pthread Basics

Notes from the web

Threads

- In shared memory multiprocessor architectures, such as SMPs, threads can be used to implement parallelism.

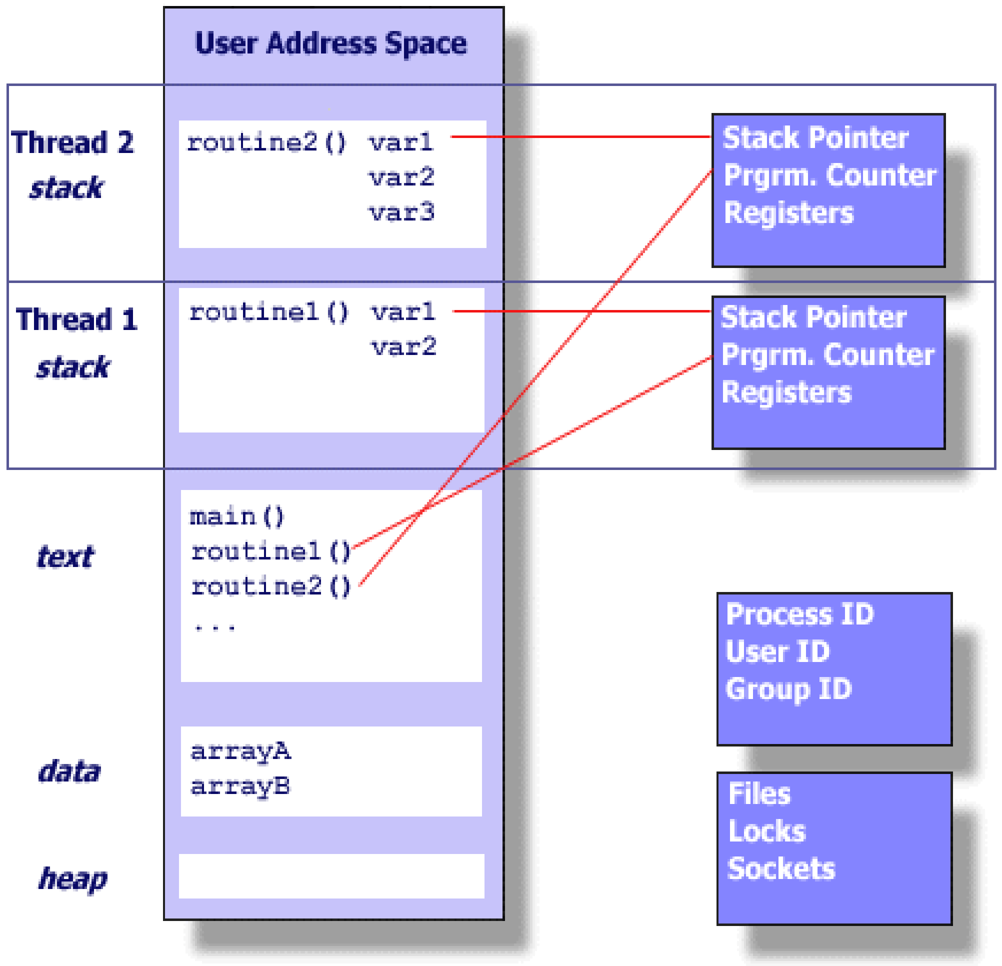
- For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard.

- Implementations that adhere to this standard are referred to as **POSIX threads**, or **Pthreads**.

What is a Thread?

- Independent flow of control possible because a thread maintains its own:
  - Program Counter
  - Stack pointer
  - Registers

  - Scheduling properties (such as policy or priority)
  - Set of pending and blocked signals
  - Thread specific data.



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
- 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.

Thread Consequences

- 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
  - Therefore requires explicit synchronization by the programmer to avoid races (Thread Safe)

What are Pthreads?

