

Lecture 5: Process Termination and Signals

CS 3281

Daniel Balasubramanian, Shervin Hajiamini

Review

- System calls for process creation, program loading, and process reaping:
 - `fork()`: creates an (almost) identical copy of the calling process
 - `exec()`: loads a new program into the address space of the calling process
 - `wait()`: allows a parent process to reap a child process
- Typical sequence: `fork()` followed by `exec()`
 - `fork()` makes a new process
 - `exec()` blows away the address space of that new process

Process termination

- When a process terminates, the kernel does not remove it from the system immediately
 - Minimal information is kept (in a terminated state) until the process is reaped by its parent
- A process can terminate in two ways
 - Normal termination:
 - This happens when you do something like `exit(0)` or return from main
 - "Successful": usually means the process returned 0
 - "Unsuccessful": usually means the process didn't return 0
 - Abnormal termination: the process was terminated by a signal
 - Example: a process dereferenced a null pointer, which cause the OS's page fault handler to send it the SIGSEGV signal, whose default action is to terminate the process
 - Important: "normal" termination does not mean a program ran "successfully"! It mostly means that it didn't get killed by a signal.

Typical flow

- A parent process creates a child process via `fork()`
- The child process runs a new program via `exec()`
- The parent calls `wait()` to wait for the child program to terminate
- The parent uses the return information from `wait()` to determine whether the process exited normally or abnormally
 - If it exited normally, the exit status information can be obtained
 - If it exited abnormally, the signal that killed it can be obtained

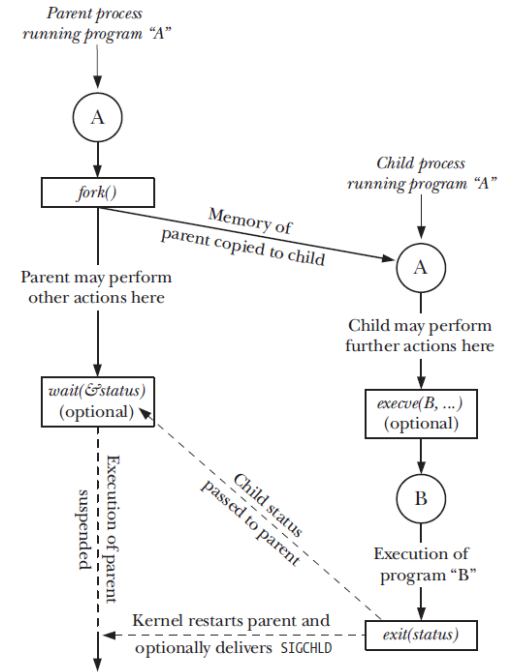


Figure 24-1: Overview of the use of `fork()`, `exit()`, `wait()`, and `execve()`

The wait() system call

- The wait() system call is used by a parent to wait for its children to have a change in state
 - What is a change in state?
 - The child terminated
 - The child was stopped by a signal
 - The child was resumed by a signal
- If the child was terminated, performing wait() allows the system to release all resources associated with the child
 - Otherwise the child remains in the “zombie” state
 - Minimal information is kept around

waitpid: Waiting for a Specific Process

- `pid_t waitpid(pid_t pid, int &status, int options)`
 - Suspends current process until specific process terminates
 - Various options (see textbook)

```
void fork11() {  
    pid_t pid[N];  
    int i;  
    int child_status;  
  
    for (i = 0; i < N; i++)  
        if ((pid[i] = fork()) == 0)  
            exit(100+i); /* Child */  
    for (i = N-1; i >= 0; i--) {  
        pid_t wpid = waitpid(pid[i], &child_status, 0);  
        if (WIFEXITED(child_status))  
            printf("Child %d terminated with exit status %d\n",  
                wpid, WEXITSTATUS(child_status));  
        else  
            printf("Child %d terminate abnormally\n", wpid);  
    }  
}
```

