

CSC 252: Computer Organization

Spring 2025: Lecture 19

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Today

- Process Control
- Signals: The Way to Communicate with Processes

Creating Processes

- Parent process creates a new child process by calling `fork`
- Child get an identical (but separate) copy of the parent's (virtual) address space (i.e., same stack copies, code, etc.)
- `int fork(void)`
 - Returns **0** to the child process
 - Returns **child's PID** to the parent process

fork Example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        exit(0);
    }

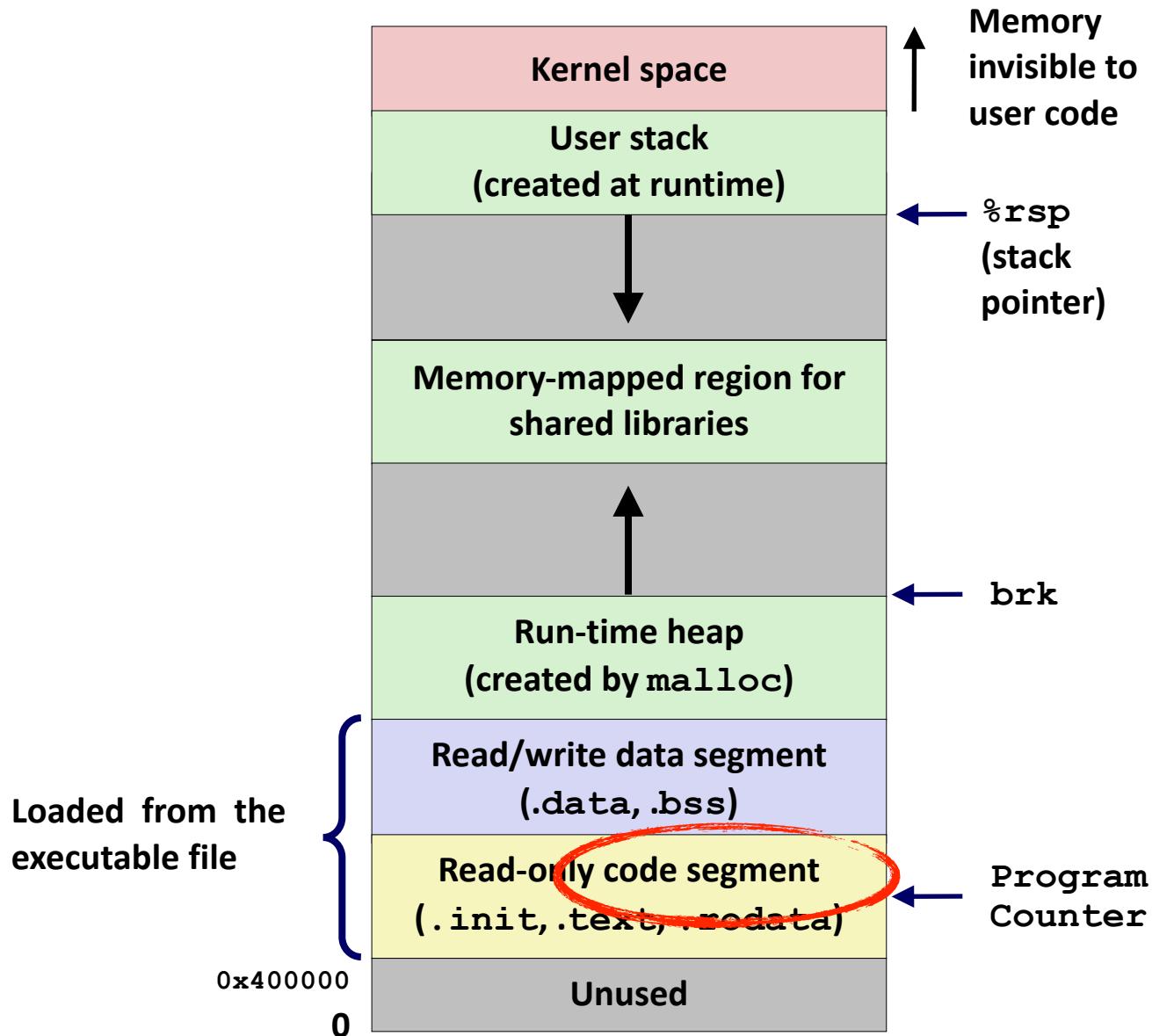
    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

fork.c

```
linux> ./fork
parent: x=0
child : x=2
```

- Call once, return twice
- Concurrent execution
 - Can't predict execution order of parent and child
- Duplicate but separate address space
 - x has a value of 1 when fork returns in parent and child
 - Subsequent changes to x are independent
- Shared open files
 - stdout is the same in both parent and child

