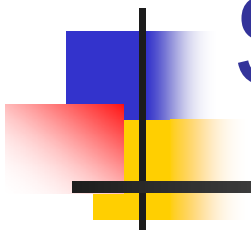


Chapter 10. Signals

System Programming

http://www.cs.ccu.edu.tw/~pahsiung/courses/sp

熊博安

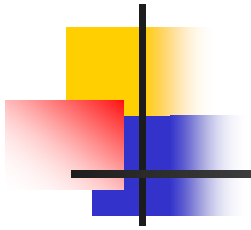
國立中正大學資訊工程學系

pahsiung@cs.ccu.edu.tw

(05)2720411 ext. 33119

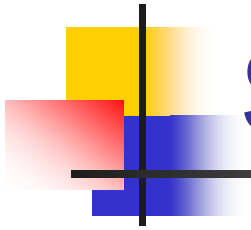
Class: EA-104

Office: EA-512



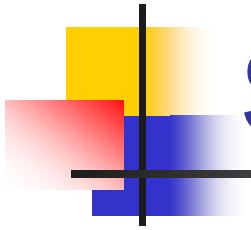
Introduction

- Signals are software interrupts
- Handles asynchronous events
  - user typing CTRL-C to stop a program
  - next program in pipeline terminated prematurely
- Earlier implementations of signals were not reliable, POSIX.1 standardized them!



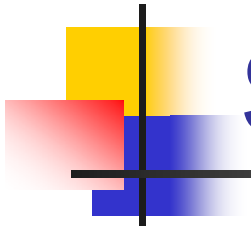
Signal Concepts

- Every signal has a name
- All begin with SIG
  - SIGABRT: abort signal from abort()
  - SIGALRM: alarm signal from alarm()
- SVR4, 4.4BSD: 31 different signals
- FreeBSD 5.21, Mac OS X 10.3, Linux 2.4.22: 31 different signals
- Solaris 9: 38 different signals



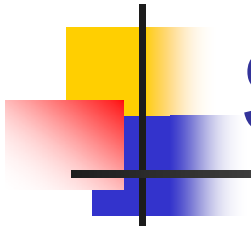
Signal Concepts

- Signal Names = +ve integer constants
- #include <signal.h>
- Signal Number 0 = NULL signal (used by kill)



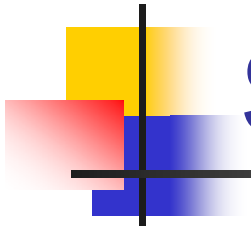
Signal Generation

- Terminal-generated signals: SIGINT
- Hardware exceptions:
  - SIGFPE: divide by 0
  - SIGSEGV: invalid memory reference
- kill(): send any signal to a process or process group (owner, superuser!)
- kill command: interface to kill()
- Software conditions:
  - SIGURG: out-of-band network data
  - SIGPIPE: pipe-write after pipe reader is terminated
  - SIGALRM: alarm clock expires



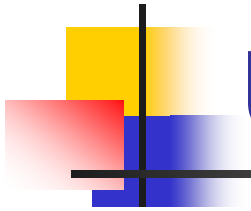
Signal Disposition (Action)

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



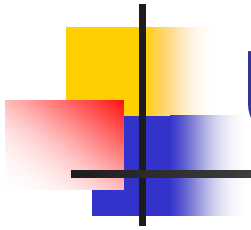
Signal Disposition (Action)

- 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
- DEFAULT ACTION: most are to terminate process (see next Figure!)



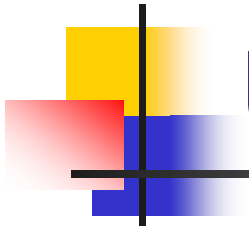
UNIX Signals

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
SIGPOLL	pollable event (poll)		XSI		•		•	terminate
SIGPROF	profiling time alarm (setitimer)		XSI	•	•	•	•	terminate



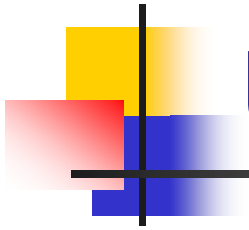
UNIX Signals

- SIGABRT: signal from abort(), process terminates abnormally
- SIGALRM: timer set with alarm() expires
- SIGBUS: hardware fault (memory)
- SIGCANCEL: used by Solaris thread library, not for general use
- SIGCHLD: child terminated, default action = ignore, catch using waipid()



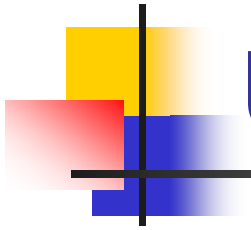
UNIX Signals

- SIGCONT: sent to a stopped process to continue (default action), vi catches this signal to redraw terminal screen
- SIGEMT: hardware fault (emulator trap)
- SIGFPE: arithmetic exception (divide by 0, floating point error, …)
- SIGFREEZE: used by Solaris to notify processes to take special actions before system freeze



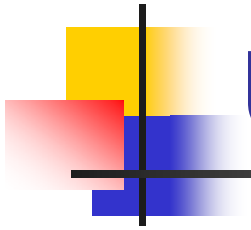
UNIX Signals

- SIGHUP: disconnect detected, session leader terminates, daemon processes reread configuration files
- SIGILL: Illegal hardware instruction
- SIGINFO: Terminal driver generates signal when we type status key (CTRL-T)
- SIGINT: Terminal driver generates signal when we type interrupt key (CTRL-C)
- SIGIO: Asynchronous I/O event



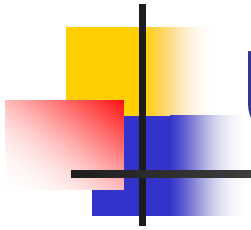
UNIX Signals

- SIGIOT: hardware fault (I/O trap)
- SIGKILL: to kill process by admin (cannot be caught or ignored)
- SIGLWP: used by Solaris thread library
- SIGPIPE: pipe write after pipe reader has terminated (same for socket)
- SIGPOLL: Specific event on pollable device



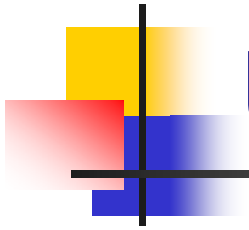
UNIX Signals

- SIGPROF: Profiling interval timer (set by setitimer) has expired
- SIGPWR: uninterruptible power supply (UPS) notifies a process of low power condition,
  - process then sends SIGPWR to init,
  - init handles system shutdown,
  - 2 entries in inittab: powerfail, powerwait



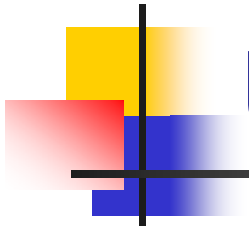
UNIX Signals

- SIGQUIT: Terminal driver generates signal when we type quit key (CTRL-\)
- SIGSEGV: invalid memory reference
- SIGSTKFLT: used by Linux (no more!)
- SIGSTOP: Job-control signal to stop process, cannot be caught or ignored
- SIGSYS: invalid system call
- SIGTERM: termination signal sent by kill



