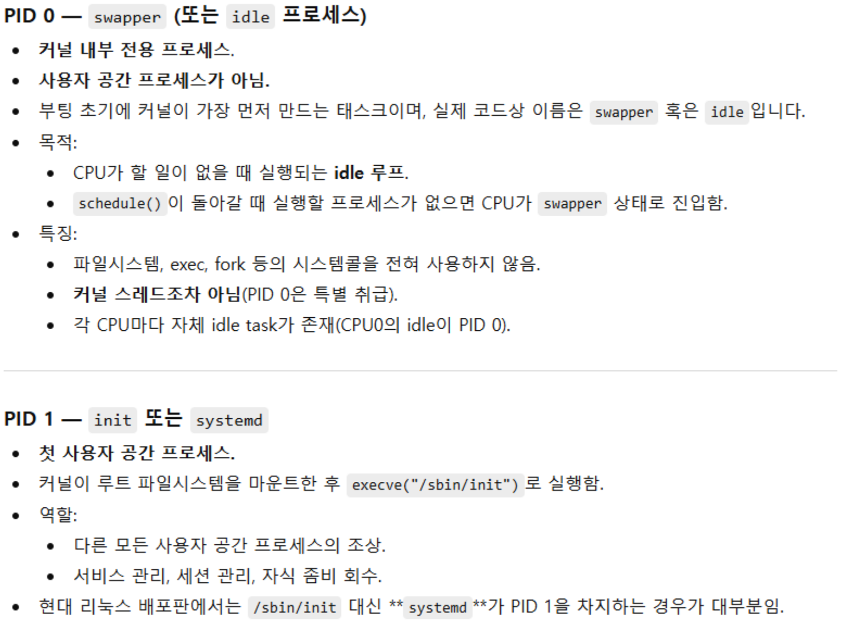
- **Basic functions**
  - `fork` spawns new process
    - Called once, returns twice
  - `exit` terminates own process
    - Called once, never returns
    - Puts it into “zombie” status
  - `wait` and `waitpid` wait for and reap terminated children
  - `execve` runs new program in existing process
    - Called once, (normally) never returns

- **Programming challenge**
  - Understanding the nonstandard semantics of the functions
  - Avoiding improper use of system resources
    - E.g. “Fork bombs” can disable a system

Unix Process Hierarchy



Note: you can view the hierarchy using the Linux `pstree` command



PID 0 — `swapper` (또는 `idle` 프로세스)

- **커널 내부 전용 프로세스.**
- **사용자 공간 프로세스가 아님.**
- 부팅 초기에 커널이 가장 먼저 만드는 태스크이며, 실제 코드상 이름은 `swapper` 혹은 `idle` 입니다.
- 목적:
  - CPU가 할 일이 없을 때 실행되는 **idle 루프**.
  - `schedule()` 이 돌아갈 때 실행할 프로세스가 없으면 CPU가 `swapper` 상태로 진입함.
- 특징:
  - 파일시스템, exec, fork 등의 시스템콜을 전혀 사용하지 않음.
  - **커널 스레드조차 아님**(PID 0은 특별 취급).
  - 각 CPU마다 자체 idle task가 존재(CPU0의 idle이 PID 0).

PID 1 — `init` 또는 `systemd`

- **첫 사용자 공간 프로세스.**
- 커널이 루트 파일시스템을 마운트한 후 `execve("/sbin/init")` 로 실행함.
- 역할:
  - 다른 모든 사용자 공간 프로세스의 조상.
  - 서비스 관리, 세션 관리, 자식 좀비 회수.
- 현대 리눅스 배포판에서는 `/sbin/init` 대신 **`systemd`**가 PID 1을 차지하는 경우가 대부분임.

Shell Programs

- A *shell* is an application program that runs programs on behalf of the user.
  - **`sh`** Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
  - **`csh`** BSD Unix C shell (**`tcsh:`** enhanced `csh` at CMU and elsewhere)
  - **`bash`** "Bourne-Again" Shell

```
int main()
{
   char cmdline[MAXLINE]; /* command line */

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

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

cution is a sequence of d/evaluate steps

Simple Shell `eval` Function

```
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];   /* Holds modified command line */
    int bg;              /* Should the job run in bg or fg? */
    pid_t pid;           /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return;   /* Ignore empty lines */

    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);
            }
        }

        /* Parent waits for foreground job to terminate */
                     if (!bg) {
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else
            printf("%d %s", pid, cmdline);
    }
    return;
}
```

What Is a “Background Job”?

- **Users generally run one command at a time**
  - Type command, read output, type another command

- **Some programs run “for a long time”**
  - Example: “delete this file in two hours”

```
unix> sleep 7200; rm /tmp/junk  # shell stuck for 2 hours
```

- **A “background” job is a process we don't want to wait for**

```
unix> (sleep 7200 ; rm /tmp/junk) &
[1] 907
unix> # ready for next command
```

Problem with Simple Shell Example

- **Our example shell correctly waits for and reaps foreground jobs**

- **But what about background jobs?**
  - Will become zombies when they terminate
  - Will never be reaped because shell (typically) will not terminate
  - Will create a memory leak that could run the kernel out of memory
  - Modern Unix: once you exceed your process quota, your shell can't run any new commands for you: fork() returns -1