Process Address Space

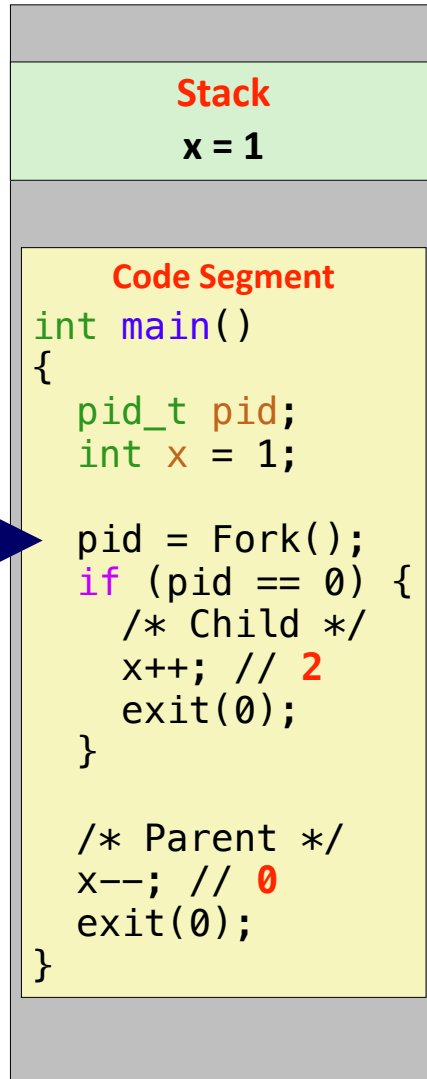


What Happens at `fork()` ?

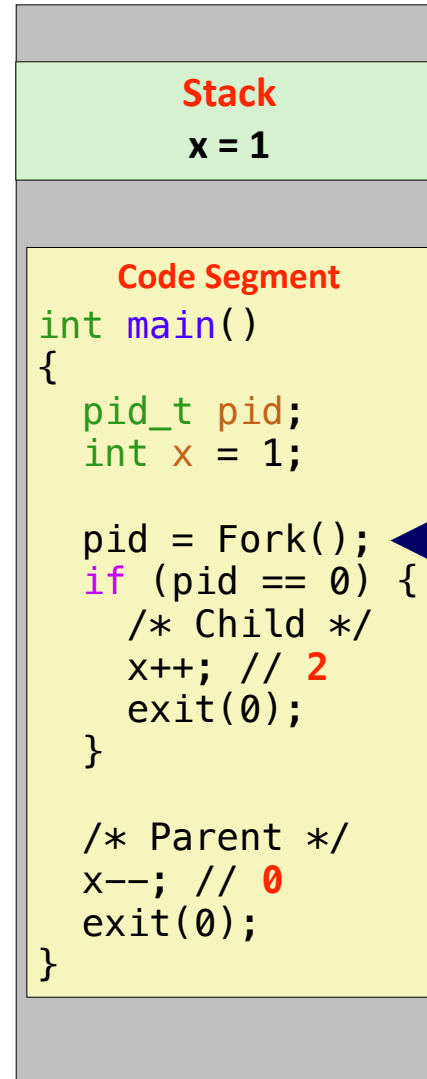
Parent Address Space

Child Address Space

Parent
Process
Program
Counter



Child
Process
Program
Counter



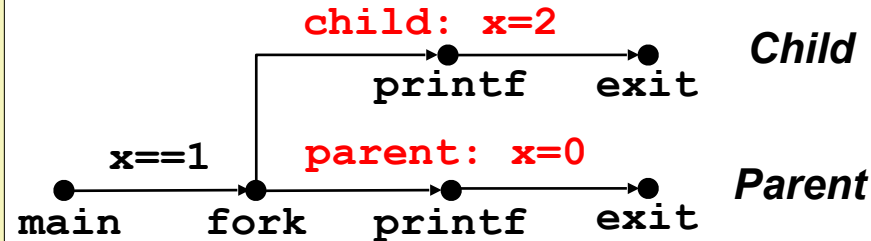
Process Graph Example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        exit(0);
    }

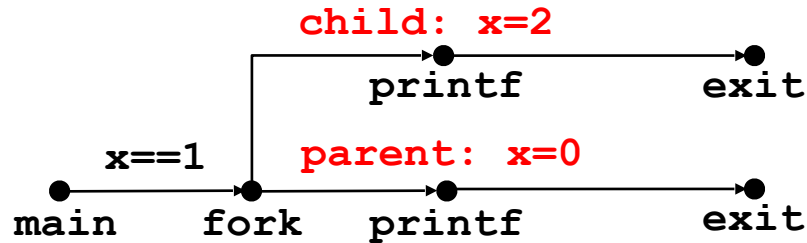
    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

fork.c

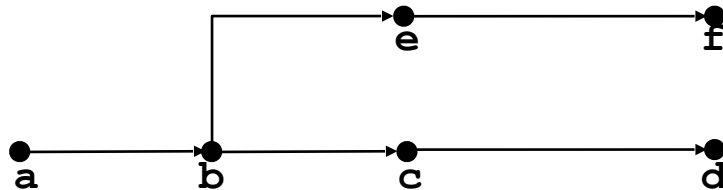


Interpreting Process Graphs

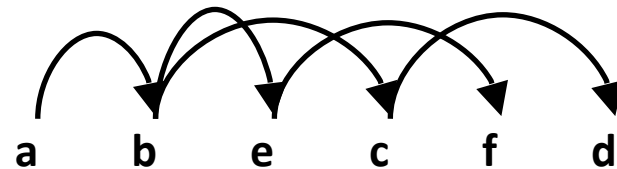
- Original graph:



