

CSC 252: Computer Organization

Spring 2020: Lecture 20

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Announcement

- Programming assignment 4 is out
 - Details: <https://www.cs.rochester.edu/courses/252/spring2020/labs/assignment4.html>
 - Due on **Apr. 17**, 11:59 PM
 - You (may still) have 3 slip days

5	6	7	8	9	10	11
				Today		
12	13	14	15	16	17	18
					Due	

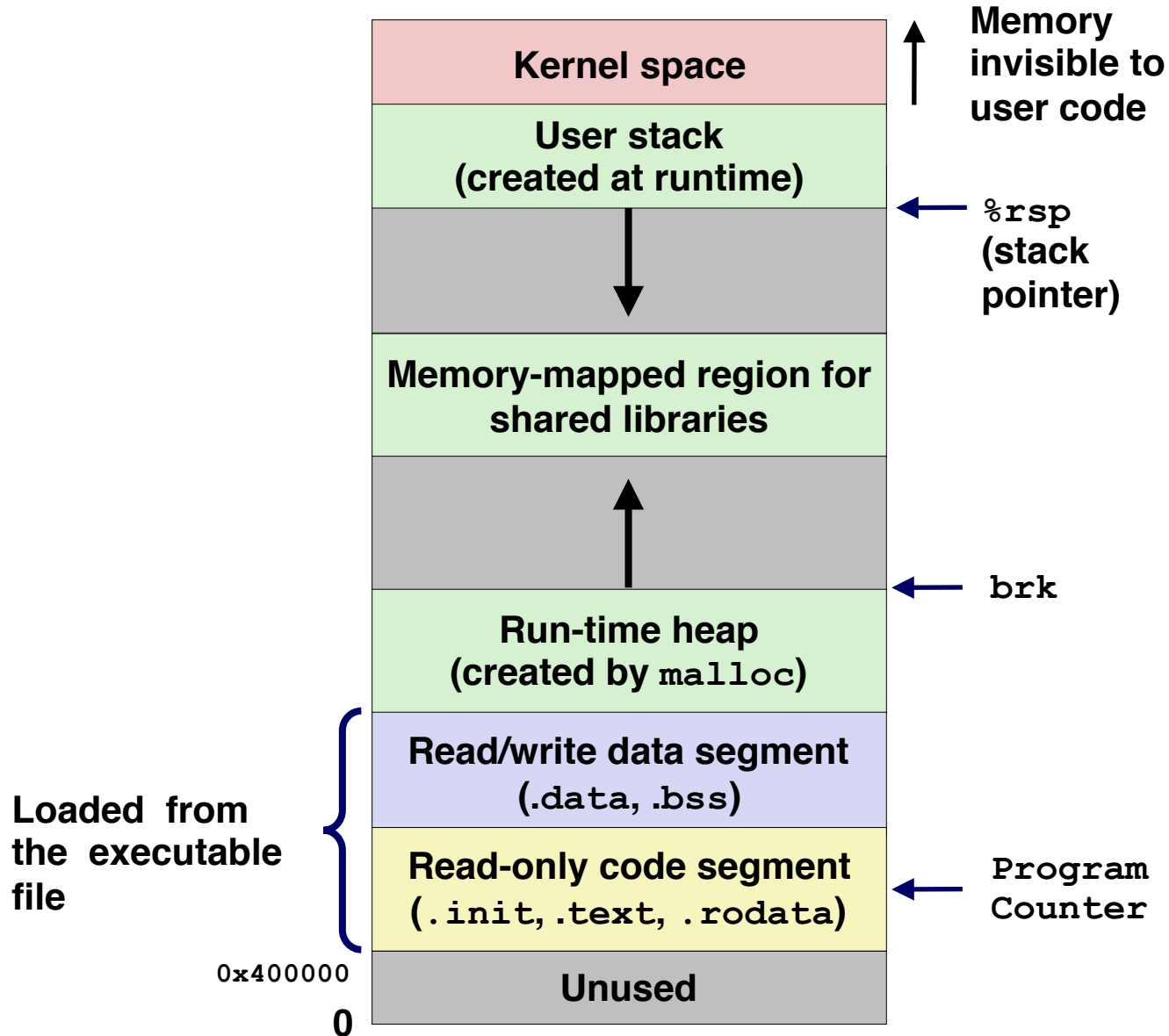
Announcement

- Programming assignment 4 will be done in C language. Seek help from TAs.
- TAs are best positioned to answer your questions about programming assignments!!!
- Programming assignments do NOT repeat the lecture materials. They ask you to synthesize what you have learned from the lectures and work out something new.

