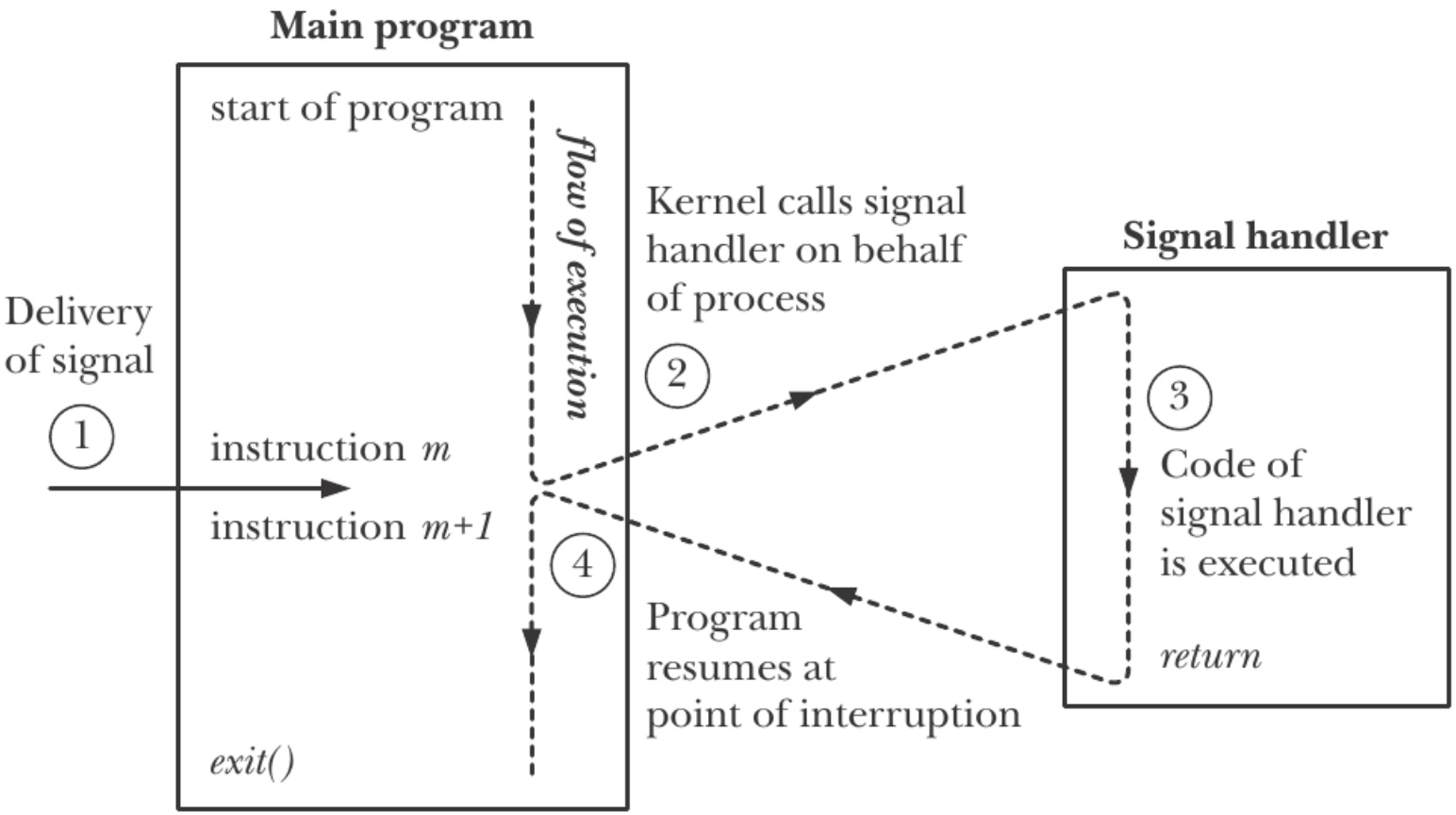
Chapter 10. Signals

朱金辉

华南理工大学软件学院

1. Introduction

- Signals are software interrupts
- Handles asynchronous events



2. Signal Concepts

- Every signal has a name
- All begin with SIG, for example:
  - SIGABRT: abort signal from abort()
  - SIGALRM: alarm signal from alarm()
- Signal Names = positive integer constants
- #include <signal.h>

