Signals

Advanced Programming in the UNIX Environment

Chun-Ying Huang <chuang@cs.nctu.edu.tw>

Outline

Introduction

The signal function

Interrupted system calls

Reentrant functions

Reliable signal model

Functions for handling signals

Other issues

- setjmp and longjmp
- The system function

Introduction

Signals are software interrupts

Most nontrivial application programs need to deal with signals

Signals provide a way of handling asynchronous events

The original signal model

- Signals could get lost
- It was difficult for a process to turn off selected signals when executing critical regions of code

The reliable signal model

- The current implementation

Signal Concepts

Every signal has a name, beginning with the prefix SIG

- SIGINT, SIGSTOP, SIGHUP, SIGALRM, …

Signals are defined by positive integer constants

- Defined in the header signal.h
- /usr/include/signal.h -> /usr/include/bits/signum.h
- signal(7)

Signal Concepts (Cont'd)

Numerous conditions can generate a signal

The terminal-generated signals

- Occur when users press certain terminal keys: Ctrl-C, Ctrl-BackSpace, Ctrl-Z, …

Hardware exceptions

- For example, divide by zero, …

Software conditions

- For example, SIGPIPE, SIGALRM, …

The kill(2) function

- Allows a process to send any signal to another process or process group

The kill(1) command

- Allows a user to send signals to other processes

Handle Signals

We can tell the kernel to do one of three things when a signal occurs

Ignore the signal

- This works for most signals, but
- We cannot ignore SIGKILL and SIGSTOP

Catch the signal

- We can register a customized signal handler
- Also, we cannot register customized handlers for SIGKILL and SIGSTOP

Let the default action apply

- Every signal has a default action

List of Linux Signals

- Different platforms may have different signal numbers: alpha/sparc, **i386/ppc/sh**, mips

Name	Value	Dfl. Action	Description
SIGHUP	1	Term	Hangup or death detected on controlling terminal/process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGTRAP	5	Core	Trace/breakpoint trap
SIGABRT	6	Core	Abort signal from abort(3)
SIGBUS	10,7,10	Core	Bus error (bad memory access)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGSEGV	11	Core	Invalid memory reference
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal

List of Linux Signals (Cont'd)

- Different platforms may have different signal numbers: alpha/sparc, **i386/ppc/sh**, mips

Name	Value	Dfl. Action	Description
SIGSTKFLT	16	Term	(unused) Stack fault on coprocessor (unused)
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process
SIGURG	16,23,21	Ign	Urgent condition on socket (4.2BSD)
SIGXCPU	24,24,30	Core	CPU time limit exceeded (4.2BSD)
SIGXFSZ	25,25,31	Core	File size limit exceeded (4.2BSD)
SIGVTALRM	26,26,28	Term	Virtual alarm clock (4.2BSD)
SIGPROF	27,27,29	Term	Profiling timer expired
SIGWINCH	28,28,20	Ign	Window resize signal (4.3BSD, Sun)
SIGIO	23,29,22	Term	I/O now possible (4.2BSD)
SIGPWR	29,30,19	Term	Power failure (System V)

The signal Function

The simplest interface to the signal features

Synopsis

- void (*signal(int signo, void (*func)(int)))(int);
- Returns: previous disposition of signal, or -1 on error
- The signo is the name of the signal
- The func is the function to be called when the signal occurs
  - The prototype of the function must be “void funcname(int)”
  - It can also be SIG_IGN or SIG_DFL

The implementation of this function differs among systems

- It is better to use the sigaction function instead.

