系統程式設計
Systems Programming

鄭卜壬教授
臺灣大學資訊工程系

Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Contents

1. OS Concept & Intro. to UNIX
2. UNIX History, Standardization & Implementation
3. File I/O
4. Standard I/O Library
5. Files and Directories
6. System Data Files and Information
7. Environment of a Unix Process
8. Process Control
9. Signals
10. Inter-process Communication
11. Thread Programming
12. Networking

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Signals

- **Objectives:**
  - An overview of signals
  - Related functions and problems (reliability & incompatibility)
- **What is a signal?**
  - Used to notify a process of important events
  - Synchronous vs. asynchronous (events)
    - Synchronous: occur as a result of instruction execution

      e.g. divided by zero, segmentation fault
    - Asynchronous: external to current execution context

      e.g. killed by other processes, child termination
  - 15 signals for Version 7, 31 signals for SVR4 & 4.3+BSD – <signal.h> (# > 0)

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Conditions to Generate Signals

- Terminal-generated signals
  - DELETE key, Ctrl-C → SIGINT (2)
- Signals from exceptions
  - divided-by-0 → SIGFPE (8)
  - illegal memory access → SIGSEGV (11)
- Function *kill*
  - Owner or superuser
- Shell command *kill*, e.g., kill –9 pid (SIGKILL)
- Signals because of software conditions
  - reader of the pipe terminated → SIGPIPE
  - expiration of an alarm clock → SIGALRM

Disposition of a Signal (Action)

- Ignore signals
  - SIGKILL and SIGSTOP can not be ignored.
  - There could be undefined behaviors for ignoring signals, such as SIGFPE.
- Catch signals
  - SIGKILL and SIGSTOP can not be caught
  - Provide a signal handler
    - e.g., calling waitpid() when a process receives SIGCHLD
- Apply the default action
  - Terminate, ignore, stop

Examples of Signals

- **SIGABRT – terminate w/core**
  - Call abort()
- **SIGALRM – terminate**
  - Call setitimer(), alarm()
- **SIGBUS – terminate w/core**
  - Implementation-defined HW fault
- **SIGCHLD – ignore**
  - sent to the parent whenever a child process terminates or stops

Examples of Signals

- **SIGCONT – continue/ignore**
  - Continue a stopped process
- **SIGFPE – terminate w/core**
  - Divid-by-0, floating point overflow, etc.
- **SIGHUP – terminate**
  - Disconnection is detected by the terminal interface (no daemons) → controlling process of a terminal
  - Triggering of the rereading of the configuration files (daemons)
- **SIGILL – terminate w/core**
  - Illegal hardware instruction

Examples of Signals

- **SIGINT – terminate**
  - DELETE key or CTRL-C
- **SIGIO – terminate/ignore**
  - Indicate an asynchronous I/O event (SIGIO=SIGPOLL, Terminate on SVR4)
- **SIGKILL – terminate**
  - Could not be ignored or caught!
- **SIGPIPE – terminate**
  - reader of the pipe/socket terminated

Examples of Signals

- **SIGPROF – terminate**
  - A profiling timer expires (setitimer)
- **SIGPWR – ignore (SVR4)**
  - System dependent on SVR4
    - UPS → init shutdowns the system
- **SIGQUIT – terminate w/core**
  - CTRL-\ triggers the terminal driver to send the signal to all foreground processes.
- **SIGSEGV – terminate w/core**
  - Invalid memory access

Examples of Signals

- **SIGSTOP – stop process (like SIGTSTP)**
  - Can not be caught or ignored
- **SIGSYS – terminate w/core**
  - Invalid system call
- **SIGTERM – terminate**
  - Termination signal sent by *kill* command
- **SIGTSTP – stop process**
  - Terminal stop signal (CTRL-Z) to all foreground processes

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Examples of Signals

- **SIGURG – ignore**
  - Urgent condition (e.g., receiving of out-of-band data on a network connection)
- **SIGUSR1 – terminate**
  - User-defined
- **SIGUSR2 – terminate**
  - User-defined
- **SIGVTALRM – terminate**
  - A virtual timer expires (setitimer)

Examples of Signals

- **SIGWINCH – ignore**
  - Changing of a terminal window size
- **SIGXCPU – terminate w/core**
  - Exceed the soft CPU time limit
- **SIGXFSZ – terminate w/core**
  - Exceed the soft file size!

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Examples of Signals

- **Remark**
  - Terminate w/core – not POSIX.1
    - No core file:
      - non-owner setuid process
      - non-grp-owner setgid process
      - no access rights at the working dir
      - file is too big (RLIMIT_CORE)
    - core.*prog* (4.3+BSD)

Set Disposition of Signal to Handler

```
#include <signal.h>
void (*signal(int signo, void (*func)(int)))(int);
```

- typedef void Sigfunc(int)

  Sigfunc *signal(int, Sigfunc *);

```
#define SIG_ERR (void (*)())-1
#define SIG_DFL (void (*)())0
#define SIG_IGN (void (*)())1
```

- signo: signal number
- func: SIG_IGN, SIG_DFL, the address of the signal handler / signal-catching function
  - SIGKILL & SIGSTOP cannot be ignored and caught
- Return: the address of the previous handler.

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

pause() forces a process to **pause/sleep** until a signal is caught. The signal cannot be set to be ignored.

Example Program: catch SIGUSR

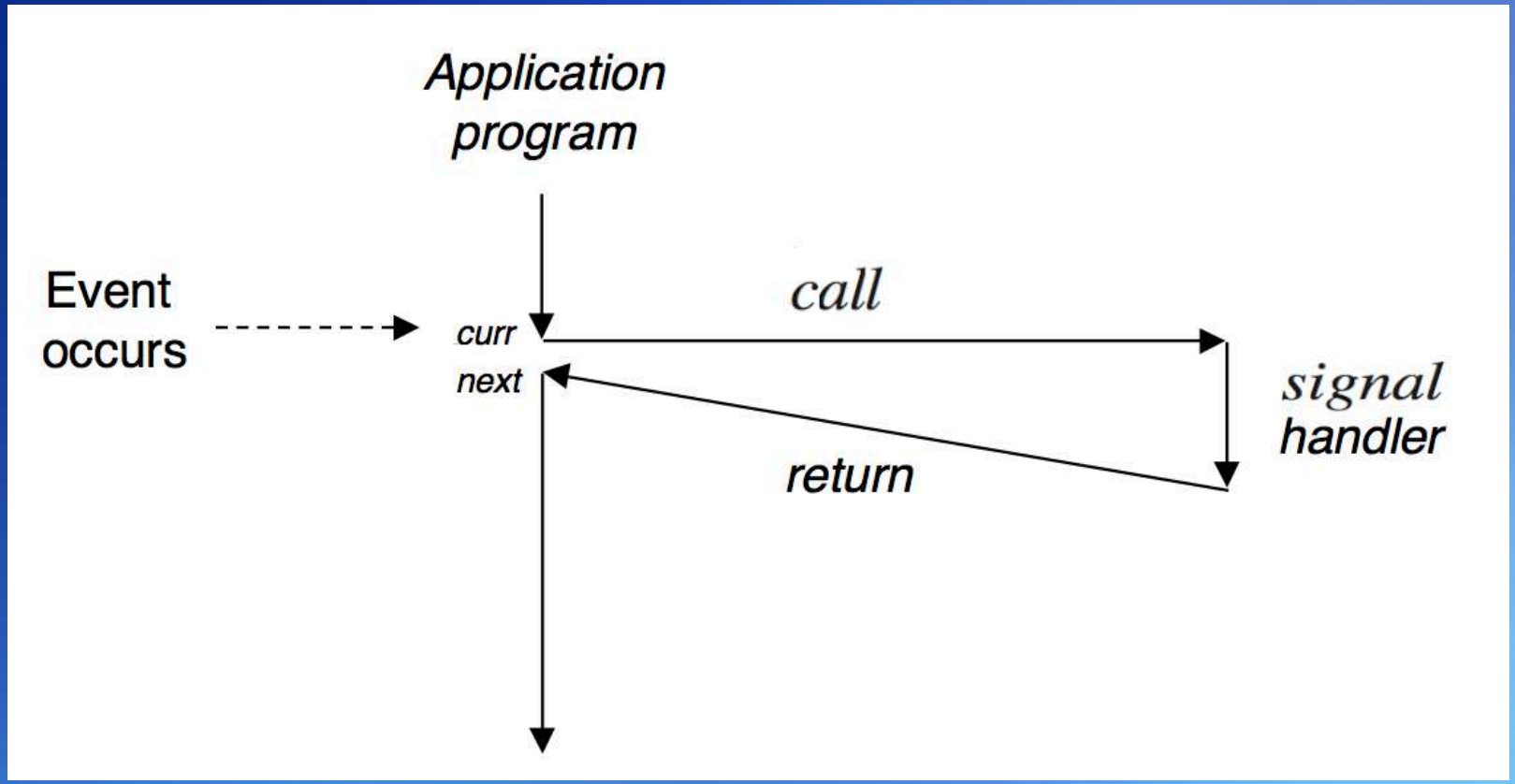
```
static void sig_usr(int);   /* one handler for both signals */

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();
}

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);
}
```

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

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Handling Signals

(Programmer's Viewpoint)



Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

signals

- **Remarks**
  - Not able to determine the current disposition of a signal without changing it.