Signal Generation

1. Terminal-generated signals: SIGINT
2. Hardware exceptions:
   - SIGFPE: divide by 0
   - SIGSEGV: invalid memory reference
3. kill command: interface to kill()
4. Software conditions:
   - SIGURG: out-of-band network data
   - SIGPIPE: pipe-write after pipe reader is terminated
   - SIGALRM: alarm clock expires

Signal Disposition (Action)

- 1) IGNORE SIGNAL: all signals can be ignored, except SIGKILL and SIGSTOP
  - superuser can kill or stop a process, or
  - hardware exceptions leave process behavior undefined if signals ignored

Signal Disposition (Action)

- 2) CATCH SIGNAL: Call a function of ours when a signal occurs.
  - Own shell: SIGINT ➔ return to main()
  - Child terminated: SIGCHLD ➔ waitpid()
  - Temporary files: SIGTERM ➔ clean up
- 3) DEFAULT ACTION: most are to terminate process (see next Figure!)

UNIX Signals and Action

Name	Description	ISO C	SUS	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	Default action
SIGABRT	abnormal termination (abort)	•	•	•	•	•	•	terminate+core
SIGALRM	timer expired (alarm)		•	•	•	•	•	terminate
SIGBUS	hardware fault		•	•	•	•	•	terminate+core
SIGCANCEL	threads library internal use						•	ignore
SIGCHLD	change in status of child		•	•	•	•	•	ignore
SIGCONT	continue stopped process		•	•	•	•	•	continue/ignore
SIGEMT	hardware fault			•	•	•	•	terminate+core
SIGFPE	arithmetic exception	•	•	•	•	•	•	terminate+core
SIGFREEZE	checkpoint freeze						•	ignore
SIGHUP	hangup		•	•	•	•	•	terminate
SIGILL	illegal instruction	•	•	•	•	•	•	terminate+core
SIGINFO	status request from keyboard			•		•		ignore
SIGINT	terminal interrupt character	•	•	•	•	•	•	terminate
SIGIO	asynchronous I/O			•	•	•	•	terminate/ignore
SIGIOT	hardware fault			•	•	•	•	terminate+core
SIGKILL	termination		•	•	•	•	•	terminate
SIGLWP	threads library internal use						•	ignore
SIGPIPE	write to pipe with no readers		•	•	•	•	•	terminate

UNIX Signals and Action

Name	Description	ISO C	SUS	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	Default action
SIGPOLL	pollable event (poll)		XSI		•		•	terminate
SIGPROF	profiling time alarm (setitimer)		XSI	•	•	•	•	terminate
SIGPWR	power fail/restart				•		•	terminate/ignore
SIGQUIT	terminal quit character		•	•	•	•	•	terminate+core
SIGSEGV	invalid memory reference	•	•	•	•	•	•	terminate+core
SIGSTKFLT	coprocessor stack fault				•			terminate
SIGSTOP	stop		•	•	•	•	•	stop process
SIGSYS	invalid system call		XSI	•	•	•	•	terminate+core
SIGTERM	termination	•	•	•	•	•	•	terminate
SIGTHAW	checkpoint thaw						•	ignore
SIGTRAP	hardware fault		XSI	•	•	•	•	terminate+core
SIGTSTP	terminal stop character		•	•	•	•	•	stop process
SIGTTIN	background read from control tty		•	•	•	•	•	stop process
SIGTTOU	background write to control tty		•	•	•	•	•	stop process
SIGURG	urgent condition (sockets)		•	•	•	•	•	ignore
SIGUSR1	user-defined signal		•	•	•	•	•	terminate
SIGUSR2	user-defined signal		•	•	•	•	•	terminate
SIGVTALRM	virtual time alarm (setitimer)		XSI	•	•	•	•	terminate
SIGWAITING	threads library internal use						•	ignore
SIGWINCH	terminal window size change			•	•	•	•	ignore
SIGXCPU	CPU limit exceeded (setrlimit)		XSI	•	•	•	•	terminate+core/ignore
SIGXFSZ	file size limit exceeded (setrlimit)		XSI	•	•	•	•	terminate+core/ignore
SIGXRES	resource control exceeded						•	ignore

3. signal Function

- #include <signal.h>
- void (*signal(int *signo*, void (**func*)(int))) (int);
- Returns: previous disposition of signal if OK, SIG_ERR on error
- signo: SIGXXXX signal name
- func:
  - SIG_IGN, or SIG_DFL, or user-defined function

```
typedef void Sigfunc(int);
Sigfunc *signal(int, Sigfunc *);
```

