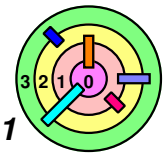
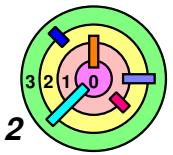
Housekeeping (Lecture 6 - 9/16/2013)

- Warmup #2 due at 11:45pm on Friday, 10/4/2013
  - must implement from scratch and code everything yourself
  - strongly recommend that you work with your potential partners
    - work/discuss at a high level
    - do *not* write a single line of code together
  - start early!
  - if you have code from a previous semester, be very careful and *not copy any code from it*
    - it's best if you just get rid of it
- Please do not set your class Google Group e-mail delivery preference to "No email"
  - if you do that, I will change it to "All email"
  - because all important announcements will be sent to the class Google Group

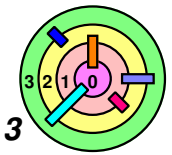


Housekeeping (Lecture 6 - 9/16/2013)

- You need to learn Unix!
  - our kernel assignments are to implement a Unix system!
- Have you installed *Ubuntu 11.10* on your laptop/desktop?
  - you are required to do your kernel assignments on Ubuntu 11.10
  - if there are any problems, I need to know now so we can get it resolved *NOW!*

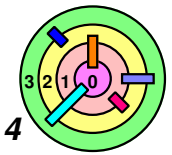


2.2.4 Thread Safety



Thread Safety

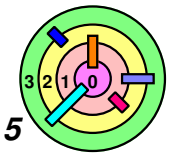
- Unix was developed way before threads were commonly used
  - Unix libraries were built without threads in mind
  - running code using these libraries with threads became unsafe
  - to make these libraries safe to run under multithreading is known as *Thread Safety*
- General problems with the old Unix API
  - global variables
    - e.g., `errno`
  - shared data
    - e.g., `printf()`



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 reentrant*
- `errno` is a system-call level *global variable*
  - Unix system-call library was implemented before multi-threading was a common practice

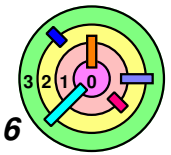


Coping

- **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
      - see textbook on exactly how this is done



Add "Reentrant" Version Of System Call

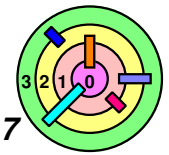
- `gethostbyname()` system call is not reentrant

```
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:**

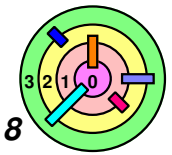
```
printf("goto statement reached");
```

- **Thread 2:**

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

- **Printed on display:**

```
goto Hello Wostatement reachedrld
```

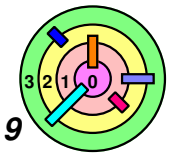


Coping

- 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



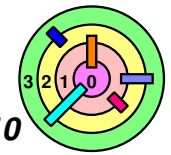
Killing Time ...

```
struct timespec timeout, remaining_time;

timeout.tv_sec = 3;       // seconds
timeout.tv_nsec = 1000; // nanoseconds

nanosleep(&timeout, &remaining_time);
```

- what if you don't want to wait for an "event" any more, after you have spent a certain amount of time waiting for it?

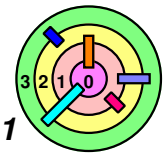
```
int pthread_cond_timedwait(
                pthread_cond_t *cond,
                pthread_mutex_t *mutex,
                struct timespec *abstime)
```

  - you need to calculate `abstime` carefully

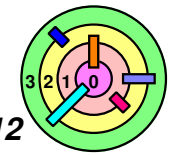


Timeouts

```
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);
```

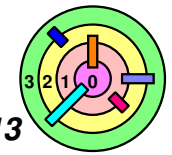


2.2.5 Deviations

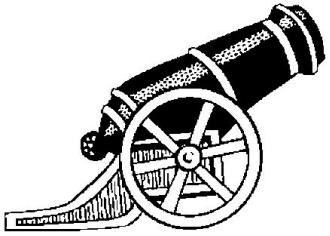


Deviations

- 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



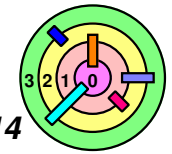
Signals



```
int x, y;

