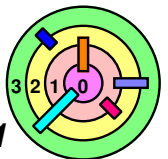
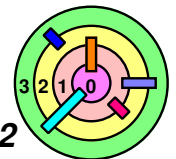


2.2.4 Thread Safety



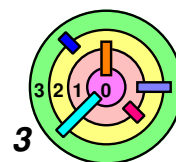
Thread Safety

- ➡ 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

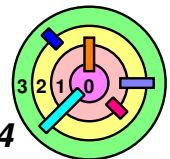
- ➡ Strictly speaking, making a function **thread-safe** is not the same as making it **reentrant**
- ➡ **thread-safe**: multiple threads can call the function in parallel or concurrently
 - ➡ **reentrant**: enter twice (even if you have only one thread)
 - how do you enter a function twice if you are not running multiple threads?
 - ◆ for a **kernel function**, you can get interrupted and call the same function inside an interrupt service routine
 - ◆ for a **user space function**, you can get interrupted and the kernel makes an **upcall** that calls the same function
 - ➡ 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

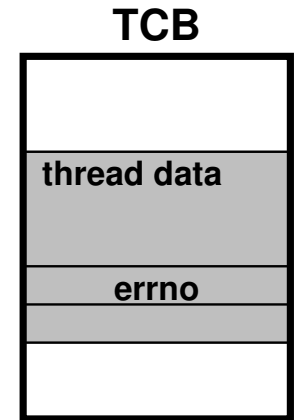
```
int IOfunc(int fd) {  
    extern int errno;  
    ...  
    if (write(fd, buffer, size) == -1) {  
        if (errno == EIO)  
            fprintf(stderr, "IO problems ... \n");  
        ...  
        return(0);  
    }  
    ...  
}
```

- ⇒ 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



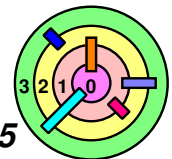
Coping

- ➡ 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

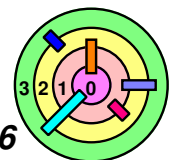
➡ `gethostbyname()` system call is not "thread safe"

```
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

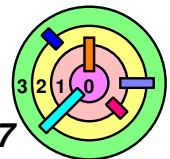
```
int gethostbyname_r(const char *name,  
                   struct hostent *ret,  
                   char *buf,  
                   size_t buflen,  
                   struct hostent **result,  
                   int *h_errnop)
```

- 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`

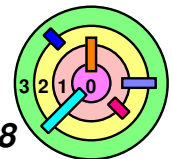


Coping

- ➡ 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;

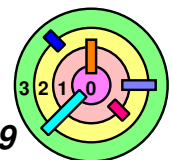
timeout.tv_sec = 3;           // seconds
timeout.tv_nsec = 1000;      // nanoseconds
nanosleep(&timeout, &remaining_time);
```

- 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



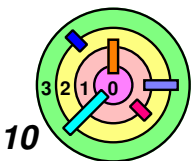
Timeouts

```

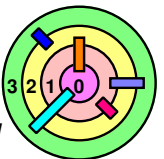
struct timespec relative_timeout, absolute_timeout;
struct timeval now;
relative_timeout.tv_sec = 3;           // seconds
relative_timeout.tv_nsec = 1000;      // nanoseconds
gettimeofday(&now, 0);
absolute_timeout.tv_sec = now.tv_sec +
    relative_timeout.tv_sec;
absolute_timeout.tv_nsec = 1000*now.tv_usec +
    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()`

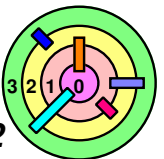


2.2.5 Deviations



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



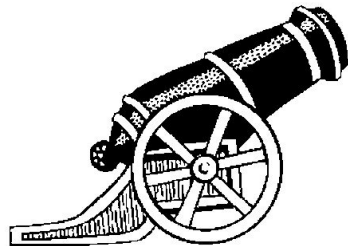
Signals