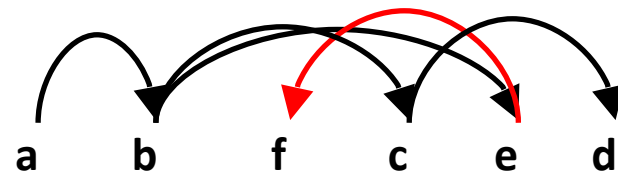
- Abstracted graph:



Feasible execution ordering:



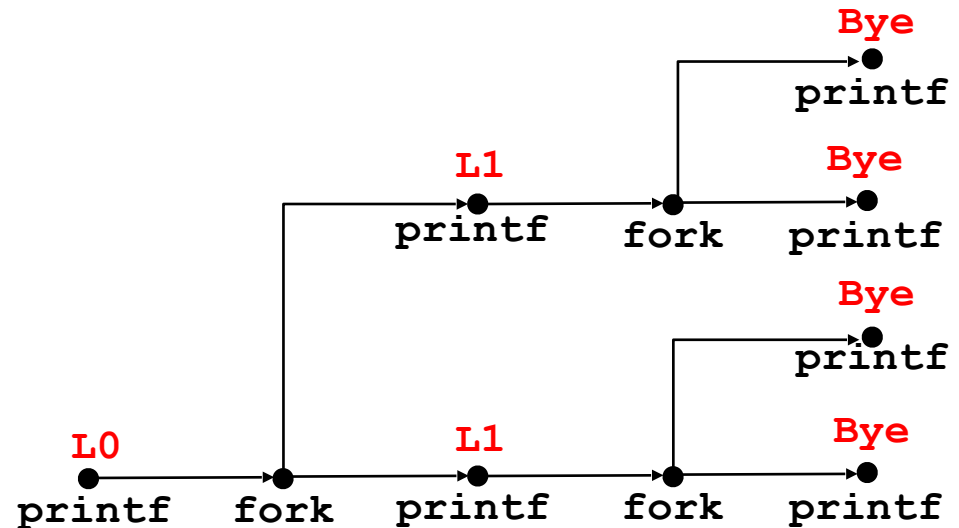
Infeasible execution ordering:



fork Example: Two consecutive forks

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

forks.c



Feasible output:

L0
L1
Bye
Bye
L1
Bye
Bye

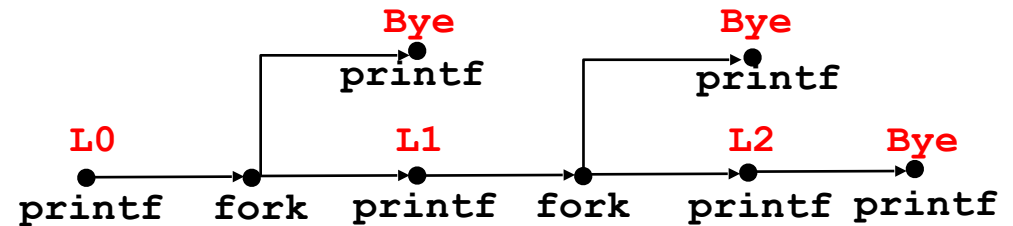
Infeasible output:

L0
Bye
L1
Bye
L1
Bye
Bye

fork Example: Nested forks in parent

```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

forks.c



Feasible output:

L0
L1
Bye
Bye
L2
Bye

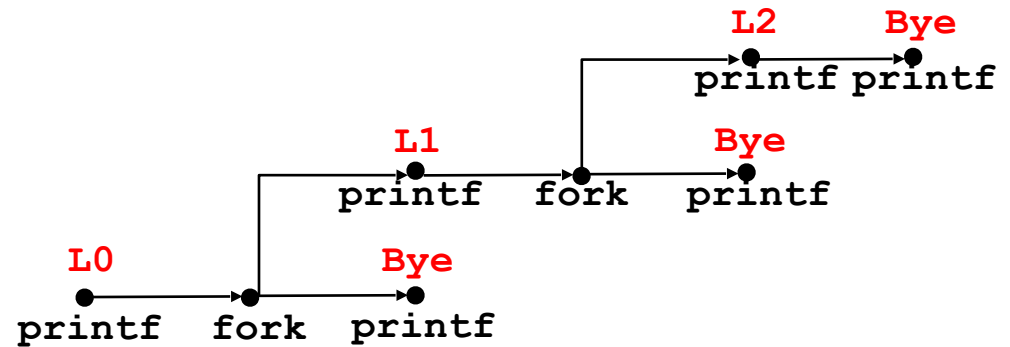
Infeasible output:

L0
Bye
L1
Bye
Bye
L2

fork Example: Nested forks in children

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

forks.c



Feasible output:

L0
Bye
L1
L2
Bye
Bye

Infeasible output:

L0
Bye
L1
Bye
Bye
L2

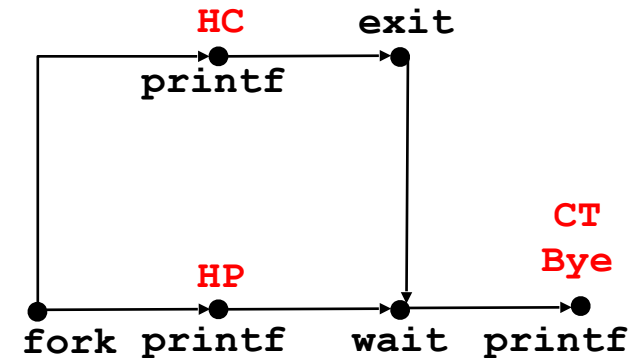
Reaping Child Processes

- When process terminates, it still consumes system resources
 - Examples: Exit status, various OS tables
 - Called a “zombie”: Living corpse, half alive and half dead
- Reaping
 - Performed by parent on terminated child (using `wait` or `waitpid`)
 - Parent is given exit status information
 - Kernel then deletes zombie child process
- What if parent doesn't reap?
 - If any parent terminates without reaping a child, then the orphaned child will be reaped by `init` process (`pid == 1`)
 - So, only need explicit reaping in long-running processes
 - e.g., shells and servers

wait: Synchronizing with Children

