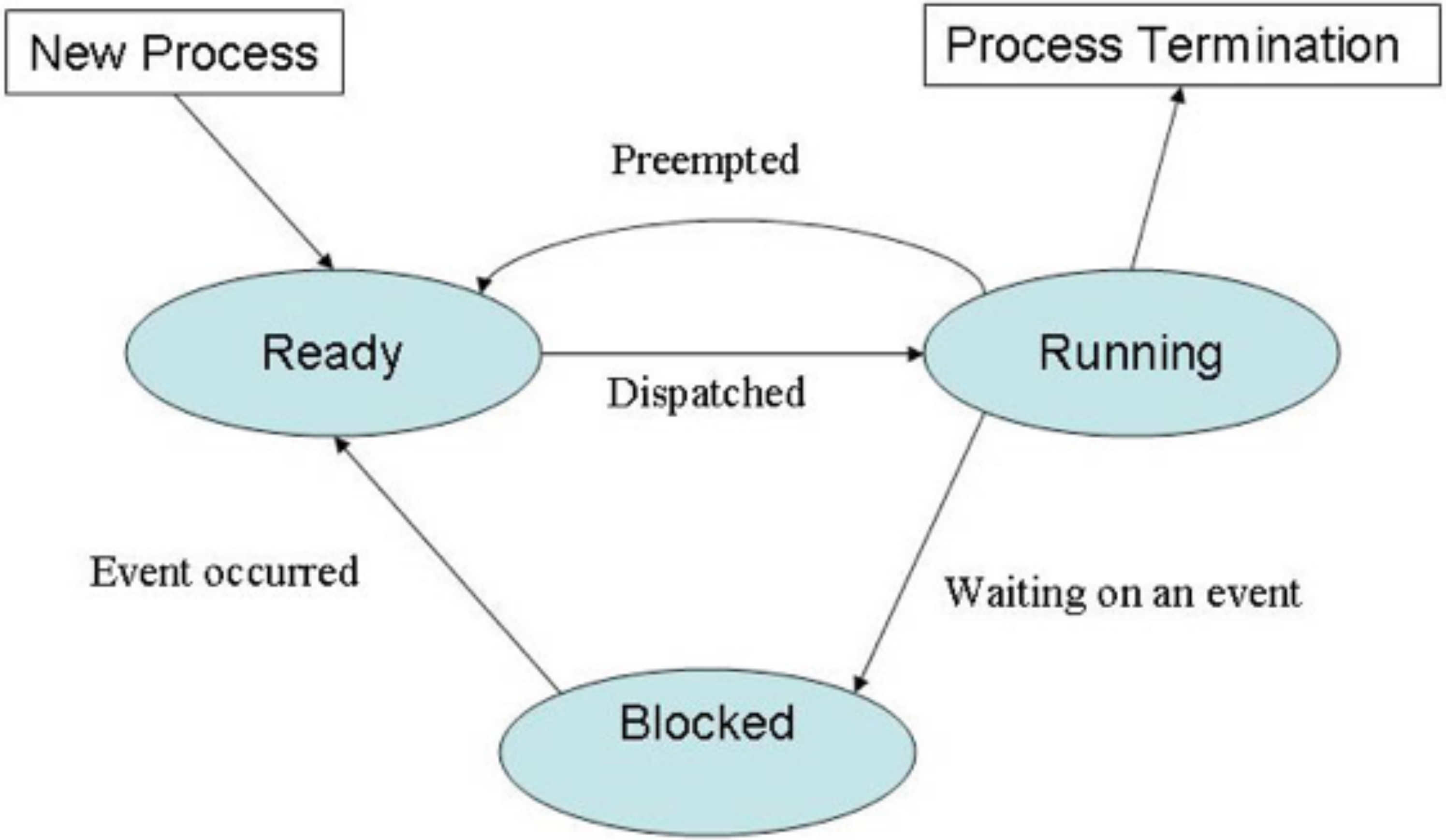
Process Control *con't*

+ Review: Process States



+ Review: `fork()`

- **Parent process creates a new running child process by calling fork**

- **`int fork(void)`**
  - Returns 0 to the child process, child's PID to parent process
  - Child is *almost* identical to parent:
    - Child gets identical copy of the parent's virtual address space.
    - Child gets identical copies of the parent's open file descriptors
    - Child gets a different PID than the parent

