Exceptional Control Flow: Signals

Read Chap 8.5-8.8

Instructor: Jennifer Wong-Ma
jwongma@uci.edu

Portions adapted from Bryant & O'Halleron Textbook References

ECF Exists at All Levels of a System

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

Previous Lecture

This Lecture

Textbook

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); /* Returns 1 if bg process (last arge ‘&’) , 0 if fg process */
    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) /* Child is reaped */
                unix_error("waitfg: waitpid error");
        }
        else
            printf("%d %s", pid, cmdline);
    }
    return;
}
```

shellex.c

Problem with Simple Shell Example

- **Our example shell correctly waits for and reaps foreground jobs**

- **But what about background jobs?**
  - They will become zombies when they terminate
  - They will never be reaped because shell (typically) will not terminate
  - They will create a memory leak that could run the kernel out of memory

ECF to the Rescue!

- **Solution: Exceptional control flow**
  - 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 a system event of some type has occurred**
  - Closely related to exceptions and interrupts (low-level H/W events)
    - *Exceptions:* abrupt change in control flow due to processor's state
    - *Interrupts: async signal* from I/O
  - Sent from the kernel (sometimes at the request of another process) to a process
  - Identifier to the processes that a type of exception has occurred for the user process

Signals

- Signal type is identified by small 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 & Dump**	**Segmentation violation**
14	**SIGALRM**	**Terminate**	**Timer signal**
17	**SIGCHLD**	**Ignore**	**Child stopped or terminated**

Signal Concepts: Sending a Signal

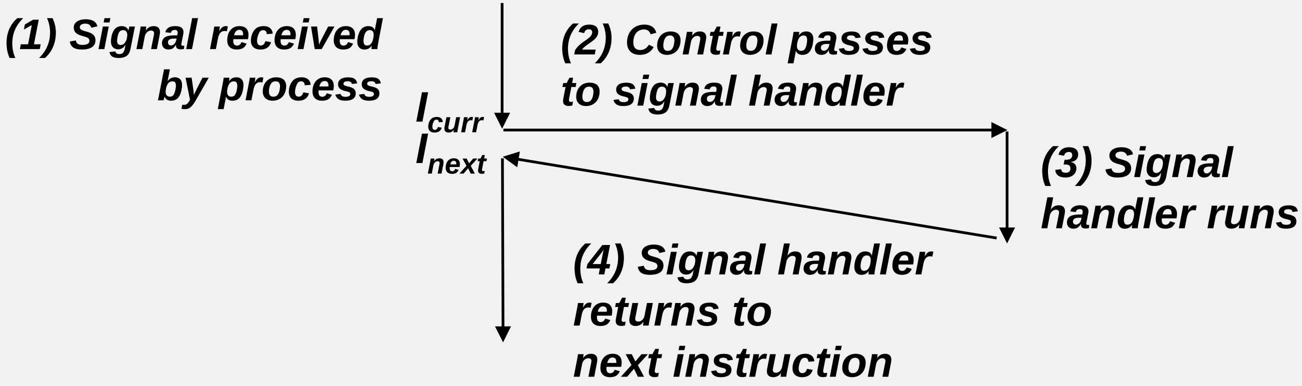
- **Kernel *sends* (delivers) a signal to a *destination process* by updating some state in the context of the destination process**

- **Kernel sends a signal for one of the following reasons:**
  - Ex: process divides by zero - H/W event
    - Detected by H/W, exception needs to occur. Kernel communicates this to process through SIGFPE (Floating point error) event
  - Ex: “Ctrl+c” on foreground process - S/W event
    - Detected by kernel, sends SIGINT (interrupt from keyboard) to each process in foreground process group
- **A processes can send a signal to itself**

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***
    - Closely related to a hardware exception handler being called in response to an asynchronous interrupt:


(1) Signal received by process
I_{curr}
I_{next}
(2) Control passes to signal handler
(3) Signal handler runs
(4) Signal handler returns to next instruction

Signal Concepts: 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 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*.

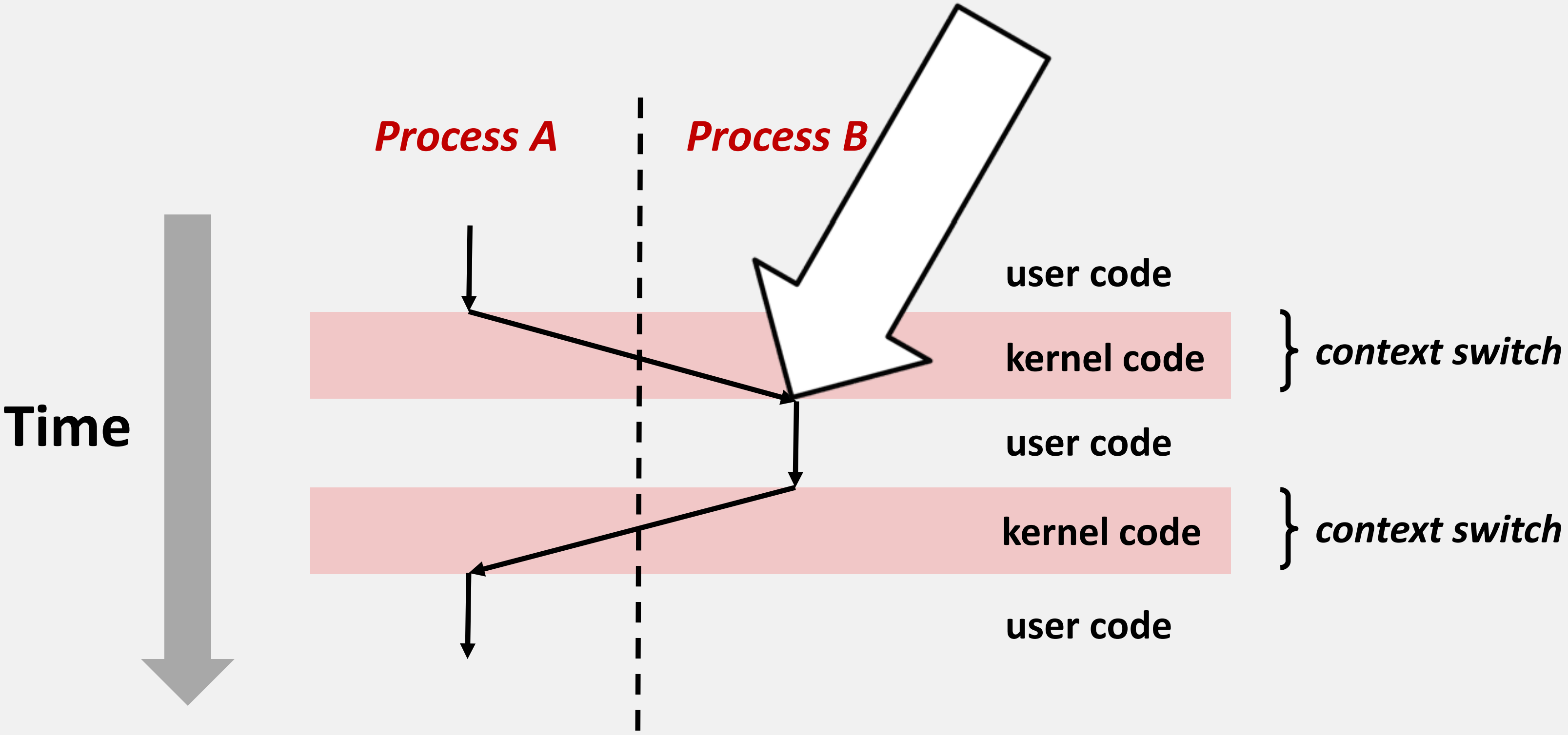
Sending Signals with `kill` Function

