POLITECNICO DI MILANO

Thread Posix

Algoritmi e Calcolo Parallelo

Daniele Loiacono

References

- The material in this set of slide is taken from tutorials by Blaise Barney from the Lawrence Livermore National Laboratory and from slides of prof. Lanzi (Informatica B, A.A. 2009/2010)

- **POSIX Threads Programming**
  Blaise Barney, Lawrence Livermore National Laboratory
  https://computing.llnl.gov/tutorials/pthreads/

- **Advanced Linux Programming**
  http://www.advancedlinuxprogramming.com/

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.
- 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.

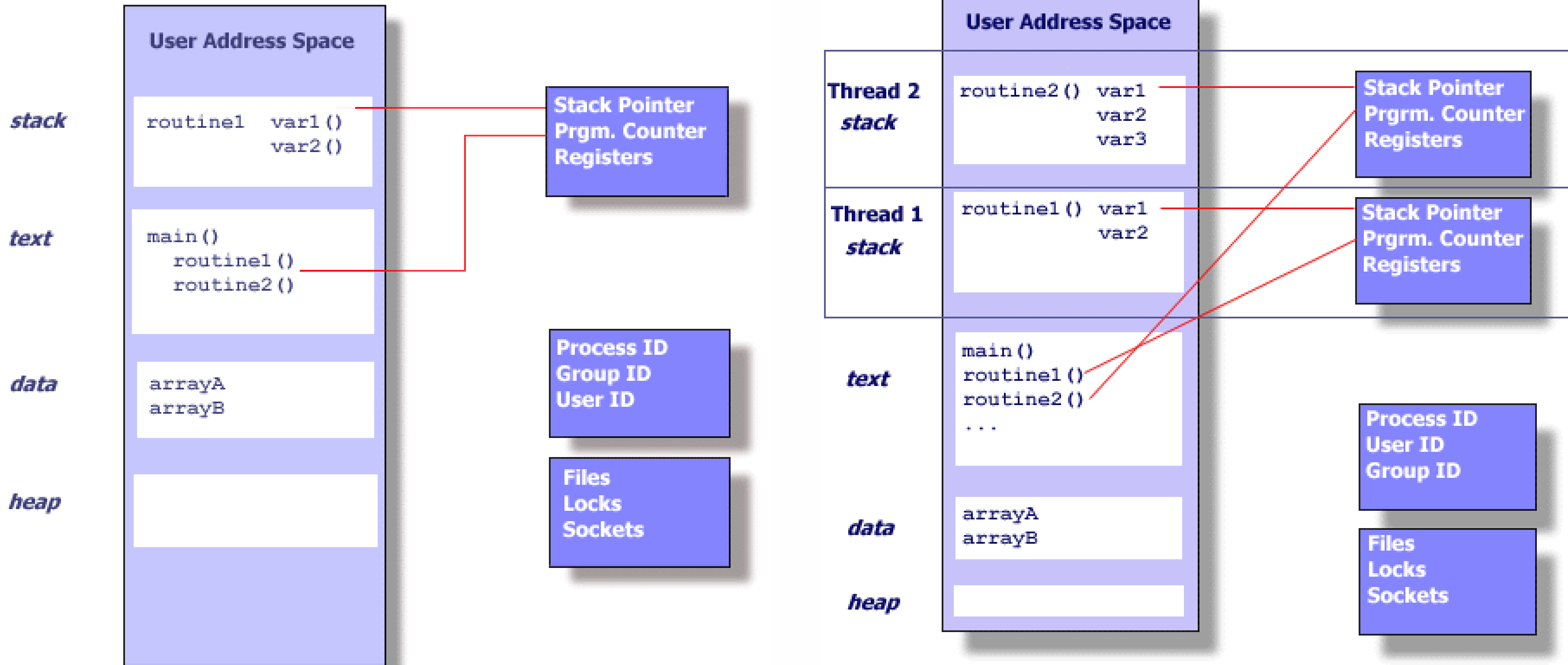
Unix Processes

- Before understanding a thread, one first needs to understand a UNIX process

- 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, Environment, Working directory, Program instructions, etc.

Unix Processes vs Threads



Threads

- Threads exists within a process and uses the process resources
- Run as independent entities that duplicate only the bare essential resources that enable them to exist as executable code
- May share the process resources with other threads that act equally independently (and dependently)
- Dies if the parent process dies - or something similar
- 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. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.