forks.c

Signals

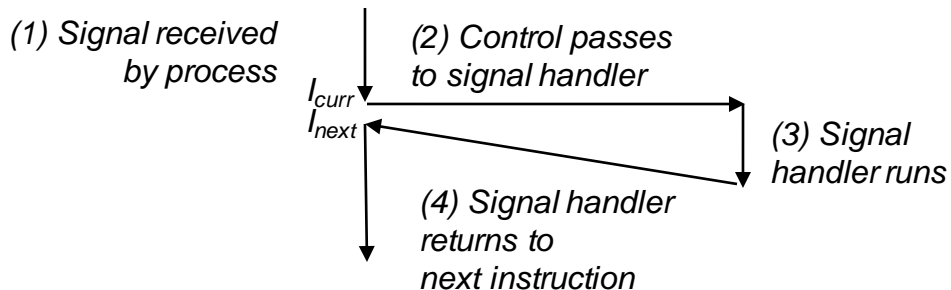
- Signals are a technique used to notify a process that some condition has occurred.
 - Also called software interrupts.
 - Example: if a process divides by zero, the signal whose name is SIGFPE (floating-point exception) is sent to the process.
- Processes can deal with signals in three ways:
 - Ignore the signal.
 - Let the default action occur. The default action for most signals is to terminate the process.
 - Catch the signal. We do this by telling the kernel to call a function of ours whenever the signal occurs. We do this by registering a signal handler.

Transferring signals

- The transfer of a signal to a destination process occurs in two distinct steps:
 - Sending a signal.
 - The kernel 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:
 - The kernel detected a system event (such as divide by zero)
 - A process invoked the kill function to send a signal to the destination process.
 - Receiving a signal.
 - The destination process receives a signal when it is forced by the kernel to react in some way to the delivery of the signal.
 - This reaction is one of the three choices described above (ignore, handle, or let default action occur).

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:



Example of Signals: Shell Homework

- The simple shell in your homework correctly waits for and reaps foreground jobs
- But what if we wanted to add 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
 - This happens via a signal

Signal Concepts: Pending and Blocked Signals

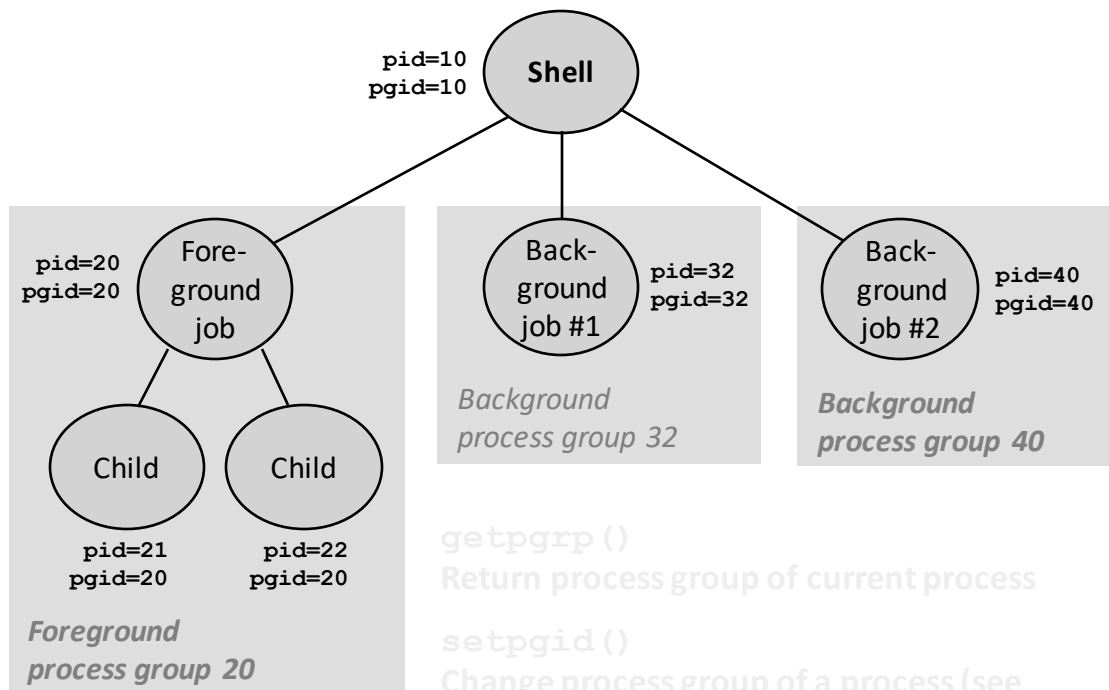
- A signal is *pending* if sent but not yet received
 - There can be at most one pending signal of any particular type
 - Important: Signals are not queued
 - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded
- A process can *block* the receipt of certain signals
 - Blocked signals can be delivered, but will not be received until the signal is unblocked
- A pending signal is received at most once

Signal Concepts: 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
 - Also referred to as the *signal mask*.

