CS 33

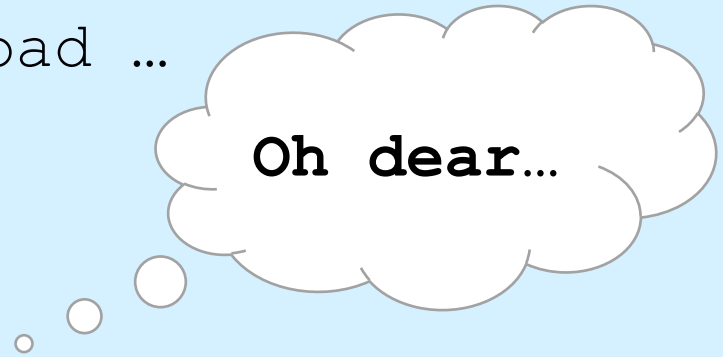
Signals Part 1

Last Words on Files

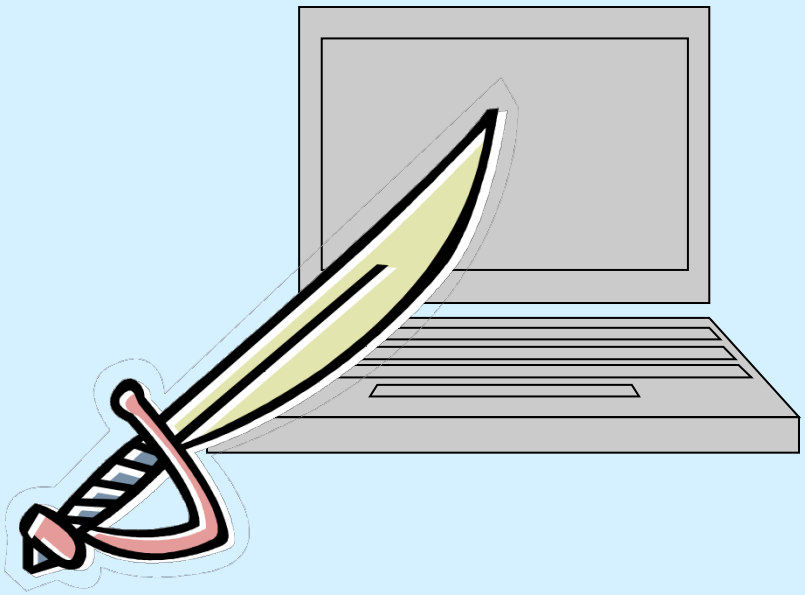
- Sequential access
  - default behavior
- Random access
  - use lseek system call
    - » see man page
- Concurrent access
  - controlled with file locking
  - system guarantees that calls to read and write are *atomic*
    - » if two concurrent writes to the same file
      - effect is as if one takes place in its entirety, then the other
      - either order might happen

Whoops …

```
$ SometimesUsefulProgram xyz
Are you sure you want to proceed? Y
Are you really sure? Y
Reformatting of your disk will begin
in 3 seconds.
Everything you own will be deleted.
There's little you can do about it.
Too bad …
```



One Approach …



A Gentler Approach

- Signals
  - get a process's attention
    - » send it a signal
  - process must either deal with it or be terminated
    - » in some cases, the latter is the only option

