Pthreads Information

3.1. An overview of Pthreads

3.1.1. Introduction

Thread: A Thread is a 'Light Weight Process'. A thread is a stream of instructions that can be scheduled as an independent unit. A thread exists within a process, and uses the process resources. Since threads are very small compared with processes, thread creation is relatively cheap in terms of CPU costs. As processes require their own resource bundle, and threads share resources, threads are likewise memory frugal. There can be multiple threads within a process. Multithreaded programs may have several threads running through different code paths "simultaneously".

Shared Memory Programming

Shared-memory systems typically provide both static and dynamic process creation. That is, processes can be created at the beginning of program execution by a directive to the operating system, or they can be created during the execution of the program. The best-known dynamic process creation function is fork. A typical implementation will allow a process to start another, or child, process by a fork. Three processes typically manage coordinating among processes in shared memory programs. The starting, or a parent, process can wait for the termination of the child process by calling join. The second prevents processes from improperly accessing shared resources. The third provides a means for synchronizing the processes.

The shared-memory model is similar to the data-parallel model. It has a single address (global naming) space. It is similar to the message passing model in that it is multithreading and synchronous. However, data reside in a single, shared address space, thus does not have to be explicitly allocated. Workload can be either explicitly or implicitly allocated. Communication is done implicitly through shared reads and writes of variables. However, synchronization is explicit.

Shared variable programs are multi threaded and asynchronous, require explicit synchronizations to maintain correct execution order among the processes. Parallel programming based on the shared memory model has not progressed as much as message passing parallel programming. An indicator is the lack of a widely accepted standard such as MPI or PVM for message passing. The current situation is that shared-memory programs are written in a platform specific language for multiprocessors (mostly SMPs and PVPs). Such programs are not portable even among multiprocessors, not to mention multicomputers (MPPs and clusters). Three platform independent shared memory programming models are X3H5, Pthreads, and OpenMP.

3.1.2. Pthreads:

The Pthreads library is a POSIX C API thread library that has standardized functions for using threads across different platforms. Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications. In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required. For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995). Implementations that adhere to this standard are referred to as POSIX threads, or Pthreads. Most hardware vendors now offer Pthreads in addition to their proprietary API's. Pthreads are defined as a set of C language programming types and procedure calls. Vendors usually provide a Pthreads implementation in the form of a header/include file and a library that you link with your program.

3.1.3. Why Pthreads

- The primary motivation for using Pthreads is to realize potential program performance gains.
- When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.
- All threads within a process share the same address space. Inter-thread communication is more efficient and in many cases, easier to use than inter-process communication.
- Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways:
- Overlapping CPU work with I/O: For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, other threads can perform CPU intensive work.
- Priority/real-time scheduling: tasks that are more important can be scheduled to supersede or interrupt lower priority tasks.
- Asynchronous event handling: tasks that service events of indeterminate frequency and duration can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests.
- Multi-threaded applications will work on a uni-processor system; yet naturally take advantage of a multiprocessor system, without recompiling.
- In a multiprocessor environment, the most important reason for using Pthreads is to take advantage of potential parallelism. This will be the focus of the remainder of this tutorial. Designing Threaded Programs.
- In order for a program to take advantage of Pthreads, it must be able to be organized into discrete, independent tasks that can execute concurrently.

3.1.4. Basic Pthreads Library Calls

Brief Introduction to Pthreads calls

- **int pthread_create***(pthread_t *thread, const pthread_attr_t *attr, void*(*start_routine) (void), void *arg) ;*

  Creates a new thread, initializes its attributes, and makes it runnable.

  The ***pthread_create*** subroutine creates a new thread and initializes its attributes using the thread attributes object specified by the *attr* parameter. The new thread inherits its creating thread's signal mask; but any pending signal of the creating thread will be cleared

for the new thread.

The new thread is made runnable, and will start executing the *start_routine* routine, with the parameter specified by the *arg* parameter. The *arg* parameter is a void pointer; it can reference any kind of data. It is not recommended to cast this pointer into a scalar data type (int for example), because the casts may not be portable.

The ***pthread_create*** subroutine returns the new thread identifier via the thread argument. The caller can use this thread identifier to perform various operations on the thread. This identifier should be checked to ensure that the thread was successfully created.

- **void pthread_exit***(void *value_ptr);*

  Terminates the calling thread.

  The ***pthread_exit*** subroutine terminates the calling thread safely, and stores a termination status for any thread that may join the calling thread. The termination status is always a void pointer; it can reference any kind of data. It is not recommended to cast this pointer into a scalar data type (int for example), because the casts may not be portable. This subroutine never returns.

