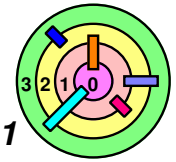
Handling Signals

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

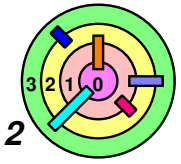


Handling Signals Asynchronously

- 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

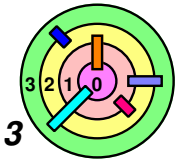
```
#include <signal.h>

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

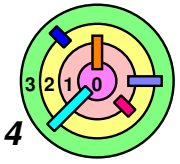
- SIG_DFL
  - use the default handler
  - usually terminates the process
  - `sigset/signal(SIGINT, SIG_DFL);`
- SIG_IGN
  - ignore the signal
  - `sigset/signal(SIGINT, SIG_IGN);`



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);
}
```

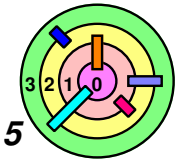
- *SIGINT is blocked* inside `handler()`
- but how do you kill this program from your console?
  - can use the "`kill`" shell command, e.g., "`kill -15 <pid>`"
- instead of using `sigset()`, you can also use `sigaction()`



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

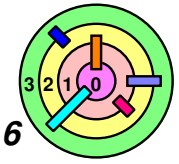


sigaction

```
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;
};
```

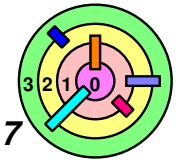
- `sigaction()` allows for more complex behavior
  - e.g., block *additional* signals (specified by `sa_mask`) when handler is called

```
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



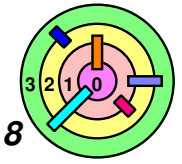
Example 1: Waiting for a Signal

```
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

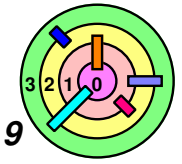
```
#include <signal.h>

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

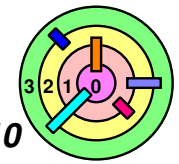


Example 2: Status Update

```
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);
}
```

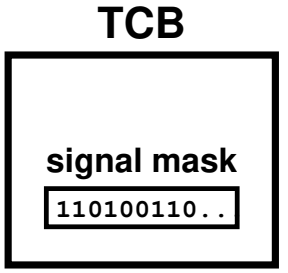
- Does this work?
  - no *(this code is not reentrant)*
  - it may get stuck in `handler()`
  - signal handler gets executed till *completion*
    - in general, keep it simple and brief

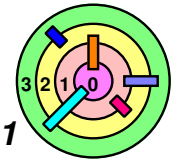
- yes, you can deadlock with yourself even if you only have one thread



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

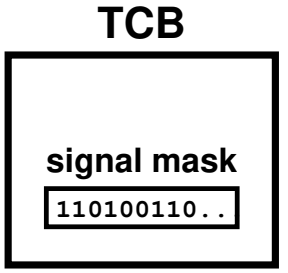




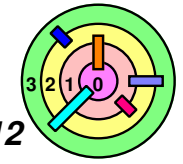
Masking (Blocking) Signals

- 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);
```

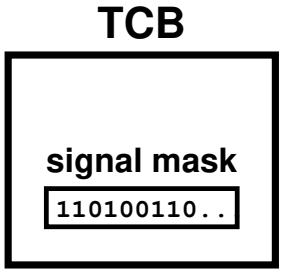


- `how` is one of three commands:
  - SIG_BLOCK: the new signal mask is the union of the current signal mask and `(*set)`
  - SIG_UNBLOCK: the new signal mask is the intersection of the current signal mask and the complement of `(*set)`
  - SIG_SETMASK: the new signal mask is `(*set)`



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



- To clear a set:

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

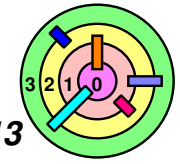
- To add or remove a signal from the set:

```
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);
```

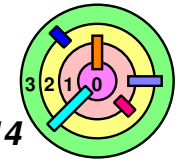
```
sigset_t set;

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



Example 1: 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 */
```

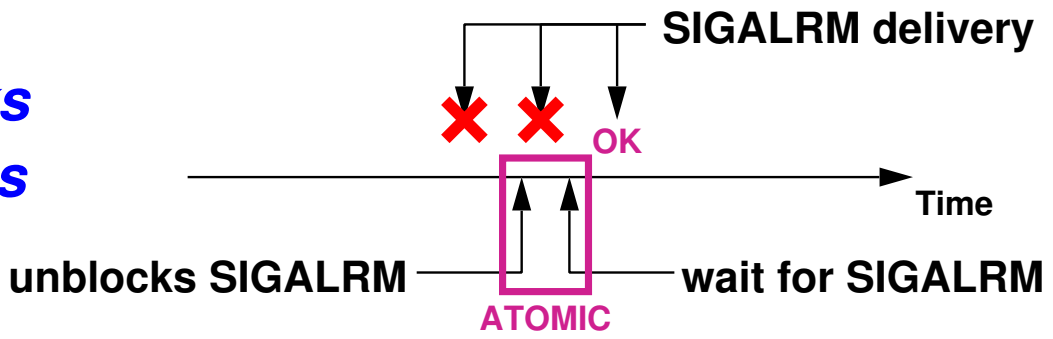
- `sigsuspend()` ***replaces*** the caller's signal mask with the set of signals pointed to by the argument
  - in the above, all signals are blocked/masked except for SIGALRM
  - ***atomically unblocks*** the signal and ***waits*** for the signal



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



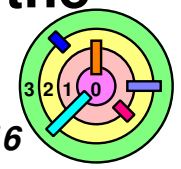
Async-Signal Safety

- There is only *one* correct way to wait for an asynchronous event *you generated*
  - 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 *someone else generated*
  - 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*

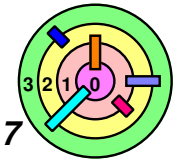


Example 2: Correct Way to Handle Status Update

```
#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;
}
```

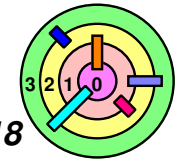
```
void long_running_proc() {
  while (a_long_time) {
    sigset_t old_set;
    sigprocmask(
        SIG_BLOCK,
        &set,
        &old_set);
    update_state(&state);
    sigprocmask(
        SIG_SETMASK,
        &old_set,
        0);
    compute_more();
  }
}

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

- now SIGINT cannot be delievered in `update_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

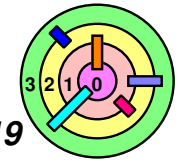


Synchronizing Asynchrony

```
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