```
void fork9() {  
    int child_status;  
  
    if (fork() == 0) {  
        printf("HC: hello from child\n");  
        exit(0);  
    } else {  
        printf("HP: hello from parent\n");  
        wait(&child_status);  
        printf("CT: child has terminated\n");  
    }  
    printf("Bye\n");  
}
```

forks.c



Feasible output:

HC
HP
CT
Bye

Infeasible output:

HP
CT
Bye
HC

`wait`: Synchronizing with Children

- Parent reaps a child by calling the `wait` function
- `int wait(int *child_status)`
 - Suspends current process until one of its children terminates
 - Return value is the **pid** of the child process that terminated
 - If **`child_status != NULL`**, then the integer it points to will be set to a value that indicates reason the child terminated and the exit status:
 - Checked using macros defined in `wait.h`
 - `WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG`, `WIFSTOPPED`, `WSTOPSIG`, `WIFCONTINUED`
 - See textbook for details

Another `wait` Example

- If multiple children completed, will take in arbitrary order
- Can use macros `WIFEXITED` and `WEXITSTATUS` to get information about exit status

```
void fork10() {
    int i, child_status;

    for (i = 0; i < N; i++)
        if (fork() == 0) {
            exit(100+i); /* Child */
        }
    for (i = 0; i < N; i++) { /* Parent */
        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 terminate abnormally\n", wpid);
    }
}
```

forks.c

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

execve: Loading and Running Programs

Executes “/bin/ls -lt /usr/include” in child process using current environment:

```
char *myargv[] = {"/bin/ls", "-lt", "/usr/include"};
char *environ[] = {"USER=droh", "PWD="/usr/droh"};

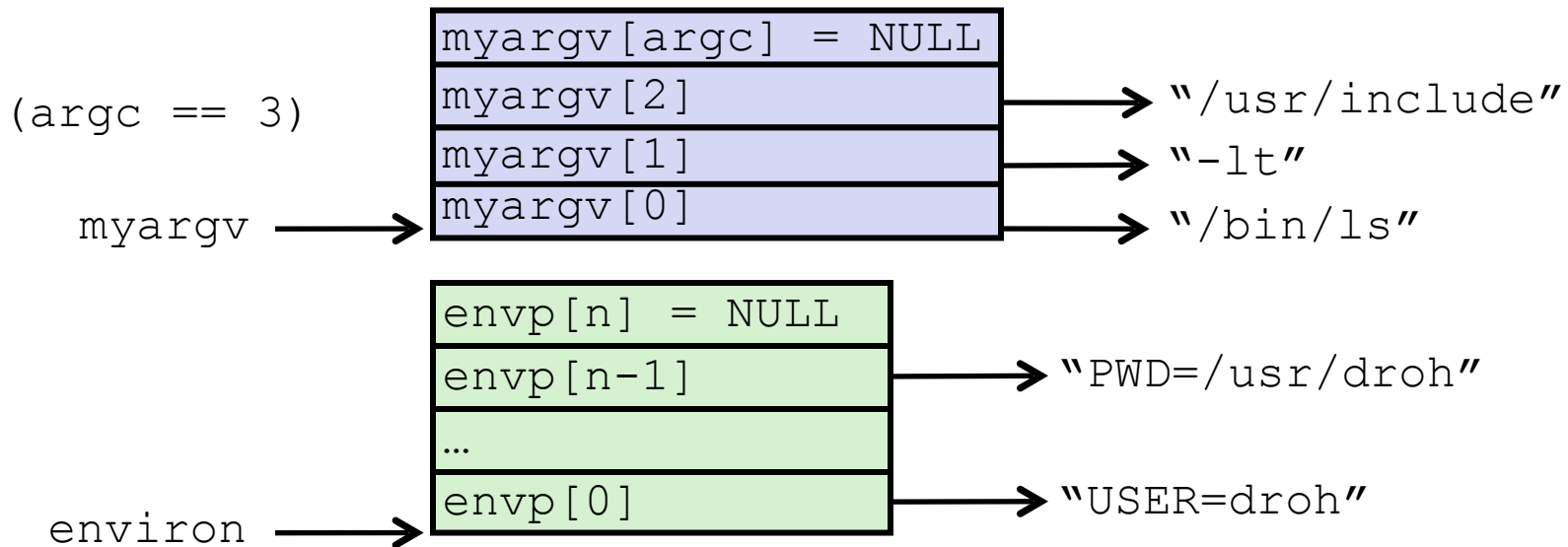
if ((pid = Fork()) == 0) {    /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```