- To take full advantage of the capabilities provided by threads, a standardized programming interface was required.

- The POSIX standard has continued to evolve and undergo revisions, including the Pthreads specification. The latest version is known as IEEE Std 1003.1, 2004 Edition.

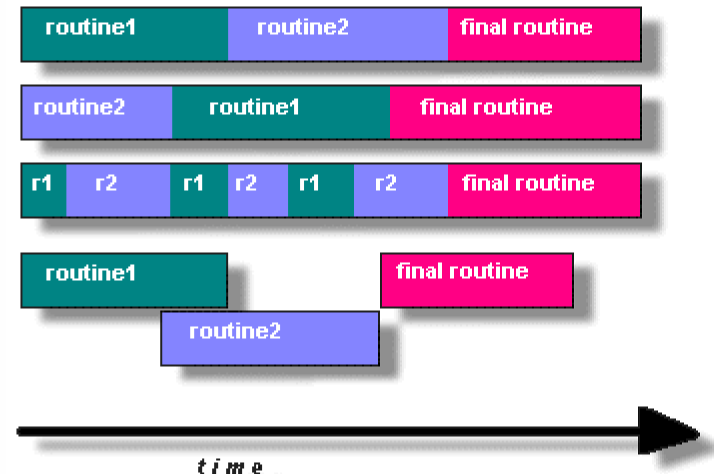
- Set of C language programming types and procedure calls
  - pthread.h header/include file
  - thread library

Designing Threaded Programs

- 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.

- 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
  - …

- 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.

Models for Threaded Programs

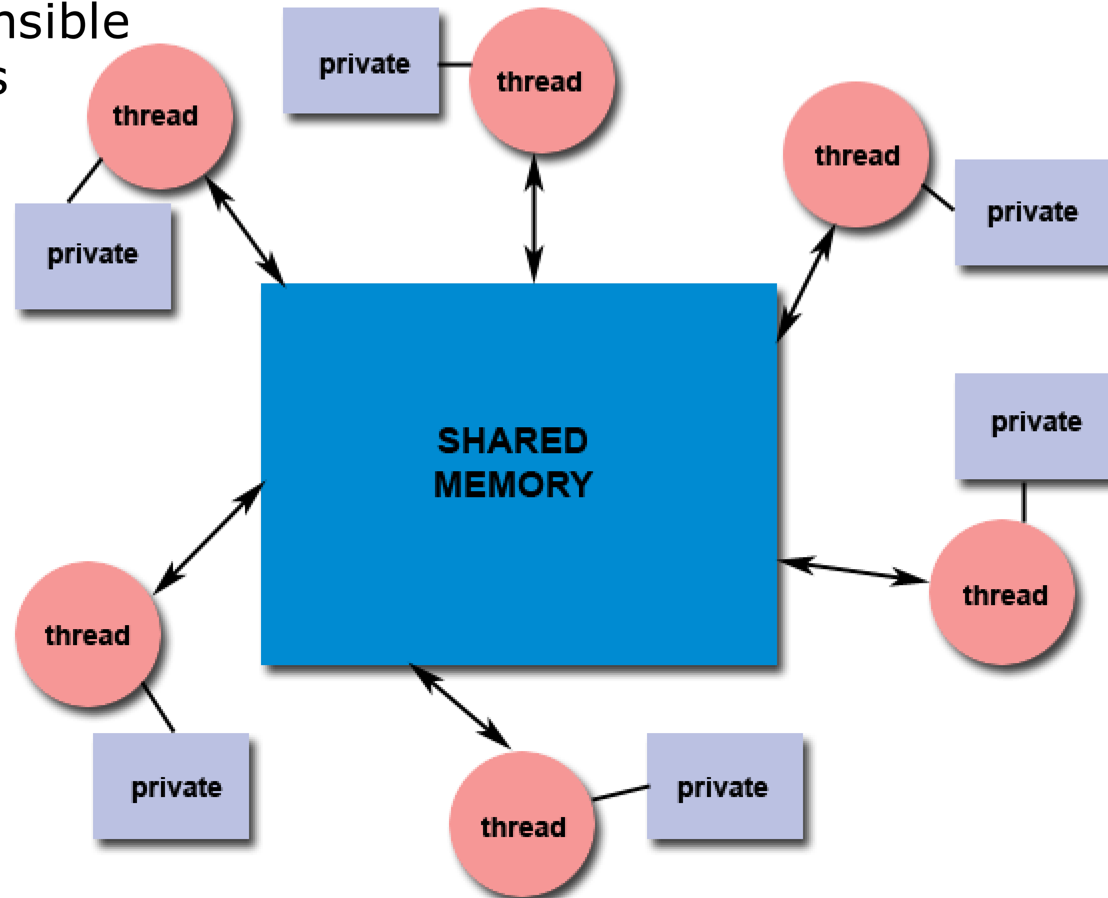
- **Manager/worker:** a single thread, the manager assigns work to other threads, the workers. Typically, the manager handles all input and parcels out work to the other tasks. At least two forms of the manager/worker model are common: static worker pool and dynamic worker pool.