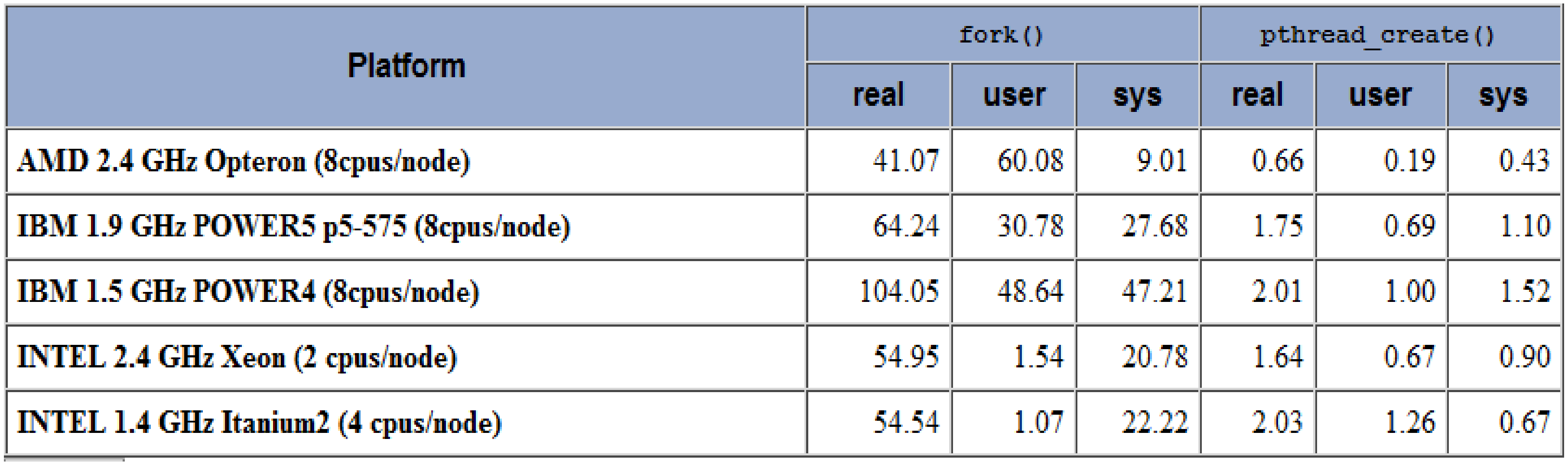
- Pthreads are defined as a set of C language programming types and procedure calls
- Implemented with a pthread.h header/include file and a thread library

Why Pthreads?

- The primary motivation is to realize parallel program performance gains
- When compared to the cost of creating and managing a process, a thread can be created with much less OS overhead
- Managing threads requires fewer system resources than managing processes
- All threads within a process share the same address space
- Inter-thread communication is more efficient and, in many cases, easier to use than inter-process communication

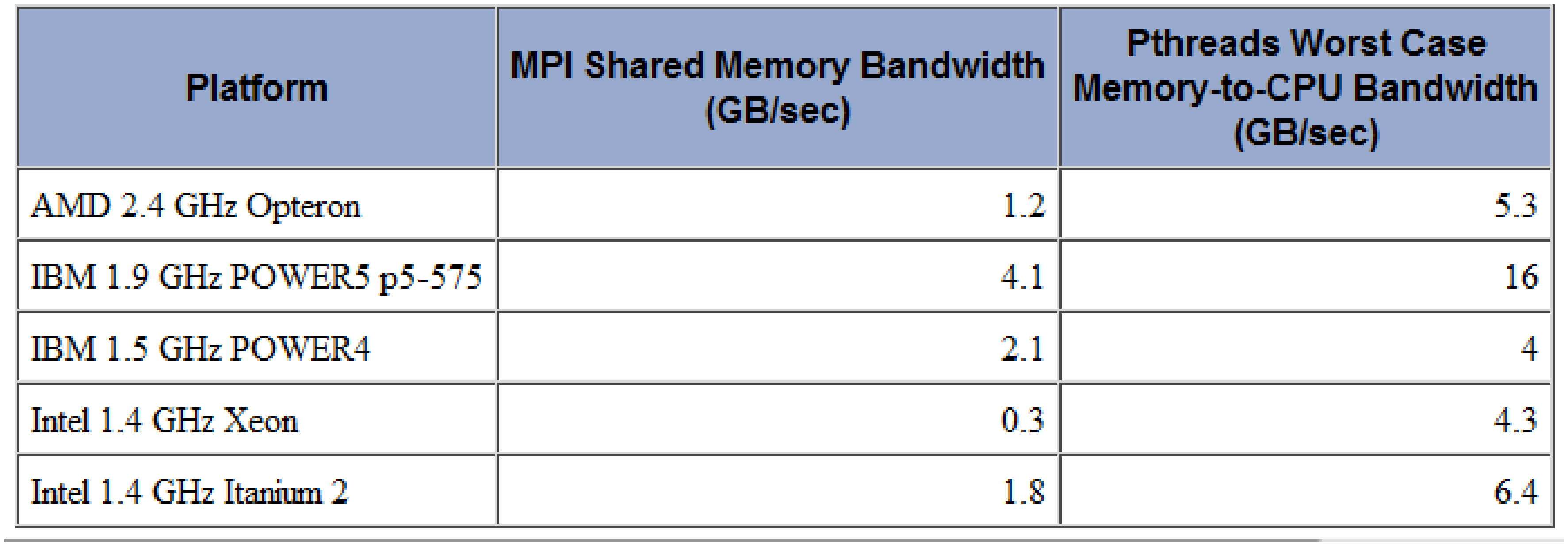
Example

- The following table compares timing results for the **fork()** subroutine and the **pthreads_create()** subroutine
- Timings reflect 50,000 process/thread creations, units are in seconds, no optimization flags



