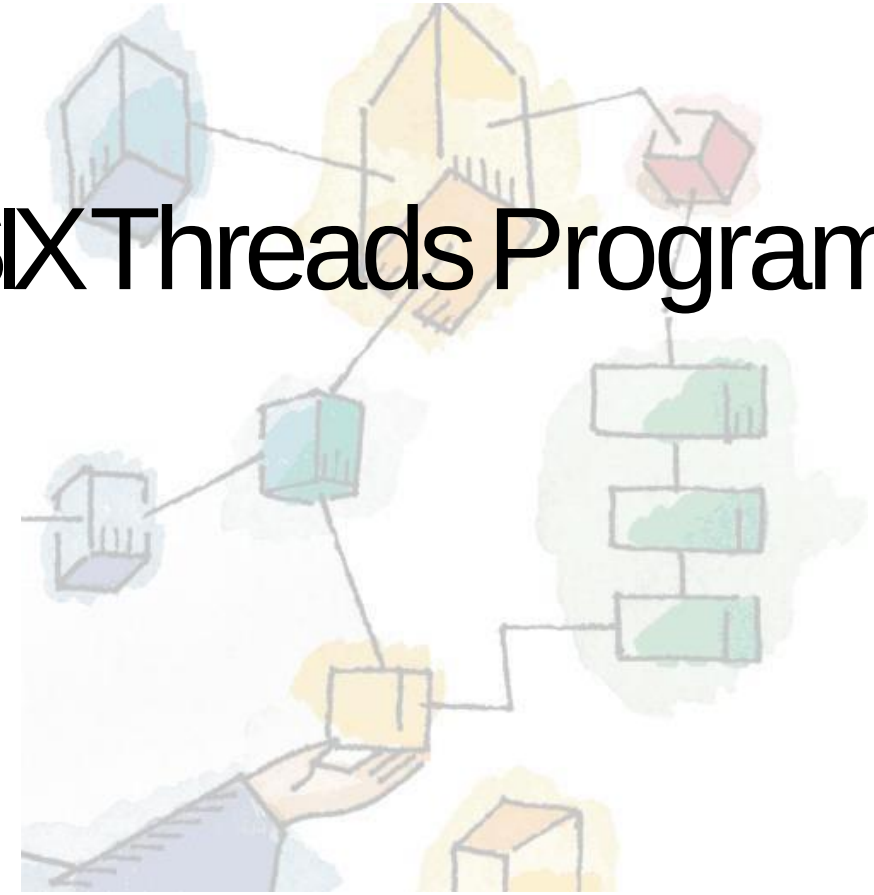
