Top of Form

|  |  |
| --- | --- |
| *Author: Blaise Barney, Lawrence Livermore National Laboratory* | UCRL-MI-133316 |

**POSIX Threads Programming**

**Table of Contents**

1. [Abstract](https://computing.llnl.gov/tutorials/pthreads/#Abstract)
2. [Pthreads Overview](https://computing.llnl.gov/tutorials/pthreads/#Overview)
   1. [What is a Thread?](https://computing.llnl.gov/tutorials/pthreads/#Thread)
   2. [What are Pthreads?](https://computing.llnl.gov/tutorials/pthreads/#Pthread)
   3. [Why Pthreads?](https://computing.llnl.gov/tutorials/pthreads/#WhyPthreads)
   4. [Designing Threaded Programs](https://computing.llnl.gov/tutorials/pthreads/#Designing)
3. [The Pthreads API](https://computing.llnl.gov/tutorials/pthreads/#PthreadsAPI)
4. [Compiling Threaded Programs](https://computing.llnl.gov/tutorials/pthreads/#Compiling)
5. [Thread Management](https://computing.llnl.gov/tutorials/pthreads/#Management)
   1. [Creating and Terminating Threads](https://computing.llnl.gov/tutorials/pthreads/#CreatingThreads)
   2. [Passing Arguments to Threads](https://computing.llnl.gov/tutorials/pthreads/#PassingArguments)
   3. [Joining and Detaching Threads](https://computing.llnl.gov/tutorials/pthreads/#Joining)
   4. [Stack Management](https://computing.llnl.gov/tutorials/pthreads/#Stack)
   5. [Miscellaneous Routines](https://computing.llnl.gov/tutorials/pthreads/#Misc)
6. [Exercise 1](https://computing.llnl.gov/tutorials/pthreads/#Exercise1)
7. [Mutex Variables](https://computing.llnl.gov/tutorials/pthreads/#Mutexes)
   1. [Mutex Variables Overview](https://computing.llnl.gov/tutorials/pthreads/#MutexOverview)
   2. [Creating and Destroying Mutexes](https://computing.llnl.gov/tutorials/pthreads/#MutexCreation)
   3. [Locking and Unlocking Mutexes](https://computing.llnl.gov/tutorials/pthreads/#MutexLocking)
8. [Condition Variables](https://computing.llnl.gov/tutorials/pthreads/#ConditionVariables)
   1. [Condition Variables Overview](https://computing.llnl.gov/tutorials/pthreads/#ConVarOverview)
   2. [Creating and Destroying Condition Variables](https://computing.llnl.gov/tutorials/pthreads/#ConVarCreation)
   3. [Waiting and Signaling on Condition Variables](https://computing.llnl.gov/tutorials/pthreads/#ConVarSignal)
9. [Monitoring, Debugging and Performance Analysis Tools for Pthreads](https://computing.llnl.gov/tutorials/pthreads/#Tools)
10. [LLNL Specific Information and Recommendations](https://computing.llnl.gov/tutorials/pthreads/#LLNL)
11. [Topics Not Covered](https://computing.llnl.gov/tutorials/pthreads/#NotCovered)
12. [Exercise 2](https://computing.llnl.gov/tutorials/pthreads/#Exercise2)
13. [References and More Information](https://computing.llnl.gov/tutorials/pthreads/#References)
14. [Appendix A: Pthread Library Routines Reference](https://computing.llnl.gov/tutorials/pthreads/#AppendixA)

|  |
| --- |
| Abstract |

In shared memory multiprocessor architectures, threads can be used to implement parallelism. Historically, hardware vendors have implemented their own proprietary versions of threads, making portability a concern for software developers. For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard. Implementations that adhere to this standard are referred to as POSIX threads, or Pthreads.

The tutorial begins with an introduction to concepts, motivations, and design considerations for using Pthreads. Each of the three major classes of routines in the Pthreads API are then covered: Thread Management, Mutex Variables, and Condition Variables. Example codes are used throughout to demonstrate how to use most of the Pthreads routines needed by a new Pthreads programmer. The tutorial concludes with a discussion of LLNL specifics and how to mix MPI with pthreads. A lab exercise, with numerous example codes (C Language) is also included.

*Level/Prerequisites:* This tutorial is ideal for those who are new to parallel programming with pthreads. A basic understanding of parallel programming in C is required. For those who are unfamiliar with Parallel Programming in general, the material covered in [EC3500: Introduction to Parallel Computing](https://computing.llnl.gov/tutorials/parallel_comp) would be helpful.

|  |
| --- |
| Pthreads Overview |

**What is a Thread?**

* Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system. But what does this mean?
* To the software developer, the concept of a "procedure" that runs independently from its main program may best describe a thread.
* To go one step further, imagine a main program (a.out) that contains a number of procedures. Then imagine all of these procedures being able to be scheduled to run simultaneously and/or independently by the operating system. That would describe a "multi-threaded" program.
* How is this accomplished?
* Before understanding a thread, one first needs to understand a UNIX 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).

|  |  |
| --- | --- |
| Unix Process | Process-thread relationship |
| **UNIX PROCESS** | **THREADS WITHIN A UNIX PROCESS** |

* Threads use and exist within these process resources, yet are able to be scheduled by the operating system and run as independent entities largely because they duplicate only the bare essential resources that enable them to exist as executable code.
* This independent flow of control is accomplished because a thread maintains its own:
  + Stack pointer
  + Registers
  + Scheduling properties (such as policy or priority)
  + Set of pending and blocked signals
  + Thread specific data.
* So, in summary, 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
  + Duplicates only the essential resources it needs to be independently schedulable
  + 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" because most of the overhead has already been accomplished through the creation of its process.
* Because threads within the same process share resources:
  + Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
  + Two pointers having the same value point to the same data.
  + Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.

|  |
| --- |
| Pthreads Overview |

**What are Pthreads?**

* 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 adhering to this standard are referred to as POSIX threads, or Pthreads.
  + Most hardware vendors now offer Pthreads in addition to their proprietary API's.
* The POSIX standard has continued to evolve and undergo revisions, including the Pthreads specification.
* Some useful links:
  + [standards.ieee.org/findstds/standard/1003.1-2008.html](http://standards.ieee.org/findstds/standard/1003.1-2008.html)
  + [www.opengroup.org/austin/papers/posix\_faq.html](http://www.opengroup.org/austin/papers/posix_faq.html)
  + [www.unix.org/version3/ieee\_std.html](http://www.unix.org/version3/ieee_std.html)
* 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 this library may be part of another library, such as libc, in some implementations.

|  |
| --- |
| Pthreads Overview |

**Why Pthreads?**

https://computing.llnl.gov/tutorials/images/arrowBullet.gifLight Weight:

* 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.
* For example, the following table compares timing results for the **fork()** subroutine and the **pthread\_create()** subroutine. Timings reflect 50,000 process/thread creations, were performed with the time utility, and units are in seconds, no optimization flags.

Note: don't expect the sytem and user times to add up to real time, because these are SMP systems with multiple CPUs/cores working on the problem at the same time. At best, these are approximations run on local machines, past and present.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Platform** | **fork()** | | | **pthread\_create()** | | |
| **real** | **user** | **sys** | **real** | **user** | **sys** |
| **Intel 2.6 GHz Xeon E5-2670 (16 cores/node)** | 8.1 | 0.1 | 2.9 | 0.9 | 0.2 | 0.3 |
| **Intel 2.8 GHz Xeon 5660 (12 cores/node)** | 4.4 | 0.4 | 4.3 | 0.7 | 0.2 | 0.5 |
| **AMD 2.3 GHz Opteron (16 cores/node)** | 12.5 | 1.0 | 12.5 | 1.2 | 0.2 | 1.3 |
| **AMD 2.4 GHz Opteron (8 cores/node)** | 17.6 | 2.2 | 15.7 | 1.4 | 0.3 | 1.3 |
| **IBM 4.0 GHz POWER6 (8 cpus/node)** | 9.5 | 0.6 | 8.8 | 1.6 | 0.1 | 0.4 |
| **IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)** | 64.2 | 30.7 | 27.6 | 1.7 | 0.6 | 1.1 |
| **IBM 1.5 GHz POWER4 (8 cpus/node)** | 104.5 | 48.6 | 47.2 | 2.1 | 1.0 | 1.5 |
| **INTEL 2.4 GHz Xeon (2 cpus/node)** | 54.9 | 1.5 | 20.8 | 1.6 | 0.7 | 0.9 |
| **INTEL 1.4 GHz Itanium2 (4 cpus/node)** | 54.5 | 1.1 | 22.2 | 2.0 | 1.2 | 0.6 |

View source code  [fork\_vs\_thread.txt](https://computing.llnl.gov/tutorials/pthreads/fork_vs_thread.txt)

https://computing.llnl.gov/tutorials/images/arrowBullet.gifEfficient Communications/Data Exchange:

* The primary motivation for considering the use of Pthreads in a high performance computing environment is to achieve optimum performance. In particular, if an application is using MPI for on-node communications, there is a potential that performance could be improved by using Pthreads instead.
* MPI libraries usually implement on-node task communication via shared memory, which involves at least one memory copy operation (process to process).
* For Pthreads there is no intermediate memory copy required because threads share the same address space within a single process. There is no data transfer, per se. It can be as efficient as simply passing a pointer.
* In the worst case scenario, Pthread communications become more of a cache-to-CPU or memory-to-CPU bandwidth issue. These speeds are much higher than MPI shared memory communications.
* For example: some local comparisons, past and present, are shown below:

|  |  |  |
| --- | --- | --- |
| **Platform** | **MPI Shared Memory Bandwidth (GB/sec)** | **Pthreads Worst Case Memory-to-CPU Bandwidth  (GB/sec)** |
| **Intel 2.6 GHz Xeon E5-2670** | 4.5 | 51.2 |
| **Intel 2.8 GHz Xeon 5660** | 5.6 | 32 |
| **AMD 2.3 GHz Opteron** | 1.8 | 5.3 |
| **AMD 2.4 GHz Opteron** | 1.2 | 5.3 |
| **IBM 1.9 GHz POWER5 p5-575** | 4.1 | 16 |
| **IBM 1.5 GHz POWER4** | 2.1 | 4 |
| **Intel 2.4 GHz Xeon** | 0.3 | 4.3 |
| **Intel 1.4 GHz Itanium 2** | 1.8 | 6.4 |

https://computing.llnl.gov/tutorials/images/arrowBullet.gifOther Common Reasons:

* 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, CPU intensive work can be performed by other threads.
  + Priority/real-time scheduling: tasks which are more important can be scheduled to supersede or interrupt lower priority tasks.
  + Asynchronous event handling: tasks which 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.
* A perfect example is the typical web browser, where many interleaved tasks can be happening at the same time, and where tasks can vary in priority.
* Another good example is a modern operating system, which makes extensive use of threads. A screenshot of the MS Windows OS and applications using threads is shown below.

