2.2.4 Thread Safety

Thread Safety

- Unix was developed way before threads were commonly used
  - Unix libraries were built without threads in mind
  - running code using these library functions with threads became unsafe
  - to make these library functions safe to run under *multithreading* is known as *Thread Safety*

- General problems with the old Unix API
  - global variables, e.g., `errno`
  - shared data, e.g., `printf()`

Thread Safety vs. Reentrancy

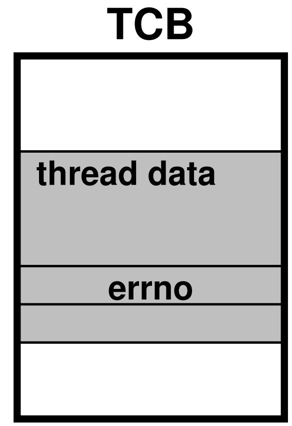
- Strictly speaking, making a function *thread-safe* is not the same as making it *reentrant*
  - *thread-safe:* multiple threads can call the function in parallel or concurrently
  - *reentrant:* enter twice (even if you have only one thread)
    - how do you enter a function twice if you are not running multiple threads?
      - for a *kernel function*, you can get interrupted and call the same funciton inside an interrupt service routine
      - for a *user space function*, you can get interrupted and the kernel makes an *upcall* that calls the same funciton
  - most of the time, making a function thread-safe and making it reentrant ends up to be the same thing
    - but you need to be careful
    - you can google "reentrancy" to see the difference between reentrant code and thread-safe code and see examples
  - we focus on *multi-threading* (and "no signal handlers")

Global Variables

```
int IOfunc(int fd) {
  extern int errno;
  ...
  if (write(fd, buffer, size) == -1) {
    if (errno == EIO)
      fprintf(stderr, "IO problems ...\n");
    ...
    return(0);
  }
  ...
}
```

- if 2 threads call this function and both failed, how do you guarantee that a thread would get the right `errno`?
  - the code is *not "thread safe"*
- `errno` is a system-call level *global variable*
  - Unix system-call library was implemented before multi-threading was a common practice

Coping



- Fix Unix's C/system-call interface
  - want backwards compatibility
- Make `errno` refer to a different location in each thread
  - e.g.,

```
#define errno __errno(thread_ID)
```

  - `__errno(thread_ID)` will return the *thread-specific* `errno`
    - need a place to store this thread-specific `errno`
    - POSIX threads provides a general mechanism to store *thread-specific data*
      - Win32 has something similar called thread-local storage
    - POSIX does not specify how this private storage is allocated and organized
      - done with an array of `(void*)`
      - then `errno` would be at a fixed index into this array
  - don't need to change application, just recompile

Add "Reentrant" Version Of System Call

- `gethostbyname()` system call is not "thread safe"

```
struct hostent *gethostbyname(const char *name)
```

  - it returns a pointer to a global variable
    - (what a terrible idea!)
  - POSIX's fix for this problem is to add a function to the system library

```
int gethostbyname_r(const char *name,
                    struct hostent *ret,
                    char *buf,
                    size_t buflen,
                    struct hostent **result,
                    int *h_errnop)
```

    - caller of this function must provide the buffer to hold the return data
      - (a good idea in general)
    - caller is aware of thread-safety
      - (a more educated programmer is desirable)

Shared Data

- **Thread 1:**

```
printf("goto statement reached");
```

- **Thread 2:**

```
printf("Hello World\n");
```

- **Printed on display:**

```
goto Hello Wostatement reachedrld
```

Coping

- Wrap library calls with synchronization constructs
- Fix the libraries
- Application can use a mutex
- If application is using the `(FILE*)` object in `<stdio.h>`, can wrap functions like `printf()` around these functions

```
void flockfile(FILE *filehandle)
int ftrylockfile(FILE *filehandle)
void funlockfile(FILE *filehandle)
```

  - basically, `flockfile()` would block until lockcount is 0
    - then it increments the lockcount
  - `funlockfile()` decrements the lockcount

Killing Time ...