    Possible solution:

    e.g.: if (^C is not ignored now) then set it to a handler

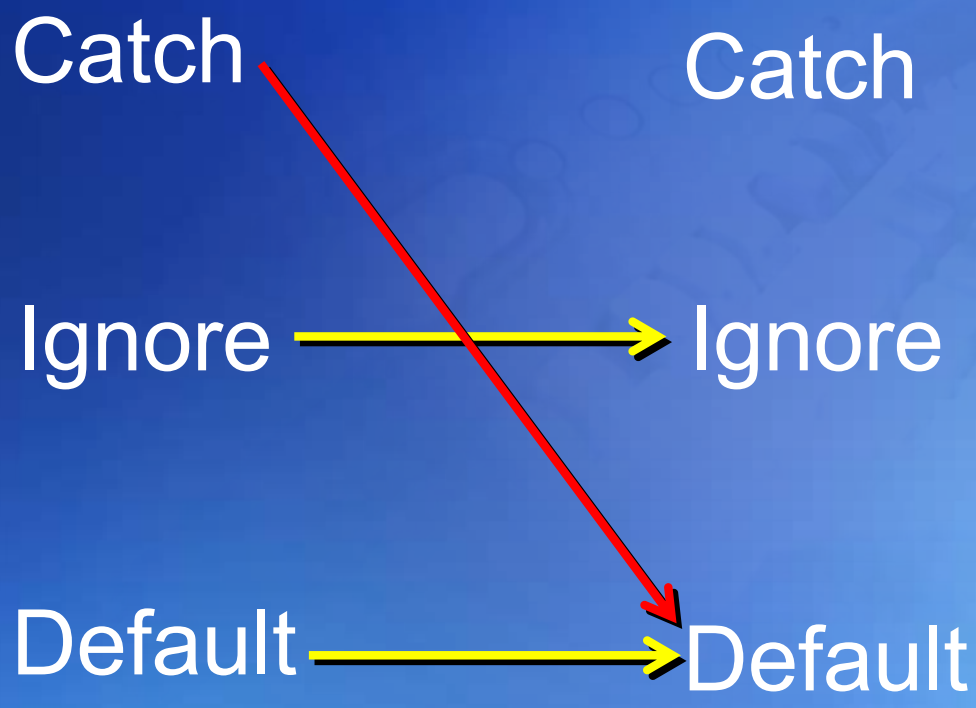
```
int sig_int();
if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, sig_int);
```

- The process catches only the signal if the signal is not currently being ignored.
  - pause() will not return for SIGCHLD

signals

- **Program Start-Up**
  - All signals are set to their default actions unless the process that calls `exec` is ignoring the signal.
    - The exec functions change the disposition of any signals that are being caught to their default action. (Why?)
  - How about fork()?
    - fork() lets the child inherit the dispositions of the parent!
  - The shells automatically set the disposition of the interrupt and quit signals of background processes to “ignored”. (Why?)

How exec() Changes Disposition?



Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Interrupted System Calls

- **Conventional Approach**
  - Slow system calls can block forever
  - “Slow” system calls could be interrupted

    → errno = EINTR
- **“Slow” System Calls (not disk I/O):**
  - Reads from or writes to files that can block the caller forever (e.g., pipes, terminal devices, and network devices)
  - Opens of files (e.g., terminal device) that block until some conditions occurs.
  - pause, wait, certain ioctl operations
  - Some IPC functions

Interrupted System Calls

- **A typical code sequence**

```
again:
    if ((n = read(fd, buff, BUFFSIZE)) < 0)   {
            if (errno == EINTR)
                    goto again;
    }
```

- **Restarting of interrupted system calls – since 4.2BSD**
  - ioctl, read, readv, write, writev, wait, and waitpid.
  - 4.3BSD allows a process to disable the restarting on a per-signal basis.

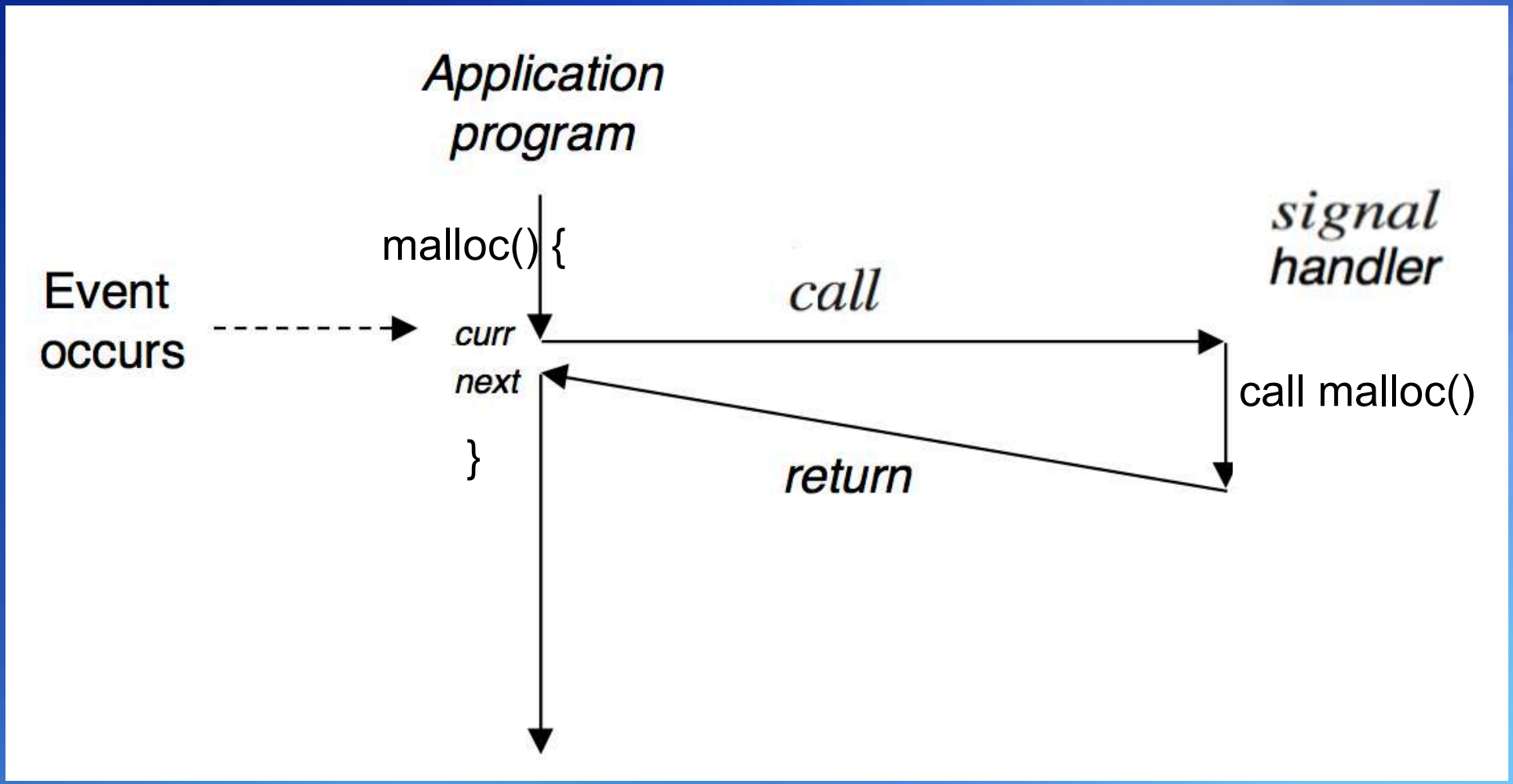
Summary of signal implementations

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

4.3+BSD: sigaction() with SA_RESTART

Reentrant Functions

- **When interrupts occur, a function could be called twice and causes problems.**
- **Potential Problem**
  - In the signal handler, we can't tell where the process was executing when the signal was caught!
- **A function is described as reentrant if it can be safely called recursively**


Application program
malloc() {
Event occurs
curr
next
}
call
signal handler
call malloc()
return

