

# Exceptional Control Flow: Signals and Nonlocal Jumps

System Programming

Woong Sul

# ECF Exists at All Levels of a System

---

- Exceptions
  - Hardware and operating system kernel software
- Process Context Switch
  - Hardware timer and kernel software

Previous Lecture

- Signals
  - Kernel software and application software

This Lecture

- Nonlocal jumps
  - Application code

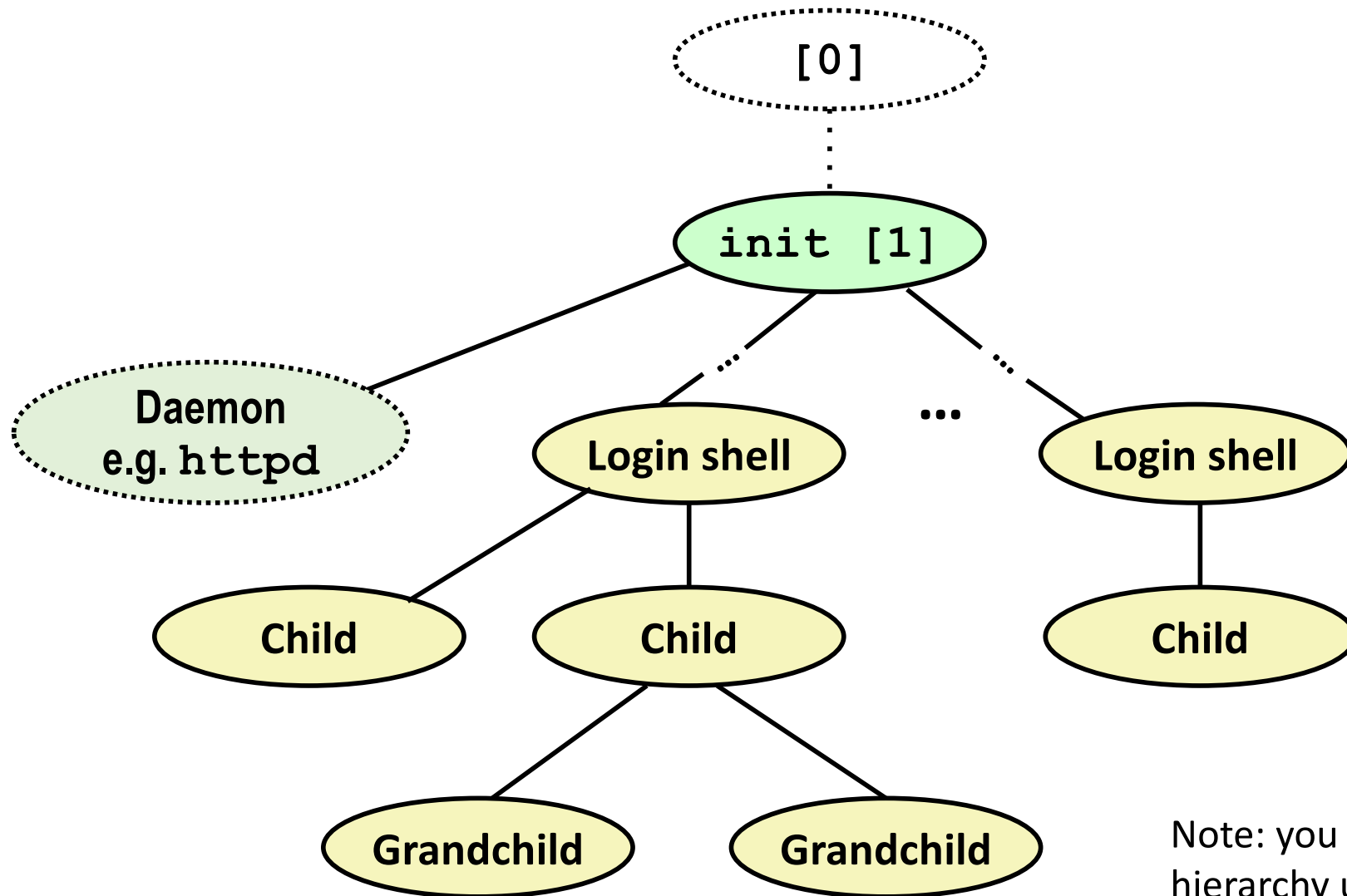
Textbook

# Today

---

- Shells
- Signals

# Linux Process Hierarchy



Note: you can view the hierarchy using the Linux `ps tree` command

# 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)
  - **cshtcsh** BSD Unix C shell
  - **bash** “Bourne-Again” Shell (default Linux shell)

