O'REILLY

OSCON™
Open Source Convention

Pthreads Programming: A Hands-on Introduction

Adrien Lamothe
adrien@adriensweb.com

Monday, July 23, 2007
8:30 am – 12:00 noon
Room D139-140

Agenda

We're going to cover fundamental concepts of multi-threaded programming and Pthreads. We'll play around with some simple C programs, that utilize Pthreads, to learn the basics and to illustrate some important concepts. Due to time limitation, we may not be able to talk about some of the finer points of Pthreads programming.

This is not a comprehensive tutorial. The goal is to get you quickly familiar with multi-threaded programming concepts and Pthreads programming. If you haven't worked with threads before, this tutorial should serve as a good starting point. If you are more experienced, you may or may not benefit from this tutorial; there will be some discussion that may interest those of you with knowledge and experience of the topic.

This set of slides will serve as our guide. Once we get to the example programs the activity will also involve actually running and changing the programs, in order to have some fun and see first-hand how Pthreads actually work.

Background

- Concurrent systems and multi-threaded programming have been with us a number of years.

- Recent developments (clusters, multi-core microprocessors) have led to renewed interest in concurrent programming.

New Multicore Processors

- Sun Microsystems Niagara – 32 cores
- IBM Cell Broadband Engine (CBE) – up to 9 cores
- Parallax Propeller Chip – 8 cores
- Intel Core 2 Duo, Quad Core – 2 and 4 cores
- AMD Dual Core (soon to release quad core)

Background

- In the past, support for concurrent programming was uneven among vendors. Concurrent programming environments and tools were mostly confined to specialized applications.

- Real-time operating systems, in particular, strongly supported concurrent programming, due in some cases to a lack of memory protection. Many real-time systems also demand peak responsiveness.

Why Bother with Concurrent Programming?

- The current trend in microprocessor design is parallelism.

- Clock speed wasn't the answer, and “hit the wall”.

- Multi-core CPUs increase both performance and energy efficiency.

- Desktop and notebook computers are now parallel machines.

Why Bother with Concurrent Programming?

If your software application demands high performance and/or responsiveness, you should consider enabling it to take advantage of the parallelism inherent in today's computer systems.

Why Bother with Concurrent Programming?

- Not all software benefits from concurrency.
- Profile your application.
- Consider the cost/benefits.
- The learning curve is not as bad as you may think.

What types of software benefit?

The following conditions mark an application as a candidate for concurrency:

- Large numbers of asynchronous events.

- Blocking or waiting is frequent and tasks independent of those in the wait state are critical.

- Program is compute-intensive (“CPU Bound”) and it is possible to decompose the work.

- Your program needs to feel responsive to users. Even a simple program with a graphical user interface can benefit from running the GUI in a separate thread. There is a nice example of this in the O'Reilly Python Cookbook, where a Tkinker GUI runs in a separate thread.

Concurrent Software Architecture

Two main approaches to concurrent programming:

- Process Model
- Threading Model

Process Model

- Application distributed across multiple processes.

- Processes work in parallel, running on separate execution units (CPUs or cores.)

- Unix a great platform for Process Model applications, due to a good set of interprocess functionality (shared memory, semaphores, process control)

Threading Model

Parallelism achieved within a single process.

- Advantages:
  - Less time to context switch.
  - Cleaner syntax (in my opinion).
  - Creating a thread is faster than creating a process.
  - Less memory usage than multiple processes.
  - A threaded process is easier to manage.

User Space vs. Kernel Space Threads

User Space Threads (“User Threads”):

- Controlled by a scheduler linked into the process.
- Coordination with kernel scheduler required.
- “Dynamic” languages (Perl, Python, Ruby) use User Threads (each with their own semantics.) Java utilizes User Threads (called “green threads”.)

Kernel Space Threads (“Kernel Threads”):

- Controlled only by kernel scheduler.
- When supported well by the OS, the better way to implement threading.

Threads in Linux

- Original LinuxThreads library was broken and not fully POSIX compliant.

- Next Generation POSIX Threads (NGPT) was offered as a replacement for LinuxThreads.

- Native POSIX Thread Library (NPTL) was introduced in 2003, along with kernel modification. Quickly became the new Linux kernel library. Linux now supports Pthreads in the kernel. Ulrich Drepper and Ingo Molnar of Red Hat were the heroes behind NPTL.

