CS 33

Data Representation (Part 4)

Normalized Encoding Example

- **Value: `float F = 15213.0;`**
  - 15213_{10} = 11101101101101_2
    = $1.1101101101101_2 \times 2^{13}$

- **Significand**
  - M = $1.\underline{1101101101101}_2$
  - `frac =` $\underline{1101101101101}0000000000_2$

- **Exponent**
  - E = **13**
  - *bias* = **127**
  - *exp* = **140** = 10001100_2

- **Result:**

0	10001100	11011011011010000000000
s	exp	frac
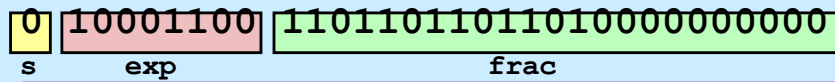
Denormalized Values

- **Condition: exp = 000…0**
- **Exponent value: E = 1 – Bias (instead of E = 0 – Bias)**
- **Significand coded with implied leading 0:**
  M = 0.xxx…x$_2$
  - **xxx…x: bits of frac, range [0,1)**
- **Cases**
  - **exp = 000…0, frac = 000…0**
    - » **represents zero value**
    - » **note distinct values: +0 and –0 (why?)**
  - **exp = 000…0, frac ≠ 000…0**
    - » **numbers closest to 0.0**
    - » **for S.P., range from .111...1 * 2^{-126} to .000...001 * 2^{-126}**
    - » **smallest normalized value is 1.0 * 2^{-126}**

Supplied by CMU.

For denormalized values, there's a single exponent value, which is 1- Bias. The significand is in a range of values greater than or equal to zero, but less than one.

For normalized values, as the numbers we wish to represent get smaller, we simply subtract one from the exponent. But with denormalized values, the exponent is as small as it can get. Thus to represent even smaller values, the significand does not start with an implied one, but with zero. The smallest positive single-precision normalized value is $1.0 * 2^{-126}$. The largest single-precision denormalized value is $.11111111111111111111111 * 2^{-126}$. The smallest non-zero single-precision denormalized value is $.00000000000000000000001 * 2^{-126}$.

Special Values

- **Condition: $\texttt{exp} = 111\ldots1$**

- **Case: $\texttt{exp} = 111\ldots1$, $\texttt{frac} = 000\ldots0$**
  - represents value ∞ (infinity)
  - operation that overflows
  - both positive and negative
  - e.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$

- **Case: $\texttt{exp} = 111\ldots1$, $\texttt{frac} \neq 000\ldots0$**
  - not-a-number (NaN)
  - represents case when no numeric value can be determined
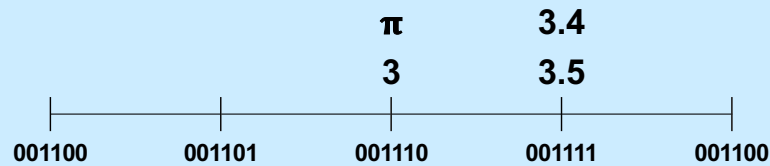 - e.g., $\text{sqrt}(-1)$, $\infty - \infty$, $\infty \times 0$

Visualization: Floating-Point Encodings

−∞   −Normalized   −Denorm   +Denorm   +Normalized   +∞

NaN   −0   +0   NaN
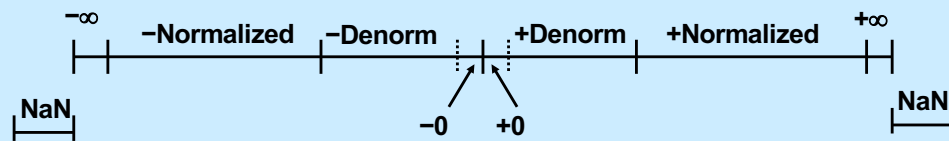
Mapping Real Numbers to Float

- **The real number 3 is represented as 0 011 10**
- **The real number 3.5 is represented as 0 011 11**
- **How is the real number 3.4 represented?**

  0 011 11
- **How is the real number π represented?**

  0 011 10

|		π	3.4	
		3	3.5	
001100	001101	001110	001111	001100

For the sake of this slide and example, assume that we have a six-bit representation of floating-point numbers. In this encoding there is one sign bit, 3 exponent bits (with a bias of 3) and 2 fraction bits. Thus 0 011 10 is $2^{3-3} * 1.5$.

