Overview of Programming Models

- Programming models provide support for expressing concurrency and synchronization.
- **Process based models** assume that all data associated with a process is private, by default, unless otherwise specified.
- **Lightweight processes and threads** assume that all memory is global.
- **Directive based programming models** extend the threaded model by facilitating creation and synchronization of threads.

What is a Thread?

- A thread is defined as an independent stream of instructions that can be packaged and executed by the operating system (similar to a process but much less overhead) .
- Multiple threads can be executed in parallel on many computer systems. This *multithreading* generally occurs by **time slicing** in which case the processing is not literally simultaneous, for the single processor is really doing only one thing at a time.
- Unlike processes, threads typically share the state information of a single process, and share memory and other resources directly.

Overview of Programming Models

- A *thread* is a single stream of control in the flow of a program. A program like:

```
for (row = 0; row < n; row++)
  for (col = 0; col < n; col++)
        c[row][col] =
        dot_product(get_row(a, row),
                    get_col(b, col));
```

can be transformed to:

```
for (row = 0; row < n; row++)
  for (col = 0; col < n; col++)
        c[row][col] =
        create_thread(dot_product(get_row(a, row),
                      get_col(b, col)));
```

In this case, one may think of the thread as an instance of a function that returns before the function has finished executing.

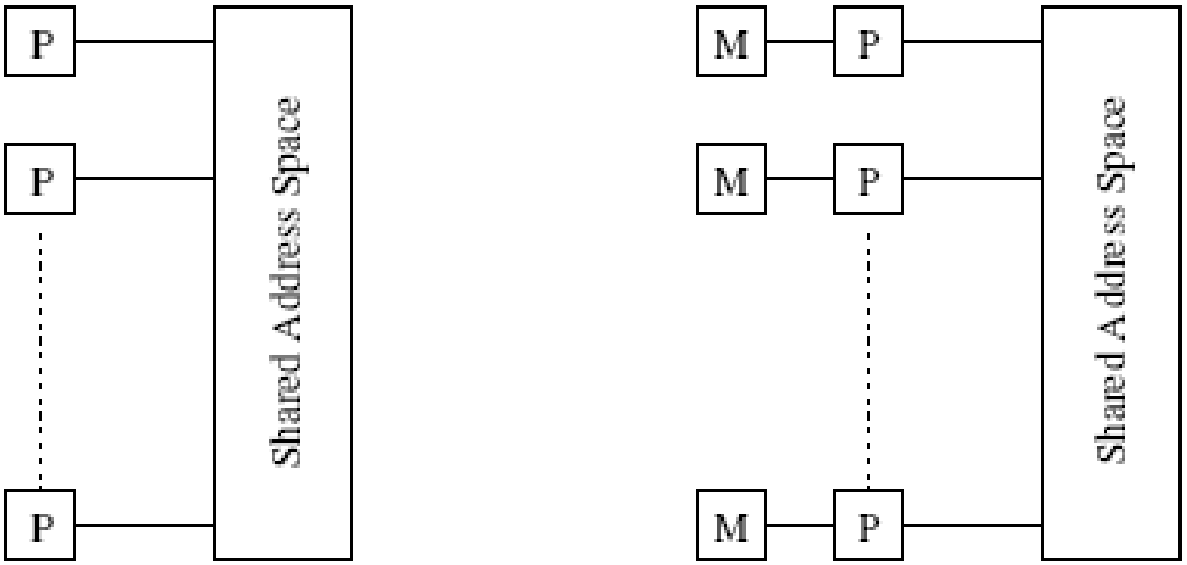
Thread Basics

- Threads provide software portability.
- Inherent support for latency hiding.
- Scheduling and load balancing.
- Ease of programming and widespread use.

Thread Basics

- All memory in the logical machine model of a thread is globally accessible to every thread.
- The stack corresponding to the function call is generally treated as being local to the thread for liveness reasons.
- This implies a logical machine model with both global memory (default) and local memory (stacks).
- It is important to note that such a flat model may result in very poor performance since memory is physically distributed in typical machines.