Program 10.2: signal Function

```
#include      <signal.h>
#include      "apue.h"
static void   sig_usr(int); /* one handler for both signals */
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();
}
```

Program 10.2: signal Function

```
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);
   return;
}
```

Program 10.2: results

- **$ a.out &**
- [1] 4720
- **$ kill –USR1 4720**
- received SIGUSR1
- **$ kill –USR2 4720**
- received SIGUSR2
- **$ kill 4720**
- [1] + Terminateda.out &

4. Unreliable Signals

- In Early systems of UNIX, signals were unreliable.
- The signals could get lost:

  a signal could occur and the process would never know about it.
- A process had little control over a signal:

  it could catch the signal or ignore it.
  - But sometimes,we would like to tell the kernel to block a signal: don't ignore it ,just remember if it occurs,and tell us later when we're ready.

5. Interrupted System Calls

- A process is blocked in a "slow" device (a system call)
- The process receives a signal
- The system call is interrupted and returns an error (errno = EINTR)

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

Interrupted System Calls

- To prevent applications from having to handle interrupted system calls, 4.2BSD introduced the automatic restarting of certain interrupted system calls.

Functions	System	Signal handler remains installed	Ability to block signals	Automatic restart of interrupted system calls?
`signal`	ISO C, POSIX.1	unspecified	unspecified	unspecified
	V7, SVR2, SVR3			never
	SVR4, Solaris			never
	4.2BSD	•	•	always
	4.3BSD, 4.4BSD, FreeBSD, Linux, Mac OS X	•	•	default
`sigaction`	POSIX.1, 4.4BSD, SVR4, FreeBSD, Linux, Mac OS X, Solaris	•	•	optional

6. Reentrant Functions

```
#include "apue.h"
#include <pwd.h>

static void
my_alarm(int signo)
{
    struct passwd   *rootptr;

    printf("in signal handler\n");
    if ((rootptr = getpwnam("root")) == NULL)
            err_sys("getpwnam(root) error");
    alarm(1);
}

int
main(void)
{
    struct passwd   *ptr;

    signal(SIGALRM, my_alarm);
    alarm(1);
    for ( ; ; ) {
        if ((ptr = getpwnam("sar")) == NULL)
            err_sys("getpwnam error");
        if (strcmp(ptr->pw_name, "sar") != 0)
            printf("return value corrupted!, pw_name = %s\n",
                    ptr->pw_name);
    }
}
```

When this program was run, the results were random.

Reentrant Functions

- Functions that can be called by two or more processes (tasks, signal handlers), with arbitrary preemption (interrupt), and still give the same predictable output results.

- NOT Reentrant function， when
  - Use static variables,
  - Call malloc or free,
  - belong to standard I/O library(use global data structure)