```
int x, y;
```

```
x = 0;
```

```
...
```

```
y = 16/x;
```

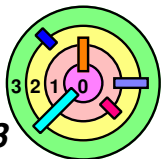


```
for (;;)
    keep_on_trying( );
```



- the original intent of Unix signals was to force the *graceful termination* of a process

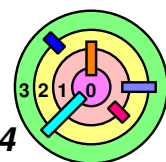
- e.g., <Ctrl+C>



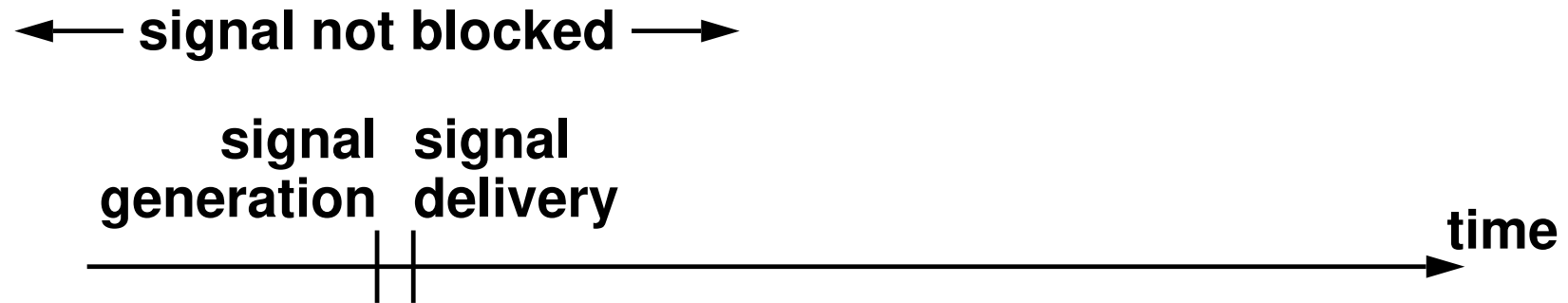
The OS to the Rescue

➡ *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*"): *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

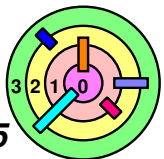


Terminology

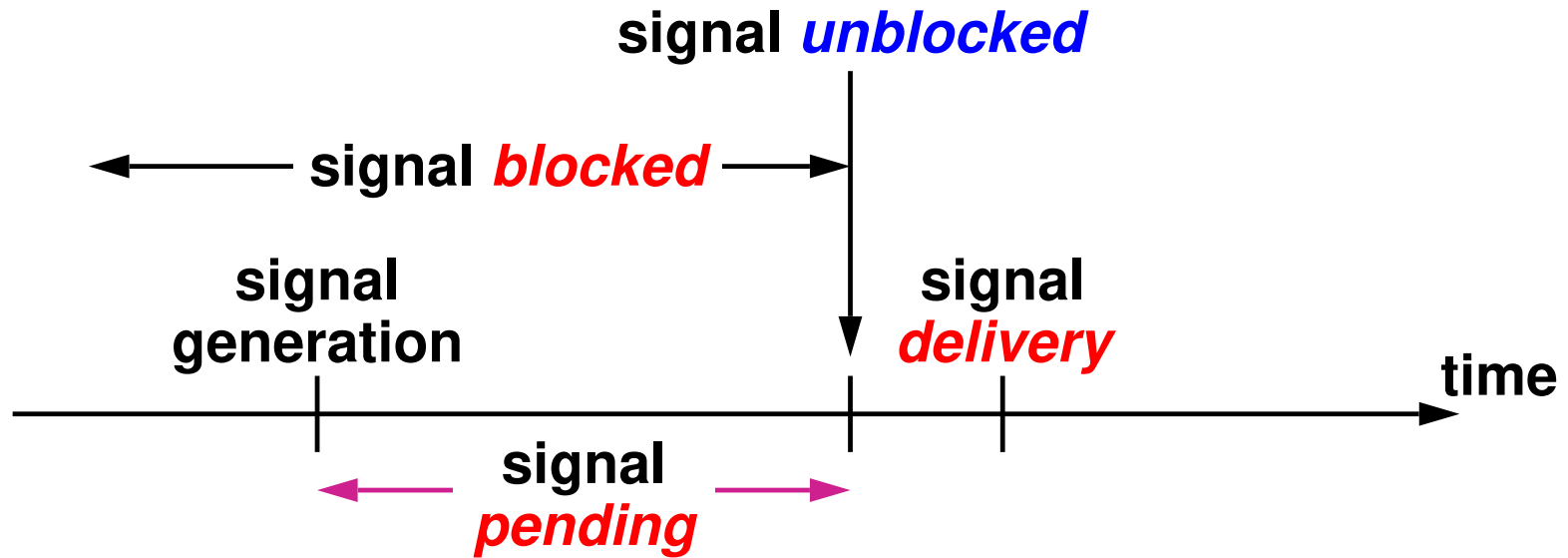


➡ Ex: <Ctrl+C>

➡ When a signal is generated, it is delivered as soon as possible *if* the signal is not "*blocked*"



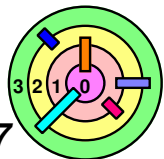
Terminology



- ➡ 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)`

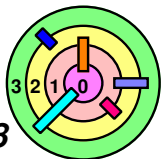
- send signal `sig` to process `pid`
- (not always) terminate with extreme prejudice

➡ 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.

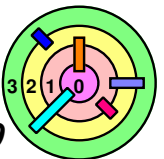
➡ `int pthread_kill(pthread_t thr, int sig)`

- 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"



Handling Signals

- ➡ Two ways to handle signals
- ▬ handling it *asynchronously*
 - using *signal handlers*
 - ▬ handling it *synchronously*
 - using `sigwait()` in a signal-catching thread



Handling Signals Asynchronously

- ➡ Signal handler
- each signal in a *process* can have *at most one handler*
 - to specify a signal handler of a process, use:

- `sigset/signal()`
 - ◆ returns the current handler (which could be the "default handler")
- `sigaction()`
 - ◆ more functionality

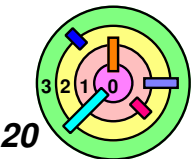
```
#include <signal.h>
```