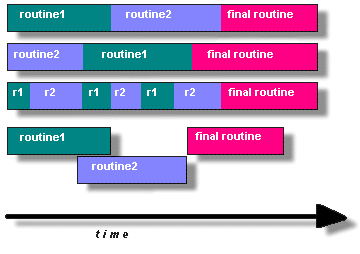
|  |
| --- |
| [https://computing.llnl.gov/tutorials/pthreads/images/resourcemonitor.600pix.jpg](https://computing.llnl.gov/tutorials/pthreads/images/resourcemonitor.jpg) *Click for larger image* |

|  |
| --- |
| Pthreads Overview |

**Designing Threaded Programs**

https://computing.llnl.gov/tutorials/images/arrowBullet.gifParallel Programming:

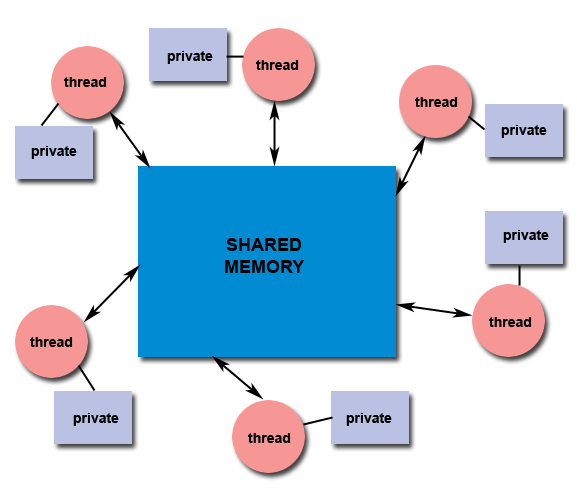
* On modern, multi-core machines, pthreads are ideally suited for parallel programming, and whatever applies to parallel programming in general, applies to parallel pthreads programs.
* There are many considerations for designing parallel programs, such as:
  + What type of parallel programming model to use?
  + Problem partitioning
  + Load balancing
  + Communications
  + Data dependencies
  + Synchronization and race conditions
  + Memory issues
  + I/O issues
  + Program complexity
  + Programmer effort/costs/time
  + ...
* Covering these topics is beyond the scope of this tutorial, however interested readers can obtain a quick overview in the [Introduction to Parallel Computing](https://computing.llnl.gov/tutorials/parallel_comp) tutorial.
* In general though, 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. For example, if routine1 and routine2 can be interchanged, interleaved and/or overlapped in real time, they are candidates for threading.



* Programs having the following characteristics may be well suited for pthreads:
  + Work that can be executed, or data that can be operated on, by multiple tasks simultaneously:
  + Block for potentially long I/O waits
  + Use many CPU cycles in some places but not others
  + Must respond to asynchronous events
  + Some work is more important than other work (priority interrupts)
* Several common models for threaded programs exist:
  + ***Manager/worker:*** a single thread, the *manager* assigns work to other threads, the *workers*. Typically, the manager handles all input and parcels out work to the other tasks. At least two forms of the manager/worker model are common: static worker pool and dynamic worker pool.
  + ***Pipeline:*** a task is broken into a series of suboperations, each of which is handled in series, but concurrently, by a different thread. An automobile assembly line best describes this model.
  + ***Peer:*** similar to the manager/worker model, but after the main thread creates other threads, it participates in the work.

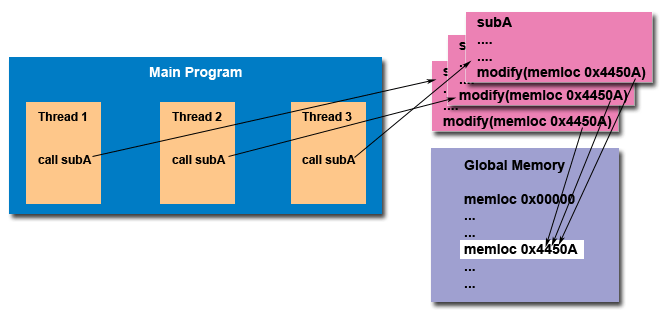
https://computing.llnl.gov/tutorials/images/arrowBullet.gifShared Memory Model:

* 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.



https://computing.llnl.gov/tutorials/images/arrowBullet.gifThread-safeness:

* Thread-safeness: in a nutshell, refers an 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:
  + This library routine accesses/modifies a global structure or location in memory.
  + As each thread calls this routine it is possible that they may try to modify this global structure/memory location at the same time.
  + If the routine does not employ some sort of synchronization constructs to prevent data corruption, then it is not thread-safe.



* The implication to users of external library routines is that if you aren't 100% certain the routine is thread-safe, then you take your chances with problems that could arise.
* Recommendation: Be careful if your application uses libraries or other objects that don't explicitly guarantee thread-safeness. When in doubt, assume that they are not thread-safe until proven otherwise. This can be done by "serializing" the calls to the uncertain routine, etc.

https://computing.llnl.gov/tutorials/images/arrowBullet.gifThread Limits:

* Although the Pthreads API is an ANSI/IEEE standard, implementations can, and usually do, vary in ways not specified by the standard.
* Because of this, a program that runs fine on one platform, may fail or produce wrong results on another platform.
* For example, the maximum number of threads permitted, and the default thread stack size are two important limits to consider when designing your program.
* Several thread limits are discussed in more detail later in this tutorial.

|  |
| --- |
| The Pthreads API |

* The original Pthreads API was defined in the ANSI/IEEE POSIX 1003.1 - 1995 standard. The POSIX standard has continued to evolve and undergo revisions, including the Pthreads specification.
* Copies of the standard can be purchased from IEEE or downloaded for free from other sites online.
* The subroutines which comprise the Pthreads API can be informally grouped into four major groups:
  1. **Thread management:** Routines that work directly on threads - creating, detaching, joining, etc. They also include functions to set/query thread attributes (joinable, scheduling etc.)
  2. **Mutexes:** Routines that deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
  3. **Condition variables:** Routines that address communications between threads that share a mutex. Based upon programmer specified conditions. This group includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.
  4. **Synchronization:** Routines that manage read/write locks and barriers.
* Naming conventions: All identifiers in the threads library begin with **pthread\_**. Some examples are shown below.

|  |  |
| --- | --- |
| **Routine Prefix** | **Functional Group** |
| **pthread\_** | Threads themselves and miscellaneous subroutines |
| **pthread\_attr\_** | Thread attributes objects |
| **pthread\_mutex\_** | Mutexes |
| **pthread\_mutexattr\_** | Mutex attributes objects. |
| **pthread\_cond\_** | Condition variables |
| **pthread\_condattr\_** | Condition attributes objects |
| **pthread\_key\_** | Thread-specific data keys |
| **pthread\_rwlock\_** | Read/write locks |
| **pthread\_barrier\_** | Synchronization barriers |

* The concept of opaque objects pervades the design of the API. The basic calls work to create or modify opaque objects - the opaque objects can be modified by calls to attribute functions, which deal with opaque attributes.
* The Pthreads API contains around 100 subroutines. This tutorial will focus on a subset of these - specifically, those which are most likely to be immediately useful to the beginning Pthreads programmer.
* For portability, the pthread.h header file should be included in each source file using the Pthreads library.
* The current POSIX standard is defined only for the C language. Fortran programmers can use wrappers around C function calls. Some Fortran compilers may provide a Fortran pthreads API.
* A number of excellent books about Pthreads are available. Several of these are listed in the [References](https://computing.llnl.gov/tutorials/pthreads/#References) section of this tutorial.

|  |
| --- |
| Compiling Threaded Programs |

* Several examples of compile commands used for pthreads codes are listed in the table below.

|  |  |  |
| --- | --- | --- |
| **Compiler / Platform** | **Compiler Command** | **Description** |
| INTEL Linux | **icc -pthread** | C |
| **icpc -pthread** | C++ |
| PGI Linux | **pgcc -lpthread** | C |
| **pgCC -lpthread** | C++ |
| GNU Linux, Blue Gene | **gcc -pthread** | GNU C |
| **g++ -pthread** | GNU C++ |
| IBM  Blue Gene | **bgxlc\_r  /  bgcc\_r** | C (ANSI  /  non-ANSI) |
| **bgxlC\_r, bgxlc++\_r** | C++ |

|  |
| --- |
| Thread Management |

**Creating and Terminating Threads**

https://computing.llnl.gov/tutorials/images/arrowBullet.gifRoutines:

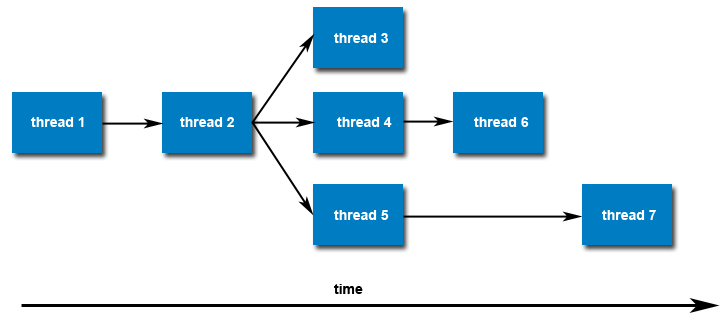
|  |
| --- |
| [**pthread\_create**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_create.txt) **(thread,attr,start\_routine,arg)**  [**pthread\_exit**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_exit.txt) **(status)**  [**pthread\_cancel**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_cancel.txt) **(thread)**  [**pthread\_attr\_init**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_init.txt) **(attr)**  [**pthread\_attr\_destroy**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_destroy.txt) **(attr)** |

https://computing.llnl.gov/tutorials/images/arrowBullet.gifCreating Threads:

* Initially, your 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. This routine can be called any number of times from anywhere within your code.
* pthread\_create arguments:
  + thread: An opaque, unique identifier for the new thread returned by the subroutine.
  + attr: An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
  + start\_routine: the C routine that the thread will execute once it is created.
  + arg: A single argument that may be passed to *start\_routine*. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.
* The maximum number of threads that may be created by a process is implementation dependent. Programs that attempt to exceed the limit can fail or produce wrong results.
* Querying and setting your implementation's thread limit - Linux example shown. Demonstrates querying the default (soft) limits and then setting the maximum number of processes (including threads) to the hard limit. Then verifying that the limit has been overridden.

