Threads 1: Introduction

1. **Introduction**
   - **threads vs. processes**
   - **what is shared and what is not**
   - **POSIX threads (Pthreads)**
   - **the main thread**
   - **Pthreads and attributes**
   - **Pthreads and I/O**
   - **pread() and pwrite()**
   - **Pthreads and signals**
2. Creation and Termination
3. Synchronization (part a)
4. Synchronization (part b)
5. Miscellaneous

Threads vs. Processes

A traditional program has a *single* **thread of execution** active within a *single* **process**.

Most modern operating systems also support **multithreading**: the ability to have *multiple concurrent threads of execution* running within the address space of a *single process*.

I.e., the ability to be concurrently executing different sequences of code from within a *single program*.

The OS mechanism that supports multiple threads of execution is also typically called a **thread**.

An **OS thread** encapsulate one of possibly multiple *threads of execution* running within the context of a *single process*.

Threads vs. Processes (contd.)

Recall from the **Concurrent Computing lectures** that there are two alternative approaches for building **concurrent programs**:

- **multi-process programs** – composed of multiple *processes*, each with completely *separate address spaces* (memory)
- **multithreaded programs** – composed of multiple *OS threads*, sharing the address space of a *single process*

An OS thread is sometimes described as a **lightweight process** because the multiple threads within a process all share the same *address space* (process memory).

As a result, creating a new OS thread requires much less additional memory and less time to create than a new process.

Threads vs. Processes (contd.)

Testing by M. Kerrisk for his 2010 Linux book shows that creating a new *process* (with `fork()`) takes on the order of *ten times as long* as creating a new OS thread (with `clone()`).

However, the time to create a fairly large process is less than 10^{-3} sec., so the difference between creating a few subprocesses vs. a few OS threads is rarely going to be noticeable.

Likewise, while multiple threads require less *address space* than multiple processes, the use of **paged virtual memory** means this may have little practical effect.

Except for extreme cases, one should generally choose between using multiple processes or multiple threads based on the need for data sharing among the threads and the potential for unwanted interactions.

Threads vs. Processes: Sharing

Multithreaded programs make it *easier to share data* between the different threads of execution than do multi-process programs.

The two main program memory components that can be shared among OS threads are:

- **global variables**
- **heap memory**, i.e., dynamic memory

A **global variable** is a variable that is declared outside of `main` or any function body.

Memory for *global variables* is allocated in the **initialized data** or **uninitialized data** memory **segments** (where `static` data is also stored).

Threads vs. Processes: Sharing (contd.)

By default, a global variable's **scope** is the *entire program*, so all threads can access it.

(If one uses the `static` keyword when declaring the variable, its scope is limited to functions defined in the *same file* only.)

Threads do *not* require the use of **IPC mechanisms** since data can be shared by simply updating a global variable or dynamic memory.

Not shared among threads:

- **local** (**automatic**) **variables** in functions
  (unique to each thread since stored on thread's stack)
- **errno** variable giving syscall error code

POSIX Threads (Pthreads)

UNIX did not originally support multithreadeding.

As multithreading was added, different UNIXes adopted slightly different threading models.

Multithreading for UNIX was eventually standardized under POSIX, and this model is known as **POSIX Threads** or **Pthreads**.

This is the threading model supported by modern Linux systems.

Multithreaded programs can be written in C using the Pthreads system calls (which is what will be covered in these lectures).

C11 added support for writing multithreaded programs to C, but C11 threads are still not widely available (e.g., in glibc).

POSIX Threads (Contd.)

The Pthreads API is much more complicated than the API for multiple processes.

There are currently around 50 Pthreads system calls in Linux!

One can get more information about Linux Pthreads with the following commands:

```
man 7 pthreads
```

```
apropos pthread_
```

Note that when compiling with GCC, one must link with `libpthread` by including the option `-lpthread` (or just `-pthread`).

Pthreads API Overview

The Pthreads syscalls all start with the prefix "`pthread_`", e.g., `pthread_create`.

The entire set of calls can be divided into categories based on the types of thread operations they involve:

- creation and termination
- synchronization
- scheduling and other attributes
- thread IDs
- threads and signals

The Main Thread

When a *process* is created (with `fork()`), there is automatically a single running thread.

This is known as the **main** or **initial** thread in documentation.

In the modern Linux Pthreads implementation, the main thread has no special powers or importance.

Because of Linux' particular approach to **thread IDs** (TIDs), it is possible to determine whether a thread is the main thread (`syscall(SYS_gettid) == getpid()`).

Not all UNIXes allow the main thread to be identified.

Pthreads and Attributes

Many **attributes** are *process-wide*, so shared by all the Pthreads in a process, including:

- PID and PPID
- process group ID, session ID, controlling terminal
- UIDs and GIDs
- open file descriptors
- file mode creation mask
- current working directory (CWD)
- resource limits
- signal dispositions

Pthreads and Attributes (contd.)

Other Pthread attributes are *thread-specific*, including:

- thread ID (TID)
- stack
- program counter
- other registers
- signal mask
- `errno`

Pthreads and I/O

File descriptors are a *process-wide* resource.

Thus, threads in the same process share the *same file descriptors* in the process' **file descriptor table**.

This can cause problems, because when one thread carries out an I/O operation on an open file, it affects the file's **offset** for *every thread*.

This is not too different from what happens by default when a *child process* is created with `fork()`, though there are differences.

`fork()` creates a *copy* of a process, so the child gets a copy of the parent's *file descriptor table*.

Pthreads and I/O(contd.)

After the `fork()` the two processes have separate *file descriptor tables*, but initially the (now separate) file descriptors point to the *same entry* in the kernel's **open files table**.

Since an open file's **offset** is stored in the *open files table*, this means that when one process performs an I/O operation, it will affect the file offset for the other process as well.

Of course since processes have *separate file descriptor tables*, one process can close an FD and reopen the file, and from then the operations in one process will not affect the other's offset.

To have separate offsets, Pthreads would have to open the same file multiple times, creating unique FDs pointing to separate *open files table* entries, with separate *offsets*.

pread() and pwrite()

There are two I/O syscalls that are particularly useful when doing I/O in multithreaded programs: `pread()` and `pwrite()`.

Unlike `read()` and `write()`, these syscalls do not modify the file's *offset* (in the kernel *open files table*).