```
typedef void (*sighandler_t) (int);
```

```
sighandler_t sigset(int signo, sighandler_t handler);  
sighandler_t signal(int signo, sighandler_t handler);
```

```
sighandler_t OldHandler = sigset(SIGINT, NewHandler);
```

- signal handler is part of the *context* of a process



Special Handlers



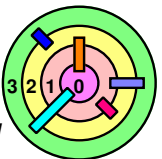
SIG_DFL

- use the default handler
- usually terminates the process
- `sigset/signal(SIGINT, SIG_DFL);`



SIG_IGN

- ignore the signal
- `sigset/signal(SIGINT, SIG_IGN);`



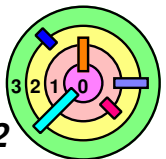
Example

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

```
int main() {  
    void handler(int);  
  
    sigset(SIGINT, handler);  
    while(1)  
        ;  
    return 1;  
}
```

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

- ⇒ *SIGINT is blocked* inside handler()
- ⇒ but how do you kill this program from your console?
 - can use the "kill" shell command, e.g., "kill -15 <pid>"
- ⇒ instead of using sigset(), you can also use sigaction()



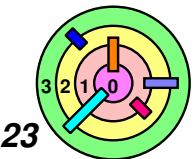
Example

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

```
int main() {  
    void handler(int);  
  
    sigset(SIGINT, handler);  
    while(1)  
        ;  
    return 1;  
}
```

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

← in some systems, you may have to re-establish the signal handler inside the signal handler if you want to receive the same signal more than once



sigaction

```
int sigaction(int sig,
              const struct sigaction *new,
              struct sigaction *old);

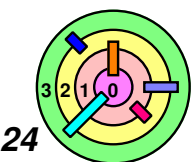
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
};
```



sigaction() allows
for more complex
behavior

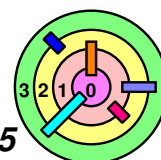
— e.g., block *additional*
signals (specified by
sa_mask) when
handler is called

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



Async-Signal Safety

- ➡ The problem with *asynchronous signal* is that you have to worry about *async-signal safety*
 - if you don't take care of it just right, bad things can happen
- ➡ *Async-Signal Safety*: Make your code safe when working with *asynchronous signals*
- ➡ The general rule to provide async-signal safety:
 - any data structure the signal handler accesses must be async-signal safe
 - i.e., an async signal must not corrupt data structures
- ➡ An alternative is to make async-signal synchronous
 - use a *signal-catching thread* to receive a particular signal

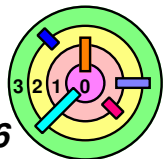


Example 1: Waiting for a Signal

```
sigset (SIGALRM, DoSomethingInteresting);  
...  
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 */
```

— can SIGALRM occur before pause() is called?

➡ Note: strickly speaking, this is *not* a deadlock
— it has a *race condition*



Example 2: Status Update

```
#include <signal.h>

computation_state_t state;

int main() {
    void handler(int);
    sigset(SIGINT, handler);
    long_running_proc();
    return 0;
}

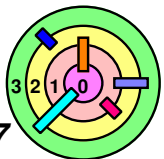
void long_running_proc() {
    while (a_long_time) {
        update_state(&state);
        compute_more();
    }
}

void handler(int signo) {
    display(&state);
}
```

- ⇒ long-running job that can take days to complete
 - the `handler()` can be used to print a progress report
 - need to make sure that `state` is in a consistent state
 - this is a synchronization issue
 - our `handler()` is not *async-signal safe*

➡ Note: this is *not* a deadlock and really not a race condition

- ⇒ this is the case where the code is not async signal safe



Example 2: Status Update

