Parent and Child Synchronization

```
  int
  main(void)
  {
      pid_t   pid;

+     TELL_WAIT();
+
      if ((pid = fork()) < 0) {
          err_sys("fork error");
      } else if (pid == 0) {
+         WAIT_PARENT();      /* parent goes first */
          charatatime("output from child\n");
      } else {
          charatatime("output from parent\n");
+         TELL_CHILD(pid);
      }
      exit(0);
  }
  static void
  charatatime(char *str)
  {
      char    *ptr;
      int     c;

      setbuf(stdout, NULL);           /* set unbuffered */
      for (ptr = str; (c = *ptr++) != 0; )
          putc(c, stdout);
  }
```

Parent and Child Synchronization

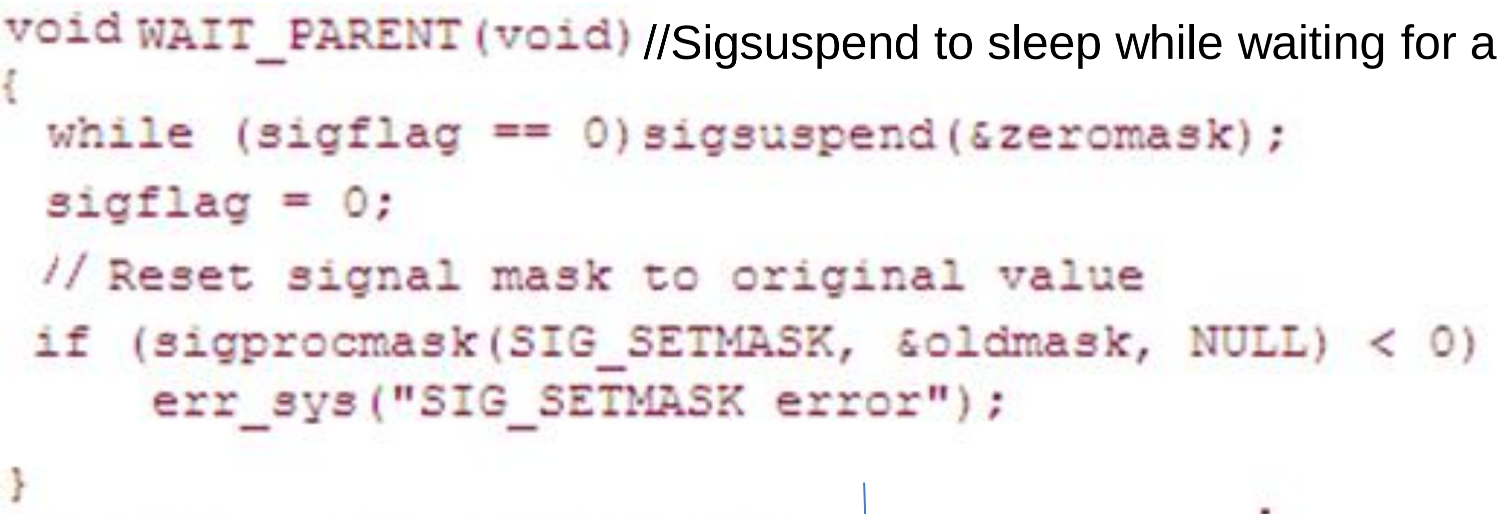
- To synchronize, consider
  - Child sending SIGUSR2 to the parent
  - Parent sending SIGUSR1 to the child

Routines for Parent-Child Synchronization

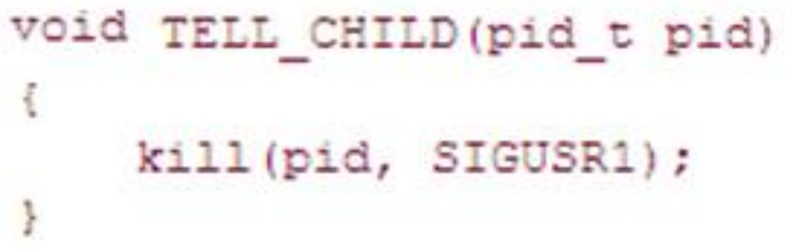
```
static volatile sig_atomic_t sigflag;
static sigset_t newmask, oldmask, zeromask;

static void sig_usr(int signo)
{
    sigflag = 1;
}
void TELL_WAIT(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("signal(SIGUSR1) error");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("signal(SIGUSR2) error");
    sigemptyset(&zeromask);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGUSR1);
    sigaddset(&newmask, SIGUSR2);
   //Block SIGUSR1 and SIGUSR2, and save current signal mask
   if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
       err_sys("SIG_BLOCK error");
```

```
main(void)
{ pid_t   pid;
  TELL_WAIT();
 if ((pid = fork()) == 0) {
     WAIT_PARENT();
     charatatime("output from child\n");
 } else {
     charatatime("output from parent\n");
     TELL_CHILD(pid);
 }
 exit(0);
}
```