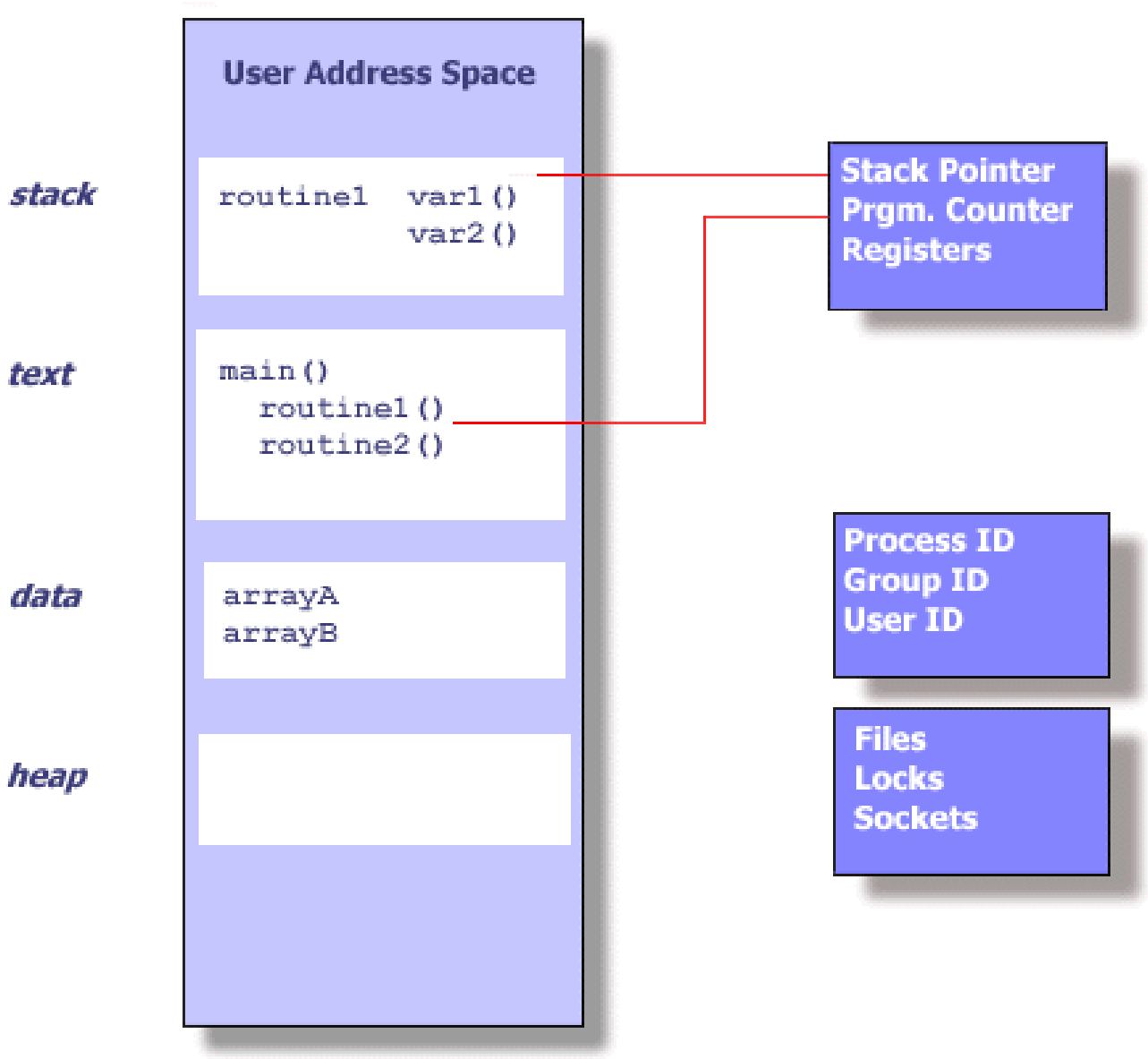
POSIX Threads Programming