```
unix> limit maxproc       # csh syntax
maxproc      202752
unix> ulimit -u           # bash syntax
202752
```

ECF to the Rescue!

- **Problem**
  - The shell doesn't know when a background job will finish
  - By nature, it could happen at any time
  - The shell's regular control flow can't reap exited background processes in a timely fashion
  - Regular control flow is ① "wait until running job completes, then reap it" fg job

    ② polling (Bg job)

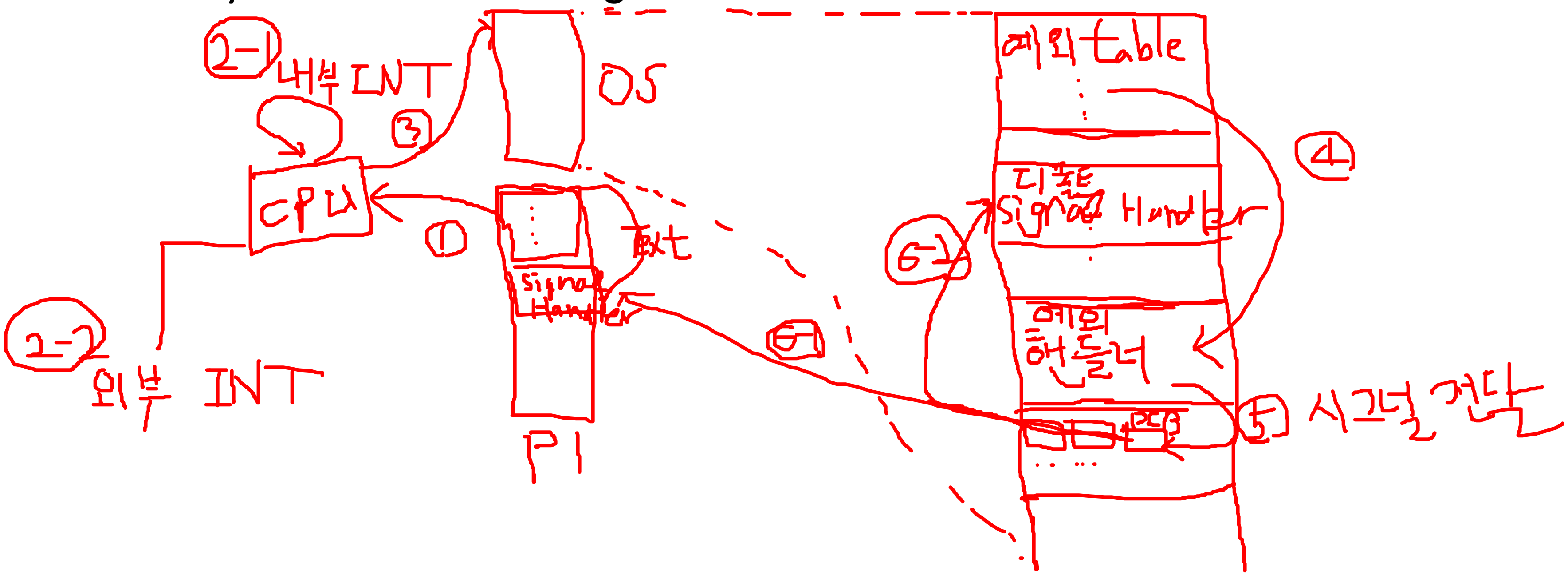
- **Solution: Exceptional control flow**
  - The kernel will interrupt regular processing to alert us when a background process completes
  - In Unix, the alert mechanism is called a ***signal***

Today

- Multitasking, shells
- **Signals**
- Nonlocal jumps

Signals

- **A *signal* is a small message that notifies a process that an event of some type has occurred in the system**
  - akin to exceptions and interrupts
  - sent from the kernel (sometimes at the request of another process) to a process
  - signal type is identified by small integer ID's (1-30)
  - only information in a signal is its ID and the fact that it arrived



Signals

- **A *signal* is a small message that notifies a process that an event of some type has occurred in the system**
  - akin to exceptions and interrupts
  - sent from the kernel (sometimes at the request of another process) to a process