Vendor Support

- Threading support varied among vendors.

- In 1995, a POSIX standard, P1003.1c (known as “Pthreads”) was issued, to standardize multi-threaded programming. This standard has largely remained the same, with some minor changes.

- Vendor support has improved (but I still don't trust them!)

- Linux supports threading well. Hey, this is OSCON! Who cares about proprietary operating systems?! (especially those posing as open) So let's just use Linux (O.K., maybe BSD or Mac OS X.) I feel better already!

Current Trends in Multi-Threaded Programming

- OpenMP (www.openmp.org) is a toolkit that aims to simplify multi-threaded programming in C/C++ and FORTRAN.

- Some compilers now have switches that cause generation of parallel code, without requiring any special programming by the developer. Sun Microsystems has this in their C/C++ compiler. Sun's motto is “let the compiler handle the complications of producing multi-threaded code.”

Current Trends in Multi-Threaded Programming

- Haskell (www.haskell.org) is a language that aims to simplify multi-threaded programming. This afternoon, in D136 from 1:30 to 5:00 pm, a Haskell tutorial will be presented.

- Transactional Memory is a technique to journal memory I/O, which promises to alleviate some of the potential pitfalls of multi-threaded programming. There will be a short talk about this on Wednesday, 9:30am-9:45pm, Portland Ballroom.

- The GNU organization has developed a library for C, called GNU Pth (for GNU Portable Threads, see www.gnu.org/software/pth). Pth was a foundation of the abandoned NGPT project.

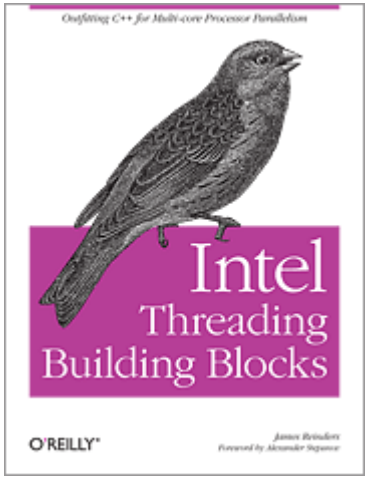
- GNU also has their own OpenMP project (called GOMP,) which appears to be inactive at this time.

- There are other OpenMP implementations.

Current Trends in Multi-Threaded Programming

- Intel Corporation has created a toolkit, a C++ class library, to simplify mutli-threaded programming and also has a debugger and profiler for threaded software. Intel plans to release a new open-source product for multi-threaded development at this OSCON (Tuesday, 8:30 am – 12:00 noon, D139-140.)

- A new O'Reilly book has just been released, titled Intel Threading Building Blocks.



Intel Threading Building Blocks (TBB)

- Intel has tied together a number of different parallel programming concepts and implemented them in TBB.

- TBB is only available for Intel processors and those that emulate them (i.e. AMD). Intel may open-source TBB, in which case you will be able to rebuild it for different platforms.

- If Intel open-sources TBB, be sure to check whether critical performance functionality has been moved into the Intel compilers. Also carefully check licensing.

- Do your own testing and benchmarking of TBB.

Intel Threading Building Blocks (TBB)

- TBB contains different types of mutexes, for different situations.

- TBB seems to do a good job supporting manipulation of complex data structures.

- TBB seems to be aimed at parallel mathematical operations.

- Complexity doesn't go away, it just gets expressed differently. In terms of coding, TBB is in some ways more complex than working with Pthreads.

- TBB is not standards compliant. It is a proprietary API.

Pthreads

Why Pthreads?

- Pthreads is a good way to start learning about concurrent programming.

- Pthreads is an international standard. YES, THAT IS A GOOD THING!

- Linux supports Pthreads well and is getting better. The recently released 2.6.22 Linux kernel improves thread performance.

Why Pthreads?

