Real Time Programming with Pthreads

A S Prakash
NRCFOSS
asprakash_at_au-kbc.org

Thread Introduction

- Thread
- Process
- Pthreads
- Why Pthreads
- Comparison

Thread

- A thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system.
- In the UNIX environment a thread:
- Exists within a process and uses the process resources
- Has its own independent flow of control as long as its parent process exists and the OS supports it
- May share the process resources with other threads that act equally independently (and dependently)
- Dies if the parent process dies - or something similar
- Is "lightweight"

Process

- A process is created by the operating system, and requires a fair amount of "overhead"
- Processes contain information about program resources and program execution state, including:
  - Process ID, process group ID, user ID, and group ID, Environment, Working directory, Program instructions, Registers, Stack, Heap, File descriptors, Signal actions, Shared libraries, Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

Pthreads

- Pthreads are defined as a set of C language programming types and procedure calls, implemented with a `pthread.h` header/include file and a thread library - though the this library may be part of another library, such as `libc`.
- Libraries implementing the POSIX Threads standard are often named Pthreads
- Standards: POSIX 1003.1, POSIX 1003.1b and POSIX 1003.1c

Why Pthreads

- To realize potential program performance gains
- the cost of creating and managing a process
- Managing threads requires fewer system resources than managing processes.
- A comparison between *fork()* and *pthread_create()*

Why Pthreads cont..

- 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.
- Overlapping CPU work with I/O
- Priority/real-time scheduling
- Asynchronous event handling
- The primary motivation for considering the use of Pthreads on an SMP architecture is to achieve optimum performance

Comparison

- the following table compares timing results for the **`fork()`** subroutine and the **`pthreads_create()`** subroutine. Timings reflect 50,000 process/thread creations, were performed with the `time` utility

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
AMD 2.4 GHz Opteron (8cpus/node)	41.07	60.08	9.01	0.66	0.19	0.43
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.24	30.78	27.68	1.75	0.69	1.10
IBM 1.5 GHz POWER4 (8cpus/node)	104.05	48.64	47.21	2.01	1.00	1.52
INTEL 2.4 GHz Xeon (2 cpus/node)	54.95	1.54	20.78	1.64	0.67	0.90
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.54	1.07	22.22	2.03	1.26	0.67

Designing Thread Programs

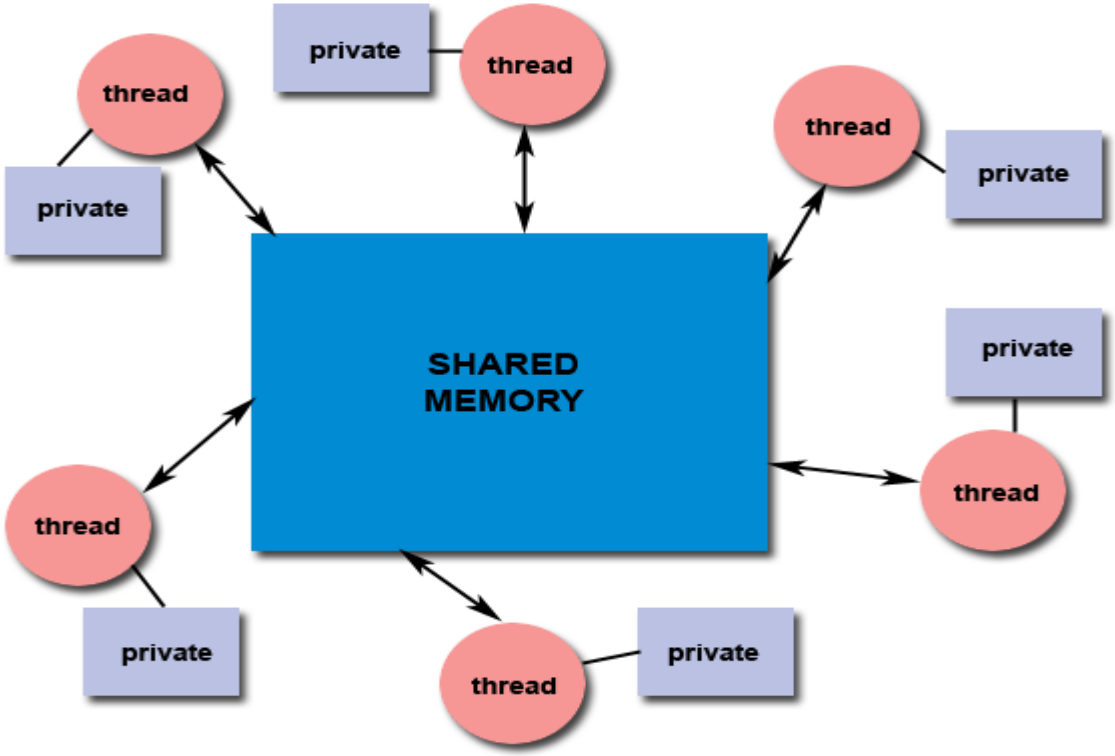
- Parallel Programing
- Shared memory model
- Thread safeness

Parallel programming

- pthreads are ideally suited for parallel programming
- In order for a program to take advantage of Pthreads, it must be able to be organized into discrete, independent tasks which can execute concurrently.

