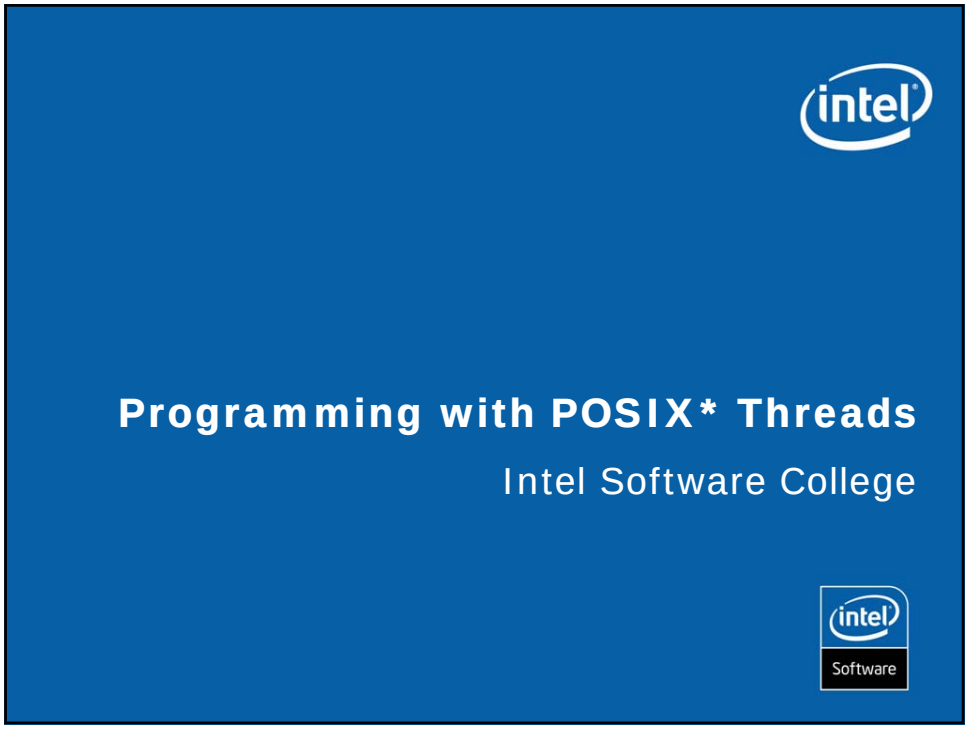

Objectives

Explore Pthreads “core” functions to create and synchronize threads

Compare Pthreads with Win32 threading API

What is Pthreads?

POSIX standard for threads programming interface (1995)

Implementations of POSIX standard are referred to as POSIX threads or Pthreads.

Latest Edition IEEE Std 1003.1,2004

Available for Linux and Unix OS family.

Available even for Windows!

- As Open Source http://sourceware.org/pthreads-win32/

C language interface

- programming types and procedure calls
- implemented as standalone library or as part of another library such as libc

Pthreads threading model

Threads exist within same process

All threads are peers

- No explicit parent-child model
- Exception: “main thread” holds process information

Pthreads API:

- ***Thread management:*** creating, detaching, joining, etc.
- ***Mutexes:*** deal with synchronization
- ***Condition variables:*** communications between threads that share a mutex

pthread_create

```
int pthread_create(tid, attr, function, arg);
```

pthread_t *tid
- handle of created thread

const pthread_attr_t *attr
- attributes of thread to be created

void *(*function)(void *)
- function to be mapped to thread

void *arg
- single argument to function

Compare with Win32 **CreateThread**

pthread_create Explained

Spawn a thread running the function

Thread handle returned via **pthread_t** structure
- Specify **NULL** to use default attributes

Single argument sent to function
- If no arguments to function, specify **NULL**

Check error codes!