  - signal type is identified by small integer ID's (1-30)
  - only information in a signal is its ID and the fact that it arrived

ID	*Name*	*Default Action*	*Corresponding Event*
2	SIGINT	Terminate	Interrupt (e.g., ctl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

Sending a Signal

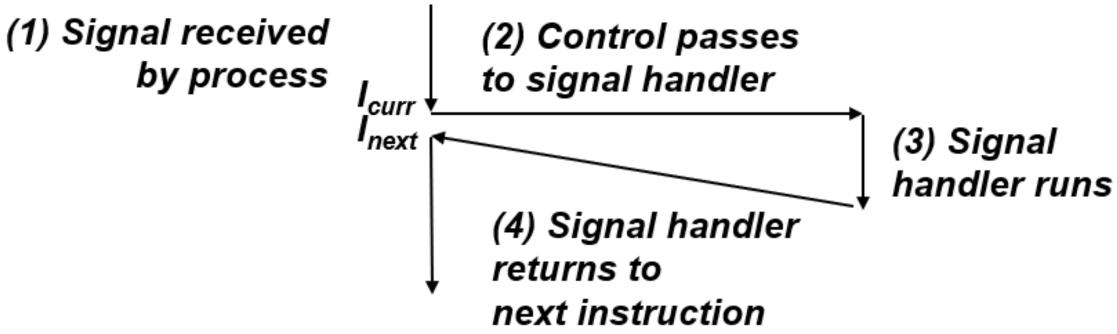
- **Kernel *sends* (delivers) a signal to a *destination process* by updating some state in the context of the destination process**

- **Kernel sends a signal for one of the following reasons:**
  - Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)
  - Another process has invoked the **`kill`** system call to explicitly request the kernel to send a signal to the destination process

Receiving a Signal

- **A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal**

- **Three possible ways to react:**
  - *Ignore* the signal (do nothing)
  - ~~*Terminate* the process (with optional core dump)~~ → 디폴트 동작
  - *Catch* the signal by executing a user-level function called ***signal handler***
    - Akin to a hardware exception handler being called in response to an asynchronous interrupt



(1) Signal received by process

I_{curr}

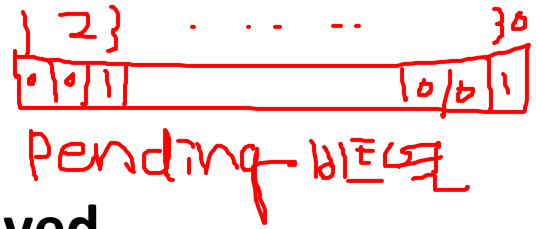
I_{next}

(2) Control passes to signal handler

(3) Signal handler runs

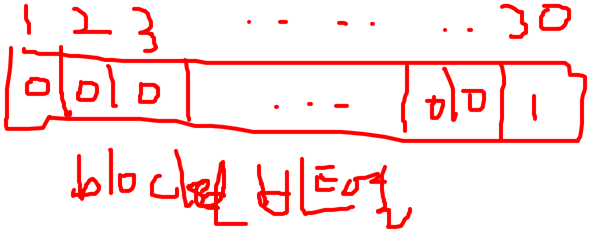
(4) Signal handler returns to next instruction

Pending and Blocked Signals