Today

- Signals: The Way to Communicate with Processes

Address Space



Signals

- A **signal** is a small message that notifies a process that an event of some type has occurred in the system
 - Sent from the **OS kernel**
 - Could be requested by another process, by user, or automatically by the kernel
 - Signal type is identified by small integer ID's (1-30)

Signals

- A **signal** is a small message that notifies a process that an event of some type has occurred in the system
 - Sent from the **OS kernel**
 - Could be requested by another process, by user, or automatically by the kernel
 - Signal type is identified by small integer ID's (1-30)

<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:

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:

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:
 - Exception: divide-by-zero (SIGFPE)

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:
 - Exception: divide-by-zero (SIGFPE)
 - Interrupt: user pressing Ctrl + C (SIGINT)

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:
 - Exception: divide-by-zero (SIGFPE)
 - Interrupt: user pressing Ctrl + C (SIGINT)
 - The termination of a child process (SIGCHLD)

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:
 - Exception: divide-by-zero (SIGFPE)
 - Interrupt: user pressing Ctrl + C (SIGINT)
 - 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.

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:
 - Exception: divide-by-zero (SIGFPE)
 - Interrupt: user pressing Ctrl + C (SIGINT)
 - 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.
 - Note: `kill` doesn't mean you are going to kill the target process. It is just a system call that allows you to send signals. Of course the signal you send could be SIGKILL.

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:

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)

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

Signal Concepts: Receiving a Signal

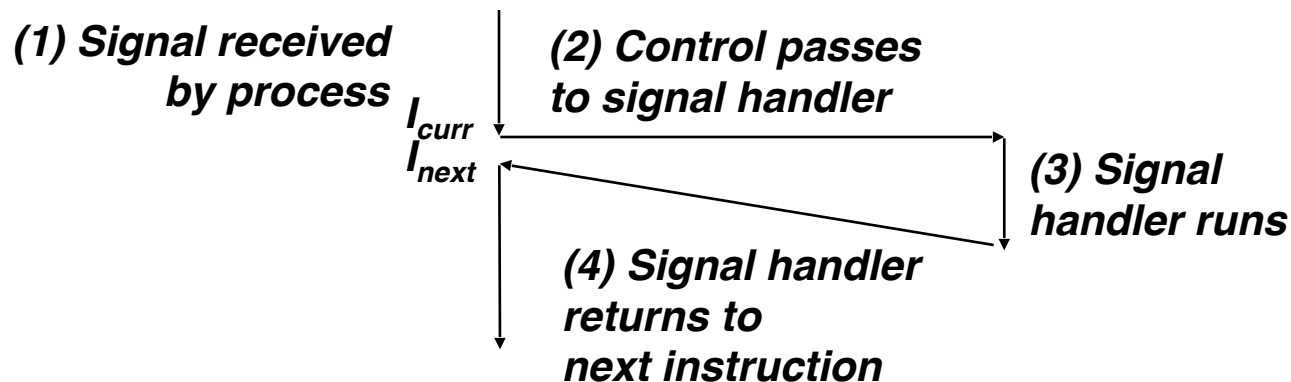
- A destination process **receives** a signal when it is forced by the kernel to react in some way to the delivery of the signal
- Some possible ways to react:
 - **Ignore** the signal (do nothing)
 - **Terminate** the process
 - **Catch** the signal by executing a user-level function called **signal handler**

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
 - **Catch** the signal by executing a user-level function called **signal handler**
 - Similar to a hardware exception handler being called in response to an asynchronous interrupt:

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
 - **Catch** the signal by executing a user-level function called **signal handler**
 - Similar to a hardware exception handler being called in response to an asynchronous interrupt:



Sending Signals with `/bin/kill` Program

- `/bin/kill` program sends arbitrary signal to a process
- Examples
 - `/bin/kill -9 24818`
Send SIGKILL to process 24818
 - `/bin/kill` itself doesn't kill the process. 9 is the ID for the SIGKILL signal, which kills the process

```
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
```