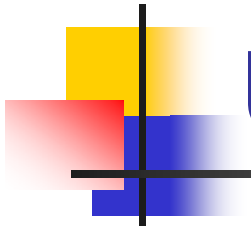
UNIX Signals

- SIGTHAW: used by Solaris to notify processes to take action after resuming from suspension
- SIGTRAP: hardware fault (trap to debugger)
- SIGTSTP: Terminal driver generates signal when we type suspend key (CTRL-Z)
- SIGTTIN: background process tries to read from controlling terminal
- SIGTTOU: background process tries to write to controlling terminal



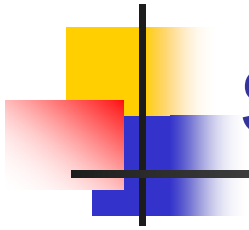
UNIX Signals

- SIGURG: urgent condition has occurred (out-of-band network data)
- SIGUSR1: user-defined for API
- SIGUSR2: user-defined for API
- SIGVTALRM: virtual interval timer set by setitimer(2) expired
- SIGWAITING: used by Solaris thread library
- SIGWINCH: ioctl() changes window size



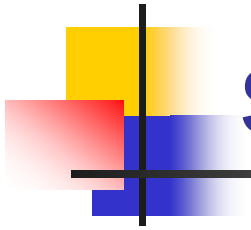
UNIX Signals

- SIGXCPU: process exceeds soft CPU time limit
- SIGXFSZ: process exceeds soft file size limit
- SIGRES: used only by Solaris for resource control



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 (signal handler)



signal Function

- signal() prototype can be simplified as:
  - **`typedef void Sigfunc(int);`**
  - **`Sigfunc *signal(int, Sigfunc *);`**
- In <signal.h>
  - **`#define SIG_ERR (void(*)())-1`**
  - **`#define SIG_DFL (void(*)())0`**
  - **`#define SIG_IGN (void(*)())1`**

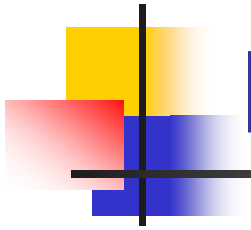


Figure 10.2: signal Function

```
#include    “apue.h"

static void sig_usr(int); /*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();
}
```

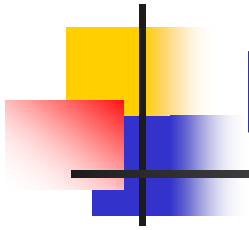


Figure 10.2: signal Function

```
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;
}
```

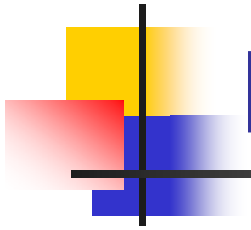
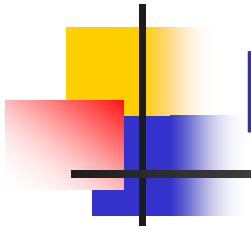


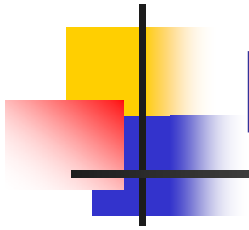
Figure 10.2: results

- **$ ./a.out &**
- [1] 7216
- **$ kill –USR1 7216**
- received SIGUSR1
- **$ kill –USR2 7216**
- received SIGUSR2
- **$ kill 7216** (SIGTERM not caught)
- [1] + Terminated ./a.out



Program Start-up

- exec
  - signals being caught → default action (old handler has no meaning in new program)
  - all other signals → left alone
- cc main.c &
  - Shell without job control, sets the following:
    - ignore SIGINT
    - ignore SIGQUIT

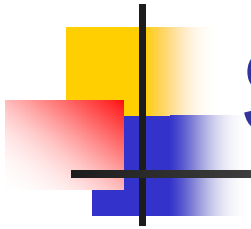


Program Start-up

Interactive programs:

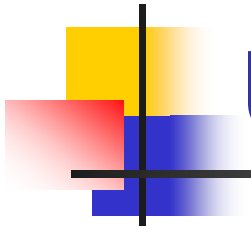
```
int sig_int(), sig_quit();

if (signal(SIGINT, SIG_IGN) != SIG_IGN)
   signal(SIGINT, sig_int);
if (signal(SIGQUIT, SIG_IGN) != SIG_IGN)
   signal(SIGQUIT, sig_quit);
```

- **Catches the signal only if the signal is not currently being IGNORED.**
- **signal() cannot check the current signal disposition, without changing it.**



Signal disposition on fork

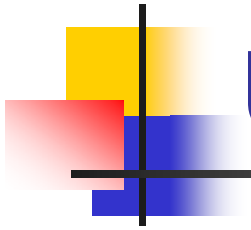
- On fork:
  - Child inherits parent’s signal dispositions
  - Old signal handler has meaning in child



Unreliable Signals

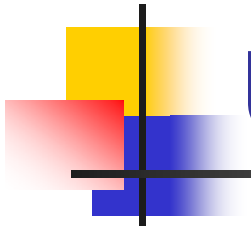
- Early systems: signal disposition set to default after catching it once, need to reestablish handler

```
int sig_int(); …
signal(SIGINT, sig_int); …
sig_int() {
   signal(SIGINT, sig_int);
        /* reestablish handler */
  … }
```



Unreliable Signals

- What if a new interrupt occurs after an interrupt occurred and signal handler not yet re-established?
- Default action: terminate process!



Unreliable Signals

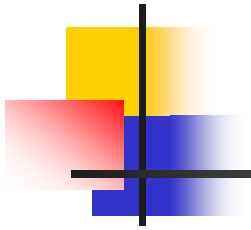
- Sometimes we need to: “Prevent the following signals from occurring, but remember if they do occur.”
- If a signal occurs, set a flag to let the process know about the occurrence of the signal.

Unreliable Signals

```
int sig_int_flag;
main() {
   int sig_int(); …
   signal(SIGINT, sig_int); …
   while (sig_int_flag == 0)   pause(); …
}
sig_int() {
   signal(SIGINT, sig_int);
   sig_int_flag = 1;
}
```

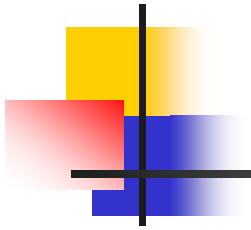
What if signal occurs here?

Signal is LOST! Program will pause forever if signal occurs only once!



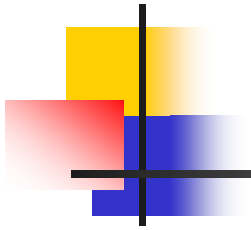
Interrupted System Calls

- 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)
- May be something happened that should wake up the blocked system call



Interrupted System Calls

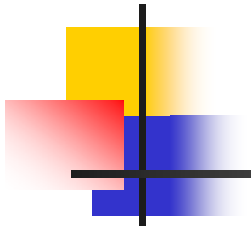
- System calls divided into 2 categories:
  - slow:
    - reads from or writes to files that can block caller forever (pipes, terminals, network devs)
    - opens of files that block until some condition occurs (terminal waiting on modem answer)
    - pause, wait
    - some ioctl operations
    - interprocessor communication
  - all others: disk I/O, etc.