```
void long_running_proc() {  
    while (a_long_time) {  
        pthread_mutex_lock(&m);  
        update_state(&state);  
        pthread_mutex_unlock(&m);  
        compute_more();  
    }  
}
```

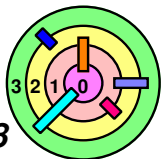
```
void handler(int signo) {  
    pthread_mutex_lock(&m);  
    display(&state);  
    pthread_mutex_unlock(&m);  
}
```



Does this work?

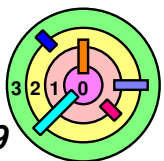
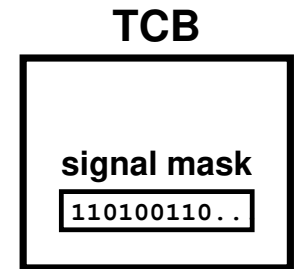
- no (*this code is **not reentrant***)
- it may get stuck in handler()
- signal handler gets executed till **completion**
 - in general, keep it simple and brief

yes, you can deadlock with yourself even if you only have one thread



Masking (Blocking) Signals

- ➡ Solution: control signal delivery by *masking/blocking the signal*
- don't mask/block all signals, just the ones you want
 - a set of signals is represented as a set of bits called `sigset_t`
 - which is just an unsigned `int`
 - if a mask bit is 1, the corresponding signal is *blocked*; otherwise, the corresponding signal is *unblocked*
 - when a child thread is created, it *inherits* signal mask from the parent thread



Masking (Blocking) Signals

➡ To examine or change the signal mask of the calling process

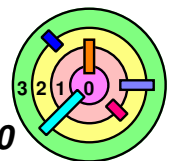
```
#include <signal.h>
int sigprocmask(
    int how,
    const sigset_t *set,
    sigset_t *old);
```

TCB

signal mask
110100110..

➡ 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)



sigset_t

- ➡ There are bunch of functions to manipulate `sigset_t`
- ➡ be careful, with *some APIs*, 1 means to *allowed/unblock* a signal, and with other APIs, 1 means to *blocked* a signal

TCB

signal mask
110100110...

- ➡ 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;
```

```
sigset_t set;
```

```
sigemptyset(&set);
```

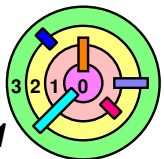
```
sigfillset(&set);
```

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

```
sigdelset(&set, SIGHUP);
```

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

```
sigdelset(&set, SIGINT);
```



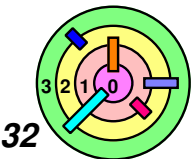
Example 1: Correct Way of Waiting for a Signal

```

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 */
sigfillset(&set);
sigdelset(&set, SIGALRM);
sigsuspend(&set); /* wait for it safely */
    /* SIGALRM masked again */
...
sigprocmask(SIG_SETMASK, &oldset, (sigset_t *)0);
    /* SIGALRM unmasked */

```

- = sigsuspend() *replaces* the caller's signal mask with the set of signals pointed to by the argument
 - in the above, all signals are blocked/masked except for SIGALRM
 - *atomically unblocks* the signal and *waits* for the signal



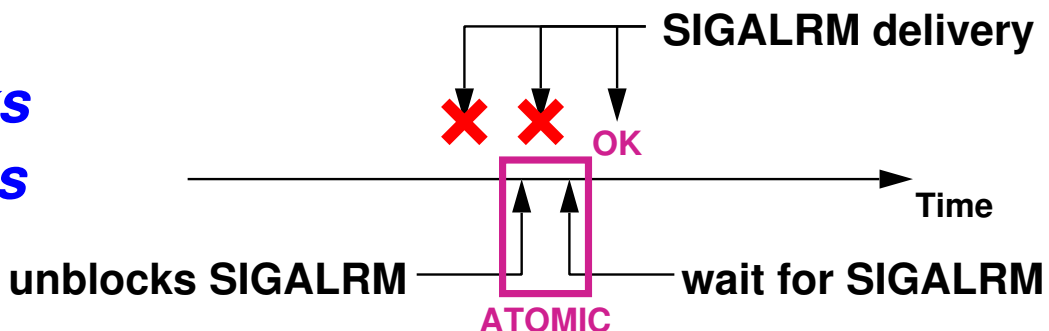
Example 1: Correct Way of Waiting for a Signal

```

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 */
sigfillset(&set);
sigdelset(&set, SIGALRM);
sigsuspend(&set); /* wait for it safely */
    /* SIGALRM masked again */
...
sigprocmask(SIG_SETMASK, &oldset, (sigset_t *)0);
    /* SIGALRM unmasked */

```

- ▣ sigsuspend()
 - *atomically unblocks* the signal and *waits* for the signal



Async-Signal Safety

- ➡ There is only **one** correct way to wait for an asynchronous event **caused by your thread**
- ➡ e.g., an alarm that **you** wind up and wait for is an asynchronous event you generate

Step 1) **block** the asynchronous event

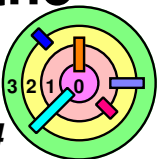
Step 2) do something that will cause the asynchronous event to get generated

Step 3) **unblock** the event and **wait** for the event in **one atomic operation**

- ➡ There is only **one** correct way to wait for an asynchronous event **caused by another thread**
- ➡ e.g., wait for a guard to become true in a guarded command

Step 1) **block** the asynchronous event

Step 2) check if the event has been generated, if not, **unblock** the event and **wait** for the event in **one atomic operation**



Example 2: Correct Way to Handle Status Update

```
#include <signal.h>

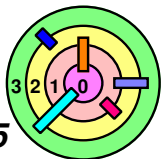
computation_state_t state;
sigset_t set;

int main() {
    void handler(int);
    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    sigset(SIGINT, handler);
    long_running_proc();
    return 0;
}
```