Stepping Back …

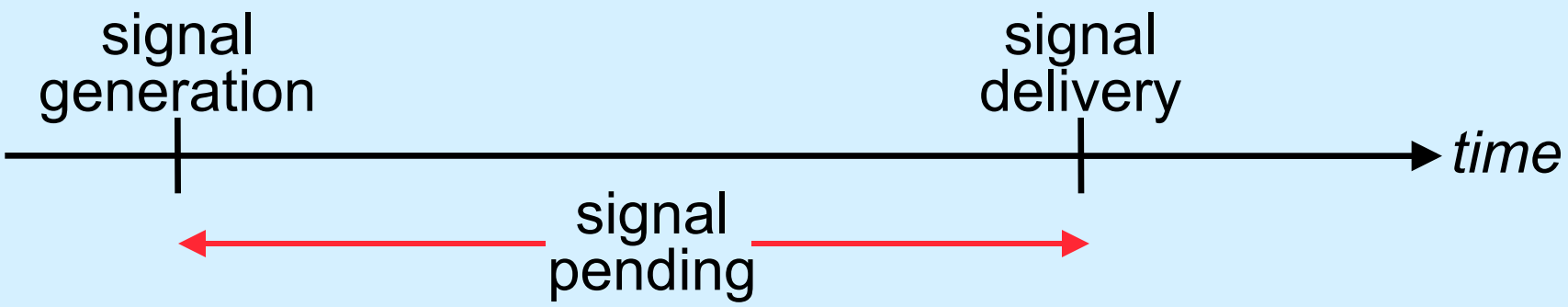
- What are we trying to do?
  - interrupt the execution of a program
    - » cleanly terminate it

      or
    - » cleanly change its course
  - not for the faint of heart
    - » it's difficult
    - » it gets complicated
    - » (not done in Windows)

Signals

- Generated (by OS) in response to
  - exceptions (e.g., arithmetic errors, addressing problems)
    - » synchronous signals
  - external events (e.g., timer expiration, certain keystrokes, actions of other processes)
    - » asynchronous signals
- Effect on process:
  - termination (possibly producing a core dump)
  - invocation of a procedure that has been set up to be a signal handler
  - suspension of execution
  - resumption of execution

Terminology



Signal Types

SIGABRT	*abort* called	term, core
SIGALRM	alarm clock	term
SIGCHLD	death of a child	ignore
SIGCONT	continue after stop	cont
SIGFPE	erroneous arithmetic operation	term, core
SIGHUP	hangup on controlling terminal	term
SIGILL	illegal instruction	term, core
SIGINT	interrupt from keyboard	term
SIGKILL	kill	forced term
SIGPIPE	write on pipe with no one to read	term
SIGQUIT	quit	term, core
SIGSEGV	invalid memory reference	term, core
SIGSTOP	stop process	forced stop
SIGTERM	software termination signal	term
SIGTSTP	stop signal from keyboard	stop
SIGTTIN	background read attempted	stop
SIGTTOU	background write attempted	stop
SIGUSR1	application-defined signal 1	stop
SIGUSR2	application-defined signal 2	stop

Sending a Signal

- `int kill(pid_t pid, int sig)`
  - send signal *sig* to process *pid*
- Also
  - *kill* shell command
  - type ctrl-c
    - » sends signal 2 (SIGINT) to current process
  - type ctrl-\
    - » sends signal 3 (SIGABRT) to current process
  - type ctrl-z
    - » sends signal 20 (SIGTSTP) to current process
  - do something illegal
    - » bad address, bad arithmetic, etc.

Handling Signals

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signo,
    sighandler_t handler);

sighandler_t OldHandler;

OldHandler = signal(SIGINT, NewHandler);
```

Special Handlers

- **SIG_IGN**
  - **ignore the signal**
  - `signal(SIGINT, SIG_IGN);`
- **SIG_DFL**
  - **use the default handler**
    - » **usually terminates the process**
  - `signal(SIGINT, SIG_DFL);`

Example

```
int main() {
  void handler(int);

  signal(SIGINT, handler);
  while(1)
    ;
  return 1;
}
void handler(int signo) {
  printf("I received signal %d. "
     "Whoopee!!\n", signo);
}
```

sigaction

```
int sigaction(int sig, const struct sigaction *new,
              struct sigaction *old);
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
};

int main() {
    struct sigaction act; void myhandler(int);
    sigemptyset(&act.sa_mask); // zeroes the mask
    act.sa_flags = 0;
    act.sa_handler = myhandler;
    sigaction(SIGINT, &act, NULL);
    …
}
```

Example

```
int main() {
  void handler(int);
  struct sigaction act;
  act.sa_handler = handler;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  sigaction(SIGINT, &act, 0);

  while(1)
    ;
  return 1;
}

void handler(int signo) {
  printf("I received signal %d. "
     "Whoopee!!\n", signo);
}
```

Quiz 1

