Exceptional Control Flow: Signals and Nonlocal Jumps

(Chapter 8)

ECF Exists at All Levels of a System

- **Exceptions**
  - Hardware and operating system kernel software
- **Process Context Switch**
  - Hardware timer and kernel software
- **Signals**
  - Kernel software
- **Nonlocal jumps**
  - Application code

Previous Slides (Exceptions, Process Context Switch)

These Slides (Signals, Nonlocal jumps)

The World of Multitasking

- **System runs many processes concurrently**

- **Process: executing program**
  - State includes memory image + register values + program counter

- **Regularly switches from one process to another**
  - Suspend process when it needs I/O resource or timer event occurs
  - Resume process when I/O available or given scheduling priority

- **Appears to user(s) as if all processes executing simultaneously**
  - Even though most systems can only execute one process at a time
  - Except possibly with lower performance than if running alone

Programmer's Model of Multitasking

Basic functions

`fork` spawns new process

Called once, returns twice

`exit` terminates own process

Called once, never returns

Puts it into "zombie" status

`wait` and `waitpid` wait for and reap terminated children

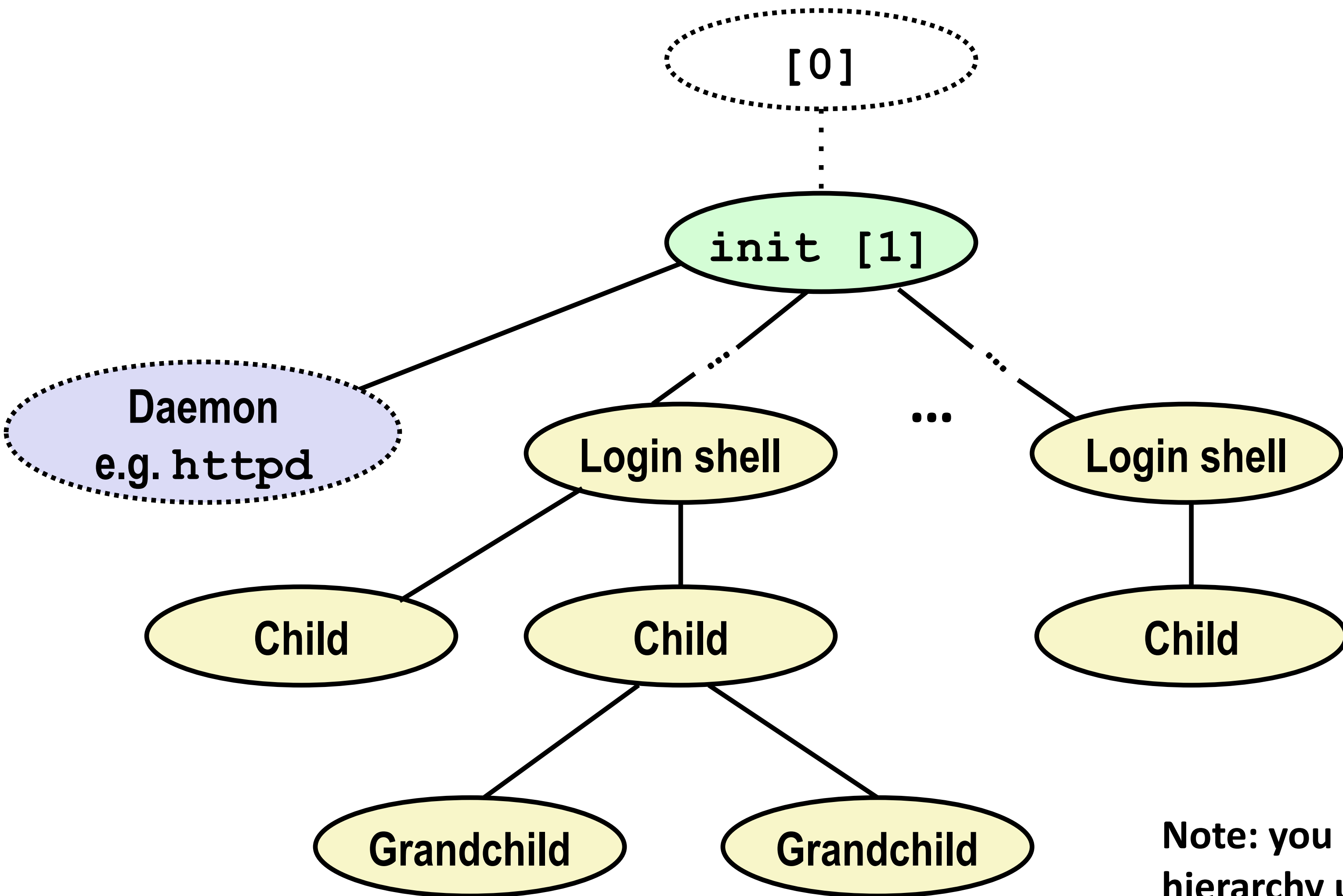
`execve` runs new program in existing process

Called once, (normally) never returns

Programming challenge

- Understanding the nonstandard semantics of the functions
- Avoiding improper use of system resources
  - e.g. "Fork bombs" can disable a system

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 execve() */
    char buf[MAXLINE];   /* Holds modified command line */
    int bg;              /* Should the job run in bg or fg? */
    pid_t pid;           /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    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;
}
```

What Is a “Background Job”?

Users generally run one command at a time

- Type command, read output, type another command

Some programs run “for a long time”

- Example: “delete this file in two hours”

```
unix> sleep 7200; rm /tmp/junk    ← shell stuck for 2 hours
```

A “background” job is a process we don't want to wait for

```
unix> (sleep 7200 ; rm /tmp/junk) &
[1] 907
unix>    ← ready for next command
```

Problem with Simple Shell Example

Our example shell correctly waits for and reaps foreground jobs.

What about 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
- Modern Unix: once you exceed your process quota, your shell can't run any new commands for you: fork() returns -1