— now SIGINT cannot be delivered in
update_state()

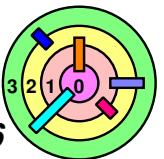
```
void long_running_proc() {
    while (a_long_time) {
        sigset_t old_set;
        sigprocmask(
            SIG_BLOCK,
            &set,
            &old_set);
        update_state(&state);
        sigprocmask(
            SIG_SETMASK,
            &old_set,
            0);
        compute_more();
    }
}

void handler(int signo) {
    display(&state);
}
```



Signals and Threads

- ➡ In Unix, signals are sent to *processes*, not threads!
 - in a single-threaded process, it's obvious which thread would handle the signal
 - in a multi-threaded process, it's not so clear
 - in POSIX, the signal is delivered to a thread chosen *at random*
- ➡ What about the signal mask (i.e., blocked/enabled signals)?
 - should one set of sigmask affect all threads in a process?
 - or should each thread gets it own sigmask?
 - this certainly makes more sense
- ➡ POSIX rules for a multithreaded process:
 - the thread that is to receive the signal is chosen *randomly* from the set of threads that do not have the signal blocked
 - if all threads have the signal blocked, then the signal remains pending until some thread unblocks it
 - ◇ at which point the signal is delivered to that thread
 - child thread *inherits* signal mask from parent thread



Synchronizing Asynchrony

```

some_state_t state;
sigset_t set;

main() {
    pthread_t thread;
    sigemptyset(&set);
    sigaddset(&set,
              SIGINT);
    sigprocmask(
        SIG_BLOCK,
        &set, 0);
    // main thread
    //      blocks SIGINT
    pthread_create(
        &thread, 0,
        monitor, 0);
    long_running_proc();
}

```

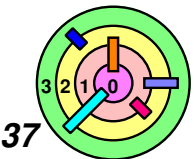
```

void long_running_proc() {
    while (a_long_time) {
        pthread_mutex_lock(&m);
        update_state(&state);
        pthread_mutex_unlock(&m);
        compute_more();
    }
}

void *monitor() {
    int sig;
    while (1) {
        sigwait(&set, &sig);
        pthread_mutex_lock(&m);
        display(&state);
        pthread_mutex_unlock(&m);
    }
    return(0);
}

```

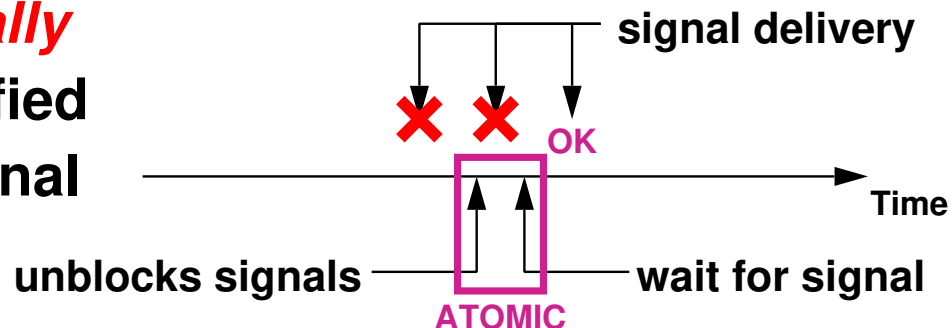
— this is **PREFERRED**, no need for signal handler!



sigwait

```
int sigwait(sigset_t *set, int *sig)
```

- ➡ `sigwait()` blocks until a signal specified in `set` is received
 - return which signal caused it to return in `sig`
 - if you have a signal handler specified for `sig`, it will *not* get invoked when the signal is delivered
 - instead, `sigwait()` will return
- ➡ You should make sure that all the threads in your process have these signals blocked!
 - this way, when `sigwait()` is called, the calling thread temporarily becomes the *only* thread in the process who can receive the signal
- ➡ `sigwait(set)` *atomically unblocks* signals specified in `set` and *waits* for signal delivery



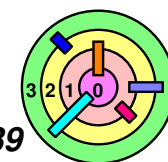
Signals and Blocking System Calls

➡ What if a signal is generated while a process is blocked in a system call?

- 1) deal with it when the system call completes
- 2) interrupt the system call, deal with signal, resume system call
- 3) interrupt system call, deal with signal, return from system call with indication that something happened

— most systems choose (3)

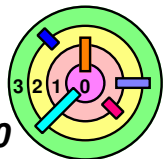
- `errno` sets to `EINTR` to mean that the system call was not completed because it was interrupted by a signal
 - ◆ this is the `errno` for the thread that was "deviated" to execute the signal handler
- this may be the reason why `pthread_cond_wait()` may return "spontaneously" even when the CV has not been signaled/broadcasted
 - ◆ what if this thread was borrowed to deliver a signal?