```
int main() {
  void handler(int);
  struct sigaction act;
  act.sa_handler = hand
  sigemptyset(&act.sa_m
  act.sa_flags = 0;
  sigaction(SIGINT, &ac

  while(1)
    ;
  return 1;
}

void handler(int signo) {
  printf("I received signal %d. "
     "Whoopee!!\n", signo);
}
```

You run the example program, then quickly type ctrl-C. What is the most likely explanation if the program then terminates?

a) you're really quick or the system is really slow
b) this "can't happen;" thus there's a problem with the system
c) there's something else going on we haven't yet explained

Getting More Out of Signals (1)

- **Getting more than the signal number**
  - **for example, which arithmetic problem caused a SIGFPE?**
- **Use sa_sigaction rather than sa_handler**

```
struct sigaction act;
act.sa_sigaction = arith_error;
   /* not sa_handler! */
sigemptyset(&act.sa_mask);
act.sa_flags = SA_SIGINFO;
   /* means that we're using sa_sigaction */
sigaction(SIGFPE, &act, 0);
```

Getting More Out of Signals (2)

```
void arith_error(int signo, siginfo_t *infop,
      void *ctx) {

   if (infop->si_code == FPE_INTDIV) {
      /* deal with integer divide by zero */
      …
   }
   …
}
```

Waiting for a Signal …

```
signal(SIGALRM, RespondToSignal);

…

struct timeval waitperiod = {0, 1000};
      /* seconds, microseconds */
struct timeval interval = {0, 0};
struct itimerval timerval;
timerval.it_value = waitperiod;
timerval.it_interval = interval;

setitimer(ITIMER_REAL, &timerval, 0);
      /* SIGALRM sent in ~one millisecond */
pause();  /* wait for it */
printf("success!\n");
```


Quiz 2

This program is guaranteed to print "success!".

a) yes

b) no

```
signal(SIGALRM, RespondToSignal);

…

struct timeval waitperiod = {0, 1000};
      /* seconds, microseconds */
struct timeval interval = {0, 0};
struct itimerval timerval;
timerval.it_value = waitperiod;
timerval.it_interval = interval;

setitimer(ITIMER_REAL, &timerval, 0);
      /* SIGALRM sent in ~one millisecond */
pause();  /* wait for it */
printf("success!\n");
```

Masking Signals

```
setitimer(ITIMER_REAL, &timerval, 0);
      /* SIGALRM sent in ~one millisecond */
```

No signals here, please!

```
pause();  /* wait for it */
```

Masking Signals

mask SIGALRM

```
setitimer(ITIMER_REAL, &timerval, 0);
      /* SIGALRM sent in ~one millisecond */
```

No signals here

unmask and wait for SIGALRM

Doing It Safely

```
sigset_t set, oldset;
sigemptyset(&set);
sigaddset(&set, SIGALRM);
sigprocmask(SIG_BLOCK, &set, &oldset);
      /* SIGALRM now masked */
…
setitimer(ITIMER_REAL, &timerval, 0);
      /* SIGALRM sent in ~one millisecond */

sigsuspend(&oldset);    /* unmask sig and wait */
      /* SIGALRM masked again */

sigprocmask(SIG_SETMASK, &oldset, (sigset_t *)0);
      /* SIGALRM unmasked */
printf("success!\n");
```

Quiz 3

This program is now guaranteed to print "success!".

a) yes

b) no

```
sigset_t set, old
sigemptyset(&set);
sigaddset(&set, SIGALRM);
sigprocmask(SIG_BLOCK, &set, &oldset);
      /* SIGALRM now masked */
…
setitimer(ITIMER_REAL, &timerval, 0);
      /* SIGALRM sent in ~one millisecond */

sigsuspend(&oldset);    /* wait for it safely */
      /* SIGALRM masked again */

sigprocmask(SIG_SETMASK, &oldset, (sigset_t *)0);
      /* SIGALRM unmasked */
printf("success!\n");
```

Signal Sets

- **To clear a set:**