+ Review: Process Graphs

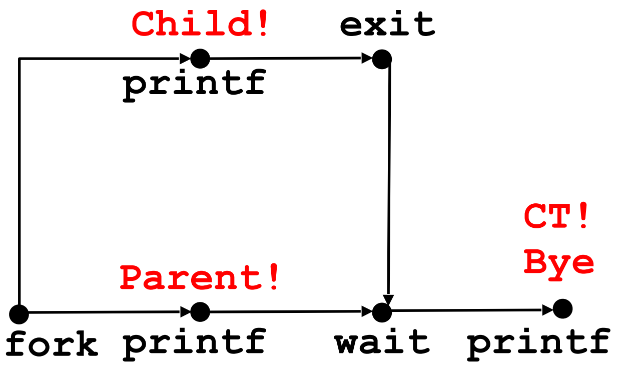
- **A *process graph* visualizes the partial ordering of statements in a concurrent program:**
  - Each vertex is the execution of a statement
  - a → b means a happens *before* b
  - Edges can be labeled with current value of variables
  - `printf` vertices can be labeled with output
- **Any sort of the graph corresponds to a feasible total ordering.**

+ `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.

+ wait(): Synchronizing with Children

```
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf(“Child!”);
       exit(0);
    } else {
        printf(“Parent!);
        wait(&child_status);
        printf(“CT!”);
    }
    printf("Bye\n");
}
```



Feasible output:
Parent!
Child!
CT
Bye

Infeasible output:
Parent!
CT
Bye
Child!

Another wait() Example

- If multiple children completed, will take in arbitrary order
- Use WIFEXITED and WEXITSTATUS to get exit status

```c
void fork10() {
    int pid[N];
    int i, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */

    for (i = 0; i < N; i++) { /* Parent */
        int child_pid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   child_pid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", child_pid);
    }
}
```

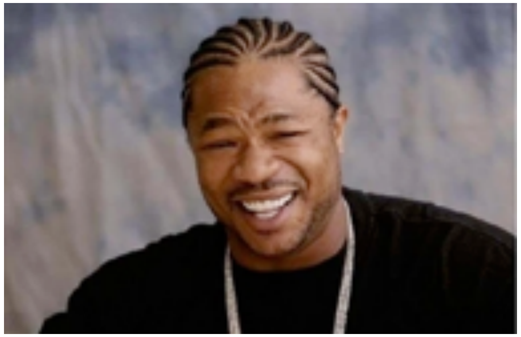
waitpid(): Waiting for a Specific Process

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

```
void fork11() {
    int pid[N];
    int i, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */

    for (i = N-1; i >= 0; i--) {
        int child_pid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   child_pid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", child_pid);
    }
}
```