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

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

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

*shellex.c*

*Execution is a  
sequence of  
read/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;
}
```

shellex.c

# Problem with Simple Shell Example

---

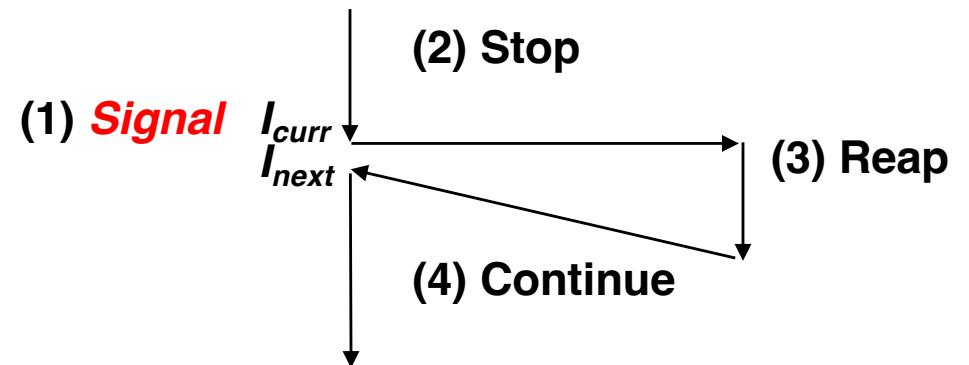
- Our example shell correctly waits for and reaps foreground jobs
- But what about background jobs?
  - Will become zombies when they terminate
  - Will never be reaped because shell (typically) will not terminate
  - Will create a memory leak that could run the kernel out of memory

# ECF to the Rescue!

- 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**

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

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





# Today

---

- Shells
- Signals

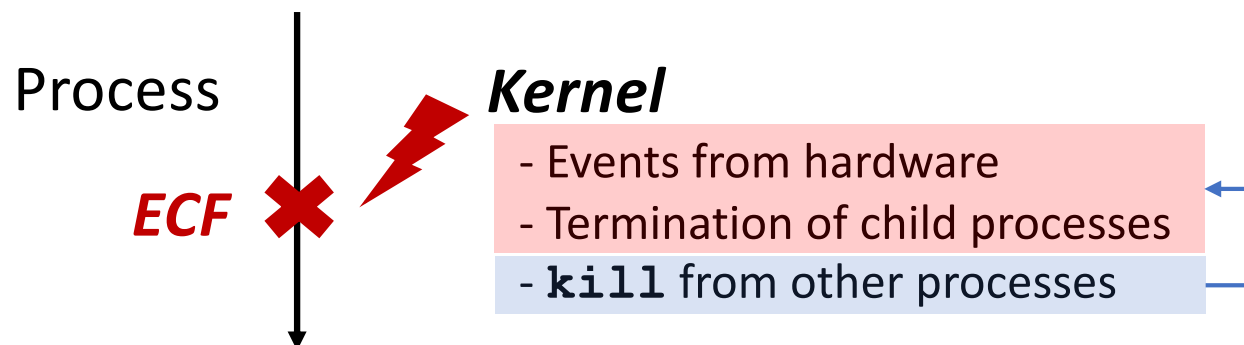
# Signals

- 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

<i>ID</i>	<i>Name</i>	<i>Default Action</i>	<i>Corresponding Event</i>
2	SIGINT	Terminate	User typed ctrl-c
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

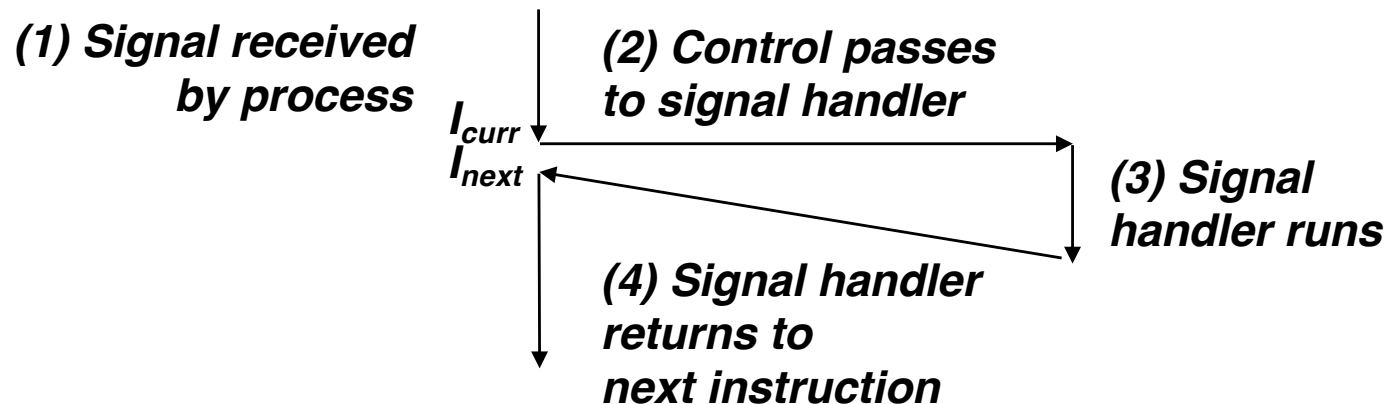
# Signal Concepts: Sending a Signal

- 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)