- To suspend your thread for a certain duration

```
struct timespec timeout, remaining_time;

timeout.tv_sec = 3;      // seconds
timeout.tv_nsec = 1000; // nanoseconds
nanosleep(&timeout, &remaining_time);
```

  - Unix/Linux is "best-effort"
  - okay to do this in `warmup2` since it only has to run on Ubuntu

- The right way to sleep is to say when you want to wake up, e.g.,

```
int pthread_cond_timedwait(
            pthread_cond_t *cond,
            pthread_mutex_t *mutex,
            struct timespec *abstime)
```

  - after `abstime`, give up waiting for an event and return with an error code
  - you need to calculate `abstime` carefully and correctly

Timeouts

```
struct timespec relative_timeout, absolute_timeout;
struct timeval now;
relative_timeout.tv_sec = 3;      // seconds
relative_timeout.tv_nsec = 1000; // nanoseconds
gettimeofday(&now, 0);
absolute_timeout.tv_sec = now.tv_sec +
    relative_timeout.tv_sec;
absolute_timeout.tv_nsec = 1000*now.tv_usec +
    relative_timeout.tv_nsec;
if (absolute_timeout.tv_nsec >= 1000000000) {
  // deal with the carry
  absolute_timeout.tv_nsec -= 1000000000;
  absolute_timeout.tv_sec++;
}
pthread_mutex_lock(&m);
while (!may_continue)
  pthread_cond_timedwait(&cv, &m, &absolute_timeout);
pthread_mutex_unlock(&m);
```

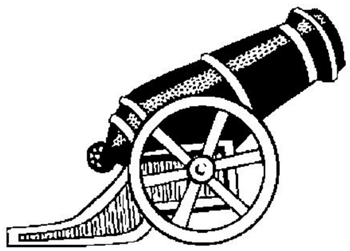
- must check return code of `pthread_cond_timedwait()`

2.2.5 Deviations

Deviations

- How do you ask another thread to deviate from its normal execution path?
  - Unix's *signal* mechanism
- How do you force another thread to terminate cleanly
  - POSIX *cancellation* mechanism

Signals



```
int x, y;

x = 0;
...
y = 16/x;
```

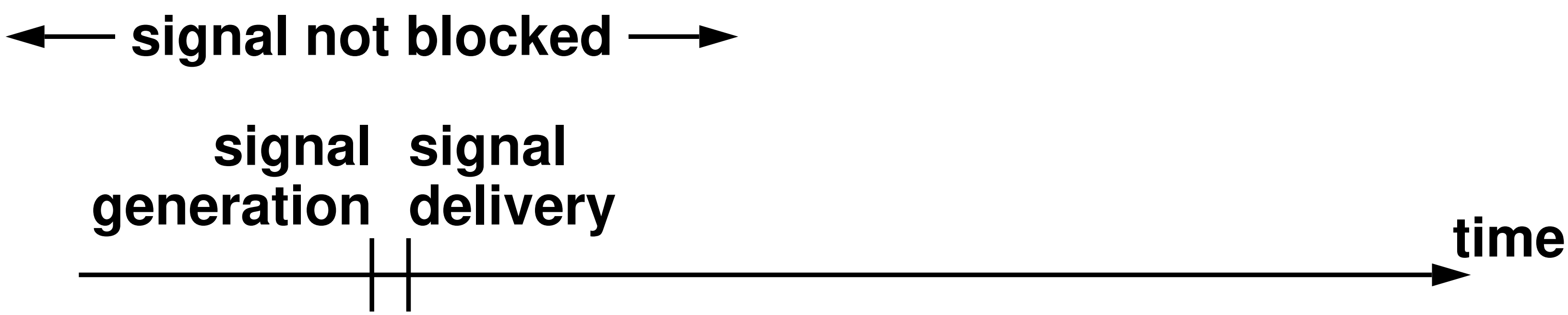
```
for (;;)
  keep_on_trying( );
```

- the original intent of Unix signals was to force the ***graceful termination*** of a process
  - e.g., <Ctrl+C>