Interrupted System Calls

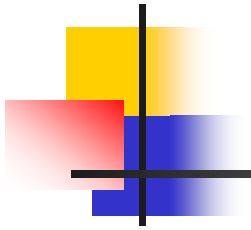
- Need to restart an interrupted system call

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



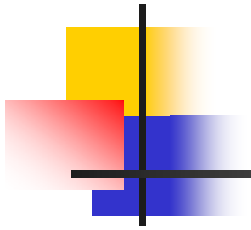
Interrupted System Calls

- 4.2 BSD
  - Always restarted interrupted system calls
  - ioctl, read, readv, write, writev, wait, waitpid
- 4.3 BSD
  - By default restarts interrupted system calls
  - Can be disabled on a per signal basis
- System V
  - Never restarted system calls by default



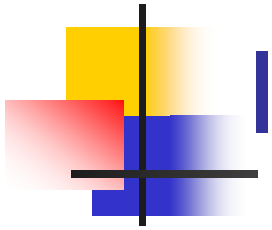
Interrupted System Calls

- POSIX.1
  - Allows implementation to restart interrupted system calls, but not required
- Single UNIX Spec (XSI extension)
  - SA_RESTART flag to allow applications to request restart of interrupted system calls



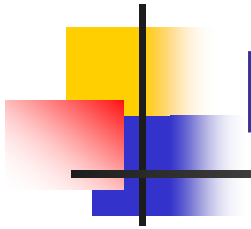
Interrupted System Calls

- FreeBSD 5.2.1, Linux 2.4.22, Mac OS X 10.3
  - Restart system calls interrupted by signals
- Solaris 9
  - Returns error (EINTR), does not restart system calls



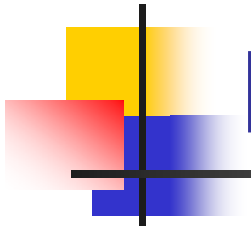
Interrupted System Calls

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, SVR4, Solaris			never
	4.2BSD	•	•	always
	4.3BSD, 4.4BSD, FreeBSD, Linux, Mac OS X	•	•	default
sigset	XSI	•	•	unspecified
	SVR3, SVR4, Linux, Solaris	•	•	never
sigvec	4.2BSD	•	•	always
	4.3BSD, 4.4BSD, FreeBSD, Mac OS X	•	•	default
sigaction	POSIX.1	•	•	unspecified
	XSI, 4.4BSD, SVR4, FreeBSD, Mac OS X, Linux, Solaris	•	•	optional



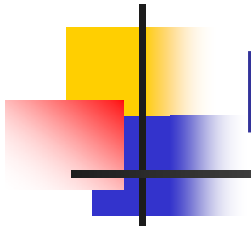
Reentrant Functions

- Process is executing
- Signal arrives and is caught
- Signal handler starts executing
- Signal handler returns
- Process resumes execution
- Any problem???



Reentrant Functions

- What if process was in the middle of malloc(), signal handler starts executing and also calls malloc()?
- malloc() maintains a linked list of all allocated areas
- process may have been in the middle of updating the linked list!

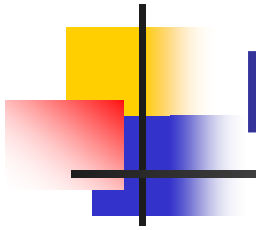


Reentrant Functions

- 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.

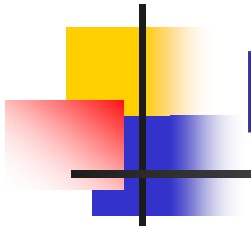
Reentrancy Conditions

- Use static variables in an atomic way,
- Do not call malloc or free,
- Does not belong to standard I/O library



Reentrant Functions

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 Functions

- Signal handlers should only call reentrant functions!
- Problems still exist:
  - errno variable:
    - updated by reentrant functions such as read(), wait(), etc.
    - signal handler should save errno, and restore it on exit

Figure 10.5: nonreentrant function call in signal handler

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

static void     my_alarm(int);

int
main(void)
{
   struct passwd        *ptr;

   signal(SIGALRM, my_alarm);
   alarm(1);

   for ( ; ; ) {
        if ( (ptr = getpwnam("sar")) == NULL)
             err_sys("getpwnam error");
```

Generates SIGALRM after 1 sec

Figure 10.5: nonreentrant function call in signal handler

```
        if (strcmp(ptr->pw_name, "sar") != 0)
          printf("return value corrupted!, pw_name = %s\n",
                ptr->pw_name);
    }
}

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);
}
```

A NONREENTRANT function called by both main() and signal handler

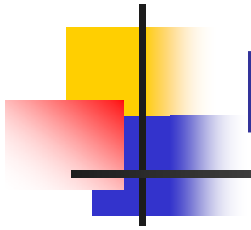
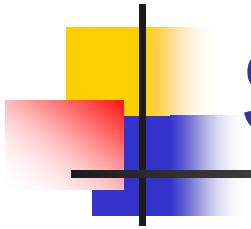


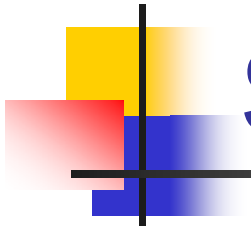
Figure 10.5: results

- Results are random
- Usually terminated by SIGSEGV
- Internal pointers corrupted when signal handler also called getpwnam() after main called it
- Sometimes terminated correctly, but return value sometimes corrupted and sometimes fine.



SIGCLD Semantics

- signal(SIGCLD, SIG_IGN)
  - no zombie processes for children
  - subsequent wait() will block until all children terminated and returns -1 with errno = ECHILD
  - different from SIG_DFL (also ignore, but without the above semantics)
- signal(SIGCLD, handler)
  - kernel checks for child to be waited for
  - if there is, kernel calls SIGCLD handler



SIGCHLD Semantics

- POSIX.1
  - Does not specify what happens when SIGCHLD is ignored
- Single UNIX Specification (XSI extension)
  - Same as for SIGCLD
- 4.4BSD and FreeBSD 5.2.1
  - Always generates zombies if SIGCHLD ignored
- SVR4, Solaris 9, Linux 2.4.22, Mac OS X 10.3
  - signal(SIGCHLD, SIG_IGN) ➔ zombies never generated

Figure 10.6: SIGCLD handler not working in System V

```
#include      <sys/wait.h>
#include      “apue.h”

static void sig_cld();

int main() {
   pid_t      pid;

   if (signal(SIGCLD, sig_cld) == SIG_ERR)
        perror("signal error");
   if ( (pid = fork()) < 0)
        perror("fork error");
   else if (pid == 0) {                  /* child */
        sleep(2);
        _exit(0);
   }
   pause();    /* parent */
   exit(0);
}
```

Figure 10.6: SIGCLD handler not working in System V

