code/ecf/shellex.c

```
1  /* parseline - Parse the command line and build the argv array */
2  int parseline(char *buf, char **argv)
3  {
4      char *delim;         /* Points to first space delimiter */
5      int argc;            /* Number of args */
6      int bg;              /* Background job? */
7
8      buf[strlen(buf)-1] = ' ';  /* Replace trailing '\n' with space */
9      while (*buf && (*buf == ' ')) /* Ignore leading spaces */
10         buf++;
11
12     /* Build the argv list */
13     argc = 0;
14     while ((delim = strchr(buf, ' '))) {
15         argv[argc++] = buf;
16         *delim = '\0';
17         buf = delim + 1;
18         while (*buf && (*buf == ' ')) /* Ignore spaces */
19                buf++;
20     }
21     argv[argc] = NULL;
22
23     if (argc == 0)  /* Ignore blank line */
24         return 1;
25
26     /* Should the job run in the background? */
27     if ((bg = (*argv[argc-1] == '&')) != 0)
28         argv[--argc] = NULL;
29
30     return bg;
31 }
```

code/ecf/shellex.c

Figure 8.25 `parseline` **parses a line of input for the shell.**

8.5 Signals

To this point in our study of exceptional control flow, we have seen how hardware and software cooperate to provide the fundamental low-level exception mechanism. We have also seen how the operating system uses exceptions to support a form of exceptional control flow known as the process context switch. In this section, we will study a higher-level software form of exceptional control flow, known as a Linux signal, that allows processes and the kernel to interrupt other processes.

Number	Name	Default action	Corresponding event
1	SIGHUP	Terminate	Terminal line hangup
2	SIGINT	Terminate	Interrupt from keyboard
3	SIGQUIT	Terminate	Quit from keyboard
4	SIGILL	Terminate	Illegal instruction
5	SIGTRAP	Terminate and dump core [a]	Trace trap
6	SIGABRT	Terminate and dump core [a]	Abort signal from `abort` function
7	SIGBUS	Terminate	Bus error
8	SIGFPE	Terminate and dump core [a]	Floating-point exception
9	SIGKILL	Terminate [b]	Kill program
10	SIGUSR1	Terminate	User-defined signal 1
11	SIGSEGV	Terminate and dump core [a]	Invalid memory reference (seg fault)
12	SIGUSR2	Terminate	User-defined signal 2
13	SIGPIPE	Terminate	Wrote to a pipe with no reader
14	SIGALRM	Terminate	Timer signal from `alarm` function
15	SIGTERM	Terminate	Software termination signal
16	SIGSTKFLT	Terminate	Stack fault on coprocessor
17	SIGCHLD	Ignore	A child process has stopped or terminated
18	SIGCONT	Ignore	Continue process if stopped
19	SIGSTOP	Stop until next SIGCONT [b]	Stop signal not from terminal
20	SIGTSTP	Stop until next SIGCONT	Stop signal from terminal
21	SIGTTIN	Stop until next SIGCONT	Background process read from terminal
22	SIGTTOU	Stop until next SIGCONT	Background process wrote to terminal
23	SIGURG	Ignore	Urgent condition on socket
24	SIGXCPU	Terminate	CPU time limit exceeded
25	SIGXFSZ	Terminate	File size limit exceeded
26	SIGVTALRM	Terminate	Virtual timer expired
27	SIGPROF	Terminate	Profiling timer expired
28	SIGWINCH	Ignore	Window size changed
29	SIGIO	Terminate	I/O now possible on a descriptor
30	SIGPWR	Terminate	Power failure

Figure 8.26 Linux signals. *Notes:* (a) Years ago, main memory was implemented with a technology known as *core memory*. "Dumping core" is a historical term that means writing an image of the code and data memory segments to disk. (b) This signal can be neither caught nor ignored. (Source: `man 7 signal`. Data from the Linux Foundation.)

A *signal* is a small message that notifies a process that an event of some type has occurred in the system. Figure 8.26 shows the 30 different types of signals that are supported on Linux systems.

Each signal type corresponds to some kind of system event. Low-level hardware exceptions are processed by the kernel's exception handlers and would not normally be visible to user processes. Signals provide a mechanism for exposing

the occurrence of such exceptions to user processes. For example, if a process attempts to divide by zero, then the kernel sends it a SIGFPE signal (number 8). If a process executes an illegal instruction, the kernel sends it a SIGILL signal (number 4). If a process makes an illegal memory reference, the kernel sends it a SIGSEGV signal (number 11). Other signals correspond to higher-level software events in the kernel or in other user processes. For example, if you type Ctrl+C (i.e., press the Ctrl key and the 'c' key at the same time) while a process is running in the foreground, then the kernel sends a SIGINT (number 2) to each process in the foreground process group. A process can forcibly terminate another process by sending it a SIGKILL signal (number 9). When a child process terminates or stops, the kernel sends a SIGCHLD signal (number 17) to the parent.

8.5.1 Signal Terminology

The transfer of a signal to a destination process occurs in two distinct steps:

Sending a signal. The kernel *sends* (*delivers*) a signal to a destination process by updating some state in the context of the destination process. The signal is delivered for one of two reasons: (1) The kernel has detected a system event such as a divide-by-zero error or the termination of a child process. (2) A process has invoked the `kill` function (discussed in the next section) to explicitly request the kernel to send a signal to the destination process. A process can send a signal to itself.

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. The process can either ignore the signal, terminate, or *catch* the signal by executing a user-level function called a *signal handler*. Figure 8.27 shows the basic idea of a handler catching a signal.