```
int sigemptyset(sigset_t *set);
```

- **To add or remove a signal from the set:**

```
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
```

- **Example: to refer to both SIGHUP and SIGINT:**

```
sigset_t set;

sigemptyset(&set);
sigaddset(&set, SIGHUP);
sigaddset(&set, SIGINT);
```

Masking (Blocking) Signals

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

- used to examine or change the signal mask of the calling process
  - *how* is one of three commands:
    - SIG_BLOCK
      - the new signal mask is the union of the current signal mask and set
    - SIG_UNBLOCK
      - the new signal mask is the intersection of the current signal mask and the complement of set
    - SIG_SETMASK
      - the new signal mask is set

Signal Handlers and Masking

- **What if a signal occurs while a previous instance is being handled?**
  - **inconvenient …**
- **Signals are masked while being handled**
  - **may mask other signals as well:**

```
struct sigaction act; void myhandler(int);
sigemptyset(&act.sa_mask); // zeroes the mask
sigaddset(&act.sa_mask, SIGQUIT);
   // also mask SIGQUIT
act.sa_flags = 0;
act.sa_handler = myhandler;
sigaction(SIGINT, &act, NULL);
```

Timed Out!

```
int TimedInput( ) {
    signal(SIGALRM, timeout);
    …
    alarm(30);     /* send SIGALRM in 30 seconds */
    GetInput();    /* possible long wait for input */
    alarm(0);      /* cancel SIGALRM request */
    HandleInput();
    return(0);
nogood:
    return(1);
}

void timeout( ) {
    goto nogood;   /* not legal but straightforward */
}
```

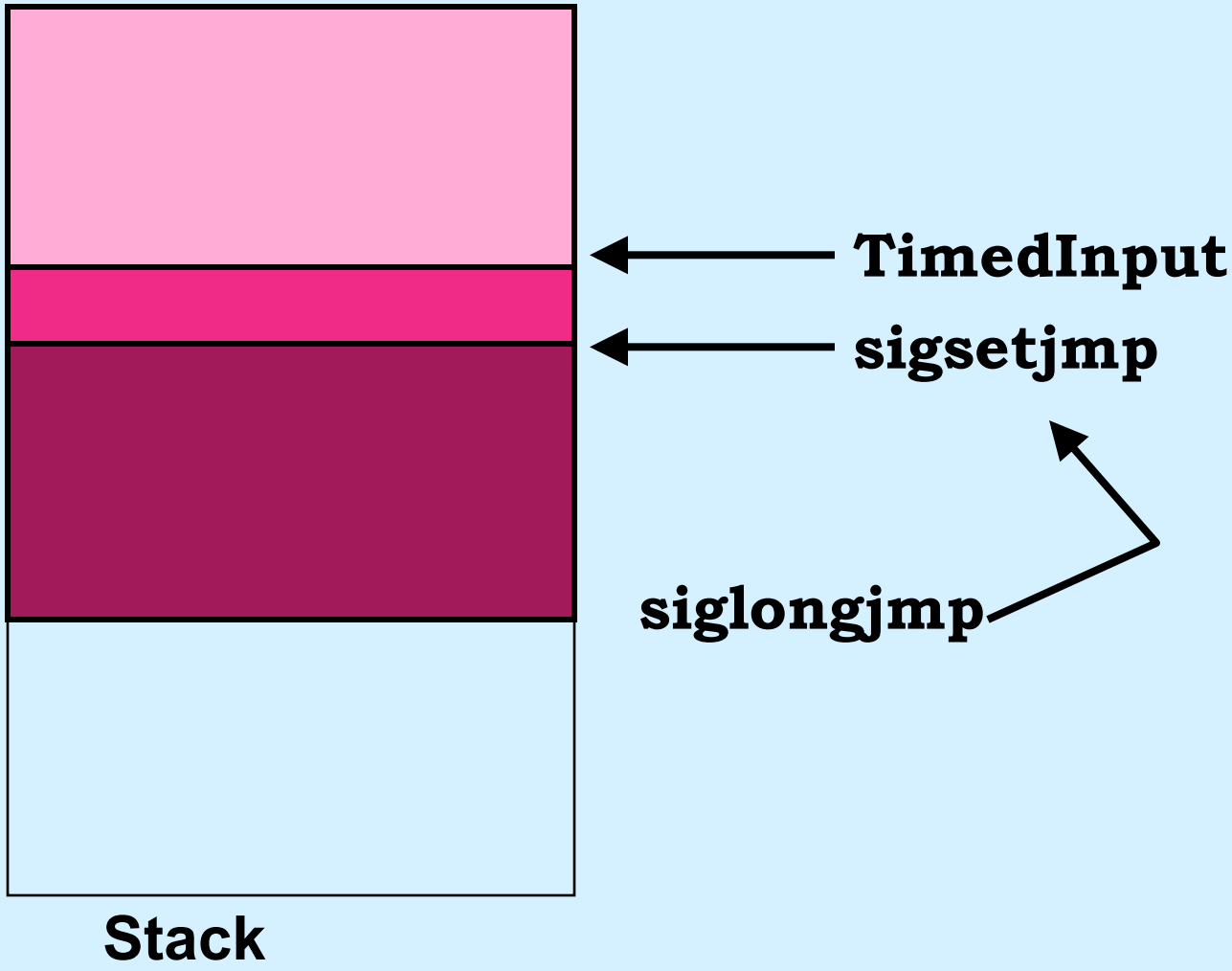

Doing It Legally (but Weirdly)