Example of Non-Reentrant Function

```
/* non-reentrant function */
char *strtoupper(char *string) {
      static char buffer[MAX_STRING_SIZE];
       int index;

       for (index = 0; string[index]; index++)
             buffer[index] = toupper(string[index]);
      buffer[index] = 0 ;
       return buffer;
}
```

From “General Programming Concepts: Writing and Debugging Programs”

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Example of Non-Reentrant Function

```
/* non-reentrant function */
char *strtoupper(char *string) {
        char *buffer;
        int index;

        /* error-checking should be performed! */
        buffer = malloc(MAX_STRING_SIZE);

        for (index = 0; string[index]; index++)
                buffer[index] = toupper(string[index]);
        buffer[index] = 0 ;
        return buffer;
}
```

From “General Programming Concepts: Writing and Debugging Programs”

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Better Solution

work only on the data provided to it by the caller

```
/* reentrant function (a better solution) */
char *strtoupper_r(char *in_str, char *out_str) {
      int index;

      for (index = 0; in_str[index]; index++)
            out_str[index] = toupper(in_str[index]);
      out_str[index] = 0 ;

      return out_str;
}
```

From “General Programming Concepts: Writing and Debugging Programs”

Some Non-reentrant Functions

- The POSIX.1 process environment functions *getlogin*(), *ttyname*() (see ISO/IEC 9945:1-1996, §4.2.4 and 4.7.2)
- The C-language functions *asctime*(), *ctime*(), *gmtime*() and *localtime*() (see ISO/IEC 9945:1-1996, §8.3.4-8.3.7)
- The POSIX.1 system database functions *getgrgid*(), *getgrnam*(), *getpwuid*() and *getpwnam*() (see ISO/IEC 9945:1-1996, §9.2.1)

Reentrant Functions

- **Non-Reentrant Functions**
  - Those which use static data structures
  - Those which return a pointer to static data
  - Those which call malloc() or free()
  - Those which are part of the standard I/O library – usage of global data structures – such as printf(). Calling printf() in your signal handlers may lead to unexpected results.
  - Those which call non-reentrant functions
- **Restoring of *errno* inside a handler**
- **Updating of a data structure – longjmp()**

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Reentrant functions specified by Single UNIX Specification

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

Call a non-reentrant function from a signal handler

```
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);
    }
}
```

Unreliable Signals

- **Unreliable Signals: "Signals could get lost!"**
- **Why?**
  - **(1) The action for a signal was reset to its default each time the signal occurred.**

```
int sig_int();
…
signal(SIGINT, sig_int);
…
sig_int() {
    signal(SIGINT, sig_int);
    …
}
```

A process could terminate!

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

(2) The process could only ignore signals, instead of turning off the signals when the process is busy.

```
int sig_int_flag;

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

sig_int() {
    signal(SIGINT, sig_int);
    sig_int_flag = 1;
}
```

A process could sleep forever!

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Reliable Signals

- **A signal is *generated* (*or sent to a process*) when**
  - the event that causes the signal occurs!
  - A flag is set in the process table.
- **A signal is *delivered* when**
  - the action for the signal is taken.

(once *blocked*, it will be not delivered)

- **A signal is *pending* during**
  - the time between its delivery and generation.

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Reliable Signals

- **A signal is *blocked* until**
  - the process unblocks the signal, or
  - the corresponding action becomes “ignore”

    (if the action is either default or a handler).
- **The system determines which signals are blocked and pending!**
  - sigpending()
- ***A signal mask* for each process defines the set of signals currently blocked from delivery**
  - sigprocmask()

Reliable Signals

- **Signals are *queued* when**
  - A blocked signal is generated more than once.
  - POSIX.1 (but not over many Unix) allows a system to deliver the signal either once or more than once.
- **Delivery order of signals**
  - No order under POSIX.1, but its Rationale states that signals related to the current process state, e.g., SIGSEGV should be delivered first.

kill and raise

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

- *pid* > 0 → to the process
- *pid* == 0 → to "all" processes with the same gid of the sender (excluding proc 0, 1, 2)
- *pid* < 0 → to "all" processes with gid == |pid|
- *pid* == -1 → broadcast signals under SVR4 and 4.3+BSD

kill and raise

- **Right permissions must be applied!**
  - Superuser is mighty!
  - Real or effective uid of the sender == that of the receiver
    - _POSIX_SAVED_IDS → receiver's saved set-uid is checked up, instead of effective uid
- **signo ==0 ~ a null signal**
  - Normal error checking is performed by kill() to see if a specific process exists.
    - kill() returns –1, and *errno* == ESRCH

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

- **Signal (not blocked) generated for the calling process**
  - signo or some other pending, unblocked signal is delivered to the process before kill() returns
  - The implementation of abort() will explain it

alarm & pause

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

```
unsigned int alarm(unsigned int secs);
```

- SIGALRM is generated when the timer expires
- There could be a delay because of processor scheduling delays.
- A previously registered alarm is replaced by the new value – the left seconds is returned!
- alarm(0) resets the alarm.
- Default: termination

alarm & pause

```
#include <unistd.h>
int pause(void);
```

- pause() suspends the calling process until a signal is caught
- Return if a signal handler is executed and returns.
  - Returns –1 with *errno* = EINTR

sleep() makes the calling process sleep until seconds have elapsed or a signal arrives which is not ignored.

Sleep1(): Simple, incomplete implementation of sleep

```
static void
sig_alrm(int signo)
{
    /* 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 */
}
```

Potential problems:

1. Any previous alarm?
2. The lost of the previous SIGALRM handler
3. A race condition (between alarm() & pause())

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Any previous alarm?

```
if ( (oldalarm = alarm(nsecs)) > 0)
    if (oldalarm < nsecs) {
        alarm( oldalarm );
      pause();
      return(alarm(0));}
    else {
      pause();
      return( alarm( oldalarm - nsecs) );
    }
```

Previous SIGALRM handler?

```
if ( (oldhandler = signal(SIGALRM, sig_alrm))
   == SIG_ERR )
   return (nsecs);

alarm( nsecs);
pause();

signal(SIGALRM, oldhandler);
return( alarm(0) );
```

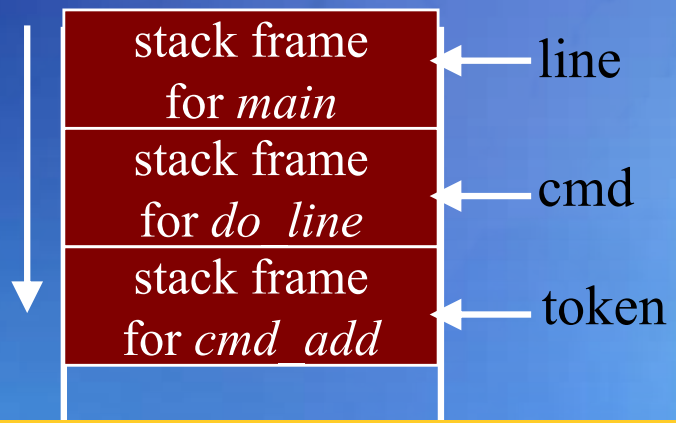
Nonlocal Jumps (Ch7.10)

- **Powerful (but dangerous) user-level mechanism for transferring control to an arbitrary location**
- **Objective:**
  - Goto to escape from a deeply nested function call!
    - break the procedure call/return discipline
  - Useful for error recovery and signal handling (Ch. 10)

```
#define TOK_ADD    5

void      do_line(char *);
void      cmd_add(void);
int       get_token(void);

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

      while (fgets(line, MAXLINE, stdin) != NULL)
          do_line(line);
      exit(0);
}

char      *tok_ptr;       /* global pointer for get_token() */

void
do_line(char *ptr)        /* process one line of input */
{
     int    cmd;

     tok_ptr = ptr;
     while ((cmd = get_token()) > 0) {
         switch (cmd) { /* one case for each command */
         case TOK_ADD:
                 cmd_add();
                 break;
         }
     }
}
```

- What if get_token() suffers a fatal error?
- How to return to main() to get the next line?

```
void
cmd_add(void)
{
    int       token;

    token = get_token();
    /* rest of processing for this command */
}

int
get_token(void)
{
    /* fetch next token from line pointed to by tok_ptr */
}
```



Note: Automatic variables are allocated within the stack frames!

setjmp and longjmp

#include <setjmp.h>

int setjmp(jmp_buf env);

- Return 0 if called directly; otherwise, it could return a value *val* from longjmp().
- *env* tends to be a global variable.

int longjmp(jmp_buf env, int val);

- longjmp() unwinds the stack and affect some variables.

- Parameter: jmp_buf

  Definition is machine-dependent.

  Includes CPU registers, stack pointers and return address (Program Counter).

Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2006
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

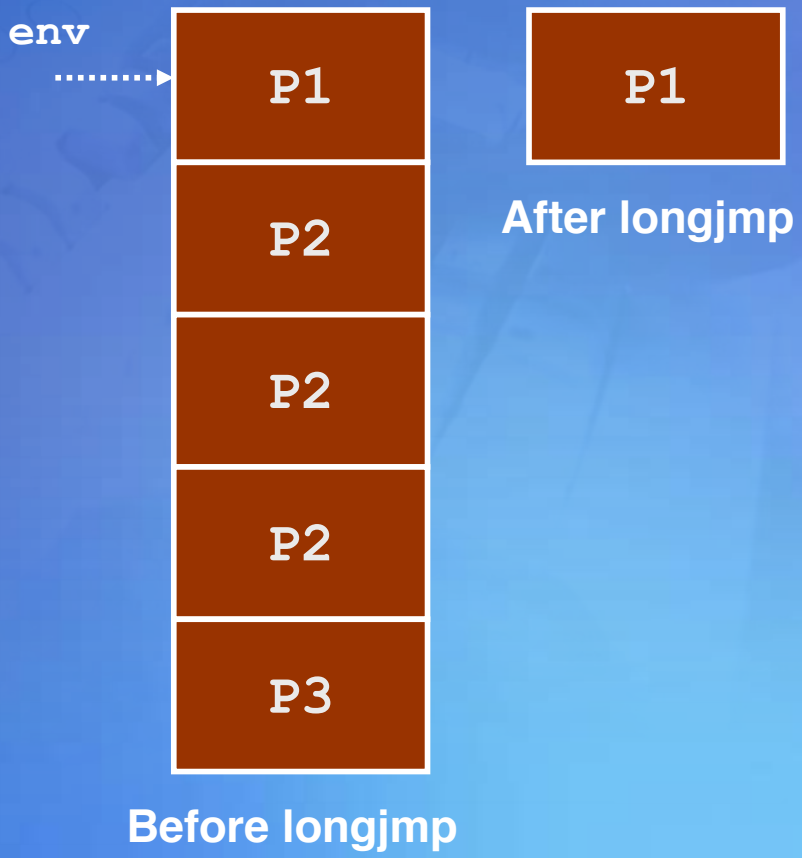
Example of setjmp() and longjmp()

```
#define TOK_ADD    5

jmp_buf jmpbuffer;

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

      if (setjmp(jmpbuffer) != 0)
          printf("error");
      while (fgets(line, MAXLINE, stdin) != NULL)
         do_line(line);
      exit(0);
}

 ...

void
cmd_add(void)
{
     int     token;

     token = get_token();
     if (token < 0)     /* an error has occurred */
         longjmp(jmpbuffer, 1);
     /* rest of processing for this command */
}
```

Potential Problems of Nonlocal Jumps

- **Works within stack discipline**
  - Can only long jump to environment of function that has been called but not yet completed

```
jmp_buf env;

P1()
{
  if (setjmp(env)) {
    /* Long Jump to here */
  }
  P2();
}

P2()
{  . . . P2(); . . . P3(); }

P3()
{
  longjmp(env, 1);
}
```

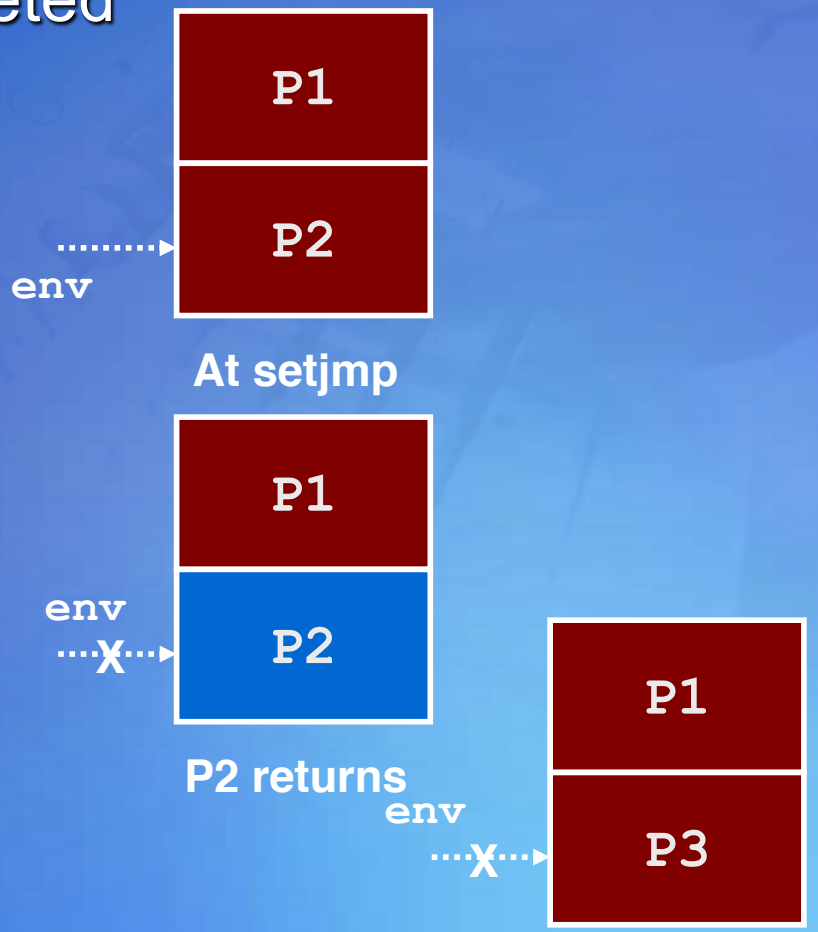


Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2006
of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Potential Problems of Nonlocal Jumps

- **Works within stack discipline**
  - Can only long jump to environment of function that has been called but not yet completed

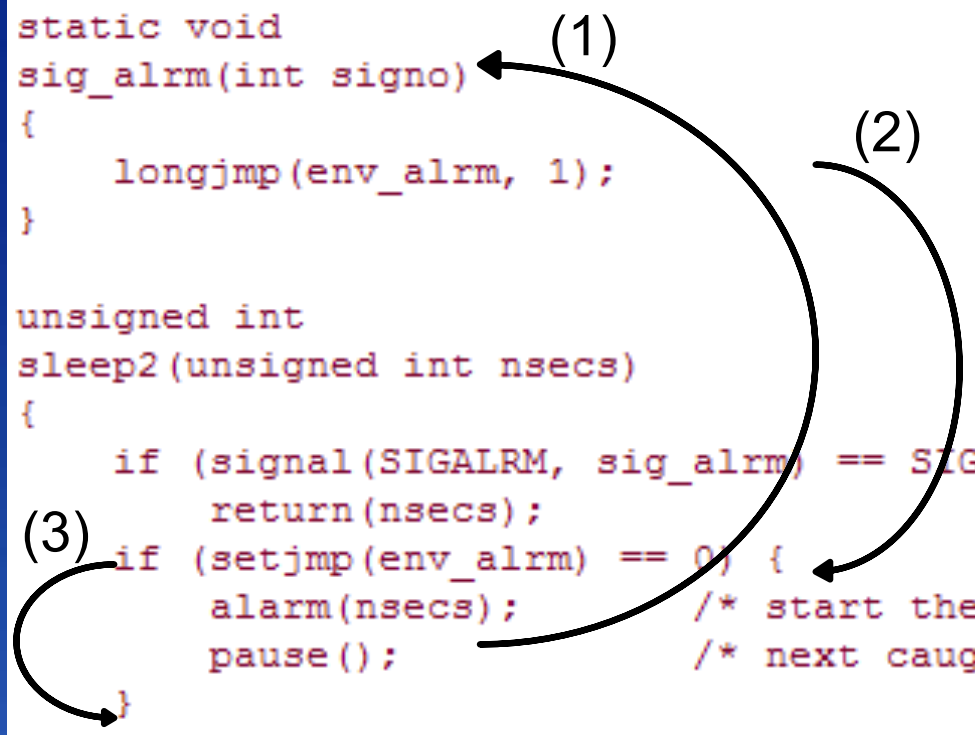
```
jmp_buf env;

P1()
{
  P2(); P3();
}

P2()
{
   if (setjmp(env)) {
    /* Long Jump to here */
  }
}

P3()
{
  longjmp(env, 1);
}
```



Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu
f Computer Science and Information Engineering
Graduate Institute of Multimedia and Networking, National Taiwan University

Sleep2(): Another (imperfect) implementation of sleep

```
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 */
}
```