|  |  |
| --- | --- |
| **bash / ksh / sh** | **tcsh / csh** |
| $ ulimit -a  core file size (blocks, -c) 16  data seg size (kbytes, -d) unlimited  scheduling priority (-e) 0  file size (blocks, -f) unlimited  pending signals (-i) 255956  max locked memory (kbytes, -l) 64  max memory size (kbytes, -m) unlimited  open files (-n) 1024  pipe size (512 bytes, -p) 8  POSIX message queues (bytes, -q) 819200  real-time priority (-r) 0  stack size (kbytes, -s) unlimited  cpu time (seconds, -t) unlimited  max user processes (-u) 1024  virtual memory (kbytes, -v) unlimited  file locks (-x) unlimited  $ ulimit -Hu  7168  $ ulimit -u 7168  $ ulimit -a  core file size (blocks, -c) 16  data seg size (kbytes, -d) unlimited  scheduling priority (-e) 0  file size (blocks, -f) unlimited  pending signals (-i) 255956  max locked memory (kbytes, -l) 64  max memory size (kbytes, -m) unlimited  open files (-n) 1024  pipe size (512 bytes, -p) 8  POSIX message queues (bytes, -q) 819200  real-time priority (-r) 0  stack size (kbytes, -s) unlimited  cpu time (seconds, -t) unlimited  max user processes (-u) 7168  virtual memory (kbytes, -v) unlimited  file locks (-x) unlimited | % limit  cputime unlimited  filesize unlimited  datasize unlimited  stacksize unlimited  coredumpsize 16 kbytes  memoryuse unlimited  vmemoryuse unlimited  descriptors 1024  memorylocked 64 kbytes  maxproc 1024  % limit maxproc unlimited  % limit  cputime unlimited  filesize unlimited  datasize unlimited  stacksize unlimited  coredumpsize 16 kbytes  memoryuse unlimited  vmemoryuse unlimited  descriptors 1024  memorylocked 64 kbytes  maxproc 7168 |

* Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.



https://computing.llnl.gov/tutorials/images/arrowBullet.gifThread Attributes:

* By default, a thread is created with certain attributes. Some of these attributes can be changed by the programmer via the thread attribute object.
* pthread\_attr\_init and pthread\_attr\_destroy are used to initialize/destroy the thread attribute object.
* Other routines are then used to query/set specific attributes in the thread attribute object. Attributes include:
  + Detached or joinable state
  + Scheduling inheritance
  + Scheduling policy
  + Scheduling parameters
  + Scheduling contention scope
  + Stack size
  + Stack address
  + Stack guard (overflow) size
* Some of these attributes will be discussed later.

https://computing.llnl.gov/tutorials/images/arrowBullet.gifThread Binding and Scheduling:

|  |  |  |
| --- | --- | --- |
|  | https://computing.llnl.gov/tutorials/images/question2.gif | Question: After a thread has been created, how do you know a)when it will be scheduled to run by the operating system, and b)which processor/core it will run on? |

Bottom of Form

* The Pthreads API provides several routines that may be used to specify how threads are scheduled for execution. For example, threads can be scheduled to run FIFO (first-in first-out), RR (round-robin) or OTHER (operating system determines). It also provides the ability to set a thread's scheduling priority value.
* These topics are not covered here, however a good overview of "how things work" under Linux can be found in the [sched\_setscheduler](https://computing.llnl.gov/tutorials/pthreads/man/sched_setscheduler.txt) man page.
* The Pthreads API does not provide routines for binding threads to specific cpus/cores. However, local implementations may include this functionality - such as providing the non-standard [pthread\_setaffinity\_np](https://computing.llnl.gov/tutorials/pthreads/man/pthread_setaffinity_np.txt) routine. Note that "\_np" in the name stands for "non-portable".
* Also, the local operating system may provide a way to do this. For example, Linux provides the [sched\_setaffinity](https://computing.llnl.gov/tutorials/pthreads/man/sched_setaffinity.txt) routine.

https://computing.llnl.gov/tutorials/images/arrowBullet.gifTerminating Threads & pthread\_exit():

* There are several ways in which a thread may be terminated:
  + The thread returns normally from its starting routine. Its work is done.
  + The thread makes a call to the pthread\_exit subroutine - whether its work is done or not.
  + The thread is canceled by another thread via the pthread\_cancel routine.
  + The entire process is terminated due to making a call to either the exec() or exit()
  + If main() finishes first, without calling pthread\_exit explicitly itself
* The pthread\_exit() routine allows the programmer to specify an optional termination *status* parameter. This optional parameter is typically returned to threads "joining" the terminated thread (covered later).
* In subroutines that execute to completion normally, you can often dispense with calling pthread\_exit() - unless, of course, you want to pass the optional status code back.
* Cleanup: the pthread\_exit() routine does not close files; any files opened inside the thread will remain open after the thread is terminated.
* **Discussion on calling pthread\_exit() from main():**
  + There is a definite problem if main() finishes before the threads it spawned if you don't call pthread\_exit() explicitly. All of the threads it created will terminate because main() is done and no longer exists to support the threads.
  + By having main() explicitly call pthread\_exit() as the last thing it does, main() will block and be kept alive to support the threads it created until they are done.

**Example: Pthread Creation and Termination**

* This simple example code creates 5 threads with the pthread\_create() routine. Each thread prints a "Hello World!" message, and then terminates with a call to pthread\_exit().

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | https://computing.llnl.gov/tutorials/images/page01.gif | | **Pthread Creation and Termination Example** | |  | | | | 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 |  | #include <pthread.h>  #include <stdio.h>  #define NUM\_THREADS 5  void \*PrintHello(void \*threadid)  {  long tid;  tid = (long)threadid;  printf("Hello World! It's me, thread #%ld!\n", tid);  pthread\_exit(NULL);  }  int main (int argc, char \*argv[])  {  pthread\_t threads[NUM\_THREADS];  int rc;  long t;  for(t=0; t<NUM\_THREADS; t++){  printf("In main: creating thread %ld\n", t);  rc = pthread\_create(&threads[t], NULL, PrintHello, (void \*)t);  if (rc){  printf("ERROR; return code from pthread\_create() is %d\n", rc);  exit(-1);  }  }  /\* Last thing that main() should do \*/  pthread\_exit(NULL);  }  [https://computing.llnl.gov/tutorials/images/output1.gif](https://computing.llnl.gov/tutorials/pthreads/samples/hello.out)  In main: creating thread 0  In main: creating thread 1  Hello World! It's me, thread #0!  In main: creating thread 2  Hello World! It's me, thread #1!  Hello World! It's me, thread #2!  In main: creating thread 3  In main: creating thread 4  Hello World! It's me, thread #3!  Hello World! It's me, thread #4! | |

|  |
| --- |
| Thread Management |

**Passing Arguments to Threads**

* The pthread\_create() routine permits the programmer to pass one argument to the thread start routine. For cases where multiple arguments must be passed, this limitation is easily overcome by creating a structure which contains all of the arguments, and then passing a pointer to that structure in the pthread\_create() routine.
* All arguments must be passed by reference and cast to (void \*).

|  |  |  |
| --- | --- | --- |
|  | https://computing.llnl.gov/tutorials/images/question2.gif | Question: How can you safely pass data to newly created threads, given their non-deterministic start-up and scheduling? |

|  |
| --- |
| https://computing.llnl.gov/tutorials/images/page01.gif**Example 1 - Thread Argument Passing**  This code fragment demonstrates how to pass a simple integer to each thread. The calling thread uses a unique data structure for each thread, insuring that each thread's argument remains intact throughout the program.  long taskids[NUM\_THREADS];  for(t=0; t<NUM\_THREADS; t++)  {  taskids[t] = t;  printf("Creating thread %ld\n", t);  rc = pthread\_create(&threads[t], NULL, PrintHello, (void \*) taskids[t]);  ...  }  [View sample output](https://computing.llnl.gov/tutorials/pthreads/samples/hello_arg1.out)  Creating thread 0  Creating thread 1  Creating thread 2  Creating thread 3  Creating thread 4  Creating thread 5  Creating thread 6  Creating thread 7  Thread 0: English: Hello World!  Thread 1: French: Bonjour, le monde!  Thread 2: Spanish: Hola al mundo  Thread 3: Klingon: Nuq neH!  Thread 4: German: Guten Tag, Welt!  Thread 5: Russian: Zdravstvytye, mir!  Thread 6: Japan: Sekai e konnichiwa!  Thread 7: Latin: Orbis, te saluto! |

|  |
| --- |
| https://computing.llnl.gov/tutorials/images/page01.gif**Example 2 - Thread Argument Passing**  This example shows how to setup/pass multiple arguments via a structure. Each thread receives a unique instance of the structure.  struct thread\_data{  int thread\_id;  int sum;  char \*message;  };  struct thread\_data thread\_data\_array[NUM\_THREADS];  void \*PrintHello(void \*threadarg)  {  struct thread\_data \*my\_data;  ...  my\_data = (struct thread\_data \*) threadarg;  taskid = my\_data->thread\_id;  sum = my\_data->sum;  hello\_msg = my\_data->message;  ...  }  int main (int argc, char \*argv[])  {  ...  thread\_data\_array[t].thread\_id = t;  thread\_data\_array[t].sum = sum;  thread\_data\_array[t].message = messages[t];  rc = pthread\_create(&threads[t], NULL, PrintHello,  (void \*) &thread\_data\_array[t]);  ...  }  [View sample output](https://computing.llnl.gov/tutorials/pthreads/samples/hello_arg2.out)  Creating thread 0  Creating thread 1  Creating thread 2  Creating thread 3  Creating thread 4  Creating thread 5  Creating thread 6  Creating thread 7  Thread 0: English: Hello World! Sum=0  Thread 1: French: Bonjour, le monde! Sum=1  Thread 2: Spanish: Hola al mundo Sum=3  Thread 3: Klingon: Nuq neH! Sum=6  Thread 4: German: Guten Tag, Welt! Sum=10  Thread 5: Russian: Zdravstvytye, mir! Sum=15  Thread 6: Japan: Sekai e konnichiwa! Sum=21  Thread 7: Latin: Orbis, te saluto! Sum=28 |

|  |
| --- |
| https://computing.llnl.gov/tutorials/images/page01.gif**Example 3 - Thread Argument Passing (Incorrect)**  This example performs argument passing incorrectly. It passes the *address* of variable t, which is shared memory space and visible to all threads. As the loop iterates, the value of this memory location changes, possibly before the created threads can access it.  int rc;  long t;  for(t=0; t<NUM\_THREADS; t++)  {  printf("Creating thread %ld\n", t);  rc = pthread\_create(&threads[t], NULL, PrintHello, (void \*) &t);  ...  }  [View sample output](https://computing.llnl.gov/tutorials/pthreads/samples/hello_arg3.out)  Creating thread 0  Creating thread 1  Creating thread 2  Creating thread 3  Creating thread 4  Creating thread 5  Creating thread 6  Creating thread 7  Hello from thread 140737488348392  Hello from thread 140737488348392  Hello from thread 140737488348392  Hello from thread 140737488348392  Hello from thread 140737488348392  Hello from thread 140737488348392  Hello from thread 140737488348392  Hello from thread 140737488348392 |

|  |
| --- |
| Thread Management |

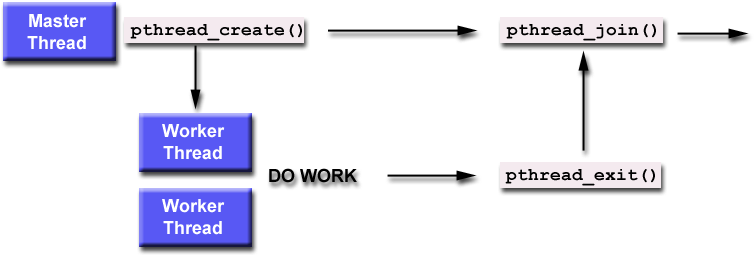
**Joining and Detaching Threads**

https://computing.llnl.gov/tutorials/images/arrowBullet.gifRoutines:

|  |
| --- |
| [**pthread\_join**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_join.txt) **(threadid,status)**  [**pthread\_detach**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_detach.txt) **(threadid)**  [**pthread\_attr\_setdetachstate**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_setdetachstate.txt) **(attr,detachstate)**  [**pthread\_attr\_getdetachstate**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_getdetachstate.txt) **(attr,detachstate)** |

https://computing.llnl.gov/tutorials/images/arrowBullet.gifJoining:

* "Joining" is one way to accomplish synchronization between threads. For example:



* The pthread\_join() subroutine blocks the calling thread until the specified threadid thread terminates.
* The programmer is able to obtain the target thread's termination return status if it was specified in the target thread's call to pthread\_exit().
* A joining thread can match one pthread\_join() call. It is a logical error to attempt multiple joins on the same thread.
* Two other synchronization methods, mutexes and condition variables, will be discussed later.

https://computing.llnl.gov/tutorials/images/arrowBullet.gifJoinable or Not?

* When a thread is created, one of its attributes defines whether it is joinable or detached. Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.
* The final draft of the POSIX standard specifies that threads should be created as joinable.
* To explicitly create a thread as joinable or detached, the attr argument in the pthread\_create() routine is used. The typical 4 step process is:
  1. Declare a pthread attribute variable of the pthread\_attr\_t data type
  2. Initialize the attribute variable with pthread\_attr\_init()
  3. Set the attribute detached status with pthread\_attr\_setdetachstate()
  4. When done, free library resources used by the attribute with pthread\_attr\_destroy()

https://computing.llnl.gov/tutorials/images/arrowBullet.gifDetaching:

* The pthread\_detach() routine can be used to explicitly detach a thread even though it was created as joinable.
* There is no converse routine.

https://computing.llnl.gov/tutorials/images/arrowBullet.gifRecommendations:

* If a thread requires joining, consider explicitly creating it as joinable. This provides portability as not all implementations may create threads as joinable by default.
* 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.

**Example: Pthread Joining**

* This example demonstrates how to "wait" for thread completions by using the Pthread join routine.
* Since some implementations of Pthreads may not create threads in a joinable state, the threads in this example are explicitly created in a joinable state so that they can be joined later.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | https://computing.llnl.gov/tutorials/images/page01.gif | | **Pthread Joining Example** | |  | | | | 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 |  | #include <pthread.h>  #include <stdio.h>  #include <stdlib.h>  #include <math.h>  #define NUM\_THREADS 4  void \*BusyWork(void \*t)  {  int i;  long tid;  double result=0.0;  tid = (long)t;  printf("Thread %ld starting...\n",tid);  for (i=0; i<1000000; i++)  {  result = result + sin(i) \* tan(i);  }  printf("Thread %ld done. Result = %e\n",tid, result);  pthread\_exit((void\*) t);  }  int main (int argc, char \*argv[])  {  pthread\_t thread[NUM\_THREADS];  pthread\_attr\_t attr;  int rc;  long t;  void \*status;  /\* Initialize and set thread detached attribute \*/  pthread\_attr\_init(&attr);  pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_JOINABLE);  for(t=0; t<NUM\_THREADS; t++) {  printf("Main: creating thread %ld\n", t);  rc = pthread\_create(&thread[t], &attr, BusyWork, (void \*)t);  if (rc) {  printf("ERROR; return code from pthread\_create() is %d\n", rc);  exit(-1);  }  }  /\* Free attribute and wait for the other threads \*/  pthread\_attr\_destroy(&attr);  for(t=0; t<NUM\_THREADS; t++) {  rc = pthread\_join(thread[t], &status);  if (rc) {  printf("ERROR; return code from pthread\_join() is %d\n", rc);  exit(-1);  }  printf("Main: completed join with thread %ld having a status  of %ld\n",t,(long)status);  }    printf("Main: program completed. Exiting.\n");  pthread\_exit(NULL);  }  [https://computing.llnl.gov/tutorials/images/output1.gif](https://computing.llnl.gov/tutorials/pthreads/samples/join.out)  Main: creating thread 0  Main: creating thread 1  Thread 0 starting...  Main: creating thread 2  Thread 1 starting...  Main: creating thread 3  Thread 2 starting...  Thread 3 starting...  Thread 1 done. Result = -3.153838e+06  Thread 0 done. Result = -3.153838e+06  Main: completed join with thread 0 having a status of 0  Main: completed join with thread 1 having a status of 1  Thread 3 done. Result = -3.153838e+06  Thread 2 done. Result = -3.153838e+06  Main: completed join with thread 2 having a status of 2  Main: completed join with thread 3 having a status of 3  Main: program completed. Exiting. | |

|  |
| --- |
| Thread Management |

**Stack Management**

https://computing.llnl.gov/tutorials/images/arrowBullet.gifRoutines:

|  |
| --- |
| [**pthread\_attr\_getstacksize**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_getstacksize.txt) **(attr, stacksize)**  [**pthread\_attr\_setstacksize**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_setstacksize.txt) **(attr, stacksize)**  [**pthread\_attr\_getstackaddr**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_getstackaddr.txt) **(attr, stackaddr)**  [**pthread\_attr\_setstackaddr**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_setstackaddr.txt) **(attr, stackaddr)** |

https://computing.llnl.gov/tutorials/images/arrowBullet.gifPreventing Stack Problems:

* The POSIX standard does not dictate the size of a thread's stack. This is implementation dependent and varies.
* Exceeding the default stack limit is often very easy to do, with the usual results: program termination and/or corrupted data.
* 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.
* The pthread\_attr\_getstackaddr and pthread\_attr\_setstackaddr routines can be used by applications in an environment where the stack for a thread must be placed in some particular region of memory.

https://computing.llnl.gov/tutorials/images/arrowBullet.gifSome Practical Examples at LC:

* Default thread stack size varies greatly. The maximum size that can be obtained also varies greatly, and may depend upon the number of threads per node.
* Both past and present architectures are shown to demonstrate the wide variation in default thread stack size.

|  |  |  |  |
| --- | --- | --- | --- |
| **Node Architecture** | **#CPUs** | **Memory (GB)** | **Default Size (bytes)** |
| Intel Xeon E5-2670 | 16 | 32 | 2,097,152 |
| Intel Xeon 5660 | 12 | 24 | 2,097,152 |
| AMD Opteron | 8 | 16 | 2,097,152 |
| Intel IA64 | 4 | 8 | 33,554,432 |
| Intel IA32 | 2 | 4 | 2,097,152 |
| IBM Power5 | 8 | 32 | 196,608 |
| IBM Power4 | 8 | 16 | 196,608 |
| IBM Power3 | 16 | 16 | 98,304 |

**Example: Stack Management**

* This example demonstrates how to query and set a thread's stack size.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | https://computing.llnl.gov/tutorials/images/page01.gif | | **Stack Management Example** | |  | | | | 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 |  | #include <pthread.h>  #include <stdio.h>  #define NTHREADS 4  #define N 1000  #define MEGEXTRA 1000000    pthread\_attr\_t attr;    void \*dowork(void \*threadid)  {  double A[N][N];  int i,j;  long tid;  size\_t mystacksize;  tid = (long)threadid;  pthread\_attr\_getstacksize (&attr, &mystacksize);  printf("Thread %ld: stack size = %li bytes \n", tid, mystacksize);  for (i=0; i<N; i++)  for (j=0; j<N; j++)  A[i][j] = ((i\*j)/3.452) + (N-i);  pthread\_exit(NULL);  }    int main(int argc, char \*argv[])  {  pthread\_t threads[NTHREADS];  size\_t stacksize;  int rc;  long t;    pthread\_attr\_init(&attr);  pthread\_attr\_getstacksize (&attr, &stacksize);  printf("Default stack size = %li\n", stacksize);  stacksize = sizeof(double)\*N\*N+MEGEXTRA;  printf("Amount of stack needed per thread = %li\n",stacksize);  pthread\_attr\_setstacksize (&attr, stacksize);  printf("Creating threads with stack size = %li bytes\n",stacksize);  for(t=0; t<NTHREADS; t++){  rc = pthread\_create(&threads[t], &attr, dowork, (void \*)t);  if (rc){  printf("ERROR; return code from pthread\_create() is %d\n", rc);  exit(-1);  }  }  printf("Created %ld threads.\n", t);  pthread\_exit(NULL);  } | |

|  |
| --- |
| Thread Management |

**Miscellaneous Routines**

|  |
| --- |
| [**pthread\_self**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_self.txt) **()**  [**pthread\_equal**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_equal.txt) **(thread1,thread2)** |

* pthread\_self returns the unique, system assigned thread ID of the calling thread.
* pthread\_equal compares two thread IDs. If the two IDs are different 0 is returned, otherwise a non-zero value is returned.
* Note that for both of these routines, the thread identifier objects are opaque and can not be easily inspected. Because thread IDs are opaque objects, the C language equivalence operator == should not be used to compare two thread IDs against each other, or to compare a single thread ID against another value.

|  |
| --- |
| [**pthread\_once**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_once.txt) **(once\_control, init\_routine)** |