Sending Signals with `/bin/kill` Program

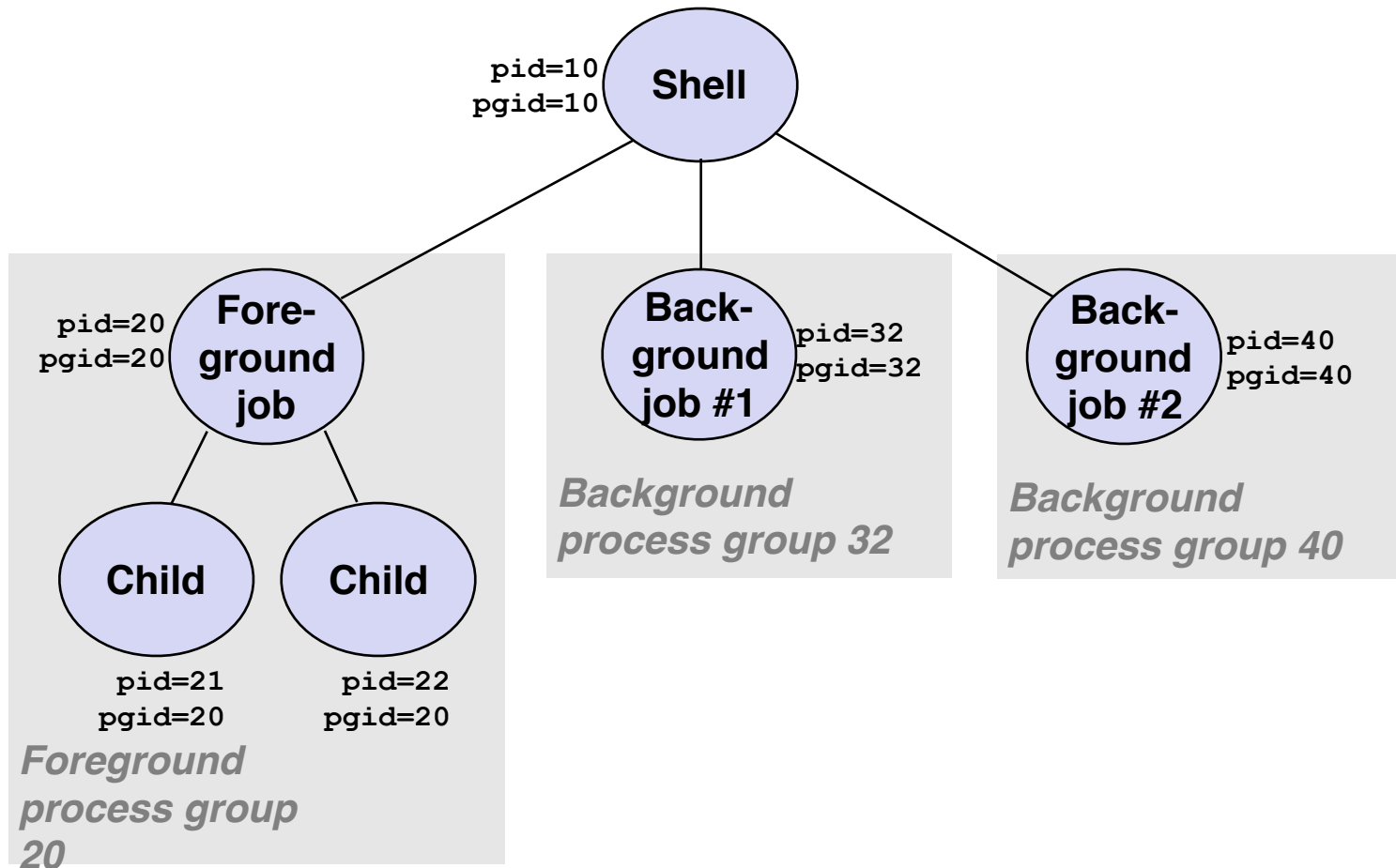
- `/bin/kill` program sends arbitrary signal to a process
- Examples
 - `/bin/kill -9 24818`
Send SIGKILL to process 24818
 - `/bin/kill` itself doesn't kill the process. 9 is the ID for the SIGKILL signal, which kills the process

```
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
```