Reentrant functions that may be called from a signal handler

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

8. Reliable Signal Terminology and Semantics

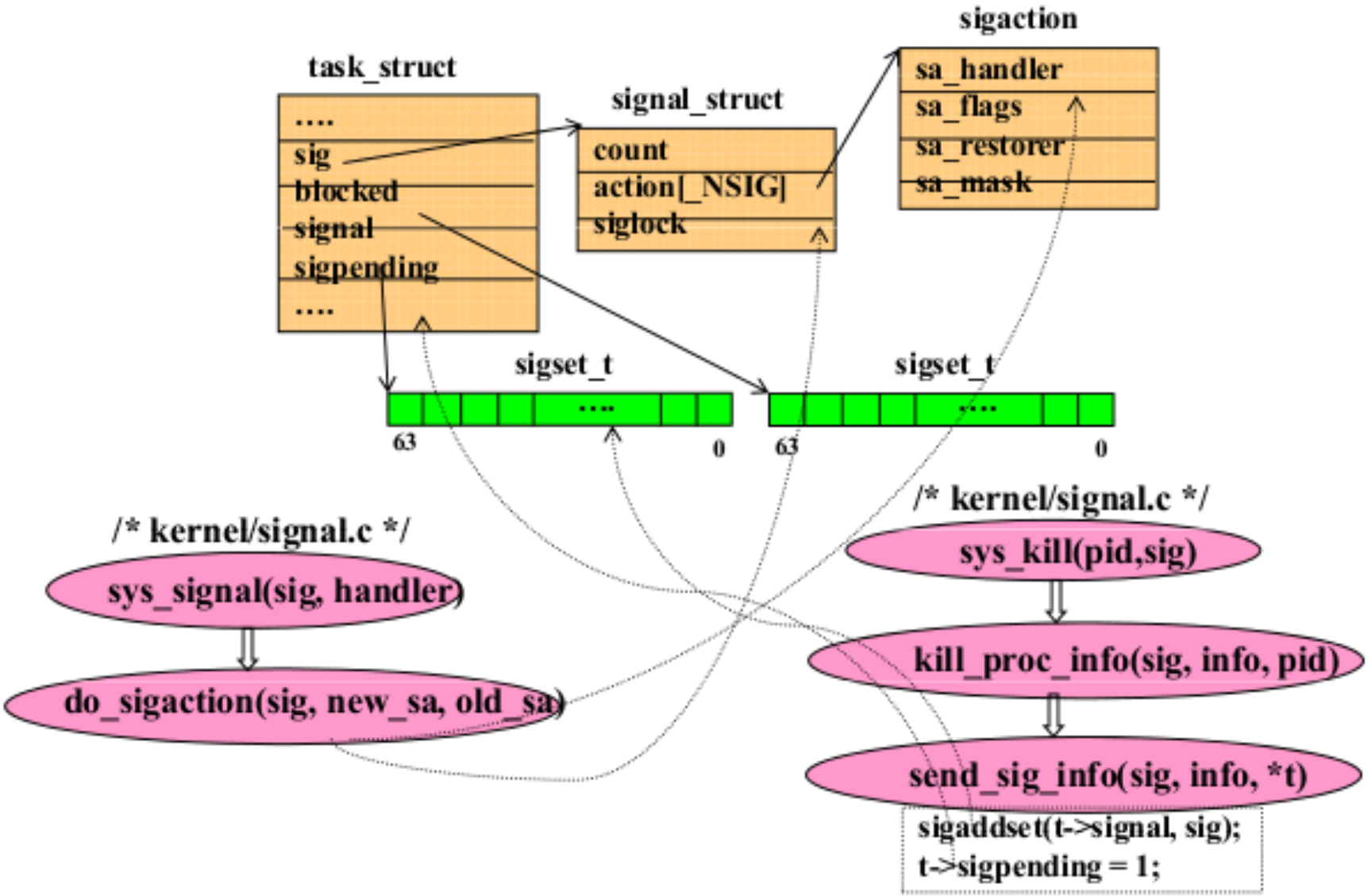
- Signal is **GENERATED** when event that causes the signal occurs, for example
  - hardware exception (divide by 0)
  - software condition (alarm timer expiring)
  - terminal-generated signal
  - call to kill function
- Kernel sets a flag in the process table indicating that the signal is generated

Reliable Signal Terminology and Semantics

- Signal is DELIVERED to a process if action for signal is taken
- Between generation and delivery, signal is called PENDING
- A process can BLOCK the delivery of a signal using SIGNAL MASK
- Signal mask can be changed by sigprocmask() (see Section 10.12)

Reliable Signal Terminology and Semantics

- sigpending() function is used to determine blocked and pending signals
- More than one blocked signal?
  - Queued? No! Just once!
  - SIGSEGV delivered first
- sigset_t: data type to store signal mask (#bits = #signals, 1 means blocked)


task_struct
....
sig
blocked
signal
sigpending
....
signal_struct
count
action[_NSIG]
siglock
sigaction
sa_handler
sa_flags
sa_restorer
sa_mask
sigset_t
....
63
0
sigset_t
....
63
0
/* kernel/signal.c */
sys_signal(sig, handler)
do_sigaction(sig, new_sa, old_sa)
/* kernel/signal.c */
sys_kill(pid,sig)
kill_proc_info(sig, info, pid)
send_sig_info(sig, info, *t)
sigaddset(t->signal, sig);
t->sigpending = 1;

9. kill and raise functions

- kill: To send a signal to a process or a process group
- raise: To send a signal to calling process (itself)