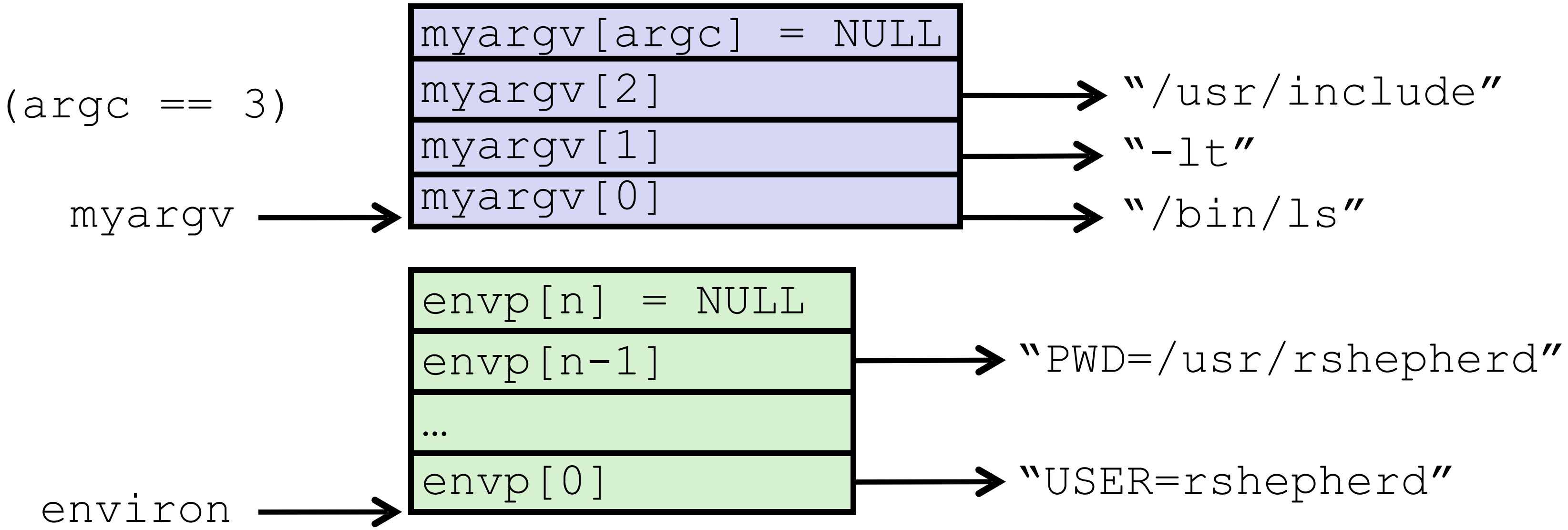
+ execve(): Loading and Running Programs

- `int execve(char* filename, char* argv[], char* envp[])`
- **Loads and runs in the current process:**
  - Executable file `filename`
  - …with argument list `argv`
  - …and environment variable list `envp`
    - "name=value" strings (e.g., `USER=rshepherd`)
    - `getenv(), putenv(), printenv()`
- **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:**



(argc == 3)

myargv

myargv[argc] = NULL

myargv[2]

myargv[1]

myargv[0]

"/usr/include"

"-lt"

"/bin/ls"

environ

envp[n] = NULL

envp[n-1]

…

envp[0]

"PWD=/usr/rshepherd"