Interrupted System Calls

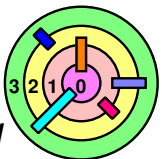
```
while(read(fd, buffer, buf_size) == -1) {  
    if (errno == EINTR) {  
        /* interrupted system call; try again */  
        continue;  
    }  
    /* the error is more serious */  
    perror("big trouble");  
    exit(1);  
}
```

- need to check the return value of `read()` because `read()` can return when less than `buf_size` bytes have been read
- can use similar code for `write()`
 - same consideration as `read()`
- please note that the above code is incomplete, it needs to handle the case where `read()` return 0 to mean end-of-input



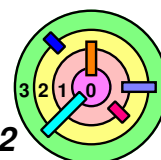
Interrupted While Underway

```
remaining = total_count; /* write this many bytes */
bptr = buf;               /* starting from here */
for ( ; ; ) {
    num_xfrd = write(fd, bptr, remaining);
    if (num_xfrd == -1) {
        if (errno == EINTR) {
            /* interrupted early */
            continue;
        }
        perror("big trouble");
        exit(1);
    }
    if (num_xfrd < remaining) {
        /* interrupted in the middle of write() */
        remaining -= num_xfrd;
        bptr += num_xfrd;
        continue;
    }
    /* success! */
    break;
}
```



Interrupted System Calls

- ➡ If a thread can "see" signal delivery (i.e., "deviated to execute a signal handler"), every `read()` and `write()` call in that thread needs to look like the previous slides
- ⇒ much easier if you use a *signal-catching thread*
 - and *block all appropriate signals* in all "regular" threads
 - for warmup2, it is strongly encouraged that you do it this way to catch `<Ctrl+C>` and avoid using a signal handler
 - ◆ when `sigwait()` returns, lock mutex, set global flag, cancel packet arrival and token depositing threads, broadcast CV, unlock mutex, and self-terminate
 - ◆ according to spec, you must not cancel server threads
 - ◆ will talk about cancellation shortly

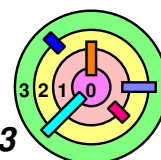


Inside A Signal Handler

➡ Which library routines are safe to use *within* signal handlers?

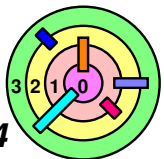
access	dup2	getgroups	rename	sigprocmask	time
aio_error	dup	getpgrp	rmdir	sigqueue	timer_getoverrun
aio_suspend	execle	getpid	sem_post	sigsuspend	timer_gettime
alarm	execve	getppid	setgid	sleep	timer_settime
cfgetispeed	_exit	getuid	setpgid	stat	times
cfgetospeed	fcntl	kill	setsid	sysconf	umask
cfsetispeed	fdatasync	link	setuid	tcdrain	uname
cfsetospeed	fork	lseek	sigaction	tcflow	unlink
chdir	fstat	mkdir	sigaddset	tcflush	utime
chmod	fsync	mkfifo	sigdelset	tcgetattr	wait
chown	getegid	open	sigemptyset	tcgetpgrp	waitpid
clock_gettime	geteuid	pathconf	sigfillset	tcsendbreak	write
close	getgid	pause	sigismember	tcsetattr	
creat	getoverrun	pipe	sigpending	tcsetpgrp	

➡ Note: in general, you should only do what's absolutely necessary inside a signal handler (and figure out where to do the rest)



Cancellation

- ➡ The user pressed <Ctrl+C>
 - or a request is generated to terminate the process
 - the chores being performed by the remaining threads are no longer needed
 - in general, we may just want to cancel a bunch of threads and not the entire process
- ➡ Concerns
 - getting cancelled at an inopportune moment
 - a mutex left locked
 - a data structure is left in an inconsistent state
 - ◆ e.g., you get a cancellation request when you are in the middle of a `insert()` operation into a doubly-linked list and `insert()` is protected by a mutex
 - cleaning up (free up resources that only this thread can free up)
 - memory leaks
 - unlocking mutex if locked



Cancellation State & Type

- ➡ Send cancellation request to a thread (*this is a **non-blocking** call*)

```
pthread_cancel(thread)
```

- ➡ Cancels enabled or disabled

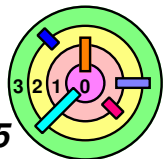
```
int pthread_setcancelstate(  
    { PTHREAD_CANCEL_DISABLE,  
      PTHREAD_CANCEL_ENABLE },  
    &oldstate)
```

- ➡ Asynchronous vs. deferred cancels

```
int pthread_setcanceltype(  
    { PTHREAD_CANCEL_ASYNCHRONOUS,  
      PTHREAD_CANCEL_DEFERRED },  
    &oldtype)
```

- ➡ By default, a thread has cancellation **enabled** and **deferred**

- it's for a good reason
- if you are going to change it, you must ask yourself, "Why?" and "Are you sure this is really a good idea?"