  - **Other processes** has invoked the kill system call to explicitly request the kernel to send a signal to the destination process



# Signal Concepts: 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
- Some possible ways to react:
  - *Ignore* the signal (do nothing)
  - *Terminate* the process (with optional core dump)
  - *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:



# Signal Concepts: Pending & Blocking

---

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

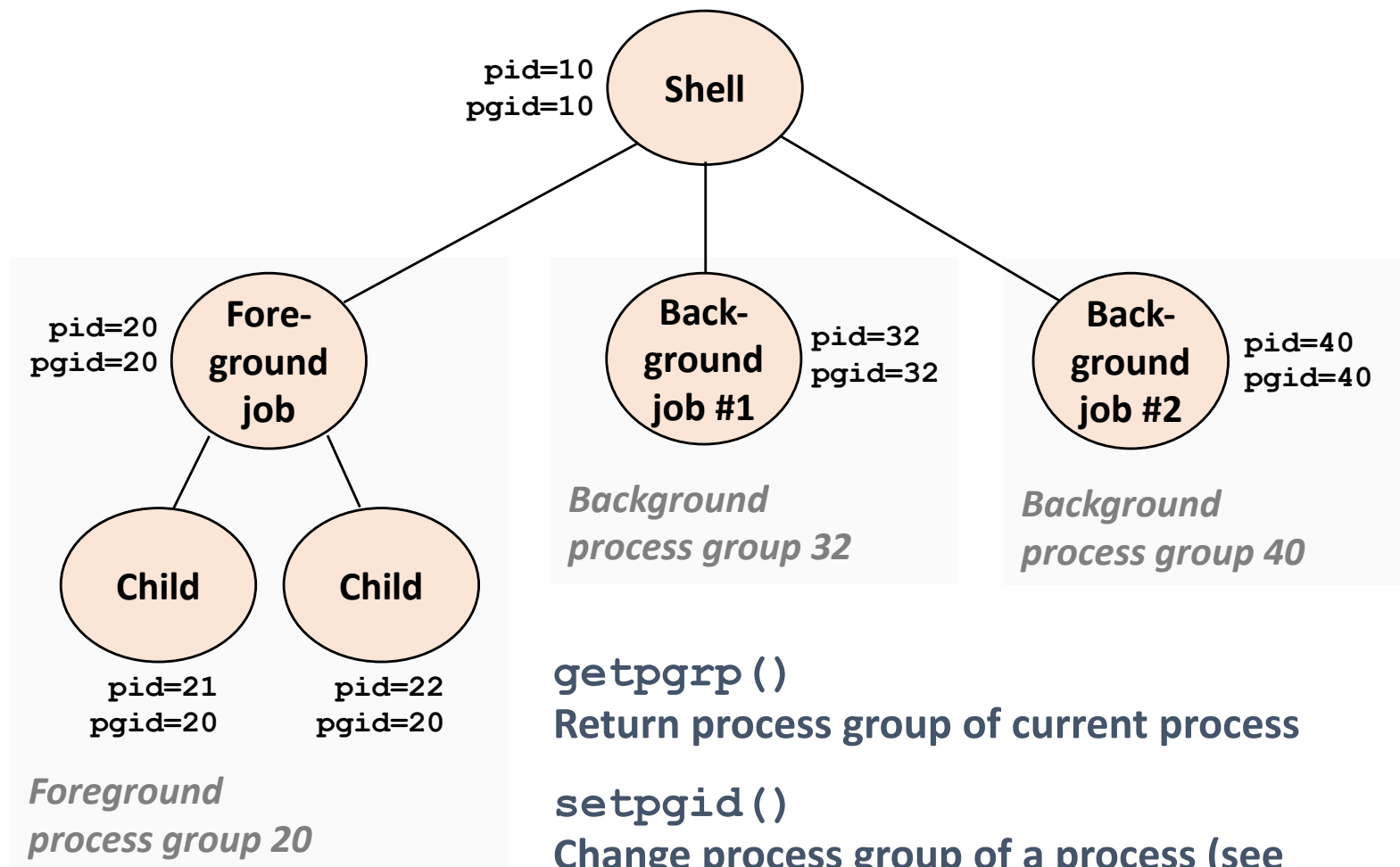
# At Most One Pending Signal

---

- 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 **`sigprocmask()`**
    - Also referred to as the signal mask

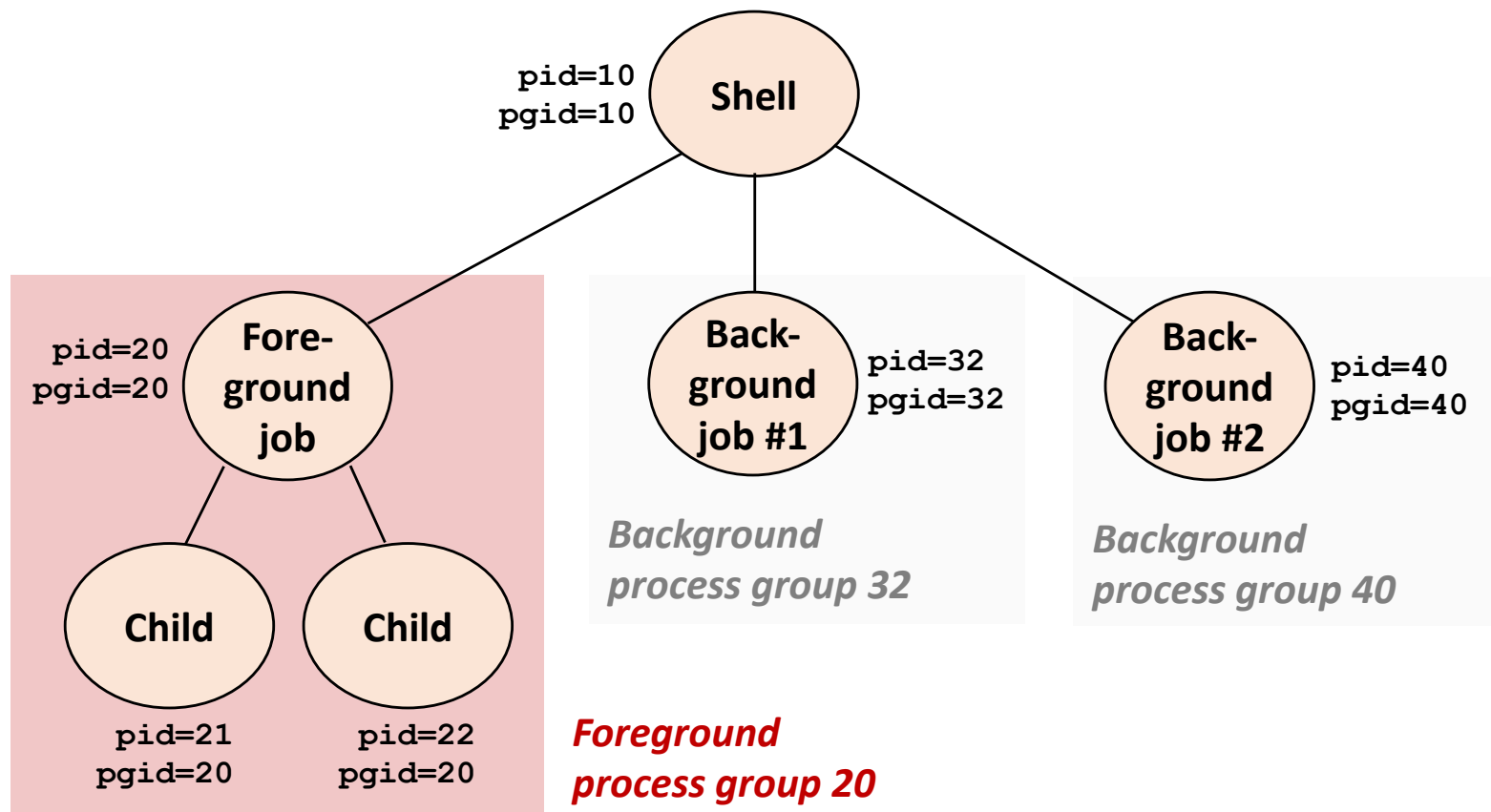
# Sending Signals: Process Groups

- Every process belongs to exactly one process group



# #1 - Sending Signals from the Keyboard

- Ctrl-c/z causes the kernel to send a **SIGINT/SIGTSTP** to every job in the foreground process group
  - **SIGINT** – default action is to terminate each process
  - **SIGTSTP** – default action is to stop (or 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

## #2 - Sending Signals with `/bin/kill`

- `/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>
```

