code/ecf/shellex.c

```
1  #include "csapp.h"
2  #define MAXARGS   128
3
4  /* Function prototypes */
5  void eval(char *cmdline);
6  int parseline(char *buf, char **argv);
7  int builtin_command(char **argv);
8
9  int main()
10 {
11     char cmdline[MAXLINE]; /* Command line */
12
13     while (1) {
14         /* Read */
15         printf("> ");
16         Fgets(cmdline, MAXLINE, stdin);
17         if (feof(stdin))
18             exit(0);
19
20         /* Evaluate */
21         eval(cmdline);
22     }
23 }
```

code/ecf/shellex.c

Figure 8.23 The main routine for a simple shell program.

code/ecf/shellex.c

```
1  /* eval - Evaluate a command line */
2  void eval(char *cmdline)
3  {
4      char *argv[MAXARGS]; /* Argument list execve() */
5      char buf[MAXLINE];   /* Holds modified command line */
6      int bg;              /* Should the job run in bg or fg? */
7      pid_t pid;           /* Process id */
8
9      strcpy(buf, cmdline);
10     bg = parseline(buf, argv);
11     if (argv[0] == NULL)
12         return;   /* Ignore empty lines */
13
14     if (!builtin_command(argv)) {
15         if ((pid = Fork()) == 0) {   /* Child runs user job */
16             if (execve(argv[0], argv, environ) < 0) {
17                 printf("%s: Command not found.\n", argv[0]);
18                 exit(0);
19             }
20         }
21
22         /* Parent waits for foreground job to terminate */
23         if (!bg) {
24             int status;
25             if (waitpid(pid, &status, 0) < 0)
26                 unix_error("waitfg: waitpid error");
27         }
28         else
29             printf("%d %s", pid, cmdline);
30     }
31     return;
32 }
33
34 /* If first arg is a builtin command, run it and return true */
35 int builtin_command(char **argv)
36 {
37     if (!strcmp(argv[0], "quit")) /* quit command */
38         exit(0);
39     if (!strcmp(argv[0], "&"))    /* Ignore singleton & */
40         return 1;
41     return 0;                     /* Not a builtin command */
42 }
```

code/ecf/shellex.c

Figure 8.24 `eval` evaluates the shell command line.

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.

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.)

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.

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.

```
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.

_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.)

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.

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.

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.

```c
#include "csapp.h"

volatile sig_atomic_t pid;

void sigchld_handler(int s)
{
    int olderrno = errno;
    pid = waitpid(-1, NULL, 0);
    errno = olderrno;
}

void sigint_handler(int s)
{
}

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);
}
```

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.

code/ecf/sigsuspend.c

```c
#include "csapp.h"

volatile sig_atomic_t pid;

void sigchld_handler(int s)
{
    int olderrno = errno;
    pid = Waitpid(-1, NULL, 0);
    errno = olderrno;
}

void sigint_handler(int s)
{
}

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);
}
```

code/ecf/sigsuspend.c

Figure 8.42 Waiting for a signal with `sigsuspend`.

code/ecf/setjmp.c

```
1  #include "csapp.h"
2
3  jmp_buf buf;
4
5  int error1 = 0;
6  int error2 = 1;
7
8  void foo(void), bar(void);
9
10 int main()
11 {
12     switch(setjmp(buf)) {
13     case 0:
14         foo();
15         break;
16     case 1:
17         printf("Detected an error1 condition in foo\n");
18         break;
19     case 2:
20         printf("Detected an error2 condition in foo\n");
21         break;
22     default:
23         printf("Unknown error condition in foo\n");
24     }
25     exit(0);
26 }
27
28 /* Deeply nested function foo */
29 void foo(void)
30 {
31     if (error1)
32         longjmp(buf, 1);
33     bar();
34 }
35
36 void bar(void)
37 {
38     if (error2)
39         longjmp(buf, 2);
40 }
```

code/ecf/setjmp.c

Figure 8.43 Nonlocal jump example. This example shows the framework for using nonlocal jumps to recover from error conditions in deeply nested functions without having to unwind the entire stack.

code/ecf/restart.c

```
1  #include "csapp.h"
2
3  sigjmp_buf buf;
4
5  void handler(int sig)
6  {
7      siglongjmp(buf, 1);
8  }
9
10 int main()
11 {
12     if (!sigsetjmp(buf, 1)) {
13         Signal(SIGINT, handler);
14         Sio_puts("starting\n");
15     }
16     else
17         Sio_puts("restarting\n");
18
19     while(1) {
20         Sleep(1);
21         Sio_puts("processing...\n");
22     }
23     exit(0); /* Control never reaches here */
24 }
```

code/ecf/restart.c

Figure 8.44 A program that uses nonlocal jumps to restart itself when the user types Ctrl+C.