* pthread\_once executes the init\_routine exactly once in a process. The first call to this routine by any thread in the process executes the given init\_routine, without parameters. Any subsequent call will have no effect.
* The init\_routine routine is typically an initialization routine.
* The once\_control parameter is a synchronization control structure that requires initialization prior to calling pthread\_once. For example:

pthread\_once\_t once\_control = PTHREAD\_ONCE\_INIT;

|  |
| --- |
| Pthread Exercise 1 |

**Getting Started and Thread Management Routines**

|  |
| --- |
| **Overview:**   * **Login to an LC cluster using your workshop username and OTP token** * **Copy the exercise files to your home directory** * **Familiarize yourself with LC's Pthreads environment** * **Write a simple "Hello World" Pthreads program** * **Successfully compile your program** * **Successfully run your program - several different ways** * **Review, compile, run and/or debug some related Pthreads programs (provided)**   **https://computing.llnl.gov/tutorials/images/point02.jpg**[**GO TO THE EXERCISE HERE**](https://computing.llnl.gov/tutorials/pthreads/exercise.html#Exercise1) |

|  |
| --- |
| Mutex Variables |

**Overview**

* Mutex is an abbreviation for "mutual exclusion". 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. The basic concept of a mutex as used in Pthreads is that only one thread can lock (or own) a mutex variable at any given time. Thus, even if several threads try to lock a mutex only one thread will be successful. No other thread can own that mutex until the owning thread unlocks that mutex. Threads must "take turns" accessing protected data.
* Mutexes can be used to prevent "race" conditions. An example of a race condition involving a bank transaction is shown below:

|  |  |  |
| --- | --- | --- |
| **Thread 1** | **Thread 2** | **Balance** |
| **Read balance: $1000** |  | **$1000** |
|  | **Read balance: $1000** | **$1000** |
|  | **Deposit $200** | **$1000** |
| **Deposit $200** |  | **$1000** |
| **Update balance $1000+$200** |  | **$1200** |
|  | **Update balance $1000+$200** | **$1200** |

* In the above example, a mutex should be used to lock the "Balance" while a thread is using this shared data resource.
* Very often the action performed by a thread owning a mutex is the updating of global variables. This is a safe way to ensure that when several threads update the same variable, the final value is the same as what it would be if only one thread performed the update. The variables being updated belong to a "critical section".
* A typical sequence in the use of a mutex is as follows:
  + Create and initialize a mutex variable
  + Several threads attempt to lock the mutex
  + Only one succeeds and that thread owns the mutex
  + The owner thread performs some set of actions
  + The owner unlocks the mutex
  + Another thread acquires the mutex and repeats the process
  + Finally the mutex is destroyed
* When several threads compete for a mutex, the losers block at that call - an unblocking call is available with "trylock" instead of the "lock" call.
* When protecting shared data, it is the programmer's responsibility to make sure every thread that needs to use a mutex does so. For example, if 4 threads are updating the same data, but only one uses a mutex, the data can still be corrupted.

|  |
| --- |
| Mutex Variables |

**Creating and Destroying Mutexes**

https://computing.llnl.gov/tutorials/images/arrowBullet.gifRoutines:

|  |
| --- |
| [**pthread\_mutex\_init**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutex_init.txt) **(mutex,attr)**  [**pthread\_mutex\_destroy**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutex_destroy.txt) **(mutex)**  [**pthread\_mutexattr\_init**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutexattr_init.txt) **(attr)**  [**pthread\_mutexattr\_destroy**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutexattr_destroy.txt) **(attr)** |

https://computing.llnl.gov/tutorials/images/arrowBullet.gifUsage:

* Mutex variables must be declared with type pthread\_mutex\_t, and must be initialized before they can be used. There are two ways to initialize a mutex variable:
  1. Statically, when it is declared. For example:   
     pthread\_mutex\_t mymutex = PTHREAD\_MUTEX\_INITIALIZER;
  2. Dynamically, with the pthread\_mutex\_init() routine. This method permits setting mutex object attributes, *attr*.

The mutex is initially unlocked.

* The *attr* object is used to establish properties for the mutex object, and must be of type pthread\_mutexattr\_t if used (may be specified as NULL to accept defaults). The Pthreads standard defines three optional mutex attributes:
  1. Protocol: Specifies the protocol used to prevent priority inversions for a mutex.
  2. Prioceiling: Specifies the priority ceiling of a mutex.
  3. Process-shared: Specifies the process sharing of a mutex.

Note that not all implementations may provide the three optional mutex attributes.

* The pthread\_mutexattr\_init() and pthread\_mutexattr\_destroy() routines are used to create and destroy mutex attribute objects respectively.
* pthread\_mutex\_destroy() should be used to free a mutex object which is no longer needed.

|  |
| --- |
| Mutex Variables |

**Locking and Unlocking Mutexes**

https://computing.llnl.gov/tutorials/images/arrowBullet.gifRoutines:

|  |
| --- |
| [**pthread\_mutex\_lock**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutex_lock.txt) **(mutex)**  [**pthread\_mutex\_trylock**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutex_trylock.txt) **(mutex)**  [**pthread\_mutex\_unlock**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutex_unlock.txt) **(mutex)** |

https://computing.llnl.gov/tutorials/images/arrowBullet.gifUsage:

* The pthread\_mutex\_lock() routine is used by a thread to acquire a lock on the specified *mutex* variable. If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked.
* pthread\_mutex\_trylock() will attempt to lock a mutex. However, if the mutex is already locked, the routine will return immediately with a "busy" error code. This routine may be useful in preventing deadlock conditions, as in a priority-inversion situation.
* pthread\_mutex\_unlock() will unlock a mutex if called by the owning thread. Calling this routine is required after a thread has completed its use of protected data if other threads are to acquire the mutex for their work with the protected data. An error will be returned if:
  + If the mutex was already unlocked
  + If the mutex is owned by another thread
* There is nothing "magical" about mutexes...in fact they are akin to a "gentlemen's agreement" between participating threads. It is up to the code writer to insure that the necessary threads all make the the mutex lock and unlock calls correctly. The following scenario demonstrates a logical error:

|  |
| --- |
| **Thread 1 Thread 2 Thread 3**  Lock Lock  A = 2 A = A+1 A = A\*B  Unlock Unlock |

|  |  |  |
| --- | --- | --- |
|  | https://computing.llnl.gov/tutorials/images/question2.gif | Question: When more than one thread is waiting for a locked mutex, which thread will be granted the lock first after it is released? |

Top of Form

**Example: Using Mutexes**

* This example program illustrates the use of mutex variables in a threads program that performs a dot product.
* The main data is made available to all threads through a globally accessible structure.
* Each thread works on a different part of the data.
* The main thread waits for all the threads to complete their computations, and then it prints the resulting sum.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | https://computing.llnl.gov/tutorials/images/page01.gif | | **Using Mutexes Example** | |  | | | | 1   2   3   4   5   6   7   8   9   10   11   12   13   14   15   16   17   18   19   20   21   22   23   24   25   26   27   28   29   30   31   32   33   34   35   36   37   38   39   40   41   42   43   44   45   46   47   48   49   50   51   52   53   54   55   56   57   58   59   60   61   62   63   64   65   66   67   68   69   70   71   72   73   74   75   76   77   78   79   80   81   82   83   84   85   86   87   88   89   90   91   92   93   94   95   96   97   98   99  100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 |  | #include <pthread.h>  #include <stdio.h>  #include <stdlib.h>  /\*  The following structure contains the necessary information  to allow the function "dotprod" to access its input data and  place its output into the structure.  \*/  typedef struct  {  double \*a;  double \*b;  double sum;  int veclen;  } DOTDATA;  /\* Define globally accessible variables and a mutex \*/  #define NUMTHRDS 4  #define VECLEN 100  DOTDATA dotstr;  pthread\_t callThd[NUMTHRDS];  pthread\_mutex\_t mutexsum;  /\*  The function dotprod is activated when the thread is created.  All input to this routine is obtained from a structure  of type DOTDATA and all output from this function is written into  this structure. The benefit of this approach is apparent for the  multi-threaded program: when a thread is created we pass a single  argument to the activated function - typically this argument  is a thread number. All the other information required by the  function is accessed from the globally accessible structure.  \*/  void \*dotprod(void \*arg)  {  /\* Define and use local variables for convenience \*/  int i, start, end, len ;  long offset;  double mysum, \*x, \*y;  offset = (long)arg;    len = dotstr.veclen;  start = offset\*len;  end = start + len;  x = dotstr.a;  y = dotstr.b;  /\*  Perform the dot product and assign result  to the appropriate variable in the structure.  \*/  mysum = 0;  for (i=start; i<end ; i++)  {  mysum += (x[i] \* y[i]);  }  /\*  Lock a mutex prior to updating the value in the shared  structure, and unlock it upon updating.  \*/  pthread\_mutex\_lock (&mutexsum);  dotstr.sum += mysum;  pthread\_mutex\_unlock (&mutexsum);  pthread\_exit((void\*) 0);  }  /\*  The main program creates threads which do all the work and then  print out result upon completion. Before creating the threads,  the input data is created. Since all threads update a shared structure,  we need a mutex for mutual exclusion. The main thread needs to wait for  all threads to complete, it waits for each one of the threads. We specify  a thread attribute value that allow the main thread to join with the  threads it creates. Note also that we free up handles when they are  no longer needed.  \*/  int main (int argc, char \*argv[])  {  long i;  double \*a, \*b;  void \*status;  pthread\_attr\_t attr;  /\* Assign storage and initialize values \*/  a = (double\*) malloc (NUMTHRDS\*VECLEN\*sizeof(double));  b = (double\*) malloc (NUMTHRDS\*VECLEN\*sizeof(double));    for (i=0; i<VECLEN\*NUMTHRDS; i++)  {  a[i]=1.0;  b[i]=a[i];  }  dotstr.veclen = VECLEN;  dotstr.a = a;  dotstr.b = b;  dotstr.sum=0;  pthread\_mutex\_init(&mutexsum, NULL);    /\* Create threads to perform the dotproduct \*/  pthread\_attr\_init(&attr);  pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_JOINABLE);  for(i=0; i<NUMTHRDS; i++)  {  /\*  Each thread works on a different set of data. The offset is specified  by 'i'. The size of the data for each thread is indicated by VECLEN.  \*/  pthread\_create(&callThd[i], &attr, dotprod, (void \*)i);  }  pthread\_attr\_destroy(&attr);  /\* Wait on the other threads \*/  for(i=0; i<NUMTHRDS; i++)  {  pthread\_join(callThd[i], &status);  }  /\* After joining, print out the results and cleanup \*/  printf ("Sum = %f \n", dotstr.sum);  free (a);  free (b);  pthread\_mutex\_destroy(&mutexsum);  pthread\_exit(NULL);  }  [https://computing.llnl.gov/tutorials/images/source1.gif](https://computing.llnl.gov/tutorials/pthreads/samples/dotprod_serial.c)*Serial version*  [https://computing.llnl.gov/tutorials/images/source1.gif](https://computing.llnl.gov/tutorials/pthreads/samples/dotprod_mutex.c)*Pthreads version* | |

|  |
| --- |
| Condition Variables |

**Overview**

* Condition variables provide yet another way for threads to synchronize. While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
* Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. This can be very resource consuming since the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.
* A condition variable is always used in conjunction with a mutex lock.
* A representative sequence for using condition variables is shown below.