- **A signal is *pending* if sent but not yet received**
  - There can be at most one pending signal of any particular type
  - Important: Signals are not queued
    - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded

- **A process can *block* the receipt of certain signals**
  - Blocked signals can be delivered, but will not be received until the signal is unblocked (pending bit 가 1이여도 받지 않음)



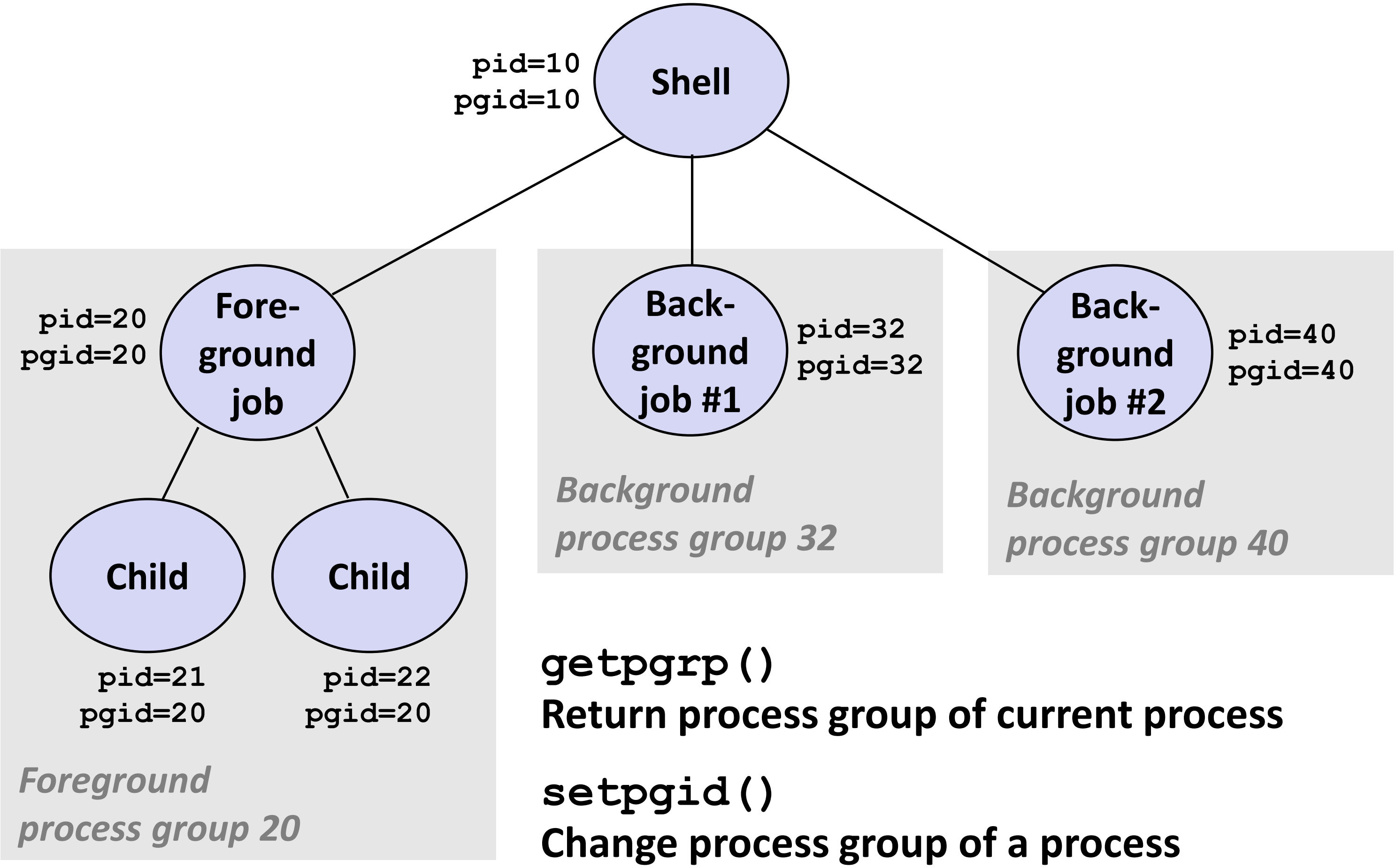
- **A pending signal is received at most once**

Pending/Blocked Bits

- **Kernel maintains `pending` and `blocked` bit vectors in the context of each process**
  - `pending`: represents the set of pending signals
    - Kernel sets bit k in `pending` when a signal of type k is delivered
    - Kernel clears bit k in `pending` when a signal of type k is received
  - `blocked`: represents the set of blocked signals
    - Can be set and cleared by using the `sigprocmask` function

Process Groups

- Every process belongs to exactly one process group



`getpgrp()`
Return process group of current process

`setpgid()`
Change process group of a process

Sending Signals with /bin/`kill` Program

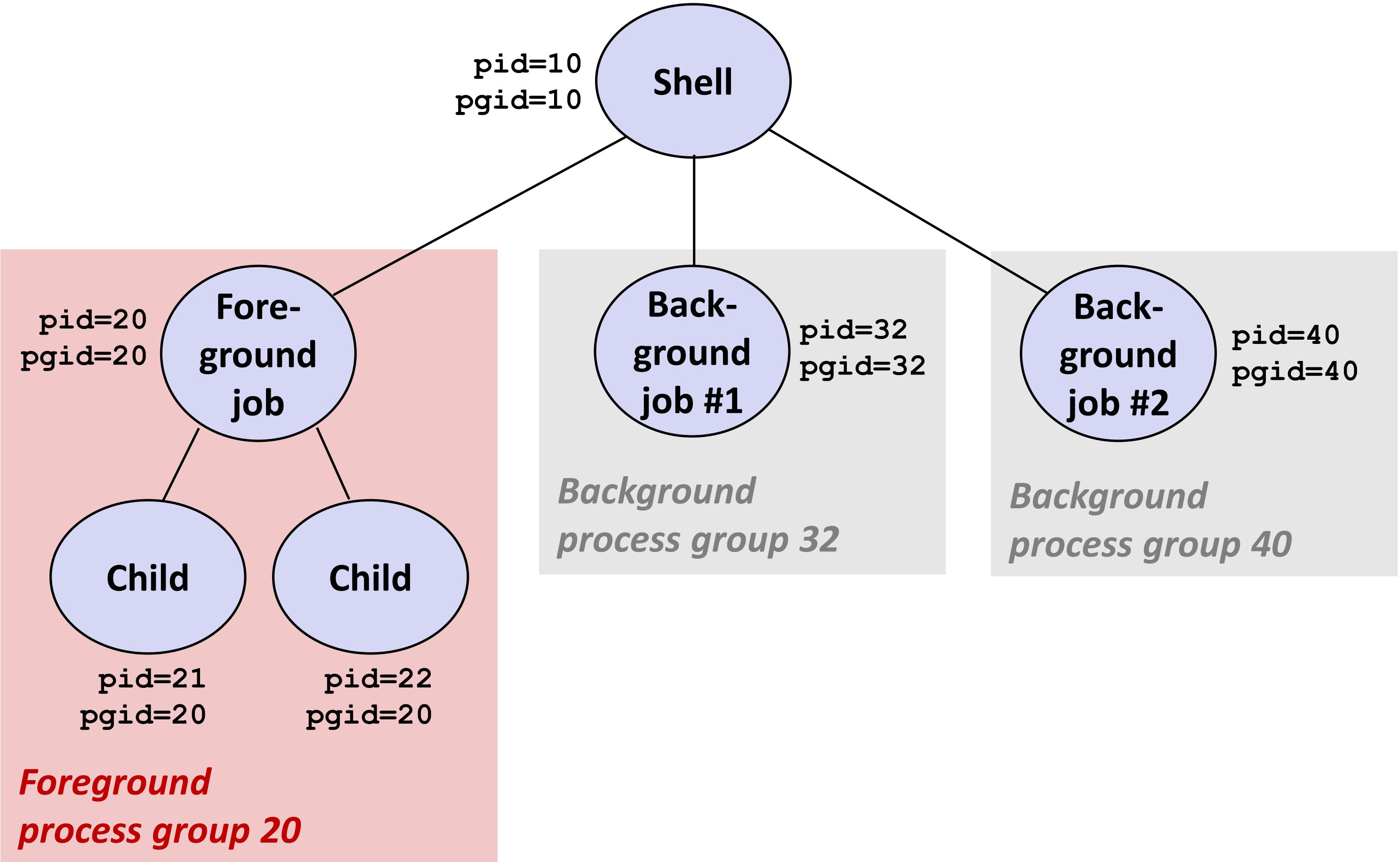
- `/bin/kill` program sends arbitrary signal to a process or process group

- Examples
  - `/bin/kill –9 24818`
    Send SIGKILL to process 24818
  - `/bin/kill –9 –24817`
    Send SIGKILL to every process in process group 24817