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

Example



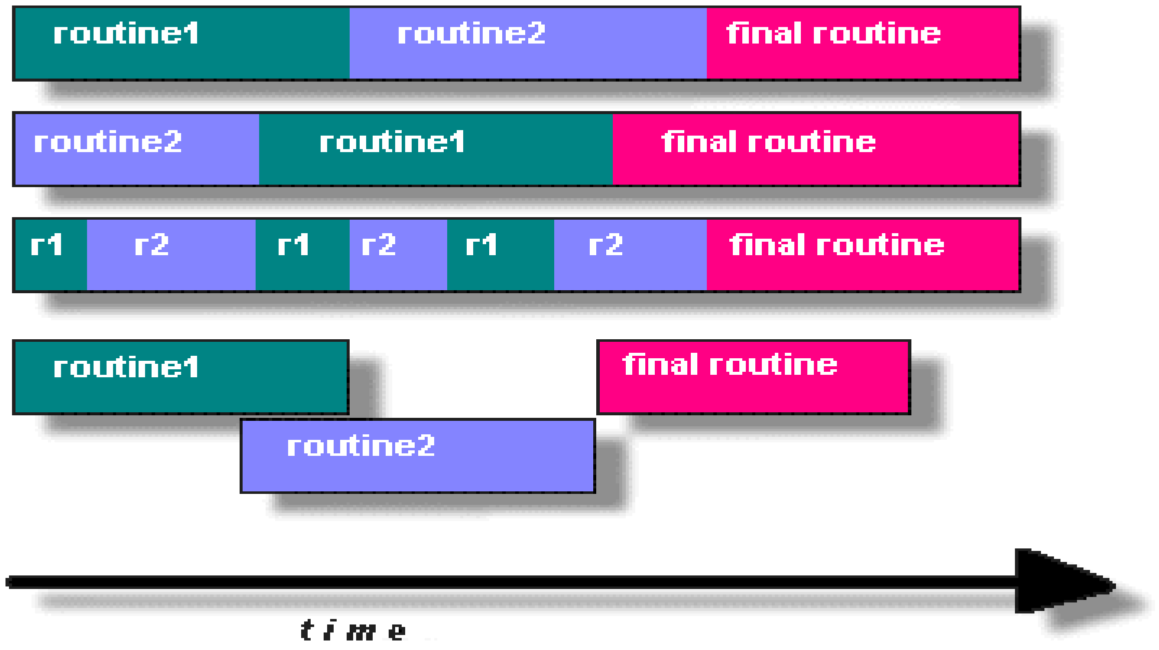
Platform	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
AMD 2.4 GHz Opteron	1.2	5.3
IBM 1.9 GHz POWER5 p5-575	4.1	16
IBM 1.5 GHz POWER4	2.1	4
Intel 1.4 GHz Xeon	0.3	4.3
Intel 1.4 GHz Itanium 2	1.8	6.4

Parallel Programming

- 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
  - ...

Parallel Programming

- To take advantage of Pthreads (parallelism), a program 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.



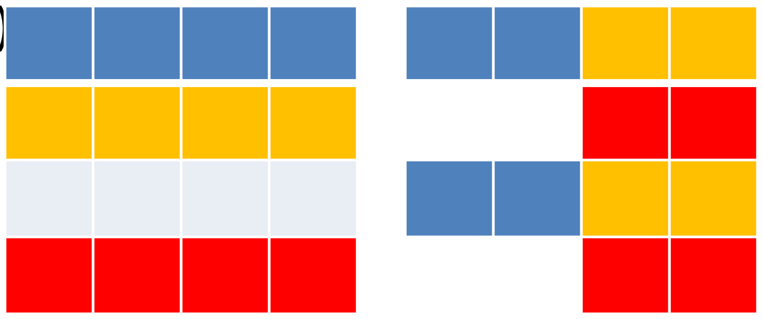
Parallel Programming

- 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
  - …
  - …

Parallel Programming

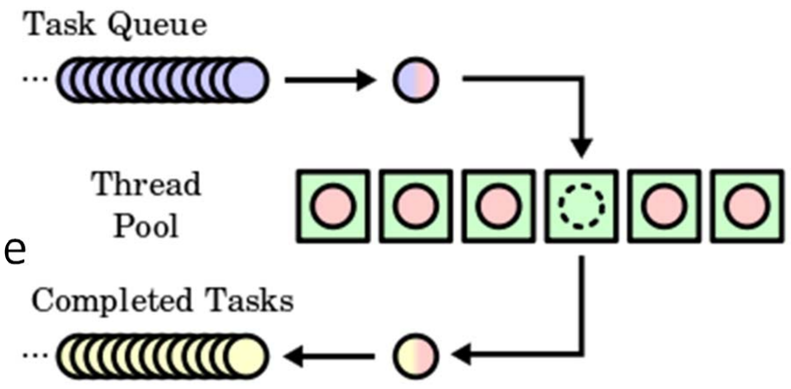
- Several common models for threaded programs exist:
- ***Manager/worker:***
  - a single thread, the ***manager,*** assigns work to other threads, the ***workers***
  - Typically, the manager handles all input and parcels out work to the other tasks
  - At least two forms of the manager/worker model are common:

1. static worker pool (many different options for load balancing but fixed) tradeoff memory locality(performance) for load balancing