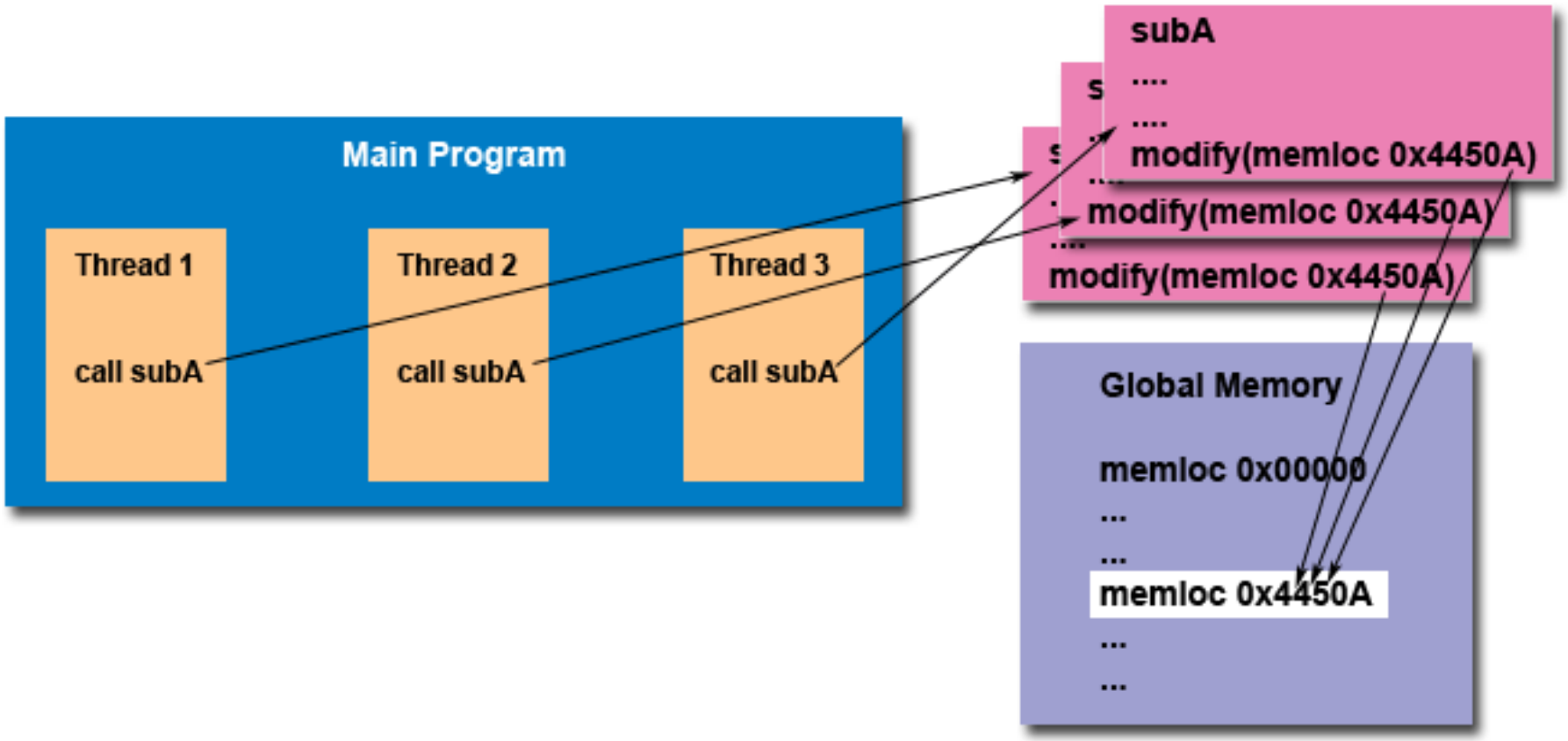
Shared Memory model

- All threads have access to the same global, shared memory
- Threads also have their own private data
- Programmers are responsible for synchronizing access (protecting) globally shared data.