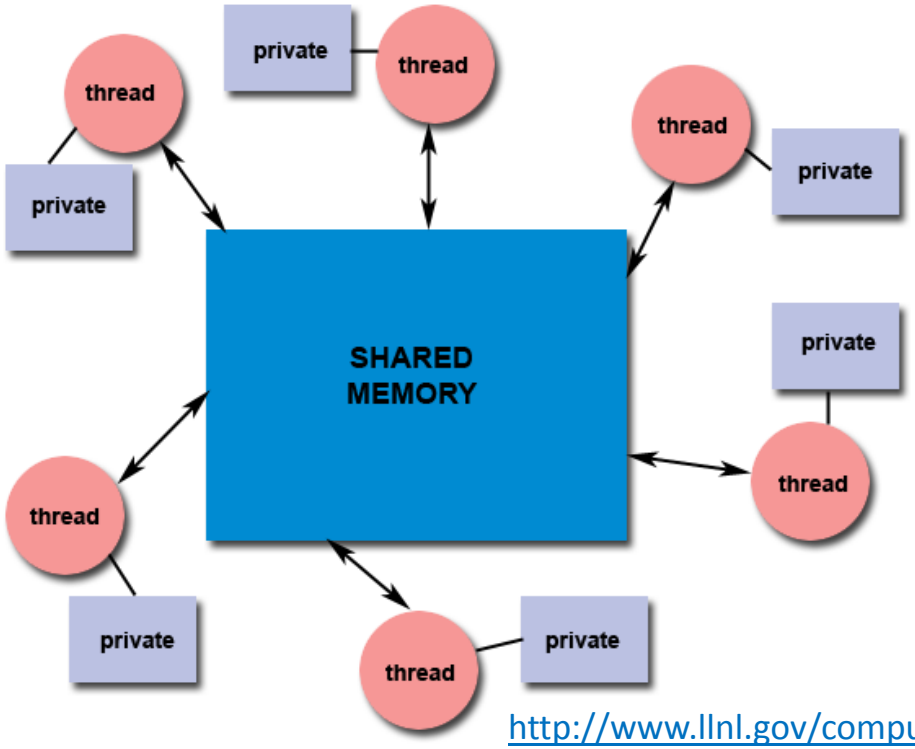
Thread Basics



- The logical machine model of a thread-based programming paradigm.

Shared Memory Model

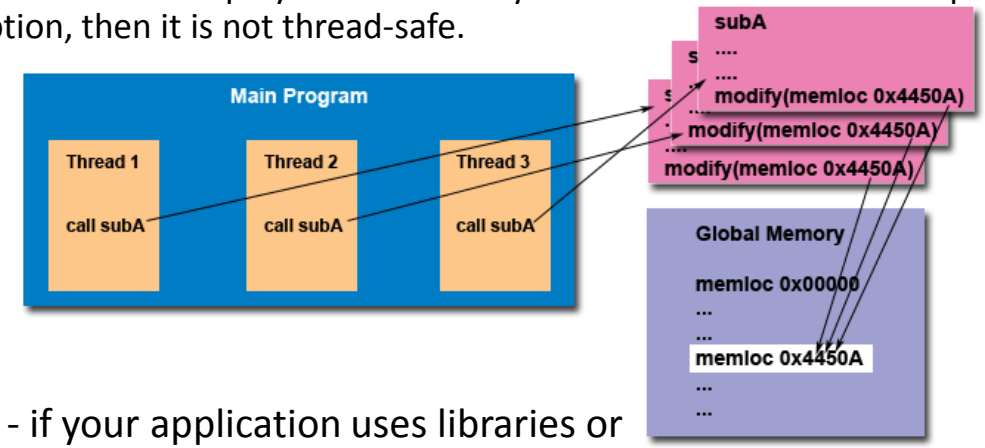
- All threads have access to the same global / shared memory.
- Additionally, threads have their own private data.
- Programmers are responsible for synchronizing access to global memory.