A signal that has been sent but not yet received is called a *pending signal*. At any point in time, there can be at most one pending signal of a particular type. If a process has a pending signal of type k, then any subsequent signals of type k sent to that process are *not* queued; they are simply discarded. A process can selectively *block* the receipt of certain signals. When a signal is blocked, it can be

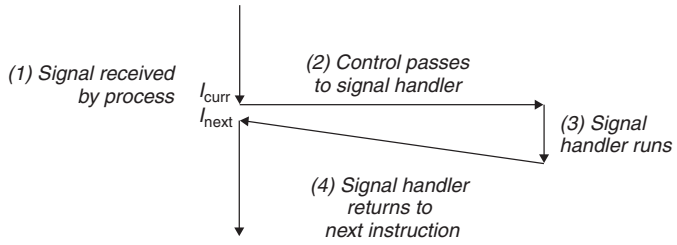

Figure 8.27
Signal handling. Receipt of a signal triggers a control transfer to a signal handler. After it finishes processing, the handler returns control to the interrupted program.

delivered, but the resulting pending signal will not be received until the process unblocks the signal.

A pending signal is received at most once. For each process, the kernel maintains the set of pending signals in the `pending` bit vector, and the set of blocked signals in the `blocked` bit vector.[1] The kernel sets bit k in `pending` whenever a signal of type k is delivered and clears bit k in `pending` whenever a signal of type k is received.

8.5.2 Sending Signals

Unix systems provide a number of mechanisms for sending signals to processes. All of the mechanisms rely on the notion of a *process group*.

Process Groups

Every process belongs to exactly one *process group*, which is identified by a positive integer *process group ID*. The `getpgrp` function returns the process group ID of the current process.

```
#include <unistd.h>

pid_t getpgrp(void);
```

Returns: process group ID of calling process

By default, a child process belongs to the same process group as its parent. A process can change the process group of itself or another process by using the `setpgid` function:

```
#include <unistd.h>

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

Returns: 0 on success, −1 on error

The `setpgid` function changes the process group of process `pid` to `pgid`. If `pid` is zero, the PID of the current process is used. If `pgid` is zero, the PID of the process specified by `pid` is used for the process group ID. For example, if process 15213 is the calling process, then

```
setpgid(0, 0);
```

creates a new process group whose process group ID is 15213, and adds process 15213 to this new group.

1. Also known as the *signal mask*.

Sending Signals with the /bin/kill Program

The /bin/kill program sends an arbitrary signal to another process. For example, the command

```
linux> /bin/kill -9 15213
```

sends signal 9 (SIGKILL) to process 15213. A negative PID causes the signal to be sent to every process in process group PID. For example, the command

```
linux> /bin/kill -9 -15213
```

sends a SIGKILL signal to every process in process group 15213. Note that we use the complete path /bin/kill here because some Unix shells have their own built-in kill command.

Sending Signals from the Keyboard

Unix shells use the abstraction of a *job* to represent the processes that are created as a result of evaluating a single command line. At any point in time, there is at most one foreground job and zero or more background jobs. For example, typing

```
linux> ls | sort
```

creates a foreground job consisting of two processes connected by a Unix pipe: one running the ls program, the other running the sort program. The shell creates a separate process group for each job. Typically, the process group ID is taken from one of the parent processes in the job. For example, Figure 8.28 shows a shell with one foreground job and two background jobs. The parent process in the foreground job has a PID of 20 and a process group ID of 20. The parent process has created two children, each of which are also members of process group 20.

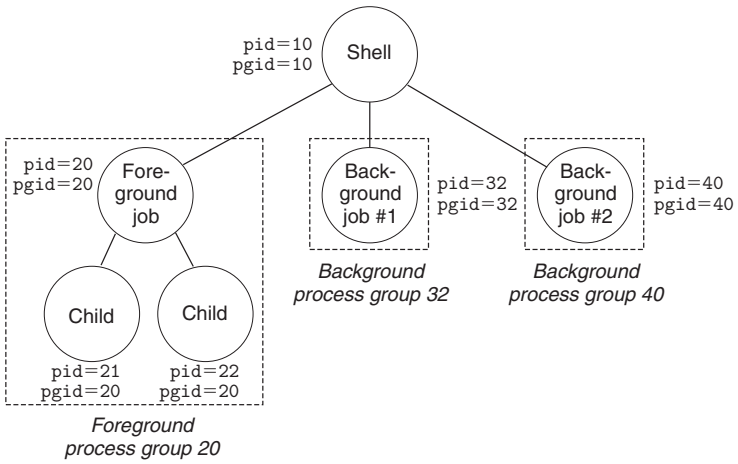

Figure 8.28
Foreground and background process groups.

Typing Ctrl+C at the keyboard causes the kernel to send a SIGINT signal to every process in the foreground process group. In the default case, the result is to terminate the foreground job. Similarly, typing Ctrl+Z causes the kernel to send a SIGTSTP signal to every process in the foreground process group. In the default case, the result is to stop (suspend) the foreground job.

Sending Signals with the `kill` Function

Processes send signals to other processes (including themselves) by calling the `kill` function.

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

Returns: 0 if OK, −1 on error

If `pid` is greater than zero, then the `kill` function sends signal number `sig` to process `pid`. If `pid` is equal to zero, then `kill` sends signal `sig` to every process in the process group of the calling process, including the calling process itself. If `pid` is less than zero, then `kill` sends signal `sig` to every process in process group |`pid`| (the absolute value of `pid`). Figure 8.29 shows an example of a parent that uses the `kill` function to send a SIGKILL signal to its child.

code/ecf/kill.c

```
1   #include "csapp.h"
2
3   int main()
4   {
5       pid_t pid;
6
7       /* Child sleeps until SIGKILL signal received, then dies */
8       if ((pid = Fork()) == 0) {
9           Pause();  /* Wait for a signal to arrive */
10          printf("control should never reach here!\n");
11          exit(0);
12      }
13
14      /* Parent sends a SIGKILL signal to a child */
15      Kill(pid, SIGKILL);
16      exit(0);
17  }
```

code/ecf/kill.c

Figure 8.29 **Using the `kill` function to send a signal to a child.**

Sending Signals with the `alarm` Function

A process can send SIGALRM signals to itself by calling the `alarm` function.