2. dynamic worker pool (χρήση queue)

- When job gets done seek next task
- Granularity of tasks vs overhead of queue management



Parallel Programming

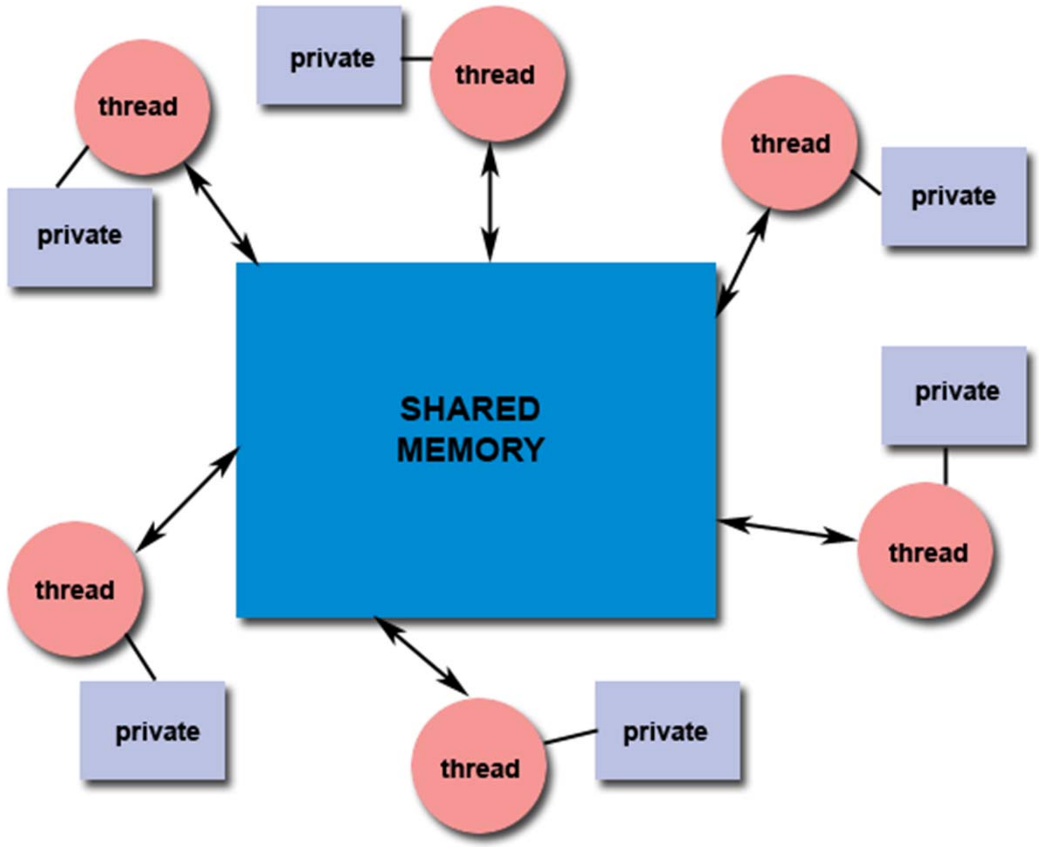
- Several common models for threaded programs exist:

- ***Pipeline:***
  - a task is broken into a series of sub-operations
  - each sub-operation 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