```
unix> limit maxproc          ← csh syntax
maxproc      31818
unix> ulimit -u              ← bash syntax
31818
```

Exceptional Control Flow to the Rescue!

Problem: The shell doesn't know when a background job will finish

- It could happen at any time
- Regular control flow: “Wait until running job completes, then reap it”
- Can't reap exited background processes in a timely fashion

Solution: Use a Signal

- The kernel will interrupt the shell to alert it when a background process completes

Signals

Terminology

- SIGKILL, SIGINT, SIGSEGV, SIGALRM, SIGFPE, SIGCHLD, …
- Sending signals
- Receiving signals
- Signal handler
- Pending, Blocked
- /bin/kill
- Process groups
- Installing handlers, catching signals

Signals

A *signal* is a message that notifies a process that an event of some type has occurred in the system

- Similar to exceptions and interrupts
- Sent from the kernel (sometimes at the request of another process) to a process
- Signal type is identified by a small integer ID (1-30)
- The only information is its ID (and the fact that it occurred)

ID	*Name*	*Default Action*	*Corresponding Event*
2	SIGINT	Terminate	Interrupt (e.g., ctl-c from keyboard)
9	SIGKILL	Terminate	Kill program [cannot override or ignore]
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated
…	…	…	…

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

  Examples:

  a divide-by-zero happened (**SIGFPE**)

  a child process terminated (**SIGCHLD**)

- Another process has invoked the **`kill()`** system call

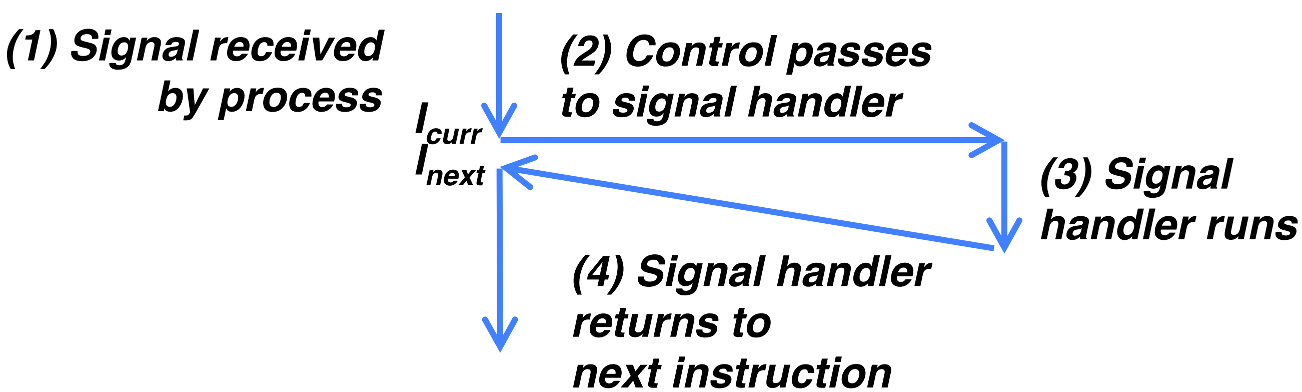
> **`kill(pid,sig)`**
>
> causes the kernel to send a signal to a process

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

What happens when the signal is received?

- ***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:



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 pending signal is *received* at most once

- A received signal will be acted upon (handled, etc.)

A process can *block* the receipt of certain signals

- The signal remains pending.
- It is not received.
- The signal is received when it is finally unblocked.

Pending/Blocked Bits

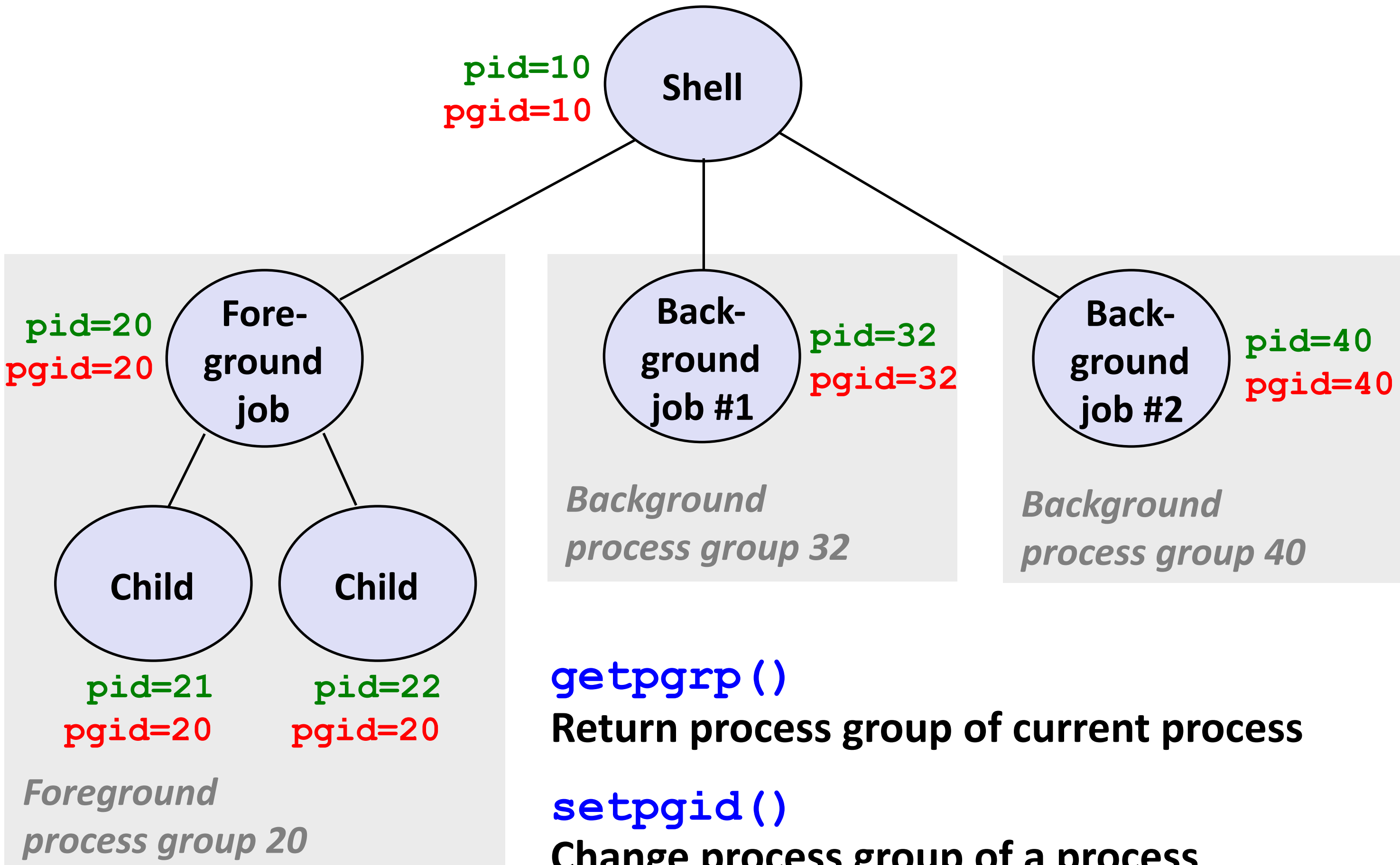
Kernel maintains `pending` and `blocked` bit vectors in the context of each process

- **`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*.

Process Groups

Every process belongs to exactly one process group



`getpgrp()`
Return process group of current process

`setpgid()`
Change process group of a process

The /bin/kill command

Send a signal to a process

(Can send any signal to a process or process group)

Example: Send SIGINT to a process

`/bin/kill -2 15887`

2 = SIGINT
9 = SIGKILL
etc…

```
linux> ./fork16
Parent: pid=15885  proc-group=26859
Child: pid=15887  proc-group=15885
Child: pid=15886  proc-group=15885
Child: pid=15888  proc-group=15885
linux> ps
  PID TTY          TIME CMD
15886 pts/13   00:00:01 fork
15887 pts/13   00:00:01 fork
15888 pts/13   00:00:01 fork
15927 pts/13   00:00:00 ps
26859 pts/13   00:00:00 csh
linux> /bin/kill -2 15887
linux> ps
  PID TTY          TIME CMD
15886 pts/13   00:00:17 fork
15888 pts/13   00:00:17 fork
16101 pts/13   00:00:00 ps
26859 pts/13   00:00:00 csh
linux>
```

The /bin/kill command

Send a signal to a process

(Can send any signal to a process or process group)

Example: Send SIGINT to a group

```
/bin/kill -2 -19691
```

2 = SIGINT
9 = SIGKILL
etc…

Sends it to all processes in the group