This means that they can be used to perform I/O on the *same file descriptor* in different threads, without the potentially serious interactions that `read()` and `write()` would have.

pread() and pwrite() (contd.)

`pread()` is like `read()`, but takes an *offset* argument telling it where to start reading:

```
ssize_t pread(int fd, void *buf, size_t count, off_t offset)
```

`pwrite()` is similar to `write`, but with an offset argument:

```
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset)
```

- `offset` is the position in *bytes* from the *start of the file* at which reading or writing is to take place

Pthreads and Signals

The POSIX **signal model** was developed long before **POSIX Threads**, so interaction of signals with Pthreads can be somewhat complicated.

Nonetheless, it is possible to use signals in multithreaded programs.

In fact, dedicating a thread to handling particular signals can be very effective.

The **Signals lectures** include detailed information about the use of signals with Pthreads.

Threads 2: Creation & Termination

1. Introduction
2. **Creation and Termination**
   - **thread creation**
   - **Pthreads example**
   - **passing data to a Pthread**
   - **race condition example**
   - **thread termination methods**
   - **returning data from a Pthread**
   - **joinable/detached threads**
   - **thread IDs (TIDs)**
   - **Pthread attributes**
3. Synchronization (part a)
4. Synchronization (part b)
5. Miscellaneous

Thread Creation

`pthread_create` is the system call to create a new pthread:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg)
```

- `thread` is the new thread's ID (returned via pointer)
- `attr` is an optional thread attributes structure; use `NULL` to create a thread with *default* attribute values
- `start_routine` is the name of the *function* that starts running in the new thread
- `arg` is the argument passed to `start_routine` (a `void*`)
- returns 0 on success, else a (positive) thread error number

Thread Creation (contd.)

A key aspect of thread creation is that a thread must begin running a particular *function*, and such functions must match a *specific prototype*.

The notation `void *(*start_routine) (void *)` means that the function supplied to `pthread_create()` must:

- take a single argument of type `void*` (i.e., any pointer type)
- return a value of type `void*` (i.e., any pointer type)

Note that `(*start_routine)` is C notation for a **function pointer**, but this is what results when you use a function *name* in a call.

Thread Creation (contd.)

The `start_routine` prototype requirements have the following implications for passing arguments to `start_routine`:

- if no arguments are required, use `NULL` as `arg`
- if a single argument is required, a pointer must be used and the desired argument obtained by *dereferencing* this pointer
- an array argument may be passed directly (since arrays are represented as pointers)
- if multiple arguments are required, define an appropriate `struct` and pass a pointer to an instance of the `struct`

The `start_routine` prototype requirements have the following implications for `start_routine`'s return value:

- the return pointer *must not reference stack addresses*
- if a return value is not required, return `NULL`

Pthreads Example

An example using two threads to simultaneously:

1. get commands from the terminal and send them to a server
2. receive output from the server and print it on the terminal

```
int main(...)
{
  ...
  int sockfd;
  ...connect to server, sockfd is FD...

  //Create second thread to read server data and print it out:
  pthread_t threadid;
  pthread_create(&threadid,NULL,handle_output,&sockfd);

  //Handle commands from the terminal in original thread:
  handle_commands(sockfd);
  ...
}
```


Pthreads Example (contd.)

```
void handle_commands(int sockfd)
{
  ...
  //Continue reading a command and sending to server, until read "bye":
  while(1) {
    fgets(buff,size,stdin);
    if (strcmp(buff,"bye\n") == 0) break;
    write(sockfd,buff,strlen(buff));
  }
  //Normal termination of entire process (all threads):
  exit(EXIT_SUCCESS);
}

void *handle_output (void *sockfd_ptr)
{
  //Recover sockfd, which was passed as pointer:
  int sockfd = *(int*)sockfd_ptr;  //Recover passed sockfd.
  ...
  while((nread = read(sockfd,buff,n)) > 0)
    write(1,buff,nread);
  ...
}
```


Passing Data to a Pthread

Starting a new Pthread is quite similar to calling a function.

However, with a function call you can pass data to the function via its *parameters*.

With Pthreads, the only mechanism to pass data/parameters to the new thread is via `pthread_create()`'s single `void* arg` parameter.

I.e., data must be passed to `start_routine` by passing a single *pointer* to a data object, which can be of *any type*.

In the example above, we saw how a file descriptor `int` was passed: by passing the address of the FD variable.

The FD had to then be recovered from the pointer by using the *dereference operator* (and compilers may require *casting* too).

Passing Data (contd.)

There is another, more subtle difference between passing data to a Pthread vs. to a function.

Because you are passing a *pointer* (i.e., memory address), that memory address must *remain valid and its contents unchanged* until the new thread is definitely done accessing the data.

Remember, however, that the OS controls when and for how long the thread calling `pthread_create()` and the newly created thread get run.

This means that a program cannot make any assumptions about when the new thread will have completed accessing the passed data during its execution!

Failing to consider this issue can lead to **race conditions**, which are very difficult to detect and diagnose.

Passed Data Race Condition Example

Server handling each client via a separate Pthread:

```
...
int listenfd = socket(...);
bind(listenfd,...);
listen(listenfd,....);

int connfd;
while (1) {
  //Accept new client connection:
  connfd = accept(listenfd,...);
  //Create a new thread to handle the new client:
  pthread_create(&threadid,NULL,handle_client,&connfd);
}
...
```


Race Condition Example (contd.)

Function run in each client thread:

```
void *handle_client (void *connfd_ptr)
{
  //Recover connfd, which was passed as pointer:
  int connfd = *(int*)connfd_ptr;

  ...use connfd to communicate with client...

  return NULL;
}
```


Race Condition Example (contd.)

The problem with this code is that `connfd`'s value can be changed before the new thread has retrieved it!

This may not be obvious, because the new Pthread will definitely be *created* before the server loops and runs `accept()` again.

However, even though the new Pthread may have been created before another `accept()`, it may not get run to the point of recovering `connfd`.

If a second client is `accept()`'ed *immediately* after `pthread_create()` returns, the value of `connfd` could get modified before the first client's thread recovers that client's FD.

Fixed Race Condition Example

Note use of *dynamic/heap memory* to fix race condition:

```
...
int listenfd = socket(...);
bind(listenfd,...);
listen(listenfd,....);

int connfd;
while (1) {
  connfd = accept(listenfd,...);
  int *connfd_ptr = malloc(sizeof(int));
  *connfd_ptr = connfd;
  pthread_create(&threadid,NULL,handle_client,connfd_ptr);
}
```

Fixed Race Condition Example (contd.)

Allocating fresh dynamic/heap memory to hold a copy of `connfd` fixes the problem, because each thread will have its own copy of `connfd`.

(`handle_client()` must `free` that memory when done with it.)

Passing a pointer to anything other than dynamic/heap memory in `pthread_create()` can be dangerous:

- *local variable* – value may be changed or (*stack*) memory deallocated before thread uses pointer
- *global/static variable* – value may be changed before thread uses pointer

Pthread Termination Methods

A Pthread's execution can be terminated in several ways:

- the `start_routine` function in `pthread_create()` calls `return`
- the thread calls `pthread_exit()`: `void pthread_exit(void *retval)`
- the thread is **cancelled** by another thread in the process calling `pthread_cancel()`
- any of the containing process' threads calls `exit()`
- the main/initial thread of the containing process calls `return` from its `main`

The first two methods are equivalent.

The first three methods terminate only a single thread.

The last two methods terminate the entire *process*—i.e., *all threads* in the process.

Returning Data from a Pthread

A thread is able to return data when it terminates, using a mechanism similar to that used to pass it data: a single `void*` value (a pointer to any oject type).

As noted above, this can be done by the thread's `start_routine` calling `return` or by the thread calling `pthread_exit()`.

This return pointer value can be obtained only by another thread that calls `pthread_join()`.

`pthread_join()` is the Pthreads equivalent of `wait()`.

(`pthread_join()` is discussed in a later lecture.)

Joinable/Detached Pthreads

Every Pthread will either be **joinable** or **detached**.

The joinable/detached status of a thread determines what happens to its return data (if any) when it terminates.

By default, a Pthread is *joinable* when created.

A *joinable* thread can have its return data obtained by another thread calling `pthread_join()`.

Only when a *terminated joinable thread* has been joined are all of its resources released.

This means that if a joinable thread terminates, and none of its peer threads "joins with it," we will have the Pthreads equivalent of a **zombie process**: resources will get wasted holding the return data.

Joinable/Detached Pthreads (contd.)

By contrast, a *detached* thread cannot be joined with.

As a side-effect of this, when a detached thread terminates, its *resources are automatically released.*

If the return data from a Pthread's `start_routine` is not going to be required, it is *advisable to set the thread to be detached*.

This can be done by having the thread call `pthread_detach()`:

```
int pthread_detach(pthread_t thread)
```

It can also be done by passing an appropriate `attr` argument to `pthread_create()` when creating the thread.

For more info, see the man page for: `pthread_attr_setdetachstate()`

Thread IDs

Each process is identified by a unique ID, the **process ID** (**PID**).

Each Pthread is identified by an ID, the **Thread ID** (**TID**).

Before returning, `pthread_create()` stores the TID of the new thread in the `pthread_t` variable pointed to by its `thread` argument.

It is this TID that is used in other `pthread_*` functions.

A thread may obtain its own TID by calling `pthread_self()`:

```
pthread_t pthread_self(void)
```

While `pthread_t` TIDs are similar to PIDs in some ways, there are critical differences!

Thread IDs (contd.)

The `pthread_self()` man page says:

- POSIX.1 allows an implementation wide freedom in choosing the type used to represent a thread ID
- E.g., either an arithmetic type or a structure is permitted
- Therefore, variables of type `pthread_t` can't portably be compared using the C equality operator (`==`); use `pthread_equal()`
- Thread identifiers should be considered opaque: any attempt to use a thread ID other than in pthreads calls is nonportable and can lead to unspecified results.
- Thread IDs are guaranteed to be unique only within a process.
- A thread ID may be reused after a terminated thread has been joined, or a detached thread has terminated.
- The thread ID returned by `pthread_self()` is not the same thing as the **kernel thread ID** returned by a call to `gettid()`.

Thread IDs (contd.)

In Linux, each thread actually has *two different* thread IDs:

1. POSIX TID:
   - the TID returned from `pthread_create()` and `pthread_self()`
   - of type `pthread_t` (need not be scalar/numeric)
   - in Linux defined as: `typedef unsigned long int pthread_t`
   - not necessarily unique system-wide (may be so in Linux)
   - TID required for other `pthread_*` functions
2. kernel TID:
   - the TID returned from `gettid()`
   - of type `pid_t` (an integer—)
   - *unique* system-wide (among all threads)
   - since unique and in *same space as PIDs*, can be used in same way to identify a thread
   - *main thread* in a process has its TID equal to its PID

Thread IDs (contd.)

Note that `gettid()` is Linux-specific.

`gettid()` has no C wrapper, so must be called as:

```
syscall(SYS_gettid)
```

(Be sure to include `syscall.h` header file.)

On Linux, can determine if thread is *main* (initial) thread:

```
syscall(SYS_gettid) == getpid()
```


Pthread Attributes

The are a set of Pthread-specific *attributes* that can be set for each Pthread when it is created.

These attributes deal with joinable/detached state, CPU scheduling and affinity, and thread stack size and position.

Attributes are set at creation time, by passing an object of type `pthread_attr_t` to `pthread_create()`.

A `pthread_attr_t` object must be initialized by calling:

```
pthread_attr_init()
```

Attributes can then be set in the object using a variety of calls named `pthread_attr_*`.

Threads 3: Synchronization (part a)

1. Introduction
2. Creation and Termination
3. **Synchronization (part a)**
   - **Pthread synchronization mechanisms**
   - **mutexes**
   - **condition variables**
   - **mutex+CV usage patterns**
4. Synchronization (part b)
5. Miscellaneous

Pthread Synchronization

Linux/UNIX provides several **synchronization mechanisms**.

Some synchronize only *processes*, some synchronize only *Pthreads*, and some can work with *both*.

The synchronization mechanims that work with *Pthreads* are:
- mutexes
- condition variables
- POSIX semaphores (not System V semaphores)
- joins (i.e., wait for thread termination)
- file locks

Pipes/FIFOs and signals can be used among Pthreads, but are not common.

Mutexes

A **mutex** is equivalent to a **binary semaphore** in functionality, but **mutexes** are *specifically for use with Pthreads*.

A mutex can be used to *lock* a **shared resource** such as a *global variable*, so that only one Pthread at a time can access/modify the resource:

- when a Pthread wants to access a shared resource, it first tries to *lock* the associated *mutex*
- if the mutex is *unlocked*, the lock call *locks the mutex and immediately returns*, and the Pthread can proceed to access the shared resource
- however if the mutex is already *locked*, the lock call will *block* until the lock holder unlocks it

Mutex Initialization

There are a number of steps and calls required to use mutexes.

Before a mutex can be accessed, it must be *initialized*.

A mutex is an object of type `pthread_mutex_t`.

Since a mutex will have to be able to be accessed from multiple threads in a process, it is common for mutexes to be stored in *global variables*.

Mutexes stored in global (or `static`) variables can be initialized by simply setting their values to `PTHREAD_MUTEX_INITIALIZER`:

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
```

This **static initialization** is simpler than **dynamic initialization**, and may be faster due to avoidance of runtime tests.

Mutex Initialization (cond.)

`pthread_mutex_init()` can be used to *dynamically intialize* a mutex:

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr)
```

This function must be used if the mutex is stored in memory that is not statically allocated: automatic/stack or dynamic/heap.

It must also be used if the mutex needs to be created with non-standard **attributes**.

The equivalent code to the static initialization above is:

```
pthread_mutex_t mtx;
pthread_mutex_init(&mtx,NULL);  //NULL for standard attributes
```


Mutex Locking and Unlocking

`pthread_mutex_lock()` *tests* and *locks/acquires* a mutex:

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

- if referenced mutex is *unlocked*, results in mutex being *locked*
- if mutex is *locked*, calling thread *blocks* until the mutex becomes unlocked
- returns 0 on success, else positive error code on error

Recall that with Dijkstra **semaphores**, the two operators were known as **decrement** (**P**) and **increment** (**V**).

A mutex is effectively a semaphore with values 1 and 0.

`pthread_mutex_lock()` implements the **decrement** operation (also referred to variously as *wait/lock/acquire*).

Mutex Locking and Unlocking (contd.)

`pthread_mutex_unlock()` *unlocks/releases* a mutex:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

- released the referenced mutex object
- returns 0 on success, else positive error code on error

`pthread_mutex_unlock()` implements the **increment** operation (also referred to variously as *post/signal/unlock/release*).

If multiple threads are blocked (in `pthread_mutex_lock()`) on a mutex that gets released, the thread that acquires the mutex will be determined by which thread gets scheduled next (by OS).

Mutex locking is a **discretionary access control mechanism**, since threads are free to access the shared resource without calling `pthread_mutex_unlock()`.

Basic Mutex Example

Mutex to protect threads count global variable:

```
int numthreads = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int main()
{
  for (int i=1; i<=5; i++) {
    pthread_t tid;
    pthread_create(&tid,NULL,threadfunc,NULL);
  }

  sleep(1);  //simple way to have all threads done

  //Print out global variable count of created threads:
  printf("Final threads count: %d\n",numthreads);

  exit(EXIT_SUCCESS);
}
```

Basic Mutex Example (contd.)

Each thread simply increments the global counter when created:

```
void *threadfunc(void *arg)
{
  pthread_mutex_lock(&mutex);
  int copy_numthreads = numthreads;
  copy_numthreads++;
  numthreads = copy_numthreads;
  pthread_mutex_unlock(&mutex);

  return NULL;
}
```

Without a mutex protecting `numthreads`, it can end up with an *incorrect final value* because the steps to increment `numthreads` from different threads may be *interleaved*!

I.e., there would be a *race condition*.

Other Mutex Syscalls

There are many other syscalls for manipulating mutexes, including:

- `int pthread_mutex_trylock(pthread_mutex_t *mutex)`
- `int pthread_mutex_destroy(pthread_mutex_t *mutex)`
- `int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex, const struct timespec *restrict abs_timeout)`
- `int pthread_mutexattr_init(pthread_mutexattr_t *attr)`
- `int pthread_mutexattr_destroy(pthread_mutexattr_t *attr)`
- `int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr, int *restrict type)`
- `int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type)`

Condition Variables

Condition variables are another Pthreads component that is frequently used in conjunction with *mutexes*.

Condition variables are used in Pthreads in a manner similar to how **signals** can be used for synchronization between processes: they allow one thread to block until another thread "signals" that some "condition" has changed, so the first thread should unblock and proceed.

Typically, the "condition" will involve one or more **shared variables** (or other resources shared among the threads).

For example, the condition might be that one or more pieces of data have been placed into a shared queue (for processing).

Condition Variables (contd.)

Operationally, a *condition variable* (CV) is an object of type `pthread_cond_t`.

Conceptually, a CV is linked with a *Boolean predicate*—i.e., a *condition* that is based on shared variables/data:

- when condition is true, some thread(s) should do something
- when condition is false, those thread(s) should block/sleep
- other thread(s) can cause a false condition to become true

Because conditions deal with *shared resources*, each condition variable is always linked with a *mutex*.

The mutex is used to protect access to the shared resources when testing and modifying the "condition."

Condition Variables (contd.)

For example, suppose that one thread generates data for others to consume, and the shared variable `avail` is used to indicate how much data is available.

Thus, the condition/predicate for the consuming threads to run is "`avail > 0`".

Both the producing and consuming threads will have to be able to access and modify `avail`.

Thus, a *mutex* must be used to protect `avail` and prevent *race conditions*.

Consumer thread(s) will access `avail` to see if it is positive: lock the mutex for `avail`, check if `avail`'s value is positive, unlock the mutex.

Condition Variables (contd.)

A huge problem with this is that it can result in a **busy wait**: if the condition is false, the consumer will immediately have to loop and do the testing again.

By using a *condition variable* (in conjunction with the mutex), the consumer thread(s) can *avoid busy waiting*:

- lock the mutex
- test `avail`
- if `avail > 0`
  - get datum; `avail--`; unlock mutex; process datum
- else [`avail == 0`]
  - unlock the mutex and *suspend* thread until *"signaled"* that `avail`'s value has changed
  - repeat from the first step

Condition Variable Initialization

As with mutexes, condition variables must be *initialized* before use.

As with mutexes, condition variable can be allocated *statically* or *dynamically*.

Statically allocated CVs can be initialized as:

```
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
```

`pthread_cond_init()` can intialize a *dynamically allocated* CV:

```
int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr)
```

Similar to mutexes, `PTHREAD_COND_INITIALIZER` initializes a CV with *standard attributes*.

Condition Variable Operations

The primary condition variable operations are **wait** and **signal**.

The CV *wait* operation will cause a thread to *block/suspend* until notification of condition change is received.

The CV *signal* operation notifies any *waiting/blocked* threads that the shared resource has changed state, causing the threads to be *unblocked*.

These operations require both:

- an (initialized) *mutex*
- an (initialized) *condition variable*

Waiting for a Condition

`pthread_cond_wait()` causes a thread to *wait/block* for a condition to become true:

```
int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex)
```

- `cond` is (a pointer to) the (initialized) *condition variable*
- `mutex` is (a pointer to) the associated (initialized) *mutex*
- call *atomically* does the following:
  (1) unlocks/releases the mutex at `mutex` and
  (2) causes the calling thread to *block* on the CV `cond`
- returns when another thread calls `pthread_cond_signal()` on the CV
- when returns (successfully), `mutex` will be locked (by the calling thread)

Signalling a Condition

`pthread_cond_signal()` "signals" *waiting/blocked* thread(s) that a condition has changed state:

```
int pthread_cond_signal(pthread_cond_t *cond)
```

- `cond` is (a pointer to) the (initialized) *condition variable*
- signals and *unblocks* at least one of the threads that are blocked on the specified CV `cond`
- has no effect if no threads are blocked on the CV when called
- if more than one thread is blocked on the CV, OS scheduling determines which thread(s) are unblocked
- it is possible that more than one thread may be unblocked by a single `pthread_cond_signal()` call (see **Spurious Wakeup**)

Condition Variables vs. Signals

We talk about condition variables as being used to "signal" another thread that some condition has changed.

However, the the condition variables mechanism has *nothing whatsoever* to do with the Linux/UNIX **signals mechanism** (**POSIX reliable signals** and **POSIX real-time signals**).

In particular, `pthread_cond_signal()` does *not* involve the *signals mechanism*.

Signals largely operate at the *process* level, so are generally not used for synchronization among *Pthreads*.

Spurious Wakeup

Due the needs of efficient implementations, it is possible that *more than one thread* blocked on the CV may get *unblocked* by each call to `pthread_cond_signal()`.

Said another way, a single `pthread_cond_signal()` call may cause *multiple threads* to return from blocked `pthread_cond_wait()` calls.

This effect is called **spurious wakeup**.

Each of the awoken threads will get run *sequentially* due to the shared mutex.

This means that the "condition" represented by the CV may be true for only the first run thread!

Thus, programs must wrap a condition-testing while loop around the condition wait call.

Mutex+CV Usage Patterns

Here is the standard usage pattern for a CV+mutex:

```
//Globals:
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
... shared_data = ...;
...

//"Producer" thread:
...
pthread_mutex_lock(&mutex);
...modify shared_data so condition(shared_data) is true...
pthread_mutex_unlock(&mutex);
pthread_cond_signal(&cv);
...
```


Mutex+CV Usage Patternd (contd.)

```
//"Consumer" thread:
...
pthread_mutex_lock(&mutex);
while (!condition(shared_data))
  pthread_cond_wait(&cv, &mutex);
...access/modify shared_data...
pthread_mutex_unlock(&mutex);
...
```


Mutex+CV Usage Patterns (contd.)

Since the return from `pthread_cond_wait()` does not guarantee the "condition" is true (for every thread), the condition predicate must be re-evaluated upon return.

That is why we cannot simply do this in the "consumer":

