Programming with POSIX* Threads 2
Based on slides from Intel Software College and
Multi-Core Programming – increasing performance through software multi-threading
by Shameem Akhter and Jason Roberts

Pthreads Mutex Variables

Simple, flexible, and efficient
Enables correct programming structures for avoiding race conditions
New types
- `pthread_mutex_t`
  - the mutex variable
- `pthread_mutexattr_t`
  - mutex attributes
Before use, mutex must be initialized
**pthread_mutex_init**

```c
int pthread_mutex_init( mutex, attr );
```

- **mutex**
  - mutex to be initialized

- **attr**
  - attributes to be given to mutex

- **ENOMEM** - insufficient memory for mutex
- **EAGAIN** - insufficient resources (other than memory)
- **EPERM** - no privilege to perform operation

---

**Alternate Initialization**

Can also use the static initializer

```c
PTHREAD_MUTEX_INITIALIZER
```

```c
pthread_mutex_t mtx1 = PTHREAD_MUTEX_INITIALIZER;
```

- Uses default attributes

Programmer must always pay attention to mutex scope

- Must be visible to threads
**pthread_mutex_lock**

```c
int pthread_mutex_lock( mutex );
```

**pthread_mutex_t *mutex**
- mutex to attempt to lock

---

**pthread_mutex_lock Explained**

Attempts to lock mutex
- If mutex is locked by another thread, calling thread is blocked

Mutex is held by calling thread until unlocked
- Mutex lock/unlock must be paired or deadlock occurs

**Errors**
- EINVAL - mutex is invalid
- EDEADLK - calling thread already owns mutex
**pthread_mutex_unlock**

```c
int pthread_mutex_unlock( mutex );
```

**pthread_mutex_t **mutex

- mutex to be unlocked

---

EINVAL - mutex is invalid
EPERM - calling thread does not own mutex

---

**Example: Use of mutex**

```c
#define NUMTHREADS 4
pthread_mutex_t gMutex; // why does this have to be global?
int g_sum = 0;

void *threadFunc(void *arg)
{
    int mySum = bigComputation();
    pthread_mutex_lock( &gMutex );
    g_sum += mySum; // threads access one at a time
    pthread_mutex_unlock( &gMutex );
}

int main()
{
    pthread_t hThread[NUMTHREADS];

    pthread_mutex_init( &gMutex, NULL );
    for (int i = 0; i < NUMTHREADS; i++)
        pthread_create(&hThread[i], NULL, threadFunc, NULL);

    for (int i = 0; i < NUMTHREADS; i++)
        pthread_join(hThread[i]);
    printf("Global sum = %f\n", g_sum);
}
```
**pthread_mutex_trylock**

```c
int pthread_mutex_trylock( mutex );
```

**pthread_mutex_t *mutex**
- mutex to be tested

**pthread_mutex_trylock**
- Tests mutex to see if it's locked
- If it is not locked, it locks it
- If it returns EBUSY the mutex is already locked
- Used when want to do more work if the mutex is already locked

**EINVAL** - mutex is invalid
**EPERM** - calling thread does not own mutex
**EBUSY** - the mutex is already locked

---

**Condition Variables**

Semaphores are conditional on the semaphore count
Condition variable is associated with an arbitrary conditional
- Same operations: wait and signal

Provides mutual exclusion
Condition Variable and Mutex

Mutex is associated with condition variable
- Protects evaluation of the conditional expression
- Prevents race condition between signaling thread and threads waiting on condition variable

Lost and Spurious Signals

Signal to condition variable is not saved
- If no thread waiting, signal is "lost"
- Thread can be deadlocked waiting for signal that will not be sent

Condition variable can (rarely) receive spurious signals
- Slowed execution from predictable signals
- Need to retest conditional expression
Condition Variable Algorithm

Avoids problems with lost and spurious signals

acquire mutex;
while (conditional is true)
  wait on condition variable;
  perform critical region computation;
  update conditional;
  signal sleeping thread(s);
release mutex;

Negation of condition needed to proceed

Mutex is automatically released when thread waits

May be optional

Condition Variables

pthread_cond_init, pthread_cond_destroy
• initialize/destroy condition variable

pthread_cond_wait
• thread goes to sleep until signal of condition variable

pthread_cond_signal
• signal release of condition variable

pthread_cond_broadcast
• broadcast release of condition variable
Condition Variable Types

Data types used

- `pthread_cond_t`
  - the condition variable
- `pthread_condattr_t`
  - condition variable attributes

Before use, condition variable (and mutex) must be initialized

**pthread_cond_init**

```c
int pthread_cond_init( cond, attr );
```

**cond**

- condition variable to be initialized

**attr**

- attributes to be given to condition variable

- **ENOMEM** - insufficient memory for condition variable
- **EAGAIN** - insufficient resources (other than memory)
- **EBUSY** - condition variable already initialized
- **EINVAL** - attribute is invalid
Alternate Initialization

Can also use the static initializer

```
PTHREAD_COND_INITIALIZER
```

```
pthread_cond_t cond1 = PTHREAD_COND_INITIALIZER;
```

- Uses default attributes

Programmer must always pay attention to condition (and mutex) scope
- Must be visible to threads

---

**pthread_cond_wait**

```
int pthread_cond_wait( cond, mutex );
```

- `pthread_cond_t *cond`
  - condition variable to wait on
- `pthread_mutex_t *mutex`
  - mutex to be unlocked
**pthread_cond_wait Explained**

Thread put to "sleep" waiting for signal on `cond`

Mutex is unlocked

- Allows other threads to acquire lock
- When signal arrives, mutex will be reacquired before `pthread_cond_wait` returns

**Errors**

- EINVAL - `cond` or mutex is invalid
- EINVAL - different mutex for concurrent waits
- EINVAL - calling thread does not own mutex

**pthread_cond_signal**

```c
int pthread_cond_signal( cond );
```

**Parameters**

- `cond` - condition variable to be signaled
**pthread_cond_signal Explained**

Signal condition variable, wake one waiting thread

If no threads waiting, no action taken
- Signal is not saved for future threads

Signaling thread need not have mutex
- May be more efficient
- Problem may occur if thread priorities used

---

`EINVAL` - cond is invalid

---

**pthread_cond_broadcast**

```c
int pthread_cond_broadcast( cond );
```

`pthread_cond_t *cond`
- condition variable to signal
**pthread_cond_broadcast Explained**

Wake all threads waiting on condition variable

If no threads waiting, no action taken

- Broadcast is not saved for future threads

Signaling thread need not have mutex

---

EINVAL EINVAL

---

**Programming with POSIX* Threads**

What’s Been Covered

How to create threads to execute work encapsulated within functions

Coordinate shared access between threads to avoid race conditions