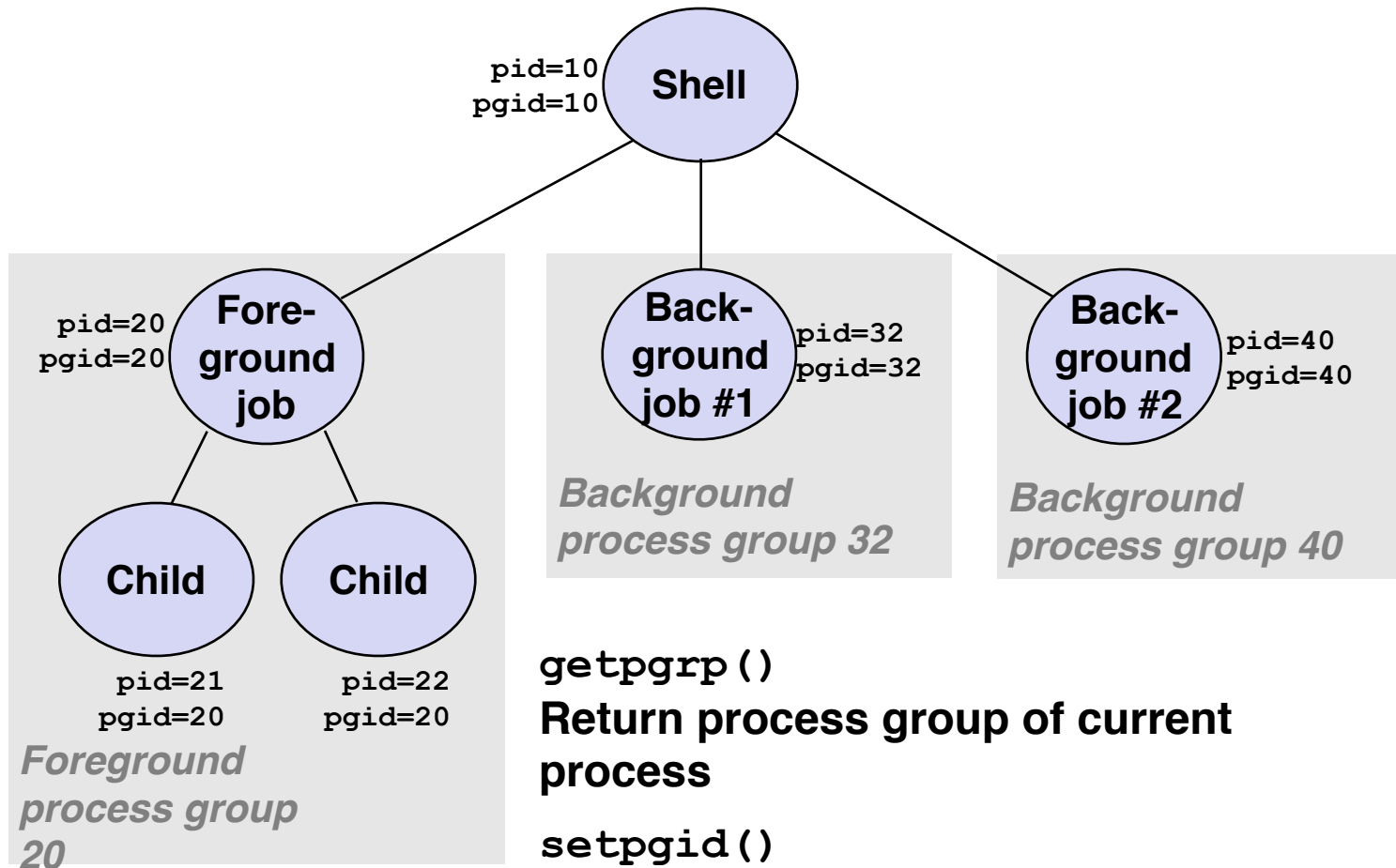
Process Groups

- Every process belongs to exactly one process group



Process Groups

- Every process belongs to exactly one process group



`getpgrp()`

Return process group of current process

`setpgid()`

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

Sending Signals with `/bin/kill` Program

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

- `/bin/kill` program sends arbitrary signal to a process or process group
- Examples
 - `/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 causes the kernel to send a SIGINT to every process in the foreground process group.
 - SIGINT – default action is to terminate each process
- Typing ctrl-z causes the kernel to send a SIGTSTP to every job in the foreground process group.
 - 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 from the Keyboard

- Can you guess how Ctrl + C might be implemented?