```
static void
sig_cld(int signo)    /* interrupts pause() */
{
   pid_t       pid;
   int         status;

   printf("SIGCLD received\n");
   if (signal(SIGCLD, sig_cld) == SIG_ERR)   /* reestablish
   handler */
        perror("signal error");

   if ( (pid = wait(&status)) < 0)           /* fetch child
   status */
        perror("wait error");
   printf("pid = %d\n", pid);
}
```

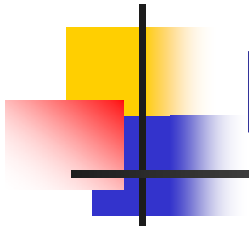


Figure 10.6: results

- SIGCLD received (repeated)
- process runs out of stack space
- process terminates normally
- on calling signal in main(), kernel checks if there is any child to be waited (there is, as we're processing SIGCLD)
- signal handler is called
- signal in signal handler is called and the whole process is repeated

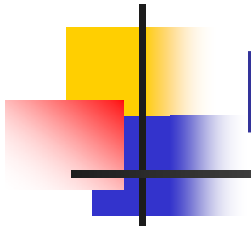
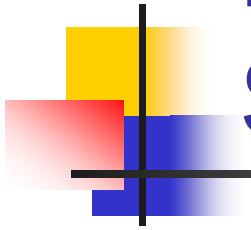


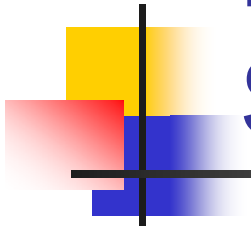
Figure 10.6 results

- FreeBSD 5.2.1 and Mac OS X 10.3
  - No problem, BSD semantics for SIGCLD different from System V semantics
- Linux 2.4.22
  - No problem, doesn't call SIGCHLD signal handler when arranging to catch SIGCHLD and child processes are ready to be waited
- Solaris 9
  - No problem, solved by extra kernel code!
- System V systems: UnixWare, OpenServer 5
  - Problem exists!!!



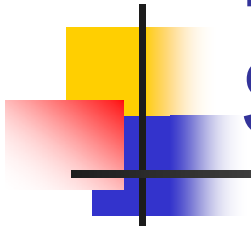
Reliable Signal Terminology and Semantics

- Signal is **GENERATED** when event that causes the signal occurs
  - 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



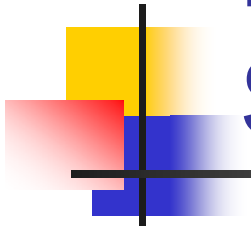
Reliable Signal Terminology and Semantics

- 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)



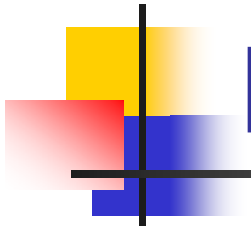
Reliable Signal Terminology and Semantics

- A blocked signal remains pending when:
  - default signal action, OR
  - catch signal using user-defined handler
- A signal stops being pending when:
  - signal is unblocked, OR
  - signal is ignored
- What to do with blocked signals is determined when it is delivered and not when it is generated (to change action)



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: POSIX.1 data type to store signal mask
  - #bits = #signals,
  - $Bit_i = 1$ ➔ signal *i* blocked

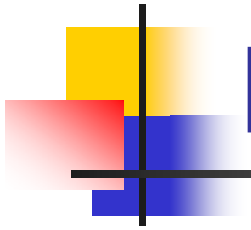


kill and raise functions

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

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

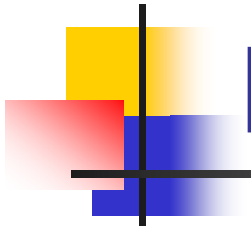
- Both return: 0 if OK, -1 on error



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

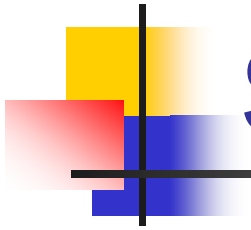
- pid > 0: sent to PID==pid
- pid < 0: sent to PID==|pid|
- pid == 0: sent to all processes with PGID == PGID of sender (with perm)
- pid == -1: all processes on the system for which the sender has permission to send signal
  - (excluding system processes)

Excluding an implementation-defined set of system processes such as kernel processes and init (pid 1)



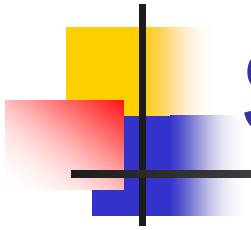
kill function

- Permission to send signals:
- Superuser: to any process
- Others: real/effective ID of sender must be equal to real/effective ID of receiver
- _POSIX_SAVED_IDS defined: receiver saved set-UID checked instead of EUID
- SIGCONT: to any process in same session



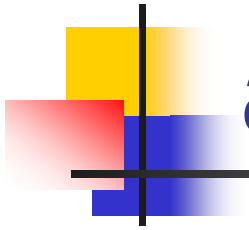
Sending NULL signal with kill

- NULL signal: signo == 0
- Sent with kill:
  - for error checking (e.g. to check if a process exists)
    - Process does not exist ➔ kill returns -1 and errno = ESRCH
  - no signal is sent



Sending signal to itself

- Suppose:
  - Send signal to itself
  - Signal is not blocked
- Result (before kill returns):
  - signo is delivered, OR
  - some other pending, unblocked signal is delivered

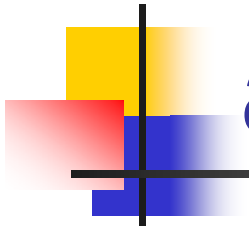


alarm Function

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

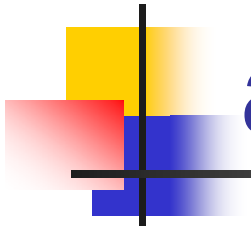
```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

- Returns: 0 or #seconds until previously set alarm



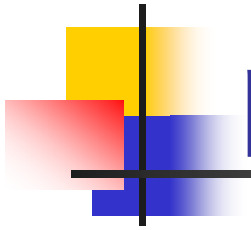
alarm Function

- Process receives signal after the specified *seconds* seconds
  - Processor scheduling delays may occur
- Only 1 alarm clock per process
- If we call alarm, and there is a previously registered alarm not expired
  - #seconds left for the previous alarm to expire is returned
  - new alarm replaces previous alarm



alarm Function

- alarm(0) ➔ a previous unexpired alarm is cancelled
- Most processes catch SIGALRM signal and do some cleanup before terminating



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**!