Mapping Real Numbers to Float

- **If R is a real number, it's mapped to the floating-point number whose value is closest to R**

Floats are Sets of Values

- **If A, B, and C are successive floating-point values**
  - **e.g., 010001, 010010, and 010011**
- **B represents all real numbers from midway between A and B through midway between B and C**

Real numbers represented by B

A   B   C
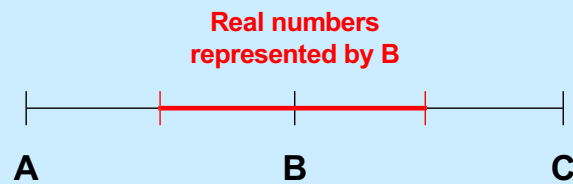
What about values that are equidistant from A and B or from B and C? There are rules for rounding such values that we don't have time to get into.

A special case is 0. Positive 0 represents a range of values that are greater than or equal to 0. Negative 0 represents a range of values that are less than or equal to zero.

+/− Zero

- **Only one zero for ints**
  - **an int is a single number, not a range of numbers, thus there can be only zero**
- **Floating-point zero**
  - **a range of numbers around the real 0**
  - **it really matters which side of 0 we're on!**
    - » **a very large negative number divided by a very small negative number should be positive**

      $-\infty/-0 = +\infty$
    - » **a very large positive number divided by a very small negative number should be negative**

      $+\infty /-0 = -\infty$

It's important to remember that a floating-point value is not a single number, but a range of numbers.

Significance

- **Normalized numbers**
  - **for a particular exponent value E and an S-bit significand, the range from 2^E up to 2^{E+1} is divided into 2^S equi-spaced floating-point values**
    - » **thus each floating-point value represents $1/2^S$ of the range of values with that exponent**
    - » **all bits of the signifcand are important**
    - » **we say that there are S significant bits – for reasonably large S, each floating-point value covers a rather small part of the range**
      - **high accuracy**
      - **for S=23 (32-bit float), accurate to one in 2^{23} (.0000119% accuracy)**

Significance

- **Unnormalized numbers**
  - **high-order zero bits of the significand aren't important**
  - **in 32-bit floating point, 0 00000000 00000000000000000000001 represents 2^{-149}**
    - » **it is the only value with that exponent: 1 significant bit (either 2^{-149} or 0)**
  - **0 00000000 00000000000000000000010 represents 2^{-148}**
    0 00000000 00000000000000000000011 represents $1.5*2^{-148}$
    - » **only two values with exponent -148: 2 significant bits (encoding those two values, as well as 2^{-149} and 0)**
  - **fewer significant bits mean less accuracy**
  - **0 00000000 00000000000000000000001 represents a range of values from $.5*2^{-9}$ to $1.5*2^{-9}$**
  - **50% accuracy**

Recall that the bias for the exponent of 8-bit IEEE FP is 7, thus for unnormalized numbers the actual exponent is -6 (-bias+1). The significand has an implied leading 0, thus 0 0000 001 represents $2^{-6} * 2^{-3}$.

With 8-bit IEEE FP. the value 0 0000 01 is interpreted as 2-9, But the number represented could be 50% or 50% more.