- **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.

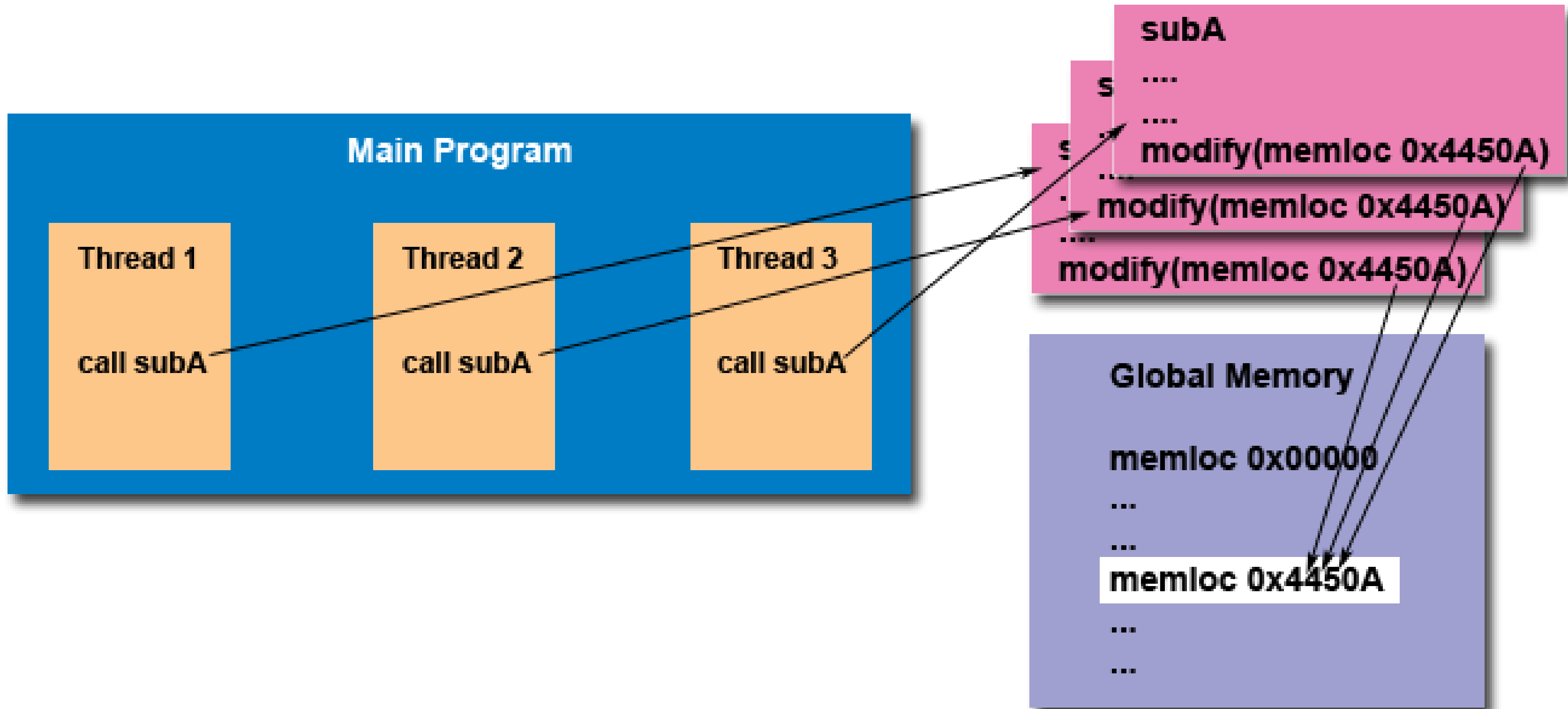
- **Peer:** similar to the manager/worker model, but after the main thread creates other threads, it participates in the work.

