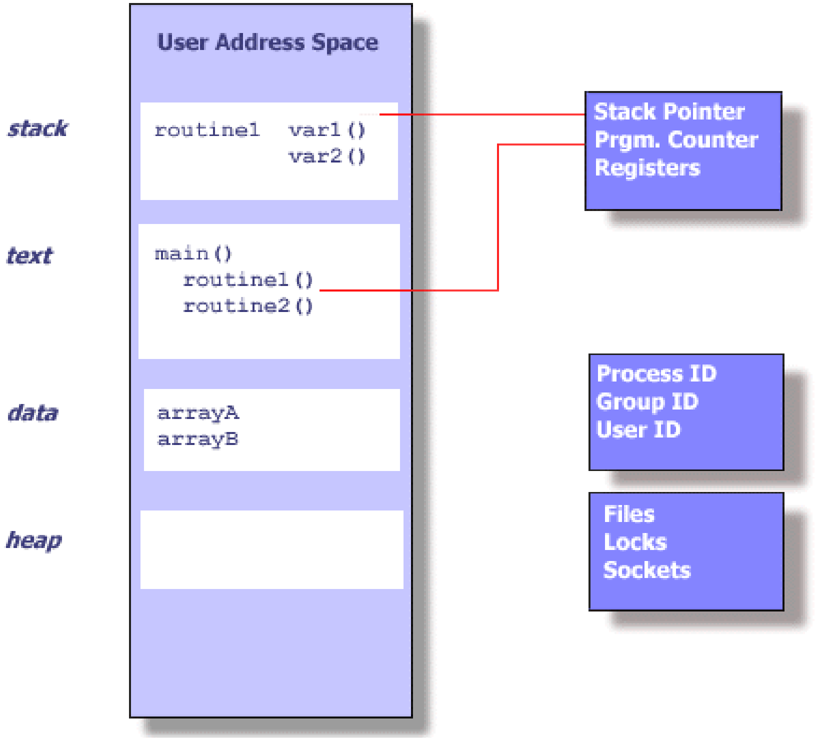
PThreads

Thanks to LLNL for their tutorial from which these slides are derived

http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html

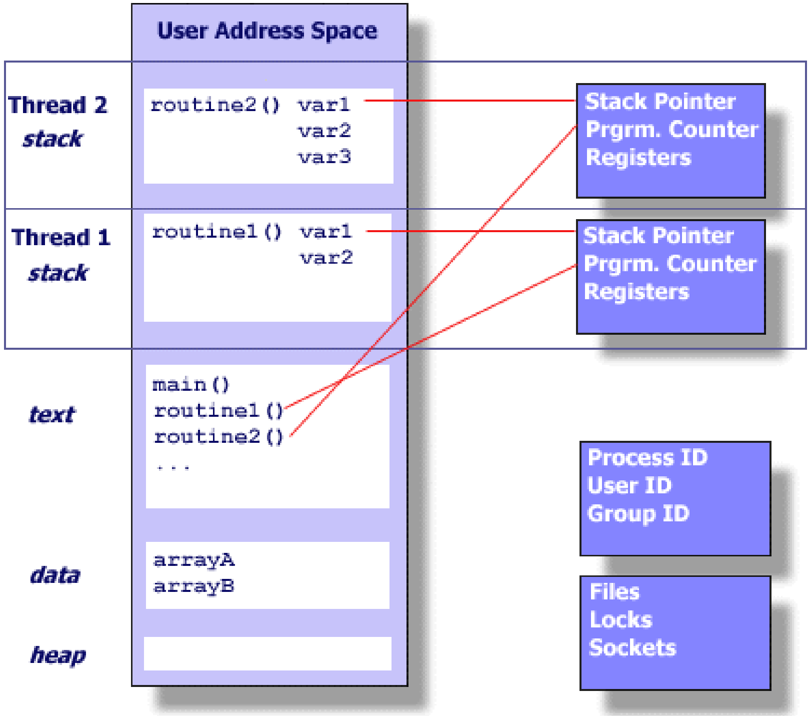
Processes and threads

- Understanding what a thread means knowing the relationship between a process and a thread. A process is created by the operating system.
  - Processes contain information about program resources and program execution state, including:
    - Process ID, process group ID, user ID, and group ID, address space
    - Environment, working directory
    - Program instructions, registers, stack, heap
    - File descriptors, inter-process communication tools (such as message queues, pipes, semaphores, or shared memory), signal actions
    - Shared libraries



Processes and threads, cont.

- Threads use and exist within these process resources, yet are able to be scheduled by the operating system and run as independent entities within a process
- A thread can possess an independent flow of control and be schedulable because it maintains its own:
  - –Stack pointer
  - –Registers
  - –Scheduling properties (such as policy or priority)
  - –Set of pending and blocked signals
  - –Thread specific data.