|  |  |
| --- | --- |
| Main Thread   * + Declare and initialize global data/variables which require synchronization (such as "count")   + Declare and initialize a condition variable object   + Declare and initialize an associated mutex   + Create threads A and B to do work | |
| Thread A   * + Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)   + Lock associated mutex and check value of a global variable   + Call pthread\_cond\_wait() to perform a blocking wait for signal from Thread-B. Note that a call to pthread\_cond\_wait() automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.   + When signalled, wake up. Mutex is automatically and atomically locked.   + Explicitly unlock mutex   + Continue | Thread B   * + Do work   + Lock associated mutex   + Change the value of the global variable that Thread-A is waiting upon.   + Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.   + Unlock mutex.   + Continue |
| Main Thread  Join / Continue | |

|  |
| --- |
| Condition Variables |

**Creating and Destroying Condition Variables**

https://computing.llnl.gov/tutorials/images/arrowBullet.gifRoutines:

|  |
| --- |
| [**pthread\_cond\_init**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_cond_init.txt) **(condition,attr)**  [**pthread\_cond\_destroy**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_cond_destroy.txt) **(condition)**  [**pthread\_condattr\_init**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_condattr_init.txt) **(attr)**  [**pthread\_condattr\_destroy**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_condattr_destroy.txt) **(attr)** |

https://computing.llnl.gov/tutorials/images/arrowBullet.gifUsage:

* Condition variables must be declared with type pthread\_cond\_t, and must be initialized before they can be used. There are two ways to initialize a condition variable:
  1. Statically, when it is declared. For example:   
     pthread\_cond\_t myconvar = PTHREAD\_COND\_INITIALIZER;
  2. Dynamically, with the pthread\_cond\_init() routine. The ID of the created condition variable is returned to the calling thread through the *condition* parameter. This method permits setting condition variable object attributes, *attr*.
* The optional *attr* object is used to set condition variable attributes. There is only one attribute defined for condition variables: process-shared, which allows the condition variable to be seen by threads in other processes. The attribute object, if used, must be of type pthread\_condattr\_t (may be specified as NULL to accept defaults).

Note that not all implementations may provide the process-shared attribute.

* The pthread\_condattr\_init() and pthread\_condattr\_destroy() routines are used to create and destroy condition variable attribute objects.
* pthread\_cond\_destroy() should be used to free a condition variable that is no longer needed.

|  |
| --- |
| Condition Variables |

**Waiting and Signaling on Condition Variables**

https://computing.llnl.gov/tutorials/images/arrowBullet.gifRoutines:

|  |
| --- |
| [**pthread\_cond\_wait**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_cond_wait.txt) **(condition,mutex)**  [**pthread\_cond\_signal**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_cond_signal.txt) **(condition)**  [**pthread\_cond\_broadcast**](https://computing.llnl.gov/tutorials/pthreads/man/pthread_cond_broadcast.txt) **(condition)** |

https://computing.llnl.gov/tutorials/images/arrowBullet.gifUsage:

* pthread\_cond\_wait() blocks the calling thread until the specified *condition* is signalled. This routine should be called while *mutex* is locked, and it will automatically release the mutex while it waits. After signal is received and thread is awakened, *mutex* will be automatically locked for use by the thread. The programmer is then responsible for unlocking *mutex* when the thread is finished with it.

**Recommendation:** Using a WHILE loop instead of an IF statement (see watch\_count routine in example below) to check the waited for condition can help deal with several potential problems, such as:

* + If several threads are waiting for the same wake up signal, they will take turns acquiring the mutex, and any one of them can then modify the condition they all waited for.
  + If the thread received the signal in error due to a program bug
  + The Pthreads library is permitted to issue spurious wake ups to a waiting thread without violating the standard.
* The pthread\_cond\_signal() routine is used to signal (or wake up) another thread which is waiting on the condition variable. It should be called after *mutex* is locked, and must unlock *mutex* in order for pthread\_cond\_wait() routine to complete.
* The pthread\_cond\_broadcast() routine should be used instead of pthread\_cond\_signal() if more than one thread is in a blocking wait state.
* It is a logical error to call pthread\_cond\_signal() before calling pthread\_cond\_wait().

|  |  |
| --- | --- |
| https://computing.llnl.gov/tutorials/images/warning5.gif | Proper locking and unlocking of the associated mutex variable is essential when using these routines. For example:   * Failing to lock the mutex before calling pthread\_cond\_wait() may cause it NOT to block. * Failing to unlock the mutex after calling pthread\_cond\_signal() may not allow a matching pthread\_cond\_wait() routine to complete (it will remain blocked). |

**Example: Using Condition Variables**

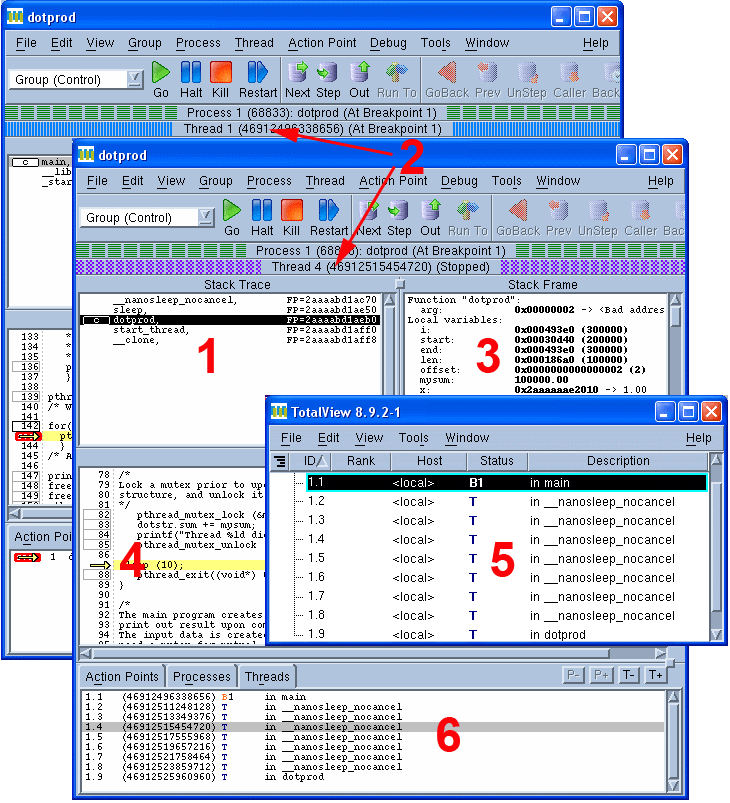
* This simple example code demonstrates the use of several Pthread condition variable routines.
* The main routine creates three threads.
* Two of the threads perform work and update a "count" variable.
* The third thread waits until the count variable reaches a specified value.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | https://computing.llnl.gov/tutorials/images/page01.gif | | **Using Condition Variables Example** | |  | | | | 1   2   3   4   5   6   7   8   9   10   11   12   13   14   15   16   17   18   19   20   21   22   23   24   25   26   27   28   29   30   31   32   33   34   35   36   37   38   39   40   41   42   43   44   45   46   47   48   49   50   51   52   53   54   55   56   57   58   59   60   61   62   63   64   65   66   67   68   69   70   71   72   73   74   75   76   77   78   79   80   81   82   83   84   85   86   87   88   89   90   91   92   93   94   95   96 |  | #include <pthread.h>  #include <stdio.h>  #include <stdlib.h>  #define NUM\_THREADS 3  #define TCOUNT 10  #define COUNT\_LIMIT 12  int count = 0;  int thread\_ids[3] = {0,1,2};  pthread\_mutex\_t count\_mutex;  pthread\_cond\_t count\_threshold\_cv;  void \*inc\_count(void \*t)  {  int i;  long my\_id = (long)t;  for (i=0; i<TCOUNT; i++) {  pthread\_mutex\_lock(&count\_mutex);  count++;  /\*  Check the value of count and signal waiting thread when condition is  reached. Note that this occurs while mutex is locked.  \*/  if (count == COUNT\_LIMIT) {  pthread\_cond\_signal(&count\_threshold\_cv);  printf("inc\_count(): thread %ld, count = %d Threshold reached.\n",  my\_id, count);  }  printf("inc\_count(): thread %ld, count = %d, unlocking mutex\n",  my\_id, count);  pthread\_mutex\_unlock(&count\_mutex);  /\* Do some "work" so threads can alternate on mutex lock \*/  sleep(1);  }  pthread\_exit(NULL);  }  void \*watch\_count(void \*t)  {  long my\_id = (long)t;  printf("Starting watch\_count(): thread %ld\n", my\_id);  /\*  Lock mutex and wait for signal. Note that the pthread\_cond\_wait  routine will automatically and atomically unlock mutex while it waits.  Also, note that if COUNT\_LIMIT is reached before this routine is run by  the waiting thread, the loop will be skipped to prevent pthread\_cond\_wait  from never returning.  \*/  pthread\_mutex\_lock(&count\_mutex);  while (count<COUNT\_LIMIT) {  pthread\_cond\_wait(&count\_threshold\_cv, &count\_mutex);  printf("watch\_count(): thread %ld Condition signal received.\n", my\_id);  }  count += 125;  printf("watch\_count(): thread %ld count now = %d.\n", my\_id, count);  pthread\_mutex\_unlock(&count\_mutex);  pthread\_exit(NULL);  }  int main (int argc, char \*argv[])  {  int i, rc;  long t1=1, t2=2, t3=3;  pthread\_t threads[3];  pthread\_attr\_t attr;  /\* Initialize mutex and condition variable objects \*/  pthread\_mutex\_init(&count\_mutex, NULL);  pthread\_cond\_init (&count\_threshold\_cv, NULL);  /\* For portability, explicitly create threads in a joinable state \*/  pthread\_attr\_init(&attr);  pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_JOINABLE);  pthread\_create(&threads[0], &attr, watch\_count, (void \*)t1);  pthread\_create(&threads[1], &attr, inc\_count, (void \*)t2);  pthread\_create(&threads[2], &attr, inc\_count, (void \*)t3);  /\* Wait for all threads to complete \*/  for (i=0; i<NUM\_THREADS; i++) {  pthread\_join(threads[i], NULL);  }  printf ("Main(): Waited on %d threads. Done.\n", NUM\_THREADS);  /\* Clean up and exit \*/  pthread\_attr\_destroy(&attr);  pthread\_mutex\_destroy(&count\_mutex);  pthread\_cond\_destroy(&count\_threshold\_cv);  pthread\_exit(NULL);  }  [https://computing.llnl.gov/tutorials/images/output1.gif](https://computing.llnl.gov/tutorials/pthreads/samples/condvar.out)  Starting watch\_count(): thread 1  inc\_count(): thread 2, count = 1, unlocking mutex  inc\_count(): thread 3, count = 2, unlocking mutex  watch\_count(): thread 1 going into wait...  inc\_count(): thread 3, count = 3, unlocking mutex  inc\_count(): thread 2, count = 4, unlocking mutex  inc\_count(): thread 3, count = 5, unlocking mutex  inc\_count(): thread 2, count = 6, unlocking mutex  inc\_count(): thread 3, count = 7, unlocking mutex  inc\_count(): thread 2, count = 8, unlocking mutex  inc\_count(): thread 3, count = 9, unlocking mutex  inc\_count(): thread 2, count = 10, unlocking mutex  inc\_count(): thread 3, count = 11, unlocking mutex  inc\_count(): thread 2, count = 12 Threshold reached. Just sent signal.  inc\_count(): thread 2, count = 12, unlocking mutex  watch\_count(): thread 1 Condition signal received.  watch\_count(): thread 1 count now = 137.  inc\_count(): thread 3, count = 138, unlocking mutex  inc\_count(): thread 2, count = 139, unlocking mutex  inc\_count(): thread 3, count = 140, unlocking mutex  inc\_count(): thread 2, count = 141, unlocking mutex  inc\_count(): thread 3, count = 142, unlocking mutex  inc\_count(): thread 2, count = 143, unlocking mutex  inc\_count(): thread 3, count = 144, unlocking mutex  inc\_count(): thread 2, count = 145, unlocking mutex  Main(): Waited on 3 threads. Final value of count = 145. Done. | |

|  |
| --- |
| Monitoring, Debugging and Performance Analysis Tools for Pthreads |

https://computing.llnl.gov/tutorials/images/arrowBullet.gifMonitoring and Debugging Pthreads:

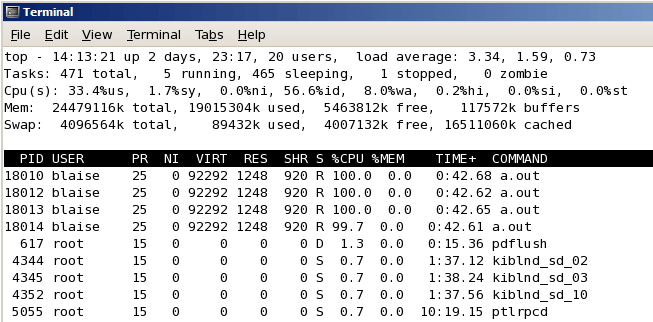
* Debuggers vary in their ability to handle Pthreads. The TotalView debugger is LC's recommended debugger for parallel programs. It is well suited for both monitoring and debugging threaded programs.
* An example screenshot from a TotalView session using a threaded code is shown below.
  1. Stack Trace Pane: Displays the call stack of routines that the selected thread is executing.
  2. Status Bars: Show status information for the selected thread and its associated process.
  3. Stack Frame Pane: Shows a selected thread's stack variables, registers, etc.
  4. Source Pane: Shows the source code for the selected thread.
  5. Root Window showing all threads
  6. Threads Pane: Shows threads associated with the selected process



* See the [TotalView Debugger tutorial](https://computing.llnl.gov/tutorials/totalview/index.html) for details.
* The Linux **ps** command provides several flags for viewing thread information. Some examples are shown below. See the [man page](https://computing.llnl.gov/tutorials/pthreads/man/ps.txt) for details.

|  |
| --- |
| % ps -Lf  UID PID PPID LWP C NLWP STIME TTY TIME CMD  blaise 22529 28240 22529 0 5 11:31 pts/53 00:00:00 a.out  blaise 22529 28240 22530 99 5 11:31 pts/53 00:01:24 a.out  blaise 22529 28240 22531 99 5 11:31 pts/53 00:01:24 a.out  blaise 22529 28240 22532 99 5 11:31 pts/53 00:01:24 a.out  blaise 22529 28240 22533 99 5 11:31 pts/53 00:01:24 a.out  % ps -T  PID SPID TTY TIME CMD  22529 22529 pts/53 00:00:00 a.out  22529 22530 pts/53 00:01:49 a.out  22529 22531 pts/53 00:01:49 a.out  22529 22532 pts/53 00:01:49 a.out  22529 22533 pts/53 00:01:49 a.out  % ps -Lm  PID LWP TTY TIME CMD  22529 - pts/53 00:18:56 a.out  - 22529 - 00:00:00 -  - 22530 - 00:04:44 -  - 22531 - 00:04:44 -  - 22532 - 00:04:44 -  - 22533 - 00:04:44 - |

* LC's Linux clusters also provide the **top** command to monitor processes on a node. If used with the **-H** flag, the threads contained within a process will be visible. An example of the **top -H** command is shown below. The parent process is PID 18010 which spawned three threads, shown as PIDs 18012, 18013 and 18014.



https://computing.llnl.gov/tutorials/images/arrowBullet.gifPerformance Analysis Tools:

* There are a variety of performance analysis tools that can be used with threaded programs. Searching the web will turn up a wealth of information.
* At LC, the list of supported computing tools can be found at: [computing.llnl.gov/code/content/software\_tools.php](https://computing.llnl.gov/code/content/software_tools.php).
* These tools vary significantly in their complexity, functionality and learning curve. Covering them in detail is beyond the scope of this tutorial.
* Some tools worth investigating, specifically for threaded codes, include:
  + Open|SpeedShop
  + TAU
  + HPCToolkit
  + PAPI
  + Intel VTune Amplifier
  + ThreadSpotter

|  |
| --- |
| LLNL Specific Information and Recommendations |

This section describes details specific to Livermore Computing's systems.

https://computing.llnl.gov/tutorials/images/arrowBullet.gifImplementations:

* All LC production systems include a Pthreads implementation that follows draft 10 (final) of the POSIX standard. This is the preferred implementation.
* Implementations differ in the maximum number of threads that a process may create. They also differ in the default amount of thread stack space.

https://computing.llnl.gov/tutorials/images/arrowBullet.gifCompiling:

* LC maintains a number of compilers, and usually several different versions of each - see the [LC's Supported Compilers](https://computing.llnl.gov/code/compilers.html) web page.
* The compiler commands described in the [Compiling Threaded Programs](https://computing.llnl.gov/tutorials/pthreads/#Compiling) section apply to LC systems.

https://computing.llnl.gov/tutorials/images/arrowBullet.gifMixing MPI with Pthreads:

* This is the primary motivation for using Pthreads at LC.
* Design:
  + Each MPI process typically creates and then manages N threads, where N makes the best use of the available cores/node.
  + Finding the best value for N will vary with the platform and your application's characteristics.
  + In general, there may be problems if multiple threads make MPI calls. The program may fail or behave unexpectedly. If MPI calls must be made from within a thread, they should be made only by one thread.
* Compiling:
  + Use the appropriate MPI compile command for the platform and language of choice
  + Be sure to include the required Pthreads flag as shown in the [Compiling Threaded Programs](https://computing.llnl.gov/tutorials/pthreads/#Compiling) section.
* An example code that uses both MPI and Pthreads is available below. The serial, threads-only, MPI-only and MPI-with-threads versions demonstrate one possible progression.
  + [Serial](https://computing.llnl.gov/tutorials/pthreads/samples/mpithreads_serial.c)
  + [Pthreads only](https://computing.llnl.gov/tutorials/pthreads/samples/mpithreads_threads.c)
  + [MPI only](https://computing.llnl.gov/tutorials/pthreads/samples/mpithreads_mpi.c)
  + [MPI with pthreads](https://computing.llnl.gov/tutorials/pthreads/samples/mpithreads_both.c)
  + [makefile](https://computing.llnl.gov/tutorials/pthreads/samples/mpithreads.makefile)

|  |
| --- |
| Topics Not Covered |

Several features of the Pthreads API are not covered in this tutorial. These are listed below. See the [Pthread Library Routines Reference](https://computing.llnl.gov/tutorials/pthreads/#Routines) section for more information.

* Thread Scheduling
  + Implementations will differ on how threads are scheduled to run. In most cases, the default mechanism is adequate.
  + The Pthreads API provides routines to explicitly set thread scheduling policies and priorities which may override the default mechanisms.
  + The API does not require implementations to support these features.
* Keys: Thread-Specific Data
  + As threads call and return from different routines, the local data on a thread's stack comes and goes.
  + To preserve stack data you can usually pass it as an argument from one routine to the next, or else store the data in a global variable associated with a thread.
  + Pthreads provides another, possibly more convenient and versatile, way of accomplishing this through ***keys***.
* Mutex Protocol Attributes and Mutex Priority Management for the handling of "priority inversion" problems.
* Condition Variable Sharing - across processes
* Thread Cancellation
* Threads and Signals
* Synchronization constructs - barriers and locks

|  |
| --- |
| Pthread Exercise 2 |

**Mutexes, Condition Variables and Hybrid MPI with Pthreads**

|  |
| --- |
| **Overview:**   * **Login to the LC workshop cluster, if you are not already logged in** * **Mutexes: review and run the provided example codes** * **Condition variables: review and run the provided example codes** * **Hybrid MPI with Pthreads: review and run the provided example codes**   **https://computing.llnl.gov/tutorials/images/point02.jpg**[**GO TO THE EXERCISE HERE**](https://computing.llnl.gov/tutorials/pthreads/exercise.html#Exercise2) |

**This completes the tutorial.**

|  |  |
| --- | --- |
| [Evaluation Form](https://computing.llnl.gov/tutorials/evaluation/index.html) | Please complete the online evaluation form - unless you are doing the exercise, in which case please complete it at the end of the exercise. |

**Where would you like to go now?**

* [Exercise](https://computing.llnl.gov/tutorials/pthreads/exercise.html)
* [Agenda](https://computing.llnl.gov/tutorials/agenda/index.html)
* [Back to the top](https://computing.llnl.gov/tutorials/pthreads/#top)

|  |
| --- |
| References and More Information |

* Author: [Blaise Barney](mailto:blaiseb@llnl.gov), Livermore Computing.
* POSIX Standard: [www.unix.org/version3/ieee\_std.html](http://www.unix.org/version3/ieee_std.html)
* "Pthreads Programming". B. Nichols et al. O'Reilly and Associates.
* "Threads Primer". B. Lewis and D. Berg. Prentice Hall
* "Programming With POSIX Threads". D. Butenhof. Addison Wesley
* "Programming With Threads". S. Kleiman et al. Prentice Hall

|  |
| --- |
| Appendix A: Pthread Library Routines Reference |

For convenience, an alphabetical list of Pthread routines, linked to their corresponding man page, is provided below.