Figure 10.7: using alarm and pause to implement sleep (incomplete)

```
#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 */
}
```

save and restore disposition

erases previous alarms

race condition between alarm & pause

Figure 10.8: Another imperfect implementation of sleep

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

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); /* start the timer */
       pause();      /* next caught signal wakes us up */
   }
   return( alarm(0) ); /*turn off timer, return unslept
   time*/
}
```

race condition avoided, but sleep2 might abort other signal handlers

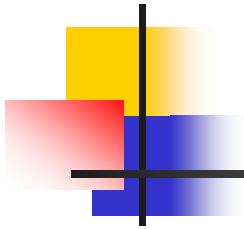


Figure 10.9: Calling sleep2

```
#include    “apue.h"

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);
}
```

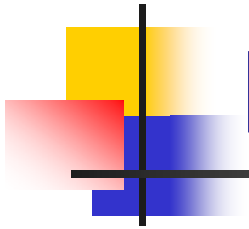


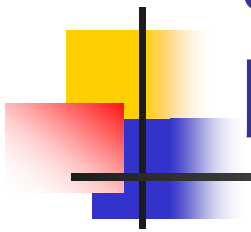
Figure 10.9: Calling sleep2

```
static void sig_int(int signo)
{
  int     i;
  volatile int j;

  printf("\nsig_int starting\n");
  for (i = 0; i < 300000; i++)
     for (j = 0; j < 4000; j++)
          k += i * j;
  printf("sig_int finished\n");
}
```

Figure 10.9: results

- **$./a.out**
- ^? /* CTRL key pressed */
- sig_int starting
- sleep2 returned: 0

SIGINT handler (sig_int) not finished because of longjmp in SIGALRM handler (sig_alrm)



alarm: upper time limit on blocking operations

- A read operation on a “slow” device can block for a long time
- An upper time limit can be imposed using the alarm function and SIGALRM
- See next program (buggy!)

Figure 10.10: read with timeout

```
#include “apue.h"

static void sig_alrm(int);

int main(void)
{
    int   n;
    char  line[MAXLINE];

    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("signal(SIGALRM) error");
    alarm(10);
    if ( (n = read(STDIN_FILENO, line, MAXLINE)) < 0)
        err_sys("read error");
    alarm(0);

    write(STDOUT_FILENO, line, n);

    exit(0);
}
```

race condition between alarm and read, set minute-long time delay to circumvent race

read not interrupted if interrupted system calls are automatically restarted

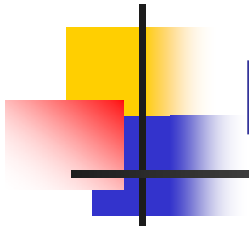


Figure 10.10: read with timeout

```
static void
sig_alrm(int signo)
{
  /* nothing to do, just return to
  interrupt the read */
}
```

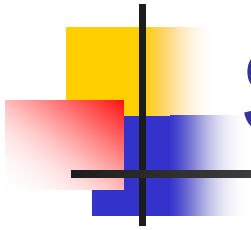
Figure 10.11: read with timeout (using longjmp)

```
#include      <setjmp.h>
#include      “apue.h"

static void           sig_alrm(int);
static jmp_buf        env_alrm;

int
main(void)
{
   int          n;
   char         line[MAXLINE];

   if (signal(SIGALRM, sig_alrm) == SIG_ERR)
       err_sys("signal(SIGALRM) error");

   if (setjmp(env_alrm) != 0)
```

Figure 10.11: read with timeout (using longjmp)

```
        err_quit("read timeout");

    alarm(10);
    if ( (n = read(STDIN_FILENO, line, MAXLINE)) < 0)
        err_sys("read error");
    alarm(0);

    write(STDOUT_FILENO, line, n);

    exit(0);
}

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

problem in interaction with other signal handlers



Signal Sets

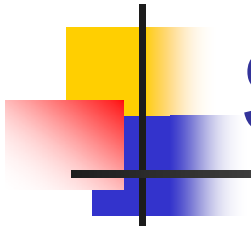
- Signal set = a set of signals
- POSIX.1: sigset_t (data type to represent multiple signals)

```
#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);
```

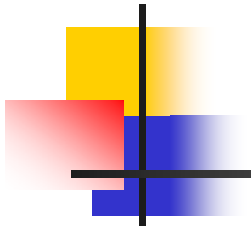
Returns: 1 if true, 0 if false, -1 on error



Signal Set

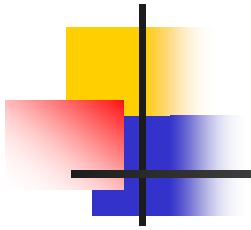
- Must call sigemptyset or sigfillset once before a signal set can be used (for initializing the set)
- For 31 signals, 32-bit integers, macros can be defined as follows:
- #define sigemptyset(ptr) (*(ptr) = 0)
- #define sigfillset(ptr)
  (*(ptr) = ~(sigset_t)0, 0)

Return Value



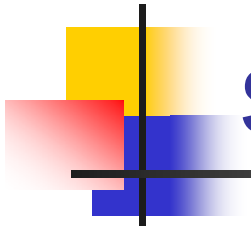
Implementations

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

#defineSIGBAD(signo) ((signo) <= 0 || (signo) >= NSIG)
/* <signal.h> usually defines NSIG to include signal number 0 */

int
sigaddset(sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

    *set |= 1 << (signo - 1);              /* turn bit on */
    return(0);
}
```



Implementations

```
int
sigdelset(sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

    *set &= ~(1 << (signo - 1));          /* turn bit off */
    return(0);
}

int
sigismember(const sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

    return( (*set & (1 << (signo - 1))) != 0 );
}
```



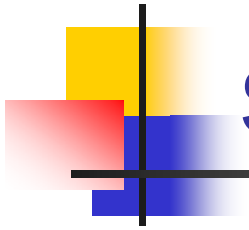
sigprocmask Function

- 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	The new signal mask for the process is the union of its current signal mask and the signal set pointed to by *set*. That is, *set* contains the additional signals that we want to block.
SIG_UNBLOCK	The new signal mask for the process is the intersection of its current signal mask and the complement of the signal set pointed to by *set*. That is, *set* contains the signals that we want to unblock.
SIG_SETMASK	The new signal mask for the process is the value pointed to by *set*.

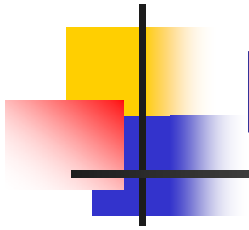
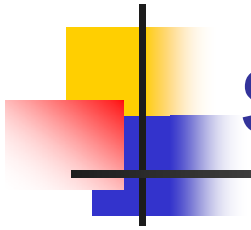


Figure 10.14: print mask

```
#include      <errno.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;
}
```



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

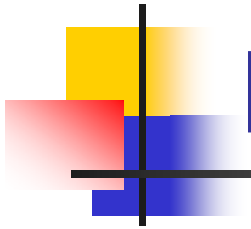


Figure 10.15: sigpending

```
#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");

   /* 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");
```

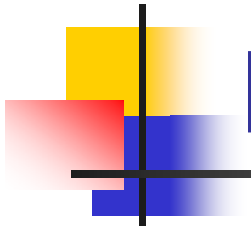


Figure 10.15: sigpending