The signal Function, an Example

```
#include "apue.h"

static void sig_usr(int signo) {       /* argument is signal number */
     if (signo == SIGUSR1)             printf("received SIGUSR1\n");
     else if (signo == SIGUSR2)        printf("received SIGUSR2\n");
     else err_dump("received signal %d\n", signo);
}

int main(void) {
     if (signal(SIGUSR1, sig_usr) == SIG_ERR)
         err_sys("can't catch SIGUSR1");
     if (signal(SIGUSR2, sig_usr) == SIG_ERR)
         err_sys("can't catch SIGUSR2");
     for ( ; ; )
         pause();
}
```

The signal Function, an Example (Cont'd)

```
$ ./a.out &                    start process in background
[1] 7216                       job-control shell prints job number and process ID
$ kill -USR1 7216              send it SIGUSR1
received SIGUSR1
$ kill -USR2 7216              send it SIGUSR2
received SIGUSR2
$ kill 7216                    now send it SIGTERM
[1]+ Terminated ./a.out
```

Signal Setup

When a program is executed, the status of all signals is either default or ignore

When a process calls fork, the child inherits its parent's signal dispositions

The exec functions change the disposition of any signals being caught to their default action

- … and leave the status of all other signals alone
- being caught: has a customized signal handler registered

The shell automatically sets the disposition of the SIGINT and SIGQUIT in the background process to be ignored

Interrupted System Calls

A system call may be interrupted

- If a process caught a signal while the process was blocked in a "slow" system call
- The system call returned an error and errno was set to EINTR

A “slow” system call can block forever, for example

- Read: If data isn't present with certain file types, e.g., pipes, terminals, networks
- Write: If the data can't be accepted immediately
- The pause and wait functions
- Certain ioctl operations
- Some of the interprocess communication functions

Interrupted System Calls (Cont'd)

Problem with interrupted system calls

- We have to handle the error return explicitly
- A typical code sequence

```
again:
    if ((n = read(fd, buf, BUFFSIZE)) < 0) {
        if (errno == EINTR)
            goto again;/* just an interrupted system call */
        /* handle other errors */
    }
```

Prevent Interrupted System Calls

4.2BSD introduced the automatic restarting of certain interrupted system calls

- So, no EINTR error is returned

The system calls that were automatically restarted are ioctl, read, readv, write, writev, wait, and waitpid

This feature can be disabled if you do not like it

Reentrant Functions (1/3)

When a signal that is being caught is handled by a process …

- The normal sequence of instructions being executed is interrupted
- The instructions in the signal handler are now executed
- If the signal handler returns, the process continued the interrupted instructions

What happens if a signal handler use the same system call as the interrupted instructions?

Examples

- The process was in the middle of allocating additional memory on its heap using malloc, and the signal handler calls malloc
- The process was in the middle of generating a temporary filename using tmpnam, and the signal handler calls tmpnam

Reentrant Functions (2/3)

Reentrant of a function may or may not cause problems

There are functions that work with reentrant

- See next page (figure 10.4 from textbook)

Many functions cannot work with reentrant, because …

- They use static data structures (data may be overwritten)
- They are part of the standard I/O library (buffering)
- They call malloc or free

Even if a function is safe with reentrant, there is only one "errno" variable

- #include <errno.h>
- May be not a problem for systems with thread local storage

Reentrant Functions (3/3)

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

Reentrant, an Example

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

void handler(int signo) {
        tmpnam(NULL);
}

int main() {
        int i;
        char *s = tmpnam(NULL);
        signal(SIGALRM, handler);
        alarm(3);
        for(i = 0; i < 6; i++) {
                printf("tmpnam = %s\n", s);
                sleep(1);
        }
        return(0);
}
```