```
linux> ./forks 16
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> /bin/kill -9 -24817
linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24823 pts/2    00:00:00 ps
linux>
```

Sending Signals from the Keyboard

- **Typing ctrl-c (ctrl-z) sends a SIGINT (SIGTSTP) to every job in the foreground process group.** ≡ kill -2 -20 (kill -19 -20)
  - SIGINT – default action is to terminate each process
  - SIGTSTP – default action is to stop (suspend) each process



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

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

STAT (process state) Legend:

First letter:

S: sleeping

T: stopped

R: running

Second letter:

s: session leader

+: foreground proc group

See “man ps” for more details

Sending Signals with `kill` Function

```
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);
    }
}
```

✓자식 exit 상태

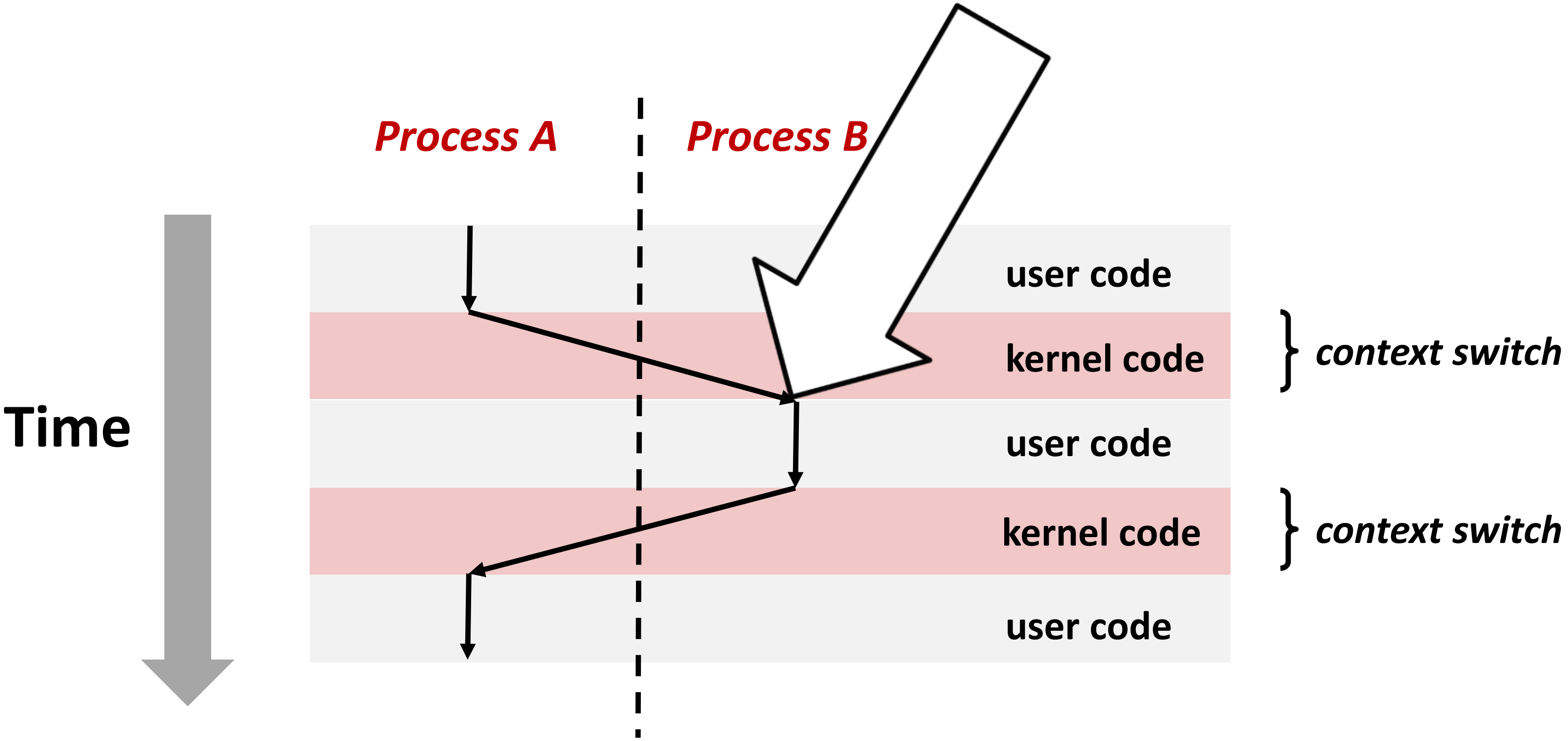
00000000 | 00000010

8 8

✓ zombie 상태

Receiving Signals

- Suppose kernel is returning from an exception handler and is ready to pass control to process *p*



Receiving Signals

- **Suppose kernel is returning from an exception handler and is ready to pass control to process *p***
- **Kernel computes `pnb = pending & ~blocked`**
  - The set of pending nonblocked signals for process *p*
- **If (`pnb` == 0)**
  - Pass control to next instruction in the logical flow for *p*
- **Else**
  - Choose least nonzero bit *k* in `pnb` and force process *p* to ***receive*** signal *k*
  - The receipt of the signal triggers some ***action*** by *p*
  - Repeat for all nonzero *k* in `pnb`
  - Pass control to next instruction in logical flow for *p*

Default Actions

- **Each signal type has a predefined *default action*, which is one of:**
  - The process terminates
  - The process terminates and dumps core
  - The process stops until restarted by a SIGCONT signal
  - The process ignores the signal

Installing Signal Handlers

- **The `signal` function modifies the default action associated with the receipt of signal `signum`:**
  - `handler_t *signal(int signum, handler_t *handler)`

- **Different values for `handler`:**
  - SIG_IGN: ignore signals of type **`signum`**
  - SIG_DFL: revert to the default action on receipt of signals of type **`signum`**
  - Otherwise, **`handler`** is the address of a ***signal handler***
    - Called when process receives signal of type **`signum`**
    - Referred to as ***"installing"*** the handler
    - Executing handler is called ***"catching"*** or ***"handling"*** the signal
    - When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal

Signal Handling Example

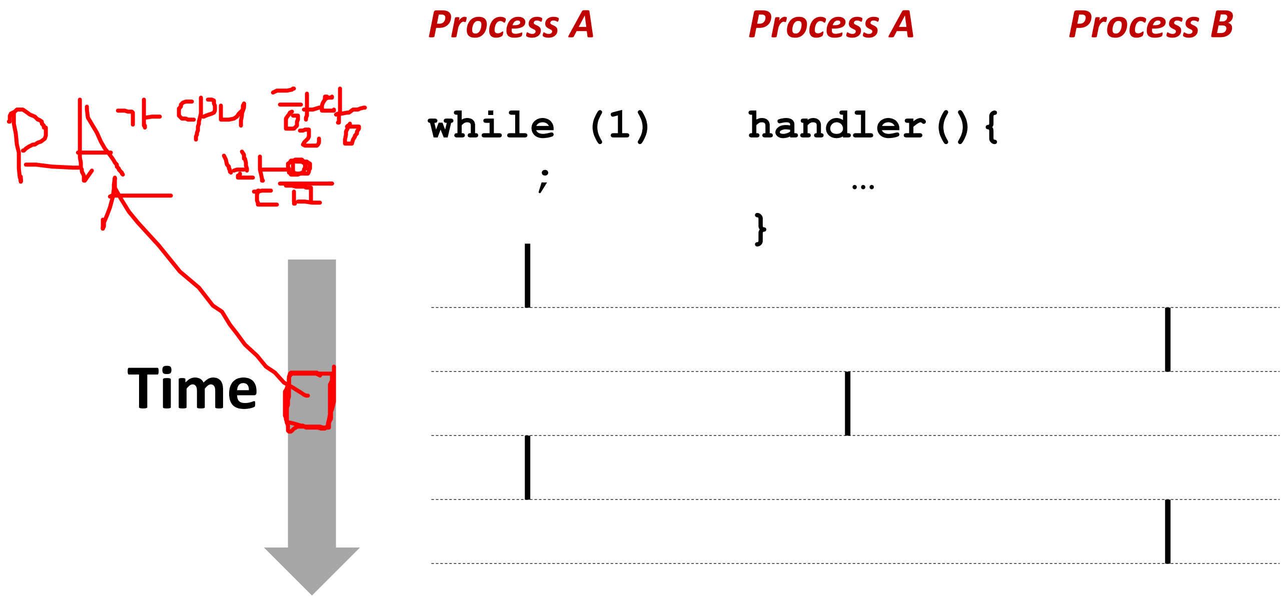
```
void int_handler(int sig) {
    safe_printf("Process %d received signal %d\n", getpid(), sig);
    exit(0);
}