[pthread\_atfork](https://computing.llnl.gov/tutorials/pthreads/man/pthread_atfork.txt)   
[pthread\_attr\_destroy](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_destroy.txt)   
[pthread\_attr\_getdetachstate](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_getdetachstate.txt)   
[pthread\_attr\_getguardsize](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_getguardsize.txt)   
[pthread\_attr\_getinheritsched](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_getinheritsched.txt)   
[pthread\_attr\_getschedparam](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_getschedparam.txt)   
[pthread\_attr\_getschedpolicy](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_getschedpolicy.txt)   
[pthread\_attr\_getscope](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_getscope.txt)   
[pthread\_attr\_getstack](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_getstack.txt)   
[pthread\_attr\_getstackaddr](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_getstackaddr.txt)   
[pthread\_attr\_getstacksize](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_getstacksize.txt)   
[pthread\_attr\_init](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_init.txt)   
[pthread\_attr\_setdetachstate](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_setdetachstate.txt)   
[pthread\_attr\_setguardsize](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_setguardsize.txt)   
[pthread\_attr\_setinheritsched](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_setinheritsched.txt)   
[pthread\_attr\_setschedparam](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_setschedparam.txt)   
[pthread\_attr\_setschedpolicy](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_setschedpolicy.txt)   
[pthread\_attr\_setscope](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_setscope.txt)   
[pthread\_attr\_setstack](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_setstack.txt)   
[pthread\_attr\_setstackaddr](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_setstackaddr.txt)   
[pthread\_attr\_setstacksize](https://computing.llnl.gov/tutorials/pthreads/man/pthread_attr_setstacksize.txt)   
[pthread\_barrier\_destroy](https://computing.llnl.gov/tutorials/pthreads/man/pthread_barrier_destroy.txt)   
[pthread\_barrier\_init](https://computing.llnl.gov/tutorials/pthreads/man/pthread_barrier_init.txt)   
[pthread\_barrier\_wait](https://computing.llnl.gov/tutorials/pthreads/man/pthread_barrier_wait.txt)   
[pthread\_barrierattr\_destroy](https://computing.llnl.gov/tutorials/pthreads/man/pthread_barrierattr_destroy.txt)   
[pthread\_barrierattr\_getpshared](https://computing.llnl.gov/tutorials/pthreads/man/pthread_barrierattr_getpshared.txt)   
[pthread\_barrierattr\_init](https://computing.llnl.gov/tutorials/pthreads/man/pthread_barrierattr_init.txt)   
[pthread\_barrierattr\_setpshared](https://computing.llnl.gov/tutorials/pthreads/man/pthread_barrierattr_setpshared.txt)   
[pthread\_cancel](https://computing.llnl.gov/tutorials/pthreads/man/pthread_cancel.txt)   
[pthread\_cleanup\_pop](https://computing.llnl.gov/tutorials/pthreads/man/pthread_cleanup_pop.txt)   
[pthread\_cleanup\_push](https://computing.llnl.gov/tutorials/pthreads/man/pthread_cleanup_push.txt)   
[pthread\_cond\_broadcast](https://computing.llnl.gov/tutorials/pthreads/man/pthread_cond_broadcast.txt)   
[pthread\_cond\_destroy](https://computing.llnl.gov/tutorials/pthreads/man/pthread_cond_destroy.txt)   
[pthread\_cond\_init](https://computing.llnl.gov/tutorials/pthreads/man/pthread_cond_init.txt)   
[pthread\_cond\_signal](https://computing.llnl.gov/tutorials/pthreads/man/pthread_cond_signal.txt)   
[pthread\_cond\_timedwait](https://computing.llnl.gov/tutorials/pthreads/man/pthread_cond_timedwait.txt)   
[pthread\_cond\_wait](https://computing.llnl.gov/tutorials/pthreads/man/pthread_cond_wait.txt)   
[pthread\_condattr\_destroy](https://computing.llnl.gov/tutorials/pthreads/man/pthread_condattr_destroy.txt)   
[pthread\_condattr\_getclock](https://computing.llnl.gov/tutorials/pthreads/man/pthread_condattr_getclock.txt)   
[pthread\_condattr\_getpshared](https://computing.llnl.gov/tutorials/pthreads/man/pthread_condattr_getpshared.txt)   
[pthread\_condattr\_init](https://computing.llnl.gov/tutorials/pthreads/man/pthread_condattr_init.txt)   
[pthread\_condattr\_setclock](https://computing.llnl.gov/tutorials/pthreads/man/pthread_condattr_setclock.txt)   
[pthread\_condattr\_setpshared](https://computing.llnl.gov/tutorials/pthreads/man/pthread_condattr_setpshared.txt)   
[pthread\_create](https://computing.llnl.gov/tutorials/pthreads/man/pthread_create.txt)   
[pthread\_detach](https://computing.llnl.gov/tutorials/pthreads/man/pthread_detach.txt)   
[pthread\_equal](https://computing.llnl.gov/tutorials/pthreads/man/pthread_equal.txt)   
[pthread\_exit](https://computing.llnl.gov/tutorials/pthreads/man/pthread_exit.txt)   
[pthread\_getconcurrency](https://computing.llnl.gov/tutorials/pthreads/man/pthread_getconcurrency.txt)   
[pthread\_getcpuclockid](https://computing.llnl.gov/tutorials/pthreads/man/pthread_getcpuclockid.txt)   
[pthread\_getschedparam](https://computing.llnl.gov/tutorials/pthreads/man/pthread_getschedparam.txt)   
[pthread\_getspecific](https://computing.llnl.gov/tutorials/pthreads/man/pthread_getspecific.txt)   
[pthread\_join](https://computing.llnl.gov/tutorials/pthreads/man/pthread_join.txt)   
[pthread\_key\_create](https://computing.llnl.gov/tutorials/pthreads/man/pthread_key_create.txt)   
[pthread\_key\_delete](https://computing.llnl.gov/tutorials/pthreads/man/pthread_key_delete.txt)   
[pthread\_kill](https://computing.llnl.gov/tutorials/pthreads/man/pthread_kill.txt)   
[pthread\_mutex\_destroy](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutex_destroy.txt)   
[pthread\_mutex\_getprioceiling](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutex_getprioceiling.txt)   
[pthread\_mutex\_init](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutex_init.txt)   
[pthread\_mutex\_lock](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutex_lock.txt)   
[pthread\_mutex\_setprioceiling](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutex_setprioceiling.txt)   
[pthread\_mutex\_timedlock](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutex_timedlock.txt)   
[pthread\_mutex\_trylock](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutex_trylock.txt)   
[pthread\_mutex\_unlock](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutex_unlock.txt)   
[pthread\_mutexattr\_destroy](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutexattr_destroy.txt)   
[pthread\_mutexattr\_getprioceiling](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutexattr_getprioceiling.txt)   
[pthread\_mutexattr\_getprotocol](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutexattr_getprotocol.txt)   
[pthread\_mutexattr\_getpshared](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutexattr_getpshared.txt)   
[pthread\_mutexattr\_gettype](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutexattr_gettype.txt)   
[pthread\_mutexattr\_init](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutexattr_init.txt)   
[pthread\_mutexattr\_setprioceiling](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutexattr_setprioceiling.txt)   
[pthread\_mutexattr\_setprotocol](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutexattr_setprotocol.txt)   
[pthread\_mutexattr\_setpshared](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutexattr_setpshared.txt)   
[pthread\_mutexattr\_settype](https://computing.llnl.gov/tutorials/pthreads/man/pthread_mutexattr_settype.txt)   
[pthread\_once](https://computing.llnl.gov/tutorials/pthreads/man/pthread_once.txt)   
[pthread\_rwlock\_destroy](https://computing.llnl.gov/tutorials/pthreads/man/pthread_rwlock_destroy.txt)   
[pthread\_rwlock\_init](https://computing.llnl.gov/tutorials/pthreads/man/pthread_rwlock_init.txt)   
[pthread\_rwlock\_rdlock](https://computing.llnl.gov/tutorials/pthreads/man/pthread_rwlock_rdlock.txt)   
[pthread\_rwlock\_timedrdlock](https://computing.llnl.gov/tutorials/pthreads/man/pthread_rwlock_timedrdlock.txt)   
[pthread\_rwlock\_timedwrlock](https://computing.llnl.gov/tutorials/pthreads/man/pthread_rwlock_timedwrlock.txt)   
[pthread\_rwlock\_tryrdlock](https://computing.llnl.gov/tutorials/pthreads/man/pthread_rwlock_tryrdlock.txt)   
[pthread\_rwlock\_trywrlock](https://computing.llnl.gov/tutorials/pthreads/man/pthread_rwlock_trywrlock.txt)   
[pthread\_rwlock\_unlock](https://computing.llnl.gov/tutorials/pthreads/man/pthread_rwlock_unlock.txt)   
[pthread\_rwlock\_wrlock](https://computing.llnl.gov/tutorials/pthreads/man/pthread_rwlock_wrlock.txt)   
[pthread\_rwlockattr\_destroy](https://computing.llnl.gov/tutorials/pthreads/man/pthread_rwlockattr_destroy.txt)   
[pthread\_rwlockattr\_getpshared](https://computing.llnl.gov/tutorials/pthreads/man/pthread_rwlockattr_getpshared.txt)   
[pthread\_rwlockattr\_init](https://computing.llnl.gov/tutorials/pthreads/man/pthread_rwlockattr_init.txt)   
[pthread\_rwlockattr\_setpshared](https://computing.llnl.gov/tutorials/pthreads/man/pthread_rwlockattr_setpshared.txt)   
[pthread\_self](https://computing.llnl.gov/tutorials/pthreads/man/pthread_self.txt)   
[pthread\_setcancelstate](https://computing.llnl.gov/tutorials/pthreads/man/pthread_setcancelstate.txt)   
[pthread\_setcanceltype](https://computing.llnl.gov/tutorials/pthreads/man/pthread_setcanceltype.txt)   
[pthread\_setconcurrency](https://computing.llnl.gov/tutorials/pthreads/man/pthread_setconcurrency.txt)   
[pthread\_setschedparam](https://computing.llnl.gov/tutorials/pthreads/man/pthread_setschedparam.txt)   
[pthread\_setschedprio](https://computing.llnl.gov/tutorials/pthreads/man/pthread_setschedprio.txt)   
[pthread\_setspecific](https://computing.llnl.gov/tutorials/pthreads/man/pthread_setspecific.txt)   
[pthread\_sigmask](https://computing.llnl.gov/tutorials/pthreads/man/pthread_sigmask.txt)   
[pthread\_spin\_destroy](https://computing.llnl.gov/tutorials/pthreads/man/pthread_spin_destroy.txt)   
[pthread\_spin\_init](https://computing.llnl.gov/tutorials/pthreads/man/pthread_spin_init.txt)   
[pthread\_spin\_lock](https://computing.llnl.gov/tutorials/pthreads/man/pthread_spin_lock.txt)   
[pthread\_spin\_trylock](https://computing.llnl.gov/tutorials/pthreads/man/pthread_spin_trylock.txt)   
[pthread\_spin\_unlock](https://computing.llnl.gov/tutorials/pthreads/man/pthread_spin_unlock.txt)   
[pthread\_testcancel](https://computing.llnl.gov/tutorials/pthreads/man/pthread_testcancel.txt)

https://computing.llnl.gov/tutorials/pthreads/  
Last Modified: 03/07/2017 23:54:00 [blaiseb@llnl.gov](mailto:blaiseb@llnl.gov)  
UCRL-MI-133316

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Bottom of Form