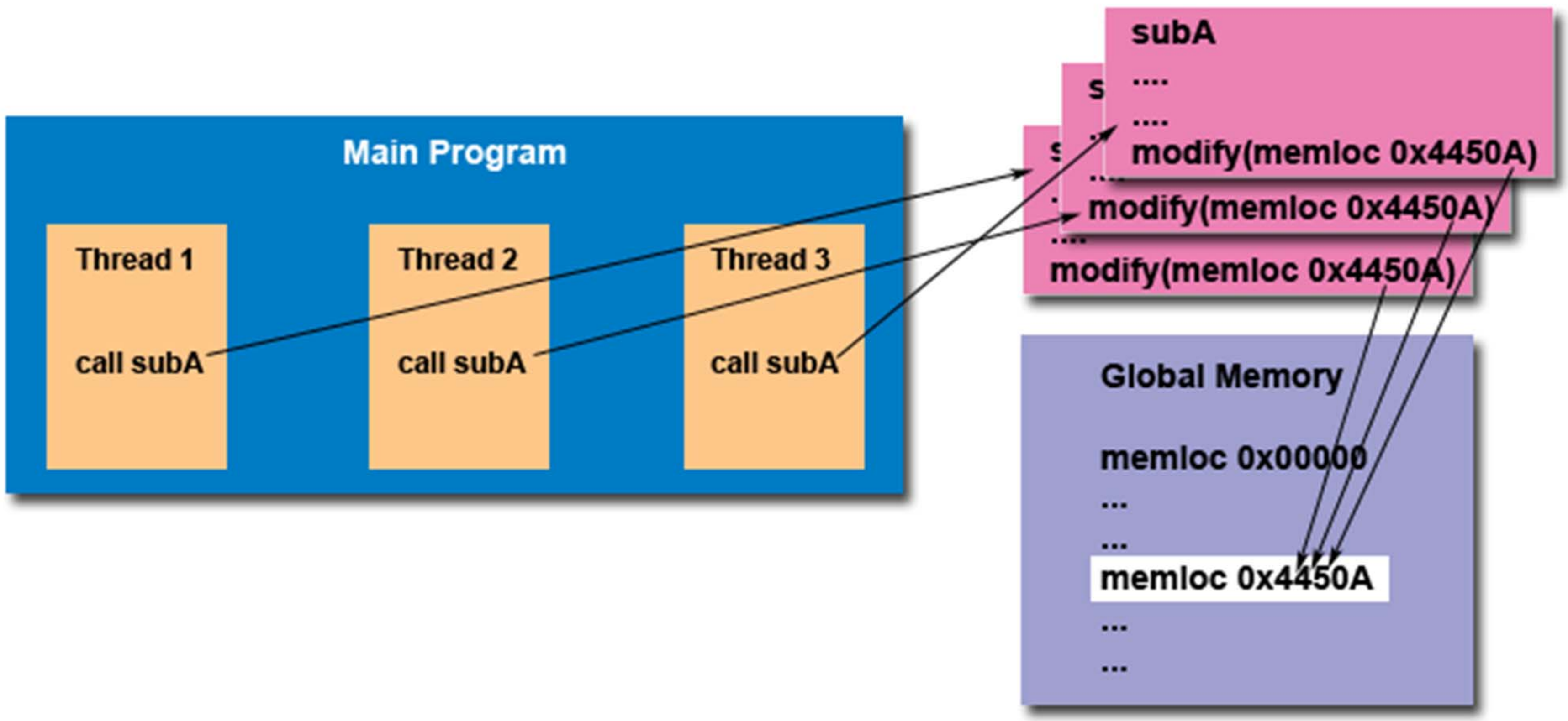
- All threads have access to the same global, shared memory
- Threads also have their own private data
- Programmers are responsible for synchronizing access (protecting) globally shared data.



Thread-safeness

- Thread-safeness: in a nutshell, refers an application's ability to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions
- Example: an 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. (potential for data race)

Thread-safeness



The Pthreads API

- The subroutines which comprise the Pthreads API can be informally grouped into three major classes:

- ***Thread management:***
  - Work directly on threads:
  - Creating, detaching, joining, etc.
  - They include functions to set/query thread attributes (joinable, scheduling etc.)

- ***Mutexes:***
  - Deal with synchronization via a "***mutex***“
  - "mutex“ is an abbreviation for "mutual exclusion“
  - 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.

The Pthreads API

- The subroutines which comprise the Pthreads API can be informally grouped into three major classes:
- ***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 identifiers in the threads library begin with **pthread_**

- More in the lab
- Useful examples

The Pthreads API

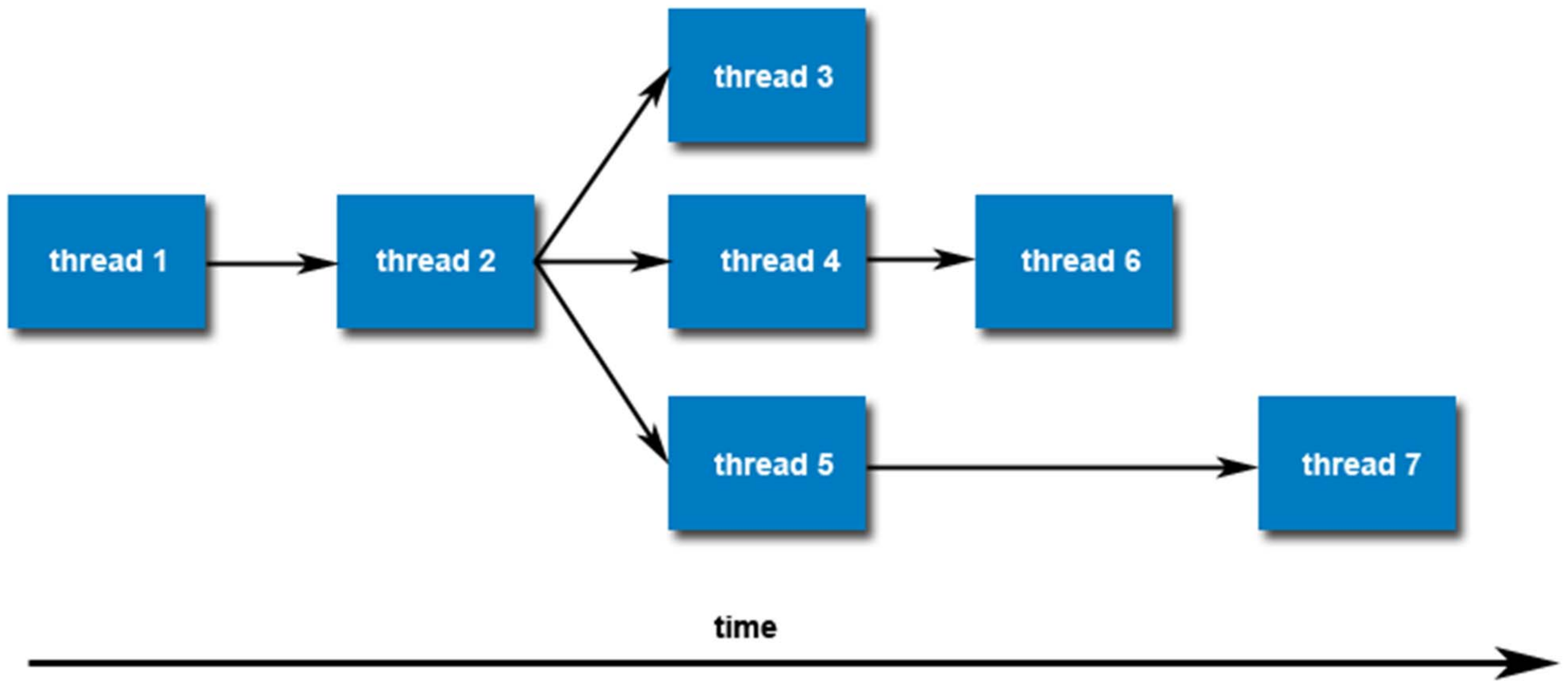
- Initially, your main() program comprises a single, default thread. All other threads must be explicitly created by the programmer

```
pthread_create(thread,attr,start_routine,arg)
```