```
linux> ./fork16
Parent: pid=19691  proc-group=26859
Child: pid=19692  proc-group=19691
Child: pid=19693  proc-group=19691
Child: pid=19694  proc-group=19691
linux> ps
  PID TTY          TIME CMD
19692 pts/13   00:00:03 fork
19693 pts/13   00:00:03 fork
19694 pts/13   00:00:03 fork
19730 pts/13   00:00:00 ps
26859 pts/13   00:00:00 csh
linux> /bin/kill -2 -19691
linux> ps
  PID TTY          TIME CMD
20058 pts/13   00:00:00 ps
26859 pts/13   00:00:00 csh
linux>
```

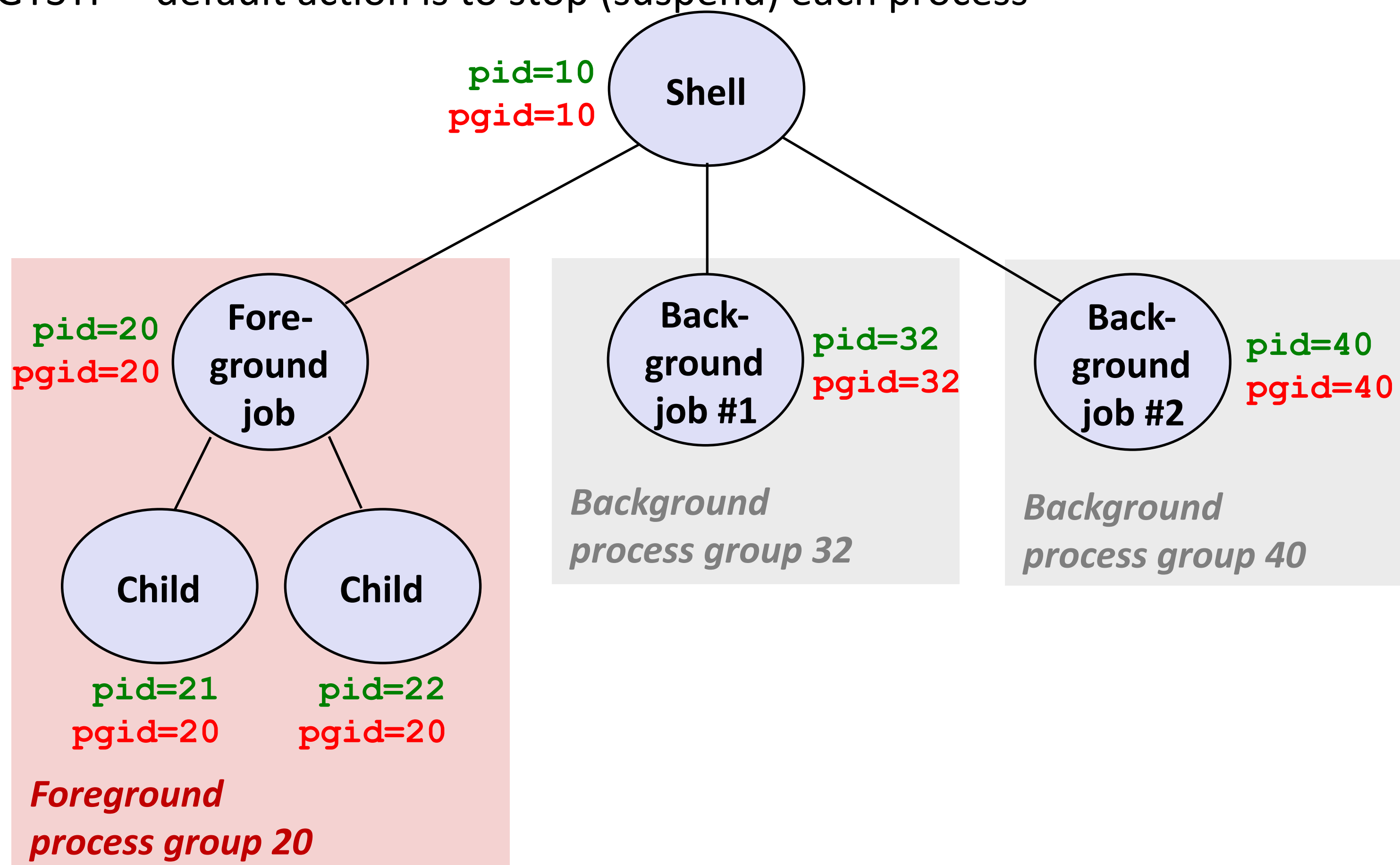
Sending Signals from the Keyboard

Typing ctrl-c sends a SIGINT to every job in the foreground process group.

SIGINT – default action is to terminate each process

Typing ctrl-z sends 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`

```
linux> ./fork17
Child: pid=28108 pgrp=28107
Parent: pid=28107 pgrp=28107
<types ctrl-z>
Suspended
linux> ps w
  PID TTY      STAT   TIME COMMAND
27699 pts/8    Ss     0:00 -tcsh
28107 pts/8    T      0:01 ./fork17
28108 pts/8    T      0:01 ./fork17
28109 pts/8    R+     0:00 ps w
bluefish> fg
./forks17
<types ctrl-c>
linux> 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

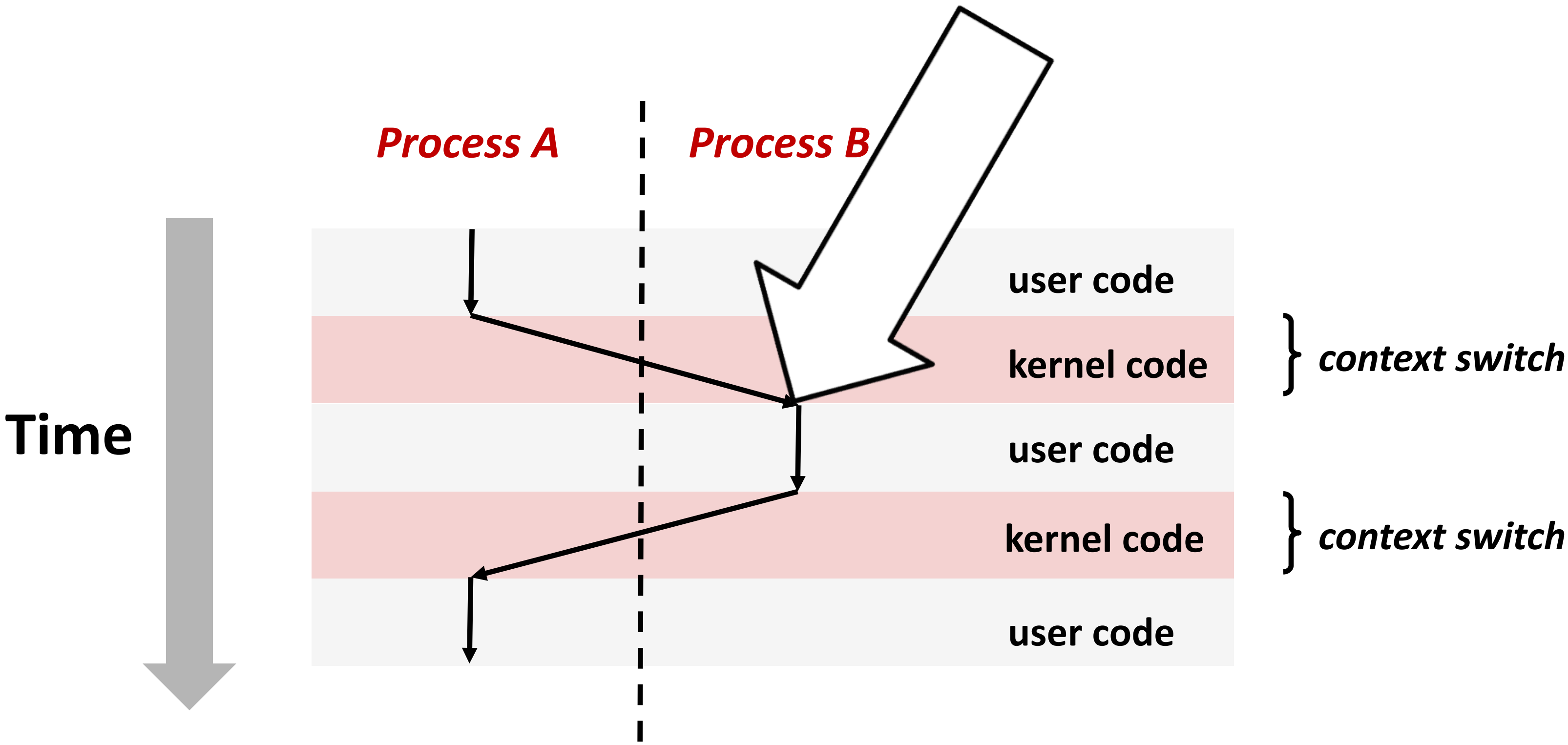
See "man ps" for more details

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) ;
        }

    /* Parent terminates the child processes */
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    /* Parent reaps terminated children */
    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 *p*



Important: All context switches are initiated by calling some exception handler.