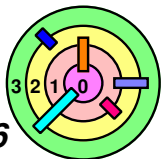


POSIX Cancellation Rules



POSIX threads cancellation rules (part 1):

- when `pthread_cancel()` gets called, the target thread is marked as having a **pending cancel**
 - the thread that called `pthread_cancel()` does not wait for the cancel to take effect
- if the target thread has cancellation **disabled**, the target thread stays in the pending cancel state
- if the target thread has cancellation **enabled** ...
 - if the cancellation type is **asynchronous**, the target thread immediately **acts on cancel** (i.e., cancellation is "delivered" by "deviating" the thread to call `pthread_exit()`)
 - if the cancellation type is **deferred**, cancellation is **delayed** until it reaches a **cancellation point** in its execution
 - ◆ cancellation points correspond to points in the thread's execution at which it is safe to **act on cancel**

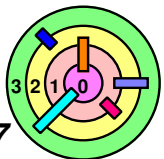


Cancellation Points

`aio_suspend`
`close`
`creat`
`fcntl` (when `F_SETLCKW`
is the command)
`fsync`
`mq_receive`
`mq_send`
`msync`
`nanosleep`
`open`
`pause`
`pthread_cond_wait`
`pthread_cond_timedwait`

`pthread_join`
`pthread_testcancel`
`read`
`sem_wait`
`sigsuspend`
`sigtimedwait`
`sigwait`
`sigwaitinfo`
`sleep`
`system`
`tcdrain`
`wait`
`waitpid`
`write`

- ▮ `pthread_mutex_lock()` is not on the list!
- ▮ `pthread_testcancel()` creates a cancellation point
 - useful if a thread contains no other cancellation point



POSIX Cancellation Rules

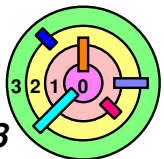


POSIX threads cancellation rules (part 2):

— when a thread *acts on cancel*

- it calls `pthread_exit()`
- in `pthread_exit()`, it first walks through a *stack of cleanup handlers*
 - ◆ when stack is empty, the thread goes into the zombie state
- remember that the thread that called `pthread_cancel()` does not wait for the cancel to take effect
 - ◆ it may join and wait for the target thread to terminate

```
pthread_cleanup_push(  
    (void) (*routine) (void *),  
    void *arg)  
pthread_cleanup_pop(int execute)
```



Example

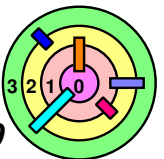
```
list_item_t list_head;

void *GatherData(void *arg) {
    list_item_t *item;
    item = (list_item_t*)malloc(sizeof(list_item_t));

    // GetDataItem() contains many cancellation points
    GetDataItem(&item->value);

    insert(item); // add item to a global list
    printf("Done.\n");
    return 0;
}
```

- ➡ How can this thread control when it acts on cancel?
- so it doesn't leak memory



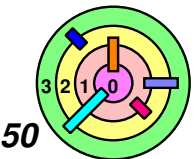
Example

```
list_item_t list_head;

void *GatherData(void *arg) {
    list_item_t *item;
    item = (list_item_t*)malloc(sizeof(list_item_t));
    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, 0);
    // GetDataItem() contains many cancellation points
    GetDataItem(&item->value);
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, 0);
    insert(item); // add item to a global list
    printf("Done.\n");
    return 0;
}
```

➡ How can this thread control when it acts on cancel?

- so it doesn't leak memory
- although this implementation is technically "correct", long delay may not be acceptable
 - need to *respond in a timely manner*



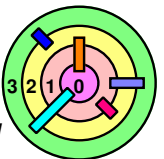
Example

```
list_item_t list_head;

void *GatherData(void *arg) {
    list_item_t *item;
    item = (list_item_t*)malloc(sizeof(list_item_t));
    pthread_cleanup_push(free, item);
    // GetDataItem() contains many cancellation points
    GetDataItem(&item->value);

    insert(item); // add item to a global list
    printf("Done.\n");
    return 0;
}
```

- ➡ Can act on cancel inside GetDataItem()
- ▬ in this case, will invoke free(item)
 - ▬ in C library, free() is defined as: `void free(void *ptr);`
 - perfectly matches the argument types for `pthread_cleanup_push()`



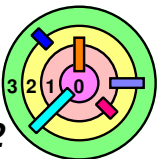
Example

```
list_item_t list_head;

void *GatherData(void *arg) {
    list_item_t *item;
    item = (list_item_t*)malloc(sizeof(list_item_t));
    pthread_cleanup_push(free, item);
    // GetDataItem() contains many cancellation points
    GetDataItem(&item->value);

    insert(item); // add item to a global list
    printf("Done.\n");
    return 0;
}
```

- ➡ What if it acts on cancel inside printf()
- will end up calling free(item) twice
 - can cause segmentation fault later