void fork13() {
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0
            while(1); /* child inf
        }
    for (i = 0; i < N; i++) {
        printf("Killing process %d
        kill(pid[i], SIGINT);
    }
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_s
(O)     if (WIFEXITED(child_status
            printf("Child %d termi
                   wpid, WEXITSTAT
(X)     else
            printf("Child %d termi
    }
}
```

✓ 자식 exit 상태

```
linux> ./forks 13
Killing process 25417
Killing process 25418
Killing process 25419
Killing process 25420
Killing process 25421
Process 25417 received signal 2
Process 25418 received signal 2
Process 25420 received signal 2
Process 25421 received signal 2
Process 25419 received signal 2
Child 25417 terminated with exit status 0
Child 25418 terminated with exit status 0
Child 25420 terminated with exit status 0
Child 25419 terminated with exit status 0
Child 25421 terminated with exit status 0
linux>
```

Signals Handlers as Concurrent Flows

- **A signal handler is a separate logical flow (not process) that runs concurrently with the main program**
  - "concurrently" in the "not sequential" sense



Process A

```
while (1)
    ;
```

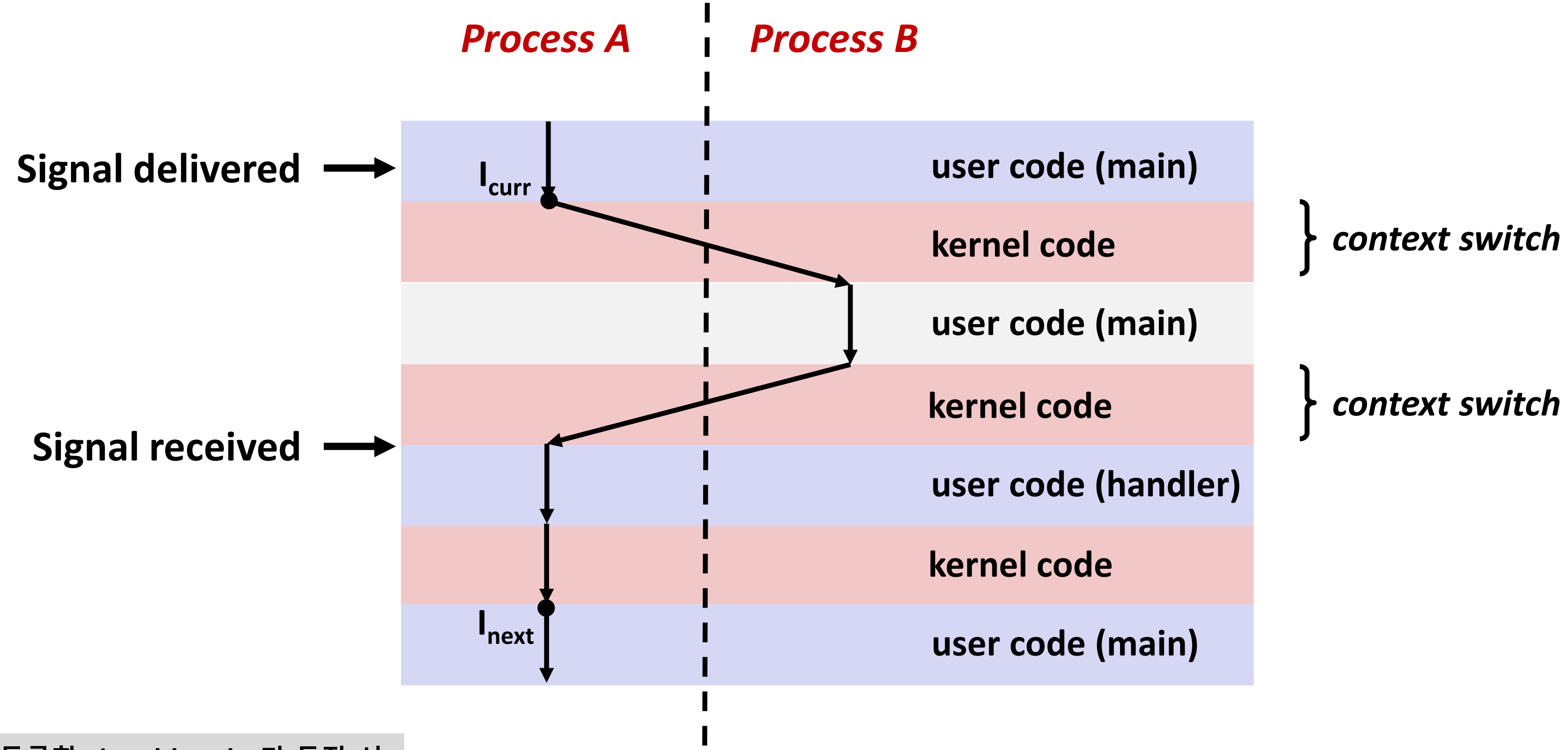