EAGAIN - insufficient resources to create thread
EINVAL - invalid attribute

Example: Thread Creation

```
#include <stdio.h>
#include <pthread.h>

void *hello (void * arg) {
      printf("Hello Thread\n");
}

main() {
  pthread_t tid;

  pthread_create(&tid, NULL, hello, NULL);

}
```

What Happens?

Waiting for a Thread

```
int pthread_join(tid, val_ptr);
```

`pthread_t tid`

- handle of *joinable* thread

`void **val_ptr`

- exit value returned by joined thread

Compare with Win32
WaitForSingleObject

pthread_join Explained

Calling thread waits for thread with handle **`tid`** to terminate

- Only one thread can be joined
- Thread must be *joinable*

Exit value is returned from joined thread

- Type returned is **`(void *)`**
- Use **`NULL`** if no return value expected

`ESRCH` - thread (pthread_t) not found
`EINVAL` - thread (pthread_t) not joinable

Thread States

Pthreads threads have two states

- *joinable* and *detached*

Threads are joinable by default

- Resources are kept until **`pthread_join`**
- Can be reset with attributes or API call

Detached threads cannot be joined

- Resources can be reclaimed at termination
- Cannot reset to be *joinable*

No equivalent for Win32 Threads

Example: Multiple Threads

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4

void *hello (void *arg) {
      printf("Hello Thread\n");
}

main() {
  pthread_t tid[NUM_THREADS];
  for (int i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], NULL, hello, NULL);

  for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}
```


What's Wrong?

What is printed for myNum?

```
void *threadFunc(void *pArg) {
  int* p = (int*)pArg;
  int myNum = *p;
  printf( "Thread number %d\n", myNum);
}
. . .
// from main():
for (int i = 0; i < numThreads; i++) {
   pthread_create(&tid[i], NULL, threadFunc, &i);
}
```

Solution – “Local” Storage

```
void *threadFunc(void *pArg)
{
  int myNum = *((int*)pArg);
  printf( “Thread number %d\n”, myNum);
}
. . .

// from main():
for (int i = 0; i < numThreads; i++) {
  tNum[i] = i;
  pthread_create(&tid[i], NULL, threadFunc, &tNum[i]);
}
```


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

Compare with Win32 Mutex and Critical Section

pthread_mutex_init

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

pthread_mutex_t *mutex

- mutex to be initialized

const pthread_mutexattr_t *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
PTHREAD_MUTEX_INITIALIZER

```
pthread_mutex_t mtx1 = PTHREAD_MUTEX_INITIALIZER;
```

- Uses default attributes

Programmer must always pay attention to mutex scope

- Must be visible to threads

Compare with Win32
InitializeCriticalSection

pthread_mutex_lock

```
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

EINVAL - mutex is invalid
EDEADLK - calling thread already owns mutex

Compare with Win32 **WaitForSingleObject** or **EnterCriticalSection**

pthread_mutex_unlock

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

pthread_mutex_t *mutex

- mutex to be unlocked

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

Compare with Win32 **ReleaseMutex** or **LeaveCriticalSection**

Example: Use of mutex

```
#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 );
}

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);
}
```

Condition Variables

Mutexes implement synchronization by controlling thread access to data

Condition variables allow threads to synchronize based upon the actual value of data

Condition variable is associated with an arbitrary conditional

- Operations: wait and signal

Provides mutual exclusion

Compare with Win32 Events

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

Negation of condition needed to proceed

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

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

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

pthread_cond_t *cond

- condition variable to be initialized

const pthread_condattr_t *attr

- attributes to be given to condition variable

ENOMEM - insufficient memory for condition variable
EAGAIN - insufficient resources (other than memory)
EBUSY - condition variable already intialized
EINVAL - **attr** 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

Compare with Win32 **CreateEvent**

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

Compare with Win32
WaitForSingleObject

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

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

pthread_cond_signal

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

pthread_cond_t *cond

- condition variable to be signaled

Compare with Win32
Pulse Event

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

```
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 - cond is invalid

Compare with Win32
Auto & Manual Reset Events

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