"USER=rshepherd"

```
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

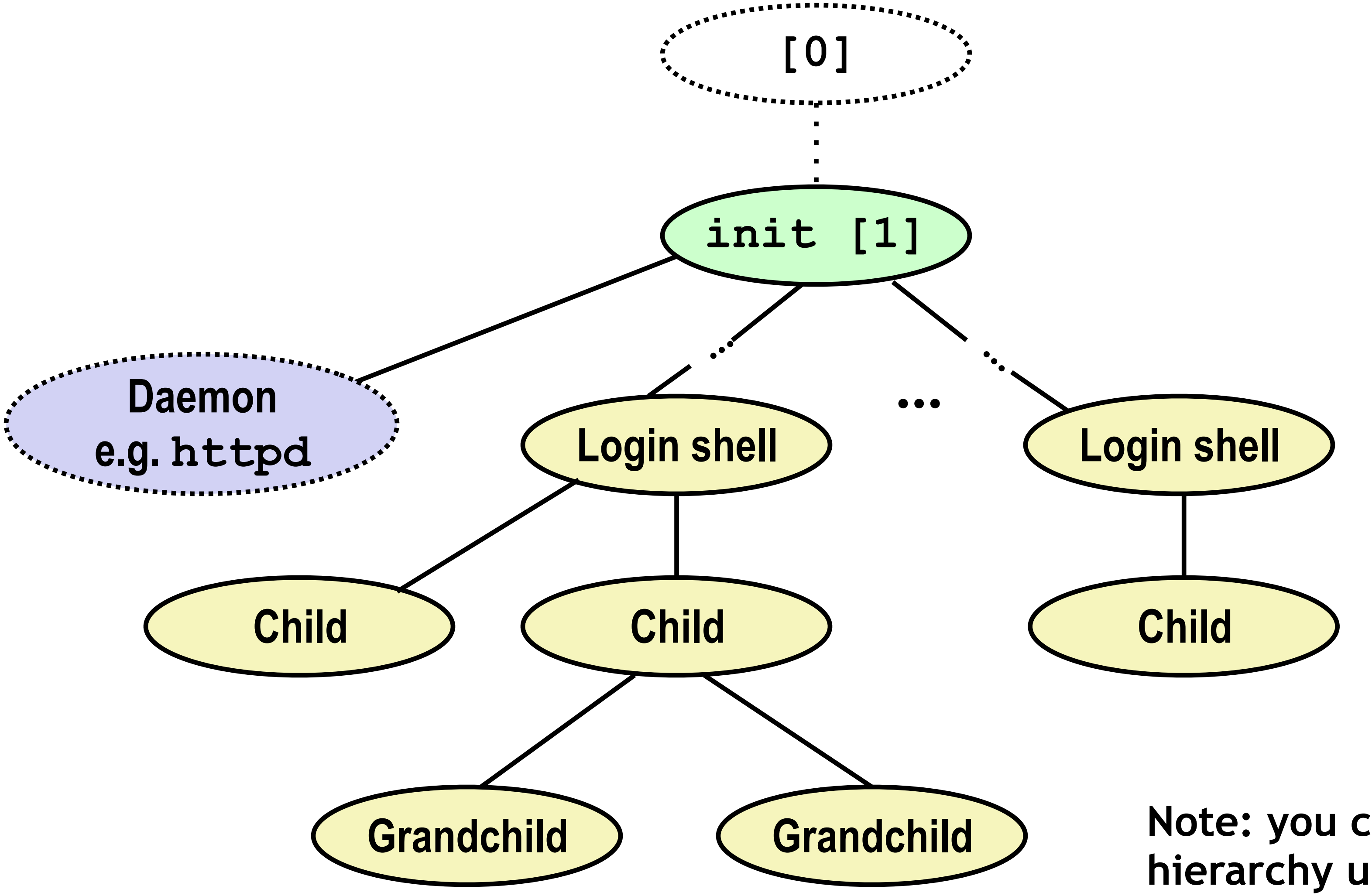
- **Processes**
  - At any given time, system has multiple active processes
  - Only one can execute at a time on a single core, though
  - Each process appears to have total control of processor + private memory space

+ Summary *con't*

- **Creating 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

Signals

+ Linux Process Hierarchy



Note: you can view the hierarchy using the Linux `pstree` 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)
  - **csh/tcsh** 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);
    }
}
```

Execution is a sequence of read/evaluate steps

+ Simple Shell eval Function

```
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list for execve() */
    int bg;              /* Should the job run in bg or fg? */
    int pid;             /* Process id */

    bg = parseline(cmdline, argv); /* Extract arguments and set bg */

    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;
}
```

+ 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 (probably) will not terminate
  - Will create a memory leak that could run the kernel out of memory

+ Solution: Exceptional control flow

- **We can leverage exceptional control flow from our programs**
  - The kernel will interrupt regular processing to alert us when a background process completes
  - In Unix, the mechanism is called a **signal**

+ 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)
  - Signal type is identified by integer ID’s (1-30)
  - Only information in a signal is its ID and the fact that it arrived

