[8.5] Signals

-by Sandesh Paudel

> operating system uses **exceptions** to support a form of **exceptional control flow** known as **context switch.**
>**linux signal:** a higher-level software form of exceptional control flow that allows processes and the kernel to interrupt other processes.

> **signal:** a small message that notifies a process that an **event** of some type has occurred in the system. Textbook 757 has 30 different types of signals that are supported on Linux.

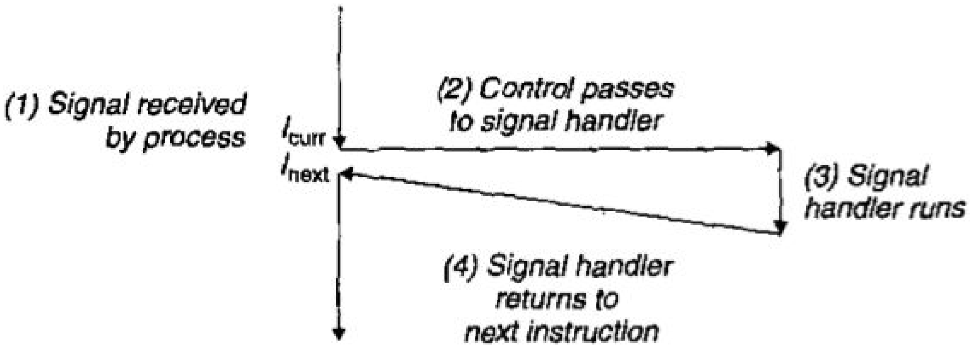
ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	User typed ctrl-c
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

> low-level hardware exceptions are processed by the **kernel's exception handlers** and are not normally visible to user processes. **Signals** provide a mechanism for exposing the occurrence of **such exceptions** to user processes.
For eg, if a process attempts to divide by zero, then the kernel sends a **SIGFPE** signal. Other signals correspond to other such events.

> If you type Ctrl+C while a process is running in the foreground, then the kernel sends a **SIGINT** to each process in the foreground process group. A process can forcibly terminate another process by sending it a **SIGKILL** signal.

Sending a signal. The kernel *sends* (*delivers*) a signal to a destination process by updating some state in the context of the destination process. The signal is delivered for one of two reasons: (1) The kernel has detected a system event such as a divide-by-zero error or the termination of a child process. (2) A process has invoked the `kill` function (discussed in the next section) to explicitly request the kernel to send a signal to the destination process. A process can send a signal to itself.

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. The process can either ignore the signal, terminate, or *catch* the signal by executing a user-level function called a *signal handler*. Figure 8.27 shows the basic idea of a handler catching a signal.