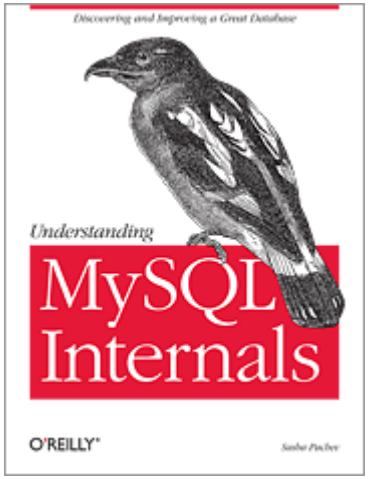
- Pthreads get the job done.
- Examples of successful Pthreads applications:

Apache

MySQL

Pthreads are a central component of MySQL

MySQL has created a C++ class to handle threads (class THD). The THD class is used to handle all client connections, and for other purposes (replication, delayed insert). The O'Reilly Understanding MySQL Internals book describes the THD class in some detail.

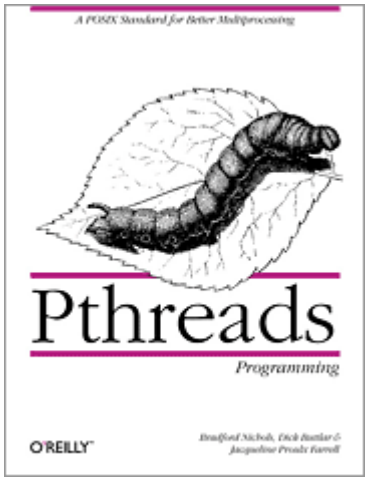


MySQL is a strong endorsement for Pthreads

MySQL is perhaps the strongest endorsement for Pthreads. Pthreads get the job done, and do it well, for this high-performance database.

What book should I read to learn Pthreads?

The O'Reilly Pthreads Programming book does an outstanding job of teaching Pthreads and fundamental concurrent programming concepts. While it may seem dated, most of the material is still relevant.



Pthread Attributes

- creation
- scheduling
- synchronization (mutexes, read/write locks, semaphores, condition variables)
- data access (keys)
- signal delivery and handling
- cancellation & destruction

How Do I Use Pthreads?

The bare essentials:

```
#include <pthread.h>

pthread_t  // pthread thread handle
pthread_create()
pthread_detach()  -or-  pthread_join()
pthread_cancel()
pthread_exit()

Keep in mind that the main() function is
also a thread! It is subject to the same
behaviour as the other threads.
```

pthread_create()

```
#include <pthread.h>

pthread_t  my_thread;
pthread_attr_t  thd_attrib;

pthread_create(&my_thread, &thd_attrib, &my_function,
&parameter);

void my_function(* parameter) {
  // do some stuff...
  }
```

- Parameters are passed by reference, so if you need to pass multiple parameters to your function, you must place them in a structure and pass a pointer to it.
- The thread handle (my_thread) is used for all subsequent references to the thread.
- The attribute parameter (2nd parameter) can be NULL, or a pthread_attr_t variable that has been initialized and set. This is how the priority and scheduling attributes of a thread are set.

our_include_file.h

```
// The important header is pthread.h, but let's include a bunch of stuff:

#include <pthread.h>


// NUM_OF_THREADS controls how many threads our examples will create.

#define NUM_OF_THREADS 9

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netdb.h>
#include <sys/un.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <netinet/in.h>
#include <syslog.h>
```

pthreads_1.c

```
#include "our_include_file.h"

void my_function(int *);

int main() {

  int number_of_threads = NUM_OF_THREADS;
  pthread_t thread_handle[NUM_OF_THREADS];
  int thread_id[NUM_OF_THREADS];
  int i;

  for (i = 0; i < number_of_threads; i++) {
    thread_id[i] = i;
    pthread_create(&thread_handle[i], NULL, (void *) my_function, (int *)
&thread_id[i]);
    // we should call pthread_detach() here, but we don't to show the
    // program will work anyway (but may end up not releasing system resources.)
    }

  for(;;) {
    // if we don't go into this loop, the program will exit before the
    // child threads have completed their jobs.
    }

  } // end main()

void my_function(int *my_thread_id) {

  printf("Hello from thread #%i!\n", *my_thread_id);

  } // end my_function()
```

Compilation

$cc -pthread -o pthread_test pthread.c

- or -

$cc -lpthread -o pthread_test pthread.c

Then, try running it:

$./pthread_test

Basic Synchronization

pthread_join()

- Calling thread waits for joined thread to terminate.