Floating Point

- **Single precision (float)**

s	exp	frac

   1   8-bits   23-bits

      – **range: $\pm1.8\times10^{-38} - \pm3.4\times10^{38}$, ~7 decimal digits**

- **Double Precision (double)**

s	exp	frac

   1   11-bits   52-bits

      – **range: $\pm2.23\times10^{-308} - \pm1.8\times10^{308}$, ~16 decimal digits**

Quiz 1

Suppose *f*, declared to be a `float`, is assigned the largest possible floating-point positive value (other than +∞). What is the value of *g = f+1.0*?

    a) 0

    b) f

    c) +∞

    d) NaN

Float is not Rational …

- **Floating addition**
  - commutative: $a +_f b = b +_f a$
    - » yes!
  - associative: $a +_f (b +_f c) = (a +_f b) +_f c$
    - » no!
      - $2 +_f (1e38 +_f -1e38) = 2$
      - $(2 +_f 1e38) +_f -1e38 = 0$

Note that the floating-point numbers in this and the next two slides are expressed in base 10, not base 2.

In this and the next few slides, $+_f$ means floating-point addition (as opposed to addition of real numbers) and $*_f$ means floating-point multiplication.

Float is not Rational …

- **Multiplication**
  - commutative: $a *_f b = b *_f a$
    - » yes!
  - associative: $a *_f (b *_f c) = (a *_f b) *_f c$
    - » no!
      - $1\mathrm{e}37 *_f (1\mathrm{e}37 *_f 1\mathrm{e}\text{-}37) = 1\mathrm{e}37$
      - $(1\mathrm{e}37 *_f 1\mathrm{e}37) *_f 1\mathrm{e}\text{-}37 = +\infty$

Float is not Rational …

- **More …**
  - **multiplication distributes over addition:**
    $a *_f (b +_f c) = (a *_f b) +_f (a *_f c)$
    - » **no!**
    - » $1e38 *_f (1e38 +_f -1e38) = 0$
    - » $(1e38 *_f 1e38) +_f (1e38 *_f -1e38) = NaN$
  - **insignificance:**
    $x = y +_f 1$
    $z = 2 /_f (x -_f y)$
    $z == 2?$
    - » **not necessarily!**
      - • **consider y = 1e38**

If y is 1e38 and we're using single-precision floating-point arithmetic, then z would be $+\infty$ (since $x -_f y$ would be 0).

CS 33

Signals Part 1

An Interlude Between Shells

- **Shell 1**
  - it can run programs
  - it can redirect I/O
- **Signals**
  - a mechanism for coping with exceptions and external events
  - the mechanism needed for shell 2
- **Shell 2**
  - it can control running programs

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 …
```

Oh dear…

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 function that has been set up to be a signal handler
  - suspension of execution
  - resumption of execution

Signals are a kernel-supported mechanism for reporting events to user code and forcing a response to them. There are actually two sorts of such events, to which we sometimes refer as **exceptions** and **interrupts**. The former occur typically because the program has done something wrong. The response, the sending of a signal, is immediate; such signals are known as **synchronous** signals. The latter are in response to external actions, such as a timer expiring, an action at the keyboard, or the explicit sending of a signal by another process. Signals send in response to these events can seemingly occur at any moment and are referred to as **asynchronous** signals.

Processes react to signals using the actions shown in the slide. The action taken depends partly on the signal and partly on arrangements made in the process beforehand.

A core dump is the contents of a process's address space, written to a file (called **core**), reflecting what the situation was when it was terminated by a signal. They can be used by gdb to see what happened (e.g., to get a backtrace). Since they're fairly large and rarely looked at, they're normally disabled. We'll look at them further shortly.

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**

This slide shows the complete list of signals required by POSIX 1003.1, the official Unix specification. In addition, many Unix systems support other signals, some of which we'll mention in the course. The third column of the slide lists the default actions in response to each of the signals. **term** means the process is terminated, **core** means there is also a core dump; **ignore** means that the signal is ignored; **stop** means that the process is stopped (suspended); **cont** means that a stopped process is resumed (continued); **forced** means that the default action cannot be changed and that the signal cannot be blocked or ignored.

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 (SIGQUIT) to current process**
  - **type ctrl-z**
    - » **sends signal 20 (SIGTSTP) to current process**
  - **do something bad**
    - » **bad address, bad arithmetic, etc.**

Note that the signals generated by typing control characters on the keyboard are actually sent to the current process group of the terminal, a concept we discuss soon.

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);
```

The **signal** function establishes a new handler for the given signal and returns the address of the previous handler.