# #3 - 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)
                ;
        }

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

    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);
    }
}

```

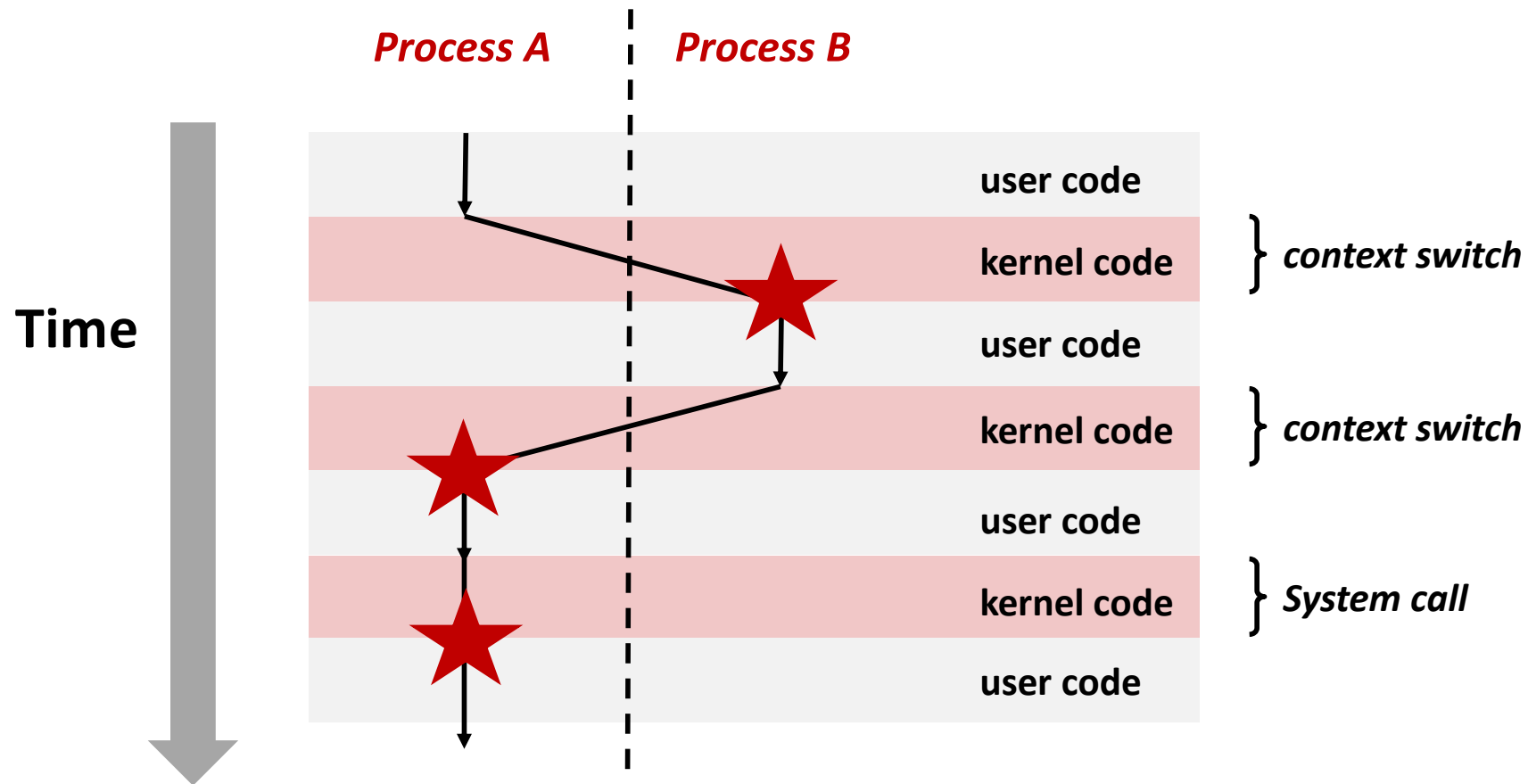
How can process terminate?

- return from `main()`
- calling `exit()`
- receiving **signals**

*forks.c*

# Receiving Signals

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



# Receiving Signals (Cnt'd)

- Suppose kernel is returning from an exception handler and is ready to pass control to process  $p$
- Kernel computes  $pnb = pending \ \& \ \sim blocked$ 
  - The set of pending non-blocked signals for process  $p$
- If no pending signal to handle  $\rightarrow pnb == 0$ 
  - Pass control to next instruction in the logical flow for  $p$
- Otherwise
  - 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 **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 user-level *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 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...\n");
    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");

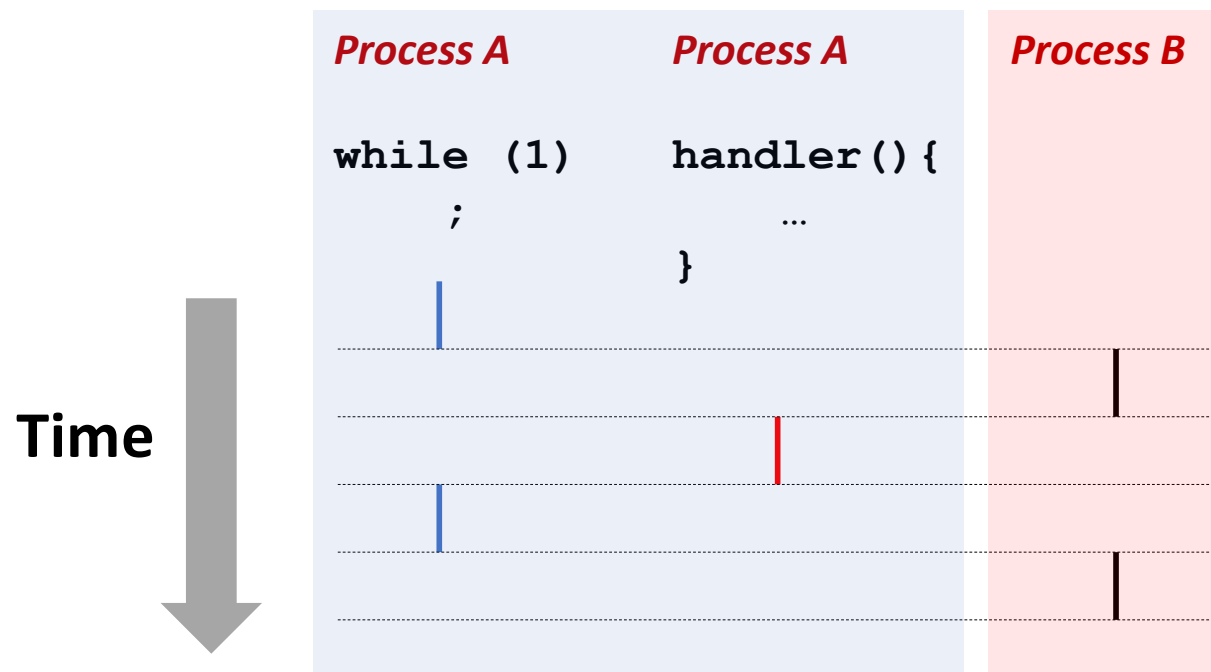
    /* Wait for the receipt of a signal */
    while (1);

    return 0;
}
```