- Also has the effect of freeing thread resources.

- If you don't call pthread_join(), you need to call pthread_detach(), which also frees thread resources.

pthreads_2.c

```
#include "our_include_file.h"

void my_function(int *);


int main() {

  int number_of_threads = NUM_OF_THREADS;
  pthread_t thread_handle[NUM_OF_THREADS];
  int thread_id[NUM_OF_THREADS];
  int i;

  for (i = 0; i < number_of_threads; i++) {
    thread_id[i] = i;
    pthread_create(&thread_handle[i], NULL, (void *) my_function, (int *)
&thread_id[i]);
    pthread_join(thread_handle[i], NULL);
   }

// in this example, the calls to pthread_join() wait for the child threads
// to complete, so we don't put the main thread into an endless loop.

} // end main()

void my_function(int *my_thread_id) {

  printf("Hello from thread #%i!\n", *my_thread_id);

  } // end my_function()
```

Threads can work together

Complex numeric calculation is a good use for threads. We'll create a simple calculation (and pretend it is a complex one), to illustrate this.

pthreads_3-1.c

```
/// Now, the threads will collectively perform a calculation

#include "our_include_file.h"

void my_function(int *);
int result = 0;

int main() {

  int number_of_threads = NUM_OF_THREADS;
  pthread_t thread_handle[NUM_OF_THREADS];
  int integer_value[NUM_OF_THREADS];
  int i;

  for (i = 0; i < number_of_threads; i++) {
    integer_value[i] = i;
    pthread_create(&thread_handle[i], NULL, (void *) my_function, (int *)
&integer_value[i]);
    pthread_join(thread_handle[i], NULL);
    }

  printf("The calculation result is: %i\n", result);
  } // end main()

void my_function(int *my_integer_value) {

  result += *my_integer_value;

  } // end my_function()
```

Pthread Barrier Synchronization

Another type of synchronization is the Pthread Barrier:

- pthread_barrier_t // data type
- pthread_barrier_init()
- pthread_barrier_wait()
- pthread_barrier_destroy()

pthreads_3-2.c

```
/// We now use a Pthread barrier to synchronize the threads

#include "our_include_file.h"

void my_function(int *);

int result = 0;
pthread_barrier_t barrier;


int main() {

  int number_of_threads = NUM_OF_THREADS;
  pthread_t thread_handle[NUM_OF_THREADS];
  int integer_value[NUM_OF_THREADS];
  int i;
  long expected_result;

  pthread_barrier_init( &barrier, NULL, ( number_of_threads + 1 ) );

  for (i = 0, expected_result = 0; i < number_of_threads; expected_result += i, i++ ) {
    integer_value[i] = i;
    pthread_create(&thread_handle[i], NULL, (void *) my_function, (int *)
&integer_value[i]);
    pthread_detach(thread_handle[i], NULL);
    }

  pthread_barrier_wait(&barrier);

  printf("The calculation expected result is: %i\n", expected_result);
  printf("The calculation actual result is: %i\n", result);
```

pthreads_3-2.c (cont.)

```
  pthread_barrier_destroy(&barrier);

  pthread_exit(NULL);

  } // end main()


void my_function(int *my_integer_value) {

  result += *my_integer_value;
  pthread_barrier_wait(&barrier);

  } // end my_function()
```

Pthread Synchronization with Mutexes

The Pthread Mutex:

- pthread_mutex_t // data type
- pthread_mutex_init()
- pthread_mutex_lock()
- pthread_mutex_unlock()
- pthread_mutex_destroy()

pthreads_3-3.c

```
// We haven't protected the result variable in the previous examples.
// Now, we will protect result, using a Pthread Mutex lock.

#include "our_include_file.h"

void my_function(int *);

long result = 0;
pthread_barrier_t barrier;
pthread_mutex_t our_mutex;


int main() {

  int number_of_threads = NUM_OF_THREADS;
  pthread_t thread_handle[NUM_OF_THREADS];
  int integer_value[NUM_OF_THREADS];
  int i;
  long expected_result;

  pthread_barrier_init( &barrier, NULL, ( number_of_threads + 1 ) );
  pthread_mutex_init(&our_mutex, NULL);

  for (i = 0, expected_result = 0; i < number_of_threads; expected_result += i, i++ ) {
    integer_value[i] = i;
    pthread_create(&thread_handle[i], NULL, (void *) my_function, (int *)
&integer_value[i]);
    pthread_detach(thread_handle[i]);
    } // end for

  pthread_barrier_wait(&barrier);
```