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

```
void sigloop() {
  while(1)
    ;
}

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

Note that the C compiler implicitly concatenates two adjacent strings, as done in printf above.

Don't forget to delete the core files when you're finished with them! Note that neither OSX nor Windows supports core dumps.

Some details on the **ulimit** command: its supports both a hard limit (which can't be modified) and a soft limit (which can later be modified). By default, **ulimit** sets both the hard and soft limits. Thus typing

```
ulimit -c 0
```

sets both the hard and soft limits of core file size to 0, meaning that you can't increase the limit later (within the execution of the current invocation of this shell).

But if you type

```
ulimit -Sc 0
```

then just the soft limit is modified, allowing you to type

```
ulimit -c unlimited
```

later.

The **sigaction** system call is the the more general means for establishing a process's response to a particular signal. Its first argument is the signal for which a response is being specified, the second argument is a pointer to a **sigaction** structure defining the response, and the third argument is a pointer to memory in which a **sigaction** structure will be stored containing the specification of what the response was prior to this call. If the third argument is null, the prior response is not returned.

The **sa_handler** member of **sigaction** is either a pointer to a user-defined handler function for the signal or one of SIG_DFL (meaning that the default action is taken) or SIG_IGN (meaning that the signal is to be ignored). The **sig_action** member is an alternative means for specifying a handler function; we won't get a chance to discuss it, but it's used when more information about the cause of a signal is needed.

When a user-defined signal-handler function is entered in response to a signal, the signal itself is masked until the function returns. Using the **sa_mask** member, one can specify additional signals to be masked while the handler function is running. On return from the handler function, the process's previous signal mask is restored.

The **sa_flags** member is used to specify various other things which we describe in upcoming slides.

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);
}
```

This has behavior identical to the previous example; we're using **sigaction** rather than *signal* to set up the signal handler.

Quiz 2

```c
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) this "can't happen"; thus there's a problem with the system
  b) you're really quick or the system is really slow (or both)
  c) what we've told you so far isn't quite correct

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");
```

Here we use the **setitimer** system call to arrange so that a SIGALRM signal is generated in one millisecond. (The system call takes three arguments: the first indicates how time should be measured; what's specified here is to use real time. See its man page for other possibilities. The second argument contains a **struct itimerval** that itself contains two **struct timevals**. One (named **it_value**) indicates how much time should elapse before a SIGALRM is generated for the process. The other (named **it_interval**), if non-zero, indicates that a SIGALRM should be sent again, repeatedly, every **it_interval** period of time. Each process may have only one pending timer, thus when a process calls **setitimer**, the new value replaces the old. If the third argument to **setitimer** is non-zero, the old value is stored at the location it points to.)

The **pause** system call causes the process to block (go to sleep) and not resume until **some** signal that is not ignored is delivered.

Quiz 3

This program is guaranteed to print "success!".
    a) no
    b) yes

```
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

If a signal is masked, then, if it occurs, it's not immediately applied to the process, but will be applied when it's no longer masked.

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");
```

Here's a safer way of doing what was attempted in the earlier slide. We mask the SIGALRM signal before calling **setitimer**. Then, rather than calling *pause*, we call **sigsuspend**, which sets the set of masked signals to its argument and, at the same instant, blocks the calling process. Thus if the SIGALRM is generated before our process calls **sigsuspend**, it won't be delivered right away. Since the call to **sigsuspend** reinstates the previous mask (which, presumably, did not include SIGALRM), the SIGALRM signal will be delivered and the process will return (after invoking the handler). When **sigsuspend** returns, the signal mask that was in place just before it was called is restored. Thus we have to restore **oldset** explicitly.