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

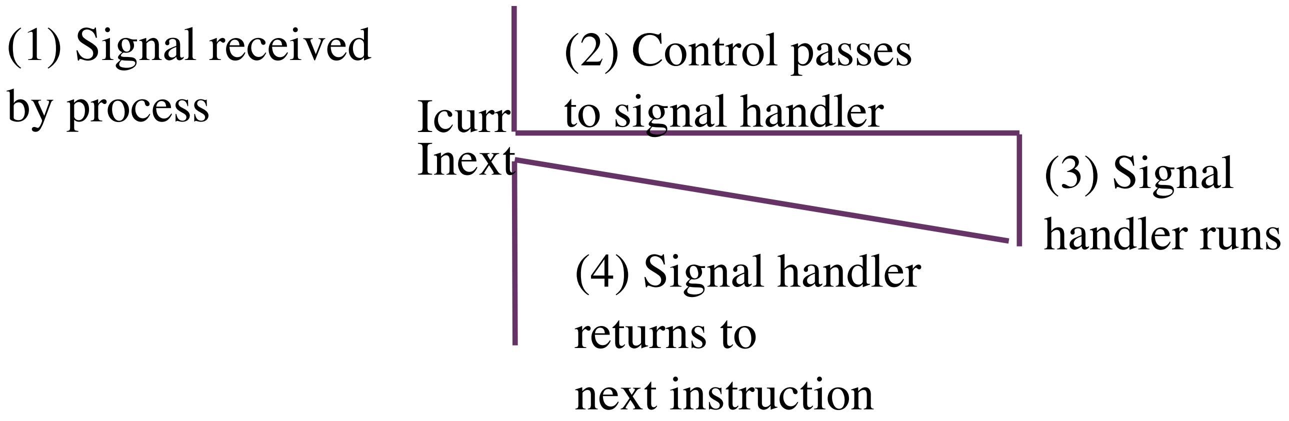
+ Signal Concepts: Sending a Signal

- **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 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: 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


(1) Signal received by process
Icurr
Inext
(2) Control passes to signal handler
(3) Signal handler runs
(4) Signal handler returns to next instruction

+ Signal Concepts: Pending & Blocked

- **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

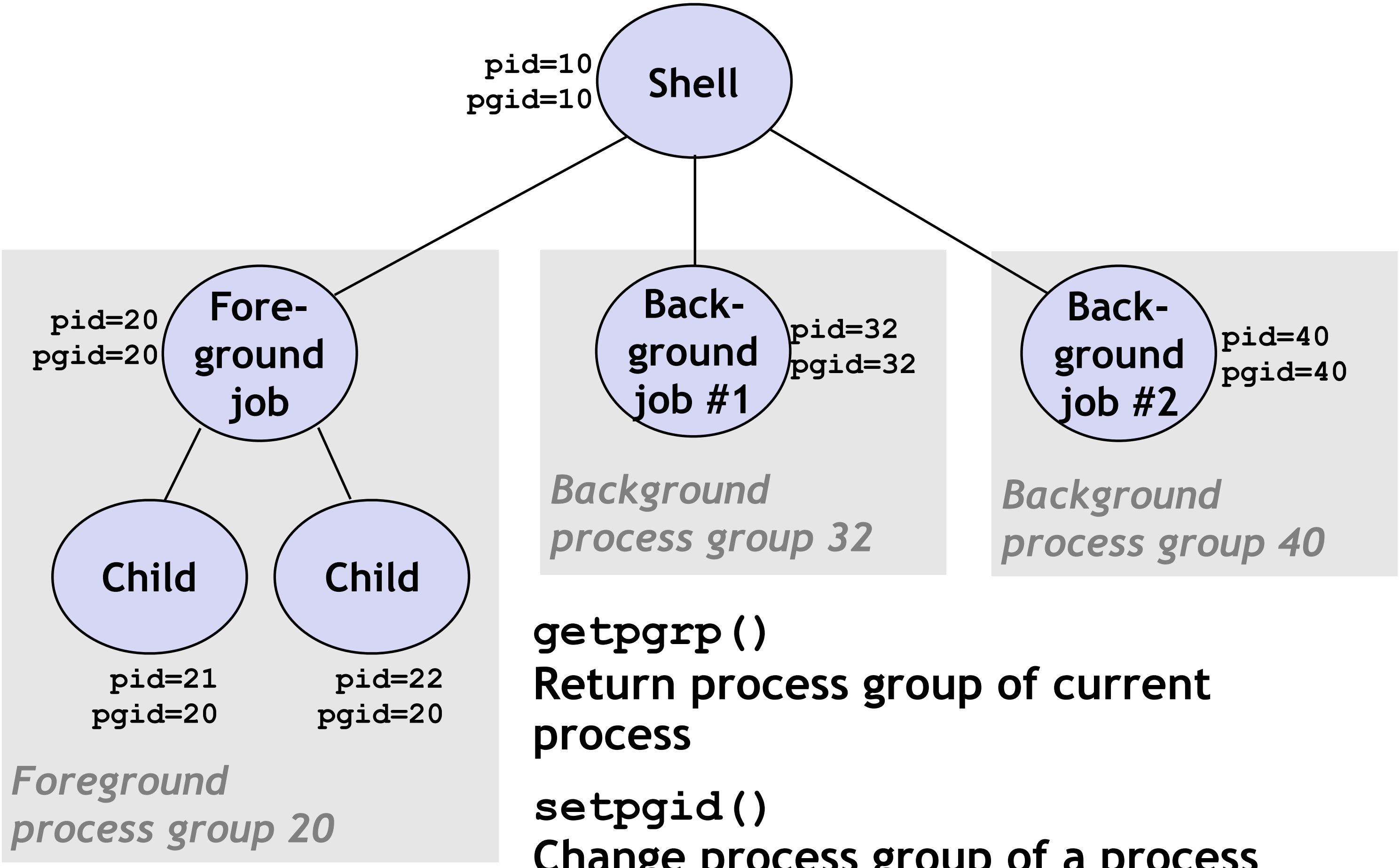
- **A *pending* signal is received at most once**