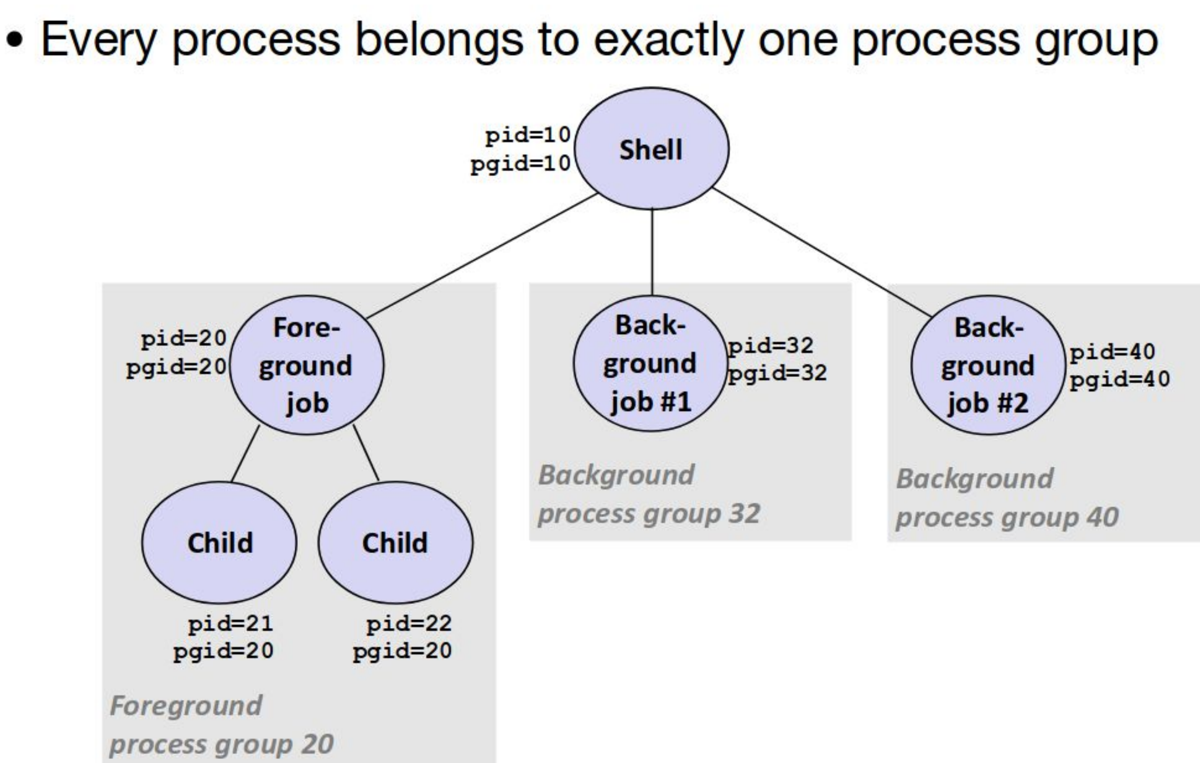


> Pending Signal: a signal that has been sent but not yet received. There can be at most one pending signal of a particular type at a time.

- If a process has a pending signal of type k, then any subsequent signals of type k sent to that process are not queued; they are **discarded.**
- A process can selectively **block** the receipt of certain signals. The blocked signal will be delivered but not received until the process unblocks it.
- For each process, the kernel maintains a set of pending signals in the **pending bit vector**, and a set of blocked signals in the **blocked bit vector.**
- The kernel sets bit k in pending whenever a signal of type k is delivered and clears bit k in pending whenever a signal of type k is received.
-

> Sending Signals

- **Process groups:** every process belongs to exactly one process group identified by a positive integer **process group ID.** It can be accessed with **getpgrp().**



-

```
#include <unistd.h>

pid_t getpgrp(void);
```

Returns: process group ID of calling process

-
- By default, a child process belongs to the same process group as its parent. A process can change the process group of itself or another process by using the **setpgid** function [from pid to pgid]:

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

Returns: 0 on success, −1 on error

-

Sending Signals with the kill Function

Processes send signals to other processes (including themselves) by calling the kill function.

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

Returns: 0 if OK, −1 on error

If pid is greater than zero, then the kill function sends signal number sig to process pid. If pid is equal to zero, then kill sends signal sig to every process in the process group of the calling process, including the calling process itself. If pid is less than zero, then kill sends signal sig to every process in process group |pid| (the absolute value of pid). Figure 8.29 shows an example of a parent that uses the kill function to send a SIGKILL signal to its child.

Sending Signals with the /bin/kill Program

The /bin/kill program sends an arbitrary signal to another process. For example, the command

```
linux> /bin/kill -9 15213
```

sends signal 9 (SIGKILL) to process 15213. A negative PID causes the signal to be sent to every process in process group PID. For example, the command

```
linux> /bin/kill -9 -15213
```

sends a SIGKILL signal to every process in process group 15213. Note that we use the complete path /bin/kill here because some Unix shells have their own built-in kill command.

Sending Signals with the alarm Function

A process can send SIGALRM signals to itself by calling the alarm function.

```
#include <unistd.h>

unsigned int alarm(unsigned int secs);
```

Returns: remaining seconds of previous alarm, or 0 if no previous alarm

> **Sending Signals via Keyboard:**
  - **Job:** abstraction to represent the **processes** that are created as a result of evaluating single command line. There can be at most one foreground job but multiple or zero background jobs.
  - **Pipe:** output of one process acts as the input of another. `linux> ls | sort`
  -

Receiving Signals

- Kernel is returning from an exception handler and is ready to pass control to process p
- Kernel computes the set of pending & nonblocked signals for process p (PNB set)
- If (PNB is empty)
  - 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 stops until restarted by a SIGCONT signal
  - The process ignores the signal