```
#include <unistd.h>

unsigned int alarm(unsigned int secs);
                Returns: remaining seconds of previous alarm, or 0 if no previous alarm
```

The `alarm` function arranges for the kernel to send a SIGALRM signal to the calling process in `secs` seconds. If `secs` is 0, then no new alarm is scheduled. In any event, the call to `alarm` cancels any pending alarms and returns the number of seconds remaining until any pending alarm was due to be delivered (had not this call to `alarm` canceled it), or 0 if there were no pending alarms.

8.5.3 Receiving Signals

When the kernel switches a process p from kernel mode to user mode (e.g., returning from a system call or completing a context switch), it checks the set of unblocked pending signals (`pending & ~blocked`) for p. If this set is empty (the usual case), then the kernel passes control to the next instruction (I_{next}) in the logical control flow of p. However, if the set is nonempty, then the kernel chooses some signal k in the set (typically the smallest k) and forces p to *receive* signal k. The receipt of the signal triggers some *action* by the process. Once the process completes the action, then control passes back to the next instruction (I_{next}) in the logical control flow of p. Each signal type has a predefined *default action*, which is one of the following:

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

Figure 8.26 shows the default actions associated with each type of signal. For example, the default action for the receipt of a SIGKILL is to terminate the receiving process. On the other hand, the default action for the receipt of a SIGCHLD is to ignore the signal. A process can modify the default action associated with a signal by using the `signal` function. The only exceptions are SIGSTOP and SIGKILL, whose default actions cannot be changed.

```
#include <signal.h>
typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
     Returns: pointer to previous handler if OK, SIG_ERR on error (does not set errno)
```

The `signal` function can change the action associated with a signal `signum` in one of three ways:

- If `handler` is SIG_IGN, then signals of type `signum` are ignored.
- If `handler` is SIG_DFL, then the action for signals of type `signum` reverts to the default action.
- Otherwise, `handler` is the address of a user-defined function, called a *signal handler*, that will be called whenever the process receives a signal of type `signum`. Changing the default action by passing the address of a handler to the `signal` function is known as *installing the handler*. The invocation of the handler is called *catching the signal*. The execution of the handler is referred to as *handling the signal*.

When a process catches a signal of type k, the handler installed for signal k is invoked with a single integer argument set to k. This argument allows the same handler function to catch different types of signals.

When the handler executes its `return` statement, control (usually) passes back to the instruction in the control flow where the process was interrupted by the receipt of the signal. We say "usually" because in some systems, interrupted system calls return immediately with an error.

Figure 8.30 shows a program that catches the SIGINT signal that is sent whenever the user types Ctrl+C at the keyboard. The default action for SIGINT

code/ecf/sigint.c

```
1   #include "csapp.h"
2
3   void sigint_handler(int sig) /* SIGINT handler */
4   {
5       printf("Caught SIGINT!\n");
6       exit(0);
7   }
8
9   int main()
10  {
11      /* Install the SIGINT handler */
12      if (signal(SIGINT, sigint_handler) == SIG_ERR)
13          unix_error("signal error");
14
15      pause(); /* Wait for the receipt of a signal */
16
17      return 0;
18  }
```

code/ecf/sigint.c

Figure 8.30 A program that uses a signal handler to catch a SIGINT signal.

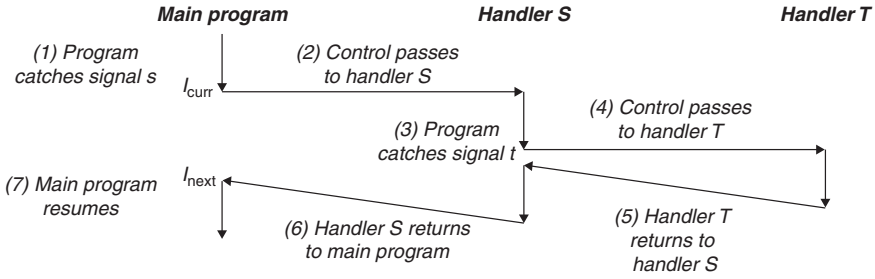

Figure 8.31 **Handlers can be interrupted by other handlers.**

is to immediately terminate the process. In this example, we modify the default behavior to catch the signal, print a message, and then terminate the process.

Signal handlers can be interrupted by other handlers, as shown in Figure 8.31. In this example, the main program catches signal s, which interrupts the main program and transfers control to handler S. While S is running, the program catches signal $t \neq s$, which interrupts S and transfers control to handler T. When T returns, S resumes where it was interrupted. Eventually, S returns, transferring control back to the main program, which resumes where it left off.

Practice Problem 8.7 (solution page 834)

Write a program called `snooze` that takes a single command-line argument, calls the `snooze` function from Problem 8.5 with this argument, and then terminates. Write your program so that the user can interrupt the `snooze` function by typing Ctrl+C at the keyboard. For example:

```
linux> ./snooze 5
CTRL+C                          User hits Crtl+C after 3 seconds
Slept for 3 of 5 secs.
linux>
```

8.5.4 Blocking and Unblocking Signals

Linux provides implicit and explicit mechanisms for blocking signals:

Implicit blocking mechanism. By default, the kernel blocks any pending signals of the type currently being processed by a handler. For example, in Figure 8.31, suppose the program has caught signal s and is currently running handler S. If another signal s is sent to the process, then s will become pending but will not be received until after handler S returns.

Explicit blocking mechanism. Applications can explicitly block and unblock selected signals using the `sigprocmask` function and its helpers.

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
                                                    Returns: 0 if OK, −1 on error

int sigismember(const sigset_t *set, int signum);
                                          Returns: 1 if member, 0 if not, −1 on error
```

The `sigprocmask` function changes the set of currently blocked signals (the `blocked` bit vector described in Section 8.5.1). The specific behavior depends on the value of `how`:

SIG_BLOCK. Add the signals in `set` to `blocked` (`blocked = blocked | set`).