Receiving Signals

Suppose kernel is returning from an exception handler and is ready to pass control to process *p*...

Kernel computes `pnb = pending & ~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 type of signal has a predefined *default action*

- The process terminates
- The process terminates and *dumps core*
- The process ignores the signal
- The process suspends execution

  (until restarted by a SIGCONT signal)

animation

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)
```

This parameter can be:

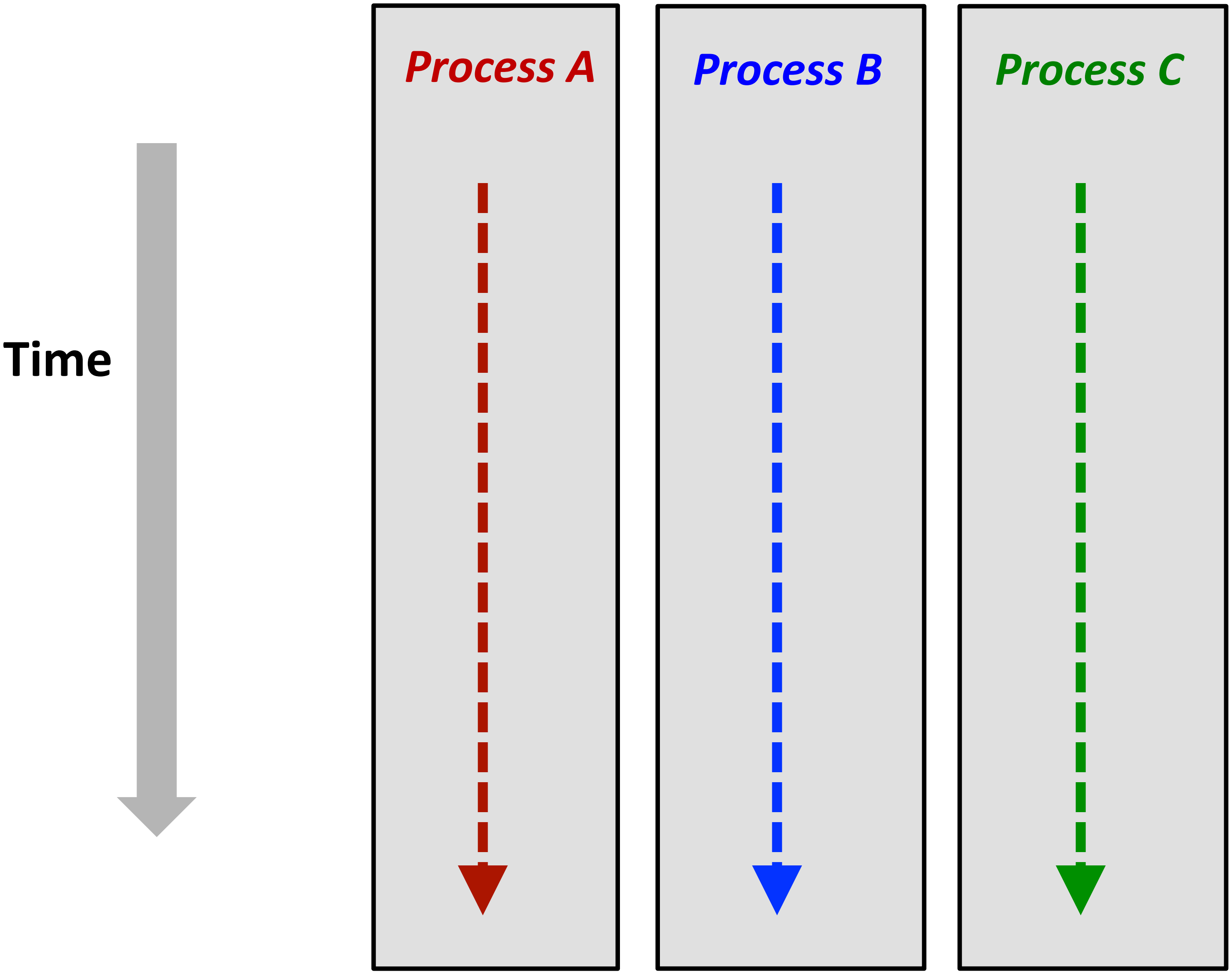
- **SIG_IGN**: Ignore signals of type **`signum`**
- **SIG_DFL**: Revert to the default action on receipt of signals of type **`signum`**
- Otherwise, **`handler`** is the address of a ***signal handler*** function
  - Called when process receives a signal of type **`signum`**.
  - Referred to as ***"installing"*** the handler.
  - Executing handler is called ***"catching"*** or ***"handling"*** the signal.
  - When the handler returns, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal.

Signal Handling Example

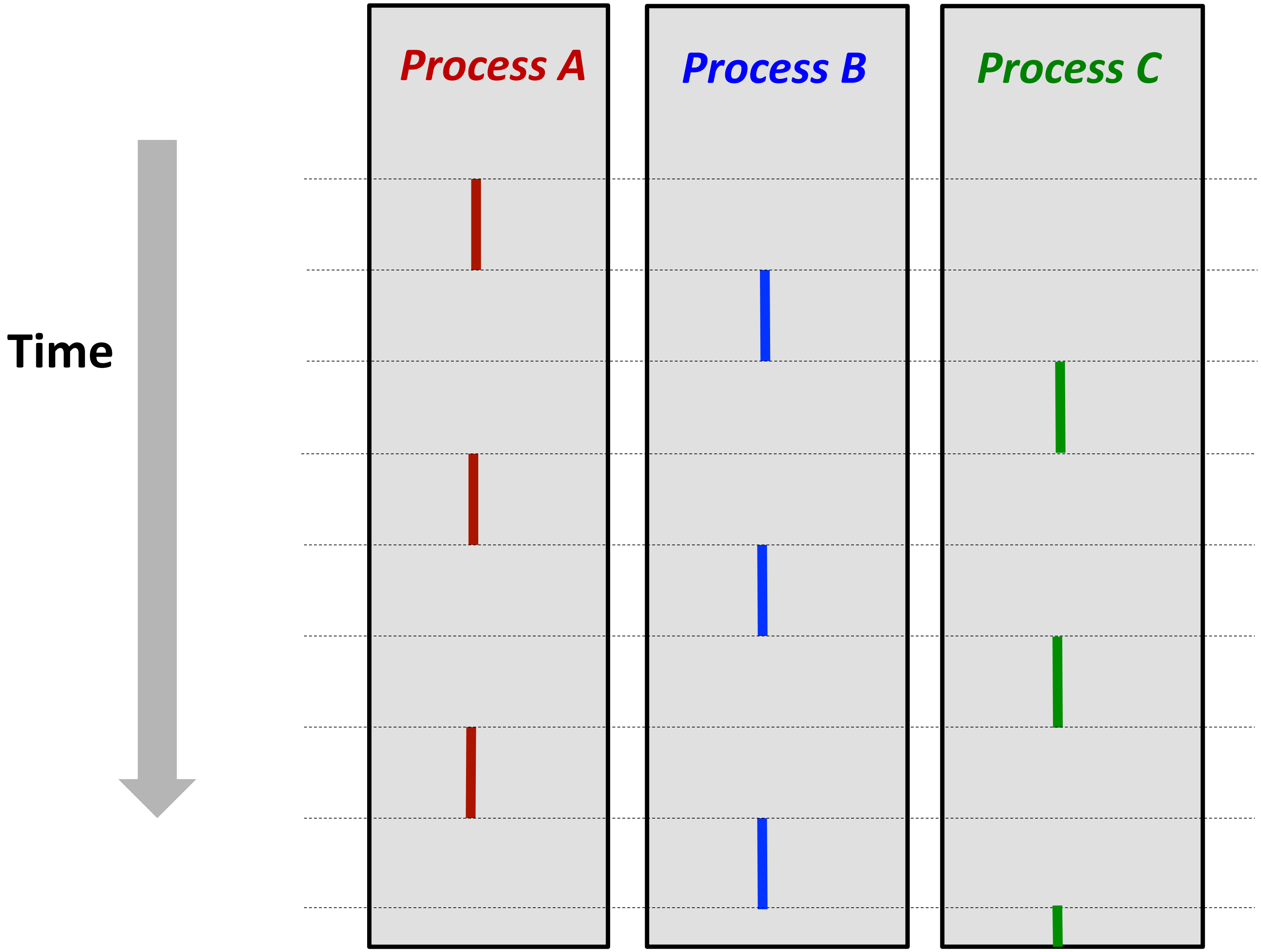
```
void int_handler(int sig) {
    safe_printf("Process %d received signal %d\n", getpid(), sig);
    exit(0);
}