Processes and threads, cont.

- A process can have multiple threads, all of which share the resources within a process and all of which execute within the same address space
- Within a multi-threaded program, there are at any time multiple points of execution
- 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

What are Pthreads?

- Historically, hardware vendors have implemented their own proprietary versions of threads.
  - Standardization required for portable multi-threaded programming
  - For Unix, this interface specified by the IEEE POSIX 1003.1c standard (1995).
    - Implementations of this standard are called POSIX threads, or Pthreads.
    - Most hardware vendors now offer Pthreads in addition to their proprietary API's
    - Pthreads are defined as a set of C language programming types and procedure calls, implemented with a `pthread.h` header/include file and a thread library
  - Multiple drafts before standardization -- this leads to problems

Posix Threads - 3 kinds

- "Real" POSIX threads, based on the IEEE POSIX 1003.1c-1995 (also known as the ISO/IEC 9945-1:1996) standard, part of the ANSI/IEEE 1003.1, 1996 edition, standard. POSIX implementations are, not surprisingly, the standard on Unix systems. POSIX threads are usually referred to as Pthreads.
- DCE threads are based on draft 4 (an early draft) of the POSIX threads standard (which was originally named 1003.4a, and became 1003.1c upon standardization).
- Unix International (UI) threads, also known as Solaris threads, are based on the Unix International threads standard (a close relative of the POSIX standard).

What are threads used for?

- Tasks that may be suitable for threading include tasks that
  - Block for potentially long waits (Tera MTA/HEP)
  - **Use many CPU cycles**
  - Must respond to asynchronous events
  - Are of lesser or greater importance than other tasks
  - **Are able to be performed in parallel with other tasks**
- Note that numerical computing and parallelism are a small part of what parallelism is used for

Three classes of Pthreads routines

- ***Thread management:*** creating, detaching, and joining threads, etc. They include functions to set/query thread attributes (joinable, scheduling etc.)
- ***Mutexes:*** Mutex functions provide for creating, destroying, locking and unlocking mutexes. They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
- ***Condition variables:*** The third class of functions 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.

Creating threads

- **pthread_create** (thread, attr, start_routine, arg)
- This routine creates a new thread and makes it executable. Typically, threads are first created from within `main()` inside a single process.
  - Once created, threads are peers, and may create other threads
  - The pthread_create subroutine returns the new thread ID via the *thread* argument. This ID should be checked to ensure that the thread was successfully created
  - The *attr* parameter is used to set thread attributes. Can be an object, or NULL for the default values
  - *start_routine* is the C routine that the thread will execute once it is created. A single argument may be passed to *start_routine* via *arg* as a void pointer.
  - The maximum number of threads that may be created by a process is implementation dependent.
- Question: After a thread has been created, how do you know when it will be scheduled to run by the operating system...especially on an SMP machine? **You don't!**

Terminating threads

- How threads are terminated:
  - The thread returns from its starting routine (the main routine for the initial thread)
  - The thread makes a call to the `pthread_exit` subroutine
  - The thread is canceled by another thread via the `pthread_cancel` routine
    - Some problems can exist with data consistency
  - The entire process is terminated due to a call to either the `exec` or `exit` subroutines.

pthread_exit(status)

- `pthread_exit()` routine is called after a thread has completed its work and is no longer required to exist
- 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
  - `pthread_exit()` routine does not close files
  - Recommended to use `pthread_exit()` to exit from all threads...especially `main()`.

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
void* PrintHello(void *threadid){
   printf("\n%d: Hello World!\n", threadid);
   pthread_exit(NULL);
}
int main (int argc, char *argv[]){
   pthread_t threads[NUM_THREADS];
   int args[NUM_THREADS];
   int rc, t;
   for(t=0;t < NUM_THREADS;t++){
      printf("Creating thread %d\n", t);
      args[t] = t;
      rc = pthread_create(&threads[t], NULL, PrintHello,
                                (void *) args[t]);
      if (rc) {
         printf("ERROR; return code from pthread_create() is
   %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

Passing arguments to a thread

- Thread startup is non-deterministic
- It is implementation dependent
- If we do not know when a thread will start, how do we pass data to the thread knowing it will have the right value at startup time?
  - Don't pass data as arguments that can be changed by another thread
  - In general, use a separate instance of a data structure for each thread.

Passing data to a thread (a simple integer)

```
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 *) &t);
   …
}
```

time

Thread 0	Thread q
t = 0;	
pthread_create(..., f, t);	
t = 1;	*thread spawn*
pthread_create(..., f, t);	f(t);
t = 2;	x = t;

What is the value of t that is used in this call to f?

The value is indeterminate.