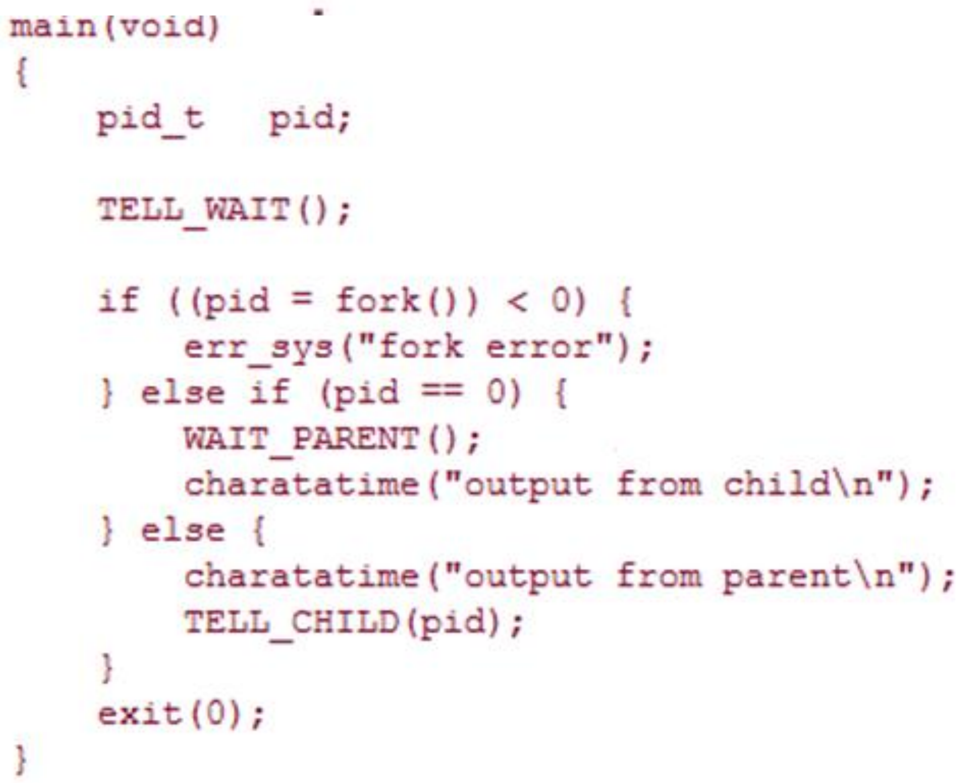
Routines Contd…



```
void WAIT_PARENT(void) //Sigsuspend to sleep while waiting for a signal to occur
{
 while (sigflag == 0)sigsuspend(&zeromask);
 sigflag = 0;
 // Reset signal mask to original value
 if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
     err_sys("SIG_SETMASK error");

}
```



```
void TELL_CHILD(pid_t pid)
 {
      kill(pid, SIGUSR1);
 }
```



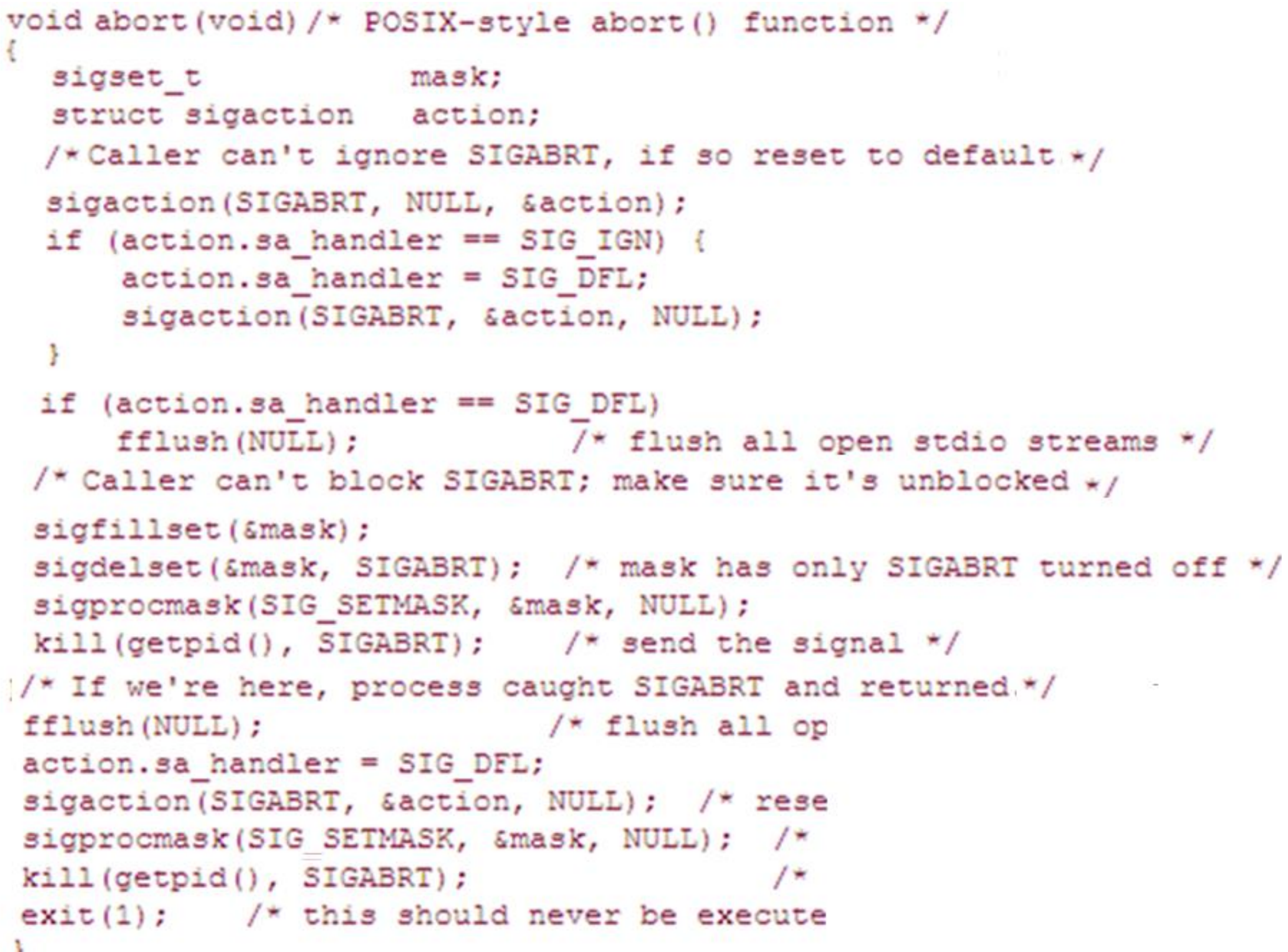
```
main(void)
{
        pid_t   pid;

        TELL_WAIT();

        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) {
            WAIT_PARENT();
            charatatime("output from child\n");
        } else {
            charatatime("output from parent\n");
            TELL_CHILD(pid);
        }
        exit(0);
}
```