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int signo);
int raise(int signo);
```

- Both return: 0 if OK, -1 on error

kill(pid_t *pid*, int *signo*)

- pid > 0: sent to PID==pid
- pid < 0: sent to PGID==|pid|
- pid == 0: sent to all processes with PGID == PGID of sender (with perm)
- pid == -1: not defined in POSIX.1 (Broadcast signals in SVR4, BSD)

10. alarm Function

- alarm() sets a timer to expire at a specified time in future
- when timer expires, SIGALRM signal is generated
- default action: terminate process

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

- Returns: 0 or #seconds until previously set alarm

pause Function

- Suspends a process until a signal is caught
- #include <unistd.h>
- int pause(void);
- Returns: -1 with errno set to EINTR
- pause returns only if a signal handler is executed and that handler returns!

11. Signal Sets

- Signal set = a set of signals
- POSIX.1: sigset_t (data type to represent multiple signals)

```
#include <signal.h>
int sigemptyset (sigset_t *set);
int sigfillset (sigset_t *set);
int sigaddset (sigset_t *set, int signo);
int sigdelset (sigset_t *set, int signo);
        Return: 0 if OK, -1 on error
int sigismember (const sigset_t *set, int signo);
        Returns: 1 if true, 0 if false
```

12. sigprocmask Function

- Examine or change signals to be blocked

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set,
                sigset_t *oset);
```

- Returns: 0 if OK, -1 on error
- oset != NULL ➔ current mask returned in oset
- set != NULL ➔ current mask modified …

sigprocmask Function

- How to modify? (set != NULL, how = …)

How	Description
SIG_BLOCK	Union of current signal mask and signal mask
SIG_UNBLOCK	Intersection of current signal mask and signal mask
SIG_SETMASK	New signal mask

Program 10.14: print mask

```
#include <errno.h>
#include <signal.h>
#include "apue.h"

void pr_mask(const char *str) {
    sigset_t      sigset;
    int           errno_save;

    errno_save = errno;    /* we can be called by signal handlers */
    if (sigprocmask(0, NULL, &sigset) < 0)
         err_sys("sigprocmask error");

    printf("%s", str);
    if (sigismember(&sigset, SIGINT))       printf("SIGINT ");
    if (sigismember(&sigset, SIGQUIT))      printf("SIGQUIT ");
    if (sigismember(&sigset, SIGUSR1))      printf("SIGUSR1 ");
    if (sigismember(&sigset, SIGALRM))      printf("SIGALRM ");
         /* remaining signals can go here */
    printf("\n");
    errno = errno_save;
}
```

13. sigpending Function

- sigpending returns the set of signals that are blocked and pending

```
#include <signal.h>

int sigpending(sigset_t *set);
```

- Returns: 0 if OK, -1 on error

Program 10.15: sigpending

```
#include <signal.h>
#include "apue.h"

static void      sig_quit(int);

int main(void) {
   sigset_t      newmask, oldmask, pendmask;

   if (signal(SIGQUIT, sig_quit) == SIG_ERR)
        err_sys("can't catch SIGQUIT");

   sigemptyset(&newmask);
   sigaddset(&newmask, SIGQUIT);
                 /* block SIGQUIT and save current signal mask */
   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");
```

Program 10.15: sigpending

```
    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);
}

static void sig_quit(int signo) {
    printf("caught SIGQUIT\n");

    if (signal(SIGQUIT, SIG_DFL) == SIG_ERR)
          err_sys("can't reset SIGQUIT");
    return;
}
```

Program 10.15: results

```
$ a.out
^\          (quit terminal)
SIGQUIT pending
caught SIGQUIT
SIGQUIT unblocked
^\Quit(coredump)
$ a.out
^\^\^\^\^\^\
SIGQUIT pending
caught SIGQUIT
SIGQUIT unblocked
^\Quit(coredump)
```

signals are not queued on this system.

14. sigaction

- Examine or modify a signal action
- Reliable Signals!
- Replace signal function
- #include <signal.h>
- int sigaction(int signo, const struct sigaction *act, struct sigaction *oact);