Sending Signals from the Keyboard

- Can you guess how Ctrl + C might be implemented?
 - Ctrl + C sends a keyboard interrupt to the CPU, which triggers an interrupt handler

Sending Signals from the Keyboard

- Can you guess how Ctrl + C might be implemented?
 - Ctrl + C sends a keyboard interrupt to the CPU, which triggers an interrupt handler
 - The interrupt handler, executed by the kernel, triggers certain piece of the kernel, which generates the signal, which is then delivered to the target process

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

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
 - That is: *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 pending signal is received at most once

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
 - That is: *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 pending signal is received at most once
- A process can **block** the receipt of certain signals

Pending and Blocked Signals

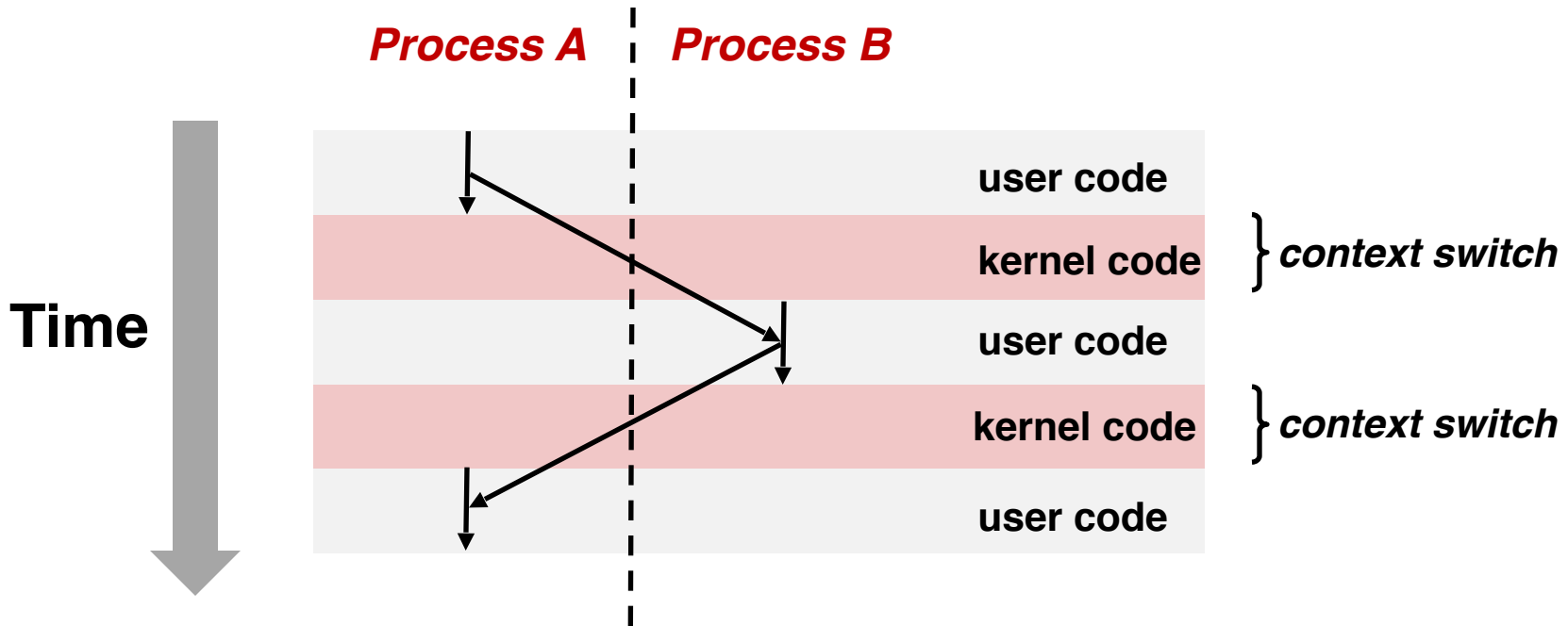
- A signal is **pending** if sent but not yet received
 - There can be at most one pending signal of any particular type
 - That is: *Signals are not queued*
 - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded
 - A pending signal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

Receiving Signals

- Kernel handles signals delivered to a process p when it switches p from kernel mode to user mode (e.g., after a context switch)



Receiving Signals

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

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)

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)