```
$ ./reent
tmpnam = /tmp/filem2ove9
tmpnam = /tmp/filem2ove9
tmpnam = /tmp/filem2ove9
tmpnam = /tmp/fileaczTnb
tmpnam = /tmp/fileaczTnb
tmpnam = /tmp/fileaczTnb
```

SIGCLD and SIGCHLD

In Linux, the two signals are the same

The signal disposition for the two signals are both SIG_DFL

Zombie avoidance

- Although the default action is to ignore the signal, it behaves different when the signal handler is *explicitly* set to SIG_IGN
- No zombie will be created if children of the calling process terminate
- This only works for Linux

Reliable Signal Model

A signal is generated for a process (or sent to a process) when the event that causes the signal occurs

Delivery of a signal

- Delivered: a process received the signal and the action for a signal is taken
- Pending: a signal is generated, but not delivered

Blocking the delivery of a signal

- If a signal is blocked and the signal handler is SIG_DFL or a handler, the signal remains pending until
  - Unblocks the signal, or
  - Change the signal handler to SIG_IGN

Reliable Signal Model (Cont'd)

What happens if a blocked signal is generated more than once before the process unblocks the signal?

- POSIX.1 allows the system to deliver the signal either once or more than once
- Most UNIX systems, however, do not queue signals

What happens if more than one signal is ready to be delivered to a process?

- POSIX.1 does not specify the order in which the signals are delivered
- However, it suggests that signals related to the current state of the process (e.g., SIGSEGV) be delivered before other signals

kill and raise Functions

Send signals

kill – send a signal to a given process ID

raise – send a signal to the calling process itself

Synopsis

- int kill(pid_t pid, int signo);
- int raise(int signo);
- Calling raise(signo) is equivalent to calling kill(getpid(), signo)
- Returns: 0 if OK, -1 on error

kill and raise Functions (Cont'd)

The targeted signal receivers

- Determined by the pid argument
- The signal sender must have permissions to send signals
- pid > 0: the given process ID
- pid == 0: send to all processes in the same group as the caller
- pid < 0: the processes in the given group ID
- pid == -1: send to all processes in the system

Send a NULL signal: kill(pid, 0)

- May be used to check the existence of a process
  - kill returns -1 and errno is set to ESRCH.
  - However, it is unreliable

alarm and pause Functions

alarm

- Set a timer (in the unit of seconds)
- When the timer expires, SIGALRM is generated
- Synopsis: unsigned int alarm(unsigned int seconds);
- Return: 0 or number of seconds until previously set alarm
  - There is only one alarm clocks per process
  - A previously registered alarm clock will be replaced by the new value
- The default action for SIGALRM is “terminate the process”

pause

- Suspend a process until it receives a signal
- Synopsis(): int pause(void);
- It always returns -1 with errno set to EINTR

abort Function

Abnormal program termination

Synopsis

- void abort(void);

This function sends the SIGABRT signal to the caller

SIGABRT can be caught by a signal handler, however

- The signal handler will not return if it calls exit, _exit, or _Exit
- If the signal handler returns, abort terminates the process
- If the signal handler calls siglongjmp, the program may continue to execute

Signal Mask

Each process has a signal mask (in the kernel)

It defines the set of signals currently blocked

Each signal has a corresponding bit in the mask

The mask can be set use the sigprocmask function

Signal Sets

A data type to represent multiple signals

The sigset_t data type

Signal set operations

- int sigemptyset(sigset_t *set); *initialize to contain no signals*
- int sigfillset(sigset_t *set); *initialize to contain all signals*
- int sigaddset(sigset_t *set, int signo); *add a signal into a set*
- int sigdelset(sigset_t *set, int signo); *delete a signal from a set*
- Returns: 0 if OK, -1 on error

Signal set membership test

- int sigismember(const sigset_t *set, int signo);
- Returns: 1 if true, 0 if false, -1 on error

The sigprocmask Function

Block or unblock signals

Synopsis

- int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
- Returns: 0 if OK, -1 on error

Operations (how)

- SIG_BLOCK
  - Blocked signals will be the union of the current signal and the signals in the set
- SIG_UNBLOCK
  - Signals in the set are removed
- SIG_SETMASK
  - Replace the current signal mask with that given in set

The sigpending Function

Get currently pending signals

Synopsis

- int sigpending(sigset_t *set);
- Returns: 0 if OK, -1 on error

An Example

The sig_quit function show a signal caught message and ***reset*** the default SIGQUIT handler

