Exceptional Flow Control
Part II

ECF Exists at All Levels of a System

Exceptions

- Hardware and operating system kernel software

Process Context Switch

- Hardware timer and kernel software

Previous Lecture

Signals

- Kernel software and application software

This Lecture

Nonlocal jumps

- Application code

Shell Programs

A *shell* is an application program that runs programs on behalf of the user.

```
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

Shell operation

Commands typically run in foreground

- Shell waits until command finishes, then reaps it

Can place commands in the background

- Running a web server
  - `httpd &`
  - Shell creates new process, but continues
  - Can execute subsequent command without prior process returning

Implementation of `eval`

```
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;
}
```

Problem with Simple Shell 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 (typically) will not terminate.**
- **Creates a memory leak that will eventually crash the kernel when it runs out of memory.**

Solution: Reaping background jobs requires an alert mechanism.

- **The kernel will interrupt regular processing to alert us when a background process completes**
- **In Unix, the alert 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.

- **Kernel abstraction for exceptions and interrupts.**
- **Sent from the kernel (sometimes at the request of another process) to a process.**
- **Different signals are identified by small integer IDs (1-30)**

Signal basics

Sending a signal

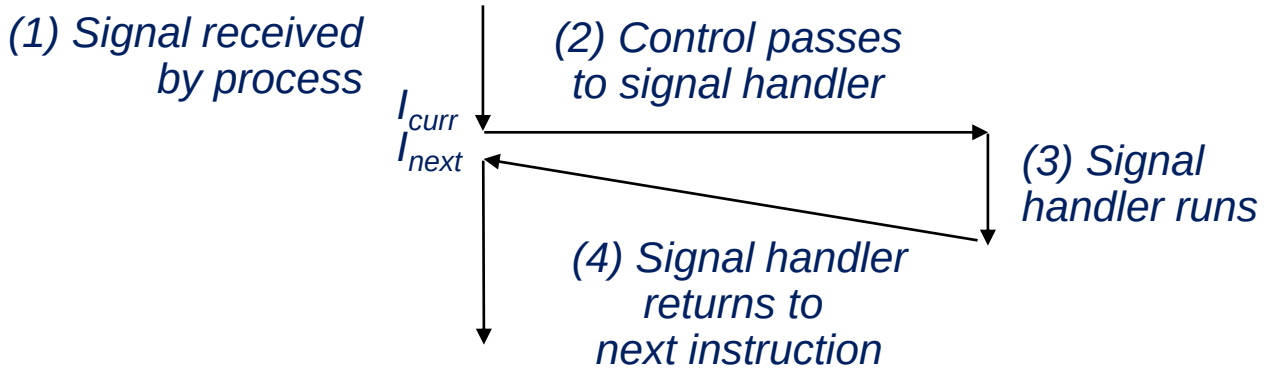
- **Kernel *sends* (delivers) a signal to a *destination process* by updating some state in the context of the destination process.**

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 & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

Signal basics

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
- Akin to a hardware exception handler being called in response to an asynchronous interrupt.



Signal terminology

A signal is *pending* if it has been 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 process can *block* the receipt of certain signals.

- **Blocked signals can eventually be delivered, but will not be received until the signal is unblocked.**

Signal implementation

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` whenever a signal of type k is delivered.
  - Kernel clears bit k in `pending` whenever a signal of type k is received
- `blocked` – represents the set of blocked signals
  - Can be set and cleared by the application using the `sigprocmask` function.

C interface

`kill()`

- Sends signal number `sig` to process `pid` if `pid` is greater than 0
- Sends signal number `sig` to process group `pid` if `pid` is less than 0
- Returns 0 on success, -1 on error

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

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);
    }
}
```

```
linux> ./sigint_nocatch
Killing process 18860
Killing process 18861
Killing process 18862
Killing process 18863
Killing process 18864
Killing process 18865
Killing process 18866
Killing process 18867
Killing process 18868
Killing process 18869
Child 18862 terminated abnormally
Child 18863 terminated abnormally
Child 18860 terminated abnormally
Child 18866 terminated abnormally
Child 18867 terminated abnormally
Child 18861 terminated abnormally
Child 18869 terminated abnormally
Child 18865 terminated abnormally
Child 18868 terminated abnormally
Child 18864 terminated abnormally
linux>
```