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

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

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

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

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

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)`

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:

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`

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`

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)**

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`

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

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

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

```
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);
}

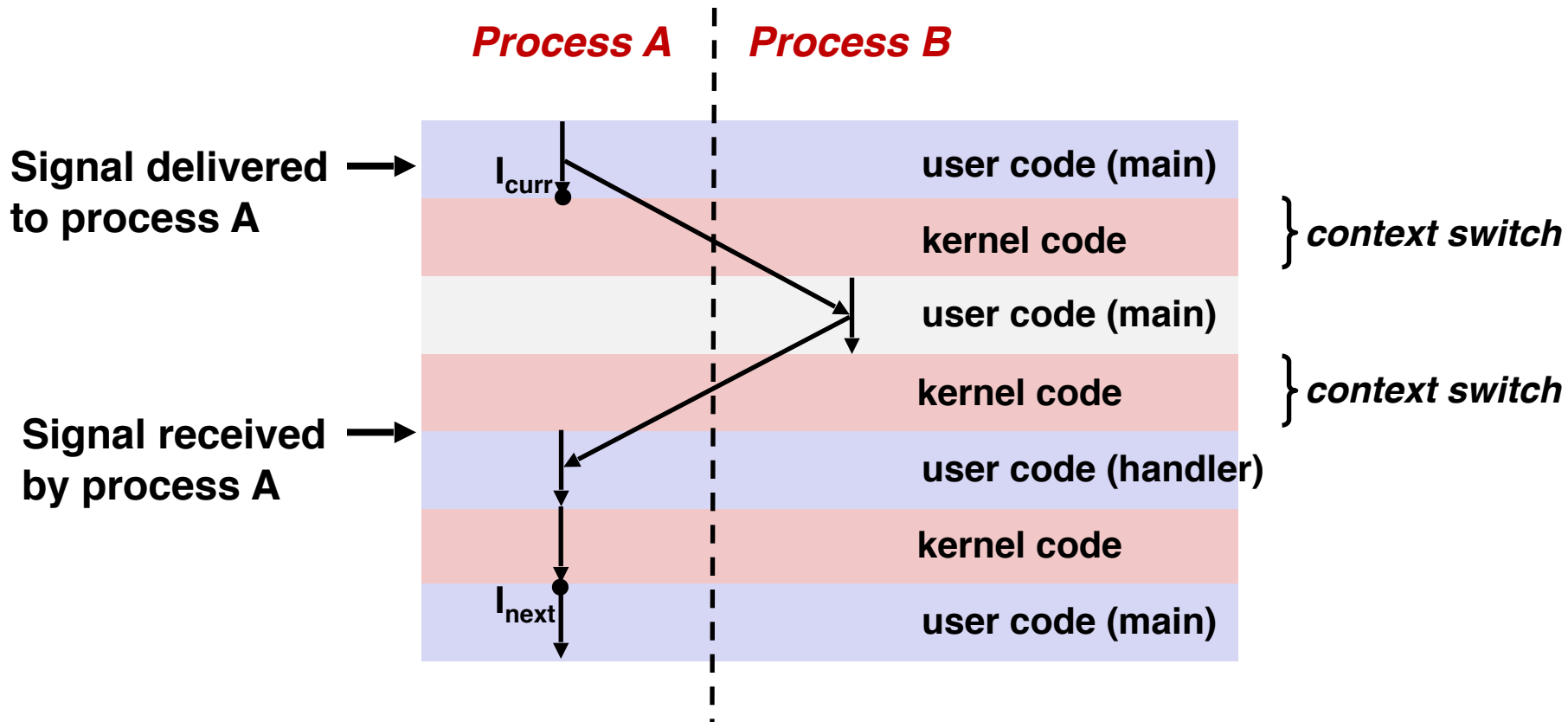
int main()
{
    /* Install the SIGINT handler */
    if (signal(SIGINT, sigint_handler) == SIG_ERR)
        unix_error("signal error");

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

    return 0;
}
```