http://www.llnl.gov/computing/tutorials/pthreads/

Thread Safeness

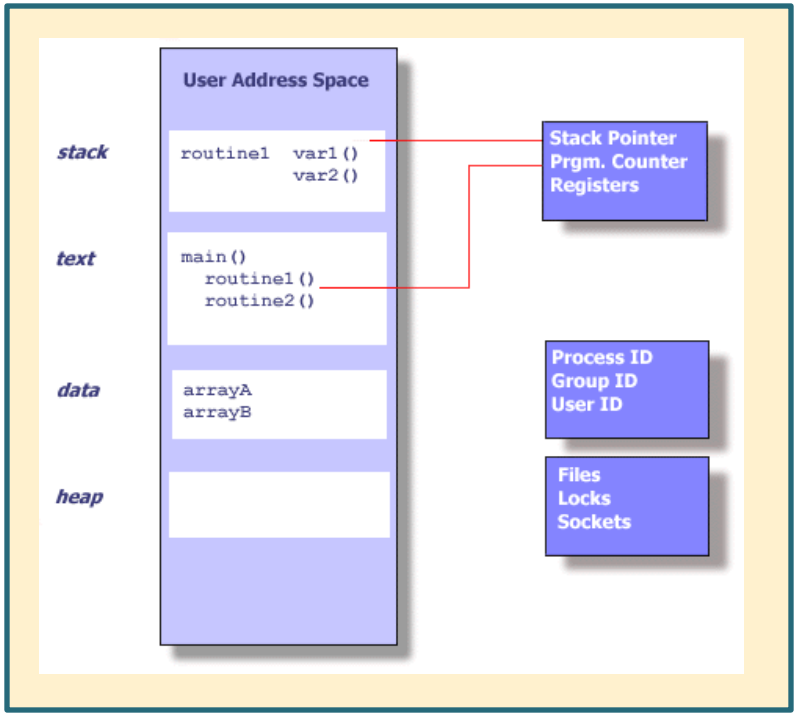
- Thread Safeness refers to the ability to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions.
- For example, suppose we have an application which creates three threads each of which calls subA(). Additionally:
  - subA() accesses/modifies a global 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.



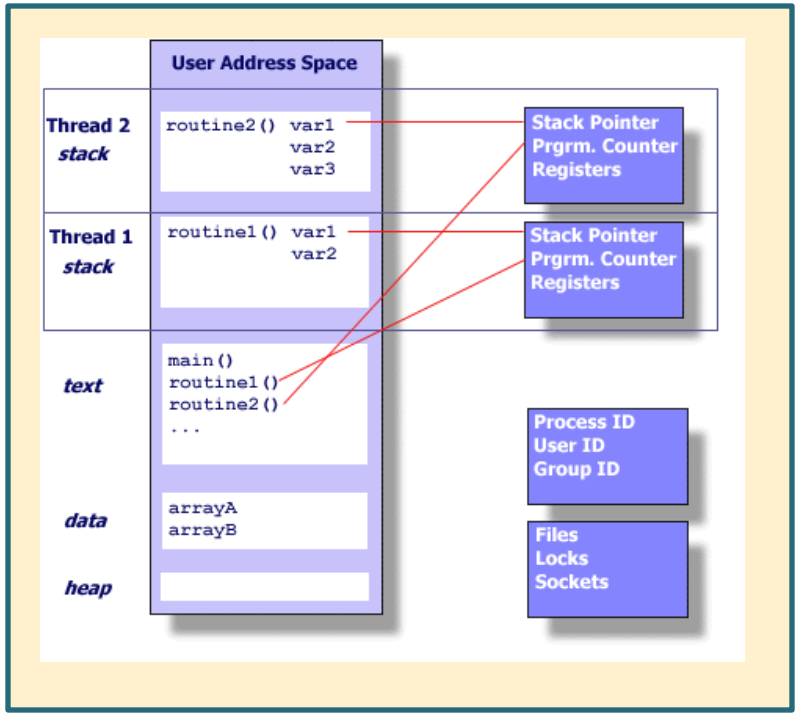
- **BE CAREFUL!** - if your application uses libraries or other objects that don't explicitly guarantee thread-safeness – assume that they are **NOT** thread-safe until proven otherwise. This can be done by "serializing" the calls to the uncertain routine.

http://www.llnl.gov/computing/tutorials/pthreads/

POSIX Threads



Standard Unix Process



Unix Process w/Threads

http://www.llnl.gov/computing/tutorials/pthreads/

Threads duplicate on the bare essential resources that enable them to exist as executable code.