SIG_UNBLOCK. Remove the signals in `set` from `blocked` (`blocked = blocked & ~set`).

SIG_SETMASK. `blocked = set`.

If `oldset` is non-NULL, the previous value of the `blocked` bit vector is stored in `oldset`.

Signal sets such as `set` are manipulated using the following functions: The `sigemptyset` initializes `set` to the empty set. The `sigfillset` function adds every signal to `set`. The `sigaddset` function adds `signum` to `set`, `sigdelset` deletes `signum` from `set`, and `sigismember` returns 1 if `signum` is a member of `set`, and 0 if not.

For example, Figure 8.32 shows how you would use `sigprocmask` to temporarily block the receipt of SIGINT signals.

```
1       sigset_t mask, prev_mask;
2
3       Sigemptyset(&mask);
4       Sigaddset(&mask, SIGINT);
5
6       /* Block SIGINT and save previous blocked set */
7       Sigprocmask(SIG_BLOCK, &mask, &prev_mask);
8       .
        .   // Code region that will not be interrupted by SIGINT
        .
9       /* Restore previous blocked set, unblocking SIGINT */
10      Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
11
```

Figure 8.32 **Temporarily blocking a signal from being received.**

8.5.5 Writing Signal Handlers

Signal handling is one of the thornier aspects of Linux system-level programming. Handlers have several attributes that make them difficult to reason about: (1) Handlers run concurrently with the main program and share the same global variables, and thus can interfere with the main program and with other handlers. (2) The rules for how and when signals are received is often counterintuitive. (3) Different systems can have different signal-handling semantics.

In this section, we address these issues and give you some basic guidelines for writing safe, correct, and portable signal handlers.

Safe Signal Handling

Signal handlers are tricky because they can run concurrently with the main program and with each other, as we saw in Figure 8.31. If a handler and the main program access the same global data structure concurrently, then the results can be unpredictable and often fatal.

We will explore concurrent programming in detail in Chapter 12. Our aim here is to give you some conservative guidelines for writing handlers that are safe to run concurrently. If you ignore these guidelines, you run the risk of introducing subtle concurrency errors. With such errors, your program works correctly most of the time. However, when it fails, it fails in unpredictable and unrepeatable ways that are horrendously difficult to debug. Forewarned is forearmed!

G0. Keep handlers as simple as possible. The best way to avoid trouble is to keep your handlers as small and simple as possible. For example, the handler might simply set a global flag and return immediately; all processing associated with the receipt of the signal is performed by the main program, which periodically checks (and resets) the flag.

G1. Call only async-signal-safe functions in your handlers. A function that is *async-signal-safe*, or simply *safe*, has the property that it can be safely called from a signal handler, either because it is *reentrant* (e.g., accesses only local variables; see Section 12.7.2), or because it cannot be interrupted by a signal handler. Figure 8.33 lists the system-level functions that Linux guarantees to be safe. Notice that many popular functions, such as `printf`, `sprintf`, `malloc`, and `exit`, are *not* on this list.

The only safe way to generate output from a signal handler is to use the `write` function (see Section 10.1). In particular, calling `printf` or `sprintf` is unsafe. To work around this unfortunate restriction, we have developed some safe functions, called the Sio (Safe I/O) package, that you can use to print simple messages from signal handlers.

_Exit	fexecve	poll	sigqueue
_exit	fork	posix_trace_event	sigset
abort	fstat	pselect	sigsuspend
accept	fstatat	raise	sleep
access	fsync	read	sockatmark
aio_error	ftruncate	readlink	socket
aio_return	futimens	readlinkat	socketpair
aio_suspend	getegid	recv	stat
alarm	geteuid	recvfrom	symlink
bind	getgid	recvmsg	symlinkat
cfgetispeed	getgroups	rename	tcdrain
cfgetospeed	getpeername	renameat	tcflow
cfsetispeed	getpgrp	rmdir	tcflush
cfsetospeed	getpid	select	tcgetattr
chdir	getppid	sem_post	tcgetpgrp
chmod	getsockname	send	tcsendbreak
chown	getsockopt	sendmsg	tcsetattr
clock_gettime	getuid	sendto	tcsetpgrp
close	kill	setgid	time
connect	link	setpgid	timer_getoverrun
creat	linkat	setsid	timer_gettime
dup	listen	setsockopt	timer_settime
dup2	lseek	setuid	times
execl	lstat	shutdown	umask
execle	mkdir	sigaction	uname
execv	mkdirat	sigaddset	unlink
execve	mkfifo	sigdelset	unlinkat
faccessat	mkfifoat	sigemptyset	utime
fchmod	mknod	sigfillset	utimensat
fchmodat	mknodat	sigismember	utimes
fchown	open	signal	wait
fchownat	openat	sigpause	waitpid
fcntl	pause	sigpending	write
fdatasync	pipe	sigprocmask	

Figure 8.33 Async-signal-safe functions. (Source: `man 7 signal`. Data from the Linux Foundation.)

```
#include "csapp.h"

ssize_t sio_putl(long v);
ssize_t sio_puts(char s[]);
                                Returns: number of bytes transferred if OK, −1 on error

void sio_error(char s[]);
                                                        Returns: nothing
```

The `sio_putl` and `sio_puts` functions emit a `long` and a string, respectively, to standard output. The `sio_error` function prints an error message and terminates.

Figure 8.34 shows the implementation of the Sio package, which uses two private reentrant functions from `csapp.c`. The `sio_strlen` function in line 3 returns the length of string `s`. The `sio_ltoa` function in line 10, which is based on the `itoa` function from [61], converts `v` to its base `b` string representation in `s`. The `_exit` function in line 17 is an async-signal-safe variant of `exit`.

Figure 8.35 shows a safe version of the SIGINT handler from Figure 8.30.