  Unlike the exit subroutine, the ***pthread_exit*** subroutine does not close files. Thus any file opened and used only by the calling thread must be closed before calling this subroutine. It is also important to note that the ***pthread_exit*** subroutine frees any thread-specific data, including the thread's stack. Any data allocated on the stack becomes invalidentifier, since the stack is freed and the corresponding memory may be reused by another thread. Therefore, thread synchronization objects (mutexes and condition variables) allocated on a thread'sstack must be destroyed before the thread calls the ***pthread_exit*** subroutine.

  Returning from the initial routine of a thread implicitly calls the ***pthread_exit*** subroutine, using the return value as parameter.

- **pthread_t pthread_self()**

  Returns the calling thread's identifier.

  The ***pthread_self*** subroutine returns the calling thread's identifier.

- **int pthread_join***(pthread_t thread, void **value_ptr);*

  The pthread_join subroutine blocks the calling thread until the thread specified in the call terminates. The target thread's termination status is returned in the status parameter.

  If the target thread is already terminated, but not yet detached, the subroutine returns immediately. It is impossible to join a detached thread, even if it is not yet terminated. The target thread is automatically detached after all joined threads have been woken up.

  This subroutine does not itself cause a thread to be terminated. It acts like the ***pthread_cond_wait*** subroutine to wait for a special condition.

- **int pthread_detach***(pthread_t thread, void **value_ptr);*

  Detaches the specified thread from the calling thread.

  The ***pthread_detach*** subroutine is used to indicate to the implementation that storage for the *thread* whose thread identifier is in the location thread can be reclaimed When that thread terminates. This storage shall be reclaimed on process exit, regardless of whether

the thread has been detached or not, and may include storage for thread return value. If thread has not yet terminated, ***pthread_detach*** shall not cause it to terminate. Multiple ***pthread_detach*** calls on the same target thread causes an error.

If the target thread is already terminated, but not yet detached, the subroutine returns immediately. It is impossible to join a detached thread, even if it is not yet terminated. The target thread is automatically detached after all joined threads have been woken up.

This subroutine does not itself cause a thread to be terminated. It acts like the ***pthread_cond_wait*** subroutine to wait for a special condition.

- **int pthread_mutex_init** *(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);*

  Initializes a mutex and sets its attributes.

  The ***pthread_mutex_init*** subroutine initializes a new mutex , and sets its attributes according to the *mutex* attributes object *attr*. The mutex is initially unlocked.

  After initialization of the mutex, the mutex attributes object can be reused for another mutex initialization, or deleted.

- **int pthread_mutex_destroy***(pthread_mutex_t *mutex);*

  Deletes a mutex.

  The ***pthread_mutex_destroy*** subroutine deletes the mutex *mutex*. After deletion of the mutex, the *mutex* parameter is not validentifier until it is initialized again by a call to the ***pthread_mutex_init*** subroutine.

- **int pthread_mutex_lock** *(pthread_mutex_t *mutex);*

  Locks a mutex.

  The mutex object referenced by *mutex* is locked by calling ***pthread_mutex_lock***. If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by *mutex* in the locked state with the calling thread as its owner.

- **int pthread_mutex_trylock** *(pthread_mutex_t *mutex);*

  Tries to lock a MuTex.

  The function ***pthread_mutex_trylock*** is identifierentical to ***pthread_mutex_lock*** except that if the mutex object referenced by *mutex* is currently locked (by any thread, including the current thread), the call returns immediately.

- **int pthread_mutex_unlock** *(pthread_mutex_t *mutex);*

  Unlocks a MuTex.

  The ***pthread_mutex_unlock*** function releases the mutex object referenced by *mutex*. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by mutex when ***pthread_mutex_unlock*** is called, resulting in the mutex becoming available, the scheduling policy is used to determine which thread shall acquire the mutex.

  The ***pthread_mutex_unlock*** function releases the mutex object referenced by *mutex*. The

manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by mutex when ***pthread_mutex_unlock*** is called, resulting in the mutex becoming available, the scheduling policy is used to determine which thread shall acquire the mutex.

3.1.5. Compilation, Linking and Execution of Pthreads Programs

use *#include<pthread.h>* in the program.
use *linker flag -lpthread* when linking.

The specific files to be used will differ with implementation on various platforms.

(A) Using command line arguments:

The compilation and execution details of Pthreads programs will vary from a system to another. The essential steps are common to all the systems.

 # cc <program name> -o <executable name> -lpthread

For example to compile a simple Hello World program user can type

 # cc Pthreads_HelloWorld.c -o Pthreads_HelloWorld -lpthread

(B) Using a Makefile:

For more control over the process of compiling and linking programs for Pthreads, you should use a '*Makefile*'. You may also use some commands in Makefile particularly for programs contained in a large number of files. The user has to specify the names of the program and appropriate paths to link some of the libraries required for Pthreads programs in the Makefile

To compile and link a Pthreads program, you can use the command

make

(C) Executing a Program:

To execute a Pthreads Program, type the name of the executable at command prompt.