abort() Function

- Causes abnormal program termination
- void abort(void);

  This function never returns
- abort() sends SIGABRT to the process.
- ANSI C requires that if the signal is caught and the signal handler returns, abort still does not return to the caller
- By catching SIGABRT, the process can be allowed to perform any cleanup

Implementation of POSIX.1 abort()



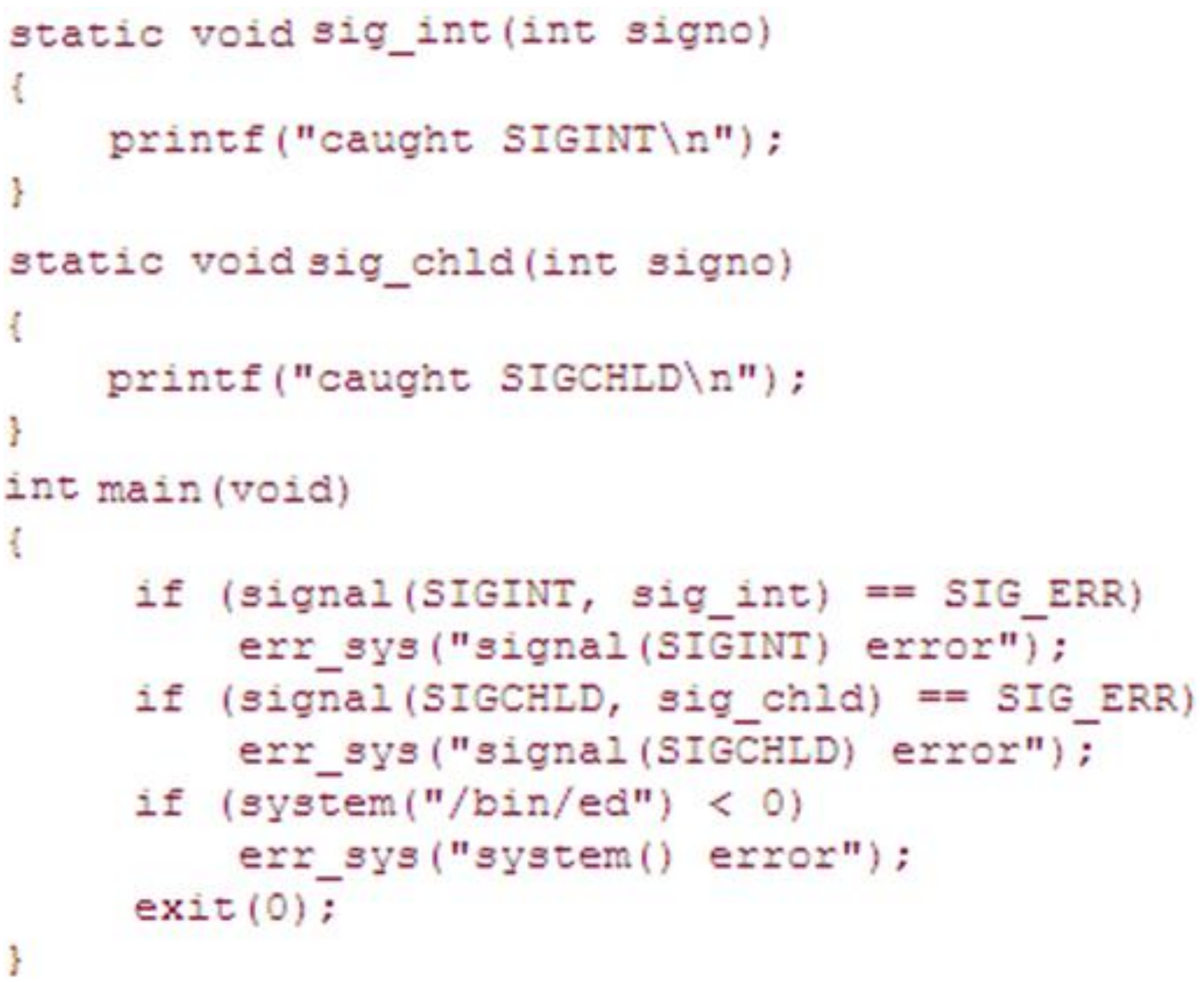
```
void abort(void) /* POSIX-style abort() function */
{
  sigset_t            mask;
  struct sigaction    action;
  /* Caller can't ignore SIGABRT, if so reset to default */
  sigaction(SIGABRT, NULL, &action);
  if (action.sa_handler == SIG_IGN) {
      action.sa_handler = SIG_DFL;
      sigaction(SIGABRT, &action, NULL);
  }
  if (action.sa_handler == SIG_DFL)
      fflush(NULL);             /* flush all open stdio streams */
  /* Caller can't block SIGABRT; make sure it's unblocked */
  sigfillset(&mask);
  sigdelset(&mask, SIGABRT);  /* mask has only SIGABRT turned off */
  sigprocmask(SIG_SETMASK, &mask, NULL);
  kill(getpid(), SIGABRT);    /* send the signal */
 /* If we're here, process caught SIGABRT and returned */
 fflush(NULL);                /* flush all op
 action.sa_handler = SIG_DFL;
 sigaction(SIGABRT, &action, NULL);  /* rese
 sigprocmask(SIG_SETMASK, &mask, NULL);  /*
 kill(getpid(), SIGABRT);                /*
 exit(1);    /* this should never be execute
}
```