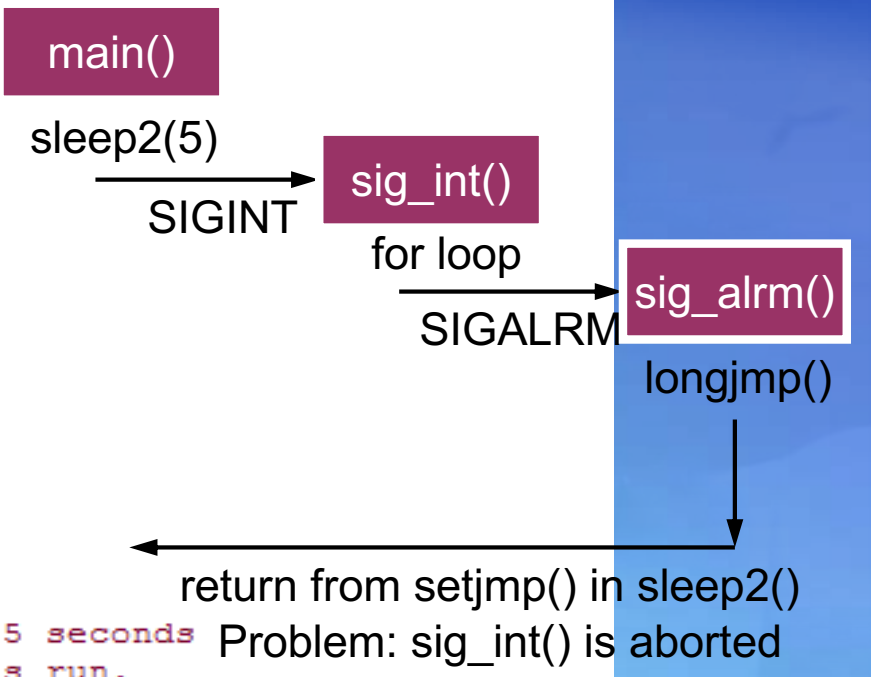


When a future SIGALRM occurs, the control goes to the right point in sleep2() → No race condition!

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Calling sleep2() from a program that catches other signals

```
unsigned int        sleep2(unsigned int);
static void         sig_int(int);

int
main(void)
{
    unsigned int        unslept;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    unslept = sleep2(5);
    printf("sleep2 returned: %u\n", unslept);
    exit(0);
}

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

    /*
     * Tune these loops to run for more than 5 seconds
     * on whatever system this test program is run.
     */
    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");
}
```



How if SIGALRM interrupts other signal handlers → they are aborted!

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

alarm & pause

- **Timeout a read() (on a “slow” device)!**
  - A race condition (between alarm() & read())
  - Automatic restarting of read()?
    - No portable way to specifically interrupt a slow system call under POSIX.1.
- **Timeout & restart a read by longjmp()!**
  - Problems with other signal handlers!

Calling read() with a timeout, using alarm()

```
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);
}

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

Calling read() with a timeout, using longjmp()

```
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)
        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);
}
```

Signal Mask

- **Signal mask is a per process property**
  - Empty in the beginning
- **Under certain circumstance, we may desire the process not to be interrupted.**
  - E.g., in the process of changing handlers or in the process of handling a delivered signal
- **It defines the set of signals for being blocked from delivery to the process**
  - Blocked signal vs. Ignored signal
    - An ignored signal will not be delivered to the receiving process.
    - A blocked signal will be queued until the receiving process is ready to process it or the signal is ignored.

Signal Sets

- **Why sigset_t ?**
  - To present a number of signals
  - The number of different signals could exceed the number of bits in an integer!

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int sig_no);
int sigdelset(sigset_t *set, int sig_no);
int sigismember(const sigset_t *set, int sig_no);
```

sigprocmask

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

- If *oset* is not null, return current signal mask by it
- If *set* is not null, check *how*
  (SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK); otherwise, ignore *how*
- If any unblocked signals are pending, at least one of unblocking signals will be delivered before the sigprocmask() returns.

Ways to change current signal mask using sigprocmask

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 replaced by the value of the signal set pointed to by *set*.

Example of showing names of signals currently blocked

```
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

```
#include <signal.h>
int sigpending(sigset_t *set);
```

- Returns the set of pending, blocked signals
- **Example (next slide)**
  - SIGQUIT is blocked, current signal mask is saved, the process sleeps for 5 seconds, and then SIGQUIT is unblocked.
  - SIGQUIT is delivered until the signal is unblocked and before sigprocmask() returns.
  - Can we use “SET_UNBLOCK” when trying to unblocking SIGQUIT?
  - No queuing of signals.

Example of signal sets and sigprocmask

```
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");
    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");
}
```

```
$ ./a.out

^\                          generate signal once (before 5 seconds are up)
SIGQUIT pending             after return from sleep
caught SIGQUIT              in signal handler
SIGQUIT unblocked           after return from sigprocmask
^\Quit(coredump)            generate signal again
$ ./a.out

^\^\^\^\^\^\^\^\^\^         generate signal 10 times (before 5 seconds are up)
SIGQUIT pending
caught SIGQUIT              signal is generated only once
SIGQUIT unblocked
^\Quit(coredump)            generate signal again
```

What happens when a quit signal occurs in A~E?

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

        // A
        signal( SIGQUIT, sig_quit );

        // B
        sigemptyset( &newmask );
        sigaddset( &newmask, SIGQUIT );
        sigprocmask( SIG_BLOCK, &newmask, &oldmask);

        sleep(5); // C: during the period of 5 seconds

        sigpending( &pendmask );
        if ( sigismember( &pendmask, SIGQUIT ) )
             printf("SIGQUIT pending\n");

        // D
        sigprocmask( SIG_SETMASK, &oldmask, NULL );

        // E
        exit( 0 );
}

static void sig_quit( int signo )
{
    printf( "caught SIGQUIT\n" );
    signal( SIGQUIT, sig_quit );
}
```

Signal Masks (fork() & exec())

- **Properties (of parent) inherited by child**
  - fork(): signal mask, dispositions
  - exec(): signal mask

    pending signals, time left until alarm clock
- **Properties (of parent) changed from child**
  - fork()

    pending signals are set to empty for child

    pending alarms are cleared for child
  - exec()

    disposition of caught signals is set to

    default action for child

sigaction

- **A better implementation of signal()**
- **Allows us to examine or modify the signal handler**
- **Signal mask:**
  - Additional signals can be masked before the handler function is called.
  - The caught signal is blocked before the handler returns to avoid signal lost.
- **Installed action remains installed until another sigaction() is called.**

sigaction

```
#include <signal.h>
int sigaction(int signo,
    const struct sigaction *act,
     struct sigaction *oact);
```

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

- signo: signal number
- act: if not-null, modify the handler
- oact: if not-null, return previous handler
- sa_mask: additional signals needed to be blocked when the action is taken. (including the delivered signal)
- sa_flags: next slide
- Unlike signal(), signal handlers remain!

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Option flags (sa_flags) for the handling of each signal

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.