Process A

```
handler(){
    …
}
```

Process B

Time

Another View of Signal Handlers as Concurrent Flows

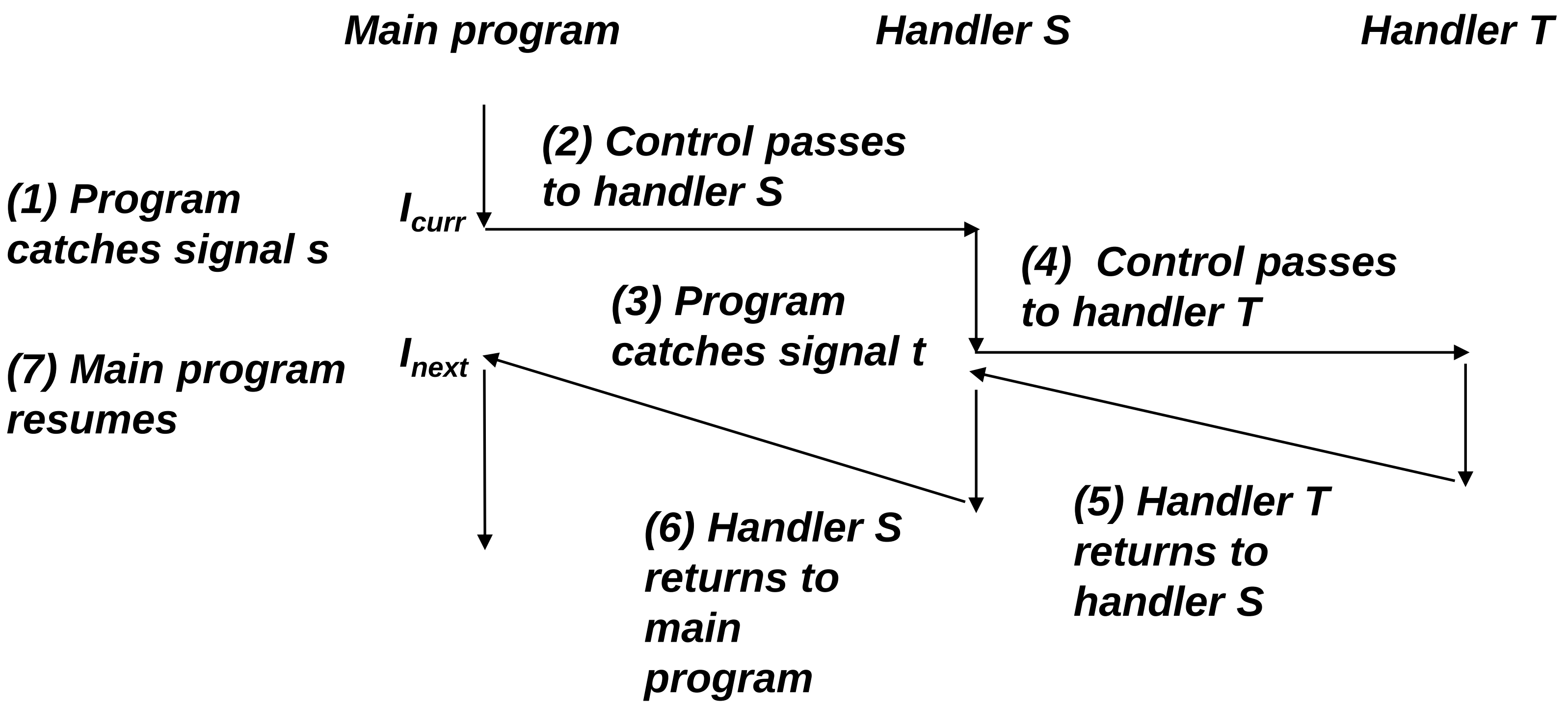


User가 등록한 signal hander가 동작 시

시그널 핸들러가 끝날 때 단순히 함수가 끝나는 것이 아니라,
커널이 유저 스택에 심어둔 `sigreturn` **트램펄린을 통해 시스템콜이 발생**하고,
커널이 원래 문맥을 복원한 후 유저 코드로 되돌아갑니다.
즉, **핸들러 종료 →** `sigreturn` **→ 커널 진입 → 문맥 복원 → 유저 모드 복귀** 순서입니다.

Nested Signal Handlers

- Handlers can be interrupted by other handlers



Main program

Handler S

Handler T

(1) Program catches signal s

I_{curr}

(2) Control passes to handler S

(3) Program catches signal t

(4) Control passes to handler T

(5) Handler T returns to handler S

(6) Handler S returns to main program

(7) Main program resumes

I_{next}

Blocking and Unblocking Signals

- **Implicit blocking mechanism**
  - Kernel blocks any pending signals of type currently being handled.
  - E.g., A SIGINT handler can't be interrupted by another SIGINT

- **Explicit blocking and unblocking mechanism**
  - `sigprocmask` function

- **Supporting functions**
  - `sigemptyset` – Create empty set
  - `sigfillset` – Add every signal number to set
  - `sigaddset` – Add signal number to set
  - `sigdelset` – Delete signal number from set

Temporarily Blocking Signals