What is a Thread?

- Recall that **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

Unix Process

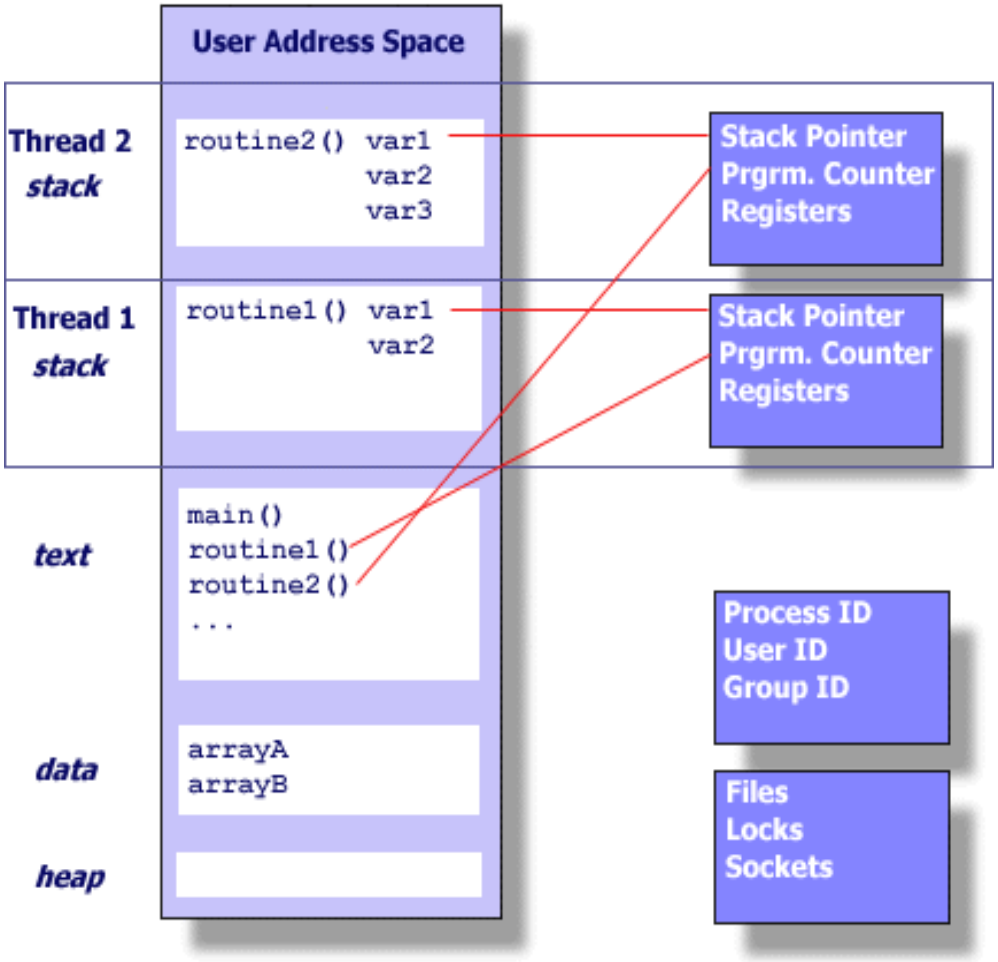


What is a Thread?

- Threads use, and exist within, the process resources
- scheduled by the operating system and run as independent entities
- duplicate only the bare essential resources that enable them to exist as executable code

What is a Thread?

- Independent flow of control possible 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.



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.

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

Why Pthreads?

- Potential performance gains and practical advantages over non-threaded applications:
  - 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.

Parallel Programming

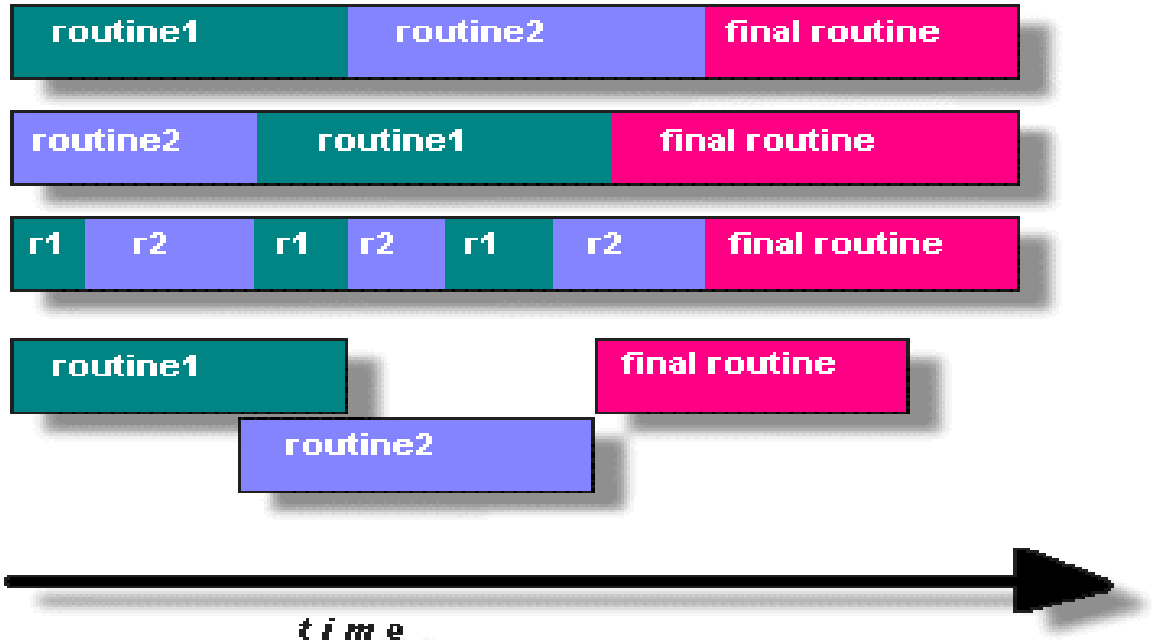
- On modern, multi-cpu machines, pthreads are ideally suited for parallel programming
- Whatever applies to parallel programming in general, applies to parallel pthreads programs