system() Function

- Implementation of system without signal handling

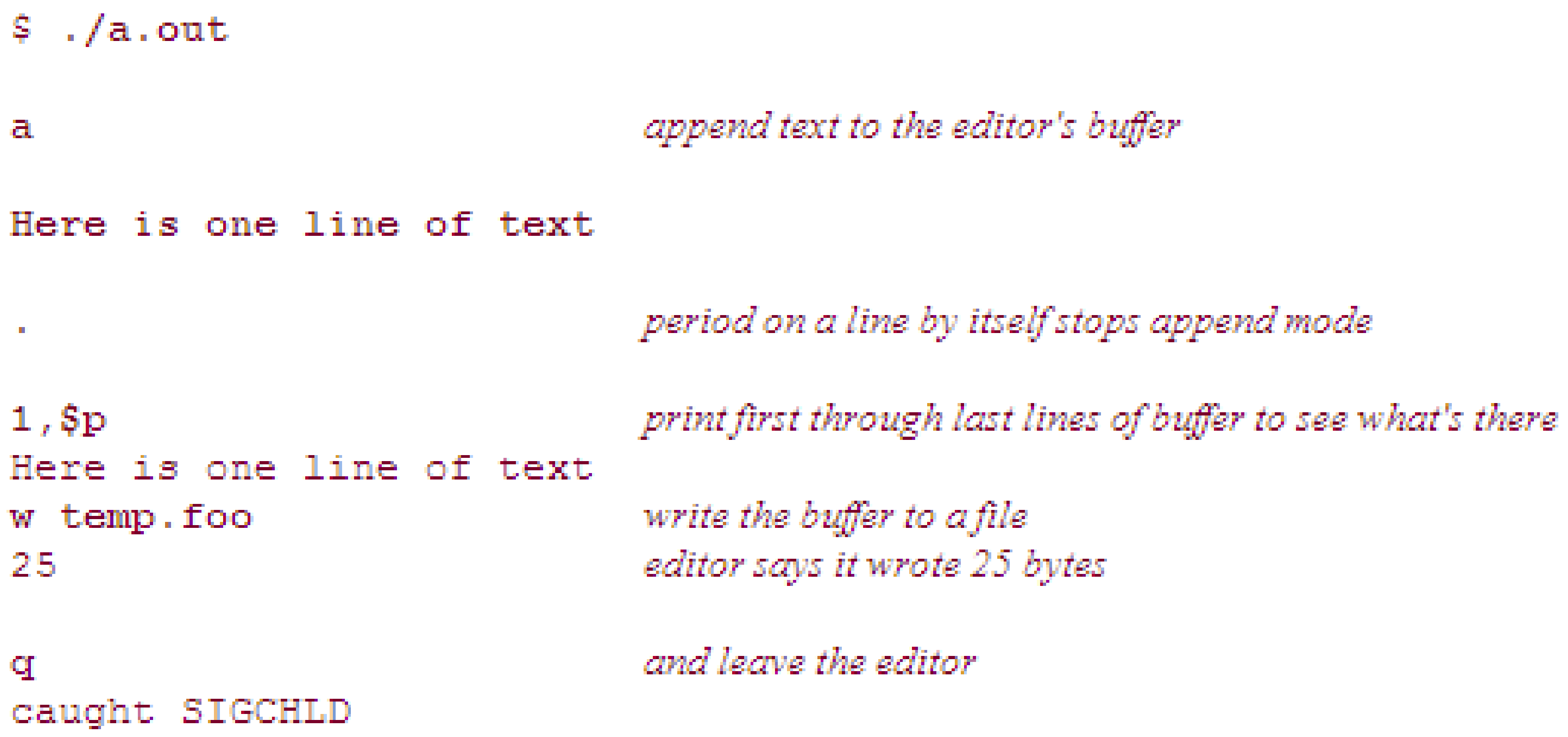
```
int system(const char *cmdstring)  {
    pid_t pid; int status;
    if (cmdstring == NULL)  return(1); /* always a command processor with UNIX */
    if ((pid = fork()) < 0) status = -1; /* probably out of processes */
    else if (pid == 0) { /* child */
            execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
             _exit(127); /* execl error */
    } else { /* parent */
                 while (waitpid(pid, &status, 0) < 0) {
                       if (errno != EINTR) {
                                 status = -1; /* error other than EINTR from waitpid() */
                                 break;
                  }
            }
    }
    return(status);
}
```