- Returns: 0 if OK, -1 on error

sigaction

```
struct sigaction {
   void (*sa_handler)(); /*signal handler*/
   sigset_t sa_mask; /* additional signals
                                    to block */
   int sa_flags; /* signal options */
};
```

sa_flags

Option	SUS	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	Description
SA_INTERRUPT			•			System calls interrupted by this signal are not automatically restarted (the XSI default for sigaction). See Section 10.5 for more information.
SA_NOCLDSTOP	•	•	•	•	•	If signo is SIGCHLD, do not generate this signal when a child process stops (job control). This signal is still generated, of course, when a child terminates (but see the SA_NOCLDWAIT option below). As an XSI extension, SIGCHLD won't be sent when a stopped child continues if this flag is set.
SA_NOCLDWAIT	XSI	•	•	•	•	If signo is SIGCHLD, this option prevents the system from creating zombie processes when children of the calling process terminate. If it subsequently calls wait, the calling process blocks until all its child processes have terminated and then returns −1 with errno set to ECHILD. (Recall Section 10.7.)
SA_NODEFER	XSI	•	•	•	•	When this signal is caught, the signal is not automatically blocked by the system while the signal-catching function executes (unless the signal is also included in sa_mask). Note that this type of operation corresponds to the earlier unreliable signals.
SA_ONSTACK	XSI	•	•	•	•	If an alternate stack has been declared with sigaltstack(2), this signal is delivered to the process on the alternate stack.
SA_RESETHAND	XSI	•	•	•	•	The disposition for this signal is reset to SIG_DFL, and the SA_SIGINFO flag is cleared on entry to the signal-catching function. Note that this type of operation corresponds to the earlier unreliable signals. The disposition for the two signals SIGILL and SIGTRAP can't be reset automatically, however. Setting this flag causes sigaction to behave as if SA_NODEFER is also set.
SA_RESTART	XSI	•	•	•	•	System calls interrupted by this signal are automatically restarted. (Refer to Section 10.5.)
SA_SIGINFO	•	•	•	•	•	This option provides additional information to a signal handler: a pointer to a siginfo structure and a pointer to an identifier for the process context.

15. sigsetjmp, siglongjmp

- Similar to setjmp and longjmp
- Difference: saves mask and restores it

```
#include <setjmp.h>
int sigsetjmp(sigjmp_buf env,
              int savemask);
```

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

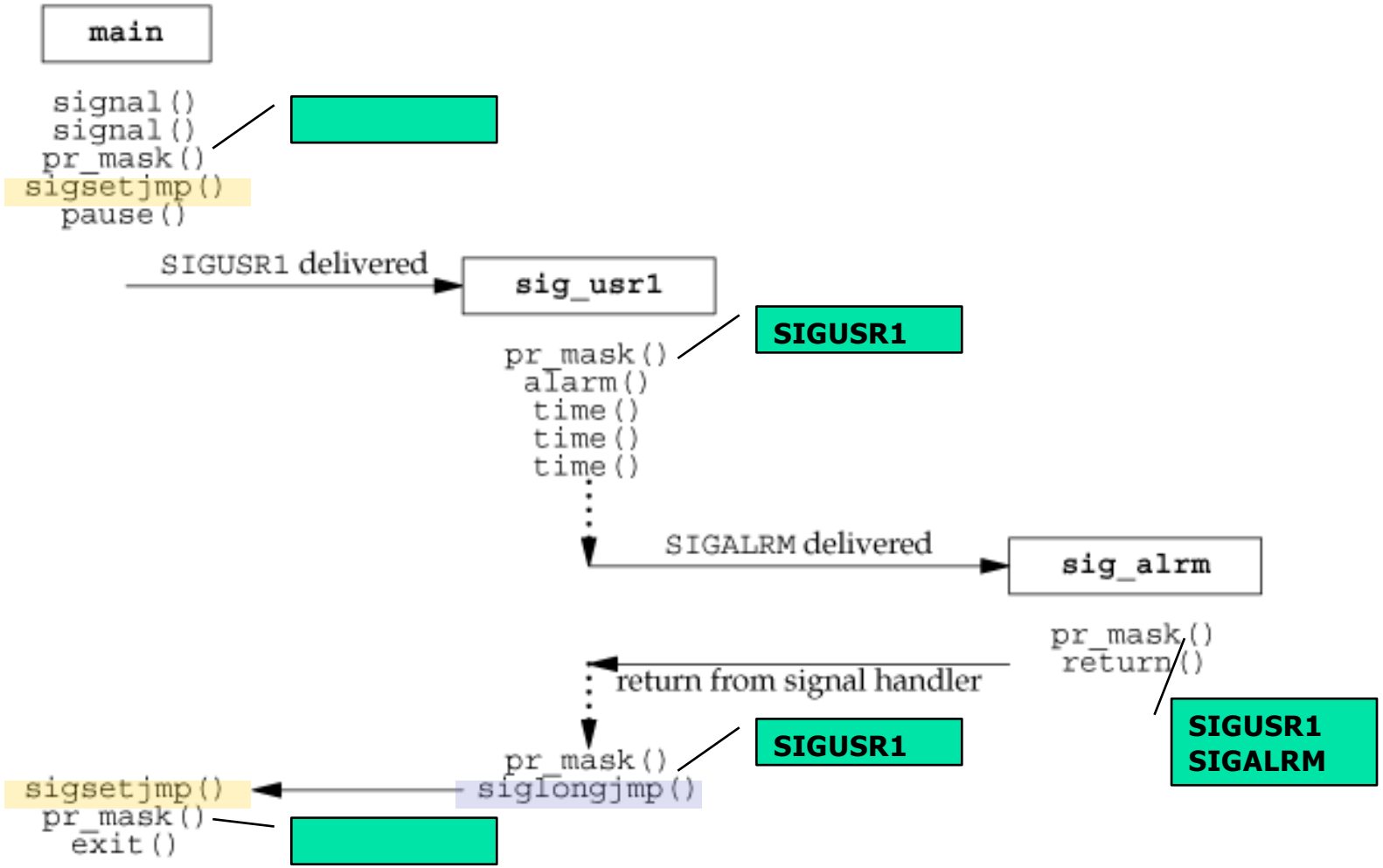
Program 10.20: jmp functions

```
#include <signal.h>
#include <setjmp.h>
#include <time.h>
#include "apue.h"

static void                sig_usr1(int), sig_alrm(int);
static sigjmp_buf          jmpbuf;
static volatile sig_atomic_t      canjump;

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();
}
```