```
$ ./fig10.15-critical
^\^\^\^\^\^\^\^\^\^\
SIGQUIT pending
caught SIGQUIT
SIGQUIT unblocked
^\Quit
```

```
int main(void) {
     sigset_t newmask, oldmask, pendmask;
     if (signal(SIGQUIT, sig_quit) == SIG_ERR)
          err_sys("can't catch SIGQUIT");
     /* Block SIGQUIT and save current signal mask. */
     sigemptyset(&newmask);
     sigaddset(&newmask, SIGQUIT);
     if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
          err_sys("SIG_BLOCK error");
     sleep(5);                                   /* SIGQUIT here will remain pending */
     if (sigpending(&pendmask) < 0)
          err_sys("sigpending error");
     if (sigismember(&pendmask, SIGQUIT))
          printf("\nSIGQUIT pending\n");
     /* Reset signal mask which unblocks SIGQUIT. */
     if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
          err_sys("SIG_SETMASK error");
     printf("SIGQUIT unblocked\n");
     sleep(5);                                   /* SIGQUIT here will terminate with core file */
     exit(0);
}
```

sigsuspend Function (1/3)

```
sigset_t newmask, oldmask;
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);
/* block SIGINT and save current signal mask */
if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
     err_sys("SIG_BLOCK error");
/* critical region of code */
/* reset signal mask, which unblocks SIGINT */
if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
     err_sys("SIG_SETMASK error");
/* window is open */
pause();              /* wait for signal to occur */
/* continue processing */
```

Objective: Block SIGINT for the critical region

What happens if a signal delivered between sigprocmask(SIG_SETMASK) and pause()?

sigsuspend Function (2/3)

We need an atomic operation that performs …

- sigprocmask(SIG_SETMASK): unblocking blocked signals, and
- pause()

The sigsuspend function, synopsis

- int sigsuspend(const sigset_t *sigmask);
- Returns -1 with errno set to EINTR
- The function set the signal mask to “sigmask” and then pause

sigsuspend Function (3/3)

A revised example

```
sigset_t newmask, oldmask;
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);
/* block SIGINT and save current signal mask */
if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
    err_sys("SIG_BLOCK error");
/* critical region of code */
/* reset signal mask, which unblocks SIGINT */
if (sigsuspend(&oldmask) != -1)
    err_sys("sigsuspend error");
/* continue processing */
```

sigsuspend Application: Wait for a Global Variable to be Set

```
volatile sig_atomic_t quitflag;     /* set nonzero by signal handler */
static void sig_int(int signo) {    /* one signal handler for SIGINT and SIGQUIT */
    if (signo == SIGINT)            printf("\ninterrupt\n");
    else if (signo == SIGQUIT)      quitflag = 1;                    /* set flag for main loop */
}
int main(void) {
    sigset_t newmask, oldmask, zeromask;
    if (signal(SIGINT, sig_int) == SIG_ERR)  err_sys("signal(SIGINT) error");
    if (signal(SIGQUIT, sig_int) == SIG_ERR) err_sys("signal(SIGQUIT) error");
    sigemptyset(&zeromask);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);
    /* Block SIGQUIT and save current signal mask. */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0) err_sys("SIG_BLOCK");
    while (quitflag == 0)
        sigsuspend(&zeromask);
    /* SIGQUIT has been caught and is now blocked; do whatever. */
    quitflag = 0;
    /* Reset signal mask which unblocks SIGQUIT. */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0) err_sys("SIG_SETMASK");
}
```

sigsuspend Application: Wait for a Global Variable to be Set (Cont'd)

```
$ ./a.out
^?                  type the interrupt character
interrupt
^?                  type the interrupt character again
interrupt
^?                  and again
interrupt
^?                  and again
interrupt
^?                  and again
interrupt
^?                  and again
interrupt
^?                  and again
interrupt
^\ $                now terminate with quit character
```

sigsuspend Applications: Process Synchronization – Initialize