- **`pthread_create`** creates a new thread and makes it executable
- This routine can be called any number of times from anywhere within your code

Pthreads

- Once created, threads are peers, and may create other threads



Pthread Execution

- **Question**:
  After a thread has been created, how do you know when it will be scheduled to run by the operating system?

- **Answer**:

  Unless you are using the Pthreads scheduling mechanism, it is up to the implementation and/or operating system to decide where and when threads will execute.

  Robust programs should not depend upon threads executing in a specific order.

Thread 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
- Some of these attributes will be discussed later

Terminating Threads

- `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

```
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);
       }
    }
  pthread_exit(NULL);
}
```

Example

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.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);
}
```

Example

Output:

```
In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
In main: creating thread 3
In main: creating thread 4
Hello World! It's me, thread #0!
Hello World! It's me, thread #1!
Hello World! It's me, thread #2!
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!
```

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:
  - create a structure which contains all of the arguments
  - then pass a pointer to that structure in the `pthread_create()` routine.
  - All arguments must be passed by reference and cast to (void *)
- **Question**:
  How can you safely pass data to newly created threads, given their non-deterministic start-up and scheduling?
- **ANSWER:**
  Make sure that all passed data is thread safe - that it can not be changed by other threads
- The three examples that follow demonstrate what not and what to do.

Argument Passing Example 1

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.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS   8

char *messages[NUM_THREADS];

void *PrintHello(void *threadid)
{
   int *id_ptr, taskid;

   sleep(1);
   id_ptr = (int *) threadid;
   taskid = *id_ptr;
   printf("Thread %d: %s\n", taskid, messages[taskid]);
   pthread_exit(NULL);
}
```

Argument Passing Example 1

```
int main(int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int *taskids[NUM_THREADS];
    int rc, t;

    messages[0] = "English: Hello World!";
    messages[1] = "French: Bonjour, le monde!";
    messages[2] = "Spanish: Hola al mundo";
    messages[3] = "Klingon: Nuq neH!";
    messages[4] = "German: Guten Tag, Welt!";
    messages[5] = "Russian: Zdravstvytye, mir!";
    messages[6] = "Japan: Sekai e konnichiwa!";
    messages[7] = "Latin: Orbis, te saluto!";
```

Argument Passing Example 1

```
//  main, continued

  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] );

     if (rc) {
           printf("ERROR; return code from pthread_create() is %d\n", rc);
           exit(-1);
     }
  }

  pthread_exit(NULL);
}
```

Argument Passing Example 1 Output

```
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!
```

Argument Passing Example 2

```
/******************************************************************************
* DESCRIPTION:
*   A "hello world" Pthreads program which demonstrates another safe way
*   to pass arguments to threads during thread creation.  In this case,
*   a structure is used to pass multiple arguments.
******************************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS   8

char *messages[NUM_THREADS];

struct thread_data
{
   int       thread_id;
   int  sum;
   char *message;
};

struct thread_data thread_data_array[NUM_THREADS];
```

Argument Passing Example 2

```
void *PrintHello(void *threadarg)
{
   int taskid, sum;
   char *hello_msg;

   struct thread_data *my_data;

   sleep(1);

   my_data = (struct thread_data *) threadarg;

   taskid = my_data->thread_id;

   sum = my_data->sum;

   hello_msg = my_data->message;

   printf("Thread %d: %s  Sum=%d\n", taskid, hello_msg, sum);

   pthread_exit(NULL);
}
```