```
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);
```

Safe Signal Handling

- **Handlers are tricky because they are concurrent with main program and share the same global data structures.**
  - Shared data structures can become corrupted.

- **We'll explore concurrency issues later in the term.**

- **For now here are some guidelines to help you avoid trouble.**

Guidelines for Writing Safe Handlers

- **G0: Keep your handlers as simple as possible**
  - e.g., Set a global flag and return
- **G1: Call only async-signal-safe functions in your handlers**
  - `printf`, `sprintf`, `malloc`, and `exit` are not safe!
- **G2: Save and restore `errno` on entry and exit**
  - So that other handlers don't overwrite your value of `errno`
- **G3: Protect accesses to shared data structures by temporarily blocking all signals.**
  - To prevent possible corruption
- **G4: Declare global variables as `volatile`**
  - To prevent compiler from storing them in a register
- **G5: Declare global flags as `volatile sig_atomic_t`**
  - *flag*: variable that is only read or written (e.g. flag = 1, not flag++)
  - Flag declared this way does not need to be protected like other globals

Async-Signal-Safety

- **Function is *async-signal-safe* if either reentrant (e.g., all variables stored on stack frame, CS:APP3e 12.7.2) or non-interruptible by signals.**
- **Posix guarantees 117 functions to be async-signal-safe**
  - Source: "`man 7 signal`"
  - Popular functions on the list:
    - `_exit, write, wait, waitpid, sleep, kill`
  - Popular functions that are **not** on the list:
    - `printf, sprintf, malloc, exit`
    - Unfortunate fact: `write` is the only async-signal-safe output function

Safely Generating Formatted Output

- **Use the reentrant SIO (Safe I/O library) from `csapp.c` in your handlers.**
  - `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

Correct Signal Handling

- **Pending signals are not queued**
  - For each signal type, one bit indicates whether or not signal is pending…
  - …thus at most one pending signal of any particular type.
- **You can’t use signals to count events, such as children terminating.**

```c
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

```
whaleshark> ./forks 14   N=5
Handler reaped child 23240
Handler reaped child 23241
```

Correct Signal Handling

- **Must wait for all terminated child processes**
  - Put `wait` in a loop to reap all terminated children

```
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>
```

Portable Signal Handling

- **Ugh! Different versions of Unix can have different signal handling semantics**
  - Some older systems restore action to default after catching signal
  - Some interrupted system calls can return with errno == EINTR
  - Some systems don't block signals of the type being handled
- **Solution: `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

Synchronizing Flows to Avoid Races

- Simple shell with a subtle synchronization error because it assumes parent runs before child.

```c
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

Synchronizing Flows to Avoid Races

- **SIGCHLD handler for a simple shell**

```
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

Corrected Shell Program without Race

```
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) {
        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

Explicitly Waiting for Signals

- Handlers for program explicitly waiting for SIGCHLD to arrive.

```
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

Explicitly Waiting for Signals

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

```
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

Explicitly Waiting for Signals

- Program is correct, but very wasteful
- Other options:

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

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

- Solution: `sigsuspend`

Waiting for Signals with `sigsuspend`

- `int sigsuspend(const sigset_t *mask)`

- **Equivalent to atomic (uninterruptable) version of:**

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

Waiting for Signals with sigsuspend

```
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

Today

- Shells
- Signals
- **Nonlocal jumps**
  - Consult your textbook

Summary

- **Signals provide process-level exception handling**
  - Can generate from user programs
  - Can define effect by declaring signal handler
  - Be very careful when writing signal handlers

- **Nonlocal jumps provide exceptional control flow within process**