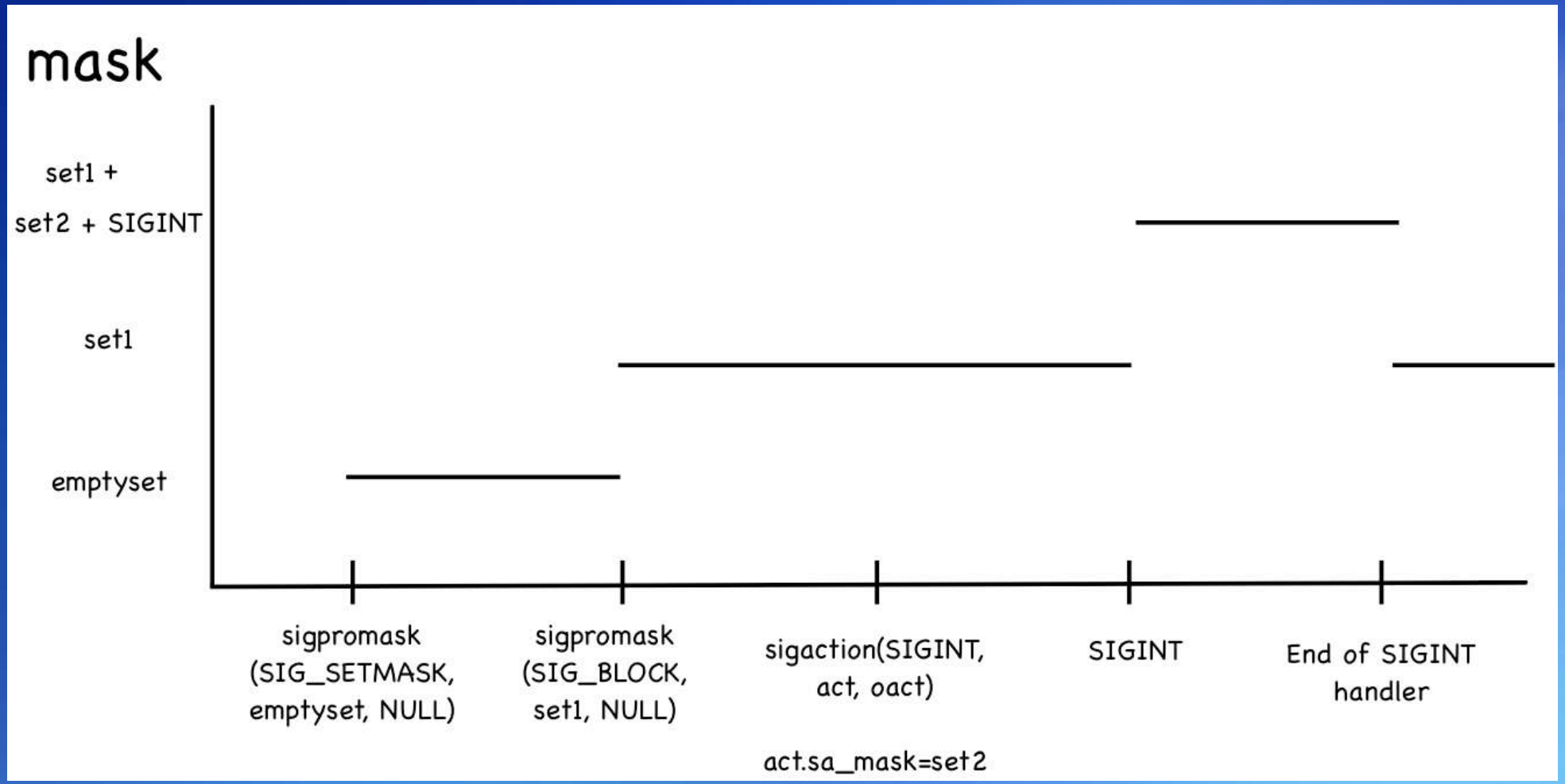
If the SA_SIGINFO flag is set, the signal handler is called as void handler(int *signo*, siginfo_t **info*, void **context*);

```
struct siginfo {
  int     si_signo;  /* signal number */
  int     si_errno;  /* if nonzero, errno value from <errno.h> */
  int     si_code;   /* additional info (depends on signal) */
  pid_t   si_pid;    /* sending process ID */
  uid_t   si_uid;    /* sending process real user ID */
  void  *si_addr;    /* address that caused the fault */
  int     si_status; /* exit value or signal number */
  long    si_band;   /* band number for SIGPOLL */
  /* possibly other fields also */
};
```

siginfo_t codes (partial)

Signal	Code	Reason
	ILL_ILLOPC	illegal opcode
	ILL_ILLOPN	illegal operand
	ILL_ILLADR	illegal addressing mode
SIGILL	ILL_ILLTRP	illegal trap
	ILL_PRVOPC	privileged opcode
	ILL_PRVREG	privileged register
	ILL_COPROC	coprocessor error
	ILL_BADSTK	internal stack error
	FPE_INTDIV	integer divide by zero
	FPE_INTOVF	integer overflow
	FPE_FLTDIV	floating-point divide by zero
	FPE_FLTOVF	floating-point overflow
SIGFPE	FPE_FLTUND	floating-point underflow
	FPE_FLTRES	floating-point inexact result
	FPE_FLTINV	invalid floating-point operation
	FPE_FLTSUB	subscript out of range

Changes of signal mask for sigaction



Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

A reliable implementation of signal using sigaction

```
/* 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) {
#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);
}
```

4.3+BSD: implement signal using sigaction

SVR4: signal() provides the older, unreliable signal semantics

Prevent any interrupted system calls from being restarted

```
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

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

- sigsetjmp() saves the current signal mask of the process in *env* if *savemask* !=0.
- setjmp & longjmp save/restore the signal mask:
  - 4.3+BSD and Mac OS X, but not SVR4, Linux 2.4, Solaris 9

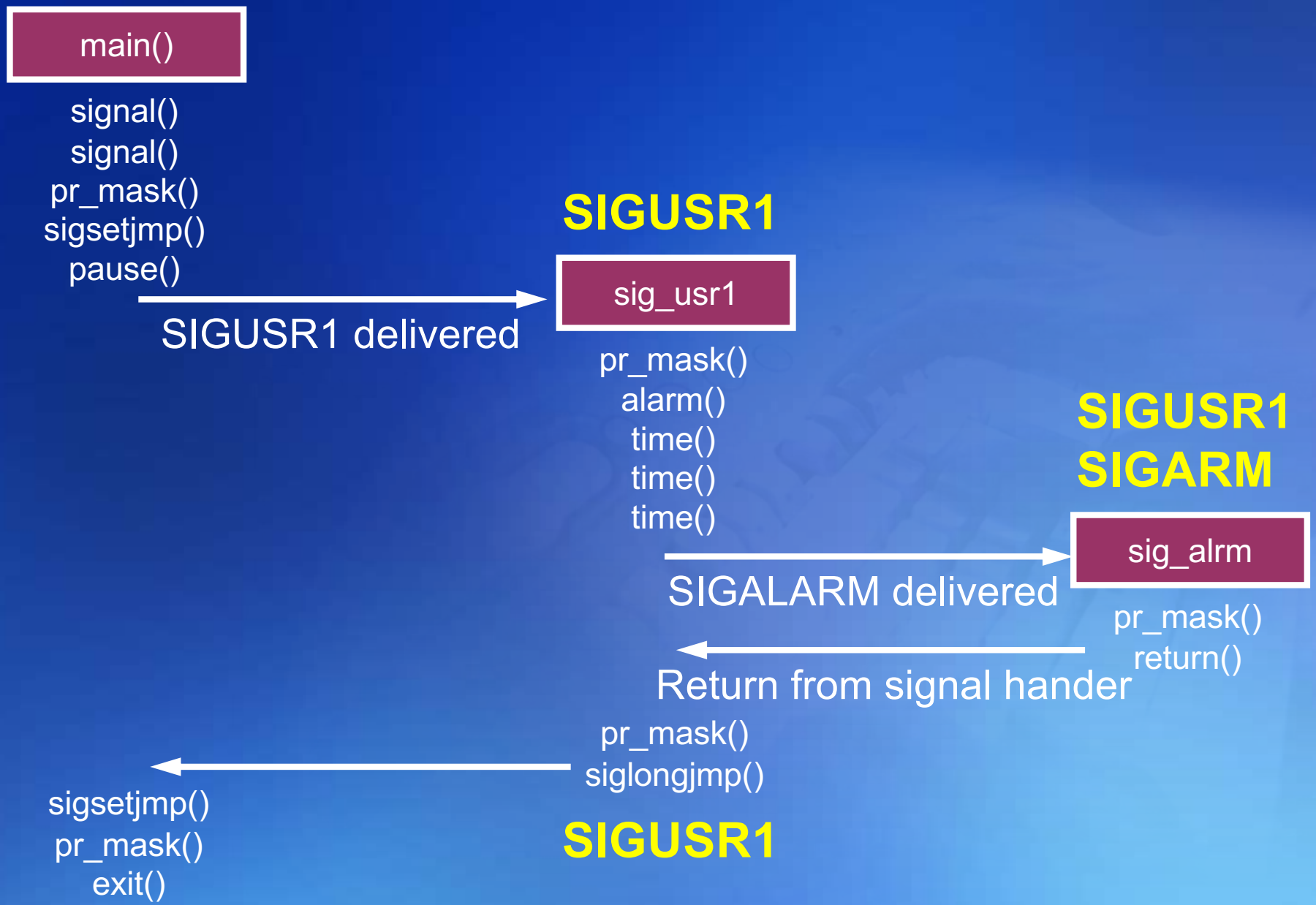
Example of signal masks, sigsetjmp, & siglongjmp

```
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();
}
```

sig_atomic_t – variables of this type could be accessed with a single instruction (no extension across the page boundary)

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

```
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 */
}

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



Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

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

setjmp and longjmp on Linux (or _setjmp and _longjmp on FreeBSD),

```
ending main: SIGUSR1
```

sigsuspend

- Recall that, race condition between alarm() and pause() may block a process forever if there is no other signal.
- Similar race condition may occur between sigprocmask() and pause(), leading to missed signals.

```
sigset_t     newmask, oldmask;

sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);

/* block SIGINT and save current signal mask */
if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
    err_sys("SIG_BLOCK error");

/* critical region of code */

/* reset signal mask, which unblocks SIGINT */
if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
    err_sys("SIG_SETMASK error");

/* window is open */
pause();  /* wait for signal to occur */

/* continue processing */
```

The problem

sigsuspend

int sigsuspend(const sigset_t *sigmask)

#include <signal.h>

```
if (sigprocmask(…) < 0)
  err_sys(…);

pause();
```

CPU Scheduling could occur!