void fork13() {
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            while(1); /* child infin
        }
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n"
        kill(pid[i], SIGINT);
    }
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_stat
        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);
    }
}
```

```
linux> ./fork13
Killing process 25417
Killing process 25418
Killing process 25419
Killing process 25420
Killing process 25421
Process 25417 received signal 2
Process 25418 received signal 2
Process 25420 received signal 2
Process 25421 received signal 2
Process 25419 received signal 2
Child 25417 terminated with exit status 0
Child 25418 terminated with exit status 0
Child 25420 terminated with exit status 0
Child 25419 terminated with exit status 0
Child 25421 terminated with exit status 0
linux>
```

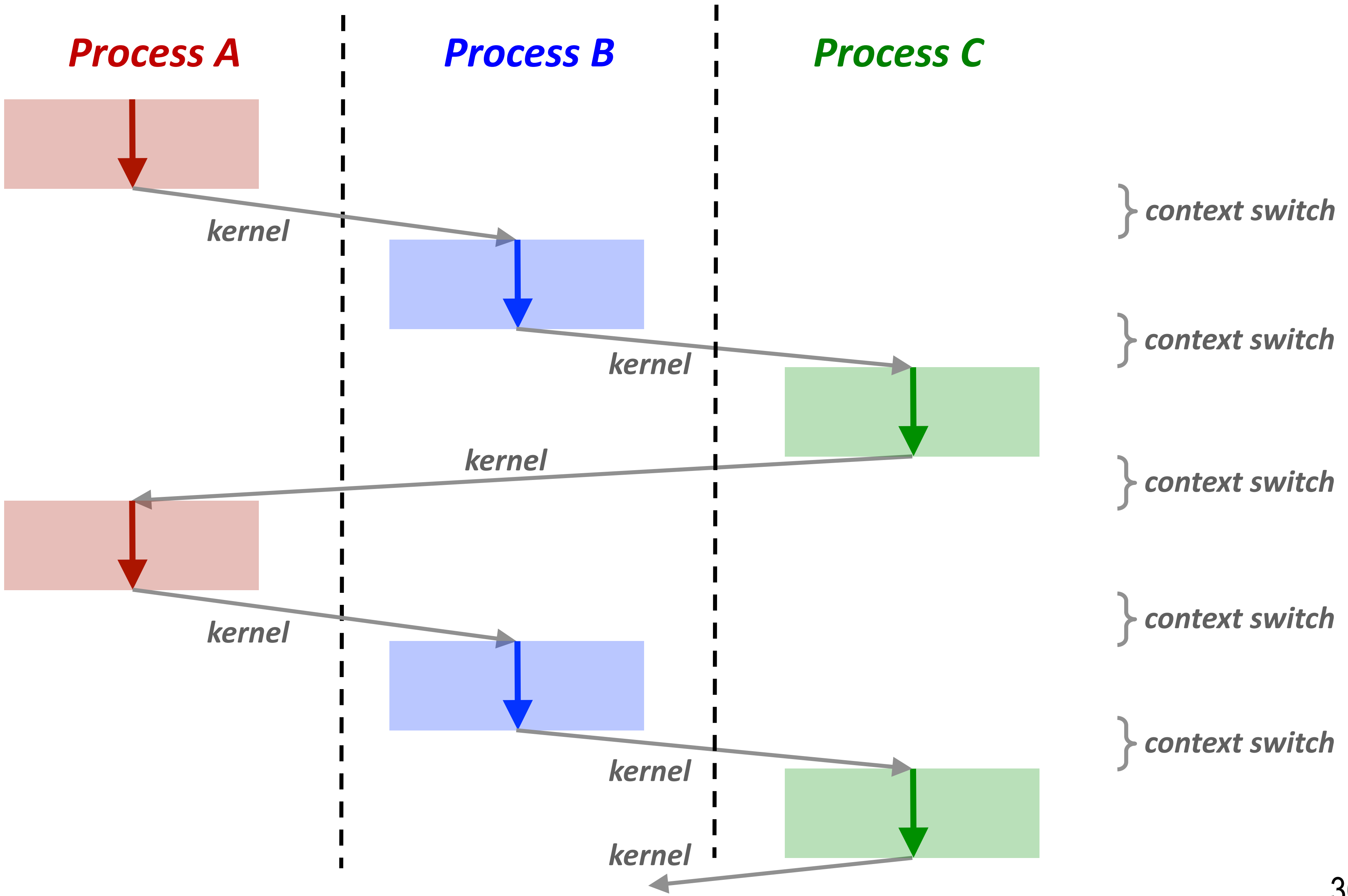
Concurrent Processes



Concurrent Processes



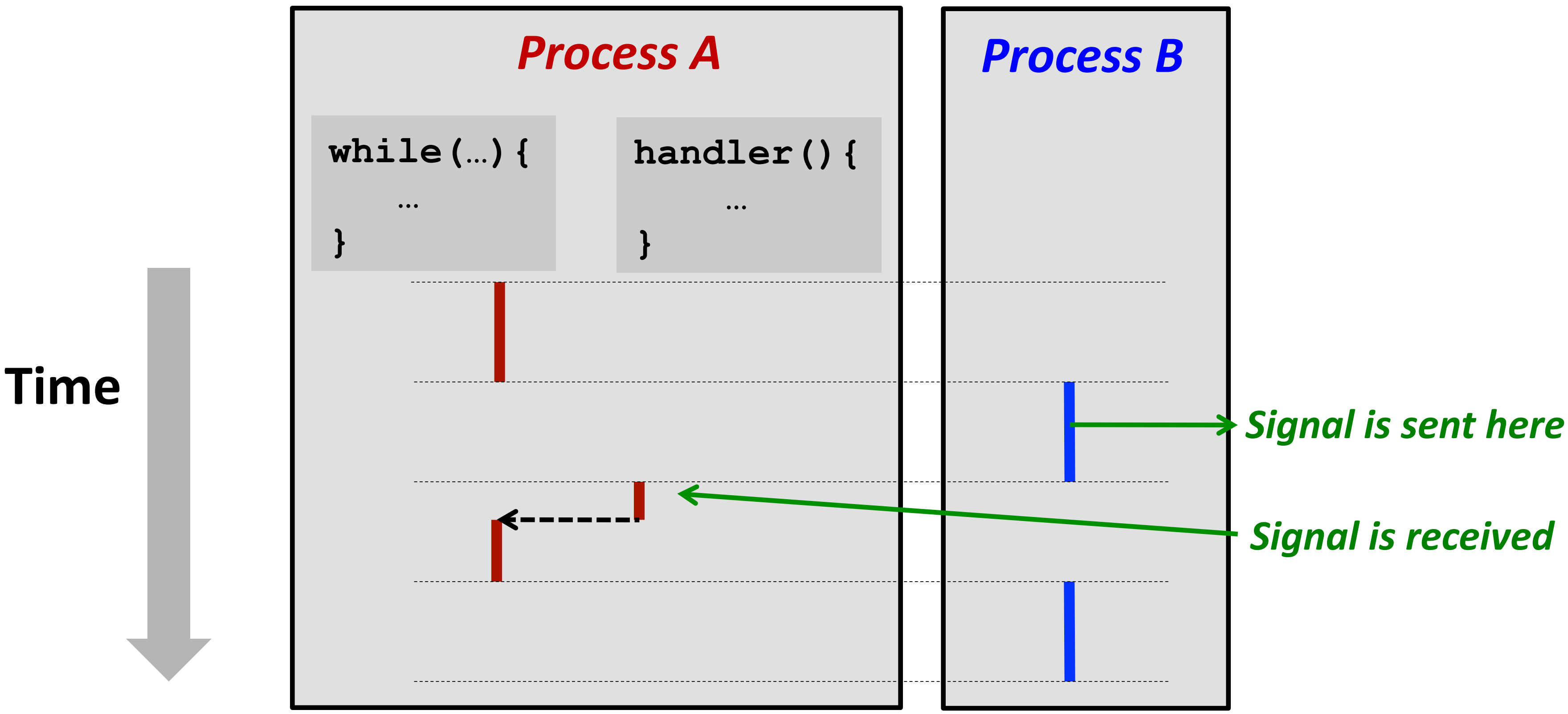
“Round-Robin” Process Scheduling



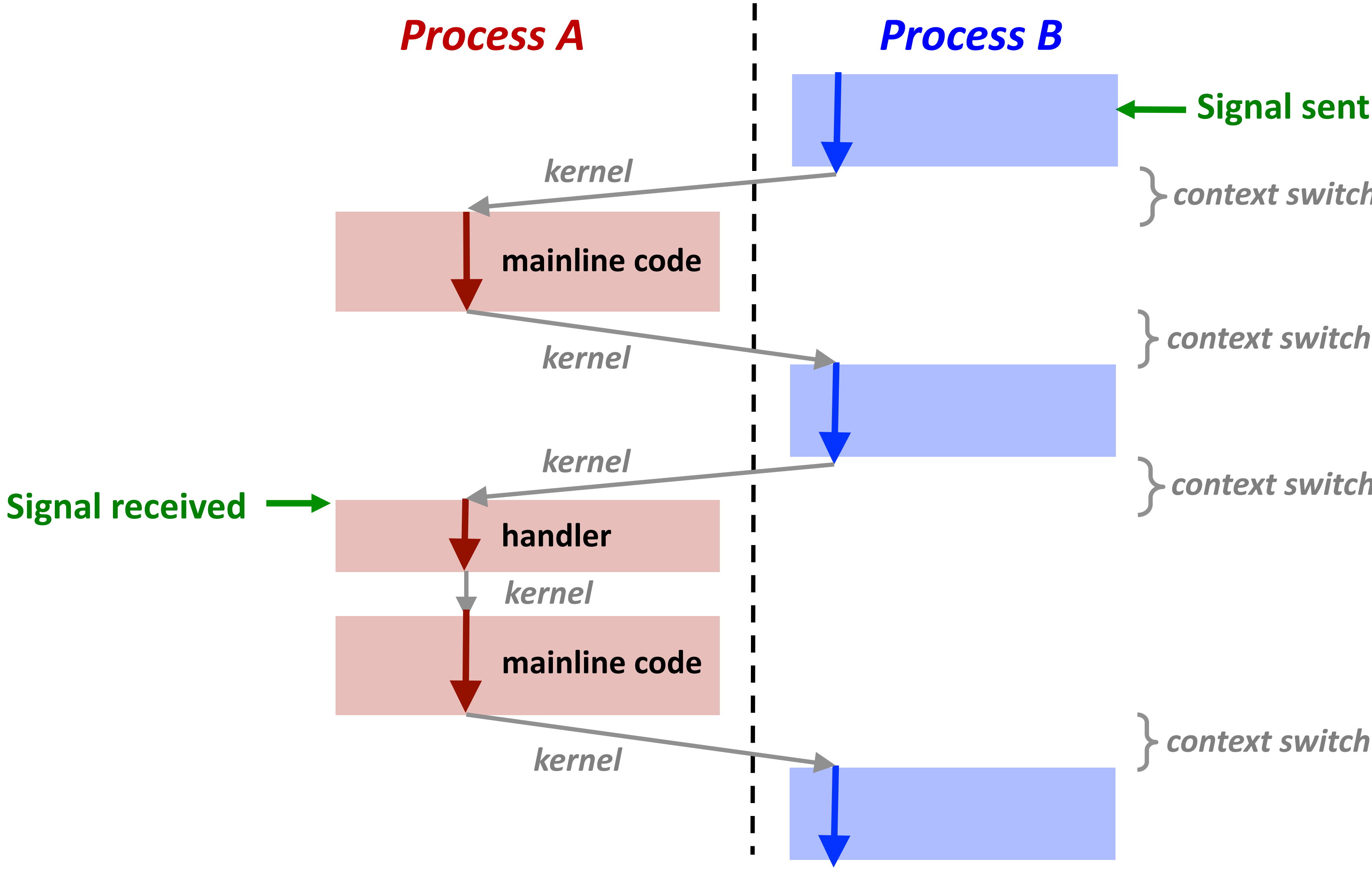
Signals Handlers as Concurrent Flows

- **A signal handler runs as a separate control flow that is "inserted" into the main program**

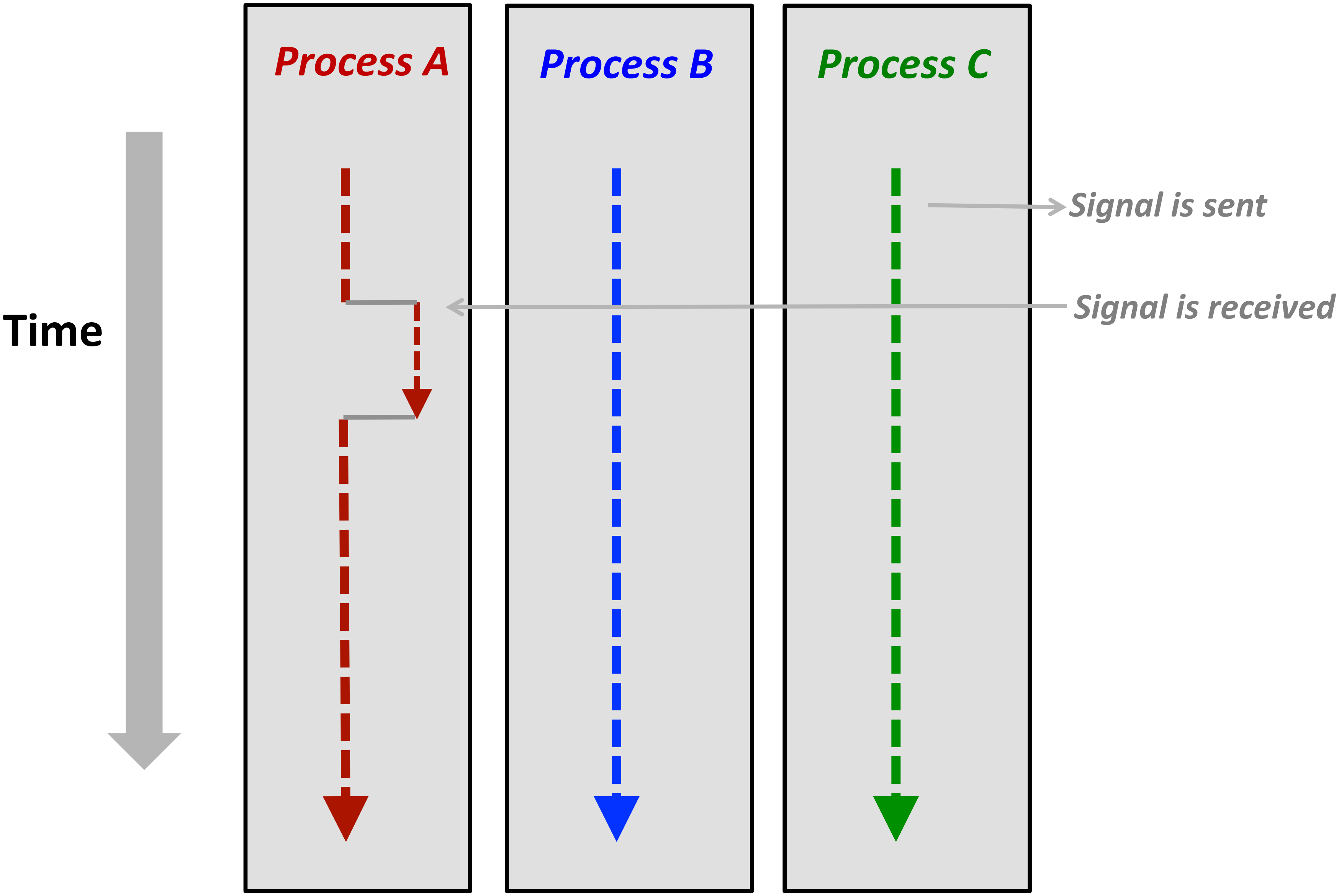
  The handler is not a separate process.



Another View of Signal Handlers as Concurrent Flows



Concurrent Processes



Signal "Funkiness"

Signal arrives during long system calls (say a `read`)?

- **Signal handler interrupts `read` call**
  - Linux: upon return from signal handler, the **`read`** call is restarted automatically
  - Some other flavors of Unix can cause the **`read`** call to fail with an **`EINTER`** error number (**`errno`**)