http://thefengs.com/wuchang/courses/cs201/class/17/sigint_nocatch

Receiving Signals

Kernel checks signals for a process *p* when it is ready to pass control to it

Kernel computes `pnb = pending & ~blocked`

- **The set of pending nonblocked signals for process *p***

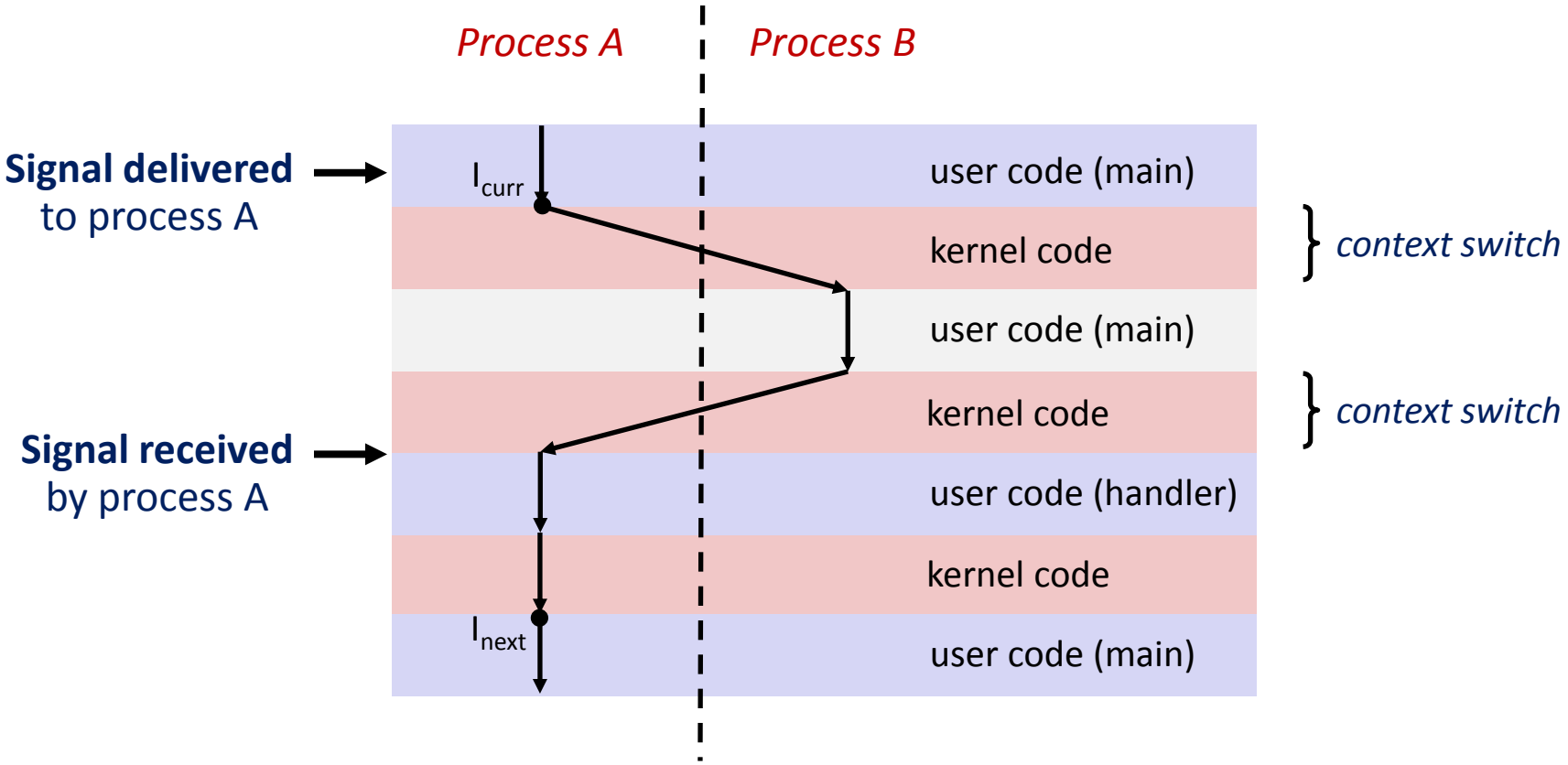
if (`pnb == 0`)

- **Pass control to next instruction in the logical flow for *p*.**

else

- **Choose least significant 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**

Signal handling



Default Actions

Each signal type has a predefined *default action*, which is one of:

- **The process terminates**
- **The process terminates and dumps core.**
- **The process stops until restarted by a SIGCONT signal.**
- **The process ignores the signal.**

Custom Signal Handlers

The `signal` function modifies the default action associated with the receipt of signal `signum`:

- `handler_t *signal(int signum, handler_t *handler)`
- Handler typically the address of a *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 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;
}
```

Signal Handling Example

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

int count = 5;
void handler(int sig) {
        printf("You think hitting ctrl-c works? %d more left!\n", count);
        count--;
        if (count == 0)
                exit(0);
}
int main() {
        signal(SIGINT, handler); /* installs ctrl-c handler */
        while (1) {}
}
```

```
linux> ./sigint
^CYou think hitting ctrl-c works? 5 more left!
^CYou think hitting ctrl-c works? 4 more left!
^CYou think hitting ctrl-c works? 3 more left!
^CYou think hitting ctrl-c works? 2 more left!
^CYou think hitting ctrl-c works? 1 more left!
linux>
```


http://thefengs.com/wuchang/courses/cs201/class/17/sigint_count

Signal Handling Example

```
void int_handler(int sig) {
    printf("Process %d received signal %d\n",
            getpid(), sig);
    exit(0);
}
int main() {
    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 infinite loop */
    /* 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);
    }
}
```

```
linux> ./forks 13
Killing process 24973
Killing process 24974
Killing process 24975
Killing process 24976
Killing process 24977
Process 24977 received signal 2
Child 24977 terminated with exit status 0
Process 24976 received signal 2
Child 24976 terminated with exit status 0
Process 24975 received signal 2
Child 24975 terminated with exit status 0
Process 24974 received signal 2
Child 24974 terminated with exit status 0
Process 24973 received signal 2
Child 24973 terminated with exit status 0
linux>
```


http://thefengs.com/wuchang/courses/cs201/class/17/sigint_catch

Signal Handler Funkiness

```
int ccount = N;
void child_handler(int sig) {
    int child_status;
    pid_t pid;
    printf("In child handler\n");
    if ((pid = wait(&child_status)) > 0) {
        ccount--;
        printf("Received signal %d from process
%d\n", sig, pid);
    }
}

int main() {
    pid_t pid[N];
    int i;
    signal(SIGCHLD, child_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            /* Child: Exit */
            exit(0);
        }
    while (ccount > 0)
        pause();/* Suspend until signal occurs */
    exit(0);
}
```

- Programmer wants parent to “wait” on each child before exiting
- Spot the bug
- Suggest a fix

http://thefengs.com/wuchang/courses/cs201/class/17/sigchld_broken

Signal Handler Funkiness

```
int ccount = N;
void child_handler(int sig) {
    int child_status;
    pid_t pid;
    printf("In child handler\n");
    if ((pid = wait(&child_status)) > 0) {
        ccount--;
        printf("Received signal %d from process
%d\n", sig, pid);
    }
}

int main() {
    pid_t pid[N];
    int i;
    signal(SIGCHLD, child_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            /* Child: Exit */
            exit(0);
        }
    while (ccount > 0)
        pause();/* Suspend until signal occurs */
    exit(0);
}
```

Pending signals are not queued

- Each signal type has a single bit indicating whether or not signal is pending even if multiple processes have sent a signal
- Parent can hang waiting for more signals if two are delivered at the same time (and only one wait is called in handler)
- Must check for all terminated children
  - Call wait in loop

http://thefengs.com/wuchang/courses/cs201/class/17/sigchld_broken

Signal Handler Funkiness

```
int ccount = N;
void child_handler(int sig) {
    int child_status;
    pid_t pid;
    printf("In child handler\n");
    while ((pid = waitpid(-1, &status, WNOHANG))> 0) {
        ccount--;
        printf("Received signal %d from process %d\n", sig, pid);
    }
}

int main() {
    pid_t pid[N];
    int i;
    signal(SIGCHLD, child_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            /* Child: Exit */
            exit(0);
        }
    while (ccount > 0)
        pause();/* Suspend until signal occurs */
    exit(0);
}
```

```
linux>  ./sigchld_noq
In child handler
Received signal 17 from process 19415
In child handler
Received signal 17 from process 19416
Received signal 17 from process 19417
In child handler
Received signal 17 from process 19418
Received signal 17 from process 19419
In child handler
Received signal 17 from process 19420
Received signal 17 from process 19421
In child handler
Received signal 17 from process 19422
Received signal 17 from process 19423
In child handler
Received signal 17 from process 19424
linux>
```


http://thefengs.com/wuchang/courses/cs201/class/17/sigchld_noq

Alarm signal

Similar to sleep, but delivers a signal instead of returning control to program

C interface

```
#include <unistd.h>
unsigned int alarm(unsigned int secs);
```

- **Sends a SIGALRM signal to current process after a specified time interval has elapsed**
- **Returns remaining secs of previous alarm or 0 if no previous alarm**

