# 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.