```
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], SIGKILL);
    }

    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

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 non-blocked 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 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
- **Ex: SIGKILL – terminate the process receiving signal**
- **Ex: SIGCHILD – ignore the signal**
- **Ex: SIGSTOP – stop until next SIGCONT (suspended process)**

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 ***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

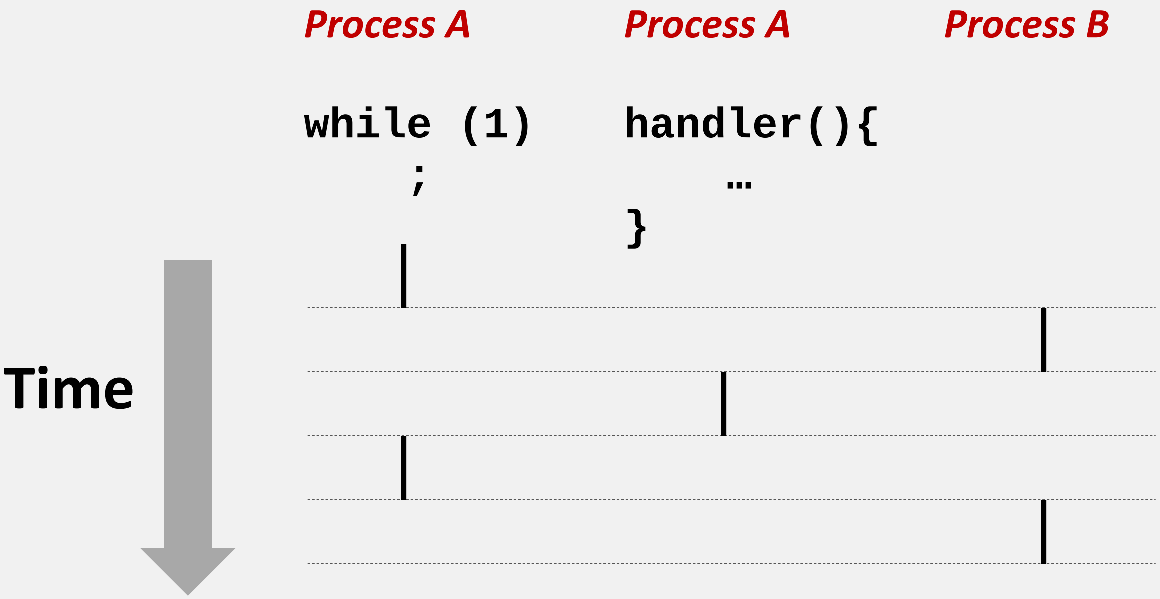
Signal Handling Example – Ctrl+C

```
typedef void (*handler_t)(int);
handler_t *signal(int signum, handler_t *handler);

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()
{
    if (signal(SIGINT, sigint_handler) == SIG_ERR) /* Install the SIGINT handler */
        unix_error("signal error");
    pause(); /* Wait for the receipt of a signal */
    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



Process A