Example

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

int beeps = 0;

/* SIGALRM handler */
void handler(int sig) {
  printf("BEEP\n");
  fflush(stdout);

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

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

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

```
linux> a.out
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
bass>
```

Chapter summary

Exceptions

- Hardware and operating system kernel software

Concurrent processes

- Hardware timer and kernel software

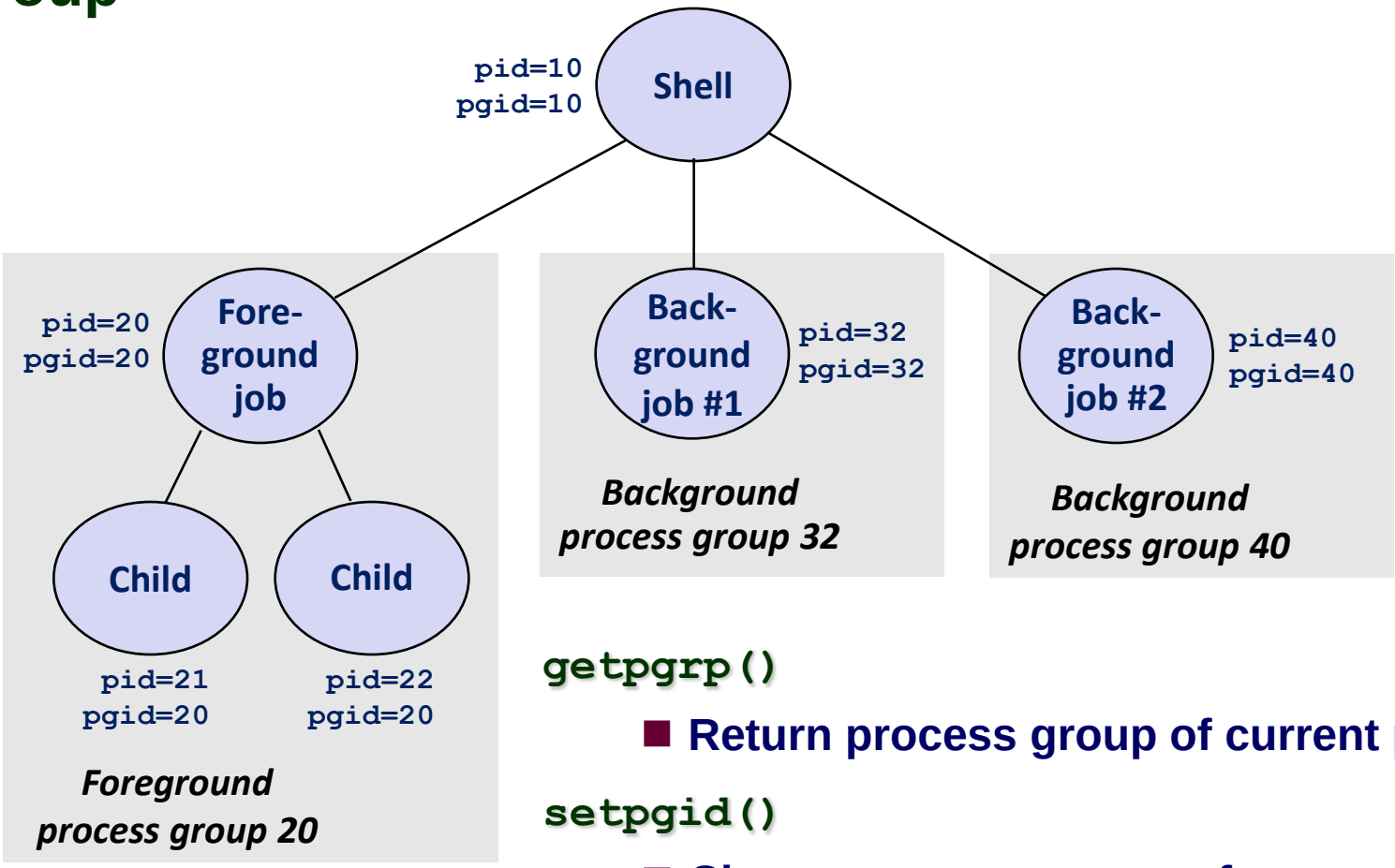
Signals

- Kernel software

Extra slides

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`

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

Examples

