Pthreads Bootcamp

Richard K. Wolf
Research Programmer
University of Illinois at Chicago
What are Pthreads?

- An API specified by the POSIX 1003.1c-1995 standard and implemented within a wide variety of operating systems.
- A portable way for developers to create multithreaded applications across a wide variety of platforms.
Why Pthreads?

- Because they often offer increased throughput.
  - Threads blocked on I/O do not affect other, running threads.
  - Throughput is increased on both uniprocessor and multiprocessor systems.
Why Pthreads?

- Because they can help conserve system resources.
  - Threads share many of the same resources within a process.
  - Creating threads is much cheaper than creating processes.
Why Pthreads?

- Because they offer a more natural programming model.
- Because they’re portable.
Considerations

- Execution context
- Scheduling
- Synchronization
Terminology

- Processes
- Threads
- Concurrency
- Parallelism
- “Synchronous”
- “Asynchronous”
Thread

- A single, independent flow of control within a program.
Process

- An entity composed of resources managed by an operating system and at least one thread.
Virtual Address Space

Identity
PID, UIC, GID

Resources
Open files, sockets, etc.

Registers
SP
PC
GP1
GP2 etc. ...

Process

Stack
main()

func1()

main() {
    ...
    
}

func1() {
    ...
    
}

fund2() {
    ...
    
}

Text, Global Data

Heap
Virtual Address Space

Thread 1

Registers

SP
PC
GP1
GP2 etc. ...

Identity

PID, UIC, GID

Resources

Open files, sockets, etc.

Virtual Address Space

Thread 2

Registers

SP
PC
GP1
GP2 etc. ...

Identity

Stack

main()
func1()

main()
func2()

main() {
  ...
}

func1() {
  ...
}

fund2() {
  ...
}

Stack

Text, Global Data

Heap
Concurrency

- Refers to tasks that appear to be running simultaneously, but which may, in fact, actually be running serially.
Parallelism

- Refers to concurrent tasks that actually run at the same time.
- Always implies multiple processors.
- Parallel tasks always run concurrently, but not all concurrent tasks are parallel.
“Asynchronous”

- A system call is asynchronous if a thread can continue running without waiting for the call to complete.
- Asynchronous calls are “non-blocking.”
- In a singly-threaded program, asynchronous calls are good.
“Synchronous”

- A system call is synchronous if a thread must wait for the call to complete before continuing.
- Synchronous calls are “blocking” calls.
- In a multithreaded program, synchronous calls are good—think synchronous!
Scheduling

- Threads are mapped from user space to kernel space by the pthreads library.
- Most operating systems utilize kernel threads (also known as lightweight processes).
- User threads are mapped onto kernel threads in one of three ways:
  - Using a one-to-one mapping.
  - Using a many-to-one mapping.
  - Using a many-to-many mapping.
Many-to-One

User Space

Library Interface

Kernel Space
Many-to-Many Userspace Interface
Programming
Considerations
What to Thread

- Programs that consist of several independent tasks.
- Most servers.
- Certain kinds of simulations.
Thread Models

- The boss/worker model
- Threads as peers
- The pipeline model
Boss/Worker

Boss

Worker

Worker

Worker
Pipelines

Task 1

Task 2

Task 3
Thread Safety

- Functions that can be called from multiple threads without destructive results are said to be thread safe.
- The use of global variables (extern or static) or static local variables makes a function thread-unsafe.
- Beware of some functions in the standard library (e.g., strtok).
Thread Safety

- Make threads safe by surrounding critical code with locks.
- Make threads safe by surrounding critical data with locks (better).
Functions that are thread-safe but do not rely on synchronization mechanisms to keep critical data safe are said be reentrant.

Functions can often be made to be reentrant by adding an extra argument in their interfaces.

Many ANSI C thread-unsafe routines have reentrant counterparts in pthreads.
Error Handling

- Pthreads routines always return either zero or an error as their return values.
- Pthreads routines do not set errno (because errno is defined as an extern int).
- However, pthreads defines an errno on a “per thread” basis for routines and system calls that rely on errno.
Thread Basics
Using Pthreads

- Include considerations
  ```
  #include <pthread.h>
  ```

- Library considerations
  ```
  cc -o myapp -lpthread myapp.c
  ```
## Pthreads Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pthread_t</code></td>
<td>Thread identifier</td>
</tr>
<tr>
<td><code>pthread_mutex_t</code></td>
<td>Mutex</td>
</tr>
<tr>
<td><code>pthread_cond_t</code></td>
<td>Condition Variable</td>
</tr>
<tr>
<td><code>pthread_key_t</code></td>
<td>Access key</td>
</tr>
<tr>
<td><code>pthread_attr_t</code></td>
<td>Thread attributes</td>
</tr>
<tr>
<td><code>pthread_mutexattr_t</code></td>
<td>Mutex attributes</td>
</tr>
<tr>
<td><code>pthread_condattr_t</code></td>
<td>Condition variable attributes</td>
</tr>
<tr>
<td><code>pthread_once_t</code></td>
<td>One-time initialization control context</td>
</tr>
</tbody>
</table>
Data Types

- Data types should be considered opaque!
- Can be initialized by a static initializer or dynamically via an “init” function call.
Spawning and Exiting

- Creating a thread
- “Joining” a thread
- “Detached” threads
Thread Routines

- **Creating a thread:**
  ```c
  int pthread_create( pthread_t *thread,
                    const pthread_attr_t *attr,
                    void *( *start )( void *arg ),
                    void *arg );
  ```

- **Exiting a thread:**
  ```c
  int pthread_exit( void *value );
  ```
Thread Routines

- **Joining a thread:**
  
  ```c
  int pthread_join( pthread_t *thread,
                   void **value );
  ```