```
static volatile sig_atomic_t sigflag;          /* set nonzero by sig handler */
static sigset_t newmask, oldmask, zeromask;
static void sig_usr(int s) {       /* signal handler for SIGUSR1 and SIGUSR2 */
    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");
}
```

sigsuspend Applications: Process Synchronization – Parent Goes First

Child waits

```
void WAIT_PARENT(void) {
    while (sigflag == 0)
        sigsuspend(&zeromask);                    /* wait for parent */
    sigflag = 0;
    /* Reset signal mask to original value. */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
}
```

Notify the child

```
void TELL_CHILD(pid_t pid) {
    kill(pid, SIGUSR1);                           /* tell child we're done */
}
```

sigsuspend Applications: Process Synchronization – Child Goes First

Parent waits

```
void WAIT_CHILD(void) {
    while (sigflag == 0)
        sigsuspend(&zeromask);                    /* wait for child */
    sigflag = 0;
    /* Reset signal mask to original value. */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
}
```

Notify the parent

```
void TELL_PARENT(pid_t pid) {
    kill(pid, SIGUSR2);                           /* tell parent we're done */
}
```

The sigaction Function (1/3)

Examine or modify (or both) the action associated with a particular signal

Synopsis

- ```
  int sigaction(int signum,
                const struct sigaction *act,
                struct sigaction *oldact);
  ```
- Return: 0 if OK, -1 on error
  ```

The sigaction Function (2/3)

The sigaction data structure

```
struct sigaction {
        void     (*sa_handler)(int);
        void     (*sa_sigaction)(int, siginfo_t *, void *);
        sigset_t   sa_mask;
        int        sa_flags;
        void     (*sa_restorer)(void);
};
```

- sa_handler – default signal handler
- sa_sigaction – alternative signal handler, when SA_SIGINFO is enabled
- sa_mask – blocked signals when executing the signal handler
- sa_flags – various options

The sigaction Function (3/3)

More on sa_flags

sa_flags	Description
SA_INTERRUPT	This option has been obsoleted. By default, Linux interrupts system calls with EINTR error on receipt of signals
SA_NOCLDSTOP	Do not receive notification when child processes stop
SA_NOCLDWAIT	Do not create zombie processes
SA_RESETHAND	Reset the default signal handler after the signal handler returns
SA_RESTART	Automatically restart some system calls, do not generate EINTR error
SA_NODEFER	By default, when executing the signal handler, the received signal is blocked. This option unblocks the received signal
SA_SIGINFO	Use sa_sigaction as the signal handler

An Implementation of the signal Function

```
#include "apue.h"        /* Reliable version of signal(), using POSIX sigaction(). */
typedef void (*sigfunc)(int);

sigfunc * signal(int signo, sigfunc *func) {
     struct sigaction act, oact;
     act.sa_handler = func;
     sigemptyset(&act.sa_mask);
     act.sa_flags = 0;
     if (signo == SIGALRM) {
#ifdef SA_INTERRUPT
          act.sa_flags |= SA_INTERRUPT;
#endif
     } else {
#ifdef SA_RESTART
          act.sa_flags |= SA_RESTART;
#endif
     }
     if (sigaction(signo, &act, &oact) < 0)
          return(SIG_ERR);
     return(oact.sa_handler);
}
```

sigsetjmp and siglongjmp Functions (1/3)

We know that jump across functions can be performed by setjmp and longjmp

A common scenario is to longjmp to the main loop from a signal handler

However, when a signal is caught, it is added to the *signal mask* automatically

- Prevents the delivery of the same signal from interrupting the signal handler

Although longjmp exits the signal handler, the signal mask remains unchanged – This may cause problems!

sigsetjmp and siglongjmp Functions (2/3)

Solutions to the above problem

- setjmp should save the signal mask
- When longjmp is called, the signal mask should be restored

