Chapter 7

Signals

Asynchronous Events: UNIX Signals

Signal lifecycle, short version:

1. A signal is **generated** by the occurrence of a particular event
2. A generated signal is **delivered** to a process
3. Once delivered, the signal must be **handled**

Two kinds of signals:

– **Synchronous events:** produced by the target process

SIGSEGV (11), SIGPIPE, SIGFPE, SIGILL, SIGABRT are typically synchronous, but can also be sent from another process

– **Asynchronous events:** produced by another process

SIGKILL (9), SIGTERM, SIGINT, SIGQUIT, SIGHUP (1), SIGCHLD, SIGSTOP, SIGUSR1, SIGUSR2

Signal Lifecycle, Long Version

1. A signal is **generated** and directed to a process
2. If the process **ignores** this signal, the signal is discarded
   — KILL, STOP cannot be ignored
3. If the process **blocks** this signal, it is kept **pending** until the process unblocks it
   — KILL, STOP cannot be blocked
4. Otherwise, the signal is **delivered** to the process
5. Once delivered, the signal must be **handled**
   – The **default** action is to ***terminate*** the process;
     - the process is ***stopped*** by STOP (^Z), TTIN, TTOU, TSTP (^S — type ^Q to continue)
     - CONT ***continues*** the process; CHLD is ignored
   – The default action is not taken if a signal handler has been installed for **catching** the signal
     — KILL, STOP cannot be caught

Sending Signals

- **kill(1)** or shell builtin:

```
kill [ -s signal ] pid ...
kill -l [ signal ]

kill -s sigspec   [pid | jobspec]
kill -n signum    [pid | jobspec]
kill [ -sigspec ] [pid | jobspec]
```

- **killall(1)** — by command name
- **kill(2)**: *#include <sys/types.h>*
  #include <signal.h>
  int *kill*(*pid_t pid*, int *sig*);
- **raise(3)**: sends a signal to calling process
- **abort(3)**: sends *SIGABRT* to calling process
- **alarm(3)**: sends *SIGALRM* to caller after specified time

Interface to Signal Handling

- **sigprocmask(2)**: blocking and unblocking signals
- **sigpending(2)** allows the examination of pending signals
- **sigsuspend(2)**: temporarily replaces the signal mask and then suspends the process until a signal is received.
- **sigaction(2)** allows to install handlers — *SIG_IGN* is the special handler used for ignoring; *SIG_DFL* refers to the default action

USP Program 8.1 — blocktest.c

```
int main(int argc,  char *argv[]) {
  double y = 0.0; int i, repeatfactor = atoi(argv[1]);
  sigset_t intmask;
  if ( sigemptyset(&intmask) || sigaddset(&intmask, SIGINT) )
   { perror("Failed to initialize the signal mask"); return 1; }
  for ( ; ; ) {
    if (sigprocmask(SIG_BLOCK, &intmask, NULL) == -1) break;
    fprintf(stderr, "SIGINT signal blocked\n");
    for (i = 0; i < repeatfactor; i++) y += sin((double)i);
    fprintf(stderr, "Blocked calculation is finished, y = %f\n", y);
    if (sigprocmask(SIG_UNBLOCK, &intmask, NULL) == -1)
      break;
    fprintf(stderr, "SIGINT signal unblocked\n");
    for (i = 0; i < repeatfactor; i++) y += sin((double)i);
    fprintf(stderr, "Unblocked calculation finished, y=%f\n", y);
  }
  perror("Failed to change signal mask"); return 1;
}
```


USP Program 8.4 — password.c

```
int password(const char *prompt, char *passbuf, int passmax) {
  int fd, firsterrno = 0;
  sigset_t signew, sigold;
  char termbuf[L_ctermid];

  if (ctermid(termbuf) == NULL) {         /* find the terminal name */
    errno = ENODEV;
    return -1;
  }
  if ((fd = open(termbuf, O_RDONLY)) == -1) /* open terminal */
    return -1;
  if ((sigemptyset(&signew) == -1) |       | /* blocked signals */
     (sigaddset(&signew, SIGINT) == -1) ||
     (sigaddset(&signew, SIGQUIT) == -1) ||
     (sigaddset(&signew, SIGTSTP) == -1) ||
     (sigprocmask(SIG_BLOCK, &signew, &sigold) == -1) ||
     (setecho(fd, 0) == -1)) {              /* set terminal echo off */
      firsterrno = errno;
```



```
    sigprocmask(SIG_SETMASK, &sigold, NULL);
    r_close(fd);
    errno = firsterrno;
    return -1;
  }
  if ((r_write(STDOUT_FILENO, (char*)prompt, strlen(prompt))
     == -1) ||
     (readline(fd, passbuf, passmax) == -1))  /* read password */
    firsterrno = errno;
  else
    passbuf[strlen(passbuf) - 1] = 0;           /* remove newline */
  if ((setecho(fd, 1) == -1) && !firsterrno) /* turn echo back on */
    firsterrno = errno;
  if ((sigprocmask(SIG_SETMASK, &sigold, NULL) == -1)
     && ! firsterrno )
    firsterrno = errno;
  if ((r_close(fd) == -1) && !firsterrno)   /* close terminal */
    firsterrno = errno;
  return firsterrno ? errno = firsterrno, -1 : 0;
}
```

sigaction(2)

```
int sigaction(int signum, const struct sigaction *act,
                          struct sigaction *oldact);
```

- *signum* cannot be *SIGKILL* or *SIGSTOP*.
- If *act* is non-null, it defines the new action for signal *signum*.
- *SIG_IGN* is the special handler used for ignoring.
- *SIG_DFL* refers to the default action.
- If *oldact* is non-null, the previous action is saved in *oldact*.