    In this case, the application program can restart the slow system call

Subtle differences like these complicate the writing of portable code that uses signals

Consult textbook for details

Safe Signal Handling

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

- Shared data structures can become corrupted.

Here are some guidelines to avoid trouble.

Guidelines for Writing Safe Handlers

- **Keep your handlers as simple as possible**
  - e.g., Set a global flag and return
- **Call only async-signal-safe functions in your handlers**
  - `printf`, `sprintf`, `malloc`, and `exit` are not safe!
- **Save and restore `errno` on entry and exit**
  - So the handler doesn't overwrite a value of `errno` that is in use
- **Protect accesses to shared data structures by temporarily blocking all signals.**
  - To prevent possible corruption
- **Declare global variables as `volatile`**
  - To prevent compiler from storing them in a register
- **Declare global flags as `volatile sig_atomic_t`**
  - *flag*: variable that is only read or written (e.g. flag = 1, not flag++)
  - Flag declared this way does not need to be protected like other globals

Async-Signal-Safety

A function is *async-signal-safe* if either reentrant or non-interruptible by signals.

Reentrant:

Can be "in execution" by several threads

Variables are local (stored on stack)

All accesses to non-local data are carefully managed

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

Safely Generating Formatted Output

Use the reentrant Sio (Safe I/O library) from `csapp.c` in your handlers.