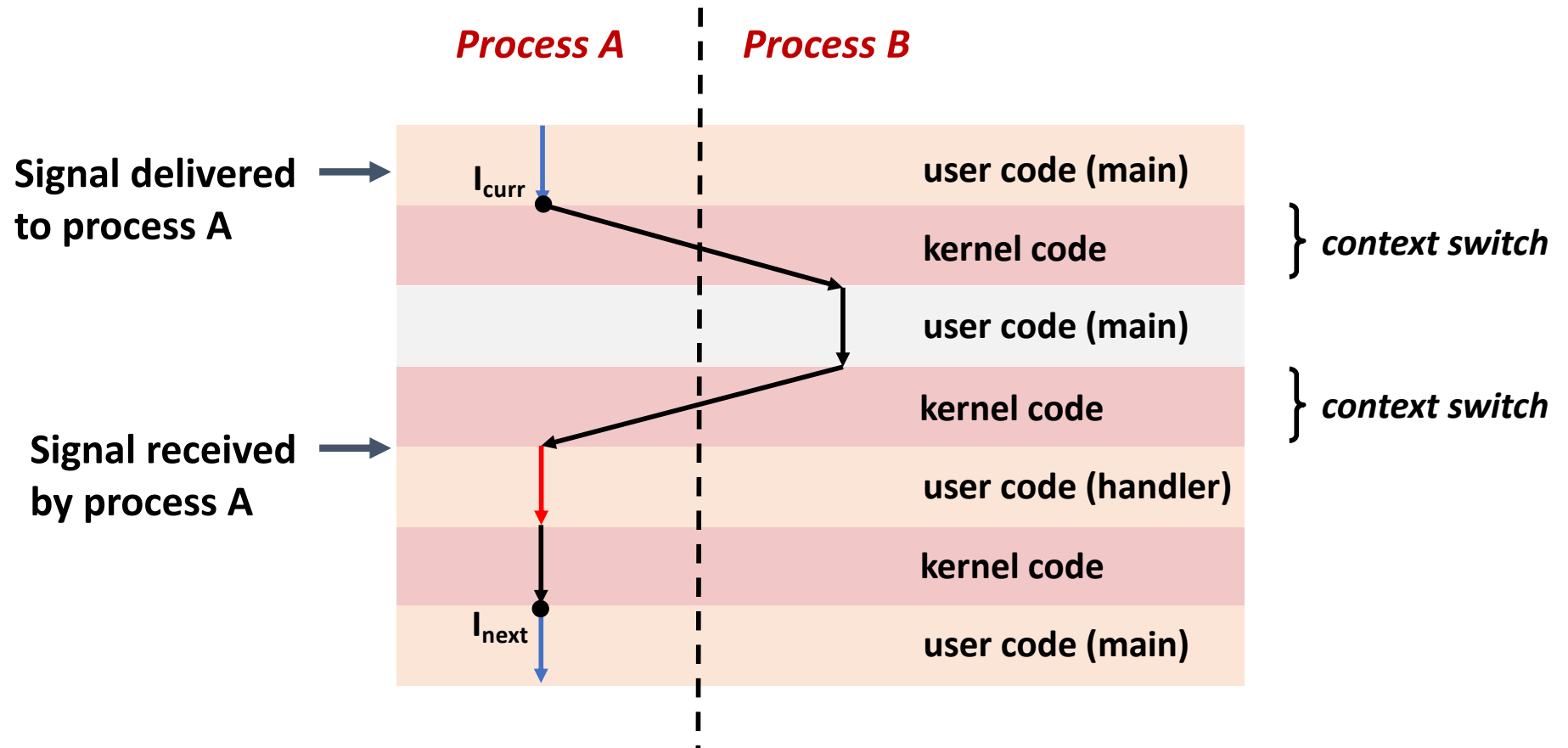


# Signal Handlers as Concurrent Flows

- A signal handler is a separate logical flow (not process) that runs concurrently with the main program
  - A logical flow from `main()`
  - Another logical flow from `handler()`
  - Two logical flows belong to the same process