`/bin/kill -9 24818`
- Send SIGKILL to process 24818

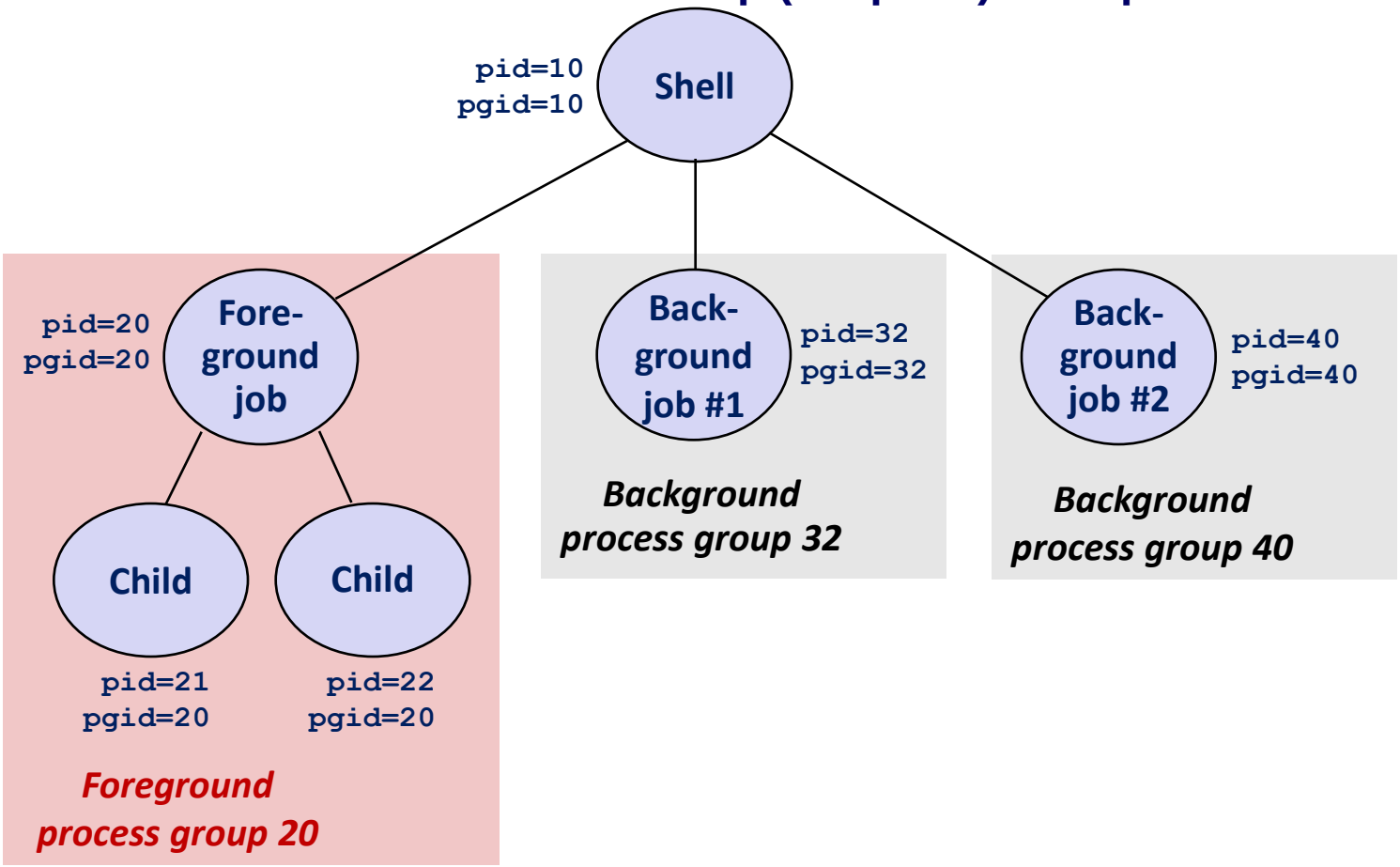
`/bin/kill -9 -24817`
- Send SIGKILL to every process in process group 24817

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

linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24818 pts/2    00:00:02 forks
24819 pts/2    00:00:02 forks
24820 pts/2    00:00:00 ps
linux> /bin/kill -9 -24817
linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24823 pts/2    00:00:00 ps
linux>
```

Sending Signals from the Keyboard

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

- **SIGTERM – default action is to terminate each process**
- **SIGSTOP – default action is to stop (suspend) each process**



Example of `ctrl-c` and `ctrl-z`