```
ssize_t Sio_puts(char s[]) /* Put string */
ssize_t Sio_putl(long v)   /* Put long */
void Sio_error(char s[])   /* Put msg & exit */
```

```
/* Safe SIGINT handler */
void sigint_handler(int sig) {
    Sio_puts("You hit ctrl-c!\n");
    sleep(2);
    Sio_puts("Let me think...");
    sleep(1);
    Sio_puts("Good bye!\n");
    _exit(0);
}
```

```
linux> ./sigintsafe
<ctrl-c>
You hit ctrl-c!
Let me think...Good bye!
linux>
```

Correct Signal Handling

```
int ccount = 0;
void child_handler(int sig) {
    int olderrno = errno;
    pid_t pid;
    if ((pid = wait(NULL)) < 0)
        Sio_error("wait error");
    ccount--;
    Sio_puts("Handler reaped child ");
    Sio_putl((long)pid);
    Sio_puts(" \n");
    sleep(1);
    errno = olderrno;
}

void fork14() {
    pid_t pid[N];
    int i;
    ccount = N;
    Signal(SIGCHLD, child_handler);

    for (i = 0; i < N; i++) {
        if ((pid[i] = Fork()) == 0) {
            Sleep(1);
            exit(0);  /* Child exits */
        }
    }
    while (ccount > 0) /* Parent spins */
        ;
}
```

Pending signals are not queued

- For each signal type, one bit indicates whether or not signal is pending…
- …thus at most one pending signal of any particular type.

You can't use signals to count events, such as children terminating.

```
linux> ./fork14
Handler reaped child 23240
Handler reaped child 23241
```

… progam hangs here!

Correct Signal Handling

Must wait for all terminated child processes

- Put **`wait`** in a loop to reap all terminated children

```
void child_handler2(int sig)
{
    int olderrno = errno;
    pid_t pid;
    while ((pid = wait(NULL)) > 0) {
        ccount--;
        Sio_puts("Handler reaped child ");
        Sio_putl((long)pid);
        Sio_puts(" \n");
    }
    if (errno != ECHILD)
        Sio_error("wait error");
    errno = olderrno;
}
```

```
linux> ./forks 15
Handler reaped child 23246
Handler reaped child 23247
Handler reaped child 23248
Handler reaped child 23249
Handler reaped child 23250
linux>
```

A Program That Reacts to Internally Generated Events

```
#include <stdio.h>
#include <signal.h>

int beeps = 0;

/* SIGALRM handler */
void handler(int sig) {
  safe_printf("BEEP\n");

  if (++beeps < 5)
    alarm(1);
  else {
    safe_printf("BOOM!\n");
    exit(0);
  }
}
```

internal.c

```
main() {
  signal(SIGALRM, handler);
  alarm(1); /* send SIGALRM in
               1 second */

  while (1) {
    /* handler returns here */
  }
}
```

```
linux> ./internal
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
linux>
```

Nonlocal Jumps

Nonlocal Jumps: `setjmp/longjmp`

Powerful (but dangerous) user-level mechanism for transferring control to an arbitrary location

- Controlled to way to break the procedure call / return discipline
- Useful for error recovery and signal handling

`int setjmp(jmp_buf j)`

- Must be called before **`longjmp`**
- Identifies a return site for a subsequent longjmp
- Called once, returns one or more times

Implementation:

- Remember where you are by storing the current ***register context***, ***stack pointer***, and ***PC value*** in **`jmp_buf`**
- Return 0

setjmp/longjmp

void longjmp(jmp_buf j, int i)

- Meaning:
  - return from the **setjmp** remembered by jump buffer **j** again ...
  - ... this time returning **i** instead of 0
- Called after **setjmp**
- Called once, but never returns

longjmp Implementation:

- Restore register context (stack pointer, base pointer, PC value) from jump buffer **j**
- Set **%eax** (the return value) to **i**
- Jump to the location indicated by the PC stored in jump buf **j**

setjmp/longjmp

Goal: return directly (jump) out of a deeply-nested function.