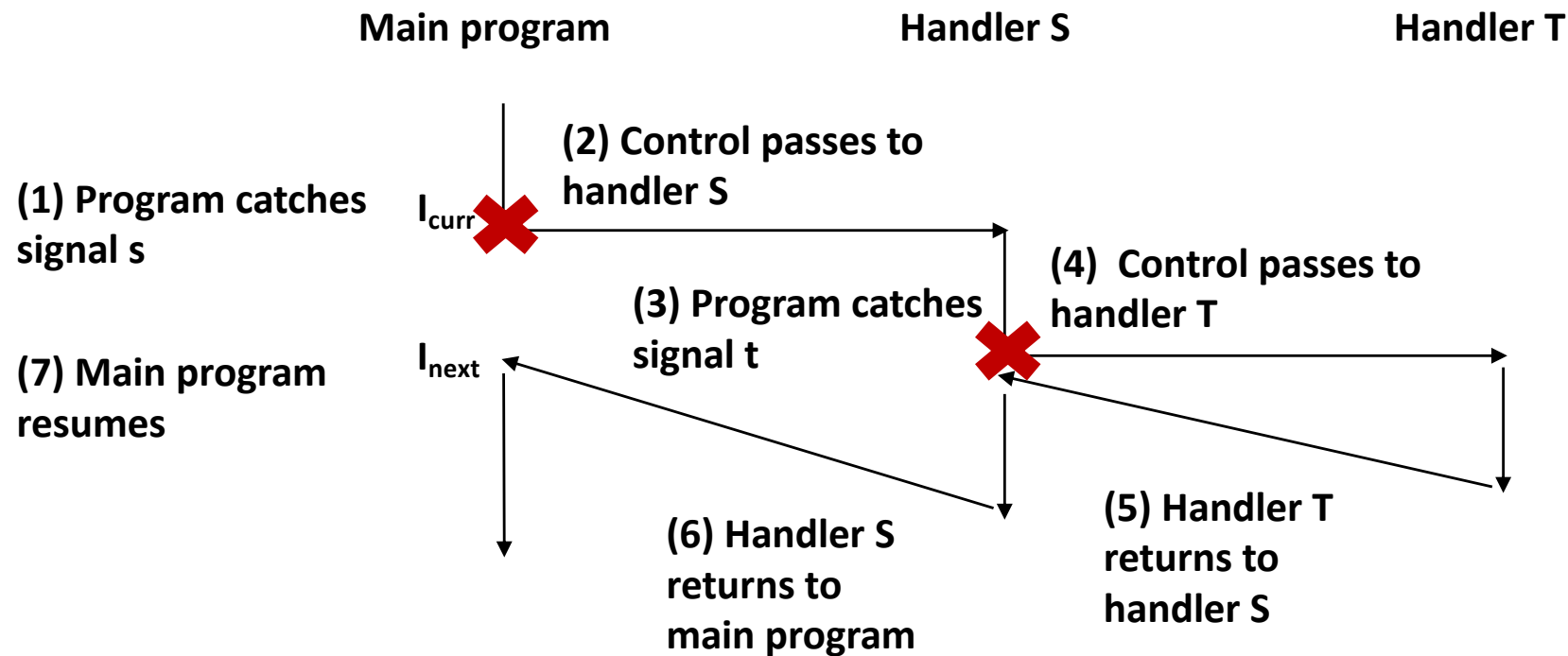


# Signal Handlers as Concurrent Flows (Cnt'd)



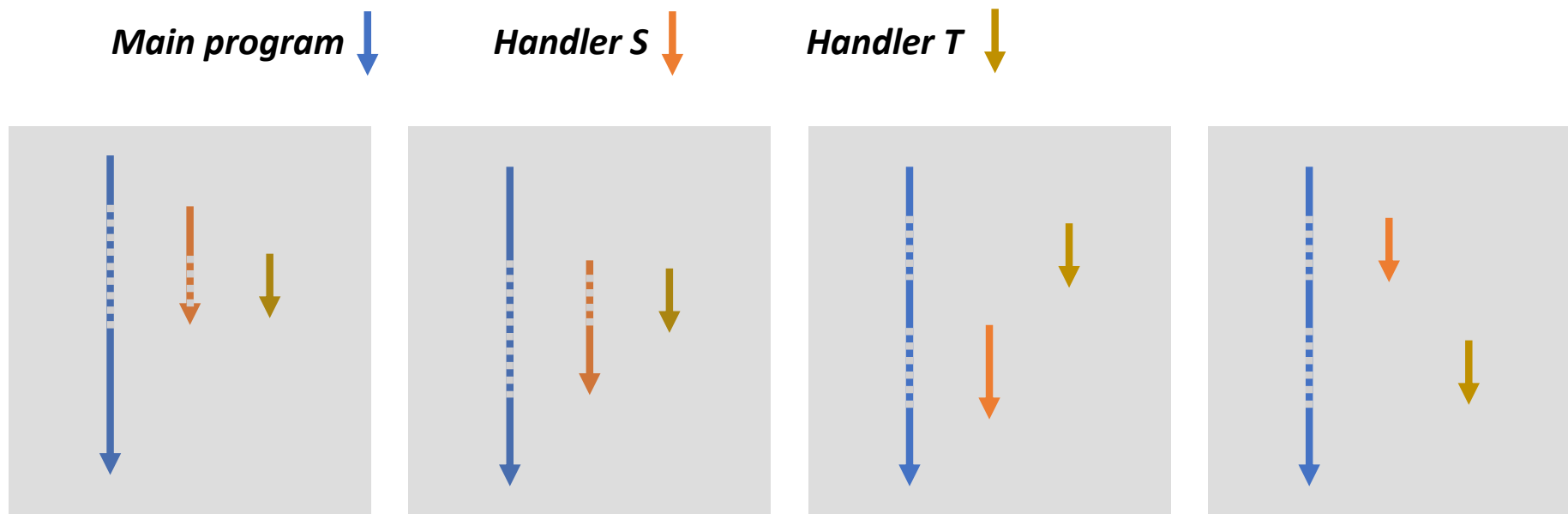
# Nested Signal Handlers

- Handlers can be interrupted by other handlers
  - N concurrent flows in the same process
  - Concurrent flows can produce unexpected results



# Nested Signal Handlers (Cnt'd)

- N-concurrent flows in the same process can lead to the unexpected results
  - All the following flows are possible, and the results may vary



# 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()`
- 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);

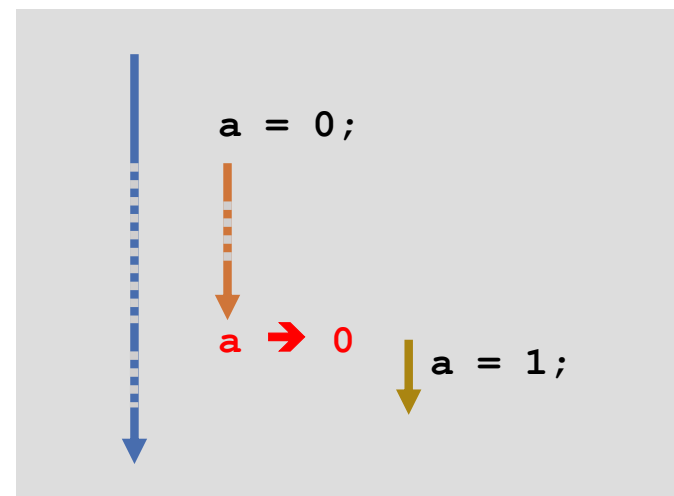
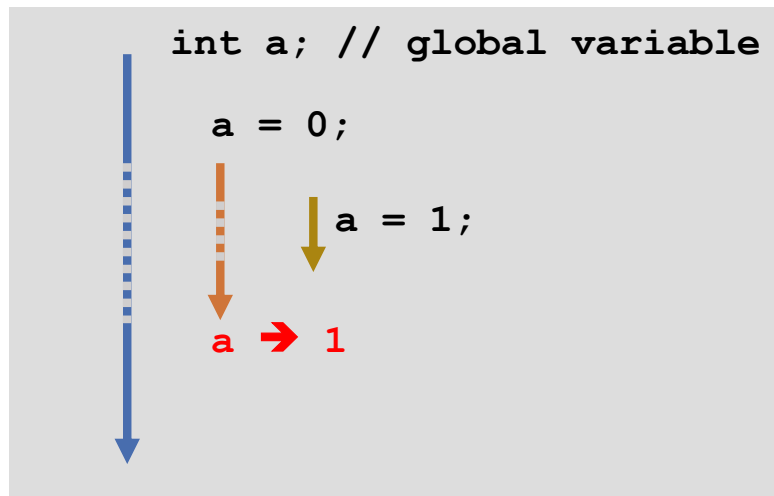
/* Block SIGINT and save previous blocked set */
Sigprocmask(SIG_BLOCK, &mask, &prev_mask);