G2. Save and restore `errno`. Many of the Linux async-signal-safe functions set `errno` when they return with an error. Calling such functions inside a handler might interfere with other parts of the program that rely on `errno`.

code/src/csapp.c

```
1   ssize_t sio_puts(char s[]) /* Put string */
2   {
3       return write(STDOUT_FILENO, s, sio_strlen(s));
4   }
5
6   ssize_t sio_putl(long v) /* Put long */
7   {
8       char s[128];
9
10      sio_ltoa(v, s, 10); /* Based on K&R itoa() */
11      return sio_puts(s);
12  }
13
14  void sio_error(char s[]) /* Put error message and exit */
15  {
16      sio_puts(s);
17      _exit(1);
18  }
```

code/src/csapp.c

Figure 8.34 The Sio (Safe I/O) package for signal handlers.

code/ecf/sigintsafe.c

```
1  #include "csapp.h"
2
3  void sigint_handler(int sig) /* Safe SIGINT handler */
4  {
5      Sio_puts("Caught SIGINT!\n"); /* Safe output */
6      _exit(0);                     /* Safe exit */
7  }
```

code/ecf/sigintsafe.c

Figure 8.35 A safe version of the SIGINT handler from Figure 8.30.

The workaround is to save `errno` to a local variable on entry to the handler and restore it before the handler returns. Note that this is only necessary if the handler returns. It is not necessary if the handler terminates the process by calling `_exit`.

G3. Protect accesses to shared global data structures by blocking all signals. If a handler shares a global data structure with the main program or with other handlers, then your handlers and main program should temporarily block all signals while accessing (reading or writing) that data structure. The reason for this rule is that accessing a data structure d from the main program typically requires a sequence of instructions. If this instruction sequence is interrupted by a handler that accesses d, then the handler might find d in an inconsistent state, with unpredictable results. Temporarily blocking signals while you access d guarantees that a handler will not interrupt the instruction sequence.

G4. Declare global variables with `volatile`. Consider a handler and `main` routine that share a global variable g. The handler updates g, and `main` periodically reads g. To an optimizing compiler, it would appear that the value of g never changes in `main`, and thus it would be safe to use a copy of g that is cached in a register to satisfy every reference to g. In this case, the `main` function would never see the updated values from the handler.

You can tell the compiler not to cache a variable by declaring it with the `volatile` type qualifier. For example:

```
volatile int g;
```

The `volatile` qualifier forces the compiler to read the value of `g` from memory each time it is referenced in the code. In general, as with any shared data structure, each access to a global variable should be protected by temporarily blocking signals.

G5. Declare flags with `sig_atomic_t`. In one common handler design, the handler records the receipt of the signal by writing to a global *flag*. The main program periodically reads the flag, responds to the signal, and

clears the flag. For flags that are shared in this way, C provides an integer data type, `sig_atomic_t`, for which reads and writes are guaranteed to be *atomic* (uninterruptible) because they can be implemented with a single instruction:

```
volatile sig_atomic_t flag;
```

Since they can't be interrupted, you can safely read from and write to `sig_atomic_t` variables without temporarily blocking signals. Note that the guarantee of atomicity only applies to individual reads and writes. It does not apply to updates such as `flag++` or `flag = flag + 10`, which might require multiple instructions.

Keep in mind that the guidelines we have presented are conservative, in the sense that they are not always strictly necessary. For example, if you know that a handler can never modify `errno`, then you don't need to save and restore `errno`. Or if you can prove that no instance of `printf` can ever be interrupted by a handler, then it is safe to call `printf` from the handler. The same holds for accesses to shared global data structures. However, it is very difficult to prove such assertions in general. So we recommend that you take the conservative approach and follow the guidelines by keeping your handlers as simple as possible, calling safe functions, saving and restoring `errno`, protecting accesses to shared data structures, and using `volatile` and `sig_atomic_t`.

Correct Signal Handling

One of the nonintuitive aspects of signals is that pending signals are not queued. Because the `pending` bit vector contains exactly one bit for each type of signal, there can be at most one pending signal of any particular type. Thus, if two signals of type k are sent to a destination process while signal k is blocked because the destination process is currently executing a handler for signal k, then the second signal is simply discarded; it is not queued. The key idea is that the existence of a pending signal merely indicates that *at least* one signal has arrived.

To see how this affects correctness, let's look at a simple application that is similar in nature to real programs such as shells and Web servers. The basic structure is that a parent process creates some children that run independently for a while and then terminate. The parent must reap the children to avoid leaving zombies in the system. But we also want the parent to be free to do other work while the children are running. So we decide to reap the children with a SIGCHLD handler, instead of explicitly waiting for the children to terminate. (Recall that the kernel sends a SIGCHLD signal to the parent whenever one of its children terminates or stops.)

Figure 8.36 shows our first attempt. The parent installs a SIGCHLD handler and then creates three children. In the meantime, the parent waits for a line of input from the terminal and then processes it. This processing is modeled by an infinite loop. When each child terminates, the kernel notifies the parent by sending it a SIGCHLD signal. The parent catches the SIGCHLD, reaps one child,

code/ecf/signal1.c

```
1  /* WARNING: This code is buggy! */
2
3  void handler1(int sig)
4  {
5      int olderrno = errno;
6
7      if ((waitpid(-1, NULL, 0)) < 0)
8          sio_error("waitpid error");
9      Sio_puts("Handler reaped child\n");
10     Sleep(1);
11     errno = olderrno;
12 }
13
14 int main()
15 {
16     int i, n;
17     char buf[MAXBUF];
18
19     if (signal(SIGCHLD, handler1) == SIG_ERR)
20         unix_error("signal error");
21
22     /* Parent creates children */
23     for (i = 0; i < 3; i++) {
24         if (Fork() == 0) {
25             printf("Hello from child %d\n", (int)getpid());
26             exit(0);
27         }
28     }
29
30     /* Parent waits for terminal input and then processes it */
31     if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
32         unix_error("read");
33
34     printf("Parent processing input\n");
35     while (1)
36         ;
37
38     exit(0);
39 }
```

code/ecf/signal1.c

Figure 8.36 `signal1`. This program is flawed because it assumes that signals are queued.

does some additional cleanup work (modeled by the `sleep` statement), and then returns.