execve: Loading and Running Programs

- `int execve(char *filename, char *argv[], char *envp[])`
- Loads and runs in the current process:
 - Executable file **filename**
 - Argument list **argv**
 - By convention **argv[0]==filename**
 - Environment variable list **envp**
 - “name=value” strings (e.g., USER=droh)
- Overwrites code, data, and stack
 - Retains PID, open files and signal context
- Called **once** and **never** returns
 - ...except if there is an error

execve Example

Executes “/bin/ls -lt /usr/include” in child process using current environment:



```
if ((pid = Fork()) == 0) {    /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```

Summary

- **Processes**

- At any given time, system has multiple active processes
- Only one can execute at a time on a single core, though
- Each process appears to have total control of processor + private memory space

- **Spawning processes**

- Call `fork`
- One call, two returns

- **Process completion**

- Call `exit`
- One call, no return

- **Reaping and waiting for processes**

- Call `wait` or `waitpid`

- **Loading and running programs**

- Call `execve` (or variant)
- One call, (normally) no return

Today

- Process Control
- Signals: The Way to Communicate with Processes

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)

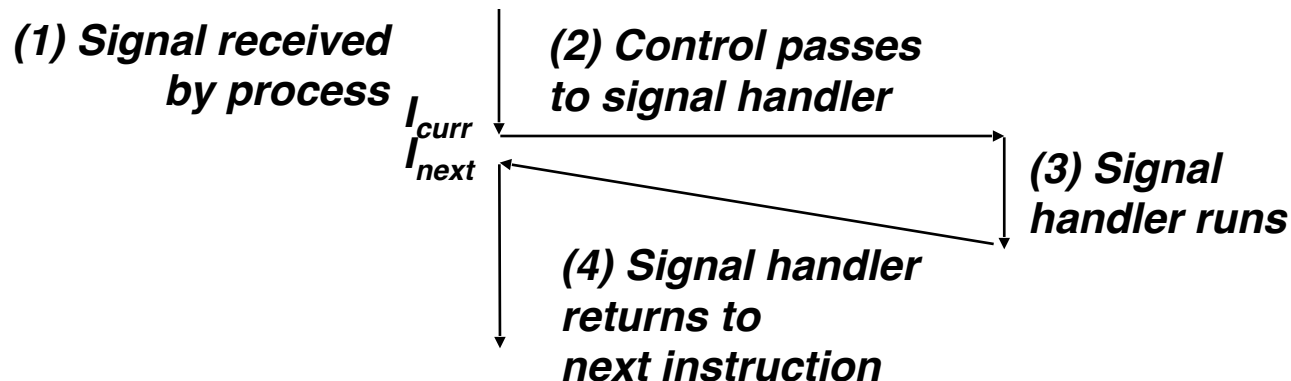
<i>ID</i>	<i>Name</i>	<i>Default Action</i>	<i>Corresponding Event</i>
2	SIGINT	Terminate	User typed ctrl-c
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

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:
 - 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:
 - **Ignore** the signal (do nothing)
 - **Terminate** the process
 - **Catch** the signal by executing a user-level function called **signal handler**



Sending Signals with `/bin/kill` Program

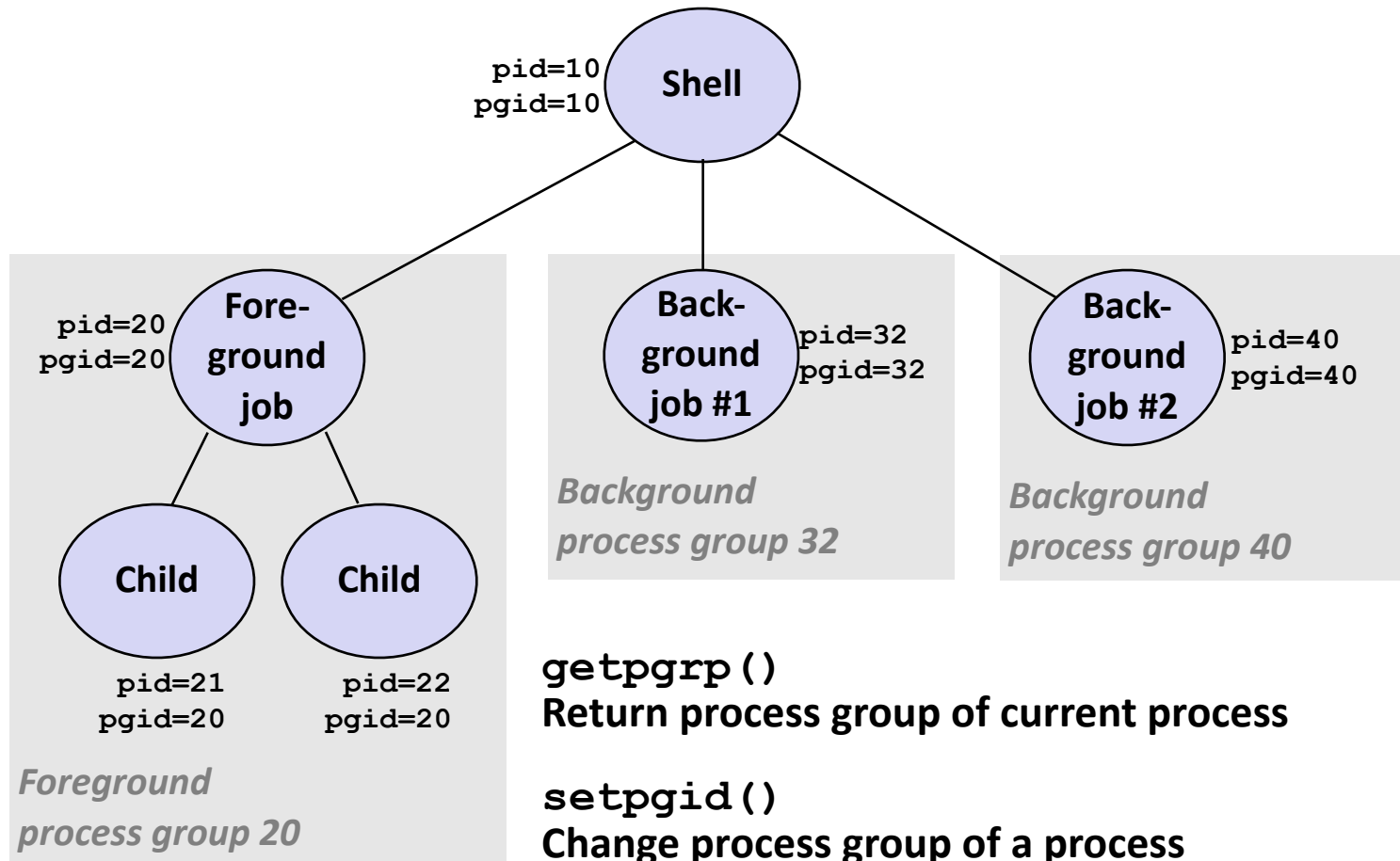
- `/bin/kill` program sends arbitrary signal to a process
- Examples
 - `/bin/kill -9 24818`
Send SIGKILL to process 24818
 - `/bin/kill` itself doesn't kill the process. 9 is the ID for the SIGKILL signal, which terminates the process