⋮ /* 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



- For now, here are some conservative 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
  - reentrant → relying on local variables (in a different stack)
  - non-interruptible → blocking 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?

```
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); /* Pretend to do something */
    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 */
}
```

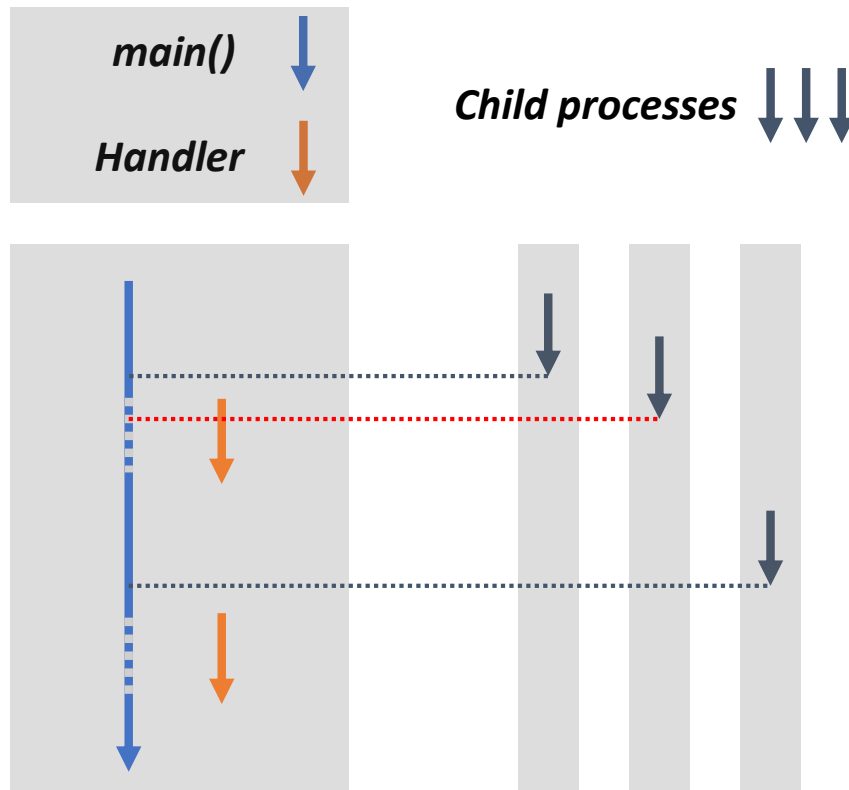
forks.c

- 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

```
$ ./forks 14
Handler reaped child 23240
Handler reaped child 23241
```

# Why Only Two Processes Reaped?

- **Implicit** blocking mechanism



# 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;
}
```

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

# 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

# Concurrent Flows to Lead Races

- SIGCHLD handler for a simple shell

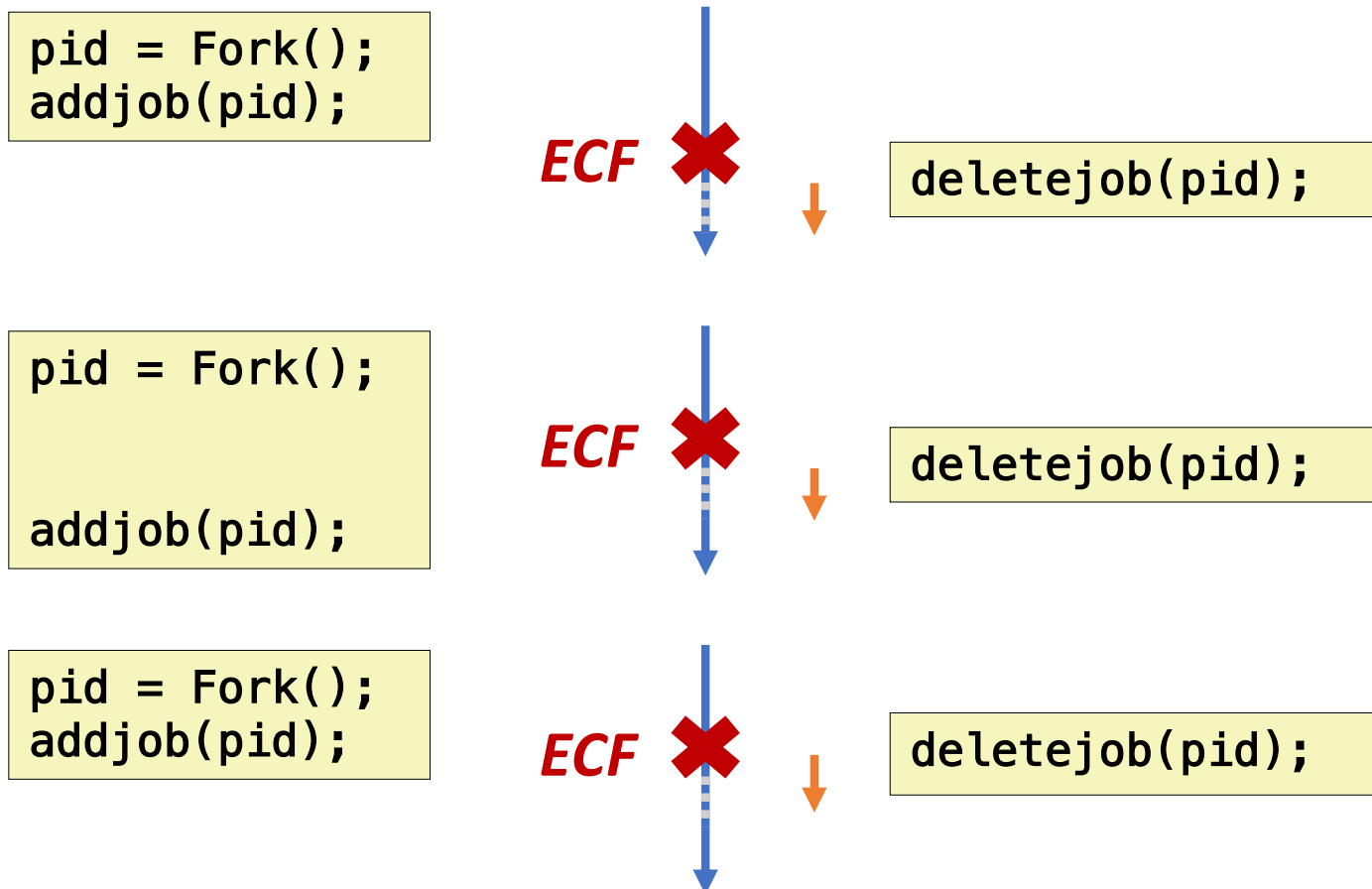
```
void handler(int sig)
{
    int olderrno = errno;
    pid_t pid;
    while ((pid = waitpid(-1, NULL, 0)) > 0) { /* Reap child */
        deletejob(pid); /* Delete the child from the job list */
    }
    if (errno != ECHILD)
        Sio_error("waitpid error");
    errno = olderrno;
}
```