The `signal1` program in Figure 8.36 seems fairly straightforward. When we run it on our Linux system, however, we get the following output:

```
linux> ./signal1
Hello from child 14073
Hello from child 14074
Hello from child 14075
Handler reaped child
Handler reaped child
CR
Parent processing input
```

From the output, we note that although three SIGCHLD signals were sent to the parent, only two of these signals were received, and thus the parent only reaped two children. If we suspend the parent process, we see that, indeed, child process 14075 was never reaped and remains a zombie (indicated by the string `<defunct>` in the output of the `ps` command):

```
Ctrl+Z
Suspended
linux> ps t
  PID TTY      STAT   TIME COMMAND
  .
  .
  .
14072 pts/3    T      0:02 ./signal1
14075 pts/3    Z      0:00 [signal1] <defunct>
14076 pts/3    R+     0:00 ps t
```

What went wrong? The problem is that our code failed to account for the fact that signals are not queued. Here's what happened: The first signal is received and caught by the parent. While the handler is still processing the first signal, the second signal is delivered and added to the set of pending signals. However, since SIGCHLD signals are blocked by the SIGCHLD handler, the second signal is not received. Shortly thereafter, while the handler is still processing the first signal, the third signal arrives. Since there is already a pending SIGCHLD, this third SIGCHLD signal is discarded. Sometime later, after the handler has returned, the kernel notices that there is a pending SIGCHLD signal and forces the parent to receive the signal. The parent catches the signal and executes the handler a second time. After the handler finishes processing the second signal, there are no more pending SIGCHLD signals, and there never will be, because all knowledge of the third SIGCHLD has been lost. *The crucial lesson is that signals cannot be used to count the occurrence of events in other processes.*

To fix the problem, we must recall that the existence of a pending signal only implies that at least one signal has been delivered since the last time the process received a signal of that type. So we must modify the SIGCHLD handler to reap

code/ecf/signal2.c

```
1   void handler2(int sig)
2   {
3       int olderrno = errno;
4
5       while (waitpid(-1, NULL, 0) > 0) {
6           Sio_puts("Handler reaped child\n");
7       }
8       if (errno != ECHILD)
9           Sio_error("waitpid error");
10      Sleep(1);
11      errno = olderrno;
12  }
```

code/ecf/signal2.c

Figure 8.37 `signal2`. An improved version of Figure 8.36 that correctly accounts for the fact that signals are not queued.

as many zombie children as possible each time it is invoked. Figure 8.37 shows the modified SIGCHLD handler.

When we run `signal2` on our Linux system, it now correctly reaps all of the zombie children:

```
linux> ./signal2
Hello from child 15237
Hello from child 15238
Hello from child 15239
Handler reaped child
Handler reaped child
Handler reaped child
CR
Parent processing input
```

Practice Problem 8.8 (solution page 835)

What is the output of the following program?

code/ecf/signalprob0.c

```
1   volatile long counter = 2;
2
3   void handler1(int sig)
4   {
5       sigset_t mask, prev_mask;
6
7       Sigfillset(&mask);
8       Sigprocmask(SIG_BLOCK, &mask, &prev_mask);  /* Block sigs */
```

```
9       Sio_putl(--counter);
10      Sigprocmask(SIG_SETMASK, &prev_mask, NULL); /* Restore sigs */
11
12      _exit(0);
13  }
14
15  int main()
16  {
17      pid_t pid;
18      sigset_t mask, prev_mask;
19
20      printf("%ld", counter);
21      fflush(stdout);
22
23      signal(SIGUSR1, handler1);
24      if ((pid = Fork()) == 0) {
25          while(1) {};
26      }
27      Kill(pid, SIGUSR1);
28      Waitpid(-1, NULL, 0);
29
30      Sigfillset(&mask);
31      Sigprocmask(SIG_BLOCK, &mask, &prev_mask);  /* Block sigs */
32      printf("%ld", ++counter);
33      Sigprocmask(SIG_SETMASK, &prev_mask, NULL); /* Restore sigs */
34
35      exit(0);
36  }
```

code/ecf/signalprob0.c

Portable Signal Handling

Another ugly aspect of Unix signal handling is that different systems have different signal-handling semantics. For example:

- *The semantics of the* `signal` *function varies.* Some older Unix systems restore the action for signal k to its default after signal k has been caught by a handler. On these systems, the handler must explicitly reinstall itself, by calling `signal`, each time it runs.
- *System calls can be interrupted.* System calls such as `read`, `wait`, and `accept` that can potentially block the process for a long period of time are called *slow system calls*. On some older versions of Unix, slow system calls that are interrupted when a handler catches a signal do not resume when the signal handler returns but instead return immediately to the user with an error condition and `errno` set to EINTR. On these systems, programmers must include code that manually restarts interrupted system calls.

code/src/csapp.c

```
1   handler_t *Signal(int signum, handler_t *handler)
2   {
3       struct sigaction action, old_action;
4
5       action.sa_handler = handler;
6       sigemptyset(&action.sa_mask); /* Block sigs of type being handled */
7       action.sa_flags = SA_RESTART; /* Restart syscalls if possible */
8
9       if (sigaction(signum, &action, &old_action) < 0)
10          unix_error("Signal error");
11      return (old_action.sa_handler);
12  }
```

code/src/csapp.c

Figure 8.38 `Signal`. A wrapper for `sigaction` that provides portable signal handling on Posix-compliant systems.