- **Detaching a thread:**
  
  ```c
  int pthread_detach( pthread_t *thread );
  ```
Thread Routines

- Getting thread info:
  
  ```c
  pthread_t pthread_self( void );
  ```

- Testing thread equality:
  
  ```c
  int pthread_equal( pthread_t thread1, pthread_t thread2 );
  ```
Demo

Basic Pthreads
Synchronization
The Problem

- Threads are reasonably easy to create.
- But ensuring that threads share data properly is much more difficult!
- In nearly every multithreaded application, you will have to establish some kind of a synchronization strategy.
Race Conditions

- A race condition exists whenever two or more threads compete when trying to access to the same data.
- Synchronization mechanisms are designed to allow multiple threads to access shared data and avoid race conditions.
Example

- Imagine a “bank account server” that spawns a separate thread for every account transaction.
- Now imagine two people trying to deposit money into a shared account at the same time...
### Example

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread1</th>
<th>Thread2</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td></td>
<td></td>
<td>$200</td>
</tr>
<tr>
<td>First Operation</td>
<td>Read Balance ($200)</td>
<td></td>
<td>$200</td>
</tr>
<tr>
<td>Second Operation</td>
<td></td>
<td>Read Balance ($200)</td>
<td>$200</td>
</tr>
<tr>
<td>Third Operation</td>
<td></td>
<td>Add $100 ($300)</td>
<td>$200</td>
</tr>
<tr>
<td>Fourth Operation</td>
<td>Add $150 ($350)</td>
<td></td>
<td>$200</td>
</tr>
<tr>
<td>Fifth Operation</td>
<td></td>
<td>Write Balance ($300)</td>
<td>$300</td>
</tr>
<tr>
<td>Sixth Operation</td>
<td>Write Balance ($350)</td>
<td></td>
<td>$350</td>
</tr>
</tbody>
</table>
Synchronization

- Pthreads can be synchronized in three different ways:
  - By making threads joinable.
  - By using mutexes.
  - By using condition variables.
- More complex synchronizations can be built up from mutexes and condition variables.
Mutexes

- A mutex is a *mutually exclusive* lock.
- All threads agree that only one thread can lock a mutex at any specific time.
**Mutex Routines**

- **Statically create a mutex:**
  
  ```c
  pthread_mutex_t = PTHREAD_MUTEX_INITIALIZER;
  ```

- **Dynamically create a mutex:**
  
  ```c
  int pthread_mutex_init( 
  pthread_mutex_t *mutex, 
  pthread_mutex_attr *pthread_mutex_attr );
  ```

- **Destroy a dynamically created mutex:**
  
  ```c
  int pthread_mutex_destroy( 
  pthread_mutex_t *mutex );
  ```
Mutex Routines

- **Locking a mutex:**
  
  ```c
  int pthread_mutex_lock(
      pthread_mutex_t *mutex);
  ```

- **Checking for a mutex lock:**
  
  ```c
  int pthread_mutex_trylock(
      pthread_mutex_t *mutex);
  ```

- **Unlocking a mutex:**
  
  ```c
  int pthread_mutex_unlock(
      pthread_mutex_t *mutex);
  ```
Demo

Mutexes
Considerations

- Consider all mutex operations to be atomic.
- Don’t copy a mutex (but you can have as many pointers to the same mutex as you like).
- You only need to destroy mutexes that you dynamically initialize.
- Associate mutexes with the data they protect.
- Avoid deadlock!
## Deadlock

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread1</th>
<th>Thread2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>pthread_mutex_lock( &amp;mutexA );</code></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td><code>pthread_mutex_lock( &amp;mutexB );</code></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td><code>pthread_mutex_lock( &amp;mutexA );</code></td>
</tr>
<tr>
<td>4</td>
<td><code>pthread_mutex_lock( &amp;mutexB );</code></td>
<td></td>
</tr>
</tbody>
</table>
Condition Variables

- Used to signify a condition in which one or more threads have an interest.
- Always associated with a mutex.
How They Work

Lock Mutex

Test Cond.

Unlock Mutex and Proceed

Unlock Sleep Lock

Unlock Mutex

Set Cond = TRUE

Unlock Mutex

Wakeup Thread
Statically create a condition variable:

```c
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Dynamically create a condition variable:

```c
int pthread_cond_init(
    pthread_cond_t *cond,
    pthread_cond_attr *pthread_cond_attr);
```

Destroy a dynamically created condition variable:

```c
int pthread_cond_destroy(
    pthread_cond_t *cond);
```
Condition Variables

- Waiting on a condition variable:
  
  ```c
  int pthread_cond_wait(
    pthread_cond_t *cond,
    pthread_mutex_t *mutex);
  ```

- A timed wait for a condition variable:

  ```c
  int pthread_cond_timedwait(
    pthread_cond_t *cond
    pthread_mutex_t *mutex,
    struct timespec *expiration);
  ```
Condition Variables

- Signaling a met condition:
  
  ```c
  int pthread_cond_signal(
      pthread_cond_t *cond );
  ```

- Broadcasting a met condition:
  
  ```c
  int pthread_cond_broadcast(
      pthread_cond_t *cond );
  ```
Advanced Stuff

- Getting and setting thread attributes
- `pthread_once()`
- Thread keys
- Thread cancellation
Bedtime Reading

- **Pthreads Programming**; Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell; O’Reilly & Associates
- **Programming with POSIX Threads**; David R. Butenhof; Addison-Wesley Professional Computing Series
- **Multithreaded Programming with Pthreads**; Bill Lewis and Daniel J. Berg; Prentice Hall
Another Perspective

The Future of NLM Development/NetWare Kernel Services Development; Russell Bateman; Novell Developer Notes