struct *sigaction*

```
struct sigaction {          /* preliminary */
   void (*sa_handler)(int);
   sigset_t sa_mask;
   int sa_flags;
 }
```

- *sa_mask* contains signals blocked during execution of the signal handler in addition to *signum*.
- *sa_flags* may contain the following:

  POSIX: *SA_NOCLDSTOP*

  Linux (additionally): *SA_ONESHOT*, *SA_ONSTACK*, *SA_RESTART*, *SA_NOMASK*, *SA_SIGINFO*

USP Example 8.15

```
#include <signal.h>
#include <stdio.h>
struct  sigaction  act;
                              /* Find current signal handler */
if (sigaction(SIGINT, NULL, &act) == –1)
  perror("Failed to get old handler for SIGINT");
else if (act.sa_handler == SIG_DFL)
{  /* ignore SIGINT */
  act.sa_handler = SIG_IGN;    /* set handler to ignore */
  if (sigaction(SIGINT, &act, NULL) == –1)
    perror("Failed to ignore SIGINT");
}
```


USP Example 8.16

```
void catch_ctrl_c(int signo)
{  char handmsg[] = "I found Ctrl-C\n";
   int msglen = sizeof(handmsg) – 1;
   write(STDERR_FILENO, handmsg, msglen); }
...
struct sigaction act;
...
act.sa_handler = catch_ctrl_c;
sigemptyset(&act.sa_mask);
act.sa_flags = 0;
if (sigaction(SIGINT, &act, NULL) < 0)
  /* handle error here */
...
```

- *fprintf*() and *strlen*() are **not async-signal safe!**

Frequent Uses of Signal Handlers

- Graceful termination
  - set termination flag (Program 8.5)
  - release resources — typically calling *exit*
- Trigger status messages (Program 8.6)
- Trigger configuration reload (many daemons)

Waiting for Signals

- int *pause*(void);

  suspends until a signal is delivered that either has a handler or terminates the process.

- int *sigsuspend*(const *sigset_t* **mask*);

  temporarily replaces signal mask and then suspends util a signal is received

- int *sigwait*(const *sigset_t* **set*, int **sig*);

  suspends until signal from *set* is pending; then unblocks and removes that signal from pending set.

The Problem with *pause*() — USP Exercise 8.21

```
static volatile sig_atomic_t sigreceived = 0;

while(sigreceived == 0)
   pause();
```

- Assume the handler sets *sigreceived* to 1
- What happens if a signal is delivered between the test of *sigreceived* and the call of *pause*?
- Blocking the signal for testing *sigreceived* …
- Unblocking and suspending needs to be atomic!

  ⇒ *sigsuspend*()

PUP Example 5.20

```
#include <signal.h>
volatile int signal_received = 0; /* external static variable */
...
sigset_t sigset, sigoldmask;
int signum;

sigprocmask(SIG_SETMASK, NULL, &sigoldmask);
sigprocmask(SIG_SETMASK, NULL, &sigset);
sigaddset(&sigset, signum);
sigprocmask(SIG_BLOCK, &sigset, NULL);
sigdelset(&sigset, signum);
while(signal_received == 0)
   sigsuspend(&sigset);
sigprocmask(SIG_SETMASK, &sigoldmask, NULL);
```

Concurrency Introduced by Signals

- While executing library function (or system call) *f*(), a program may catch a signal
- If the signal handler executes *f*(), too, two incarnations of *f*() exist concurrently
- ⇒ *f*() has to be **reentrant!**
  - Use only **async-signal safe** calls in signal handlers!
  - Carefully analyze shared data structures for possible interference.

    Block signals for avoiding interference.
  - Check return behaviour of system calls in program for signals (*EINTR*) — in doubt, restart system call.

break and continue in Nested Loops — Problem

```
for ( ... )
 { ...
  for ( ... )
   { ...
    if ( disaster() )
     break;   /* breaks only out of inner loop! */
    ...
   }
  ...
 }
 ...    /* clean up the mess */
```

Some languages allow to break out of specified loops.

break and continue in Nested Loops — Another Solution

```
for ( ... )
 { ...
  for ( ... )
   { ...
    if ( disaster() )
     goto error;   /* breaks out of both loops! */
    ...
   }
  ...
 }
error: ...    /* clean up the mess */
```

Labels are **local** to the function.

longjmp and *setjmp* — non-local goto

- int *setjmp*(*jmp_buf env*) saves the stack context/environment in *env* for later use by *longjmp*() and returns 0. The stack context will be invalidated if the function which called *setjmp*() returns.
- void *longjmp*(*jmp_buf env*, int *val*) restores the environment saved by the last call of *setjmp*() with the corresponding *env* argument. After *longjmp*() is completed, program execution continues as if the corresponding call of *setjmp*() had just returned the value *val* (never 0).
- goto considered harmful …

siglongjmp and *sigsetjmp*

- Analogous to *longjmp* and *setjmp*, but also can save signal mask in environment.
- goto still considered harmful …

siglongjmp and *sigsetjmp* Example: USP Program 8.12

```
static sigjmp_buf jmpbuf;
static volatile sig_atomic_t jumpok = 0;
void int_handler(int errno)
{ if (jumpok == 0) return; else siglongjmp(jmpbuf, 1); }

void main(void)
{  struct sigaction act;
   act.sa_handler = int_handler; act.sa_flags = 0;
   sigemptyset(&act.sa_mask);
   if (sigaction(SIGINT, &act, NULL) == –1)
    { perror("Error setting up SIGINT handler"); return 1; }
         ...
   if (sigsetjmp(jmpbuf, 1))
      fprintf(stderr, "Returned to main loop due to ^c\n");
   jumpok = 1;
   while ( 1 )   ...   /*  start of main loop */
```