Using system() to Invoke *ed* Editor



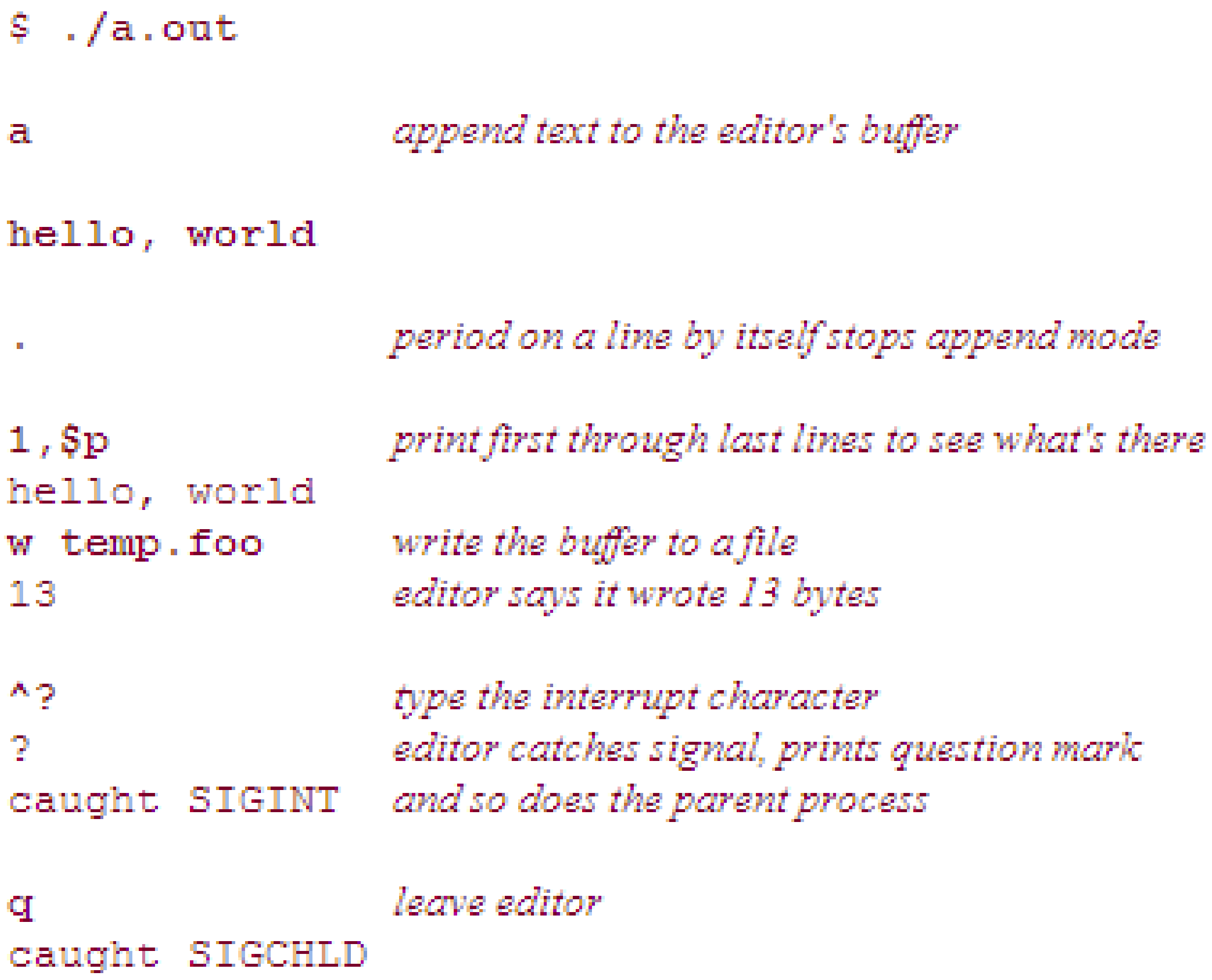
```
static void sig_int(int signo)
{
    printf("caught SIGINT\n");
}
static void sig_chld(int signo)
{
    printf("caught SIGCHLD\n");
}
int main(void)
{
     if (signal(SIGINT, sig_int) == SIG_ERR)
         err_sys("signal(SIGINT) error");
     if (signal(SIGCHLD, sig_chld) == SIG_ERR)
         err_sys("signal(SIGCHLD) error");
     if (system("/bin/ed") < 0)
         err_sys("system() error");
     exit(0);
}
```