In general

- Unless you know something is read-only
  - Only good way to know what the value is when the thread starts is to have a separate copy of argument for each thread.
  - Complicated data structures may share data at a deeper level
    - This not so much of a problem with numerical codes since the data structures are often simpler than with integer codes (although not true with sparse codes and complicated meshes)

Thread identifiers

- **pthread_t pthread_self** ()
  - `pthread_self()` routine returns the unique, system assigned thread ID of the calling thread
- int pthread_equal (thread1, thread2)
  - `pthread_equal()` routine compares two thread IDs.
    - 0 if different, non-zero if the same.
    - Note that for both of these routines, the thread identifier objects are ***opaque***
    - 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_join** (threadId, status)
- 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 specified through `pthread_exit()`, in the *status* parameter
  - This can be a void pointer and point to anything
- It is impossible to join a detached thread (discussed next)

Detatched threads are not joinable

- **`pthread_attr_init (attr)`**
- **`Pthread_attr_setdetachstate(attr, detachstate)`**
- **`Pthread_attr_getdetachstate(attr, detatchstate)`**
- **`Pthread_attr_destroy (attr)`**
- **`Pthread_detach (threadid, status)`**
- According to the Pthreads standard, all threads should default to joinable, but older implementations may not be compliant.

See PThreadsAttr.pdf (next page)

```
include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 3
void *BusyWork(void *null) {
   int i;
   double result=0.0;
   for (i=0; i < 1000000; i++) {
      result = result + (double)random();
   }
   printf("result = %e\n",result);
   pthread_exit((void *) 0);
}
```

```
int main (int argc, char *argv[]) {
  pthread_t thread[NUM_THREADS];
  pthread_attr_t attr;
  int rc, t, 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("Creating thread %d\n", t);
    rc = pthread_create(&thread[t], &attr, BusyWork, NULL);
    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], (void **)&status);
     if (rc) {
        printf("ERROR; return code from pthread_join() is %d\n", rc);
        exit(-1);
     }
     printf("Completed join with thread %d status= %d\n",t, status);
  }
  pthread_exit(NULL);
}
```

this is ok

Locks in pthreads: allow critical sections to be formed

- Unlike Java, locks and objects are disjoint because unlike Java, can't assume you have objects
- **pthread_mutex_init** (mutex, attr)
- pthread_mutex_destroy (mutex)
- pthread_mutexattr_init (attr)
- pthread_mutexattr_destroy (attr)

Using locks

- **pthread_mutex_lock** (mutex)
  – Acquire lock if available
  – Otherwise wait until lock is available
- pthread_mutex_trylock (mutex)
  – Acquire lock if available
  – Otherwise return lock-busy error
- pthread_mutex_unlock (mutex)
  – Release the lock to be acquired by another pthread_mutex_lock or trylock call
  – Cannot make assumptions about which thread will be woken up
- See `http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html` for an example

Using barriers

```
pthread_barrier_t barrier;
pthread_barrierattr_t attr;
    unsigned count;
    int ret;
    ret = pthread_barrierattr_init(&attr);
    ret = pthread_barrier_init(&barrier, &attr, count);
    ret = pthread_barrier_wait(&barrier);
    ret = pthread_barrier_destroy(&barrier);
```

The only barrier attribute is the process shared attribute. The default is PTHREAD_PROCESS_PRIVATE: only threads that belong to the process that created the barrier can wait on a barrier with this attribute. PTHREAD_PROCESS_SHARED allows threads of any process that accesses the memory the barrier is allocated in to access the barrier.

Using *condition* variables

- Allows one thread to signal to another thread that a condition is true
- Prevents programmer from having to loop on a mutex call to poll if a condition is true.

Condition variable scenario

- **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**
- Execute up to where some condition should be true (e.g. count = some value)
- Lock associated mutex and check value of a global variable (e.g. count). If valid value:
- Call `pthread_cond_wait()`
  - performs a blocking wait for signal from Thread-B.
  - call to `pthread_cond_wait()` unlocks the associated mutex variable so Thread-B can use it.
  - **Wake up on signal -- 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 on
  - Check if the value of the global Thread-A wait variable fulfills the desired condition
  - **signal Thread-A.**
- Unlock mutex
- Continue

OpenMP <--> Pthreads

- omp parallel for

  The programmer must partition the loop iteration space and give different parts of the iteration space to different threads. Need a barrier at the end

- omp parallel

  Have the appropriate number of threads execute the task in the parallel region

- omp parallel sections

  Code in each section sent to a different thread with a barrier at the end

- tasks

  Just spawn a thread with the task as the called routine.

Summary

- OpenMP build on Pthreads
- Consistency model for Pthreads between synchronization and thread creation/destruction calls is up to the individual compiler