Shared Memory Model

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



Thread Safeness



- **Thread management:** routines that work directly on threads - creating, detaching, joining, etc. Functions to set/query thread attributes.
- **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.
- **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.
- **Synchronization:** routines that manage read/write locks and barriers.

Thread Management: Creation

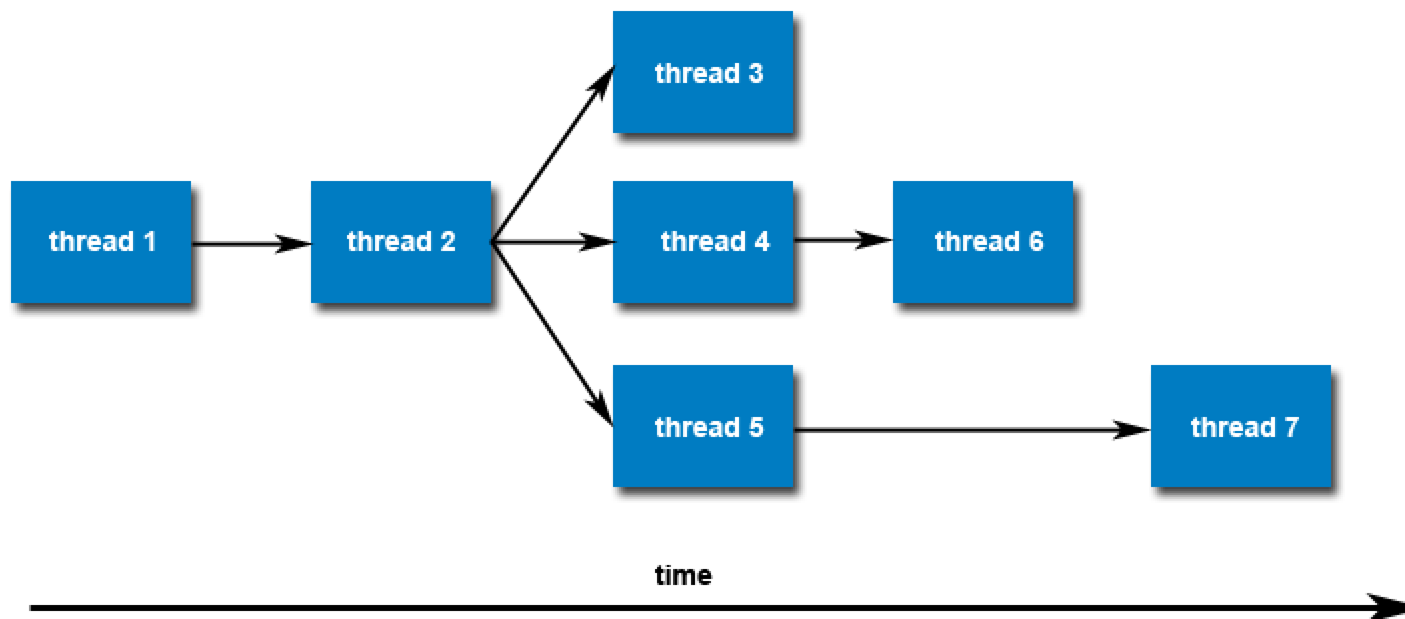
- Initially, your main() program comprises a single, default thread. All other threads must be explicitly created by the programmer.

- **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(thread, attr, start_routing, arg)**
  - **thread:** unique identifier for the new thread returned by the subroutine.
  - **attr:** Aused 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:** argument 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.

Thread Management: Creation

- Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.
- The maximum number of threads that may be created by a process is implementation dependent.



Thread Management: Termination

- 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 subroutine
  - The thread is canceled by another thread via the pthread_cancel routine (not covered here).
  - The entire process is terminated due to a call to either the exec or exit subroutines.

- 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.

Thread Management: Termination

- 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.