POSIX.1 does not define how do setjmp and longjmp handle signal masks

- FreeBSD and Mac OS X save and restore signal mask automatically
- However, Linux does not. We have to work with sigsetjmp and siglongjmp

sigsetjmp and siglongjmp Functions (3/3)

Synopsis

- int sigsetjmp(sigjmp_buf env, int savemask);
  - Returns: 0 if called directly, nonzero if returning from a call to siglongjmp
- void siglongjmp(sigjmp_buf env, int val);

There is an additional argument to the sigsetjmp

- savemask: If it is non-zero, the signal mask is preserved (in env) and then restored on siglongjmp

sigsetjmp and siglongjmp Functions, an Example (1/3)

```
static sigjmp_buf               jmpbuf;
static volatile sig_atomic_t    canjump;

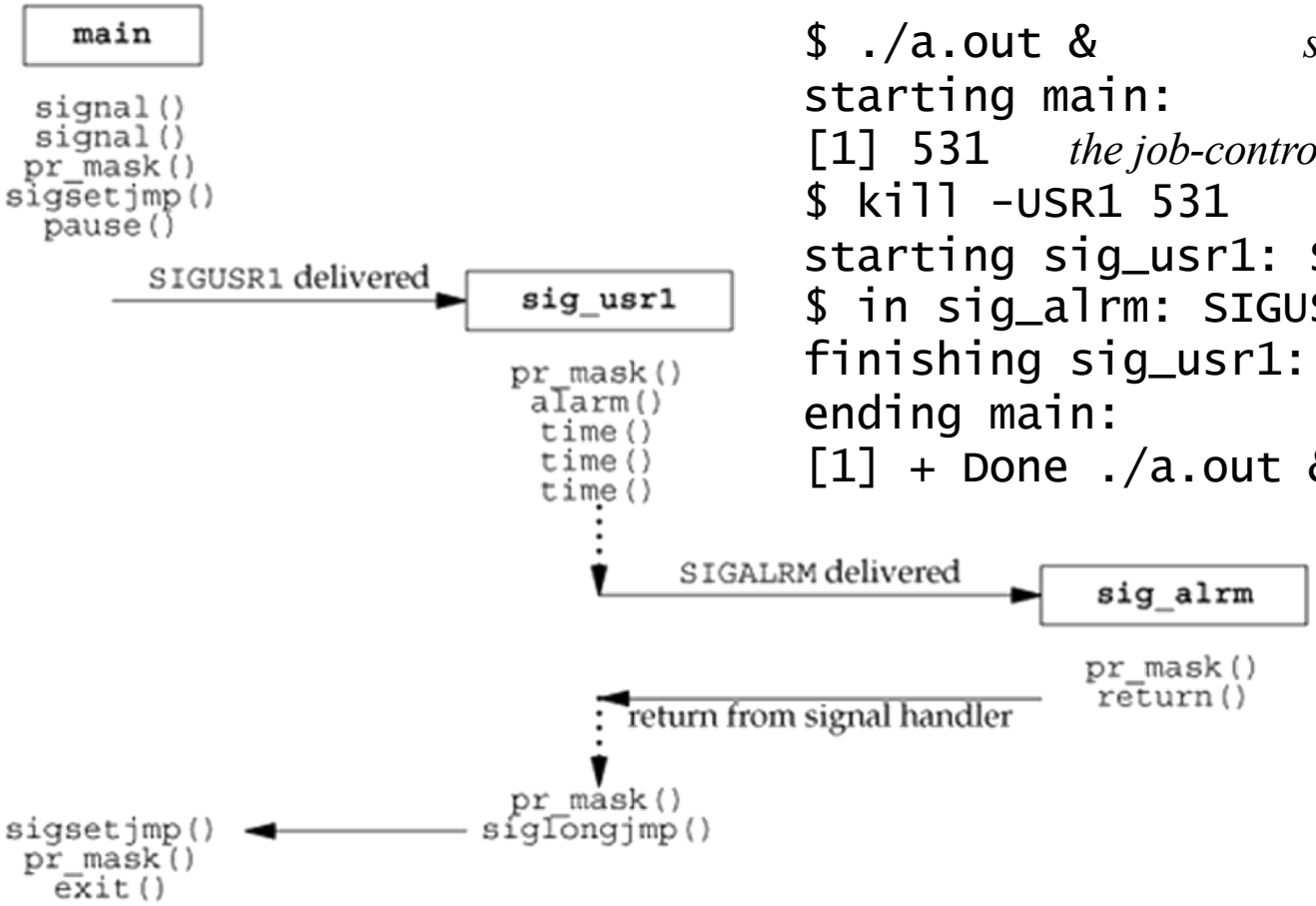
static void sig_alrm(int signo) { pr_mask("in sig_alrm: "); }

static void sig_usr1(int signo) {
    time_t  starttime;
    if (canjump == 0) return;                   /* unexpected signal, ignore */
    pr_mask("starting sig_usr1: ");
    alarm(3);                                   /* SIGALRM in 3 seconds */
    starttime = time(NULL);
    for ( ; ; )                                 /* busy wait for 5 seconds */
       if (time(NULL) > starttime + 5)  break;
    pr_mask("finishing sig_usr1: ");
    canjump = 0;
    siglongjmp(jmpbuf, 1);                  /* jump back to main, don't return */
}
```