```
sigjmp_buf context;

int TimedInput( ) {
   signal(SIGALRM, timeout);
   if (sigsetjmp(context, 1) == 0) {
      alarm(30);  // cause SIGALRM in 30 seconds
      GetInput(); // possible long wait for input
      alarm(0);   // cancel SIGALRM request
      HandleInput();
      return 0;
   } else
      return 1;
}

void timeout() {
   siglongjmp(context, 1);    /* legal but weird */
}
```

sigsetjmp/siglongjmp



Quiz 4

```
sigjmp_buf ctx;
int SaveIt() {
  return sigsetjmp(ctx, 1);
}

int TimedInput() {
  ...
  if (SaveIt() == 0) {
      alarm(30);
      GetInput();
      alarm(0);
      HandleInput();
      return 0;
  } else return 1;
}
```

Does this work?

a) yes
b) no

```
void timeout() {
   siglongjmp(ctx, 1);
}
```

Exceptions

- **Other languages support exception handling**

```
try {
  something_a_bit_risky();
} catch(ArithmeticException e) {
  deal_with_it(e);
}
```

- **Can we do something like this in C?**

Exception Handling in C

```
void Exception(int sig) {
  THROW(sig)
}

int computation(int a) {
  return a/(a-a);
}
```

```
int main() {
  signal(SIGFPE, Exception);
  signal(SIGSEGV, Exception);
  TRY {
    computation(1);
  } CATCH(SIGFPE) {
    fprintf(stderr,
      "SIGFPE\n");
  } CATCH(SIGSEGV) {
    fprintf(stderr,
      "SIGSEGV\n");
  } END

  return 0;
}
```

Exception Handling in C

```
#define TRY \
 { \
   int excp; \
   if ((excp = \
     sigsetjmp(ctx, 1)) == 0)

#define CATCH(a_excp) \
   else if (excp == a_excp)

#define END }

#define THROW(excp) \
  siglongjmp(ctx, excp);
```

Exception Handling in C

```
sigjmp_buf ctx;

int main() {
 ...
 {
   int excp;
   if ((excp = sigsetjmp(ctx, 1)) == 0) {
     computation(1);
   } else if (excp == SIGFPE) {
     fprintf(stderr, "SIGFPE\n");
   } else if (excp == SIGSEGV) {
     fprintf(stderr, "SIGFPE\n");
   }
 }
 return 0;
}
```

```
void exception(int sig) {
 siglongjmp(ctx, sig);
}
```

THROW

TRY

CATCH

CATCH

END