To deal with these issues, the Posix standard defines the `sigaction` function, which allows users to clearly specify the signal-handling semantics they want when they install a handler.

```
#include <signal.h>

int sigaction(int signum,  struct sigaction *act,
              struct sigaction *oldact);
                                          Returns: 0 if OK, −1 on error
```

The `sigaction` function is unwieldy because it requires the user to set the entries of a complicated structure. A cleaner approach, originally proposed by W. Richard Stevens [110], is to define a wrapper function, called `Signal`, that calls `sigaction` for us. Figure 8.38 shows the definition of `Signal`, which is invoked in the same way as the `signal` function.

The `Signal` wrapper installs a signal handler with the following signal-handling semantics:

- Only signals of the type currently being processed by the handler are blocked.
- As with all signal implementations, signals are not queued.
- Interrupted system calls are automatically restarted whenever possible.
- Once the signal handler is installed, it remains installed until `Signal` is called with a `handler` argument of either SIG_IGN or SIG_DFL.

We will use the `Signal` wrapper in all of our code.

8.5.6 Synchronizing Flows to Avoid Nasty Concurrency Bugs

The problem of how to program concurrent flows that read and write the same storage locations has challenged generations of computer scientists. In general, the number of potential interleavings of the flows is exponential in the number of instructions. Some of those interleavings will produce correct answers, and others will not. The fundamental problem is to somehow *synchronize* the concurrent flows so as to allow the largest set of feasible interleavings such that each of the feasible interleavings produces a correct answer.

Concurrent programming is a deep and important problem that we will discuss in more detail in Chapter 12. However, we can use what you've learned about exceptional control flow in this chapter to give you a sense of the interesting intellectual challenges associated with concurrency. For example, consider the program in Figure 8.39, which captures the structure of a typical Unix shell. The parent keeps track of its current children using entries in a global job list, with one entry per job. The `addjob` and `deletejob` functions add and remove entries from the job list.

After the parent creates a new child process, it adds the child to the job list. When the parent reaps a terminated (zombie) child in the SIGCHLD signal handler, it deletes the child from the job list.

At first glance, this code appears to be correct. Unfortunately, the following sequence of events is possible:

1. The parent executes the `fork` function and the kernel schedules the newly created child to run instead of the parent.
2. Before the parent is able to run again, the child terminates and becomes a zombie, causing the kernel to deliver a SIGCHLD signal to the parent.
3. Later, when the parent becomes runnable again but before it is executed, the kernel notices the pending SIGCHLD and causes it to be received by running the signal handler in the parent.
4. The signal handler reaps the terminated child and calls `deletejob`, which does nothing because the parent has not added the child to the list yet.
5. After the handler completes, the kernel then runs the parent, which returns from `fork` and incorrectly adds the (nonexistent) child to the job list by calling `addjob`.

Thus, for some interleavings of the parent's main routine and signal-handling flows, it is possible for `deletejob` to be called before `addjob`. This results in an incorrect entry on the job list, for a job that no longer exists and that will never be removed. On the other hand, there are also interleavings where events occur in the correct order. For example, if the kernel happens to schedule the parent to run when the `fork` call returns instead of the child, then the parent will correctly add the child to the job list before the child terminates and the signal handler removes the job from the list.

This is an example of a classic synchronization error known as a *race*. In this case, the race is between the call to `addjob` in the main routine and the call to

code/ecf/procmask1.c

```
1   /* WARNING: This code is buggy! */
2   void handler(int sig)
3   {
4       int olderrno = errno;
5       sigset_t mask_all, prev_all;
6       pid_t pid;
7
8       Sigfillset(&mask_all);
9       while ((pid = waitpid(-1, NULL, 0)) > 0) { /* Reap a zombie child */
10          Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
11          deletejob(pid); /* Delete the child from the job list */
12          Sigprocmask(SIG_SETMASK, &prev_all, NULL);
13      }
14      if (errno != ECHILD)
15          Sio_error("waitpid error");
16      errno = olderrno;
17  }
18
19  int main(int argc, char **argv)
20  {
21      int pid;
22      sigset_t mask_all, prev_all;
23
24      Sigfillset(&mask_all);
25      Signal(SIGCHLD, handler);
26      initjobs(); /* Initialize the job list */
27
28      while (1) {
29          if ((pid = Fork()) == 0) { /* Child process */
30              Execve("/bin/date", argv, NULL);
31          }
32          Sigprocmask(SIG_BLOCK, &mask_all, &prev_all); /* Parent process */
33          addjob(pid);  /* Add the child to the job list */
34          Sigprocmask(SIG_SETMASK, &prev_all, NULL);
35      }
36      exit(0);
37  }
```

code/ecf/procmask1.c

Figure 8.39 A shell program with a subtle synchronization error. If the child terminates before the parent is able to run, then `addjob` and `deletejob` will be called in the wrong order.

deletejob in the handler. If addjob wins the race, then the answer is correct. If not, the answer is incorrect. Such errors are enormously difficult to debug because it is often impossible to test every interleaving. You might run the code a billion times without a problem, but then the next test results in an interleaving that triggers the race.

Figure 8.40 shows one way to eliminate the race in Figure 8.39. By blocking SIGCHLD signals before the call to fork and then unblocking them only after we have called addjob, we guarantee that the child will be reaped *after* it is added to the job list. Notice that children inherit the blocked set of their parents, so we must be careful to unblock the SIGCHLD signal in the child before calling execve.

8.5.7 Explicitly Waiting for Signals

Sometimes a main program needs to explicitly wait for a certain signal handler to run. For example, when a Linux shell creates a foreground job, it must wait for the job to terminate and be reaped by the SIGCHLD handler before accepting the next user command.

Figure 8.41 shows the basic idea. The parent installs handlers for SIGINT and SIGCHLD and then enters an infinite loop. It blocks SIGCHLD to avoid the race between parent and child that we discussed in Section 8.5.6. After creating the child, it resets pid to zero, unblocks SIGCHLD, and then waits in a spin loop for pid to become nonzero. After the child terminates, the handler reaps it and assigns its nonzero PID to the global pid variable. This terminates the spin loop, and the parent continues with additional work before starting the next iteration.