+ 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



`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 specified 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

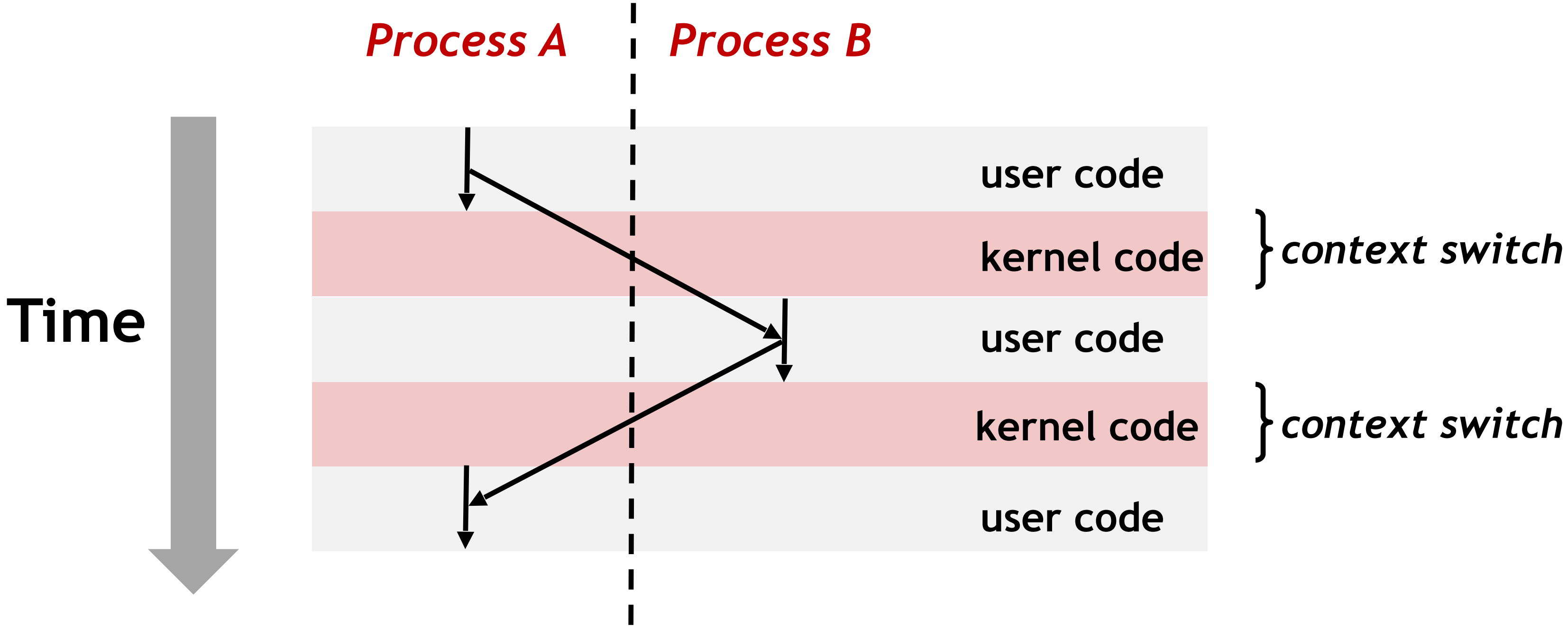
```
linux> ./forks 16
Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817

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 with kill Function