Installing Signal Handlers

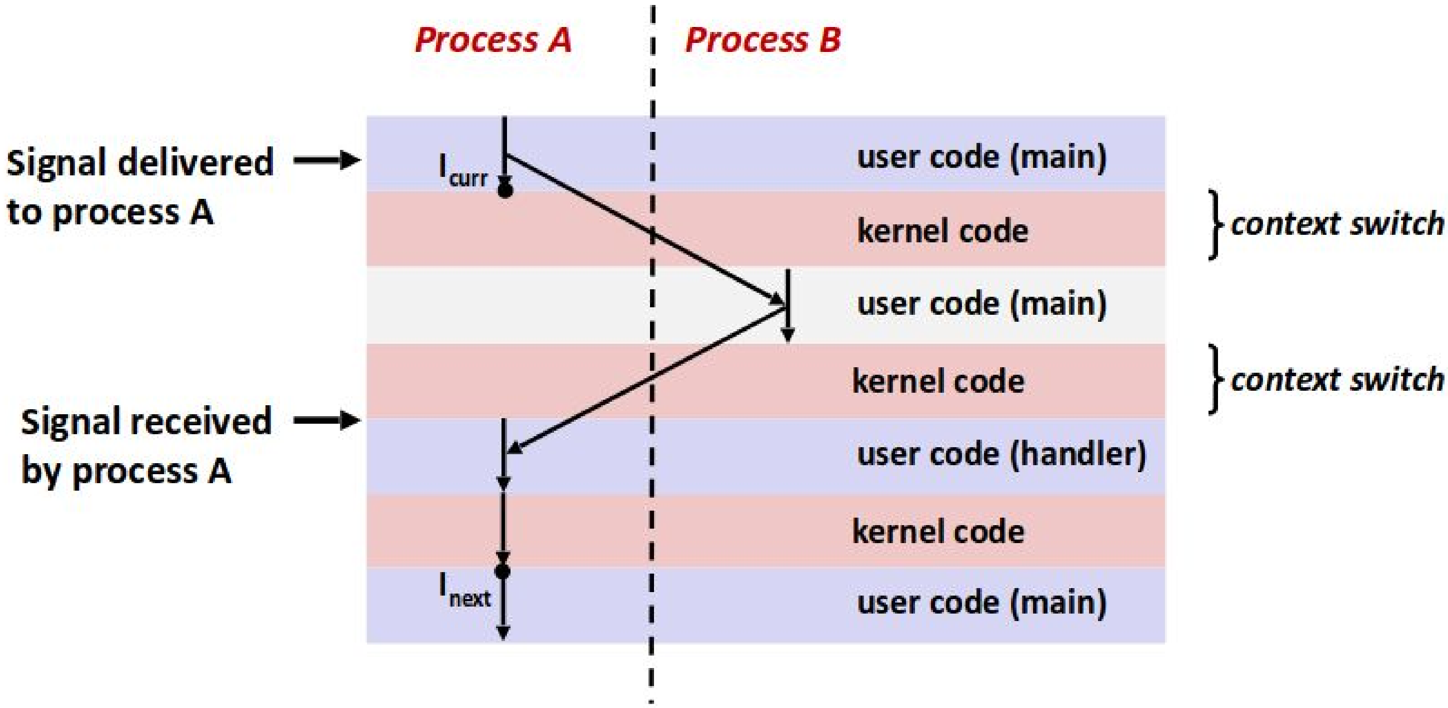
- The signal function modifies the default action associated with the receipt of signal signum:
  - `handler_t *signal(int signum, handler_t *handler)`

- 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 function (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