```
...
pthread_mutex_lock(&mutex);
pthread_cond_wait(&cv, &mutex);
...access/modify shared_data...
pthread_mutex_unlock(&mutex);
...
```

While `condition(shared_data)` may have been true when some thread called `pthread_cond_signal()`, this thread cannot be sure it is still true by the time it runs.

Threads 4: Synchronization (part b)

1. Introduction
2. Creation and Termination
3. Synchronization (part a)
4. **Synchronization (part b)**
   - **mutexes vs. semaphores**
   - **mutex & semaphore differences**
   - **semaphore semantics with mutex+CV**
   - **joining with terminated threads**
   - **file locks**
   - **producer-consumer examples**
5. Miscellaneous

Mutexes vs. POSIX Semaphores

POSIX semaphores can be used to synchronize processes *and* Pthreads.

Please see the **Process Synchronization lectures** for details of using POSIX semaphores.

Given that POSIX semaphores must be able to work between processes, mutexes will typically be *more efficient* for threads than will semaphores.

In particular, mutex operations can involve only machine code operations on common memory, while semaphore operations always require a syscall (kernel code).

Thus, if you are writing a multithreaded program, use mutexes (plus condition variables), not semaphores.

Mutexes vs. Semaphores (contd.)

Unfortunately, many people are confused about the relationship between semaphores and mutexes, resulting in a great deal of *misinformation* posted on the Internet.

Key point: *conceptually*, there is *no difference at all* between a binary (0/1) semaphore and a mutex!

Dijkstra's original papers that introduced this synchronization mechanism referred to them as **semaphores**, whether used for mutal exclusion or for more general producer-consumer problems.

0/1 semaphores used to enforce mutual exclusion (i.e., mutex), Dijkstra termed **binary semaphores**.

Those with a wider range of positive values, Dijkstra termed **general semaphores**.

Mutexes vs. Semaphores (contd.)

POSIX semaphores can supply the semantics of Dijkstra's binary and general semaphores.

(POSIX) mutexes can supply the semanatics of Dijkstra's binary semaphores only.

However, by combining a *mutex*, a *condition variable*, and a *shared integer*, it is possible to duplicate the semanatics of Dijkstra's general semaphores.

This means that to a large extent, mutexes and semaphores are functionally interchangeable in Linux/UNIX.

Nonetheless, there are differences (beyond Dijkstra's semantics) that can be useful to be aware of.

Mutex & Semaphore Differences

Broadcast signaling/unlocking:

- `pthread_cond_broadcast()` signals *all* threads waiting on a condition variable
  - can be used when the shared resource has changed in a way that more than one thread can proceed (e.g., producer-consumer problems with multiple consumers, where the producer can add multiple items)
  - makes it easier to implement a **read-write lock**: wake up all waiting readers when a writer releases its lock
  - can be used in a **two-phase commit algorithm** to notify all clients of an impending commit
- semaphores do *not* have anything comparable (`sem_post()` wakes only a *single* blocked process)

Mutex & Semaphore Differences (contd.)

Lock ownership:

- when a thread locks a *mutex* it is said to *own* the mutex
- *only the mutex owner* can unlock a locked mutex
- this is generally considered as enforcing good style
- no such ownership is enforced with POSIX semaphores
- any process can increment a semaphore decremented (to 0) by another process
- so must be careful when using semaphores for mutual exclusion

Mutex & Semaphore Differences (contd.)

Testing/peeking:

- both mutexes and semaphores allow testing/peeking to see if lock/wait will block (without actually blocking)
- `pthread_mutex_trylock()`
- `sem_trywait()`
- `sem_getvalue()`

Limited-time locking/waiting:

- both mutexes (plus condition variables) and semaphores can limit the length of time lock/wait calls can block
- `pthread_mutex_timedlock()`
- `pthread_cond_timedwait()`
- `sem_timedwait()`

Mutex & Semaphore Differences (contd.)

Lock count:

- mutexes can be set to support concept of **lock count**: single owner can lock multiple times and must unlock same number of times for lock to be released
- this is semantics beyond Dijkstra
- effectively like "binary semaphore" that can also go negative
- no comparable semantics for semaphores
- however, could duplicate effect with code using a second semaphore whose value was this lock count

Mutex & Semaphore Differences (contd.)

Error situations:

- mutexes can be set so that errors are thrown on: (these are not errors by default)
  - attempt to unlock an unlocked mutex
  - attempt by non-owner to unlock locked mutex
  - attempt by owner to lock already locked mutex
- there are no comparable errors for semaphores

Mutex & Semaphore Differences (contd.)

Interaction with signals mechanism:

- `pthread_cond_signal()` is not **async-signal-safe**, so not safe to use in an *asynchronously invoked signal handler*
- thus condition variablse (with mutexes) are not suitable for releasing a waiting thread by "signaling" from a signal handler (invoked asynchronously)
- `sem_post()` is *async-signal-safe*, so POSIX semaphores can be used from within asynchronously invoked signal handlers
- CVs/mutexes can be used with signals if the signals are waited for *"synchronously"* with `sigwaitinfo()`

Semaphore Semantics with Mutex+CV

As noted above, it is possible to duplicate the semantics of Dijkstra's general semaphores using Pthread mutexes + CVs.

Duplicating a (single) general semphore requires *three objects*:

1. a mutex
2. a condition variable
3. a shared integer variable (holds the semaphore value)

These objects might be initialized as follows:

```
//Globals:
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
int semaphore_value = SEM_INIT_VALUE;
```


Semaphore Semantics with Mutex+CV (contd.)

Increment/post/release step require 4 lines of code:

```
pthread_mutex_lock(&mutex);
semaphore_value++;
pthread_mutex_unlock(&mutex);
pthread_cond_signal(&cv);
```

Decrement/wait/acquire step requires 5 lines of code:

```
pthread_mutex_lock(&mutex);
while (semaphore_value == 0)
  pthread_cond_wait(&cv, &mutex);
semaphore_value--;
pthread_mutex_unlock(&mutex);
```

Semaphore Semantics with Mutex+CV (contd.)

The same thing could be accomplished with POSIX semaphores.

Unnamed semaphores are used among threads.

Intialization

```
//Globals:
sem_t *sem;

//inside main thread:
sem_init(&sem,0,SEM_INIT_VALUE;
```

Increment/post/release step:

```
sem_post(&sem);
```

Decrement/wait/acquire step:

```
sem_wait(&sem);
```


Joining with Terminated Threads

`pthread_join()` provides Pthreads with functionality similar to that provided by `wait()`/`waitpid()` for processes:
`int pthread_join(pthread_t threadid, void **retval)`

- causes calling thread to *block* until the specified (sibling) thread *terminates*
- `threadid` is the thread ID, as obtained from `pthread_create()` or `pthread_self()` (but not that from `gettid()`)
- unless `retval` is `NULL`, return value from joined thread (as returned by `pthread_exit()` is copied to location `*retval`
- returns 0 on success, else (positive) error code

This call can be used to make a thread wait to proceed beyond some point until another thread has completed its subtask.

Joining with Terminated Threads (contd.)

A few details about `pthread_join()`:

- target thread must be **joinable** (else error)
- if target thread has already terminated, `pthread_join()` returns immediately
- if target thread is *cancelled*, `PTHREAD_CANCELED` is placed in `*retval`
- if multiple threads simultaneously try to join with the same thread, the results are *undefined*
- if the thread calling `pthread_join()` gets *canceled*, target thread remains *joinable*
- returns error if two threads try to join with each other
- there is no analog of `waitpid(-1, &status, 0)`, i.e., "join with any terminated thread"

File Locks

File locks are useful with Pthreads just as with processes.

The three process-level calls can also be used with Pthreads:

- `fcntl()` – primary syscall, can lock regions or entire files
- `flock()` – not in SUS, can lock entire files
- `lockf()` – in SUS, in Linux just a simpler interface to `fcntl()`

In addition, there are a set of C library routines that can be used to lock *streams* (`FILE*`'s) in *Pthreads*:

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

Producer-Consumer Example #1

Multithreaded example using *bounded* queue and *mutexes*:

- producer run in main thread, consumer in second thread
- *global* record *queue*
- *fixed-size* records
- *bounded* queue size (e.g., implemented as circular array)
- producer adds single record whenever it accesses queue
- consumer removes single record whenever it accesses queue
- requires three (Dijkstra) *semaphores* for synchronization:
  - *binary semaphore* to enforce *mutual exclusion* accessing queue
  - *general semaphore* count of available/unread records in queue, to limit when consumer retrieves records
  - *general semaphore* number of free record slots in queue, to limit when producer stores new records
- Dijkstra semaphores implemented as follows:
  - binary semaphore: *mutex*
  - general semaphore count of records: *global* (`int`) + *mutex* + *condition variable*
  - general semaphore number of free slots: *global* (`int`) + *mutex* + *condition variable*

Producer-Consumer Example #1 (contd.)

Global objects, shared among threads:

```
//Queue for records:
queue_t *queue;

//Global mutexes, etc:
pthread_mutex_t q_op_mtx = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_t q_avail_mtx = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t q_avail_cv = PTHREAD_COND_INITIALIZER;
int q_avail_sem = 0;

pthread_mutex_t q_free_mtx = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t q_free_cv = PTHREAD_COND_INITIALIZER;
int q_free_sem = QUEUE_MAX;
```


Producer-Consumer Example #1 (contd.)

