Parallel Programming with pthreads

Mike Bailey

mjb@cs.oregonstate.edu

Oregon State University

pthreads Multithreaded Programming

- Pthreads is short for “Posix Threads”
- Posix is an IEEE standard for a Portable Operating System (section 1003.1c)
- Pthreads is a library that you link with your program

The pthread paradigm is to let *you* spawn functions as separate threads

- A thread is spawned by transferring control to a specific function that you have defined.
- The thread terminates when: (1) the function returns, or (2) when pthread_exit( ) is called
- All threads share a single executable, a single set of global variables, and a single heap (malloc, new)
- Each thread has its own stack (function arguments, private variables)
- pthreads is considered to be a low-level API. Oftentimes, other parallel APIs are written in terms of pthreads (e.g., OpenMP).

Compiling pthreads Programs

On Linux:

g++ -o program program.cpp -lm -pthread -fopenmp

On Windows:

From the class web site, get the files:

- pthread.h
- sched.h
- pthreadVC2.lib
- pthreadVC2.dll

These files came from:
http://sourceware.org/pthreads-win32

pthreads Data Types

pthread_t	*Thread id*
pthread_attr_t	*Thread attribute*
pthread_mutex_t	*Mutex id*
pthread_mutexattr_t	*Mutex attribute*
pthread_cond_t	*Condition id*
pthread_condattr_t	*Condition attribute*
pthread_barrier_t	*Barrier id*
pthread_once_t	*Call-once id*

Most of the **pthread_*_t** variables have corresponding **pthread_*_init( )** functions that *must* be called before using the variables

A Way to Clarify Referencing Memory Addresses

If you are an OpenGL programmer, the .h files you #include give you access to constructs like this:

typedef GLuint unsigned int;

so that your code can say:

GLuint a;
glGenBuffers(1, &a);

I have found it handy to do the same thing for addresses. I like to say:

typedef void * address_t ;

so that my code can look like this:

```
int Arg = 0;
pthread_create( &Thread, NULL, Func, (address_t)&Arg );

int *statusp;
pthread_join( Thread, (address_t *)&statusp );
```

instead of like this:

```
int Arg = 0;
pthread_create( &Thread, NULL, Func, (void *)&Arg );

int *statusp;
pthread_join( Thread, (void **)&statusp );
```

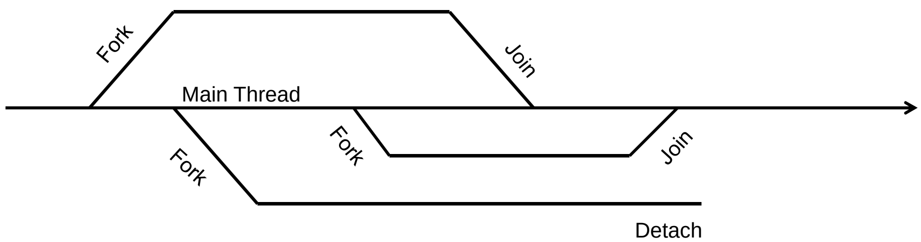

Creating pthreads

The *pthread* paradigm is to spawn an application's threads as function calls:

```
#include <pthread.h>
typedef   void *    address_t ;

pthread_t       Thread1, Thread2;
void *          Func1( void * );
void *          Func2( void * );
. . .
int val1 = 0;
int status1 = pthread_create( &Thread1, NULL, Func1, (address_t) &val1 );

switch( status1 )
{
      case 0:
                fprintf( stderr, "Thread 1 started successfully\n" );
                break;
      case EAGAIN:
                fprintf( stderr, "Thread 1 failed because of insufficient resources\n" );
                break;
      case EINVAL:
                fprintf( stderr, "Thread 1 failed because of invalid arguments\n" );
                break;
      default:
                fprintf( stderr, "Thread 1 failed for unknown reasons\n" );
}

int val2 = 1;
int status2 = pthread_create( &Thread2, NULL, Func2, (address_t) &val2 );
. . .
```

The NULL in **pthread_create** indicates that this thread's attributes are being defaulted