```
while (1)
    ;
```

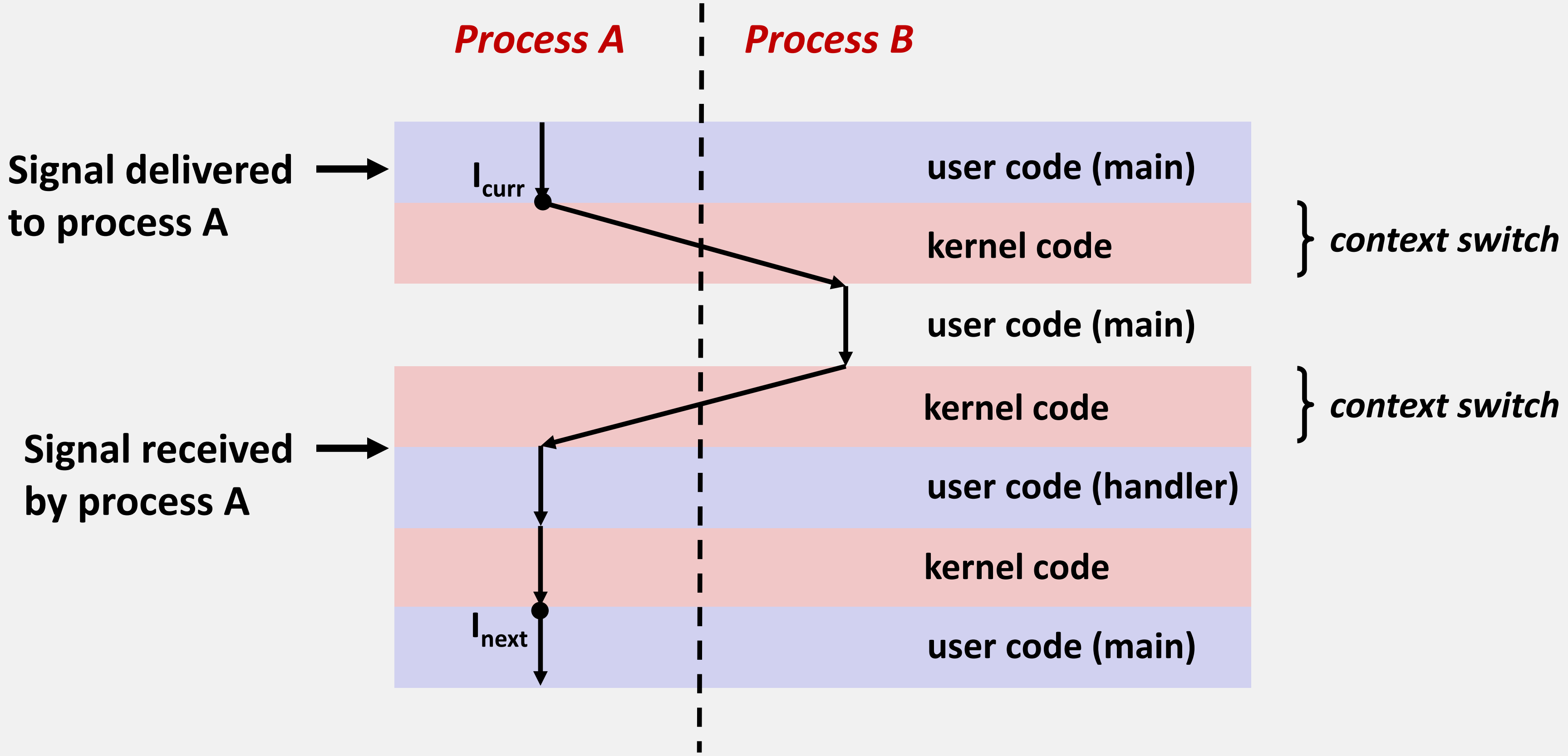
Process A

```
handler(){
    …
}
```

Process B

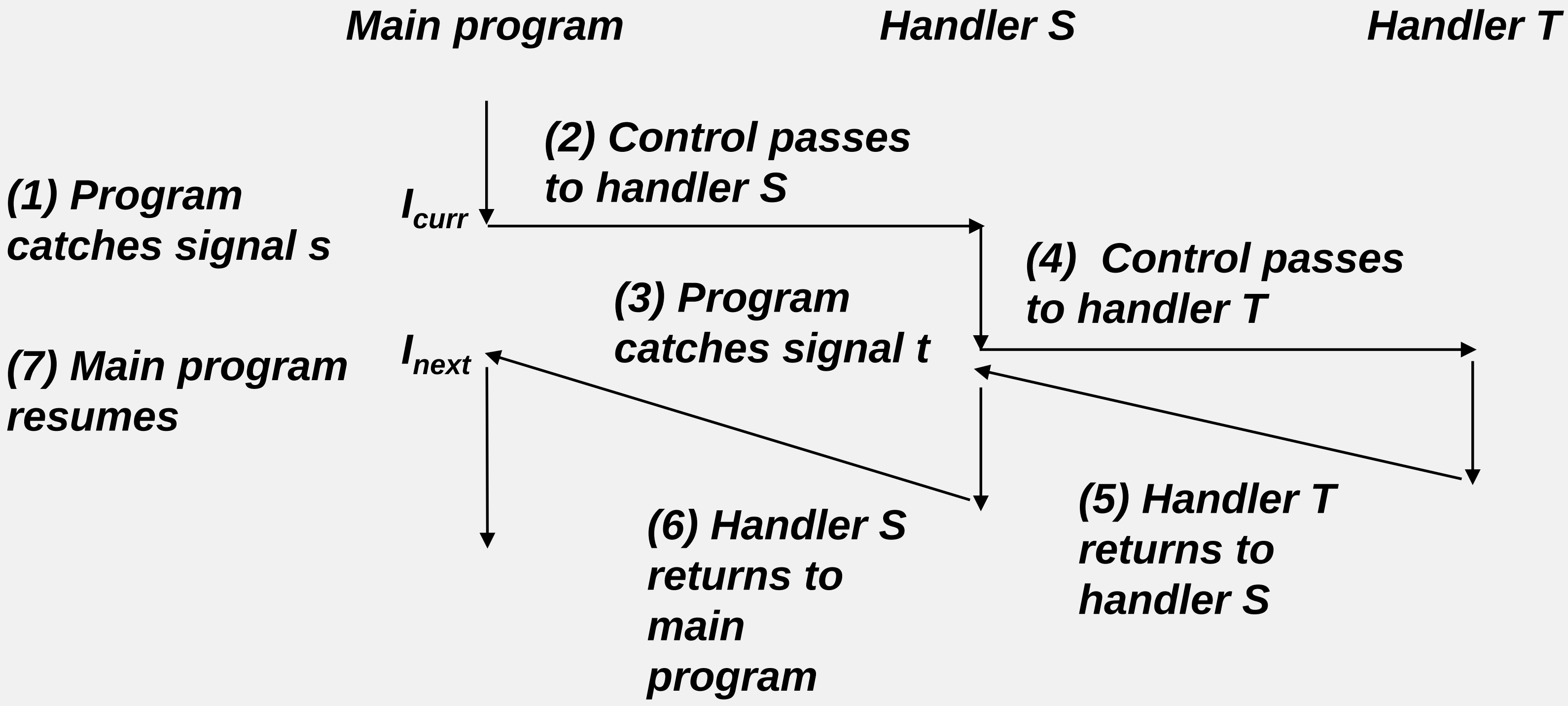
Time

Another View of Signal Handlers as Concurrent Flows



Nested Signal Handlers

- Handlers can be interrupted by other handlers



Main program

Handler S

Handler T

(1) Program catches signal s

I_{curr}

(2) Control passes to handler S

(3) Program catches signal t

(4) Control passes to handler T

(5) Handler T returns to handler S

(6) Handler S returns to main program

(7) Main program resumes

I_{next}

Blocking and Unblocking Signals

- **Temporary blocking of signals is way to prevent interrupts during critical parts of your code**
  - If signals arrive in that part of the program, they are delivered later, after you unblock them
- **Ex: Sharing data between a signal handler and rest of program**
  - If the type of the data is not atomic (completed in 1 instruction), then the signal handler could run when the rest of the program has only half finished reading or writing the data. Result: confusing consequences!
  - *Solution:* Block the signal handler from running while the rest of the program is examining or modifying that data—by blocking the appropriate signal around the parts of the program that touch the data

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

Signal Handling

```c
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);               /* Pretend cleanup work */
    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) {
             printf( “Hi %d, (int) getpid());
            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
Hi 23240
Hi 23241
Hi 23242
Handler reaped child 23240
Handler reaped child 23241
…wait forever…
```

Correct Signal Handling

- **Must wait for all terminated child processes**
  - Put `wait` **in a loop to <u>reap all terminated children</u>**

```
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
    - Must include code to manually restart interrupted system calls!
  - Some systems don't block signals of the type being handled
- **Solution:** `sigaction` **wrapper**

```
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); /* Removes the child to the job list */
    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

```c
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, mask_child, prev_one;

    Sigfillset(&mask_all);
    Sigemptyset(&mask_child);
    Sigaddset(&mask_child, SIGCHLD);
    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        Sigprocmask(SIG_BLOCK, &mask_child, &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