```
int main(int argc, char **argv)
{
    while (1) {
        if ((pid = Fork()) == 0) { /* Child process */
            Execve("/bin/date", argv, NULL);
        }
        addjob(pid); /* Add the child to the job list */
    }
}
```

procmask0.c

# Concurrent Flows to Lead Races (Cnt'd)

- `main()` and `handler()` are concurrent flows
- Any schedule can be chosen





# Synchronizing Flows to Avoid Races

- Blocking prevents processes from being interleaved
- But a subtle synchronization error still exists

```
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

# Synchronizing Flows to Avoid Races (Cnt'd)

- Blocking prevents processes from being interleaved
- But a subtle synchronization error still exists

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

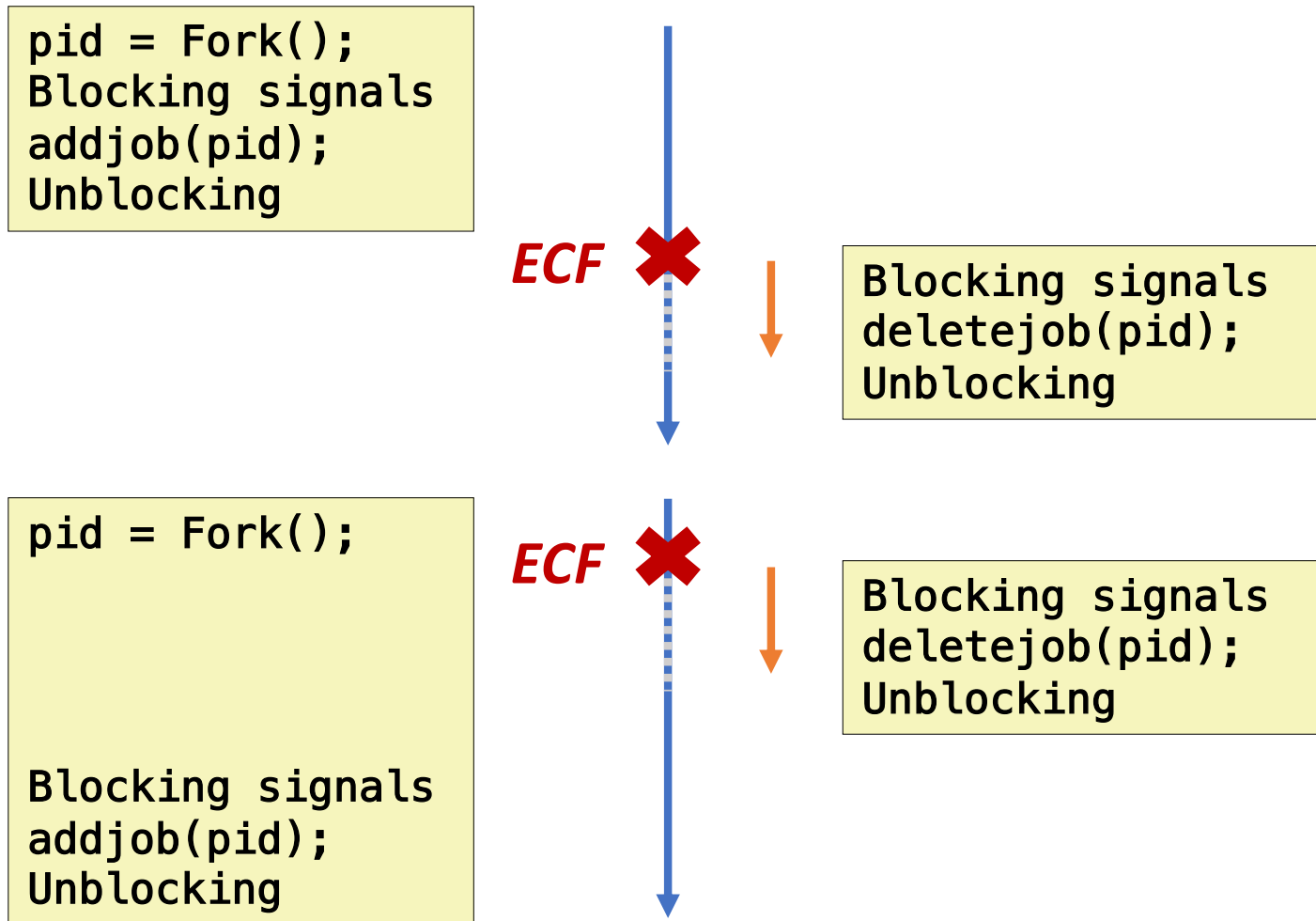
    Sigfillset(&mask_all);
    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    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 (Cnt'd)

- `addjob()` & `deletejob()` require synchronization



# Corrected Shell Program to Avoid 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 **explicitly waiting for** `SIGCHLD` to arrive
  - Now, `sigchld_handler()` uses `wait()`

```
volatile sig_atomic_t pid;

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

void sigint_handler(int s)
{
    /.... /
}
```

waitforsignal.c

# Explicitly Waiting for Signals (Cnt'd)

```
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);
}
```

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

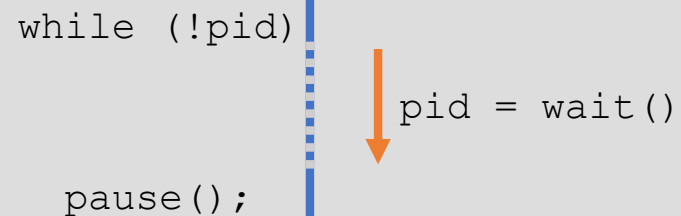
waitforsignal.c

# Explicitly Waiting for Signals (Cnt'd)

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

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

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



```
while (!pid)  
  
    pause();
```

pid = wait()

- Solution: `sigsuspend()`

# Waiting for Signals with `sigsuspend()`

- `int sigsuspend(const sigset_t *mask)`
- Equivalent to atomic (or 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

# 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