Spawning the pthreads Follows a Fork-Join or Fork-Detach Model



A Simple (but complete) pthreads Program

```
#include <stdio.h>
#include <math.h>
#ifdef WIN32
#include "pthread.h"
#else
#include <pthread.h>
#endif
typedef   void *    address_t ;

const int       SUCCESS = 0;
const int       FAIL = -1;

void *          Func1( address_t );
void *          Func2( address_t );

int
main( int argc, char *argv[ ] )
{
                pthread_t id1;
                int arg1 = 0;
                int status = pthread_create( &id1, NULL, Func1, (address_t)&arg1 );
                fprintf( stderr, "pthread_create status 1 = %d\n", status );

                pthread_t id2a;
                int arg2a = 1;
                status = pthread_create( &id2a, NULL, Func2, (address_t)&arg2a );
                fprintf( stderr, "pthread_create status 2a = %d\n", status );

                pthread_t id2b;
                int arg2b = 2;
                status = pthread_create( &id2b, NULL, Func2, (address_t)&arg2b );
                fprintf( stderr, "pthread_create status 2b = %d\n", status );
```

A Simple (but complete) pthreads Program

```
        address_t statusp;
        pthread_join( id1, &statusp );
        fprintf( stderr, "Return status 1 = %d\n", * (int *)statusp );

        pthread_join( id2a, &statusp );
        fprintf( stderr, "Return status 2a = %d\n", * (int *)statusp );

        pthread_join( id2b, &statusp );
        fprintf( stderr, "Return status 2b = %d\n", * (int *)statusp );

        pthread_exit( NULL );
        return 0;
}

void *
Func1( address_t  args )
{
        fprintf( stderr, "Hello from Func1 / Thread ID 0x%08x\n", pthread_self( )  );
        return (void *)&SUCCESS;
}

void *
Func2( address_t  args )
{
        int which = * (int *)args;

        fprintf( stderr, "Hello from Func2 / %d / Thread ID 0x%08x\n", which, pthread_self( )  );
        return (void *)&SUCCESS;
}
```


Output on Linux:

```
pthread_create status 1 = 0
Hello from Func1 / Thread ID 0xd77f9700
pthread_create status 2a = 0
Hello from Func2 / 1 / Thread ID 0xd6df8700
Hello from Func2 / 2 / Thread ID 0xd63f7700
pthread_create status 2b = 0
Return status 1 = 0
Return status 2a = 0
Return status 2b = 0
```

Output on Windows:

```
pthread_create status 1 = 0
Hello from Func1 / Thread ID 0x00851980
pthread_create status 2a = 0
Hello from Func2 / 1 / Thread ID 0x00851a18
pthread_create status 2b = 0
Hello from Func2 / 2 / Thread ID 0x00851d28
Return status 1 = 0
Return status 2a = 0
Return status 2b = 0
```

A Tale of Two pthreads

What's the difference between these two pieces of code?

1

```
int val1 = 0;
int status1 = pthread_create( &Thread1, NULL, Func1, (address_t) &val1 );


int val2 = 1;
int status2 = pthread_create( &Thread2, NULL, Func2, (address_t) &val2 );
. . .
```

2

```
int val = 0;
int status1 = pthread_create( &Thread1, NULL, Func1, (address_t) &val );


val = 1;
int status2 = pthread_create( &Thread2, NULL, Func2, (address_t) &val );
. . .
```

Hint: Go back and look at this:

```
void *
Func2( address_t args )
{
        int which = * (int *) args;
```


Using the Same Spawned Function in a Loop: A Dangerous Way

This is where it can get ugly . . .

2

```
int val = 0;
int status1 = pthread_create( &Thread1, NULL, Func, (address_t) &val );


val = 1;
int status2 = pthread_create( &Thread2, NULL, Func, (address_t) &val );
. . .
```

3

```
pthread_t   Threads[ NUM ];
for( int i = 0; i < NUM; i++ )
{
        int status = pthread_create( &Threads[ i ], NULL, Func, (address_t) &i );
}

. . .
```

Using the Same Spawned Function in a Loop: A Better Way

4

```
pthread_t   Threads[ NUM ];
int  Args[ NUM ];

for( int i = 0; i < NUM; i++ )
{
            Args[ i ] = i;
            int status = pthread_create( &Threads[ i ], NULL, Func, (address_t) &Args[ i ] );
}

. . .
```


If You'd Rather Import the Number of Threads Dynamically Instead of Statically as a #define

As a static #define :

```
pthread_t   Threads[ NUM THREADS];
int  Args[ NUM THREADS ];

for( int i = 0; i < NUM THREADS; i++ )
{
            Args[ i ] = i;
            int status = pthread_create( &Threads[ i ], NULL, Func, (address_t) &Args[ i ] );
}
```