```c
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);
    }
}
```

+ Receiving Signals

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



+ Receiving Signals

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

- **Kernel computes pnb = pending & ~blocked**
  - The set of pending nonblocked signals for process B

- **If (pnb == 0)**
  - Pass control to next instruction in the logical flow for B

- **Else**
  - Choose nonzero bit k in pnb and force process B to receive signal k
  - The receipt of the signal triggers some action by B
  - Repeat for all nonzero bits in pnb
  - Pass control to next instruction in logical flow for B

+ 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
- **What if we do not like the default action?**
  - Signal handlers

+ 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 for signals of type `signum`
  - Otherwise, handler is the address of a user-level *signal handler*
    - Called when process receives signal of type `signum`
    - 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.

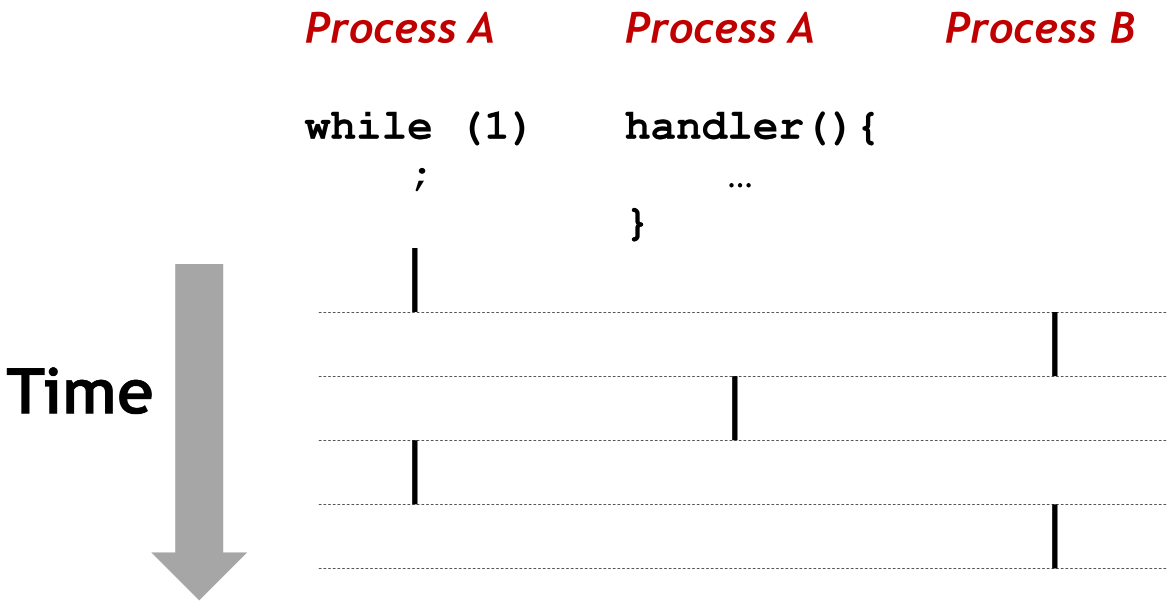