- 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


```
#include <pthread.h>
#include <cstdlib>
#include <iostream>
#define NUM_THREADS     5
using namespace std;

void *PrintHello(void *threadid);
int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   for(long t=0; t<NUM_THREADS; t++){
      cout << "In main: creating thread" << t << endl;
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         cout << "ERROR; return code from pthread_create() is";
         cout << rc << endl;
         exit(-1); }
   }
   pthread_exit(NULL);
}
```

Example


```
void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   cout <<  "Hello World! It's me, thread" << tid << endl;
   pthread_exit(NULL);
}
```

Example: Output of Two Runs


```
In main: creating thread 0
Hello World! It's me, thread #0!
In main: creating thread 1
Hello World! It's me, thread #1!
In main: creating thread 2
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!
```

```
In main: creating thread 0
Hello World! It's me, thread #0!
In main: creating thread 1
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!
```

Passing Arguments to Threads

- pthread_create() permits the programmer to pass one argument to the thread start routine
- When multiple arguments must be passed, a structure which contains all of the arguments must be used

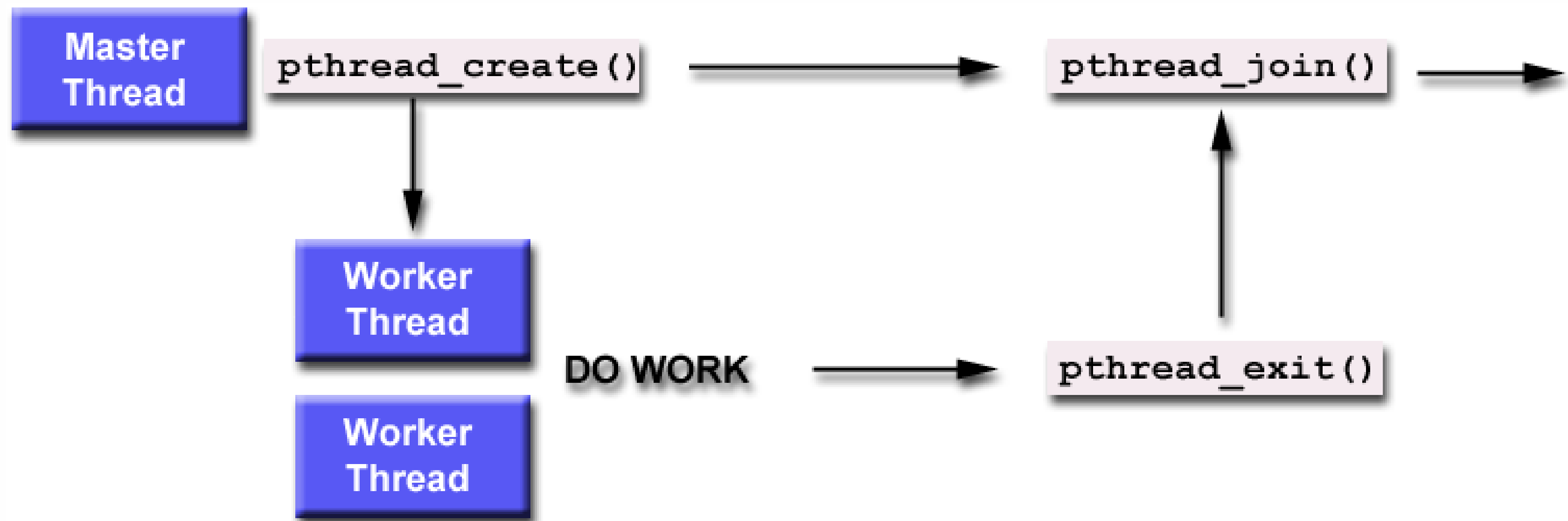
Example: Passing One Argument


```
struct thread_data{
   int  thread_id;
   int  sum;
};

void *PrintHello(void *threadarg);

int main (int argc, char *argv[])
{
   ...
   struct thread_data data[NUM_THREADS];
   ...
   thread_data_array[t].thread_id = t;
   thread_data_array[t].sum = sum;
   rc = pthread_create(&threads[t], NULL, PrintHello,
         (void *) &data[t]);
   ...
}
```

Example: Passing One Argument


```
void *PrintHello(void *arg)
{
   struct thread_data *mydata;
   mydata = (struct thread_data*) arg;

   cout <<  "Hello World! It's me, thread " << mydata->thread_id
       << " sum is " << mydata->sum << endl;
   pthread_exit(NULL);
}
```

Joining Threads



- "Joining" is one way to accomplish synchronization between threads.
- The `pthread_join()` subroutine blocks the calling thread until the specified 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.