sigsetjmp and siglongjmp Functions, an Example (2/3)

```
int main(void) {
    if (signal(SIGUSR1, sig_usr1) == SIG_ERR)
       err_sys("signal(SIGUSR1) error");
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
       err_sys("signal(SIGALRM) error");
    pr_mask("starting main: ");                    /* {Prog prmask} */
    if (sigsetjmp(jmpbuf, 1)) {
       pr_mask("ending main: ");
       exit(0);
    }
    canjump = 1;    /* now sigsetjmp() is OK */
    for ( ; ; ) pause();
}
```

sigsetjmp and siglongjmp Functions, an Example (3/3)




```
$ ./a.out &              start process in background
starting main:
[1] 531      the job-control shell prints its process ID
$ kill -USR1 531         send the process SIGUSR1
starting sig_usr1: SIGUSR1
$ in sig_alrm: SIGUSR1 SIGALRM
finishing sig_usr1: SIGUSR1
ending main:                        just press RETURN
[1] + Done ./a.out &
```

system Function

We have introduced a simple implementation of the system function in Chapter 8

- It does not handle signals

However, POSIX.1 requires the system function to ignore SIGINT/SIGQUIT and block SIGCHLD

Why?

- Suppose the system function creates a child process P
- When running a command using the system function, SIGINT and SIGQUIT should be sent only to process P, not to the calling process
- The SIGCHLD should be blocked so that the calling process will not confuse between the termination of P and other children processes

system Function

Please refer to Figure 10.28 on the textbook

The complete implementation of the system function should …

- Preserve original signal action for SIGINT and SIGQUIT
- Block SIGCHLD
- Fork the child, and, for the child
  - Restore signal handler
  - Unblock SIGCHLD
  - Execute the command /bin/sh -c "…"
- For the parent
  - Wait for the termination of the child
- Finally, restore signal handler and unblock SIGCHLD

Job Control Signals

Signal	Description
SIGCHLD	Child process has stopped or terminated
SIGSTOP	Continue process, if stopped
SIGCONT	Stop signal (can't be caught or ignored)
SIGTSTP	Interactive stop signal
SIGTTIN	Read from controlling terminal by member of a background process group
SIGTTOU	Write to controlling terminal by member of a background process group

Except SIGCHLD, most application programs don't handle these signals

An exception is a process that is managing the terminal

- For example, the vi editor, which is a full screen editor
- It need to save/restore terminal state when the process is stopped/continued

Q & A