```
int main()
{
  //Initialize (circular array) queue:
  queue =  queue_init();

  //Create thread to run consumer:
  pthread_t threadid;
  pthread_create(&threadid,NULL,consumer,NULL);

  //In main thread here:
  producer();

  //Should not get here, infinite P-C loops:
  pthread_join(threadid,NULL);
  return EXIT_FAILURE;
  }
```


Producer-Consumer Example #1 (contd.)

```
void producer()
{
  record_t new_record;

  while(1) {
    //Wait for free slot in queue and decrement:
    pthread_mutex_lock(&q_free_mtx);
    while (q_free_sem == 0)
      pthread_cond_wait(&q_free_cv, &q_free_mtx);
    q_free_sem--;
    pthread_mutex_unlock(&q_free_mtx);

    pthread_mutex_lock(&q_op_mtx);  //acquire lock on queue
    new_record = ...create new record...
    queue_enqueue(queue, new_record);  //store new record in queue
    pthread_mutex_unlock(&q_op_mtx);  //release lock on queue

    //Increment avail count and signal:
    pthread_mutex_lock(&q_avail_mtx);
    q_avail_sem++;
    pthread_mutex_unlock(&q_avail_mtx);
    pthread_cond_signal(&q_avail_cv);
  }

  return;
}
```

Producer-Consumer Example #1 (contd.)

```
void *consumer(void *ignore)
{
  record_t next_record;

  while(1) {
    //Wait for non-zero avail count and decrement:
    pthread_mutex_lock(&q_avail_mtx);
    while (q_avail_sem == 0)
      pthread_cond_wait(&q_avail_cv, &q_avail_mtx);
    q_avail_sem--;
    pthread_mutex_unlock(&q_avail_mtx);

    pthread_mutex_lock(&q_op_mtx);  //acquire lock on queue
    queue_dequeue(queue, &next_record);  //retrieve next record in the queue
    pthread_mutex_unlock(&q_op_mtx);  //release lock on queue

    //Increment free count and signal:
    pthread_mutex_lock(&q_free_mtx);
    q_free_sem++;
    pthread_mutex_unlock(&q_free_mtx);
    pthread_cond_signal(&q_free_cv);

    ...process next_record...
  }

  return NULL;
}
```


Producer-Consumer Example #2

Multithreaded example using *bounded* queue and *semaphores*:

- Same setup as example #1, except for synchronization mechanisms.
- Dijkstra semaphores implemented as follows:
  - *binary semaphore*: POSIX semaphore (unnamed)
  - count of available records: POSIX semaphore (unnamed)
  - number of free record slots: POSIX semaphore (unnamed)

Producer-Consumer Example #2 (contd.)

Global objects, shared among threads:

```
//Queue for records:
queue_t *queue;

//Global (unnamed) semaphores:
sem_t q_op;
sem_t q_avail;
sem_t q_free;
```


Producer-Consumer Example #2 (contd.)

```
int main()
{
  //Initialize (unnamed) semaphores:
  sem_init(&q_op,0,1);
  sem_init(&q_avail,0,0);
  sem_init(&q_free,0,QUEUE_MAX);

  //Initialize (circular array) queue:
  queue =  queue_init();

  //Create thread to run consumer:
  pthread_t threadid;
  pthread_create(&threadid,NULL,consumer,NULL);

  //In main thread here:
  producer();

  //Should not get here, infinite P-C loops:
  pthread_join(threadid,NULL);
  return EXIT_FAILURE;
  }
```

Producer-Consumer Example #2 (contd.)

```
void producer()
{
  record_t new_record;

  while(1) {
    //Wait for free slot in queue and decrement:
    sem_wait(&q_free);

    sem_wait(&q_op);  //acquire lock on queue

    new_record = ...create new record...
    queue_enqueue(queue, new_record);  //store new record in queue

    sem_post(&q_op);  //release lock on queue

    //Increment avail count and signal:
    sem_post(&q_avail);
  }

  return;
}
```


Producer-Consumer Example #2 (contd.)

```
void *consumer(void *ignore)
{
  record_t next_record;

  while(1) {
    //Wait for non-zero avail count and decrement:
    sem_wait(&q_avail);

    sem_wait(&q_op);  //acquire lock on queue

    queue_dequeue(queue, &next_record);  //retrieve next record in the queue

    sem_post(&q_op);  //release lock on queue

    //Increment free count and signal:
    sem_post(&q_free);

    ...process next_record...
  }

  return NULL;
}
```


Running the P-C Examples

Running the P-C example code requires the following:

- define the record type `record_t`
- include the **circular array queue** code from the **Process Synchronization lectures**
- replace "create new record" with code to create a record consistent with `record_t`
- replace "process next record" with code to do something wth a record (e.g., print it)

Running the P-C Examples (contd.)

For example, we can create records that are strings with the format "`Message #%03d`" by doing:

- add before queue definition:
  `typedef char record_t[13];`
- add message counting to `producer()` and `consumer()`
- make "create new record":
  `snprintf(new_record,13,"Message #%03d",rec_cnt);`
- make "process next record":
  `printf("next_record: %s\n",next_record);`

Threads 5: Miscellaneous

1. Introduction
2. Creation and Termination
3. Synchronization (part a)
4. Synchronization (part b)
5. **Miscellaneous**
   - **Threads and CWD**
   - **Signals and Pthreads**

Threads and CWD

The *current working directory* (CWD) is a *process-wide* attribute.

This means that if one thread calls `chdir()` (or `fchdir()`), the CWD is changed for *all* threads in the process.

This can be annoying if different threads need to deal with **relative paths** for different directories.

It could require that each thread would have to construct appropriate **absolute paths** every time they `open()/creat()` a file.

Luckily, it is possible to avoid this problem by using `openat()`.

openat()

`openat()` is like `open()` but accepts a *directory* argument that it uses with *relative* file paths:

```
int openat(int dirfd, const char *pathname, int flags)
int openat(int dirfd, const char *pathname, int flags, mode_t mode)
```

`dirfd` is a **file descriptor** for a **directory** that has been previously `open()`'d (do it `O_RDONLY`).

By opening a directory and using `openat()` instead of `open()`, a thread can get much of the effect of having its own unique CWD.

Notes:

- if `dirfd` has value `AT_FDCWD`, then `openat()` interprets relative pathnames just as `open()` does (relative to CWD)
- `openat()` interprets *absolute pathnames* identically to `open()`

Signals and Pthreads

The signal model was developed long before **POSIX Threads**.

The interaction of signals with Pthreads can make it somewhat complex to use signals in **multithreaded programs**.

Nonetheless, it is possible to use signals in multithreaded programs.

In fact, the ability to have thread(s) dedicated to handling signals can be very useful.

When using signals in multithreaded programs , it is critical to understand that some aspects of signals apply **process-wide** and some are **thread-specific**.

E.g., while most older signal syscalls apply **process-wide**, several signal-related `pthread_` syscalls are **thread-specific**.

Signals and Pthreads (contd.)

Here are key points about signals and threads:

- **process-wide** vs. **thread-specific**:
  - most signals are *process-wide* by default
  - **synchronous signals** (e.g., `SIGFPE`) are *thread-specific*
  - `kill()` delivers a signal to an entire *process*
  - `raise()` delivers a signal to the calling *thread*
  - `pthread_kill()` and `pthread_sigqueue` will deliver a signal to a specific Pthread
  - target thread must be in same process as sender
  - there is no way to signal a single thread in another process
  - **timers** are *process-wide* resources (so shared by all threads)

Signals and Pthreads (contd.)

key points about signals and threads (contd.):

- signal **blocking**:
  - **signal masks** (signals being **blocked**) are a *per-thread* attribute
  - this means signal *blocking* is *thread-specific*
  - use `pthread_sigmask()` to set *signal masks* in multithreaded programs
  - in multithreaded programs, the behavior of `sigprocmask()` is *undefined*
  - a newly created thread *inherits a copy* of its creator's *signal mask*
  - the set of **pending signals** for the new thread is *empty*
  - `sigpending()` gives the set of pending signals for the calling *thread* only
  - this is the *union* of pending thread-specific and process-directed signals

Signals and Pthreads (contd.)

key points about signals and threads (contd.):

- signal **disposition**:
  - signal **disposition** is a *per-process* attribute
  - this means all threads will must have the *same disposition* for each signal
  - if a delivered signal's disposition is *termination*, the *entire process* (all threads) is terminated
- signal **handling**:
  - if a *process-wide* signal is *caught*, one thread is *randomly chosen* to run the handler
  - **synchronous signal handling** is better with Pthreads
  - approach: block signal(s) in all threads, then call `sigwait()` in *"handler thread"*

Pthreads Signal Syscalls

Signal-related syscalls that are specifically for use with *Pthreads*:

- `pthread_sigmask` – set signal mask for a thread (`sigprocmask()`'s behavior is undefined in multithreaded program)
- `pthread_kill` – send signal to one thread (in same process) (`kill()` sends signal to entire process)
- `pthread_sigqueue` – queue a signal and data to a thread in process (`sigqueue()` sends signal to entire process)

Synchronous Signal Handling

In multithreaded programs that must respond to particular signals, *asynchronous* signal handlers are generally *not* the best approach.

It is better to use **synchronous signal handling**:

- *block* signals to be handled in all threads
- create a thread for signal handling
  (or can create separate thread for each signal)
- have the handling thread(s) use `sigwait()` (or related) to set the thread up to respond to its particular signal(s)
- when `sigwait()` returns, the handling thread carries out the actions that would traditionally have been in handler functions

Example: Asynchronous Signal Handler

Classic *asynchronous* SIGINT handler example:

```
int main(void)
{
  struct sigaction act;
  memset(&act,0,sizeof(act));
  act.sa_handler = sigint_handler;
  sigemptyset(&act.sa_mask);
  sigaction(SIGINT,&act,NULL);  //Set SIGINT to be caught

  ...main program code...

  return EXIT_SUCCESS;
}

void sigint_handler(int sig)
{
  ...handler actions...
  return;
}
```


Example: Synchronous Signal Handling

Thread-based *synchronous* SIGINT handler example:

```
int main(void)
{
  sigset_t mask;
  sigemptysetset(&mask);
  sigaddset(&mask, SIGINT);  //Block SIGINT for entire process
  sigprocmask(SIG_SETMASK, &mask, NULL);

  pthread_t tid;
  pthread_create(&tid, NULL, sigint_handler_thread, NULL);

  ...main program code...
  return EXIT_SUCCESS;
}

void *sigint_handler_thread(void *ignore)
{
  sigset_t catching;
  sigemptyset(&catching);
  sigaddset(&catching, SIGINT);
  int sig;
  while(1) {
    sigwait(&catching, &sig);  //Wait for SIGINT (to be pending)
    ...handler actions...
  }
  return NULL;
}
```


sigwait() and related

Three syscalls for synchronous signal handling with threads:

- `int sigwait(const sigset_t *set, int *sig)`
- `int sigwaitinfo(const sigset_t *set, siginfo_t *info)`
- `int sigtimedwait(const sigset_t *set, siginfo_t *info, const struct timespec *timeout)`

Operation:

- suspend execution of the calling thread until one of the signals specified in `set` becomes *pending*
- *accept* the signal (removing from pending) and return
- if a signal in `set` is pending when called, return immediately
- if multiple signals in `set` are pending, retrieved signal is determined by usual ordering rules (see "`man 7 signal()`")

sigwait() and related (contd.)

`sigwait()`:

- passes signal number back in `sig`
- returns 0 on success, else a positive error number

`sigwaitinfo()`:

- passes `siginfo_t` structure describing the signal back in `info`
- returns signal number on success, else -1 with `errno` set

`sigtimedwait()` is just like `sigwaitinfo()` except:

- `timeout` argument specifies max time for which the thread can be suspended waiting for a signal