Argument Passing Example 2

```
int main(int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int *taskids[NUM_THREADS];
    int rc, t, sum;

    sum=0;

    messages[0] = "English: Hello World!";
    messages[1] = "French: Bonjour, le monde!";
    messages[2] = "Spanish: Hola al mundo";
    messages[3] = "Klingon: Nuq neH!";
    messages[4] = "German: Guten Tag, Welt!";
    messages[5] = "Russian: Zdravstvytye, mir!";
    messages[6] = "Japan: Sekai e konnichiwa!";
    messages[7] = "Latin: Orbis, te saluto!";
```

Argument Passing Example 2

```
// main, continued

    for(t=0;t<NUM_THREADS;t++) {

            sum = sum + t;

            thread_data_array[t].thread_id = t;
            thread_data_array[t].sum = sum;
            thread_data_array[t].message = messages[t];

            printf("Creating thread %d\n", t);

             rc = pthread_create(&threads[t], NULL, PrintHello,
                                  (void *) &thread_data_array[t] );
             if (rc) {
                 printf("ERROR; return code from pthread_create() is %d\n", rc);
                 exit(-1);
            }
      }
     pthread_exit(NULL);
}
```

Argument Passing Example 2 Output

```
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
```

Example 3 - Thread Argument Passing (Incorrect)

```
/******************************************************************************
 * DESCRIPTION:
 *   This "hello world" Pthreads program demonstrates an unsafe (incorrect)
 *   way to pass thread arguments at thread creation.  In this case, the
 *   argument variable is changed by the main thread as it creates new threads.
 ******************************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS   8

void *PrintHello(void *threadid)
{
  long taskid;
  sleep(1);
  taskid = (long) *threadid;
  printf("Hello from thread %ld\n", taskid);
  pthread_exit(NULL);
}
```

Example 3 - Thread Argument Passing (Incorrect)

```
int main(int argc, char *argv[])
{
     pthread_t threads[NUM_THREADS];
     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);

               if (rc) {
                    printf("ERROR; return code from pthread_create() is %d\n", rc);
                    exit(-1);
                }
     }

     pthread_exit(NULL);
}
```

Argument Passing Example 3 Output

```
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 1
Hello from thread 1
Hello from thread 7
Hello from thread 7
Hello from thread 7
Hello from thread 7
Hello from thread 7
Hello from thread 7
```

Compilation

- The `pthread.h` header file must be the first included file of each source file using the threads library.

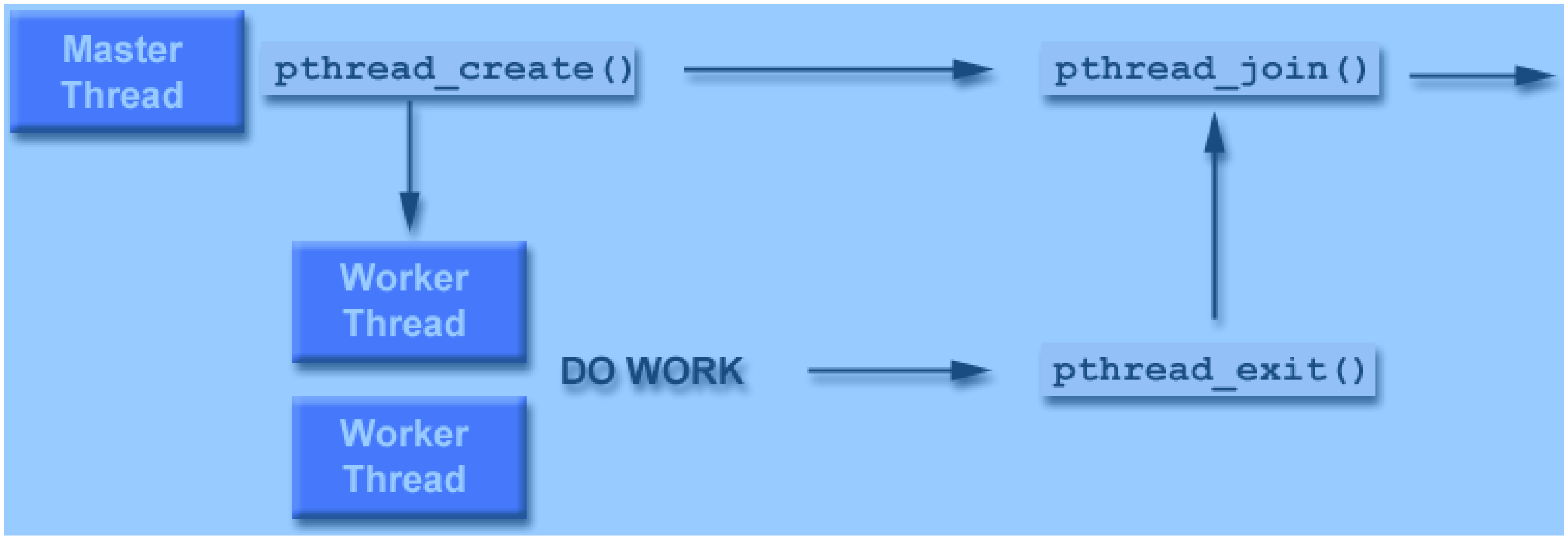
- Otherwise, the -D_THREAD_SAFE compilation flag should be used