Sending Signals: Process Groups

- Every process belongs to exactly one process group



Sending Signals with `/bin/kill` Program

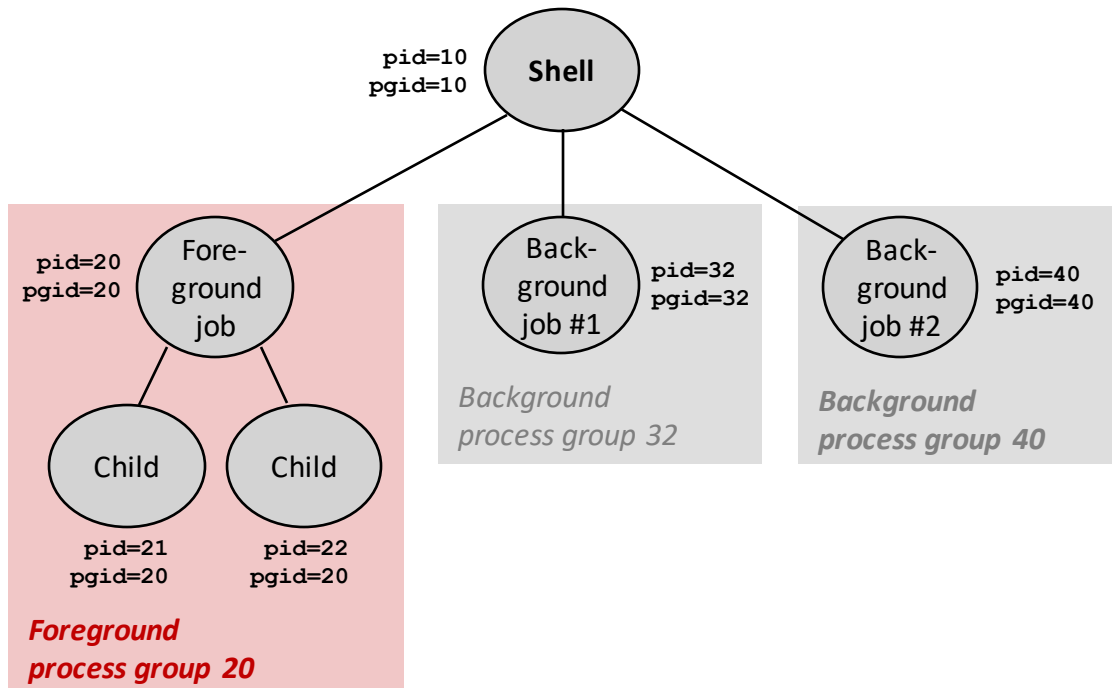
- `/bin/kill` program sends arbitrary signal to a process or process group
- Examples
 - `/bin/kill -9 24818`
Send SIGKILL to process 24818
 - `/bin/kill -9 -24817`
Send SIGKILL to every process in process group 24817

```
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) 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 (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;
    int child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            /* Child: Infinite Loop */
            while(1)
                ;
        }

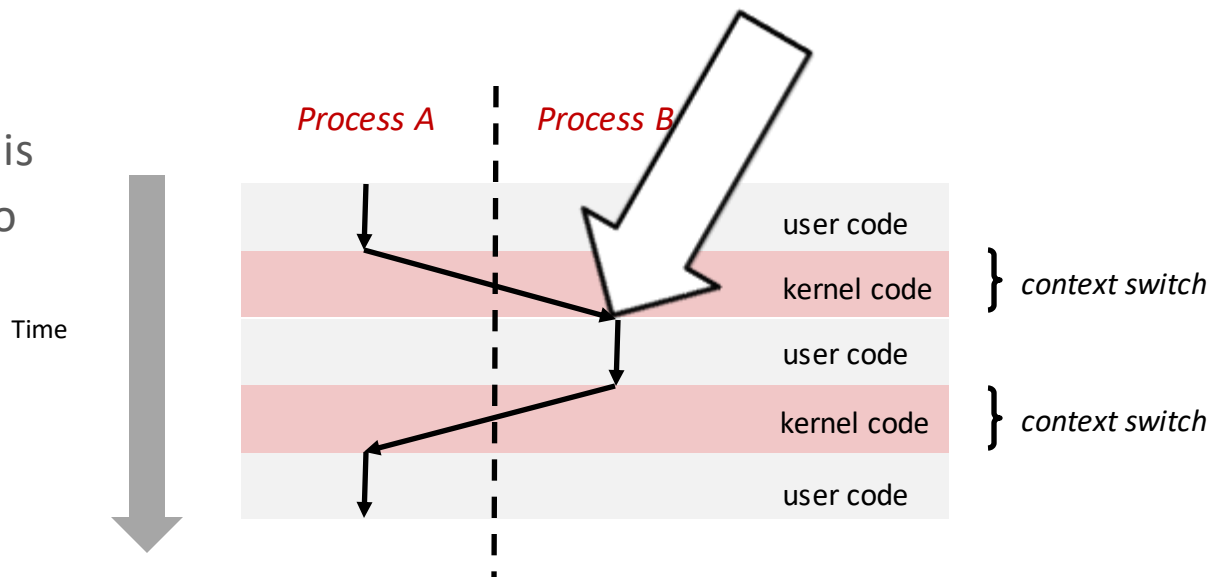
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    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);
    }
}
```