The OS to the Rescue

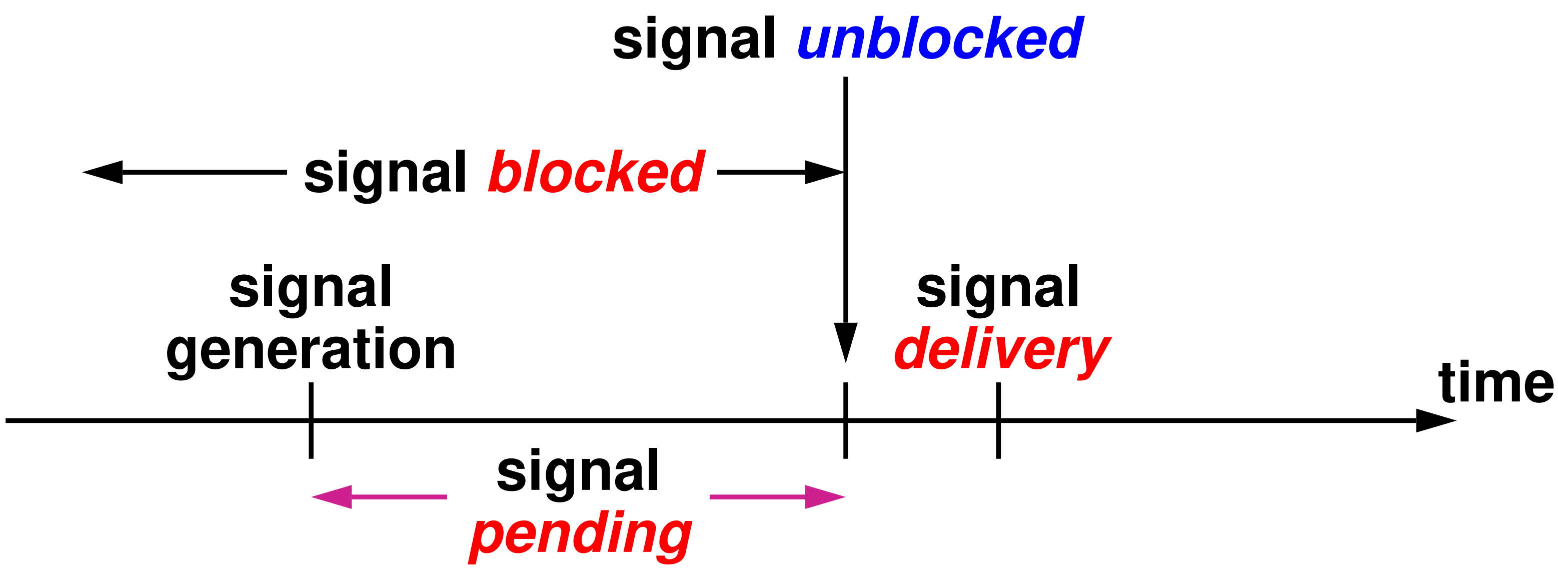
- ***Signals***
  - some would call a ***signal*** a ***software interrupt***
    - but it's really not
      - it's a "***callback mechanism***"
      - implemented in the OS by performing an ***upcall***
  - generated (by OS) in response to
    - exceptions (e.g., arithmetic errors, addressing problems)
    - external events (e.g., timer expiration, certain keystrokes, actions of other processes such as to terminate or pause the process)
    - user defined events
  - effect on process *(i.e., when the signal is "**delivered**"):*
    - termination (possibly after producing a core dump)
    - invocation of a procedure that has been set up to be a signal handler *(requires an **upcall**)*
    - suspension of execution
    - resumption of execution

Terminology



- Ex: <Ctrl+C>
- When a signal is generated, it is delivered as soon as possible *if* the signal is not "***blocked***"

Terminology



- Ex: <Ctrl+C>
- A signal is *pending* if it's generated but *blocked*
  - when the signal becomes unblocked, it will be *delievered*
- If you replaced the word "signal" with "interrupt" and "blocked/unblocked" with "disabled/enabled", everything would be correct for a hardware interrupt

Signal Types

Name	Description	Default Action
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`
  - (not always) terminate with extreme prejudice
- Also
  - type Ctrl-c (or <Ctrl+C>)
    - sends signal 2 (SIGINT) to current process
  - `kill` shell command
    - send SIGINT to process with pid=12345: "`kill -2 12345`"
  - do something illegal
    - bad address, bad arithmetic, etc.
- `int pthread_kill(pthread_t thr, int sig)`
  - send signal `sig` to thread `thr` (in the same process as the calling thread)
  - avoid using this and use *pthread cancellation* mechanism instead if you want to "kill a thread"