As with **pause**, **sigsuspend** returns only if an unmasked signal that is not ignored is delivered.

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);
  ```

A number of signal-related operations involve sets of signals. These sets are normally represented by a bit vector of type **sigset_t**.

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**

In addition to ignoring signals, you may specify that they are to be blocked (that is, held pending or masked). When a signal type is masked, signals of that type remains pending and do not interrupt the process until they are unmasked. When the process unblocks the signal, the action associated with any pending signal is performed. This technique is most useful for protecting critical code that should not be interrupted. Also, as we've already seen, when the handler for a signal is entered, subsequent occurrences of that signal are automatically masked until the handler is exited, hence the handler never has to worry about being invoked to handle another instance of the signal it's already handling.

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);
```

```
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 */
}
```

This slide sketches something that one might want to try to do: give a user a limited amount of time (in this case, 30 seconds — the **alarm** function causes the system to send the process a SIGALRM signal in the given number of seconds) to provide some input, then, if no input, notify the caller that there is a problem. Here we'd like our timeout handler to transfer control to someplace else in the program, but we can't do this. (Note also that we should cancel the call to **alarm** if there is input. So that we can fit all the code in a single slide, we've left this part out.)

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 */
}
```

To get around the problem of not being able to use a **goto** statement to get out of a signal handler, we introduce the **setjmp/longjmp** facility, also known as the *nonlocal goto*. A call to **sigsetjmp** stores context information (about the current locus of execution) that can be restored via a call to **siglongjmp**. A bit more precisely: **sigsetjmp** stores into its first argument the values of the program-counter (instruction-pointer), stack-pointer, and other registers representing the process's current execution context. If the second argument is non-zero, the current signal mask is saved as well. The call returns 0. When **siglongjmp** is called with a pointer to this context information as its first argument, the current register values are replaced with those that were saved. If the signal mask was saved, that is restored as well. The effect of doing this is that the process resumes execution where it was when the context information was saved: inside of **sigsetjmp**. However, this time, rather than returning zero, it returns the second argument passed to **siglongjmp** (1 in the example).

To use this facility, you must include the header file **setjmp.h**.

sigsetjmp/siglongjmp
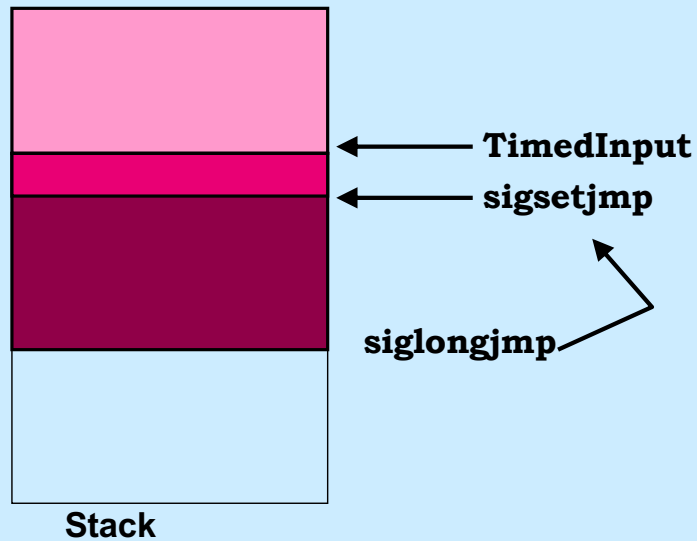
Stack

The effect of **sigsetjmp** is to save the registers relevant to the current stack frame; in particular, the instruction pointer, the base pointer (if used), and the stack pointer, as well as the return address and the current signal mask. A subsequent call to **siglongjmp** restores the stack to what it was at the time of the call to **sigsetjmp**. Note that **siglongjmp** should be called only from a stack frame that is farther on the stack than the one in which **sigsetjmp** was called.