POSIX Threads

- Independent flow of control is achieved by having each thread maintain its own:
  - Stack Pointer
  - Registers
  - Scheduling Properties (priority, etc.)
  - Signals
  - Thread Specific Data

http://www.llnl.gov/computing/tutorials/pthreads/

POSIX Threads

- Also, because threads share resources within the same process
  - Changes made by one thread to shared resources, i.e., closing a file, will be visible to all other threads.
  - Pointers with the same addresses point to the same data.
  - Explicit synchronization must be performed by the programming when accessing (reading/writing) the same memory (race conditions).

http://www.llnl.gov/computing/tutorials/pthreads/

The POSIX Thread API 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 which adhere to this standard are referred to as POSIX threads, or Pthreads. Most hardware vendors now offer Pthreads in addition to their proprietary API's.

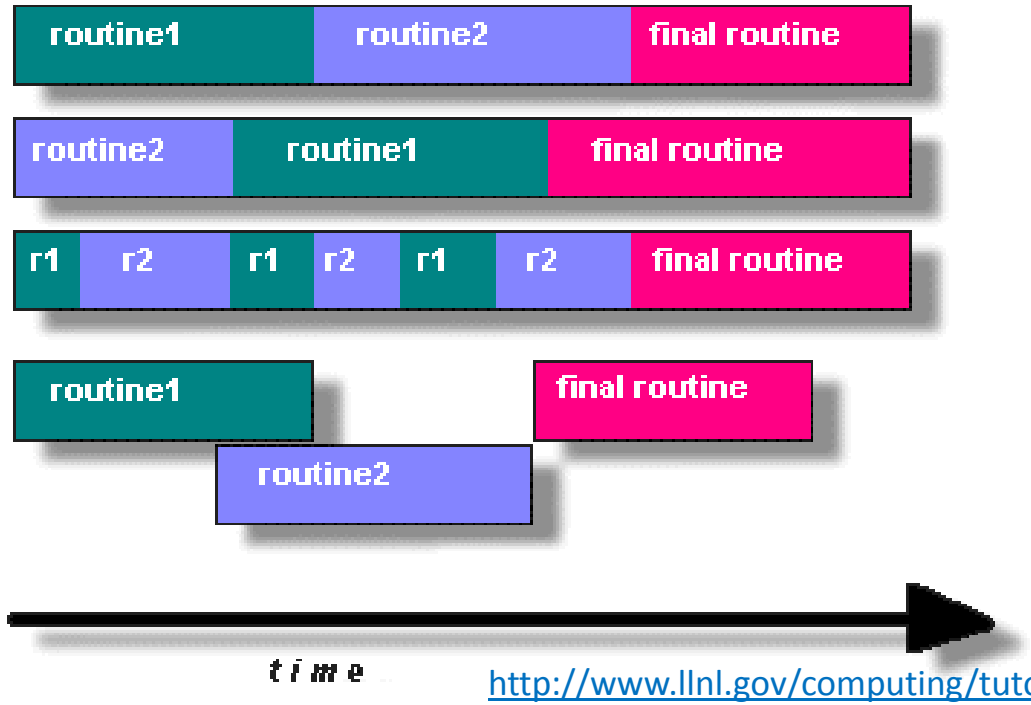
http://www.llnl.gov/computing/tutorials/pthreads/

POSIX Threads (Pthreads)

- Commonly referred to as Pthreads, POSIX has emerged as the standard threads API, supported by most vendors.
- The concepts discussed here are largely independent of the API and can be used for programming with other thread APIs (NT threads, Solaris threads, Java threads, etc.) as well.
- 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.