forks.c

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 $pnb = \text{pending} \& \sim\text{blocked}$
 - 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 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`:
 - `typedef void (*sighandler_t)(int);`
 - `sighandler_t signal(int signum, sighandler_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);
}

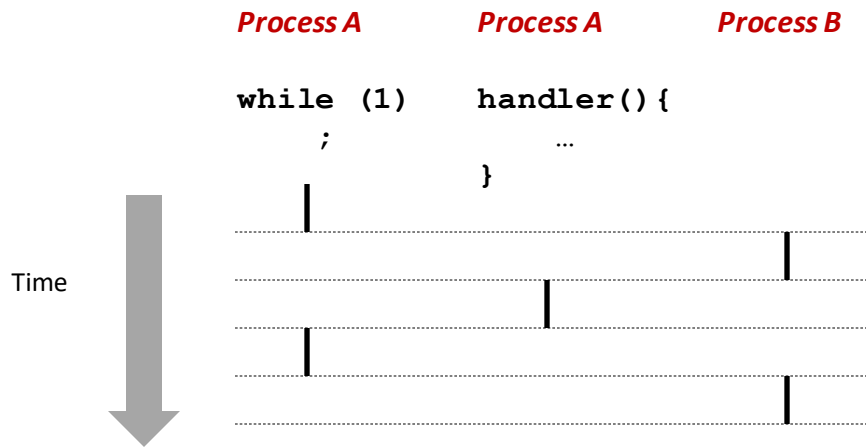
int main()
{
    /* Install the SIGINT handler */
    if (signal(SIGINT, sigint_handler) == SIG_ERR)
        unix_error("signal error");