Thread-safeness

- Application's ability to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions
- For example, suppose that your application creates several threads, each of which makes a call to the same library routine:



Pthread API

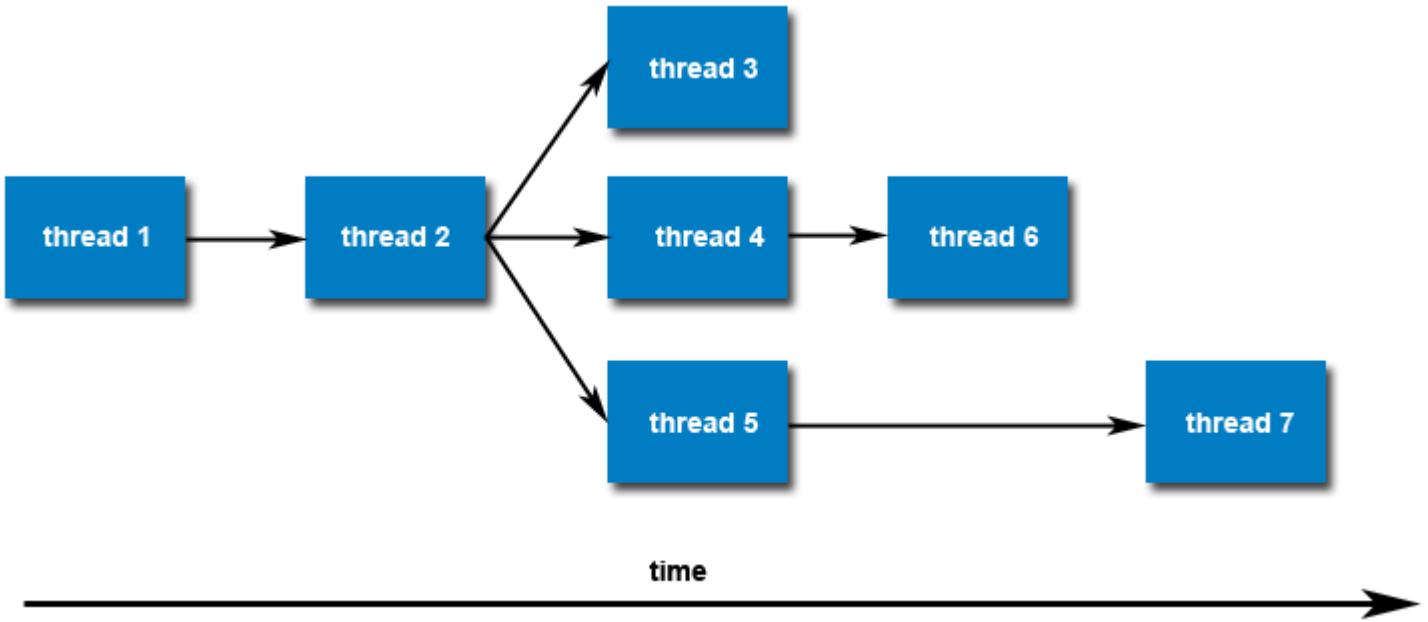
- Thread management
- Stack management
- Mutex
- Conditional variables

Thread Management

- Creating threads
- Terminating threads
- Joining threads
- Detaching threads

Creating Threads

- *main()* program comprises a single, default thread. All other threads must be explicitly created by the programmer
- *pthread_create* creates a new thread and makes it executable
- *pthread_create (thread,attr,start_routine,arg)*



Terminating threads

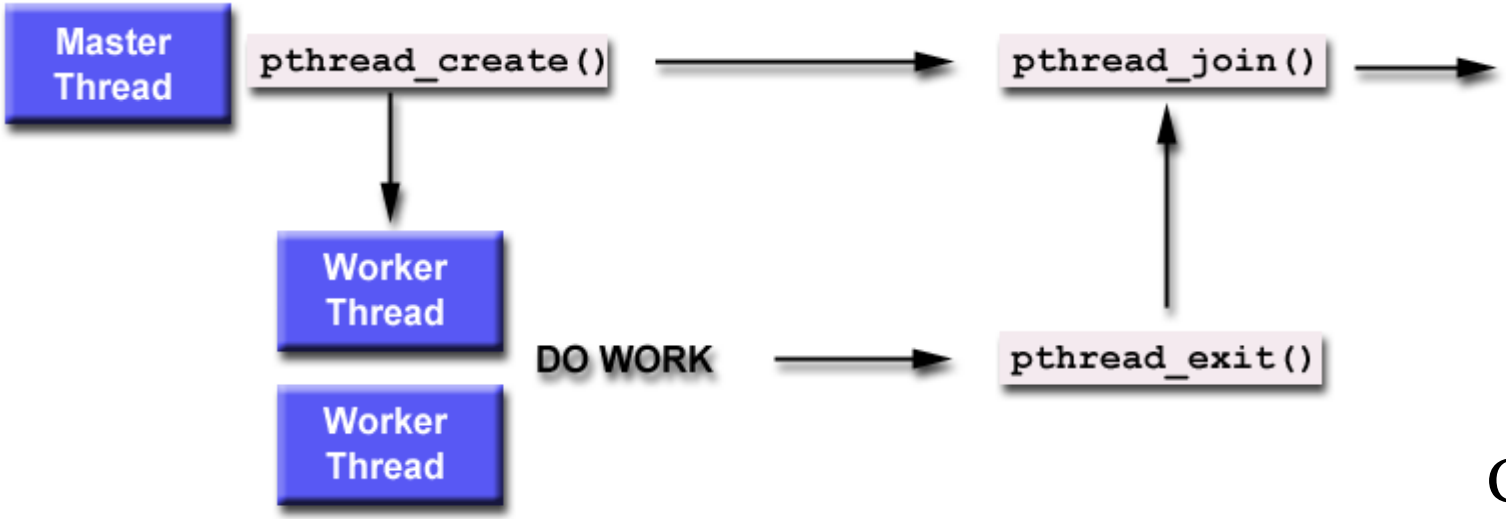
- pthread_exit (status)
- The thread returns from its starting routine (the main routine for the initial thread).
- The thread makes a call to the **pthread_exit** subroutine
- The thread is canceled by another thread via the *pthread_cancel* routine
- The entire process is terminated due to a call to either the exec or exit subroutines.