- Reset the signal mask and put the process to sleep in a single atomic operation
- Set the signal mask to *sigmask* and suspend until a signal is caught or until a signal occurs that terminates the process.
- If a signal is caught and if the signal handler returns, then sigsuspend() returns
- Return –1. errno = EINTR if the signal handler returns
- The mask is restored when the function returns.

A correct way to protect a critical region from a signal

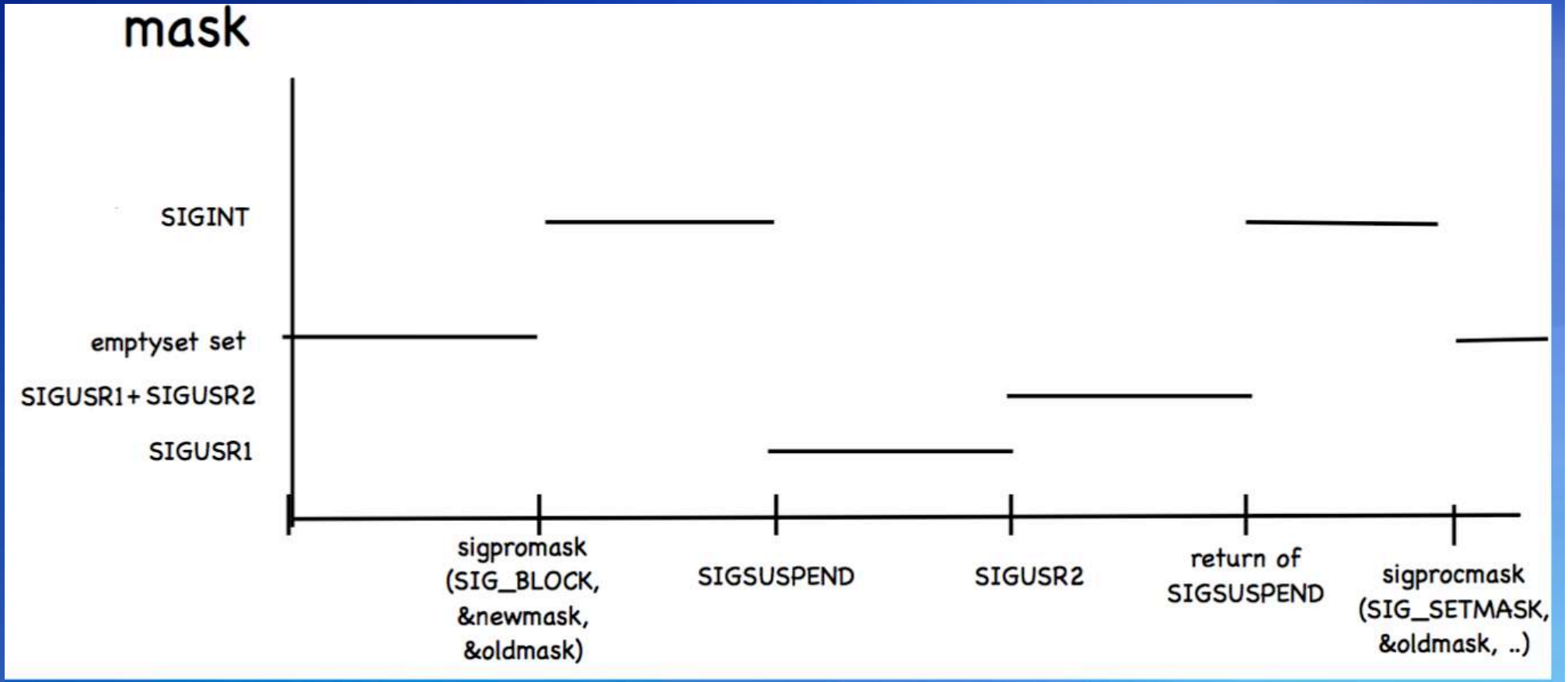
```
static void sig_int(int);

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.
     */
    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: ");
}
```

```
$ ./a.out
program start:
in critical region: SIGINT

^?                                  type the interrupt character
in sig_int: SIGINT SIGUSR1
after return from sigsuspend: SIGINT
program exit:
```

Another Example

```
// supose signal() is implemented by sigaction()
sigset_t  newmask, oldmask, waitmask;

signal(SIGINT, sig_int);
signal(SIGUSR1, sig_int);
signal(SIGUSR2, sig_int);

sigemptyset(&waitmask); sigemptyset(&newmask);
sigaddset(&waitmask, SIGUSR1);
sigaddset(&newmask, SIGINT);

sigprocmask(SIG_BLOCK, &newmask, &oldmask);

sigsuspend(&waitmask); <-------------------- send SIGUSR2 to the process

sigprocmask(SIG_SETMASK, &oldmask, NULL);
```

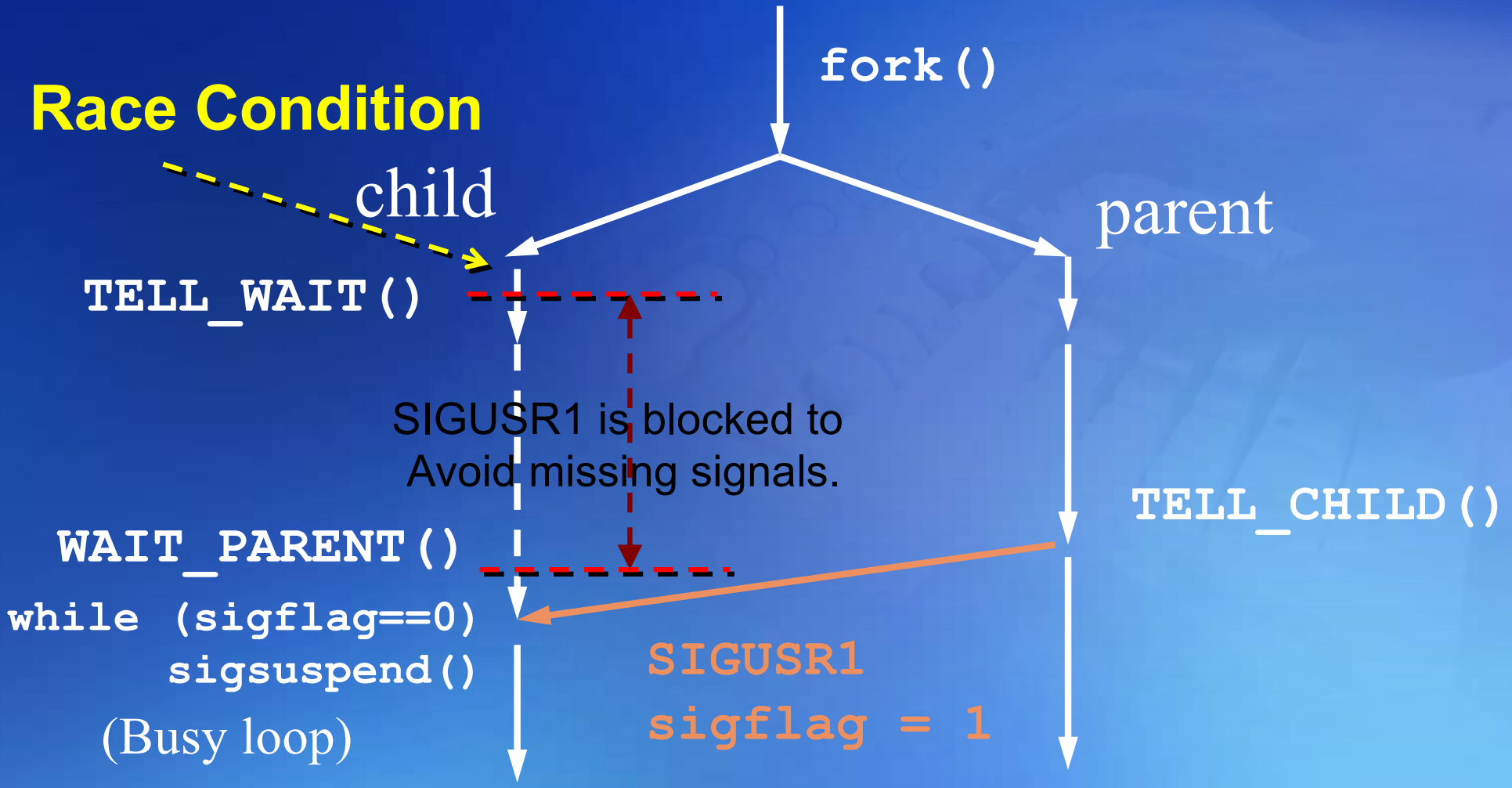
Changes of signal mask



Avoid Race Condition (Chap 8)

