CSCI2467: Systems Programming Concepts

Slideset 7: Exceptional Control Flow

Source: CS:APP Chapter 8, Bryant & O'Hallaron

Course Instructors:

Matthew Toups
Caitlin Boyce

Course Assistants:

Saroj Duwal
David McDonald

Spring 2020

THE UNIVERSITY *of* NEW ORLEANS

DEPARTMENT OF COMPUTER SCIENCE

Shell lab

- Two due dates! Two!

Shell lab

- Two due dates! Two!
- You need a lot of time to write and debug your code

Shell lab

- Two due dates! Two!
- You need a lot of time to write and debug your code
- Pace yourselves, the work can be split into distinct sections

Shell lab

- Two due dates! Two!
- You need a lot of time to write and debug your code
- Pace yourselves, the work can be split into distinct sections
- First due date is **Friday March 13**

Shell lab

- Two due dates! Two!
- You need a lot of time to write and debug your code
- Pace yourselves, the work can be split into distinct sections
- First due date is **Friday March 13**

- You should be reading through lab writeup and Chapter 8 now

Shell lab

- Two due dates! Two!
- You need a lot of time to write and debug your code
- Pace yourselves, the work can be split into distinct sections
- First due date is **Friday March 13**
  - You should be reading through lab writeup and Chapter 8 now
  - The next few lectures will be very relevant

Your demo on Friday

- Show `eval()` **working**

  (however this will not be your final version)
- Can your shell run simple programs?

Your demo on Friday

- Show `eval()` **working**

  (however this will not be your final version)
- Can your shell run simple programs?

```
tsh> /bin/ps
     (ps output)
tsh> /bin/ls -l
     (long ls output)
tsh> ./myspin 4
     (runs for 4 seconds then ends)
```

Tips

- Start `eval()` with:

Tips

- Start `eval()` with:

```
bg = parseline(cmdline, argv);
```

Tips

- Start `eval()` with:

  ```
  bg = parseline(cmdline, argv);
  ```

- but before that, must declare variable names:

Tips

- Start `eval()` with:

  `bg = parseline(cmdline, argv);`

- but before that, must declare variable names:

  `int bg;` and `char *argv[MAXARGS];`

Tips

- Start `eval()` with:

  `bg = parseline(cmdline, argv);`
- but before that, must declare variable names:

  `int bg;` and `char *argv[MAXARGS];`
- No need to `strcpy()` in eval

Tips

- Start eval() with:

  `bg = parseline(cmdline, argv);`

- but before that, must declare variable names:

  `int bg;` and `char *argv[MAXARGS];`

- No need to strcpy() in eval
- In lieu of Fork() (capital F):

Tips

- Start `eval()` with:

  `bg = parseline(cmdline, argv);`

- but before that, must declare variable names:

  `int bg;` and `char *argv[MAXARGS];`

- No need to `strcpy()` in eval
- In lieu of `Fork()` (capital F):

  See slide "System call error handling" (44) for example of calling `fork()` which checks for error (-1)

Tips

- Start `eval()` with:

  `bg = parseline(cmdline, argv);`
- but before that, must declare variable names:

  `int bg;` and `char *argv[MAXARGS];`
- No need to `strcpy()` in eval
- In lieu of `Fork()` (capital F):

  See slide "System call error handling" (44) for example of calling `fork()` which checks for error (-1)
- After checking `fork()` for error case (< 1), do the parent/child split

Tips

- Start `eval()` with:

  `bg = parseline(cmdline, argv);`

- but before that, must declare variable names:

  `int bg;` and `char *argv[MAXARGS];`

- No need to `strcpy()` in eval
- In lieu of `Fork()` (capital F):

  See slide "System call error handling" (44) for example of calling `fork()` which checks for error (-1)

- After checking `fork()` for error case (< 1), do the parent/child split

  (result of `fork` was saved in `pid`)

Tips

- Start `eval()` with:

  `bg = parseline(cmdline, argv);`
- but before that, must declare variable names:

  `int bg;` and `char *argv[MAXARGS];`
- No need to `strcpy()` in eval
- In lieu of `Fork()` (capital F):

  See slide "System call error handling" (44) for example of calling `fork()` which checks for error (-1)
- After checking `fork()` for error case (< 1), do the parent/child split

  (result of `fork` was saved in `pid`)

  Use `if(pid == 0)` to begin child's code

What to expect Friday

- We have many students so we need to do this quickly

What to expect Friday

- We have many students so we need to do this quickly
- Detailed help available after class

What to expect Friday

- We have many students so we need to do this quickly
- Detailed help available after class
- Does your shell actually **run** other programs?