Program 10.20: jmp functions

```
static void  sig_usr1(int signo) {
    time_tstarttime;

    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 */
}

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

Time line for Program 10.20



Program 10.20: results

- $ **a.out &**
- starting main:
- [1] 531
- $ **kill –USR1 531**
- starting sig_usr1: SIGUSR1
- $ in sig_alrm: SIGUSR1 SIGALRM
- finishing sig_usr1: SIGUSR1
- ending main:
- [1] + Done a.out &

16. sigsuspend Function

```
sigset_t newmask, oldmask;
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);
if(sigprocmask(SIG_BLOCK, &newmask, &oldmask) , 0)
   err_sys("SIG_BLOCK error");
/* CRITICAL REGION OF CODE */
…
if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
  err_sys("SIG_SETMASK error");
/* Bug: what is SIGINT arrives now … */
pause();
```

sigsuspend Function

- set signal mask & put process to sleep for a signal (1 atomic operation)

```
sigpromask(SIG_SETMASK, &mask, &prevMask); //assign new mask
pause();
sigprocmask(SIG_SETMASK, &preMask, NULL); //restore old mask
```

- #include <signal.h>

  int sigsuspend(const sigset_t *sigmask);

- Returns: -1 with errno set to EINTR

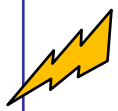
Sigsuspend details

- temporarily replaces the signal mask of the calling process with the mask given by mask , then suspends the process until delivery of a signal whose action is to invoke a signal handler or to terminate a process.
- If the signal terminates the process, then sigsuspend() does not return.
- If the signal is caught, then sigsuspend() returns after the signal handler returns, and the signal mask is restored to the state before the call to sigsuspend().
- It is not possible to block SIGKILL or SIGSTOP; specifying these signals in mask, has no effect on the process's signal mask.

sigsuspend example Fig 10-23

```
#include "apue.h”

volatile sig_atomic_t quitflag; /* set nonzero by signal handler*/

static voidsig_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()
{
   …
   while(quitflag==0)
      sigsuspend(&zeromask);
   quitflag=0;
   …
}
```



**Ctrl **

```
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 error");

  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 error");
  exit(0);
}
```

```
static voidsig_int(int signo)
{
  if (signo == SIGINT)
    printf("\ninterrupt\n");
  else if (signo == SIGQUIT)
    quitflag = 1;
}
```

Program 10.24: check textbook

- Using signals to implement parent/child synchronization (Section 8.9)
- TELL_WAIT
- TELL_PARENT
- TELL_CHILD
- WAIT_PARENT
- WAIT_CHILD