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.

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`.

```
#include <setjmp.h>

int setjmp(jmp_buf env);
int sigsetjmp(sigjmp_buf env, int savesigs);
                        Returns: 0 from setjmp, nonzero from longjmps
```

The `setjmp` function saves the current *calling environment* in the `env` buffer, for later use by `longjmp`, and returns 0. The calling environment includes the program counter, stack pointer, and general-purpose registers. For subtle reasons beyond our scope, the value that `setjmp` returns should not be assigned to a variable:

```
rc = setjmp(env);  /* Wrong! */
```

However, it can be safely used as a test in a `switch` or conditional statement [62].

```
#include <setjmp.h>

void longjmp(jmp_buf env, int retval);
void siglongjmp(sigjmp_buf env, int retval);
                                                    Never returns
```

The `longjmp` function restores the calling environment from the `env` buffer and then triggers a return from the most recent `setjmp` call that initialized `env`. The `setjmp` then returns with the nonzero return value `retval`.

The interactions between `setjmp` and `longjmp` can be confusing at first glance. The `setjmp` function is called once but returns *multiple times:* once when the `setjmp` is first called and the calling environment is stored in the `env` buffer, and once for each corresponding `longjmp` call. On the other hand, the `longjmp` function is called once but never returns.

An important application of nonlocal jumps is to permit an immediate return from a deeply nested function call, usually as a result of detecting some error condition. If an error condition is detected deep in a nested function call, we can use a nonlocal jump to return directly to a common localized error handler instead of laboriously unwinding the call stack.

Figure 8.43 shows an example of how this might work. The `main` routine first calls `setjmp` to save the current calling environment, and then calls function `foo`, which in turn calls function `bar`. If `foo` or `bar` encounter an error, they return immediately from the `setjmp` via a `longjmp` call. The nonzero return value of the `setjmp` indicates the error type, which can then be decoded and handled in one place in the code.

The feature of `longjmp` that allows it to skip up through all intermediate calls can have unintended consequences. For example, if some data structures were allocated in the intermediate function calls with the intention to deallocate them at the end of the function, the deallocation code gets skipped, thus creating a memory leak.

code/ecf/setjmp.c

```c
#include "csapp.h"

jmp_buf buf;

int error1 = 0;
int error2 = 1;

void foo(void), bar(void);

int main()
{
    switch(setjmp(buf)) {
    case 0:
        foo();
        break;
    case 1:
        printf("Detected an error1 condition in foo\n");
        break;
    case 2:
        printf("Detected an error2 condition in foo\n");
        break;
    default:
        printf("Unknown error condition in foo\n");
    }
    exit(0);
}

/* Deeply nested function foo */
void foo(void)
{
    if (error1)
        longjmp(buf, 1);
    bar();
}

void bar(void)
{
    if (error2)
        longjmp(buf, 2);
}
```

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

```
#include "csapp.h"

sigjmp_buf buf;

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

int main()
{
    if (!sigsetjmp(buf, 1)) {
        Signal(SIGINT, handler);
        Sio_puts("starting\n");
    }
    else
        Sio_puts("restarting\n");

    while(1) {
        Sleep(1);
        Sio_puts("processing...\n");
    }
    exit(0); /* Control never reaches here */
}
```

code/ecf/restart.c

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

Another important application of nonlocal jumps is to branch out of a signal handler to a specific code location, rather than returning to the instruction that was interrupted by the arrival of the signal. Figure 8.44 shows a simple program that illustrates this basic technique. The program uses signals and nonlocal jumps to do a soft restart whenever the user types Ctrl+C at the keyboard. The `sigsetjmp` and `siglongjmp` functions are versions of `setjmp` and `longjmp` that can be used by signal handlers.

The initial call to the `sigsetjmp` function saves the calling environment and signal context (including the pending and blocked signal vectors) when the program first starts. The main routine then enters an infinite processing loop. When the user types Ctrl+C, the kernel sends a SIGINT signal to the process, which catches it. Instead of returning from the signal handler, which would pass control back to the interrupted processing loop, the handler performs a nonlocal jump back to the beginning of the `main` program. When we run the program on our system, we get the following output:

Aside Software exceptions in C++ and Java

The exception mechanisms provided by C++ and Java are higher-level, more structured versions of the C `setjmp` and `longjmp` functions. You can think of a `catch` clause inside a `try` statement as being akin to a `setjmp` function. Similarly, a `throw` statement is similar to a `longjmp` function.

```
linux> ./restart
starting
processing...
processing...
Ctrl+C
restarting
processing...
Ctrl+C
restarting
processing...
```

There a couple of interesting things about this program. First, To avoid a race, we must install the handler *after* we call `sigsetjmp`. If not, we would run the risk of the handler running before the initial call to `sigsetjmp` sets up the calling environment for `siglongjmp`. Second, you might have noticed that the `sigsetjmp` and `siglongjmp` functions are not on the list of async-signal-safe functions in Figure 8.33. The reason is that in general `siglongjmp` can jump into arbitrary code, so we must be careful to call only safe functions in any code reachable from a `siglongjmp`. In our example, we call the safe `sio_puts` and `sleep` functions. The unsafe `exit` function is unreachable.

8.7 Tools for Manipulating Processes

Linux systems provide a number of useful tools for monitoring and manipulating processes:

STRACE. Prints a trace of each system call invoked by a running program and its children. It is a fascinating tool for the curious student. Compile your program with `-static` to get a cleaner trace without a lot of output related to shared libraries.

PS. Lists processes (including zombies) currently in the system.

TOP. Prints information about the resource usage of current processes.

PMAP. Displays the memory map of a process.

`/proc`. A virtual filesystem that exports the contents of numerous kernel data structures in an ASCII text form that can be read by user programs. For example, type `cat /proc/loadavg` to see the current load average on your Linux system.