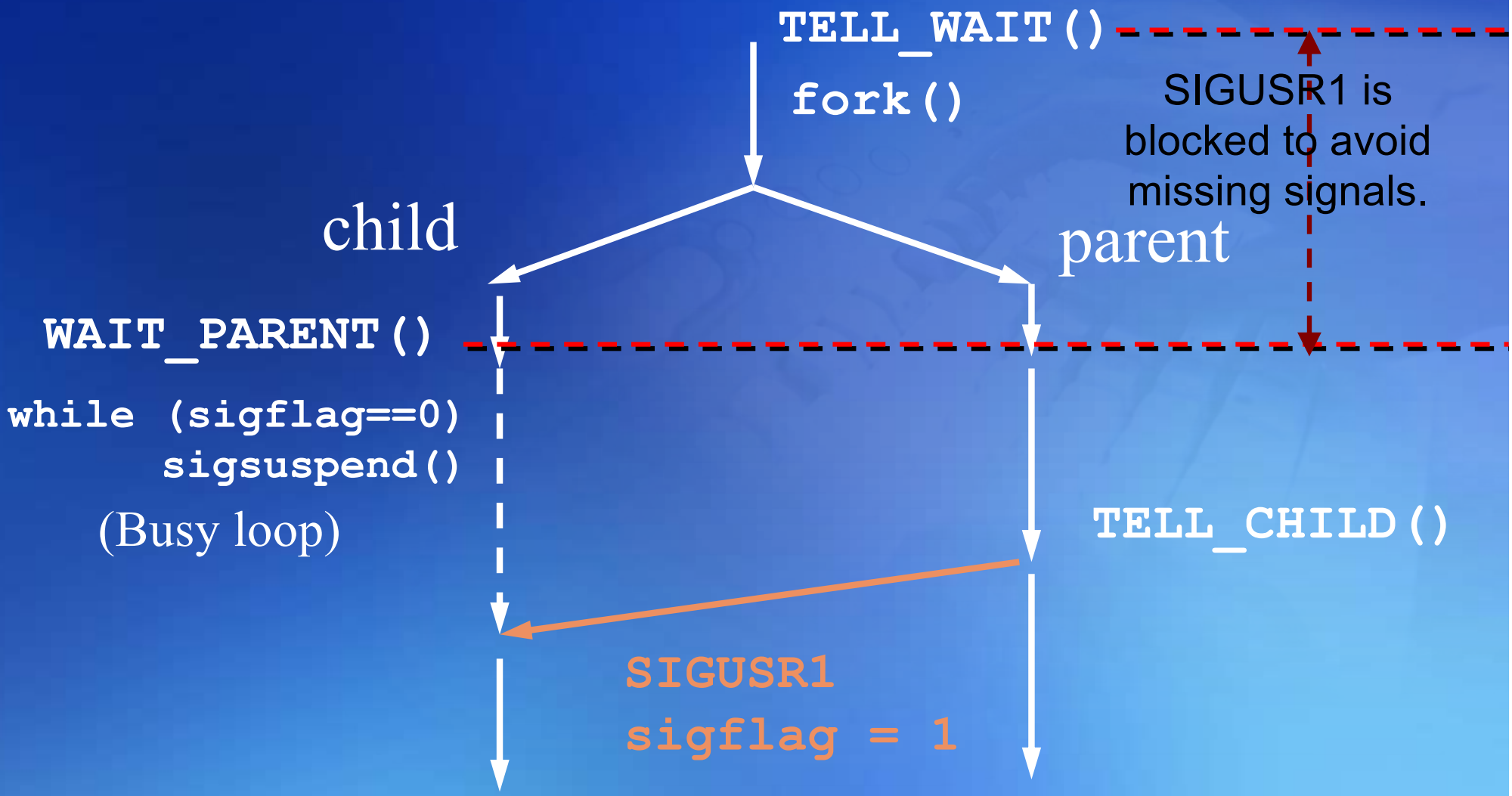
```
    static void charatatime(char *);

    int
    main(void)
    {
        pid_t   pid;

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

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

Process Synchronization (Wrong)



Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Process Synchronization (Correct)



Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Routines to allow a parent and child to synchronize using signals

```
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;
}

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");
}
```

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

void
WAIT_PARENT(void)
{
    while (sigflag == 0)
        sigsuspend(&zeromask);    /* and wait for parent */
    sigflag = 0;

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

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

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");
}
```

sleep

#include <unistd.h>

unsigned int sleep(unsigned int secs);

- Suspend until (1) the specified time elapsed, or (2) a signal is caught and the handler returns - returns the unslept seconds.

- **Problems:**
  - alarm(10), 3 secs, sleep(5)?
    - Another SIGALRM in 2 seconds? Both SVR4 & 4.3BSD – not POSIX.1 requirements
    - Under SVR4, alarm(6), 3 secs, sleep(5) → sleep() returns in 3 secs
  - alarm() & sleep() both use SIGALRM.

Reliable implementation of sleep()

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

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

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

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

    alarm(nsecs);

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

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

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

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

Not handle any interactions with previously set alarms (undefined by POSIX.1)

```
main()
{
   pid_t pid;
   sigset_t newmask, waitmask, oldmask;

   if ((pid = fork()) == 0) {
       struct sigaction action;

       sigemptyset( &newmask );   sigemptyset( &waitmask );
       sigaddset( &newmask, SIGUSR1 );
       sigaddset( &waitmask, SIGUSR2 );
       sigprocmask( SIG_BLOCK, &newmask, &oldmask );

       action.sa_flags = 0;
       sigemptyset( &action.sa_mask );
       action.sa_handler = catchit;
       sigaction( SIGUSR2, &action, NULL );
       sigaction( SIGINT, &action, NULL );
       sigaddset( &action.sa_mask, SIGINT );
       sigaction( SIGUSR1, &action, NULL );

       // Line A: critical section

       sigsuspend( &waitmask );     // Line B

    } else {
       int stat;
       kill( pid, SIGUSR1 );  kill( pid, SIGUSR2 );  kill( pid, SIGINT );
       pid = wait( &stat );
     }
     _exit(0);
}
void catchit(int signo)
{
     // Line C
}
```

(a) (4 points) Although *sigsuspend*() is used here, there is still a race condition. Why?

(b) (6 points) Suppose the child process receives SIGUSR2 at line A, SIGINT when blocked at line B, and SIGUSR1 when SIGINT is caught at line C, how the child process' signal mask changes? Explain your answer.

(c) (8 points) Suppose the child process receives SIGUSR1 at line A, SIGUSR2 when blocked at line B, and SIGINT when SIGUSR1 is caught at line C, how the child process' signal mask changes? Explain your answer.

abort

```
#include <stdlib.h>
```

void abort(void)

- Sends SIGABRT to the process
  - raise(SIGABRT)
- ANSI C requires that if the signal is caught, and the signal handler returns, then abort still doesn't return to its caller.
- The SIGABRT won't return if its handler calls exit, _exit, longjmp, siglongjmp.

abort

- ANSI C
  - The Implementation determines whether output streams are flushed and whether temporary files are deleted.
- POSIX.1
  - POSIX.1 specifies that abort overrides the blocking or ignoring of the signal by the process.
  - If abort() terminates a process, all open standard I/O streams are closed (by fclose()); otherwise, nothing happens.

POSIX.1 implementation of abort()

```
void
abort(void)          /* POSIX-style abort() function */
{
    sigset_t           mask;
    struct sigaction   action;

    /*
     * Caller can't ignore SIGABRT, if so reset to default.
     */
    sigaction(SIGABRT, NULL, &action);
    if (action.sa_handler == SIG_IGN) {
        action.sa_handler = SIG_DFL;
        sigaction(SIGABRT, &action, NULL);
    }
    if (action.sa_handler == SIG_DFL)
        fflush(NULL);           /* flush all open stdio streams */

    /*
     * Caller can't block SIGABRT; make sure it's unblocked.
     */
    sigfillset(&mask);
    sigdelset(&mask, SIGABRT);  /* mask has only SIGABRT turned off */
    sigprocmask(SIG_SETMASK, &mask, NULL);
    kill(getpid(), SIGABRT);    /* send the signal */

    /*
     * If we're here, process caught SIGABRT and returned.
     */
    fflush(NULL);               /* flush all 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 ... */
}
```

What You Should Know

- **What is a signal?**
- **Unreliable signals in multi-tasking environment**
  - Atomic operations are extremely important for handling signals
- **The use of signal to communicate between processes.**
- **Without properly handling other signals in your signal handlers, you may see unexpected results.**
  - Reentrant functions
  - Interrupted system calls
  - abort, sleep (3 versions), sigsetjmp, siglongjmp, etc.
- **Reliable signals**
  - Generation, pending, delivery
  - Signal mask, blocking/unblocking
  - Critical region