While this code is correct, the spin loop is wasteful of processor resources. We might be tempted to fix this by inserting a pause in the body of the spin loop:

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

Notice that we still need a loop because pause might be interrupted by the receipt of one or more SIGINT signals. However, this code has a serious race condition: if the SIGCHLD is received after the while test but before the pause, the pause will sleep forever.

Another option is to replace the pause with sleep:

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

While correct, this code is too slow. If the signal is received after the while and before the sleep, the program must wait a (relatively) long time before it can check the loop termination condition again. Using a higher-resolution sleep function such as nanosleep isn't acceptable, either, because there is no good rule for determining the sleep interval. Make it too small and the loop is too wasteful. Make it too high and the program is too slow.

code/ecf/procmask2.c

```
1   void handler(int sig)
2   {
3       int olderrno = errno;
4       sigset_t mask_all, prev_all;
5       pid_t pid;
6
7       Sigfillset(&mask_all);
8       while ((pid = waitpid(-1, NULL, 0)) > 0) { /* Reap a zombie child */
9           Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
10          deletejob(pid); /* Delete the child from the job list */
11          Sigprocmask(SIG_SETMASK, &prev_all, NULL);
12      }
13      if (errno != ECHILD)
14          Sio_error("waitpid error");
15      errno = olderrno;
16  }
17
18  int main(int argc, char **argv)
19  {
20      int pid;
21      sigset_t mask_all, mask_one, prev_one;
22
23      Sigfillset(&mask_all);
24      Sigemptyset(&mask_one);
25      Sigaddset(&mask_one, SIGCHLD);
26      Signal(SIGCHLD, handler);
27      initjobs(); /* Initialize the job list */
28
29      while (1) {
30          Sigprocmask(SIG_BLOCK, &mask_one, &prev_one); /* Block SIGCHLD */
31          if ((pid = Fork()) == 0) { /* Child process */
32              Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
33              Execve("/bin/date", argv, NULL);
34          }
35          Sigprocmask(SIG_BLOCK, &mask_all, NULL); /* Parent process */
36          addjob(pid);  /* Add the child to the job list */
37          Sigprocmask(SIG_SETMASK, &prev_one, NULL);  /* Unblock SIGCHLD */
38      }
39      exit(0);
40  }
```

code/ecf/procmask2.c

Figure 8.40 Using `sigprocmask` to synchronize processes. In this example, the parent ensures that `addjob` executes before the corresponding `deletejob`.

code/ecf/waitforsignal.c

```
1  #include "csapp.h"
2
3  volatile sig_atomic_t pid;
4
5  void sigchld_handler(int s)
6  {
7      int olderrno = errno;
8      pid = waitpid(-1, NULL, 0);
9      errno = olderrno;
10 }
11
12 void sigint_handler(int s)
13 {
14 }
15
16 int main(int argc, char **argv)
17 {
18     sigset_t mask, prev;
19
20     Signal(SIGCHLD, sigchld_handler);
21     Signal(SIGINT, sigint_handler);
22     Sigemptyset(&mask);
23     Sigaddset(&mask, SIGCHLD);
24
25     while (1) {
26         Sigprocmask(SIG_BLOCK, &mask, &prev); /* Block SIGCHLD */
27         if (Fork() == 0) /* Child */
28             exit(0);
29
30         /* Parent */
31         pid = 0;
32         Sigprocmask(SIG_SETMASK, &prev, NULL); /* Unblock SIGCHLD */
33
34         /* Wait for SIGCHLD to be received (wasteful) */
35         while (!pid)
36             ;
37
38         /* Do some work after receiving SIGCHLD */
39         printf(".");
40     }
41     exit(0);
42 }
```

code/ecf/waitforsignal.c

Figure 8.41 Waiting for a signal with a spin loop. This code is correct, but the spin loop is wasteful.

The proper solution is to use `sigsuspend`.

```
#include <signal.h>

int sigsuspend(const sigset_t *mask);
                                                    Returns: −1
```

The `sigsuspend` function temporarily replaces the current blocked set with `mask` and then suspends the process until the receipt of a signal whose action is either to run a handler or to terminate the process. If the action is to terminate, then the process terminates without returning from `sigsuspend`. If the action is to run a handler, then `sigsuspend` returns after the handler returns, restoring the blocked set to its state when `sigsuspend` was called.

The `sigsuspend` function is equivalent to an *atomic* (uninterruptible) version of the following:

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

The atomic property guarantees that the calls to `sigprocmask` (line 1) and `pause` (line 2) occur together, without being interrupted. This eliminates the potential race where a signal is received after the call to `sigprocmask` and before the call to `pause`.

Figure 8.42 shows how we would use `sigsuspend` to replace the spin loop in Figure 8.41. Before each call to `sigsuspend`, SIGCHLD is blocked. The `sigsuspend` temporarily unblocks SIGCHLD, and then sleeps until the parent catches a signal. Before returning, it restores the original blocked set, which blocks SIGCHLD again. If the parent caught a SIGINT, then the loop test succeeds and the next iteration calls `sigsuspend` again. If the parent caught a SIGCHLD, then the loop test fails and we exit the loop. At this point, SIGCHLD is blocked, and so we can optionally unblock SIGCHLD. This might be useful in a real shell with background jobs that need to be reaped.

The `sigsuspend` version is less wasteful than the original spin loop, avoids the race introduced by `pause`, and is more efficient than `sleep`.

8.6 Nonlocal Jumps

C provides a form of user-level exceptional control flow, called a *nonlocal jump*, that transfers control directly from one function to another currently executing function without having to go through the normal call-and-return sequence. Nonlocal jumps are provided by the `setjmp` and `longjmp` functions.