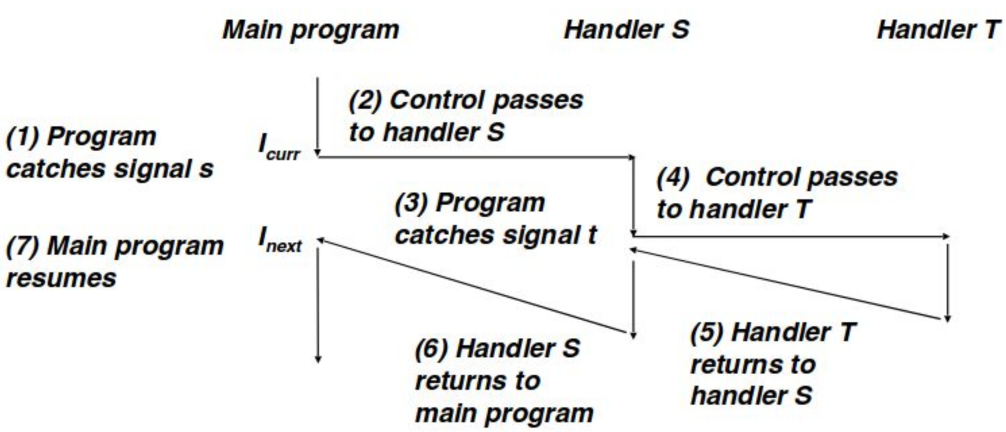
```c
void sigint_handler(int sig) /* SIGINT handler */
{
    printf("So you think you can stop the bomb with ctrl-c, do you?\n");
    sleep(2);
    printf("Well...");
    fflush(stdout);
    sleep(1);
    printf("OK. :-)\n");
    exit(0);
}

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;
}
```

sigint.c



Nested Signal Handlers

- Handlers can be interrupted by other handlers



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);
```

- Explicit blocking and unblocking signal
  - `sigprocmask` function
  - `sigemptyset` – Create empty set
  - `sigfillset` – Add every signal number to set
  - `sigaddset` – Add signal number to set
  - `sigdelset` – Delete signal number from set

Safe Signal Handling

- Handlers are tricky because they are concurrent with main program and may share the same global data structures.

```
static int x = 5;
void handler(int sig)
{
    x = 10;
}

int main(int argc, char **argv)
{
    int pid;
    Signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    if (x == 5)
        y = x * 2; // You'd expect y == 10
    exit(0);
}
```

What if the following happens:

- Parent process executes and finishes `if (x == 5)`
- Context switch to child, which then terminates, sends a SIGCHLD signal
- Another context switch back to parent, and now the kernel needs to execute the SIGCHLD handler
- When return to parent process, **y == 20**!

Safe Signal Handling

- Handlers are tricky because they are concurrent with main program and may share the same global data structures.
  - Programmers have no control over the execution ordering between the main program and the signal handler, that is:
    - when a signal happens/delivers (depends on user or other process)
    - when the signal handler will be executed (depends on kernel)
  - If not careful, shared data structures can be corrupted

Fixing the Signal Handling Bug

```
static int x = 5;
void handler(int sig)
{
    x = 10;
}

int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;
    sigfillset(&mask_all);
    signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
    if (x == 5)
        y = x * 2; // You'd expect y == 10
    Sigprocmask(SIG_SETMASK, &prev_all, NULL);

    exit(0);
}
```

- Block all signals before accessing a shared, global data structure.
- Can't use a lock (later in this course)

Async-Signal-Safety

- Function is async-signal-safe if either reentrant (e.g., no access to global variables) 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

Another Unsafe Signal Handler Example

- Assume a program wants to do the following:
  - The parent creates multiple child processes
  - When each child process is created, add the child PID to a queue
  - When a child process terminates, the parent process removes the child PID from the queue
- One possible implementation:
  - An array for keeping the child PIDs
  - Use a loop to fork child, and add PID to the array after fork
  - Install a handler for SIGCHLD in parent process
  - The SIGCHLD handler removes the child PID

First Attempt

```c
void handler(int sig)
{
    pid_t pid;

    while ((pid = wait(NULL)) > 0) { /* Reap child */
        deletejob(pid); /* Delete the child from the job list */
    }
}

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

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

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

Second Attempt

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

    sigfillset(&mask_all);
    while ((pid = wait(NULL)) > 0) {
        sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
        deletejob(pid);
        sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
}
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) {
            Execve("/bin/date", argv, NULL);
        }
        sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
        addjob(pid);
        sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
    exit(0);
}
```

The following can happen:

- Child runs, and terminates
- Kernel sends SIGCHLD
- Context switch to parent, but before it can run, kernel has to handle SIGCHLD first
- The handler deletes the job, which does nothing
- The parent process resumes and adds a terminated child to job list

Third Attempt (The Correct One)

```
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);
}
```