Joining and Detaching Threads

- Routines:
- `pthread_join(threadid,status)`
- `pthread_detach(threadid,status)`
- `pthread_attr_setdetachstate(attr,detachstate)`
- `pthread_attr_getdetachstate(attr,detachstate)`
- The possible attribute states for the last two routines are:
  - PTHREAD_CREATE_DETACHED or
  - PTHREAD_CREATE_JOINABLE.

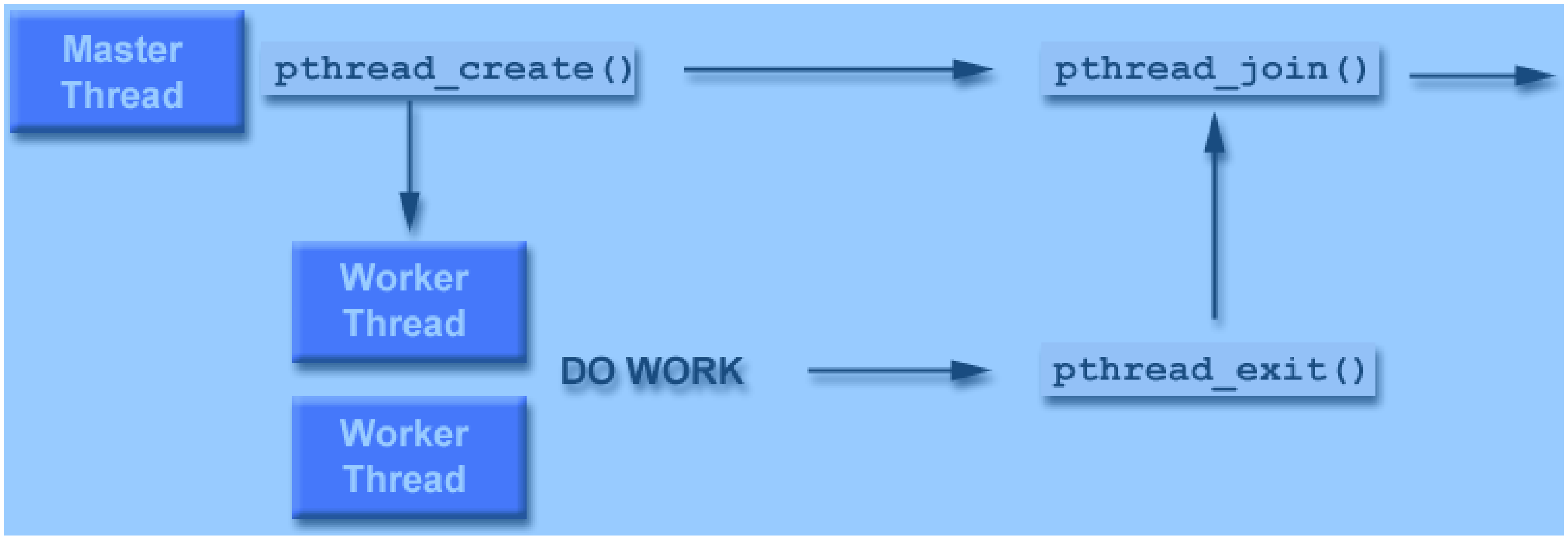
Joining

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



Joining

- 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()`
- It is a logical error to attempt simultaneous multiple joins on the same target thread



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.

Joinable or Not?

- 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()`

Detach

- The `pthread_detach()`routine 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: Pthread Joining

- **Example Code - 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

Example: Pthread Joining

```
#include <pthread.h>
 #include <stdio.h>
#include <stdlib.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);
}
```

Example: Pthread Joining

```
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);
         }
      }
```

Example: Pthread 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) {
         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);
}
```

Example: Pthread Joining

```
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.
```

- Other….

Miscellaneous Routines

- `pthread_self()`
  returns the unique, system assigned thread ID of the calling thread
- `pthread_equal(thread1,thread2)`
  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.

Miscellaneous Routines

- `pthread_once(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_yield Subroutine

- **Purpose**:
  Forces the calling thread to relinquish use of its processor
- **NOTE: this routine is NOT available on Linux machines and some Solaris machines. Use sched_yield, from unistd.h, instead.**
- **Library**:
  Threads Library (`libpthreads.a`)
- **Syntax**:

```
#include <pthread.h>
void pthread_yield ()
```

- **Description**:
  The pthread_yield subroutine forces the calling thread to relinquish use of its processor, and to wait in the run queue before it is scheduled again

  If the run queue is empty when the pthread_yield subroutine is called, the calling thread is immediately rescheduled.