```
linux> ./forks 17
Child: pid=24868 pgrp=24867
Parent: pid=24867 pgrp=24867
 <typed ctrl-z>
Suspended
linux> ps a
  PID TTY      STAT   TIME COMMAND
24788 pts/2    S      0:00 -usr/local/bin/tcsh -i
24867 pts/2    T      0:01 ./forks 17
24868 pts/2    T      0:01 ./forks 17
24869 pts/2    R      0:00 ps a
bass> fg
./forks 17
<typed ctrl-c>
linux> ps a
  PID TTY      STAT   TIME COMMAND
24788 pts/2    S      0:00 -usr/local/bin/tcsh -i
24870 pts/2    R      0:00 ps a
```

STAT (process state)
Legend:
- S: sleeping
- T: stopped
- R: running

Blocking and Unblocking Signals

Implicit blocking mechanism

- Kernel blocks any pending signals of type currently being handled.
- E.g., A SIGINT handler can't be interrupted by another SIGINT

Explicit blocking and unblocking mechanism

- `sigprocmask` function

Supporting functions

- `sigemptyset` – Create empty set
- `sigfillset` – Add every signal number to set
- `sigaddset` – Add signal number to set
- `sigdelset` – Delete signal number from set

Temporarily 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);
```

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.

We'll explore concurrency issues later in the term.

For now here are some guidelines to help you avoid trouble.

Guidelines for Writing Safe Handlers

- **G0: Keep your handlers as simple as possible**
  - e.g., Set a global flag and return
- **G1: Call only async-signal-safe functions in your handlers**
  - `printf`, `sprintf`, `malloc`, and `exit` are not safe!
- **G2: Save and restore `errno` on entry and exit**
  - So that other handlers don't overwrite your value of `errno`
- **G3: Protect accesses to shared data structures by temporarily blocking all signals.**
  - To prevent possible corruption
- **G4: Declare global variables as `volatile`**
  - To prevent compiler from storing them in a register
- **G5: 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

Function is *async-signal-safe* if either reentrant (e.g., all variables stored on stack frame, CS:APP3e 12.7.2) 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

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 */
```

```
void sigint_handler(int sig) /* Safe SIGINT handler */
{
    Sio_puts("So you think you can stop the bomb with ctrl-
c, do you?\n");
    sleep(2);
    Sio_puts("Well...");
    sleep(1);
    Sio_puts("OK. :-)\n");
    _exit(0);
}
```

sigintsafe.c

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 */
        ;
}
```

forks.c

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.

```
whaleshark> ./forks 14
Handler reaped child 23240
Handler reaped child 23241
```

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;
}
```

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

Portable Signal Handling

- Ugh! Different versions of Unix can have different signal handling semantics
  - Some older systems restore action to default after catching signal
  - Some interrupted system calls can return with errno == EINTR
  - Some systems don't block signals of the type being handled
- Solution: `sigaction`

```
handler_t *Signal(int signum, handler_t *handler)
{
    struct sigaction action, old_action;

    action.sa_handler = handler;
    sigemptyset(&action.sa_mask); /* Block sigs of type being handled */
    action.sa_flags = SA_RESTART; /* Restart syscalls if possible */

    if (sigaction(signum, &action, &old_action) < 0)
        unix_error("Signal error");
    return (old_action.sa_handler);
}
```

csapp.c

Synchronizing Flows to Avoid Races

Simple shell with a subtle synchronization error because it assumes parent runs before child.

```
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) { /* Child */
            Execve("/bin/date", argv, NULL);
        }
        Sigprocmask(SIG_BLOCK, &mask_all, &prev_all); /* Parent */
        addjob(pid);  /* Add the child to the job list */
        Sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
    exit(0);
}
```


procmask1.c

Synchronizing Flows to Avoid Races

SIGCHLD handler for a simple shell

```
void handler(int sig)
{
    int olderrno = errno;
    sigset_t mask_all, prev_all;
    pid_t pid;

    Sigfillset(&mask_all);
    while ((pid = waitpid(-1, NULL, 0)) > 0) { /* Reap child */
        Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
        deletejob(pid); /* Delete the child from the job list */
        Sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
    if (errno != ECHILD)
        Sio_error("waitpid error");
    errno = olderrno;
}
```

procmask1.c

Corrected Shell Program without Race

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


procmask2.c

Explicitly Waiting for Signals

Handlers for program explicitly waiting for SIGCHLD to arrive.