```
    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");
}
```

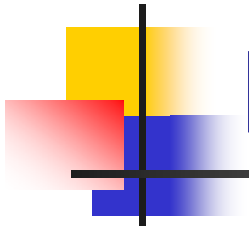
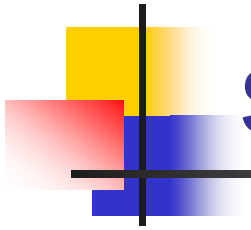


Figure 10.15: results

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



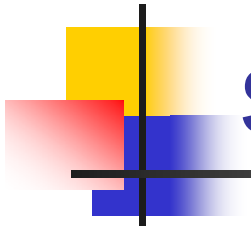
sigaction

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

nonnull ➔ modify action

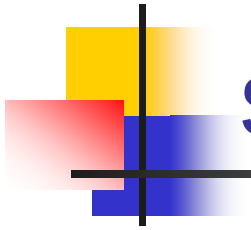
nonnull ➔ return action

- Returns: 0 if OK, -1 on error



sigaction

```
struct sigaction {
   void (*sa_handler)(int); /*addr of signal handler or
                              SIG_IGN or SIG_DFL */
   sigset_t sa_mask;        /* additional signals to block */
   int sa_flags;            /* signal options, Fig. 10.16 */
   /* alternate handler */
   void (*sa_sigaction)(int, siginfo_t *, void *);
};
```



sa_flags

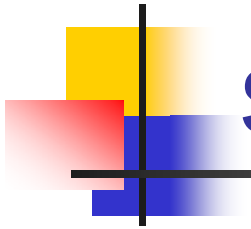
- Check Figure 10.16 for sa_flags
  - SA_INTERRUPT (not in standard, Linux only)
  - SA_NOCLDSTOP (POSIX.1, all 4 platforms)
  - SA_NOCLDWAIT (XSI, all 4 platforms)
  - SA_NODEFER (XSI, all 4 platforms)
  - SA_ONSTACK (XSI, all 4 platforms)
  - SA_RESETHAND (XSI, all 4 platforms)
  - SA_RESTART (XSI, all 4 platforms)
  - SA_SIGINFO (POSIX.1, all 4 platforms)

Implementation of signal using sigaction (reliable semantics)

```
#include     “apue.h"

/* Reliable version of signal(), using POSIX
   sigaction().  */

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) {
```

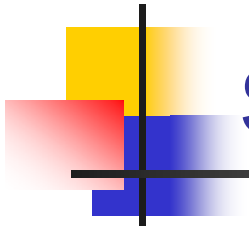
Implementation of signal using sigaction

```
#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);
}
```



signal (without restart)

```
#include “apue.h"

Sigfunc *
signal_intr(int signo, Sigfunc *func)
{
    struct sigaction      act, oact;

    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
#ifdef   SA_INTERRUPT
    act.sa_flags |= SA_INTERRUPT;
#endif
    if (sigaction(signo, &act, &oact) < 0)
         return(SIG_ERR);
    return(oact.sa_handler);
}
```



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);
```

Figure 10.20: jmp

```
#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: ");       /* Figure 10.14 */

   if (sigsetjmp(jmpbuf, 1)) {
        pr_mask("ending main: ");
        exit(0);       }
   canjump = 1;       /* now sigsetjmp() is OK */
   for ( ; ; ) pause();
}
```

Defined by ISO C

Guaranteed atomic write

(1) **Does not extend across page boundaries**

(2) **Accessed with a single machine instruction**

(3) **Volatile: accessed by 2 threads: main and signal handler**

Figure 10.20: jmp functions

```
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 */
}
```

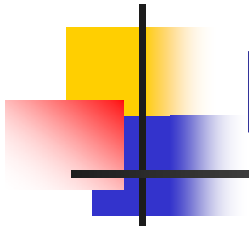
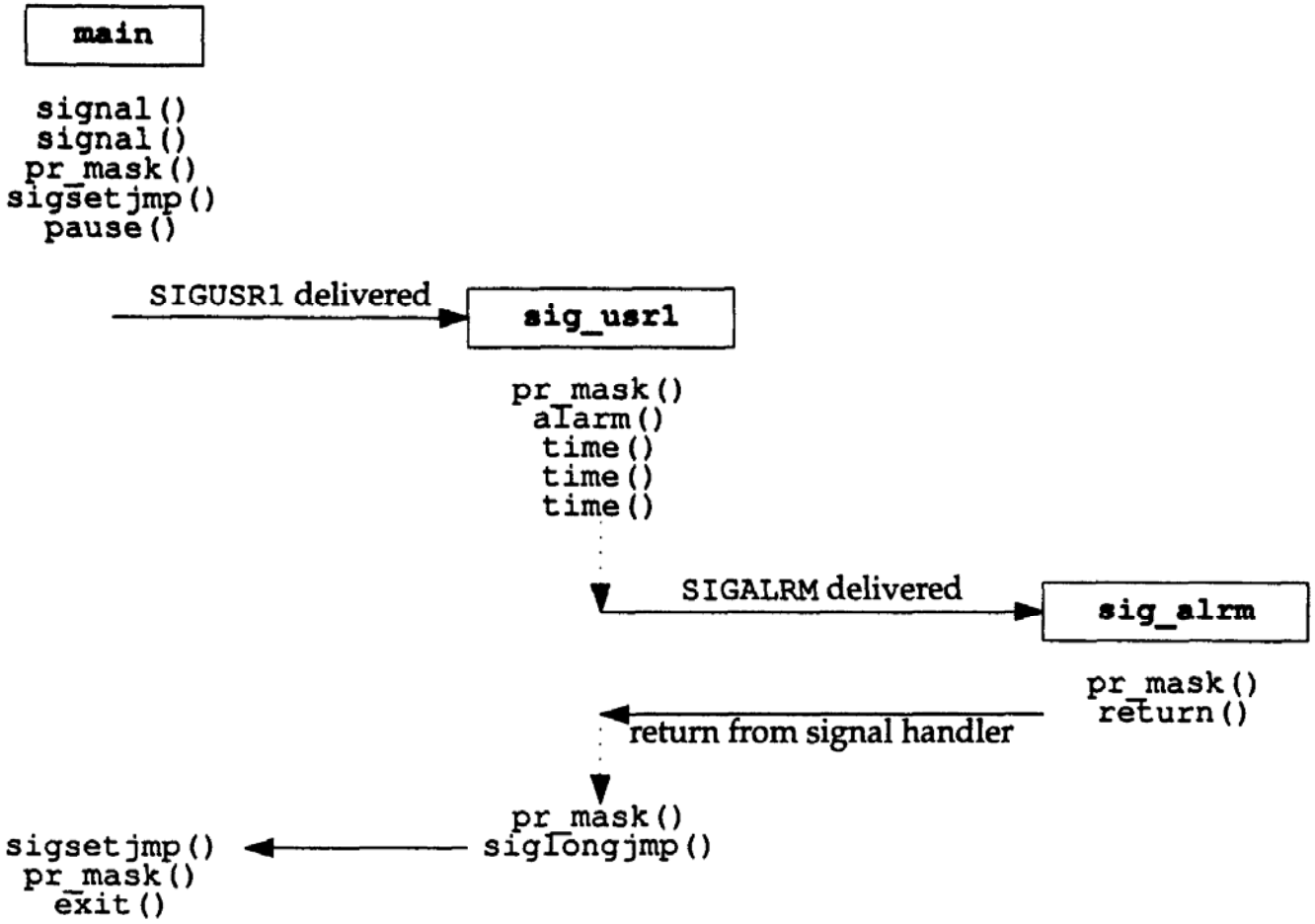