pthreads_3-3.c (cont.)

```
  printf("The calculation expected result is: %i\n", expected_result);
  printf("The calculation actual result is: %i\n", result);

  pthread_barrier_destroy(&barrier);
  pthread_mutex_destroy(&our_mutex);
  pthread_exit(NULL);

  } // end main()


void my_function(int *my_integer_value) {

  pthread_mutex_lock(&our_mutex);
  result += *my_integer_value;
  pthread_mutex_unlock(&our_mutex);

  pthread_barrier_wait(&barrier);

  pthread_exit((void*) 0);

  } // end my_function()
```

Signal Handling with Pthreads

- Threads have their own signal masks.
- All threads share the *sigaction* structure of the process.
- Synchronously generated signals are delivered to specific threads (i.e. exceptions are handled by the offending thread, a pthread_kill() signal is delivered to the targeted thread.
- Asynchronously generated signals are delivered to a thread that can handle the signal.
- Threads can define their own signal handlers, but their usefulness is limited by not being able to lock mutexes.
- In the end, it is best to simply handle signals as usual.

Shared Data

Shared data is the bugaboo of parallel programming. You'll encounter constant warnings to avoid excessive shared data reads. Get over it. All computer systems share resources, in different areas. Yes, you should minimize shared data access, but at some point your threads will need to share some data.

While on the subject of shared data...

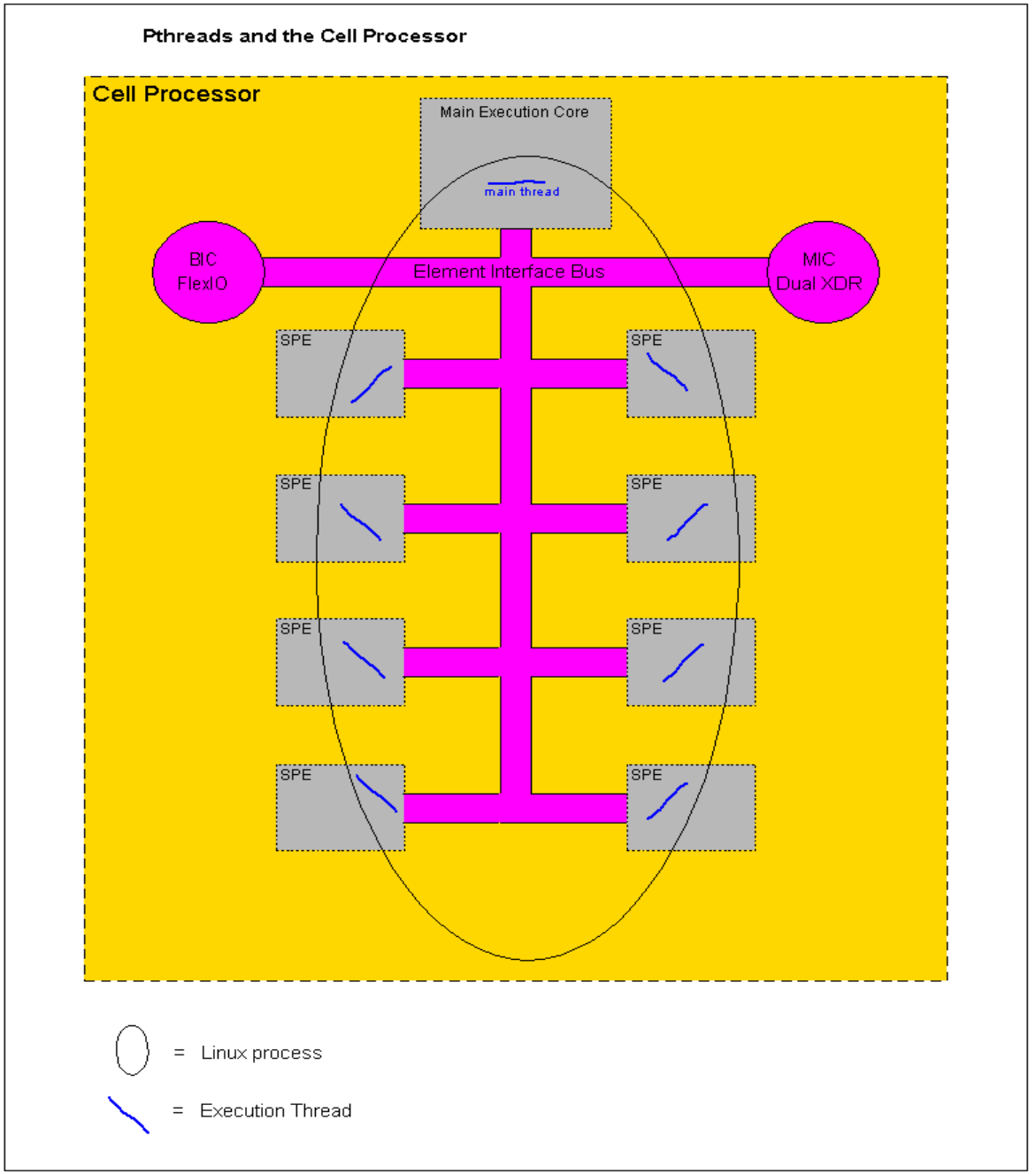
IBM Cell Broadband Engine (CBE, or the “Cell”)