What to expect Friday

- We have many students so we need to do this quickly
- Detailed help available after class
- Does your shell actually **run** other programs?

```
parseline(cmdline, argv)
```

What to expect Friday

- We have many students so we need to do this quickly
- Detailed help available after class
- Does your shell actually **run** other programs?

  `parseline(cmdline, argv)`

  `fork()` and check for error (< 0)

What to expect Friday

- We have many students so we need to do this quickly
- Detailed help available after class
- Does your shell actually **run** other programs?

  `parseline(cmdline, argv)`

  `fork()` and check for error (< 0)

  `if (pid == 0)` to begin child-only code

What to expect Friday

- We have many students so we need to do this quickly
- Detailed help available after class
- Does your shell actually **run** other programs?

  `parseline(cmdline, argv)`

  `fork()` and check for error (< 0)

  `if (pid == 0)` to begin child-only code

  `execve(argv[0], argv, environ)`

What to expect Friday

- We have many students so we need to do this quickly
- Detailed help available after class
- Does your shell actually **run** other programs?

  `parseline(cmdline, argv)`

  `fork()` and check for error (< 0)

  `if (pid == 0)` to begin child-only code

  `execve(argv[0], argv, environ)`

  (either executes, or prints “not found” error message)

Bottom line

not so bad!

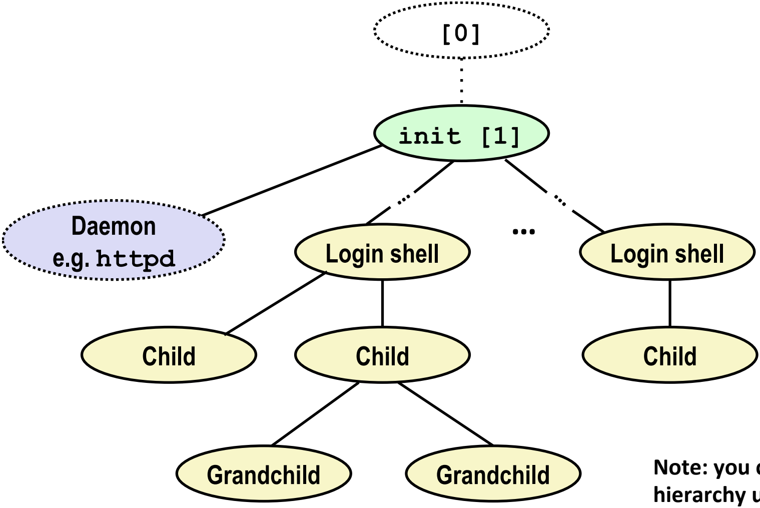
- Can be done in 10-20 lines of code, all in `eval()` function
- Correctness tests (`make test05` and `checktsh.py`) will come later

Today

- Shell lab

1. ECF, Signals and the command shell
   - Shells
   - Signals
     - Sending and receiving signals
     - Synchronization
     - Explicitly waiting for signals