Execution



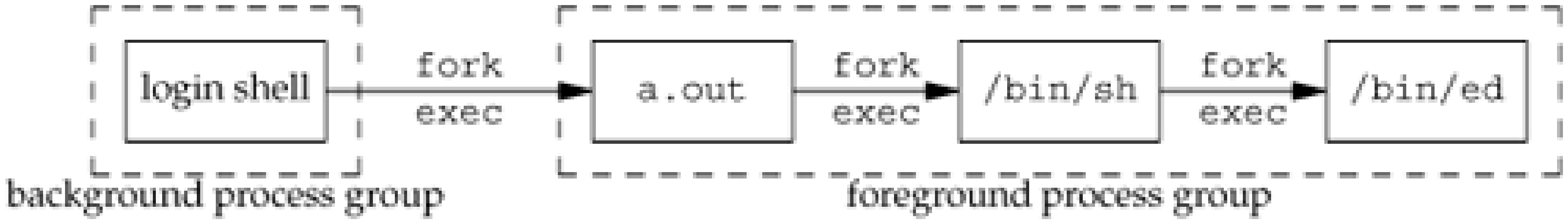
```
$ ./a.out

a                          append text to the editor's buffer

Here is one line of text

.                          period on a line by itself stops append mode

1,$p                       print first through last lines of buffer to see what's there
Here is one line of text
w temp.foo                 write the buffer to a file
25                         editor says it wrote 25 bytes

q                          and leave the editor
caught SIGCHLD
```

Execution



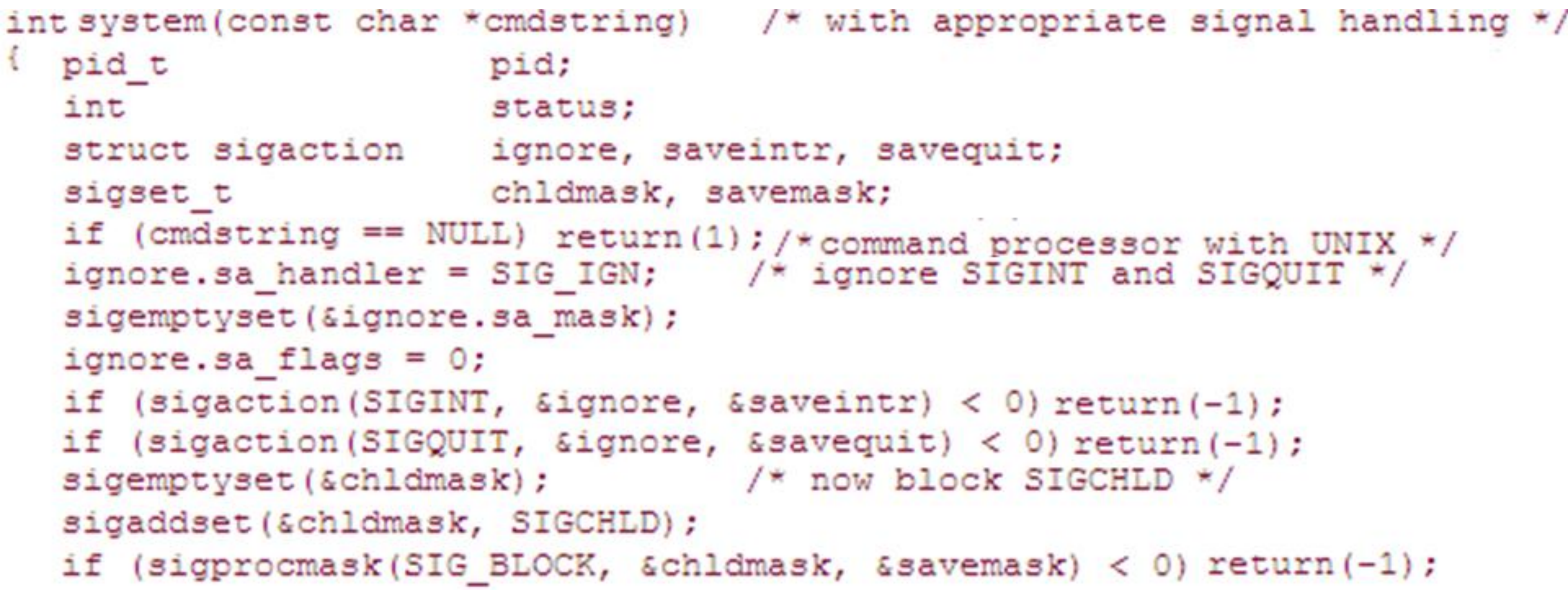
```
$ ./a.out

a                  append text to the editor's buffer

hello, world

.                  period on a line by itself stops append mode

1,$p               print first through last lines to see what's there
hello, world
w temp.foo         write the buffer to a file
13                 editor says it wrote 13 bytes

^?                 type the interrupt character
?                  editor catches signal, prints question mark
caught SIGINT      and so does the parent process

q                  leave editor
caught SIGCHLD
```

Foreground and Background Process Groups



- SIGINT is sent to all three foreground processes
- Shell ignores it
- a.out process and *ed* catch the signal
- a.out should ignore the signals and *ed* only should catch the signal

Correct POSIX.1 Pmplementation of system()



```
int system(const char *cmdstring)    /* with appropriate signal handling */
{  pid_t                   pid;
   int                     status;
   struct sigaction        ignore, saveintr, savequit;
   sigset_t                chldmask, savemask;
   if (cmdstring == NULL) return(1); /*command processor with UNIX */
   ignore.sa_handler = SIG_IGN;      /* ignore SIGINT and SIGQUIT */
   sigemptyset(&ignore.sa_mask);
   ignore.sa_flags = 0;
   if (sigaction(SIGINT, &ignore, &saveintr) < 0) return(-1);
   if (sigaction(SIGQUIT, &ignore, &savequit) < 0) return(-1);
   sigemptyset(&chldmask);           /* now block SIGCHLD */
   sigaddset(&chldmask, SIGCHLD);
   if (sigprocmask(SIG_BLOCK, &chldmask, &savemask) < 0) return(-1);
```