As a dynamically-imported number (from a file, command line, etc) :

```
pthread_t   * Threads = new pthread_t  [ NumThreads ];
int  * Args = new int  [ NumThreads ];

for( int i = 0; i < NumThreads; i++ )
{
            Args[ i ] = i;
            int status = pthread_create( &Threads[ i ], NULL, Func, (address_t) &Args[ i ] );
}
```

Passing in Multiple Arguments to the Spawned Function

```
pthread_t   Threads[ NUM ];

struct  abc
{
            float a;
            int b;
            char *c;
}  Args[ NUM ];


for( int i = 0; i < NUM; i++ )
{
            int status = pthread_create( &Threads[ i ], NULL, Func, (address_t) &Args[ i ] );
}

. . .
```


Is There Any Problem with Doing Something Like This?

Goal: Want to pass an integer value of 10 into the spawning function Func( )

```
int status = pthread_create( &Threads[ i ], NULL, Func, (address_t) 10 );
```

or:

```
int value = 10;
int status = pthread_create( &Threads[ i ], NULL, Func, (address_t) value );
```

```
void *
Func( address_t  args )
{
            int ten = (int) args;

            . . .
```

Is There Any Problem with Doing Something Like This?
No, It will work, but it is always bad style to mix pointers and integers

Goal: Want to pass an integer value of 10 into the spawning function Func( )

We'd rather you do it this way:

```
int value = 10;
int status = pthread_create( &Threads[ i ], NULL, Func, (address_t) &value );
```

```
void *
Func( address_t  args )
{
		int *ip = (int *) args;
		int  ten  = *ip;
		. . .
```


Waiting for pthreads to Finish

```
address_t  statusp1;
address_t  statusp2;

pthread_join( Thread1, (address_t *)&statusp1 );
pthread_join( Thread2, (address_t *)&statusp2 );


if( statusp1 != NULL )
		fprintf( stderr, "Thread 1 exited with status %d\n", * (int *)statusp1 );


if( statusp2 != NULL )
		fprintf( stderr, "Thread 2 exited with status %d\n", * (int *)statusp2 );
```

A thread's status is the integer value that the spawned-off function returned, using its return statement.

Other Useful pthreads Management Functions

pthread_detach(pthread_t thread); *Detach a thread*

pthread_join(pthread_t thread, address_t * (&status_ptr)); *Wait for a thread to finish*

pthread_exit(address_t value); *Terminate this thread, returning value to any thread that is waiting for it*

pthread_cancel(pthread_t thread); *Cancel a thread*

pthread_kill(pthread_t thread, int sig); *Send a signal to a thread (e.g., SIGINT, SIGKILL)*

pthread_self( ) *Returns the thread id of this thread*

pthread_equal(pthread_t id1, pthread_t id2) *Tells you if two thread ids refer to the same thread. It returns 0 (false) or !0 (true).*

Forcing a Function to Be Called Just Once

void InitFunc(void); *Typically a function that sets some things up*

pthread_once_t inits;

pthread_once_init(&inits); *You must remember to do this*

pthread_once(&inits, InitFunc); *No matter how many times this line of code gets executed, InitFunc( ) will only be called once*

Getting and Setting a pthread's Information

```
pthread_attr_t   attr ;
int *            stackaddr;
size_t           stacksize;


pthread_attr_init( &attr );   You must remember to do this



pthread_attr_getstackaddr(   &attr,  (address_t *) &stackaddr );

pthread_attr_getstacksize(   &attr,               &stacksize );



pthread_attr_setstackaddr(   &attr,  (address_t) stackaddr );

pthread_attr_setstacksize(   &attr,              stacksize );
```

Supposedly, these functions have been deprecated in favor of:

```
pthread_attr_setstack(   &attr,  (address_t)     stackaddr,   stacksize );
pthread_attr_getstack(   &attr,  (address_t *) &stackaddr, &stacksize );
```


On the OSU EECS *babylon* Linux machine:

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

int
main( int argc, char *argv[ ] )
{
	pthread_attr_t  attr;
	size_t          stacksize;

	pthread_attr_init( &attr );
	pthread_attr_getstacksize( &attr, &stacksize );

	fprintf( stderr, "Stack Size = %d = 0x%08x\n", stacksize, stacksize );

	return 0;
}
```

Stack Size = 10485760 = 0x00a00000

= 10 MB

pthreads Mutexes

Goal: create a mutual exclusion ("mutex") lock that only one thread can acquire at a time:

```
pthread_mutex_t        Sync;
. . .
pthread_mutex_init( &Sync, NULL );    You must remember to do this
. . .
pthread_mutex_lock( &Sync );
            << code that needs the mutual exclusion >>
pthread_mutex_unlock( &Sync );


pthread_mutex_trylock( &Sync );


pthread_mutex_unlock ( &Sync );
```

The NULL in pthread_mutex_init( ) indicates that this mutex's attribute object is being defaulted

pthread_mutex_lock( ) blocks, waiting for the mutex lock to become available

If the lock is not available, **pthread_mutex_trylock( )** does not block. This is good if there is some more computing that could be done if the lock is not yet available. If the lock is available, trylock( ) acquires it.

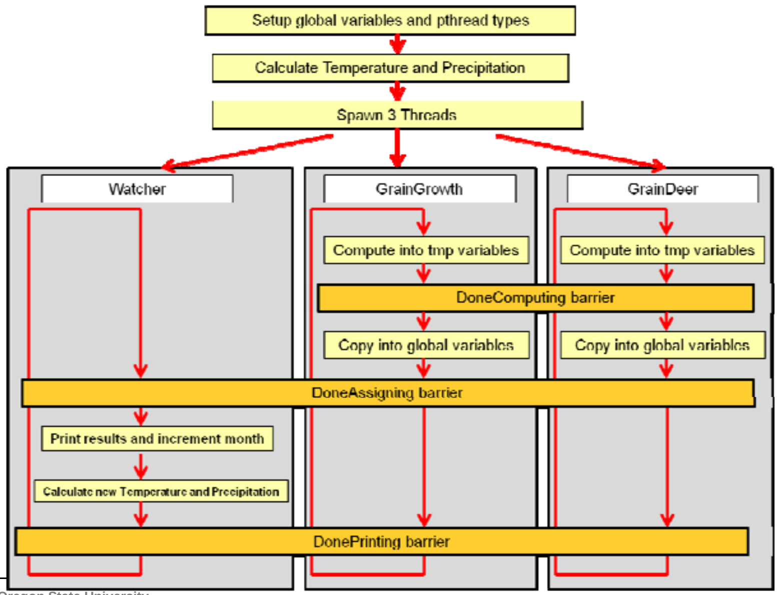
pthreads Barriers

```
#define NUMTHREADS      16

pthread_barrier_t    barrier;

pthread_barrier_init(  &barrier, NULL, NUMTHREADS );    You must remember to do this


pthread_barrier_wait( &barrier );
```

This is implemented with an internally-kept mutex variable, condition variable, and a count of how many threads have gotten to this point.

When NUMTHREADS threads finally call **pthread_barrier_wait( ),** the barrier is released.

Project #4 Use of Barriers



pthreads Condition Variables: Overview

This is *really* useful. It lets threads be suspended while waiting for some event to happen. Otherwise, they would have to keep polling. And, you are the one who gets to decide what the event is and when it occurs.

Thread #1

Program: Init a condition variable and a mutex

Program: Lock the mutex

Program: Call *pthread_cond_wait*

Pthreads: Suspends this thread's execution

Pthreads: Unlocks the mutex

Thread #2

Program: Lock the mutex

Program: Call *pthread_cond_signal* or *pthread_cond_broadcast*

Program: Unlock the mutex

Pthreads: Locks the mutex

Pthreads: Wakes the thread up

Program: Do what needs to be done

Program: Unlock the mutex as soon as it can

pthreads Condition Variables: Functions

```
pthread_mutex_t    lock;
pthread_cond_t     cond ;
struct timespec    delta_time;
```

```
struct timespec
{
    time_t  tv_sec;   // seconds
    long    tv_nsec;  // nanoseconds
} ;
```

```
pthread_mutex_init(  &lock,   NULL );
pthread_cond_init(   &cond,   NULL );
```

You must remember to do this

`pthread_cond_wait(  &cond,  &lock );` *Suspend this thread*

`pthread_cond_timedwait( &cond, &lock, &delta_time );` *Suspend this thread, but allow a timeout to wake it up*

`pthread_cond_broadcast( &cond );` *Wakeup all threads waiting*

`pthread_cond_signal( &cond );` *Wakeup one thread waiting*