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

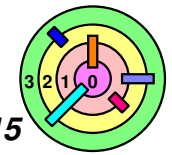
```
for (;;)
  keep_on_trying( );
```

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

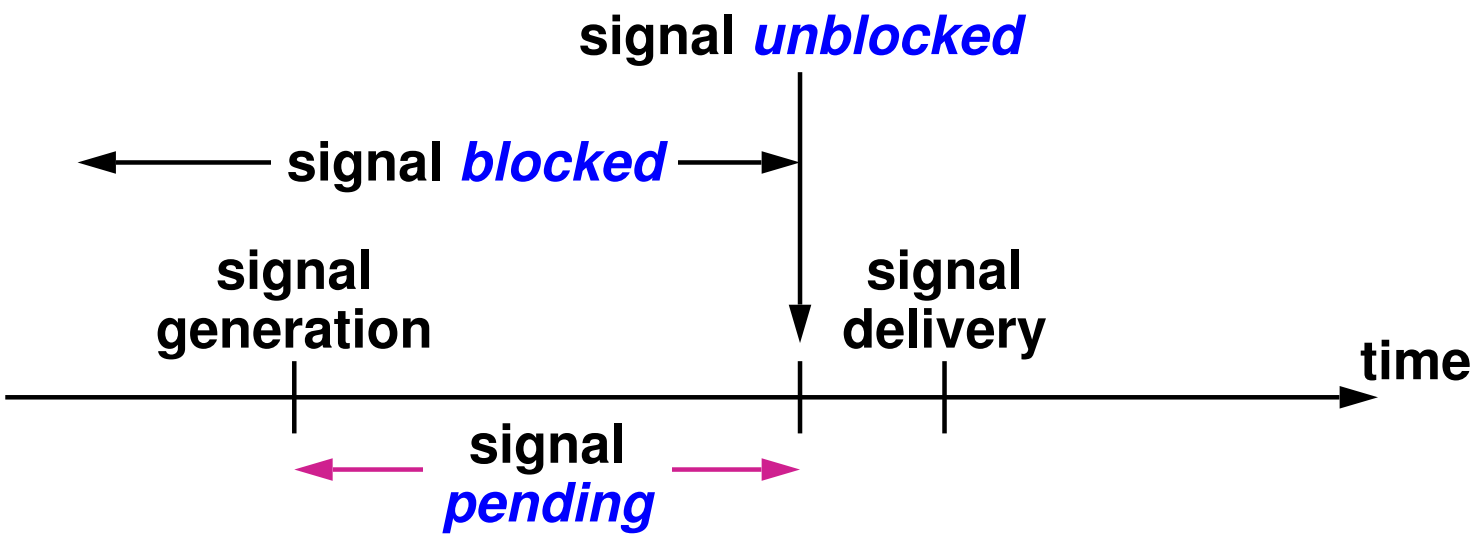


The OS to the Rescue

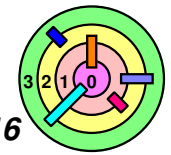
- ***Signals***
  - a signal is a *software interrupt*
  - generated (by OS) in response to
    - exceptions (e.g., arithmetic errors, addressing problems)
    - external events (e.g., timer expiration, certain keystrokes, actions of other processes such as to terminate or pause the process)
    - user defined events
  - effect on process:
    - termination (possibly after 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

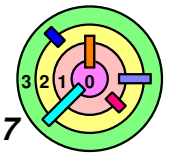


- A signal is *pending* if it's generated but *blocked*
  - when the signal becomes unblocked, it can be delievered
- Ex: <Cntrl+C>



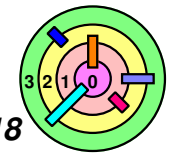
Signal Types

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



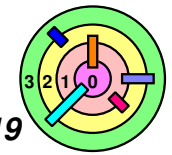
Sending a Signal

- `int kill(pid_t pid, int sig)`
  - send signal `sig` to process `pid`
  - (not always) terminate with extreme prejudice
- Also
  - type Ctrl-c (or <Cntrl+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`



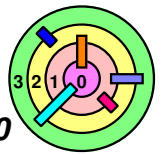
Handling Signals

```
#include <signal.h>

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

sighandler_t OldHandler;

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



Special Handlers

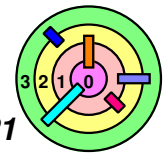
- SIG_IGN
  - ignore the signal
  - `signal(SIGINT, SIG_IGN);`
- SIG_DFL
  - use the default handler
  - usually terminates the process
  - `sigset(SIGINT, SIG_DFL);`



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);
}
```

- 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()`

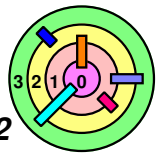


sigaction

```
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 in handler

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



Signal Mask

- ***Signal Mask:*** **a set of signals is represented as a set of bits (sigset)**
  - **if a mask bit is 1, the corresponding signal is *blocked***
  - **although sometimes, bits that are set correspond to *allowed* signals**

- **To clear a set:**

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

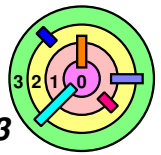
- **To add or remove a signal from the set:**

```
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);
```

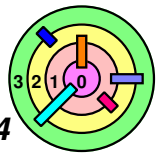
```
sigset_t set;

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



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?