Why Pthreads?

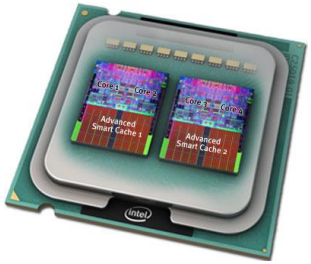
- In general though, in order for a program to take advantage of Pthreads, it must be able to be organized into discrete, independent tasks which can execute concurrently. For example, if routine1 and routine2 can be interchanged, interleaved and/or overlapped in real time, they are candidates for threading.



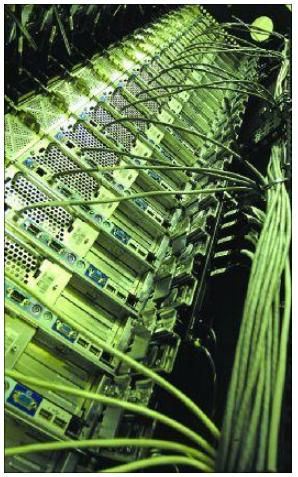
http://www.llnl.gov/computing/tutorials/pthreads/

Why Pthreads?

- The primary motivation for using Pthreads is to realize potential program performance gains.
- When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.
- On modern, multi-cpu machines, Pthreads are ideally suited for parallel programming, and whatever applies to parallel programming in general, applies to parallel Pthreads programs.



Intel Quad Core
(Threads)



Cluster
(MPI)

Pthreads API

- The Pthreads API can be informally grouped into three major classes:
  1. ***Thread Management***: functions that work directly on threads, i.e., creating, detaching, joining, etc. These include functions to set/query thread attributes (joinable, scheduling etc.)

  2. ***Mutexes***: functions that deal with synchronization, called a "mutex", which include functions for creating, destroying, locking and unlocking mutexes. They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.

  3. ***Condition Variables***: functions that address communications between threads that share a mutex. They are based upon programmer specified conditions. This class includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.

Naming Conventions

- All functions in the thread library begin with **pthread_**

Routine Prefix	Functional Group
pthread_	Threads themselves and misc 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

- The Pthreads API contains over 60 routines.
- For portability, the ***pthread.h*** header file should be included in each source file using the Pthreads library.

http://www.llnl.gov/computing/tutorials/pthreads/

Compiling Threaded Programs

- All functions in the thread library begin with **pthread_**

Compiler / Platform	Compiler Command	Description
IBM AIX	xlc_r / cc_r	C (ANSI / non-ANSI)
	xlC_r	C++
	xlf_r -qnosave xlf90_r -qnosave	Fortran - using IBM's Pthreads API
INTEL Linux	icc -pthread	C
	icpc -pthread	C++
PathScale Linux	pathcc -pthread	C
	pathCC -pthread	C++
PGI Linux	pgcc -lpthread	C
	pgCC -lpthread	C++
GNU Linux, AIX	**gcc -pthread**	**GNU C**
	g++ -pthread	**GNU C++**

http://www.llnl.gov/computing/tutorials/pthreads/

Creating Threads

- Routines:
  - **pthread_create** (thread, attr, start_routine, arg)
  - **pthread_attr_init** (attr)
  - **pthread_attr_destroy** (attr)

  - 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.

http://www.llnl.gov/computing/tutorials/pthreads/

Creating 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.

- The maximum number of threads that may be created by a process is implementation dependent.

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

Terminating Threads

- There are several ways in which a Pthread may be terminated:
  - The thread returns from its starting routine (the main routine for the initial thread).
  - The thread makes a call to the **pthread_exit()** routine.
  - The thread is cancelled by another thread via the **pthread_cancel()**routine.
  - The entire process is terminated due to a call to either the exec or exit routines.

- **pthread_exit()** is used to explicitly exit a thread. Typically, the pthread_exit() routine is called after a thread has completed its work and is no longer required to exist.

Terminating Threads

- If main() finishes before the threads it has created, and exits with pthread_exit(), the other threads will continue to execute. Otherwise, they will be automatically terminated when main() finishes.