```
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



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 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

Default Actions to Signals

- 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

```
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");

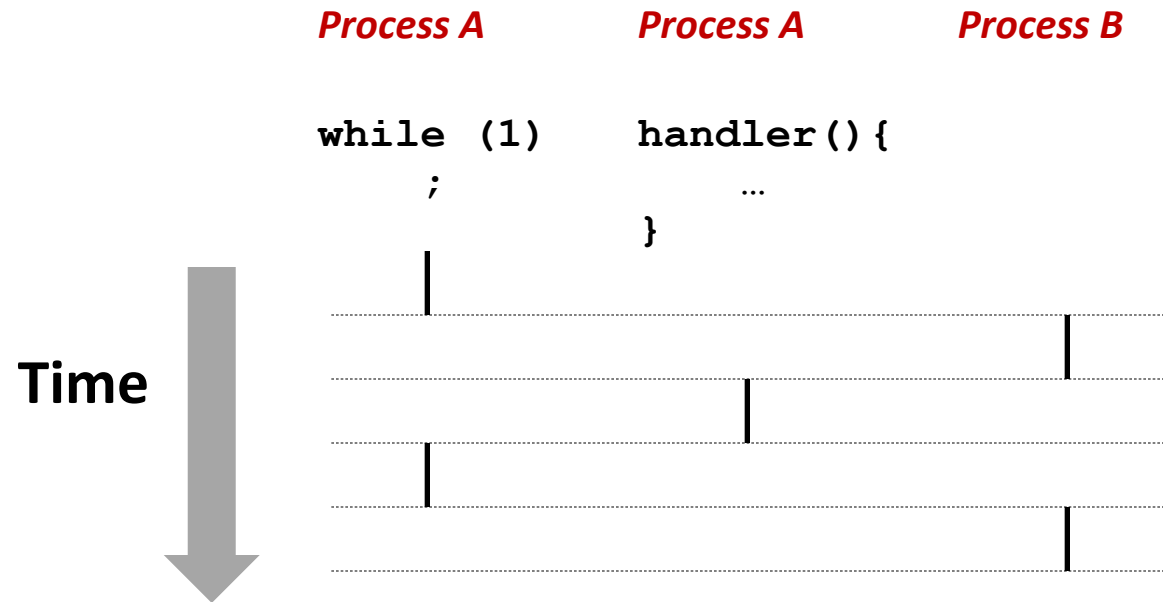
    /* Wait for the receipt of a signal */
    pause();

    return 0;
}
```