<Name of the Executable>

For example, to execute a simple *HelloWorld* Program, user must type:

 # HelloWorld

The output must look similar to the following:

Hello World!

3.1.6. Compilation,Linking and Execution of Pthreads Programs on PARAM 10000:

PARAM 10000 runs SunOS 5.6. Sun has its own proprietary implementation of threads for SMP systems. They are called Solaris Threads.
Sun has included the support for POSIX Threads also.

The files and libraries needed are as follows:

POSIX Threads:
/usr/include/pthread.h
*/lib/libpthread.**
*/lib/libposix4.**

libposix4.* is needed if semaphore support is needed.

Compiling Pthreads Programs:

use *#include<pthread.h>* in the program.
use *linker flag -lpthread* when linking.
use *linker flag -lpthread4* when semaphores are used.

The standard Sun Workshop Compiler can be used to compile Pthread programs on PARAM 10000.

For the Hands-On Session on PARAM 10000, the application user can use *Makefile.pthread*

3.1.7. Example Program : "pthreads_HelloWorld.c"

The simple Pthreads program is "*HelloWorld*" program, in which the threads print the message "*Hello World*!" The number of threads is hard coded into the program - 2, thread1 prints "Hello" and thread2 prints "*World!*". The main thread just creates the child threads.

The simple Pthreads program in C language in which the threads print "*Hello World!*" message is explained below. We describe the features of the entire code and describe the program in details. We include the appropriate header file for threads support. The function *printmsg* is the routine that will be executed by the child threads. This routine prints the string that has been passed to it.

The following segment of the code explains these features. The description of program is as follows:

```
# include <pthread.h>
void * printmsg(char *s){
printf(" %s", s);
return;
}
```

Following the routine is the main function, the starting point for the program. We start by declaring the variables for child threads. *pthread_t* is the type of the variable to be declared. We declare two child threads.

```
pthread_t thread1, thread2;
```

After declaring the variables, we need to initialize the threads. We use *pthread_create* routine provided by the Pthreads library to accomplish the same.

```
pthread_create(&thread1, NULL, (void*(*)(void *))printmsg,
(void*)"Hello
"); /* Assign work to thread1.*/
pthread_create(&thread2, NULL, (void*(*)(void *))printmsg,
```

```
(void*)"World!"); /* Assign work to thread 2. */
```

pthread_create will initialize the thread, the thread attributes, the address of the routine the thread has to start executing and the parameters for that routine.

As soon as a thread is created, it will start executing the routine that has been assigned to it by *pthread_create*.

From the code it can be clearly seen that both the worker threads are executing the same routine. Each thread has its own copy of the stack variables for the routine. thread1 will execute the routine with *"Hello"* as the parameter and thread2 with "World!" as the parameter. Depending on the implementation of the standard on your platform, either of the threads may execute first giving the output as either *"Hello World!"* or *"World!Hello"*.

After creating the workers and assigning work to them, we have to inform the main thread to wait for them to finish. If this is not done, the main thread does its work and then terminates without checking if the worker threads have finished or not, terminating any worker thread that may be active at that time. We use the routine *pthread_wait* for this purpose. This is a blocking call. The call will return only after the thread specified has terminated.

```
/* Join the threads to inform the main thread to wait till
they are done.
*/
pthread_join(thread1, NULL);
pthread_join(thread2, NULL);
```

Here we inform the main thread to wait for the termination of *thread1* and *thread2*. After both the calls have returned, the main thread will continue and terminate.

Here we inform the main thread to wait for the termination of *thread1* and *thread2*. After both the calls have returned, the main thread will continue and terminate.

3.1.8. List of Extended Tools Available in Pthreads

- Etnus Total View Supports thread debugging.
- GDB supports threads minimally. - http://www.delorie.com/gnu/docs/gdb/gdb_25.html
- Smart GDB for Threads Debugging: http://hegel.ittc.ukans.edu/projects/smartgdb/
- The debugger that comes with the *DevPro* compiler set from Sun understands threads.
- Use the TNF utilities (included as part of the Solaris system) to trace, debug, and gather performance analysis information from your applications and libraries. The TNF utilities integrate trace information from the kernel and from multiple user processes and threads, and so are especially useful for multithreaded code.
- LockLint: *LockLint* verifies the consistent use of mutex and readers/writer locks in multithreaded ANSI C programs.LockLint performs a static analysis of the use of mutex and readers/writer locks, and looks for inconsistent use of these locking techniques. In looking for inconsistent use of locks, LockLint detects the most common causes of data races and deadlocks.