Linux process hierarchy



Note: you can view the hierarchy using the Linux `pstree` command

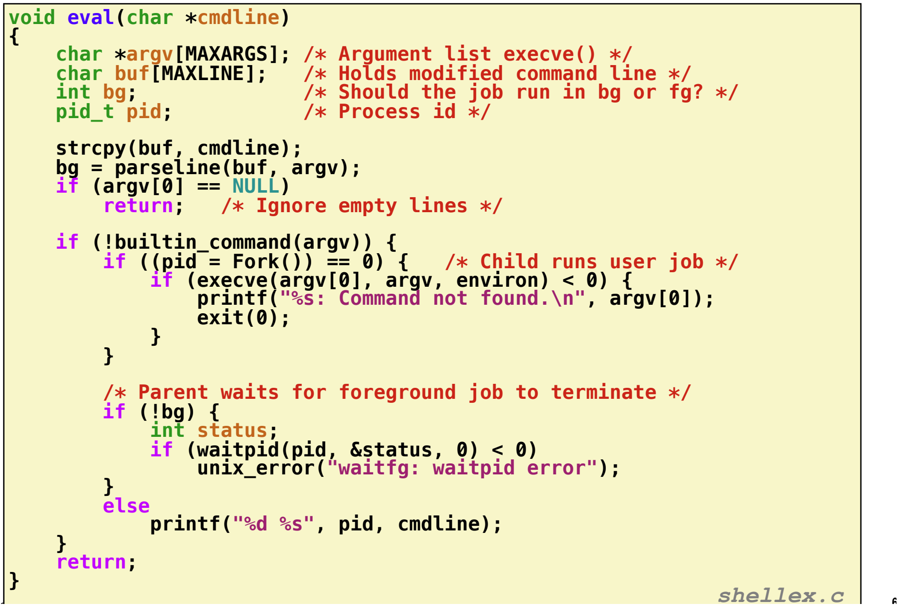
What is a shell?

- A shell is an application program that runs programs on behalf of the user.
- sh Orig. Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
- bash GNU "Bourne-Again" Shell (1989)
- tsh "tiny" shell (you, 2019)

```
int main()
{
    char cmdline[MAXLINE]; /* command line */

    while (1) {
        /* read */
        printf("> ");
        Fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);

        /* evaluate */
        eval(cmdline);
    }
}
                                        shellex.c
```

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;
}
```

shellex.c

Problem with simple shell example given?

- Example shell given: 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
- will create a memory leak that could eventually deplete system memory

Signals to the rescue!

- Solution: use the tools of exceptional control flow!
  - the OS kernel will interrupt regular processing to alert us when a background process completes
  - in Unix this alert mechanism is called a *signal*

Today

- Shell lab

1. ECF, Signals and the command shell
   - Shells
   - Signals
     - Sending and receiving signals
     - Synchronization
     - Explicitly waiting for signals

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) to a process
  - 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

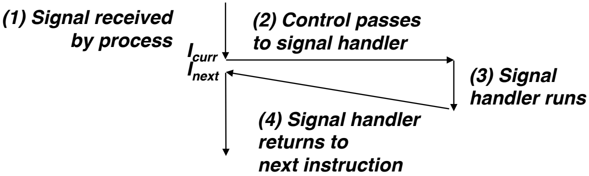
- **Kernel *sends* (delivers) a signal to a *destination process* by updating some state in the context of the destination process**

- **Kernel sends a signal for one of the following reasons:**
  - Kernel has detected a system event such as divide-by-zero (SIGFPE) or 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

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

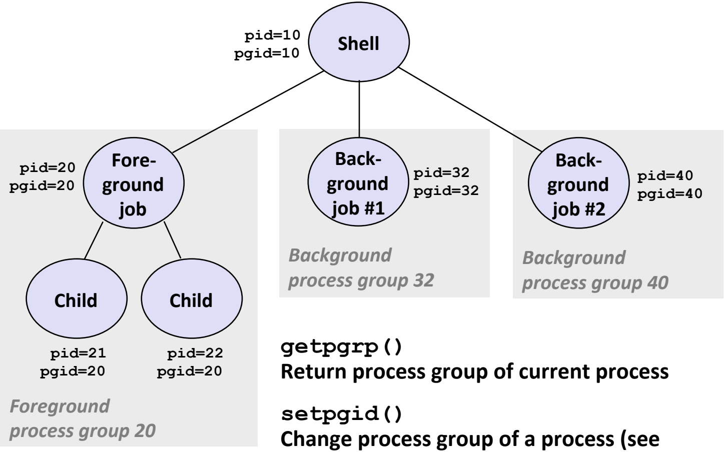
- **A signal is *pending* if sent but not yet received**
  - There can be at most one pending signal of any particular type
  - Important: Signals are not queued
    - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded
- **A process 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: 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)

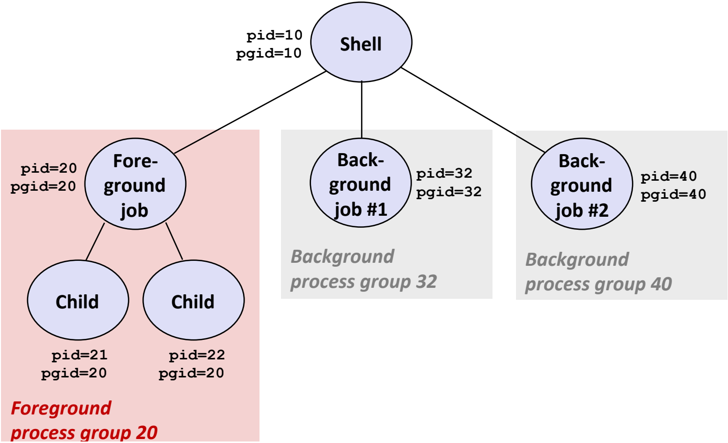
Sending signals with /bin/kill

- **/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) 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

Sending SIGINT and SIGTSTP