Joining and Detaching Threads

- pthread_join (threadid,status)
- pthread_detach (threadid,status)
- pthread_attr_setdetachstate (attr,detachstate)
- pthread_attr_getdetachstate (attr,detachstate)

Joining Threads

- **Joining threads**
- "Joining" is one way to accomplish synchronization between threads
- *pthread_join()* subroutine blocks the calling thread until the specified *threadid* thread terminates.
- ***pthread_join (threadid,status)***



Contd..

Detaching Threads

- **Detaching:**
- The **pthread_detach()** routine can be used to explicitly detach a thread even though it was created as joinable.
- There is no converse routine.
- If you know in advance that a thread will never need to join with another thread, consider creating it in a detached state. Some system resources may be able to be freed.

Stock management

- Stack overview

Stack overview

- *pthread_attr_getstacksize (attr, stacksize)*
- *pthread_attr_setstacksize (attr, stacksize)*
- *pthread_attr_getstackaddr (attr, stackaddr)*
- *pthread_attr_setstackaddr (attr, stackaddr)*
- POSIX standard does not dictate the size of a thread's stack.
- Safe and portable programs do not depend upon the default stack limit, but instead, explicitly allocate enough stack for each thread by using the *pthread_attr_setstacksize* routine.

Mutex variables

- Overview
- Creating and Destroying Mutexes
- Locking and Unlocking Mutexes

Mutex overview

- Mutex variables are one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur
- A mutex variable acts like a "lock" protecting access to a shared data resource
- Mutexes can be used to prevent "race" conditions

Creating and Destroying Mutexes

- *pthread_mutex_init (mutex,attr)*
- *pthread_mutex_destroy (mutex)*
- *pthread_mutexattr_init (attr)*
- *pthread_mutexattr_destroy (attr)*
- Pthreads standard defines three optional mutex attributes:
- **Protocol:** Specifies the protocol used to prevent priority inversions for a mutex.
- **Prioceiling**: Specifies the priority ceiling of a mutex.
- **Process-shared:** Specifies the process sharing of a mutex.

Locking and Unlocking Mutexes

- *pthread_mutex_lock (mutex)*
- *pthread_mutex_trylock (mutex)*
- *pthread_mutex_unlock (mutex)*
- *pthread_mutex_lock() routine is used by a thread to acquire a lock on the specified mutex variable*
- *pthread_mutex_trylock() will attempt to lock a mutex*
- *pthread_mutex_unlock()* will unlock a mutex if called by the *owning thread*

Conditional Variables

- Overview
- References

Conditional variables overview

- Condition variables provide yet another way for threads to synchronize
- It allow threads to synchronize based upon the actual value of data
- Condition variable is a way to achieve the synchronization without polling.
- condition variable is always used in conjunction with a mutex lock
- *pthread_cond_init (condition,attr)*
- *pthread_cond_destroy (condition)*
- *pthread_condattr_init (attr)*
- *pthread_condattr_destroy (attr)*

Reference

- **Pthreads Programming by Bradford Nichols,** by *Dick Buttlar, Jacqueline Proulx Farrell*
- Programming with POSIX Threads *by David R. Butenhof*
- Multithreaded Programming with Pthreads *by Bil Lewis, Daniel J. Berg*
- http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html
- http://publib.boulder.ibm.com/iseries/v5r2/ic2924/index.htm?info/apis/rzah4mst.htm
- https://computing.llnl.gov/tutorials/pthreads/#ConditionVariables
-

Queries

- My mail id:: asprakash@au-kbc.org
- My Open Source site, http://electronica.org.in

Thank you