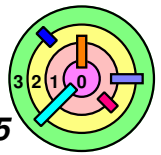


Example 2: Status Update

```
#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);
}
```

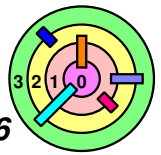
- long-running job that can take days to complete
  - the `handler()` can be used to print a progress report
  - need to make sure that `state` is in a consistent state
  - this is a synchronization issue
  - our `handler()` is not *async-signal safe*



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
  - it may hang in `handler()` and cause *deadlock*
  - signal handler usually gets executed till *completion*
    - in general, keep it simple and brief

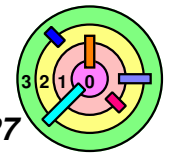


Masking (Blocking) Signals

- Solution: ***mask/block the signal***
  - don't mask/block all signals, just the one you want

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

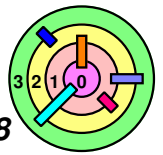
- used to examine or change the signal mask of the calling process
- `how` is one of three commands:
  - SIG_BLOCK: the new signal mask is the union of the current signal mask and `set`
  - SIG_UNBLOCK: the new signal mask is the intersection of the current signal mask and the complement of `set`
  -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



Example 1: 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

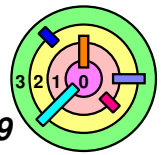


Example 2: Status Update

```
#include <signal.h>

computation_state_t state;
sigset_t set;

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

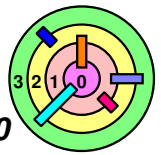
- now SIGINT cannot be delievered in `update_state()`

```
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);
}
```



Async-Signal Safety

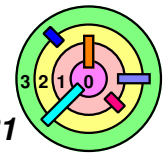
- ***Async-Signal Safety:*** **Make your code safe when working with asynchronous signals**
- **The general rule to provide async-signal safety:**
  - **any data structure the signal handler accesses must be async-signal safe**
    - **i.e., an async signal cannot corrupt data structures**
- **An alternative is to make async-signal synchronous**
  - **use another thread to receive a particular signal**



Signals and Blocking System Calls

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

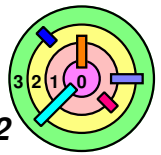
1) deal with it when the system call completes
2) interrupt the system call, deal with signal, resume system call
3) interrupt system call, deal with signal, return from system call with indication that something happened



Interrupted System Calls

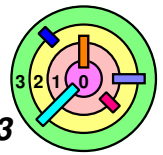
```
while(read(fd, buffer, buf_size) == -1) {
  if (errno == EINTR) {
    /* interrupted system call; try again */
    continue;
  }
  /* the error is more serious */
  perror("big trouble");
  exit(1);
}
```

- 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 writing
  - same consideration as `read()`



Interrupted While Underway

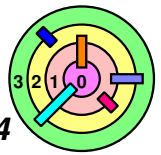
```
remaining = total_count; /* write this many bytes */
bptr = buf;              /* starting from here */
for ( ; ; ) {
  num_xfrd = write(fd, bptr, remaining);
  if (num_xfrd == -1) {
    if (errno == EINTR) {
      /* interrupted early */
      continue;
    }
    perror("big trouble");
    exit(1);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



Inside A Signal Handler

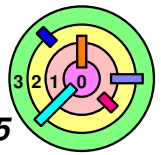
Which library routines are safe to use *within* signal handlers?

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	



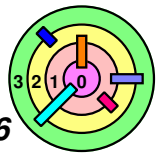
Signals and Threads

- 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 threads, 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
- Moreover, `sigprocmask()` is implemented as a system call
  - somewhat expensive to use



Signals and Threads

- **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**

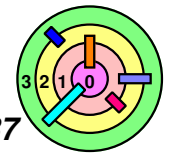


Synchronizing Asynchrony

```
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);
}
```

- 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`
- 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
    - even if the calling thread has a handler for the signal, the handler will not be invoked