Summary

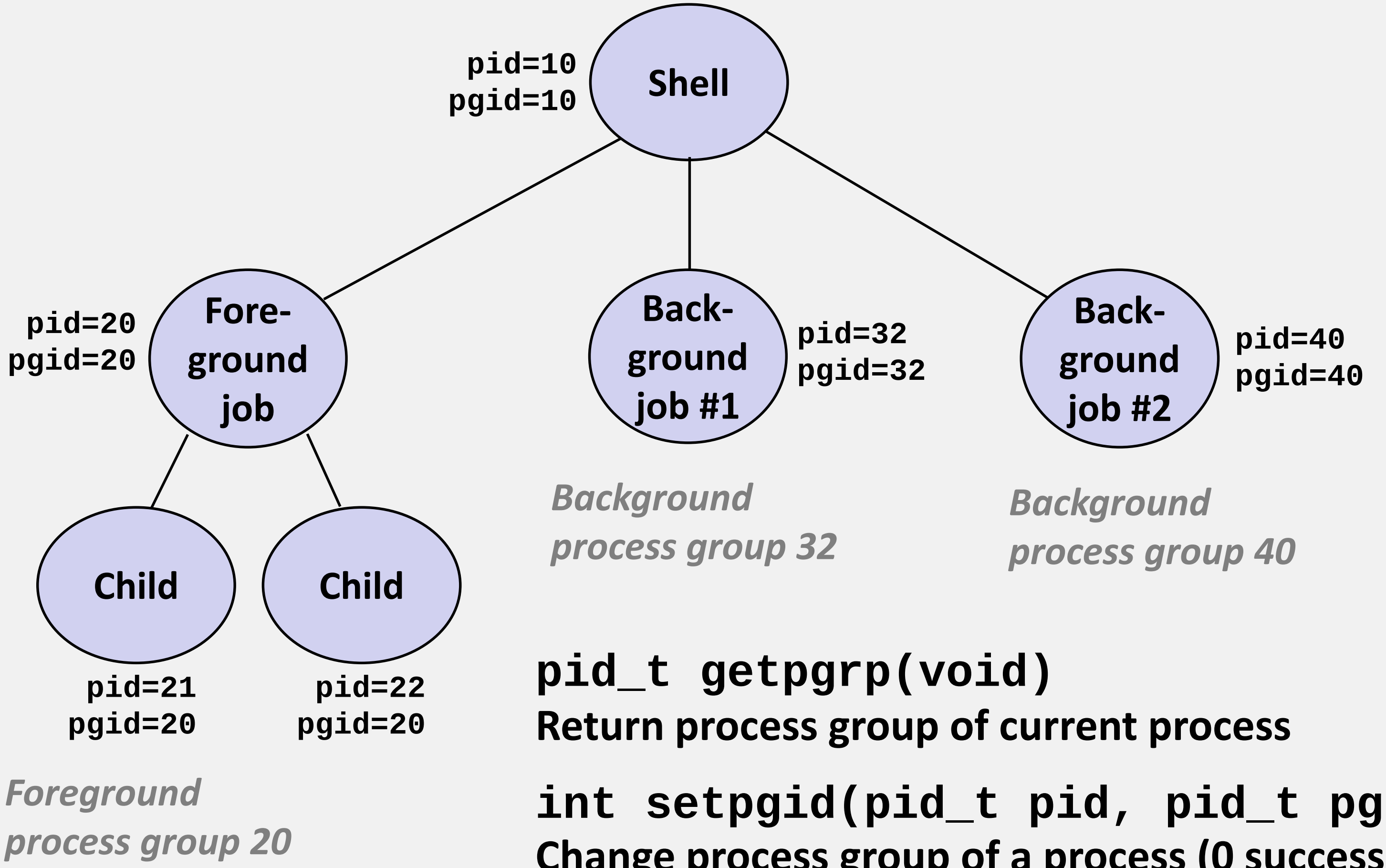
- **Signals provide process-level exception handling**
  - Can generate from user programs
  - Can define effect by declaring signal handler
  - Be very careful when writing signal handlers

- **Nonlocal jumps provide exceptional control flow within process**
  - Within constraints of stack discipline

Extra Slides

Sending Signals: Process Groups

- Every process belongs to exactly one process group



```
pid_t getpgrp(void)
```