3.1.9. Pthreads Information on The Web

Pthreads Web pages:

- http://www.gnu.org/software/pth/related.html
- http://www.humanfactor.com/pthreads/
- http://www.etnus.com/Products/TotalView/started/threads0.html
- http://docs.sun.com
- http://www.as400.ibm.com/developer/threads/uguide/document.htm
- http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html
- http://userpages.umbc.edu/~schmitt/331F96/tshida1/thread.html

The usenet newsgroups:

- NewsGroup: **comp.programing.threads**
- Groups on Google

3.1.10.Reference Books on Pthreads

- Steve Kleiman, Devang Shah and Bart Smaalders, Programming With Threads. SunSoft Press. ISBN: 0-13-172389-8.
- Threads Primer - A Guide to Multithreaded Programming by Bil Lewis and Daniel J. Berg, First edition. ISBN 0-13-443698-9
- Pthreads Programing. by Bradford Nichols, Dick Buttlar and Jacqueline Proulx Farrell. ISBN: 1565921151

3.1.11. Pthreads FAQ

What are Threads?

A Thread is an encapsulation of the flow of control in a program. Most people are used to writing single-threaded programs - that is, programs that only execute one path through their code "at a time". Multithreaded programs may have several threads running through different code paths "simultaneously".

Why are threads interesting?

A context switch between two threads in a single process is considerably cheaper than a context switch between two processes. In addition, the fact that all data except for stack and registers are shared between threads makes them a natural vehicle for expressing tasks that can be broken down into subtasks that can be run cooperatively.

What are Pthreads?

Pthreads stands from POSIX Threads. POSIX threads, based on the IEEE POSIX 1003.1c-1995 (also known as the ISO/IEC 9945-1:1996) standard, part of the ANSI/IEEE 1003.1, 1996 edition, standard.

Lightweight process

A lightweight process (also known in some implementations, confusingly, as a kernel thread) is a schedulable entity that the kernel is aware of. On most systems, it consists of some execution context and some accounting information (i.e. much less than a full-blown process).
Several operating systems allow lightweight processes to be "bound" to particular CPUs; this guarantees that those threads will only execute on the

specified CPUs.

MT safety

If some piece of code is described as MT-safe, this indicates that it can be used safely within a multithreaded program, and that it supports a "reasonable" level of concurrency. This isn't very interesting; what you, as a programmer using threads, need to worry about is code that is not MT-safe. MT-unsafe code may use global and/or static data. If you need to call MT-unsafe code from within a multithreaded program, you may need to go to some effort to ensure that only one thread calls that code at any time. Wrapping a global lock around MT-unsafe code will generally let you call it from within a multithreaded program, but since this does not permit concurrent access to that code, it is not considered to make it MT-safe.

Scheduling

Scheduling involves deciding what thread should execute next on a particular CPU. It is usually also taken as involving the context switch to that thread.

What are the different kinds of threads?

There are two main kinds of threads implementations:

- ***User-space threads, and***
- ***Kernel-supported threads.***

There are several sets of differences between these different threads implementations.

- ***Architectural differences***

  User-space threads live without any support from the kernel; they maintain all of their state in user space. Since the kernel does not know about them, they cannot be scheduled to run on multiple processors in parallel.

  Kernel-supported threads fall into two classes. In a "pure" kernel-supported system, the kernel is responsible for scheduling all threads. Systems in which the kernel cooperates with a user-level library to do scheduling are known as two-level, or hybrid, systems. Typically, the kernel schedules LWPs, and the user-level library schedules threads onto LWPs. Because of its performance problems (caused by the need to cross the user/kernel protection boundary twice for every thread context switch), the former class has fewer members than does the latter (at least on Unix variants). Both classes allow threads to be run across multiple processors in parallel.

- ***Performance differences***

  In terms of context switch time, user-space threads are the fastest, with two-level threads coming next (all other things being equal). However, if you have a multiprocessor, user-level threads can only be run on a single CPU, while both two-level and pure kernel-supported threads can be run on multiple CPUs

simultaneously.

- ***Potential problems with functionality***

  Because the kernel does not know about user threads, there is a danger that ordinary blocking system calls will block the entire process (this is bad) rather than just the calling thread. This means that user-space threads libraries need to jump through hoops in order to provide "blocking" system calls that don't block the entire process. This problem also exists with two-level kernel-supported threads, though it is not as acute as for user-level threads. What usually happens here is that system calls block entire LWPs. This means that if more threads exist than do LWPs and all of the LWPs are blocked in system calls, then other threads that could potentially make forward progress are prevented from doing so.

  The Solaris threads library provides a reasonable solution to this problem. If the kernel notices that all LWPs in a process are blocked, it sends a signal to the process. This signal is caught by the user-level threads library, which can create another LWP so that the process will continue to make progress.

Contents