The CBE has a main execution core and up to 8 “Synergistic Processing Elements” (SPEs). SPEs are execution cores. What makes them different and special, is they each have their own memory. This alleviates much of the trouble associated with shared memory.

CBE has very fast data buses, to facilitate message passing and broadband data transmission.

I find CBE the most interesting new processor architecture.

IBM Cell Broadband Engine (CBE, or the “Cell”)



IBM Cell Broadband Engine (CBE, or the “Cell”)

IBM has taken a hybrid approach to programming CBE, adding a library with functions applying to the SPE's.

Each SPE has dedicated memory. This greatly reduces the need for threads to share physical memory.

System Considerations for Concurrency/Parallelism

To achieve the best results from concurrency, the entire computer system needs to be designed with as much parallelism as possible. This means things such as:

- Parallel network interface controllers
- Simultaneous memory access
- Dedicated memory for execution cores

Even so, there are always going to be bottlenecks, there is no way to escape this. The way to minimize the effect of bottlenecks is to increase processing speed at bottleneck points. Thus, system design becomes an exercise in balancing bandwidth and concurrency.

Problems to avoid when programming threads

- Race Conditions
- Priority Inversion
- Deadly Embrace
- Thread unsafe libraries (due to reentrancy and thread cancellation)
- Running out of process resources

Things to do when working with Pthreads

- Keep things as clean and simple as possible
- Minimize shared data

Other Topics (time permitting)

- Thread Scheduling
- Thread Cancellation
- Test harnesses for our programs
- Debugging with GNU gdb
- Condition Variables
- Process resources and thread attributes
- Cleanup functions
- Thread cancellation
- Scheduling Pthreads
- Setting process resources

Summary

3 ½ hours is only enough time to introduce the topic and look at some simple Pthreads examples.

Pthreads programming is quite different than sequential programming. It involves thinking a bit differently about what is happening with your code and is also a bit tricky. Many people are comparing concurrent programming to object-oriented programming, in that it requires a change of thinking to accommodate the paradigm shift. I think this is a good analogy.

Despite uneven implementation among vendors, Pthreads is proven technology that powers some of the most pervasive and successful infrastructure software.

Summary (cont.)

Of course, Pthreads isn't for everyone. If you enjoy (out of either pleasure or necessity) the advantages of working with dynamic languages, then by all means continue working with them. Your programs can still benefit from the threading mechanisms of those languages. Some people have developed Python extensions, written in C, that allow them to access Pthreads from their Python programs.

Those of you developing server software, demanding high performance, should definitely take a close look at Pthreads. Download MySQL source code and see how they have harnessed Pthreads.

If your software is mathematically intensive, you should take a look at Intel Threading Building Blocks (TBB).

Thank you for coming, enjoy the rest of the show.

Adrien Lamothe

adrien@adriensweb.com