Figure 10.20: jmp functions

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

Time line for Figure 10.20



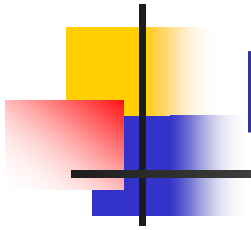
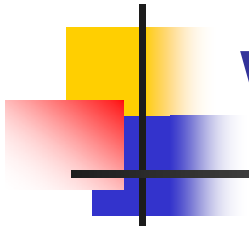


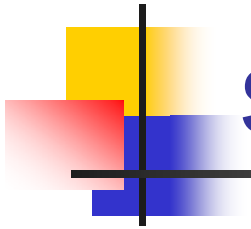
Figure 10.20: results

- $ **./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 &



Without save/restore signals

- If we use
  - setjmp, longjmp in Linux and Solaris, or
  - _setjmp, _longjmp in FreeBSD, Mac OS X
    - Final line of output:
      `ending main: SIGUSR1`
    - Main function is executing with SIGUSR1 blocked, after call to setjmp
    - This is not what we want!

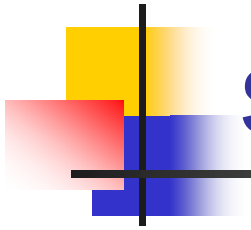


sigsuspend Function

- To protect critical section by blocking a signal

```
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");
pause();
```

What if signal occurs HERE?



sigsuspend Function

- Reset signal mask & put process to sleep for a signal (1 atomic operation)
- #include <signal.h>
- int sigsuspend(const sigset_t *sigmask);
- Returns: -1 with errno set to EINTR
- No successful return

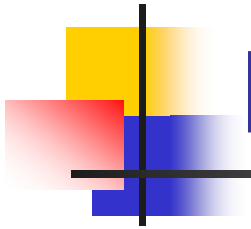


Figure 10.22: sigsuspend

```
#include       “apue.h"
static void sig_int(int);

int main(void) {
   sigset_t    newmask, oldmask, waitmask;
   pr_mask(“program start: “);
   if (signal(SIGINT, sig_int) == SIG_ERR)
       err_sys("signal(SIGINT) error");

   sigemptyset(&waitmask);
   sigaddset(&waitmask, SIGUSR1);
   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 */
```

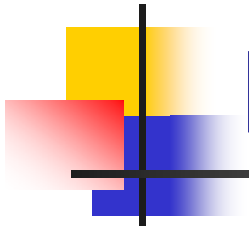


Figure 10.22: sigsuspend

```
    pr_mask("in critical region: ");

    /* Pause, allowing all signals except SIGUSR1 */
    if (sigsuspend(&waitmask) != -1)
        err_sys("sigsuspend error");
    pr_mask("after return from sigsuspend: ");

    /* Reset signal mask which unblocks SIGINT */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
    /* And continue processing ... */
    pr_mask("program exit: ");
    exit(0);
}

static void sig_int(int signo)
{
    pr_mask("\nin sig_int: "); return; }
```

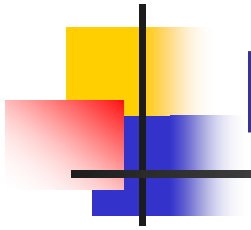


Figure 10.22: results

- $ **./a.out**
- in critical region: SIGINT
- ^?
- in sig_int: SIGINT SIGUSR1
- after return from sigsuspend: SIGINT
- program exit:

Restores mask to its value before the call to sigsuspend

Figure 10.23: wait for global variable using sigsuspend()

```
#include "apue.h"

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

static void sig_int(int signo)
    /* 1 signal handler for SIGINT and SIGQUIT */
{
    if (signo == SIGINT)
        printf("\ninterrupt\n");
    else if (signo == SIGQUIT)
        quitflag = 1;  /* set flag for main loop */
}
```

Figure 10.23: wait for global variable using sigsuspend()

```
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);
```

Figure 10.23: wait for global variable using sigsuspend()

```
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);
```

Figure 10.23: wait for global variable using sigsuspend()

```
    /* 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);
}
```

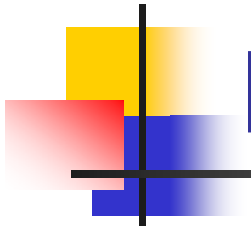


Figure 10.23: results

- $ **./a.out**
- ^?
- interrupt
- ^?
- interrupt
- ^?
- interrupt
- ^?
- interrupt
- ^?
- interrupt
- ^?
- interrupt
- ^\ $

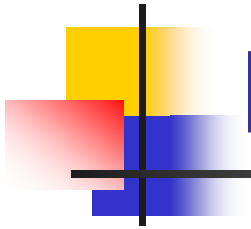


Figure 10.24: TELL_WAIT

```
#include "apue.h"

static volatile sig_atomic_t sigflag;
                                /* set nonzero by sig handler */
static sigset_t newmask, oldmask, zeromask;

static void
sig_usr(int signo)
   /* one signal handler for SIGUSR1 and SIGUSR2 */
{
   sigflag = 1;
}
```

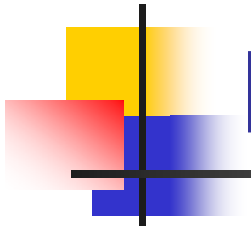


Figure 10.24: TELL_WAIT

```
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");
}
```

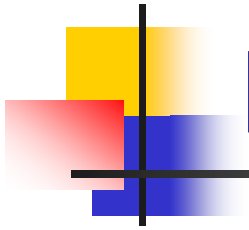


Figure 10.24: TELL_WAIT

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

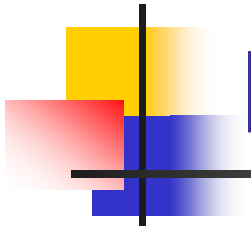


Figure 10.24: TELL_WAIT

```
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");
}
```

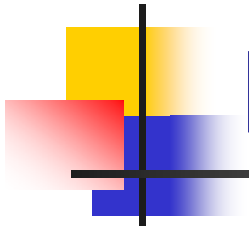


Figure 10.24: TELL_WAIT

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

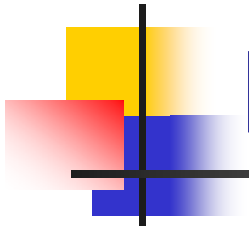
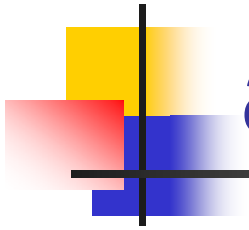