```
volatile sig_atomic_t pid;

void sigchld_handler(int s)
{
    int olderrno = errno;
    pid = Waitpid(-1, NULL, 0); /* Main is waiting for nonzero pid */
    errno = olderrno;
}

void sigint_handler(int s)
{
}
```

waitforsignal.c

Explicitly Waiting for Signals

Similar to a shell waiting for a foreground job to terminate.

```
int main(int argc, char **argv) {
    sigset_t mask, prev;
    Signal(SIGCHLD, sigchld_handler);
    Signal(SIGINT, sigint_handler);
    Sigemptyset(&mask);
    Sigaddset(&mask, SIGCHLD);

    while (1) {
    Sigprocmask(SIG_BLOCK, &mask, &prev); /* Block SIGCHLD */
    if (Fork() == 0) /* Child */
            exit(0);
    /* Parent */
    pid = 0;
    Sigprocmask(SIG_SETMASK, &prev, NULL); /* Unblock SIGCHLD */

    /* Wait for SIGCHLD to be received (wasteful!) */
    while (!pid)
            ;
    /* Do some work after receiving SIGCHLD */
        printf(".");
    }
    exit(0);
}
```


waitforsignal.c

Explicitly Waiting for Signals

Program is correct, but very wasteful

Other options:

```
while (!pid)  /* Race! */
          pause();
```

```
while (!pid) /* Too slow! */
    sleep(1);
```

Solution: `sigsuspend`

Waiting for Signals with `sigsuspend`

```
int sigsuspend(const sigset_t *mask)
```

Equivalent to atomic (uninterruptable) version of:

```
sigprocmask(SIG_BLOCK, &mask, &prev);
pause();
sigprocmask(SIG_SETMASK, &prev, NULL);
```

Waiting for Signals with sigsuspend

```
int main(int argc, char **argv) {
    sigset_t mask, prev;
    Signal(SIGCHLD, sigchld_handler);
    Signal(SIGINT, sigint_handler);
    Sigemptyset(&mask);
    Sigaddset(&mask, SIGCHLD);

    while (1) {
        Sigprocmask(SIG_BLOCK, &mask, &prev); /* Block SIGCHLD */
        if (Fork() == 0) /* Child */
            exit(0);

       /* Wait for SIGCHLD to be received */
        pid = 0;
        while (!pid)
            Sigsuspend(&prev);

       /* Optionally unblock SIGCHLD */
        Sigprocmask(SIG_SETMASK, &prev, NULL);
    /* Do some work after receiving SIGCHLD */
        printf(".");
    }
    exit(0);
}
```


sigsuspend.c

Nonlocal Jumps: `setjmp/longjmp`

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

- Controlled 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` (cont)

`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 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` Example

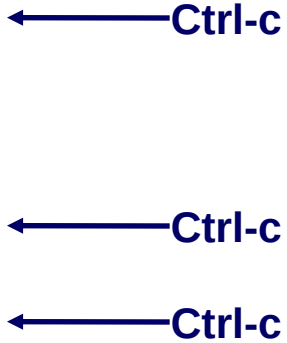
```
#include <setjmp.h>
jmp_buf buf;

main() {
   if (setjmp(buf) != 0) {
      printf("back in main due to an error\n");
   else
      printf("first time through\n");
   p1(); /* p1 calls p2, which calls p3 */
}
...
p3() {
   <error checking code>
   if (error)
      longjmp(buf, 1)
}
```

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

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

sigjmp_buf buf;

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

main() {
  signal(SIGINT, handler);

  if (setjmp(buf)==0)
    printf("starting\n");
  else
    printf("restarting\n");
```

```
while(1) {
    sleep(1);
    printf("processing...\n");
  }
}
```

```
bass> a.out
starting
processing...
processing...
restarting
processing...
processing...
processing...
restarting
processing...
restarting
processing...
processing...
```