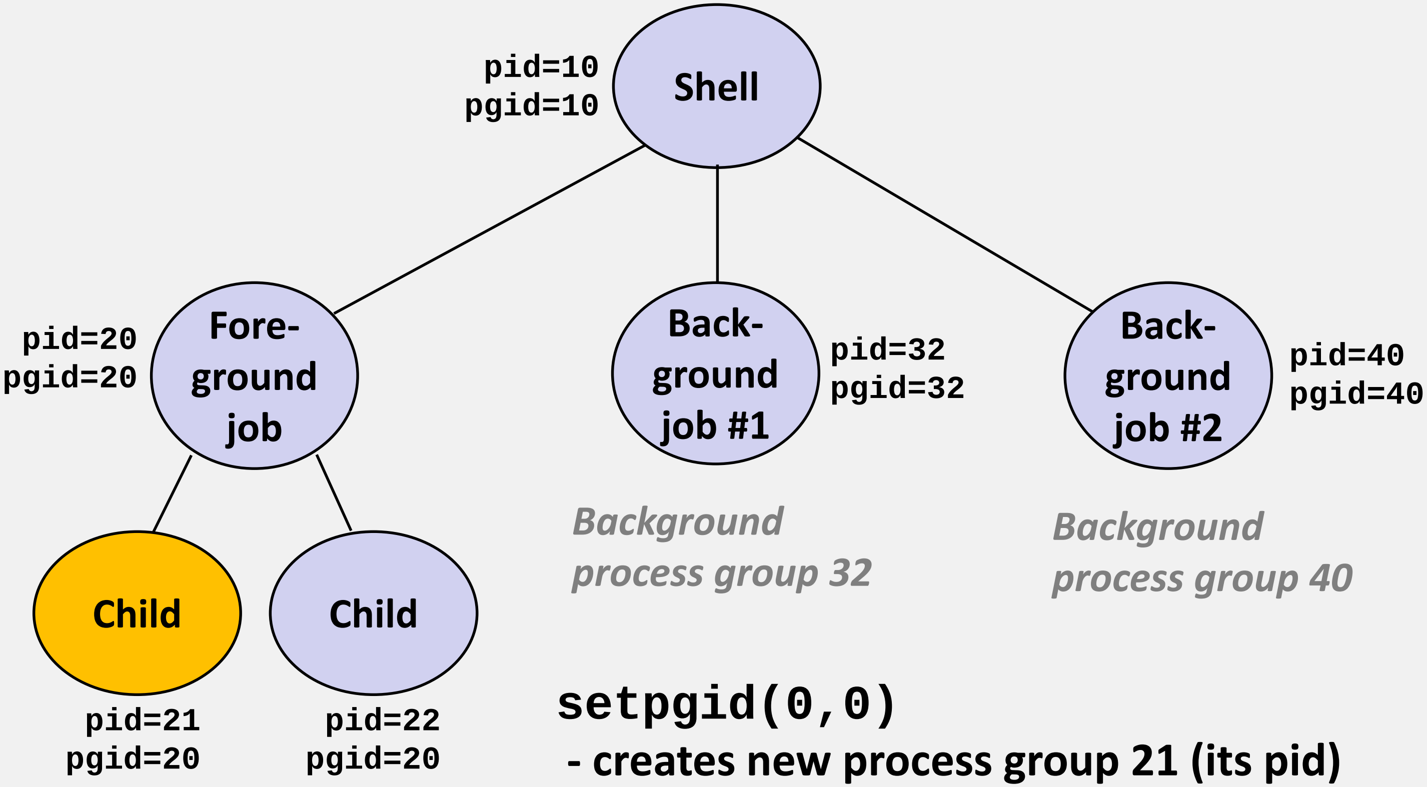
Return process group of current process

```
int setpgid(pid_t pid, pid_t pgid)
```

Change process group of a process (0 success, -1 error)

Sending Signals: Process Groups

- Every process belongs to exactly one process group



Foreground process group 20

`setpgid(0,0)`

- creates new process group 21 (its pid)