- The programmer may optionally specify a termination *status*, which is stored as a void pointer for any thread that may join the calling thread.

- Cleanup: the pthread_exit() routine does not close files; any files opened inside the thread will remain open after the thread is terminated.

Example: Thread Creation

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   int tid;
   tid = (int)threadid;
   printf("Hello World! It's me, thread #%d!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %d\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);
      }
   }
   pthread_exit(NULL);
}
```

- 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()**.

http://www.llnl.gov/computing/tutorials/pthreads/

Example: Argument Passing

- 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 ***).

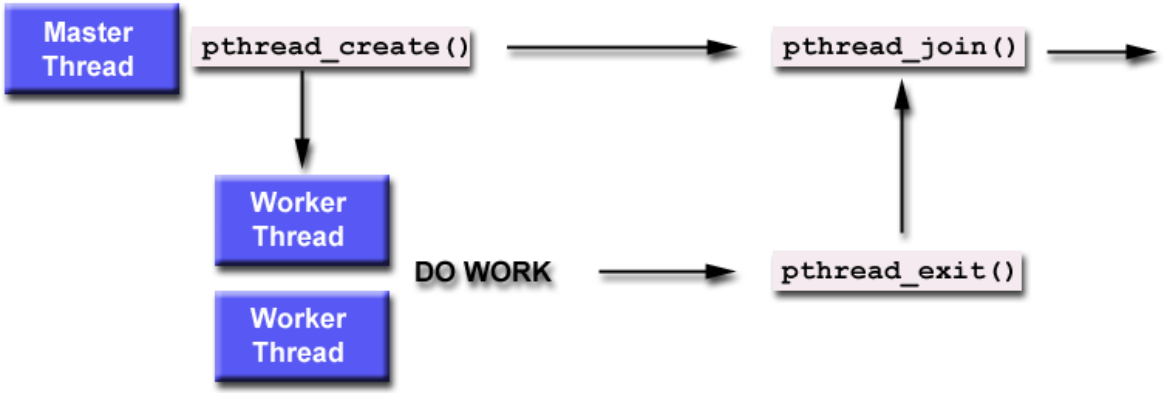
```
int *taskids[NUM_THREADS];

for(t=0; t<NUM_THREADS; t++)
{
   taskids[t] = (int *) malloc(sizeof(int));
   *taskids[t] = t;
   printf("Creating thread %d\n", t);
   rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) taskids[t]);
   ...
}
```

http://www.llnl.gov/computing/tutorials/pthreads/

Thread Management

- Routines:
  - **pthread_join**(threadid,status)
  - **pthread_detach**(threadid,status)
  - **pthread_attr_setdetachstate**(attr,detachstate)
  - **pthread_attr_getdetachstate**(attr,detachstate)
- "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.

http://www.llnl.gov/computing/tutorials/pthreads/

Joining and Detaching Threads

Joinable 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. However, not all implementations may follow this.

- 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:
  - Declare a pthread attribute variable of the ***pthread_attr_t*** data type
  - Initialize the attribute variable with **pthread_attr_init**()
  - Set the attribute detached status with **pthread_attr_setdetachstate**()
  - When done, free library resources used by the attribute with **pthread_attr_destroy**()

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.

http://www.llnl.gov/computing/tutorials/pthreads/

Stack Management

Routines

- **pthread_attr_getstacksize**(attr, stacksize)
- **pthread_attr_setstacksize**(attr, stacksize)
- **pthread_attr_getstackaddr**(attr, stackaddr)
- **pthread_attr_setstackaddr**(attr, stackaddr)

Preventing 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.

http://www.llnl.gov/computing/tutorials/pthreads/

Stack Management Example

```
#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;
  size_t mystacksize;

  pthread_attr_getstacksize (&attr, &mystacksize);
  printf("Thread d%: stack size = %li bytes \n", threadid,
      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, 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 %d threads.\n", t);
  pthread_exit(NULL);
}
```

http://www.llnl.gov/computing/tutorials/pthreads/