Example

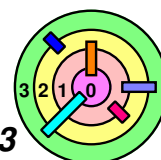
```
list_item_t list_head;

void *GatherData(void *arg) {
    list_item_t *item;
    item = (list_item_t*)malloc(sizeof(list_item_t));
    pthread_cleanup_push(free, item);
    // GetDataItem() contains many cancellation points
    GetDataItem(&item->value);
    pthread_cleanup_pop(0);
    insert(item); // add item to a global list
    printf("Done.\n");
    return 0;
}
```



What if it acts on cancel inside printf()

- will end up calling free(item) twice
 - can cause segmentation fault later
- pop free(item) off the cleanup stack



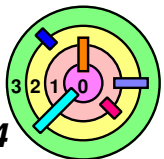
Example

```
list_item_t list_head;
```

```
void *GatherData(void *arg) {
    list_item_t *item;
    item = (list_item_t*)malloc(sizeof(list_item_t));
    pthread_cleanup_push(free, item); // {
    // GetDataItem() contains many cancellation points
    GetDataItem(&item->value);
    pthread_cleanup_pop(0); // }
    insert(item); // add item to a global list
    printf("Done.\n");
    return 0;
}
```

must match up (like a pair of brackets)

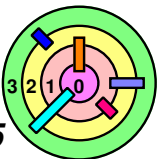
- ➡ pthread_cleanup_push() and the corresponding pthread_cleanup_pop() must match up (like a pair of brackets)
 - must not call pthread_cleanup_push() in one function and call the corresponding pthread_cleanup_pop() in another
 - compile-time error



Cancellation and Cleanup

```
void close_file(int fd) {  
    close(fd);  
}  
  
fd = open(file, O_RDONLY);  
pthread_cleanup_push(close_file, fd);  
while(1) {  
    read(fd, buffer, buf_size);  
    // ...  
}  
pthread_cleanup_pop(0);
```

- ▢ should close any opened files when you clean up
- ▢ int is compatible with void*
 - well, sort of
 - void* can be a 64-bit quantity, so may need to be careful (best to be explicit)

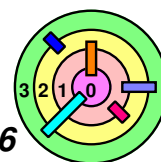


Cancellation and Conditions

```
pthread_mutex_lock(&m);
pthread_cleanup_push(CleanupHandler, argument);

while(should_wait)
    pthread_cond_wait(&cv, &m);
// ... (code containing other cancellation points)
pthread_cleanup_pop(0);
pthread_mutex_unlock(&m);
```

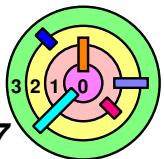
- = should CleanupHandler() call pthread_mutex_unlock() ?
 - remember, if the thread is canceled between push() and pop(), we need to ensure that the mutex is *locked*
 - pthread_cond_wait() is a cancellation point
 - ◆ must not unlock the mutex twice!
- = should CleanupHandler() call pthread_mutex_lock() then call pthread_mutex_unlock() ?
 - what if the mutex is locked?
- = application cannot solve this problem since there is *no way to check if a mutex is locked or not*



Cancellation and Conditions

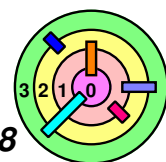
```
pthread_mutex_lock(&m);  
pthread_cleanup_push(pthread_mutex_unlock, &m);  
  
while(should_wait)  
    pthread_cond_wait(&cv, &m);  
// ... (code containing other cancellation points)  
pthread_cleanup_pop(1);
```

- ▮ pthreads library implementation ensures that a thread, when acting on a cancel inside `pthread_cond_wait()`, would first lock the mutex, before calling the cleanup routines
 - this way, the above code would work correctly



Warmup2 Cancellation

- ➡ Only packet arrival and token depositing threads are allowed to be canceled
 - use a <Ctrl+C>-catching thread (i.e., use `sigwait()`)
 - make cancellation requests only when mutex is *locked*
 - this makes it impossible for those threads to act on cancel when they have the mutex locked
 - can make the following simplification for these two threads:
 - at the start of their first procedures, *disable cancellation*
 - right before calling `usleep()`, *enable cancellation*
 - when `usleep()` returns, *disable cancellation again*
 - this way, during the time the mutex is locked, cancellation is always disabled
 - ◆ therefore, don't have to worry about using cleanup routines to unlock mutex
 - but didn't we just say that it's not a good idea to disable cancellation?
 - also, need to take care of a *race condition*



Cancellation & C++

```
void tcode() {  
    A a1;  
    pthread_cleanup_push(handler, 0);  
    foo();  
    pthread_cleanup_pop(0);  
}
```

```
void foo() {  
    A a2;  
    pthread_testcancel();  
}
```

- are the destructors of a1 and a2 getting called?
 - not sure
 - they should get called
 - some C++ implementation does not do this correctly!

➡ Note: current C++ standard also does *not* support thread cancellation

- standard C++ threads must self-terminate!