- adds itself to the group

Sending Signals with /bin/`kill` Program

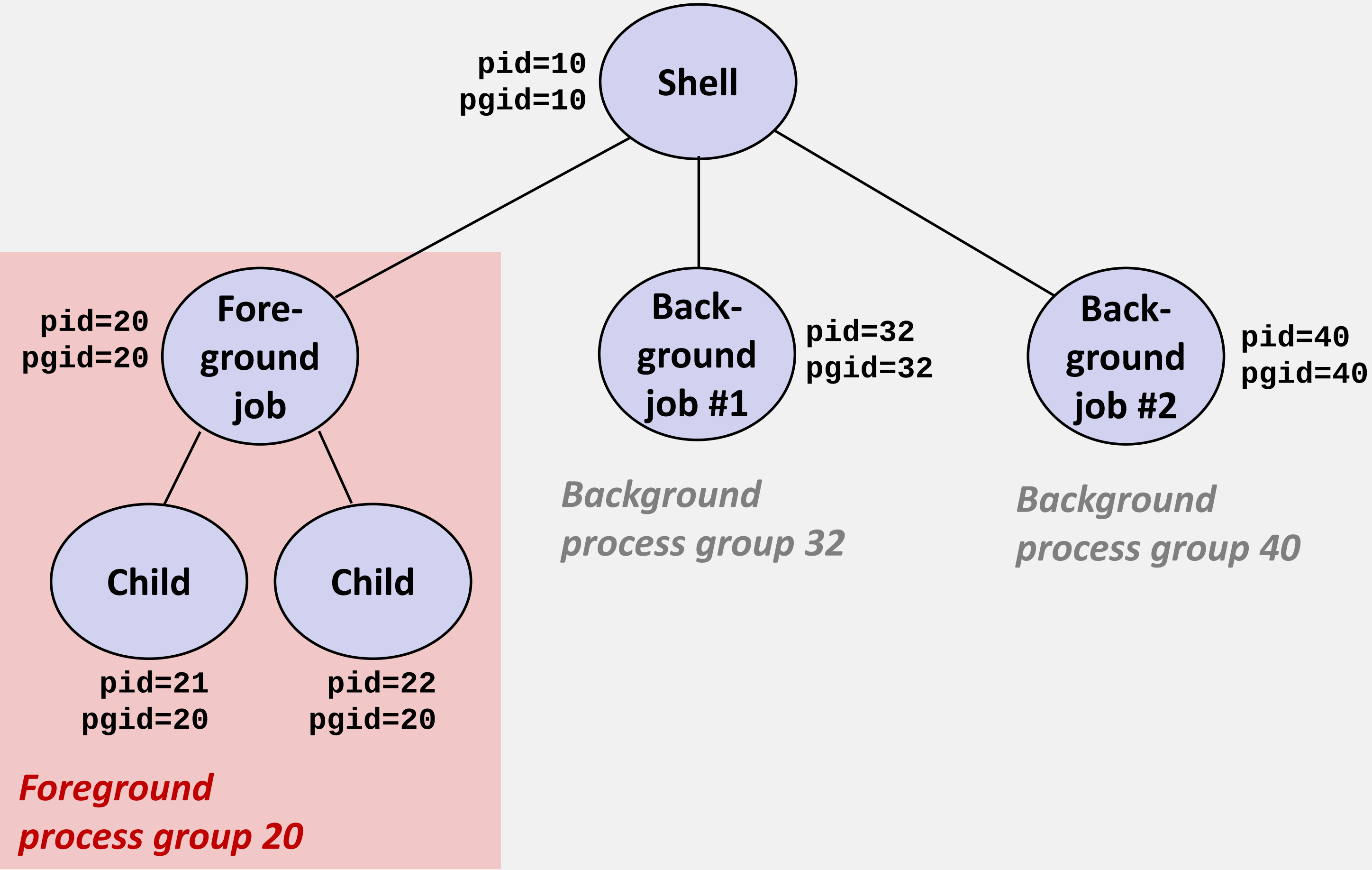
- `/bin/kill` **program sends arbitrary signal to a process or process group**

- **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) causes the kernel to send a SIGINT (SIGTSTP) to every job in the foreground process group.**
  - SIGINT – default action is to terminate each process
  - 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