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

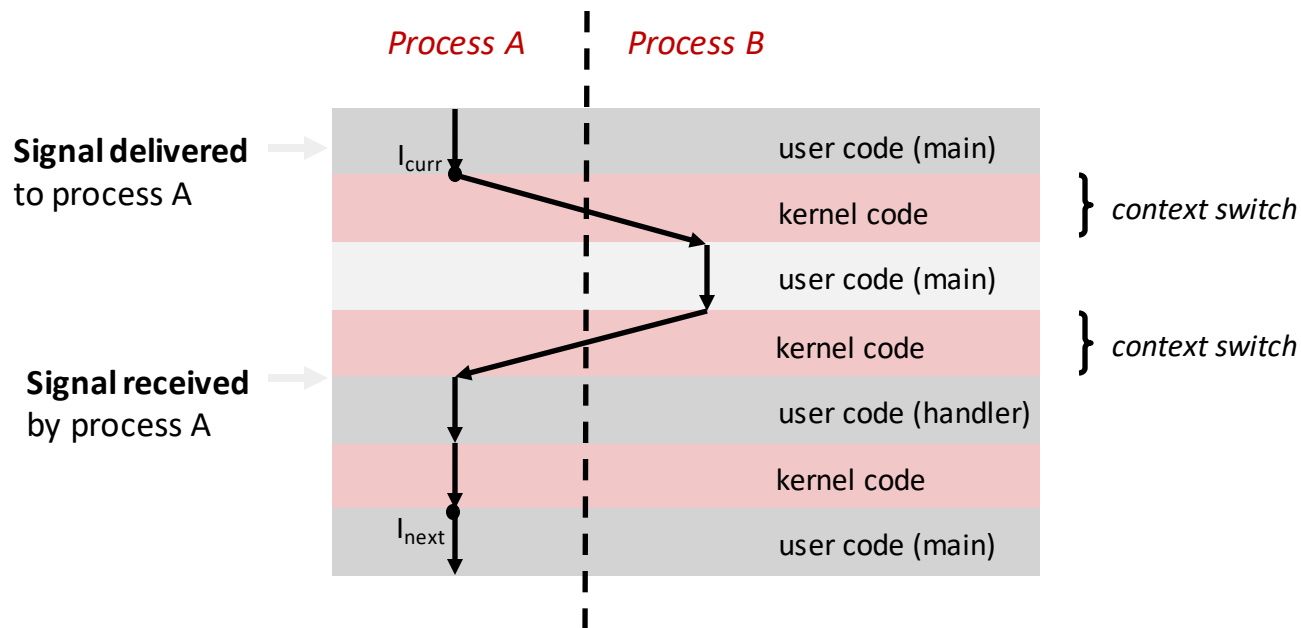
    return 0;
}
```

Signals Handlers as Concurrent Flows

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

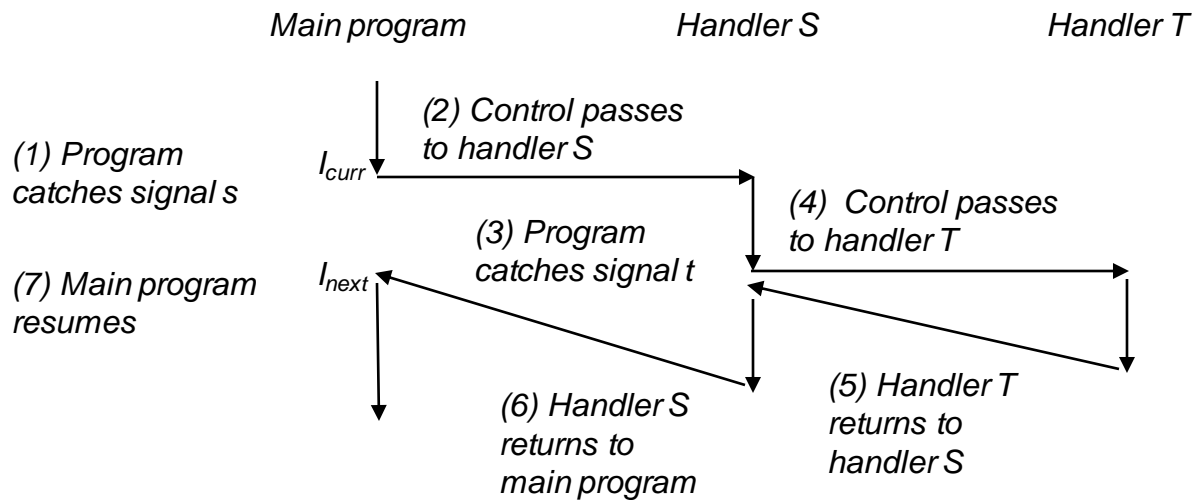


Another View of Signal Handlers as Concurrent Flows



Nested Signal Handlers

- Handlers can be interrupted by other handlers



Core dumps

- The default action for some signals, like SIGABRT (abnormal termination) or SIGSEGV (invalid memory reference), is to terminate the process and generate a core dump.
- A core dump file contains an image of the virtual memory of the process, which can be loaded into a debugger in order to inspect the state of the process at the time that it terminated.