system() function Contd …

```
    if ((pid = fork()) < 0) {
        status = -1;    /* probably out of processes */
    } else if (pid == 0) {          /* child */
        /* restore previous signal actions & reset signal mask */
        sigaction(SIGINT, &saveintr, NULL);
        sigaction(SIGQUIT, &savequit, NULL);
        sigprocmask(SIG_SETMASK, &savemask, NULL);

        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);     /* exec error */
    } else {                        /* parent */
       while (waitpid(pid, &status, 0) < 0)
           if (errno != EINTR) {
               status = -1; /* error other than EINTR from waitpid() */
               break;
           }
    }

    /* restore previous signal actions & reset signal mask */
    if (sigaction(SIGINT, &saveintr, NULL) < 0)
        return(-1);
    if (sigaction(SIGQUIT, &savequit, NULL) < 0)
        return(-1);
    if (sigprocmask(SIG_SETMASK, &savemask, NULL) < 0)
        return(-1);

    return(status);
}
```

POSIX Implementation

- No signal is sent to caller when interrupt or quit character is typed
- When ed exits, SIGCHILD is sent to caller and it is blocked. Caller calls waitpid, collects termination status. POSIX.1 states that if wait or waitpid returns the status of a child process while SIGCHLD is pending, then SIGCHLD should not be delivered to the process unless the status of another child process is also available.
- If the caller has any pending SIGCHLD (of own children), that will be pending after waitpid and will be unblocked after sigprocmask.

sleep ()

- unsigned int sleep(unsigned int seconds);
  Returns: 0 or number of unslept seconds
- This function causes the calling process to be suspended until either
  1. The amount of wall clock time specified by seconds has elapsed
     - Returns 0
  2. A signal is caught by the process and the signal handler returns
     - Returns number of unslept seconds
- if we do an alarm(10) and 3 wall clock seconds later do a sleep(5), what happens?
  - Sleep() will return in 5 seconds (assuming that some other signal isn't caught in that time), but will another SIGALRM be generated 2 seconds later?
  - These details depend on the implementation.

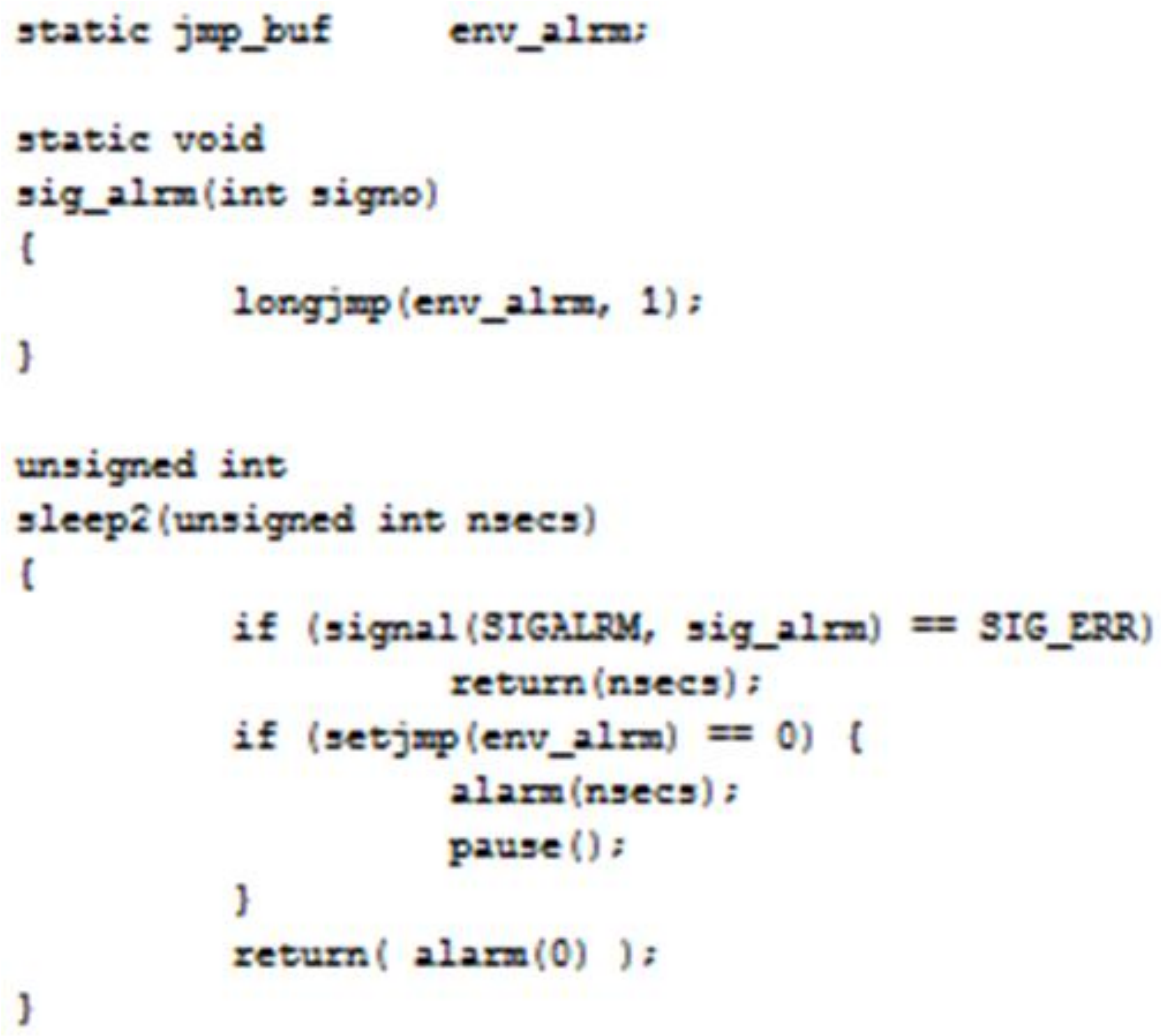
sleep() - Simple, incomplete implementation

```
#include        <signal.h>
#include        <unistd.h>

static void
sig_alrm(int signo)
{
        return; /* nothing to do, just return to wake up the pause */
}

unsigned int
sleep1(unsigned int nsecs)
{
        if (signal(SIGALRM, sig_alrm) == SIG_ERR)
                return(nsecs);
        alarm(nsecs);                   /* start the timer */
        pause();                                /* next caught signal wakes us up */
        return( alarm(0) );             /* turn off timer, return unslept time */
}
```