Joinable or Not? Detaching?

- 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.

- 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

- pthread_detach() can be used to explicitly detach a thread even though it was created as joinable. There is no converse routine

Recommendations

- 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 of Joining Threads: Initialization


```
...
#define NUM_THREADS    4

void *BusyWork(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);
```

Example of Joining Threads: Creation


```
for(t=0; t<NUM_THREADS; t++) {
   cout << "Main: creating thread " << t << endl;
   rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
   if (rc) {
      cout << "ERROR; return code from pthread_create()
               is " << rc << endl;
      exit(-1);
      }
   }
```

Example of Joining Threads: Joining


```
    /* 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) {
          cout << "ERROR; return code from pthread_join()
                 is " << rc << endl;
          exit(-1);
          }
       cout << "Main: completed join with thread " << t << "having a
                status of << (long)status << endl;
       }

cout << "Main: program completed. Exiting" << endl;
pthread_exit(NULL);
}
```

Mutex Variables

- 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.
- Example: two tasks have access to the same bank account. When one task gains access, other tasks cannot access it.
- A mutex variable acts like a "lock" protecting access to a shared data resource.
- In Pthreads, one thread can lock (or own) a mutex variable at any given time.
- 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.

Mutex Variables

- 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

Creating and Destroying Mutexes

Functions

- pthread_mutex_init (mutex,attr)
- pthread_mutex_destroy (mutex)
- pthread_mutexattr_init (attr)
- pthread_mutexattr_destroy (attr)

Usage

- Mutex variables must be declared with type pthread_mutex_t, and must be initialized (statically or dinamically) before they can be used.

  pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;

  pthread_mutex_init()

- 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).

Creating and Destroying Mutexes

- 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.

Locking and Unlocking Mutexes

Functions

- pthread_mutex_lock (mutex)
- pthread_mutex_trylock (mutex)
- pthread_mutex_unlock (mutex)

Usage

- pthread_mutex_lock() 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

Example of Mutex Variables Usage: Variables & Declarations


```
#include <pthread.h>
#include <cstdlib>
#include <iostream>

using namespace std;

#define NUMTHRDS 4
#define VECLEN 100

/* Global data */
double a[VECLEN];
double b[VECLEN];
double sum;
pthread_t callThd[NUMTHRDS];
pthread_mutex_t mutexsum;

void *dotprod (void *arg);
```

Example of Mutex Variables Usage: Main (1)


```
int
main (int argc, char *argv[])
{
  long i;
  void *status;
  pthread_attr_t attr;

  /* initialize values */
  for (i = 0; i < VECLEN; i++)
    {
      a[i] = 1.0;
      b[i] = a[i];
    }

  pthread_mutex_init (&mutexsum, NULL);
```

Example of Mutex Variables Usage: Main (2)


```
   /* 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);
```

Example of Mutex Variables Usage: Main (3)


```
 /* Wait on the other threads */
for (i = 0; i < NUMTHRDS; i++)
  {
    pthread_join (callThd[i], &status);
  }

/* After joining, print out the results and cleanup */
cout << "Sum =  " << sum << endl;
pthread_mutex_destroy (&mutexsum);
pthread_exit (NULL);
}
```

Example of Mutex Variables Usage: dotprod (1)


```
void *
dotprod (void *arg)
{

  /* Define and use local variables for convenience */
  int i, start, end, len, mysum;
  long offset;
  offset = (long) arg;

  len = VECLEN / NUMTHRDS;
  start = offset * len;
  end = start + len;
```

Example of Mutex Variables Usage: dotprod (2)


```
 /*
    Perform the dot product and assign result
    to the appropriate variable in the structure.
   */
  mysum = 0;
  for (i = start; i < end; i++)
    mysum += (a[i] * b[i]);

  /*
    Lock a mutex prior to updating the value in the shared
    structure, and unlock it upon updating.
   */
  pthread_mutex_lock (&mutexsum);
  sum += mysum;
  pthread_mutex_unlock (&mutexsum);

  pthread_exit ((void *) 0);
}
```