Figure 10.24: TELL_WAIT

```
void
WAIT_CHILD(void)
{
    while (sigflag == 0)
        sigsuspend(&zeromask);    /* and wait for child
    */
    sigflag = 0;

    /* Reset signal mask to original value. */
   if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
}
```



abort Function

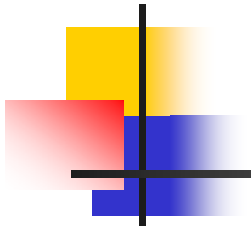
- Causes abnormal program termination by sending SIGABRT to caller using raise(SIGABRT)

```
#include <stdlib.h>
void abort(void);
```

  - Function never returns
- ISO C
  - If SIGABRT caught and signal handler returns, abort still does not return to caller.
    - exit, _exit, _Exit, longjmp, siglongjmp
- POSIX.1
  - Abort overrides blocking and ignoring of the signal by process

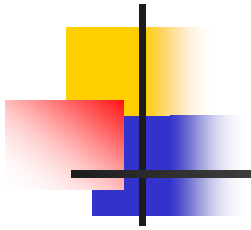
Implementation of POSIX.1 abort

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

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);
    }
```



Implementation of POSIX.1 abort

```
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 */
```

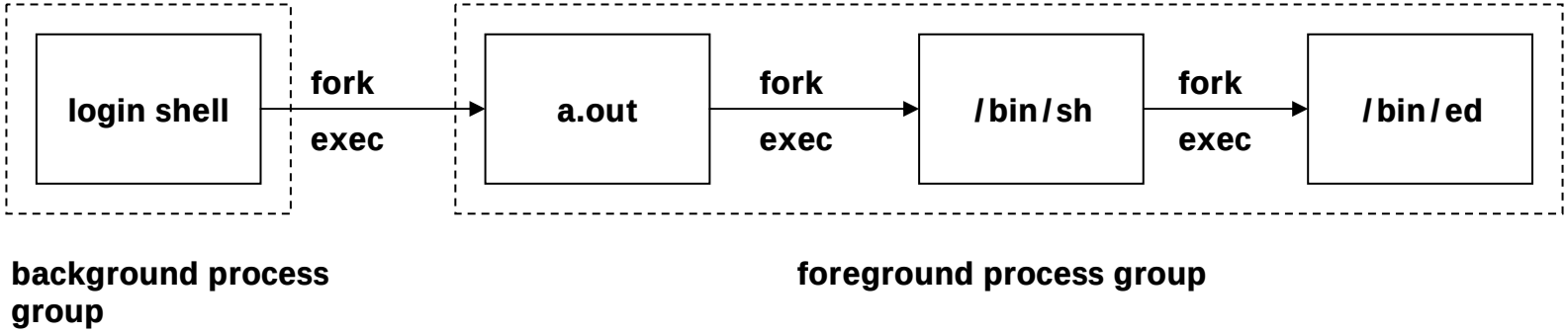


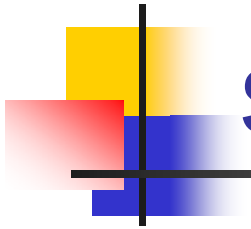
Implementation of POSIX.1 abort

```
    /* If we're here, process caught SIGABRT and returned. */
    fflush(NULL);                   /* flush all open stdio streams */
    action.sa_handler = SIG_DFL;
    sigaction(SIGABRT, &action, NULL);   /* reset to default */
    sigprocmask(SIG_SETMASK, &mask, NULL); /* just in case ... */

    kill(getpid(), SIGABRT);                /* and one more time */

    exit(1);        /* this should never be executed ... */
}
```

system Function

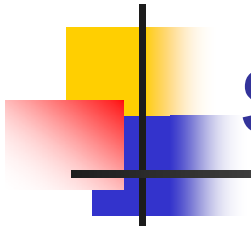
- Signal handling for calling system()
- Must ignore SIGINT, SIGQUIT
- Must block SIGCHLD





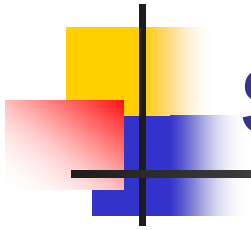
system Function

```
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <unistd.h>

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);          /* always a command processor with UNIX */
```



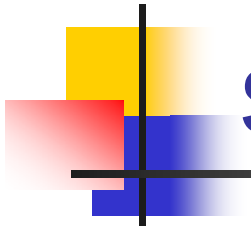
system Function

```
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);
```



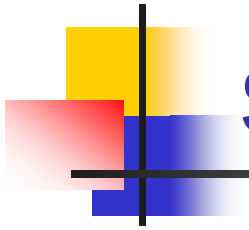
system Function

```
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 */
}
```



system Function

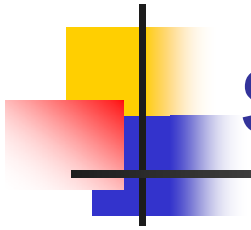
```
    else {                                                  /* parent */
            while (waitpid(pid, &status, 0) < 0)
                    if (errno != EINTR) {
                            status = -1; /* error other than EINTR
    from waitpid() */
                            break;
                    }
    }
    /* 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);
}
```



sleep Function

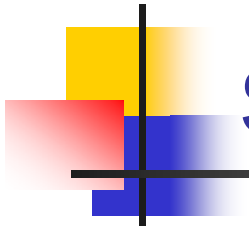
```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

- Returns 0 or number of unslept seconds
- Causes process to be suspended until:
  - Amount of wall clock time specified by seconds has elapsed, OR
  - A signal is caught by the process and the signal handler returns.



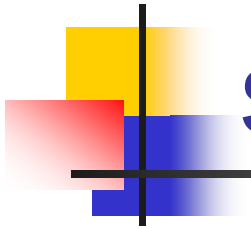
sleep Function

```
#include "apue.h"

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



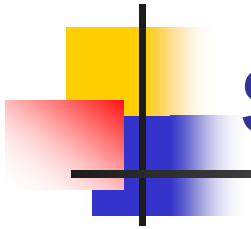
sleep Function

- Solaris 9
  - Implements sleep using alarm
  - Previously scheduled alarm is properly handled
    - alarm(10) → 3 wall clock seconds later → sleep(5) → SIGALRM received 2 seconds after waking from sleep
    - alarm(6) → 3 wall clock seconds later → sleep(5) → sleep returns in 3 seconds (not 5 seconds), return value of sleep is 2 (#unslept seconds)
- FreeBSD 5.2.1, Linux 2.4.22, Mac OS X 10.3
  - Implement sleep using nanosleep(2) (without signals, declared in Single UNIX Specification)



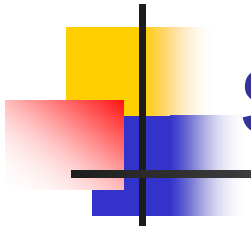
sleep Function

```
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);
```



sleep Function

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

alarm(nsecs);

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



sleep Function

```
    /* 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);
}
```