sleep() implementation - Problems

- If the caller has already an alarm set, that alarm is erased by the first call to alarm
- We have modified the disposition of SIGALRM, we should save the disposition and restore it
- Race condition between first call to alarm and pause

Another (imperfect) implementation of sleep (SVR2)



```
static jmp_buf      env_alrm;

static void
sig_alrm(int signo)
{
        longjmp(env_alrm, 1);
}

unsigned int
sleep2(unsigned int nsecs)
{
        if (signal(SIGALRM, sig_alrm) == SIG_ERR)
                return(nsecs);
        if (setjmp(env_alrm) == 0) {
                alarm(nsecs);
                pause();
        }
        return( alarm(0) );
}
```

Problem with implementation

- Race condition is eliminated
- Interaction with other signals
  - If SIGALRM interrupts some other signal handler, when longjmp is called, it aborts the other signal handler.

Calling sleep2 from a program that catches other signals

```
unsigned int      sleep2(unsigned int);
static void       sig_int(int);
int main(void)
{  unsigned int      unslept;
   if (signal(SIGINT, sig_int) == SIG_ERR) err_sys("signal(SIGINT) error");
   unslept = sleep2(5);
   printf("sleep2 returned: %u\n", unslept);
   exit(0);
}
 static void sig_int(int signo)
 {  int i;
    volatile int      j;
    printf("\nsig_int starting\n");
    for (i = 0; i < 2000000; i++) j += i * i;
    printf("sig_int finished\n");
    return;
 }
```

sleep in Unix flavors

- Solaris 9 implements sleep using alarm. Manual page says that a previously scheduled alarm is properly handled.
  - For example, in the preceding scenario, before sleep returns, it will reschedule the alarm to happen 2 seconds later; sleep returns 0 in this case.
  - Also, if we do an alarm(6) and 3 wall clock seconds later do a sleep(5), the sleep returns in 3 seconds, not in 5 seconds. Here, the return value from sleep is 2.
- FreeBSD 5.2.1, Linux 2.4.22, and Mac OS X 10.3, the delay is provided by nanosleep, a function specified to be a high-resolution delay by the real-time extensions in the Single UNIX Specification. This function allows the implementation of sleep to be independent of signals.

Posix.1 Implementation of sleep()

- Handles signals reliably, avoid race conditions between alarm and pause
- No effect on other signal handler that may be executing when SIGALRM is handled
- Do not handle interaction with previously set alarms.

Posix.1 Implementation of sleep()

```
static void
sig_alrm(int signo)
{
    /* nothing to do, just returning wakes up sigsuspend() */
}

unsigned int
sleep(unsigned int nsecs)
{
    struct sigaction    newact, oldact;
    sigset_t            newmask, oldmask, suspmask;
    unsigned int        unslept;

    /* set our handler, save previous information */
    newact.sa_handler = sig_alrm;
    sigemptyset(&newact.sa_mask);
    newact.sa_flags = 0;
    sigaction(SIGALRM, &newact, &oldact);

    /* block SIGALRM and save current signal mask */
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGALRM);
    sigprocmask(SIG_BLOCK, &newmask, &oldmask);

    alarm(nsecs);
```

Implementation of sleep()

```
    suspmask = oldmask;
    sigdelset(&suspmask, SIGALRM);    /* make sure SIGALRM isn't blocked */
    sigsuspend(&suspmask);            /* wait for any signal to be caught */

    /* some signal has been caught,   SIGALRM is now blocked */

    unslept = alarm(0);
    sigaction(SIGALRM, &oldact, NULL);  /* reset previous action */

    /* reset signal mask, which unblocks SIGALRM */
    sigprocmask(SIG_SETMASK, &oldmask, NULL);
    return(unslept);
}
```

Values with signal masks

- oldmask - initial when process starts
- newmask - block SIGALRM // 1. any previously set alarm expires, should not get confused. 2. if Alarm expires before pause, SIGALRM is blocked
- suspmask- oldmask – SIGALRM// ensure SIGALRM not blocked, some signal will be caught
- //if alarm expires before pause, blocked SIGALRM is unblocked by sigsuspend call, delivers SIGALRM and breaks pause
- newmask - block SIGALRM //pause broke due to some other signal, alarm expires, block SIGALRM
- Oldmask - //unblocks SIGALRM