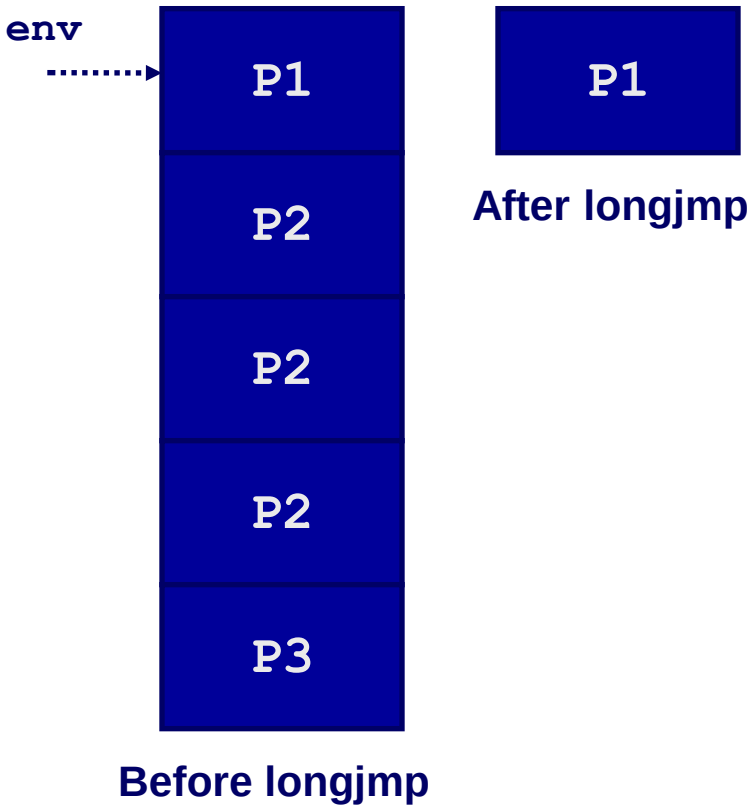


Limitations of Nonlocal Jumps

Works within stack discipline

- Can only long jump to environment of function that has been called but not yet completed
- Good: P1's stack frame still valid

```
jmp_buf env;

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

P2()
{  . . . P2(); . . . P3(); }

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

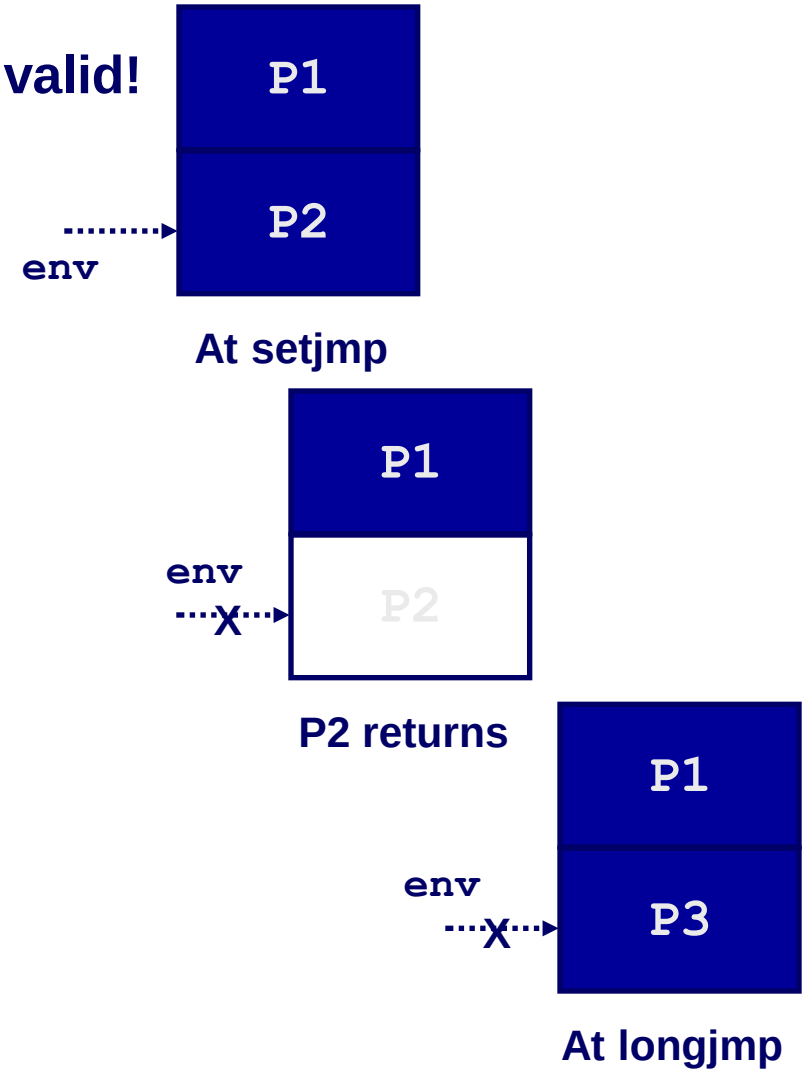


Limitations of Long Jumps (cont.)

Works within stack discipline

- Can only long jump to environment of function that has been called but not yet completed
- Bad: Need P2's stack frame to be valid!

```
jmp_buf env;

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

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

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



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