```
Parent: pid=58227 pgrp=58227
Child: pid=58232 pgrp=58227
^Z
[1]+  Stopped                ./forks 17
csadmin@systems-lab:~/2467$ ps w
   PID TTY      STAT   TIME COMMAND
 58227 pts/1    T      0:00 ./forks 17
 58232 pts/1    T      0:00 ./forks 17
 66587 pts/1    R+     0:00 ps w
 98531 pts/1    Ss     0:00 -bash
csadmin@systems-lab:~/2467$ fg %1
./forks 17
^C
csadmin@systems-lab:~/2467$ ps w
   PID TTY      STAT   TIME COMMAND
 96604 pts/1    R+     0:00 ps w
 98531 pts/1    Ss     0:00 -bash
```

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

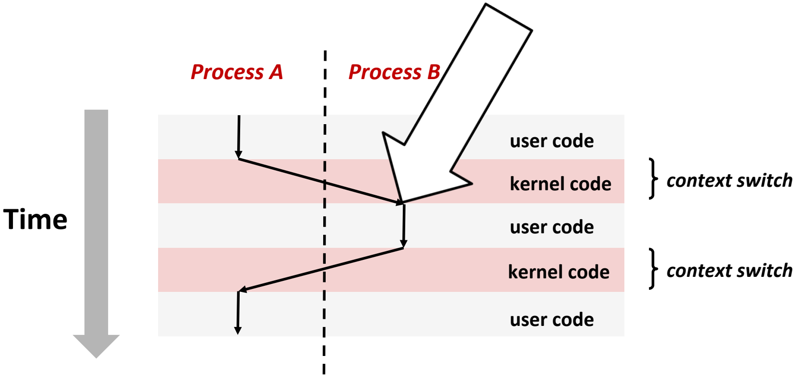
Sending signals with the `kill` function

```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);
    }
}
```

forks.c

Receiving Signals

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



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 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

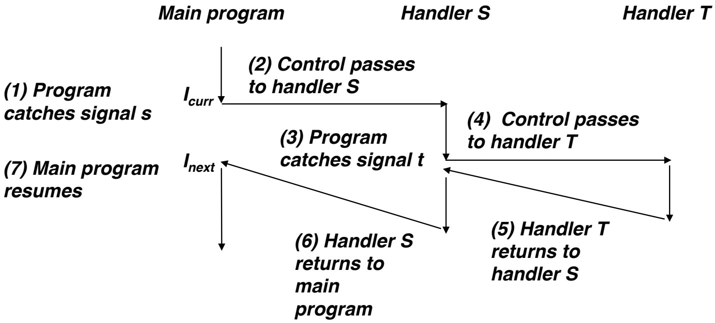
Installing signal handlers

- **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

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

- **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

Example using sigprocmask()

```
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);
```

Synchronizing flows to avoid races

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

```c
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 condition

```c
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.**

```c
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.

```c
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