sigint.c

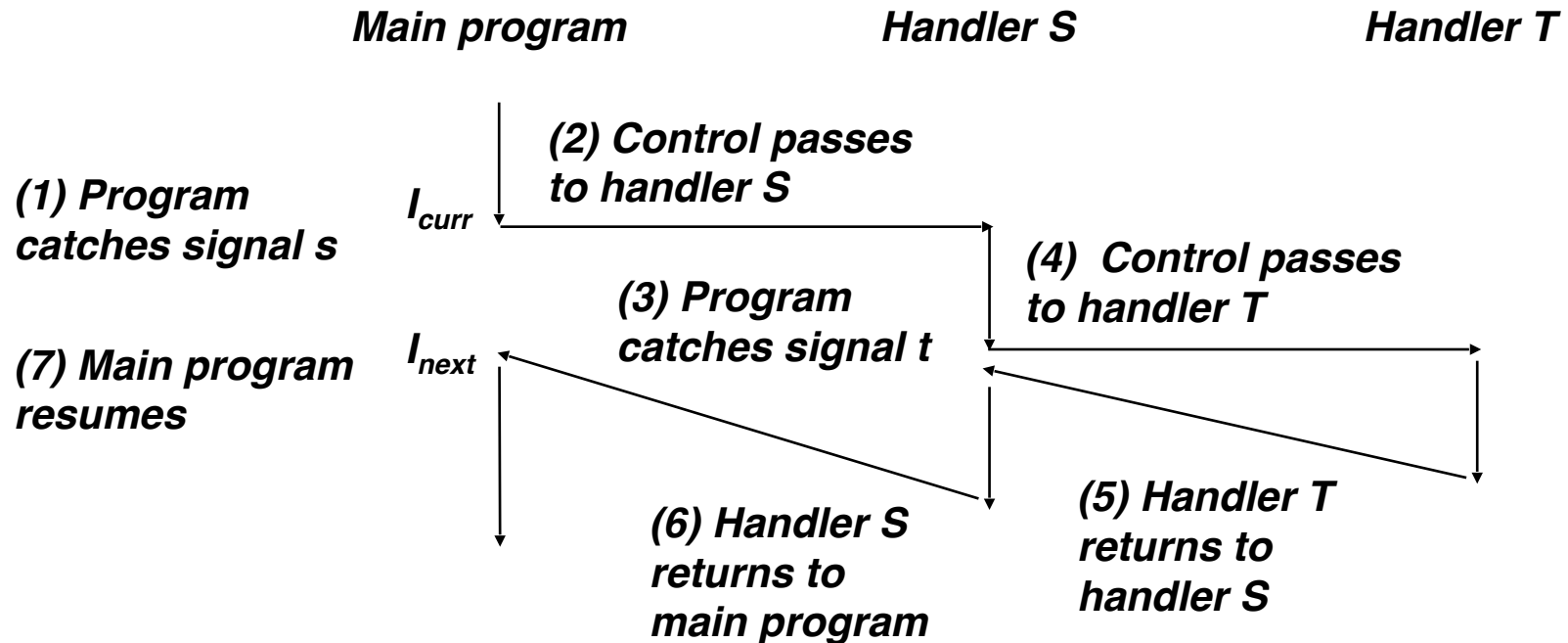
Signals Handlers as Concurrent Flows

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



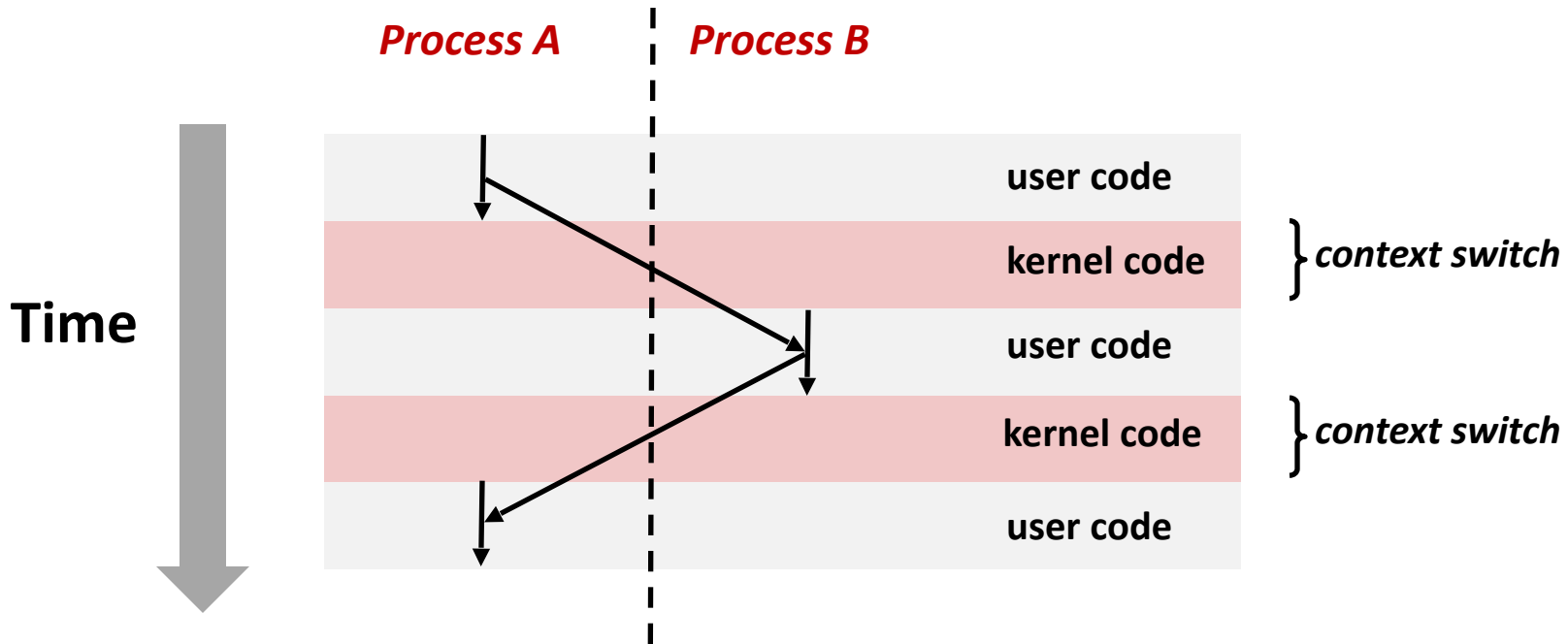
Nested Signal Handlers

- Handlers can be interrupted by other handlers



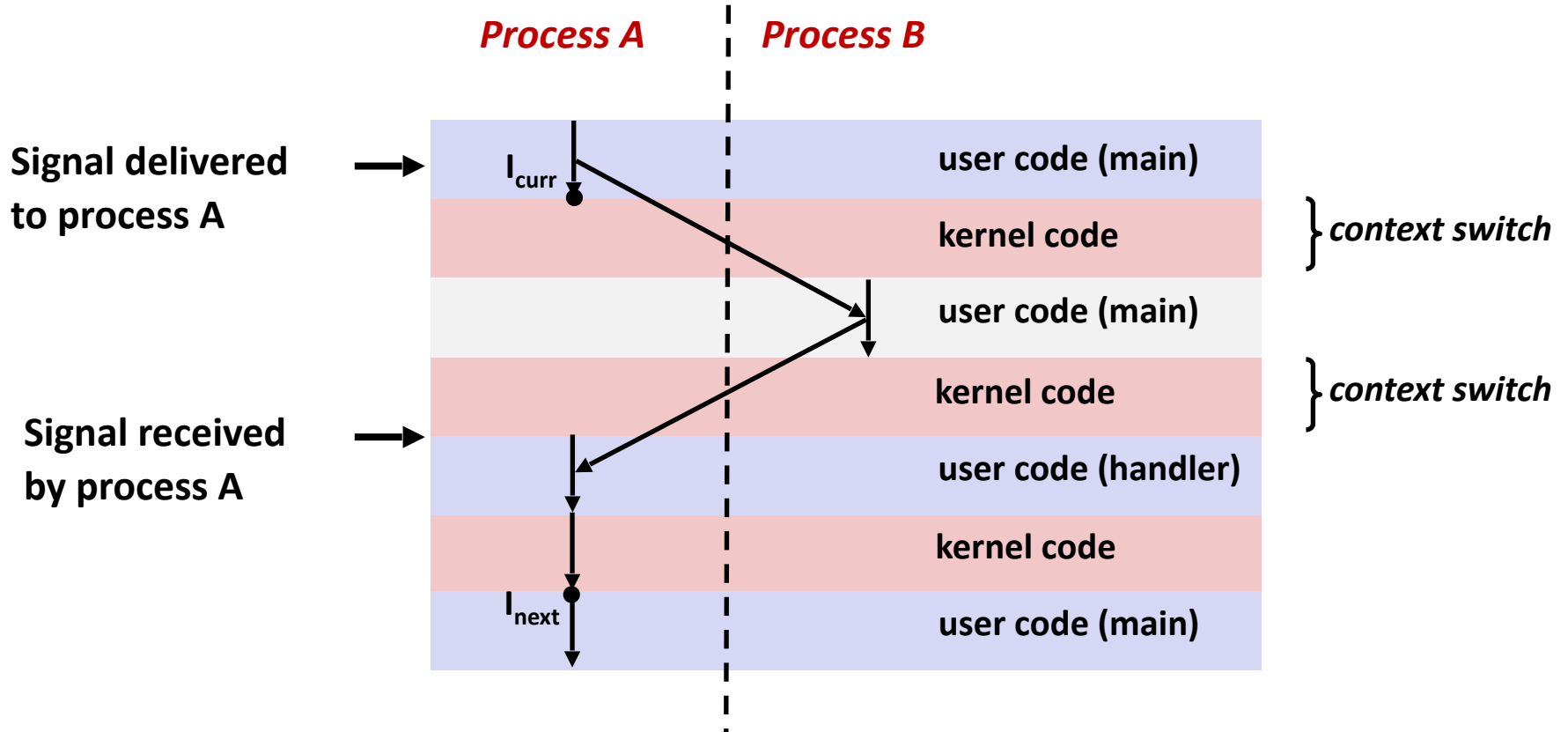
Receiving/Responding to Signals

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



Receiving/Responding to Signals

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



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 for a process
 - 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/mask** the receipt of certain signals
 - Blocked signals can be delivered, i.e., in the pending state, but will not be received/responded to *until the signal is unblocked*

Pending/Blocked Bits

- Kernel maintains pending and masked 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
 - masked: 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

- Right before kernel is ready to pass control to process p
- Kernel computes the set of pending & nonmasked signals for process p (PNM set)
- If (PNM is empty), i.e., no signal is pending & nonmasked
 - No signals to respond to; simply pass control to next instruction in the logical flow for p
- Else
 - Choose least nonzero bit k in pnm and force process p to **receive** signal k, i.e., by executing the corresponding signal handler
 - Repeat for all nonzero k in pnm
 - Pass control to next instruction in logical flow for p

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, 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 it either has no access to globally shared variables (a.k.a., reentrant) or is 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