```
void foo(void) {
    …
    if (errorXXX)
      longjmp(buf, 1);
    …
    bar();
}

void bar(void) {
    …
    if (errorYYY)
        longjmp(buf, 2);
    …
}
```

setjmp/longjmp

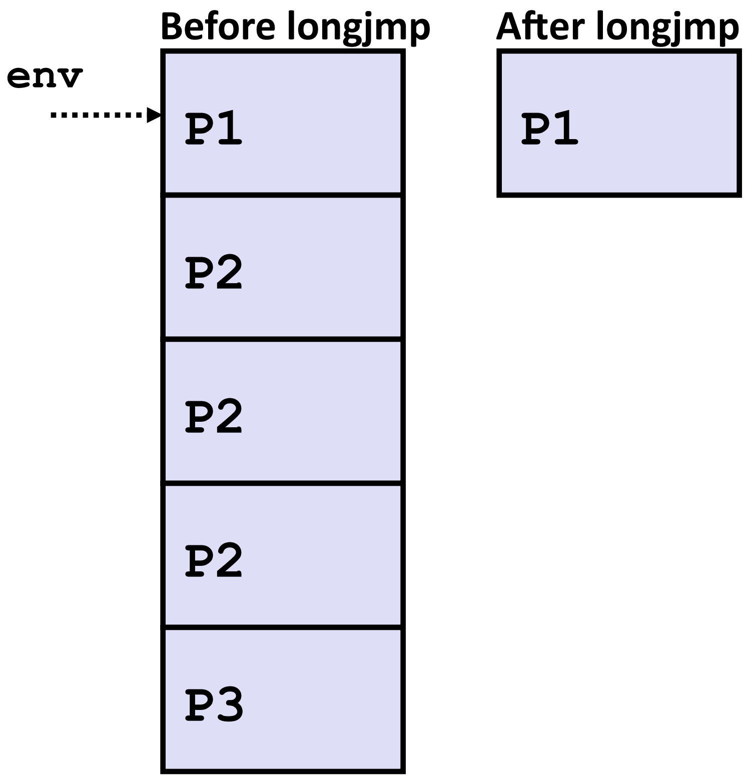
```
jmp_buf buf;

void foo(void), bar(void);

int main()
{
    switch(setjmp(buf)) {
    case 0:
        foo();
        break;
    case 1:
        printf("Detected an errorXXX condition in foo\n");
        break;
    case 2:
        printf("Detected an errorYYY condition in foo\n");
        break;
    default:
        printf("Unknown error condition in foo\n");
    }
}
```

Limitations of Long Jumps

Works within stack discipline

- Can only long jump to environment of function that has been called but not yet completed

```
jmp_buf env;

P1() {
  if (setjmp(env)) {
    /* Long Jump to here */
  } else {
    P2();
  }
}

P2(){
  … P2(); … P3();
}

P3() {
  longjmp(env, 1);
}
```

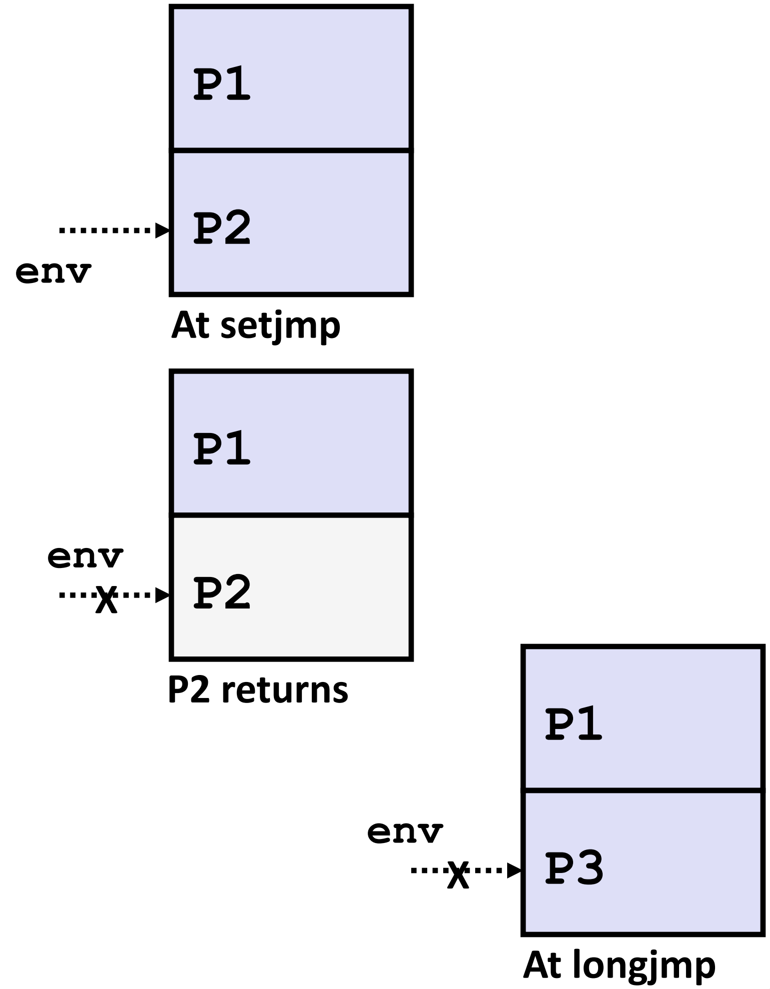


Limitations of Long Jumps

Works within stack discipline

Can only long jump to environment of function that has been called but not yet completed

```
jmp_buf env;

P1() {
  P2(); P3();
}

P2() {
   if (setjmp(env)) {
    /* Long Jump to here */
  }
}

P3() {
  longjmp(env, 1);
}
```



Putting It All Together: A Program That Restarts Itself When `ctrl-c`'d

```
#include "csapp.h"

sigjmp_buf buf;

void handler(int sig) {
    siglongjmp(buf, 1);
}

int main() {
    if (!sigsetjmp(buf, 1)) {
        Signal(SIGINT, handler);
        Sio_puts("STARTING\n");
    }
    else
        Sio_puts("RESTART !\n");

    while(1) {
        Sleep(1);
        Sio_puts("processing...\n");
    }
    exit(0); /* Control never reaches here */
}
```

```
linux> ./restart
STARTING
processing...
processing...
processing...        <- Ctrl-c
RESTART !
processing...
processing...
RESTART !            <- Ctrl-c
processing...
processing...
processing...
```

Summary

Signals provide process-level exception handling

- Can generate from user programs
- Can define effect by declaring signal handler

Some caveats

- Very high overhead
  - >10,000 clock cycles
  - Only use for exceptional conditions
- Don't have queues
  - Just one bit for each pending signal type

Nonlocal jumps provide exceptional control flow within process

- Within constraints of stack discipline