- **Returns the previous value of the signal handler, or SIG_ERR on error.**

+ Signal Handling Example

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

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**



Process A

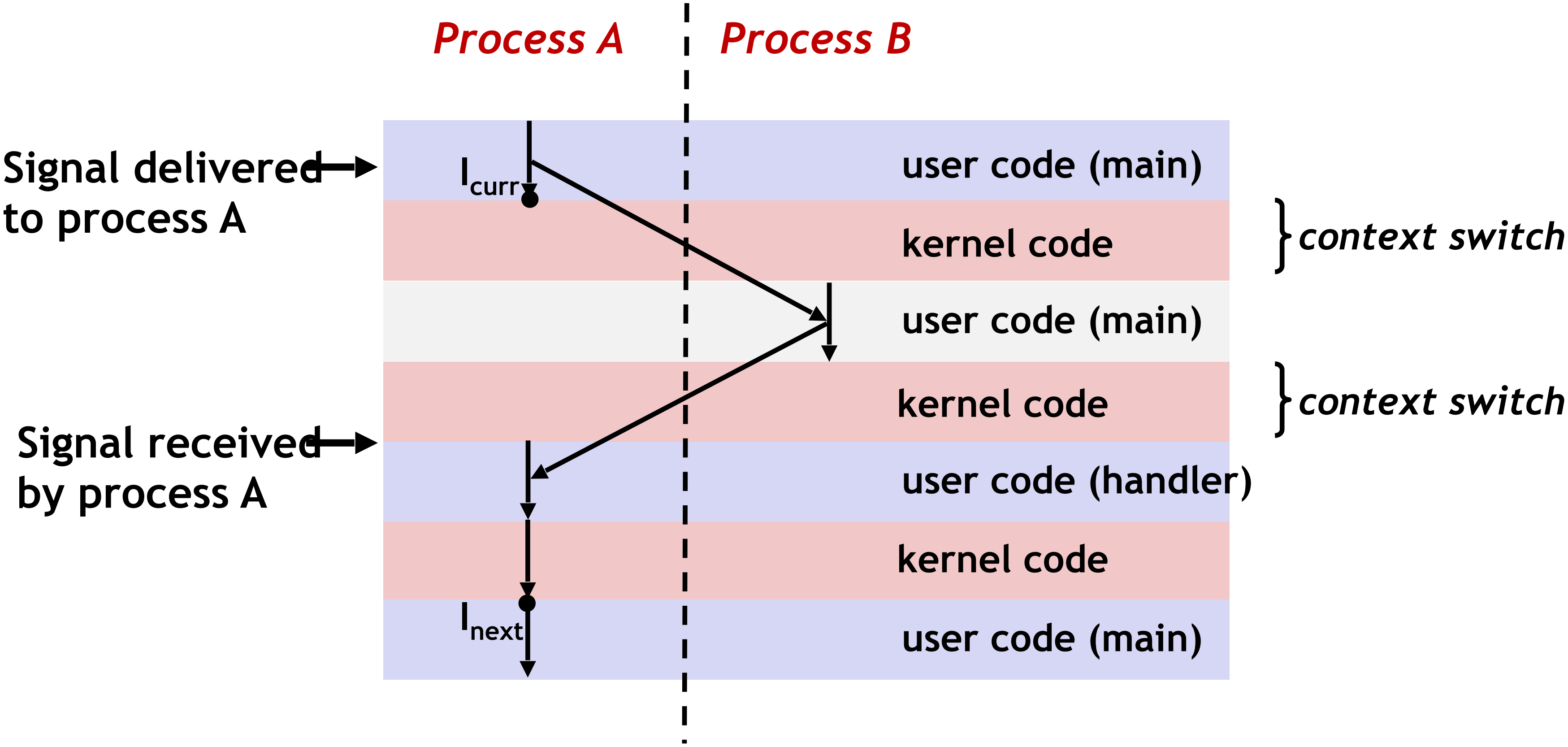
```
while (1)
    ;
```

Process A

```
handler(){
    …
}
```

Process B

+ Signal Handlers as Concurrent Flows *con't*



+ Concurrency Issue with Signal Handlers

```
int N = 999;

void sigint_handler(int sig) /* SIGINT handler */ {
    x = rand() % 2 == 0 ? x : x+=1 ;
}

int main() {
     int array[N];
     if (signal(SIGINT, sigint_handler) == SIG_ERR)
        unix_error("signal error");

     pause();

     printf(“%d”, array[N]);

    return 0;
}
```

+ Summary

- **Signals provide process-level exception handling**
  - Can generate from user programs
  - Can define effect by declaring signal handler
- **Handlers are tricky because they are concurrent with main program and share the same global data structures.**
  - Shared data structures can become corrupted.
- **We’ll explore concurrency issues later in the term.**