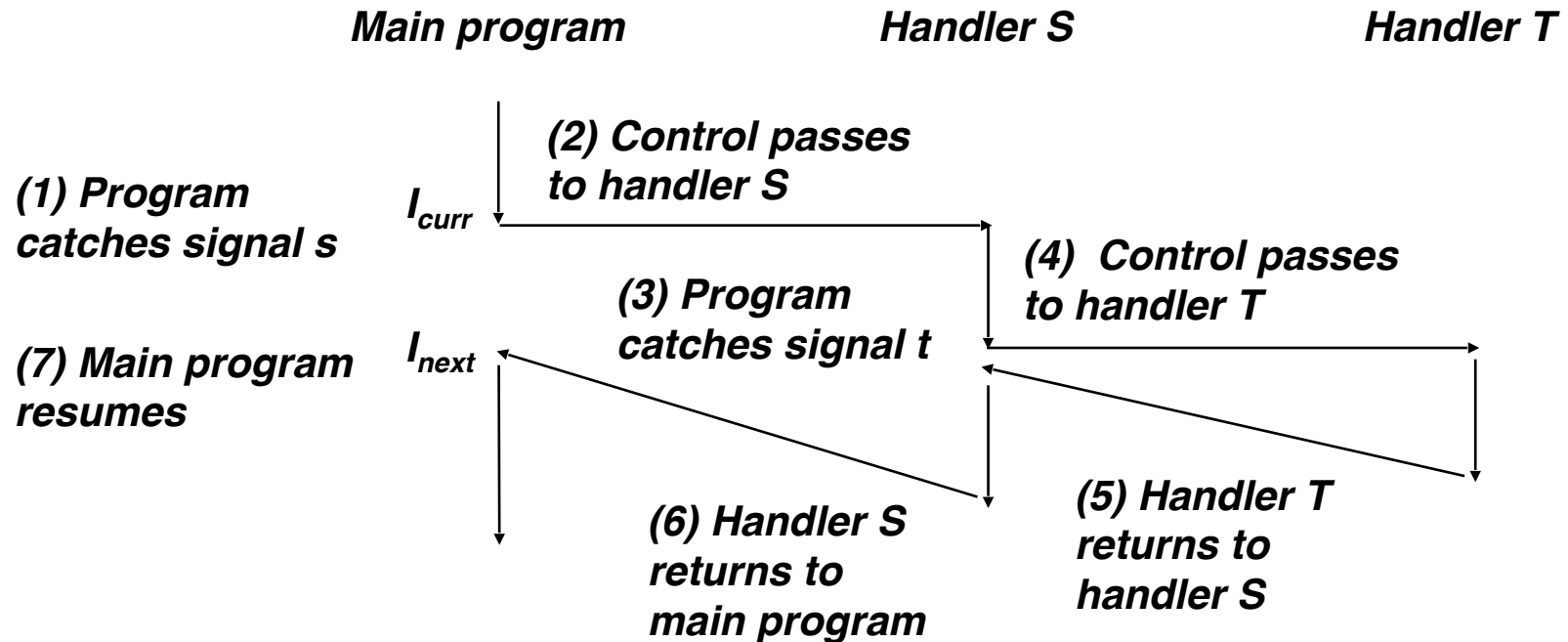
sigint.c

View Signal Handlers as Concurrent Flows



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**.

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, y = 0;
    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);
}
```

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, y = 0;
    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:

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, y = 0;
    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)`

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, y = 0;
    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

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, y = 0;
    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

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, y = 0;
    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.

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

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”

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:

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`

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:

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`

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

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

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

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

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

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);
        }
        /* Add the child to the job list */
        addjob(pid);
    }
    exit(0);
}
```

First Attempt

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

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

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);
        }
        /* Add the child to the job list */
        addjob(pid);
    }
    exit(0);
}
```

The following can happen:

First Attempt

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

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

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);
        }
        /* Add the child to the job list */
        addjob(pid);
    }
    exit(0);
}
```

The following can happen:

- Child runs, and terminates

First Attempt

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

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

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);
        }
        /* Add the child to the job list */
        addjob(pid);
    }
    exit(0);
}
```

The following can happen:

- Child runs, and terminates
- Kernel sends SIGCHLD

First Attempt

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

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

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);
        }
        /* Add the child to the job list */
        addjob(pid);
    }
    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

First Attempt

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

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

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);
        }
        /* Add the child to the job list */
        addjob(pid);
    }
    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

First Attempt

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

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

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);
        }
        /* Add the child to the job list */
        addjob(pid);
    }
    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

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);
}
```


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);
        }
        addjob(pid); /* Add the child to the job list */
        Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
    }
    exit(0);
}
```