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, 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
  - 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)

Parallel Programming

- Pthreads can also be used for serial applications to emulate parallel execution
- For most people, the typical web browser runs desktop/laptop machine with a single CPU
- Many things can "appear" to be happening at the same time

Parallel Programming

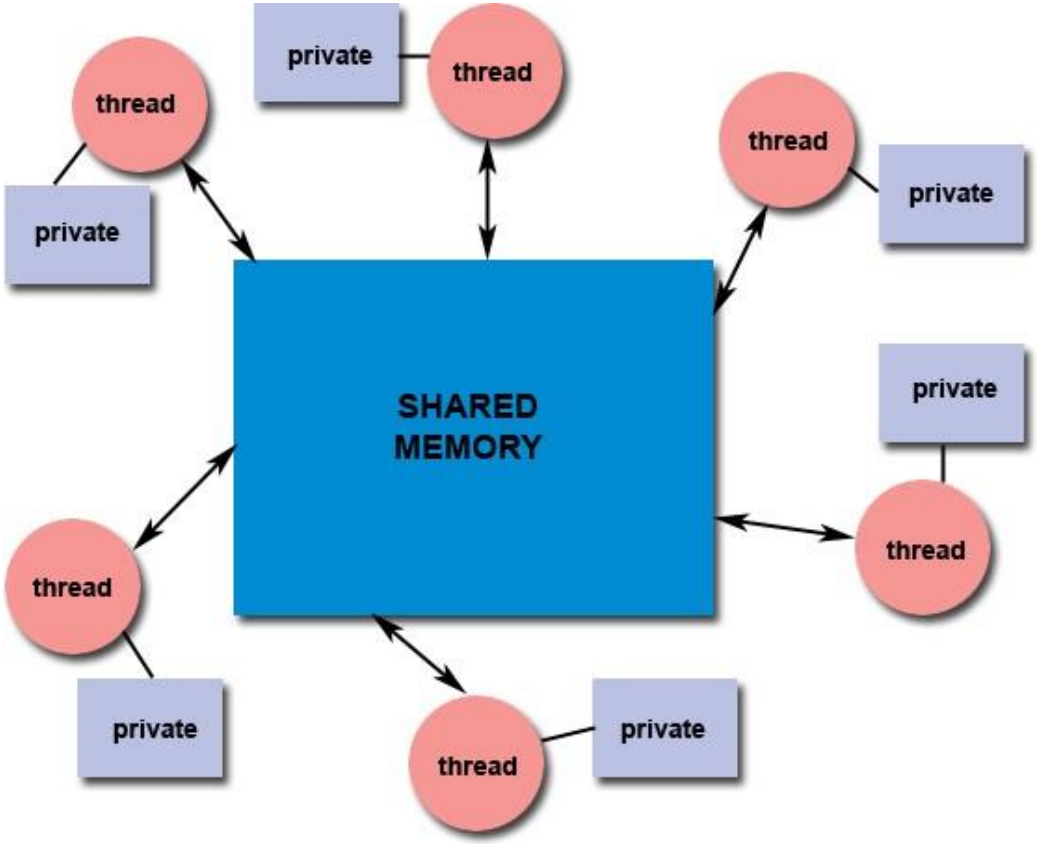
- Several common models for threaded programs exist:

- ***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.

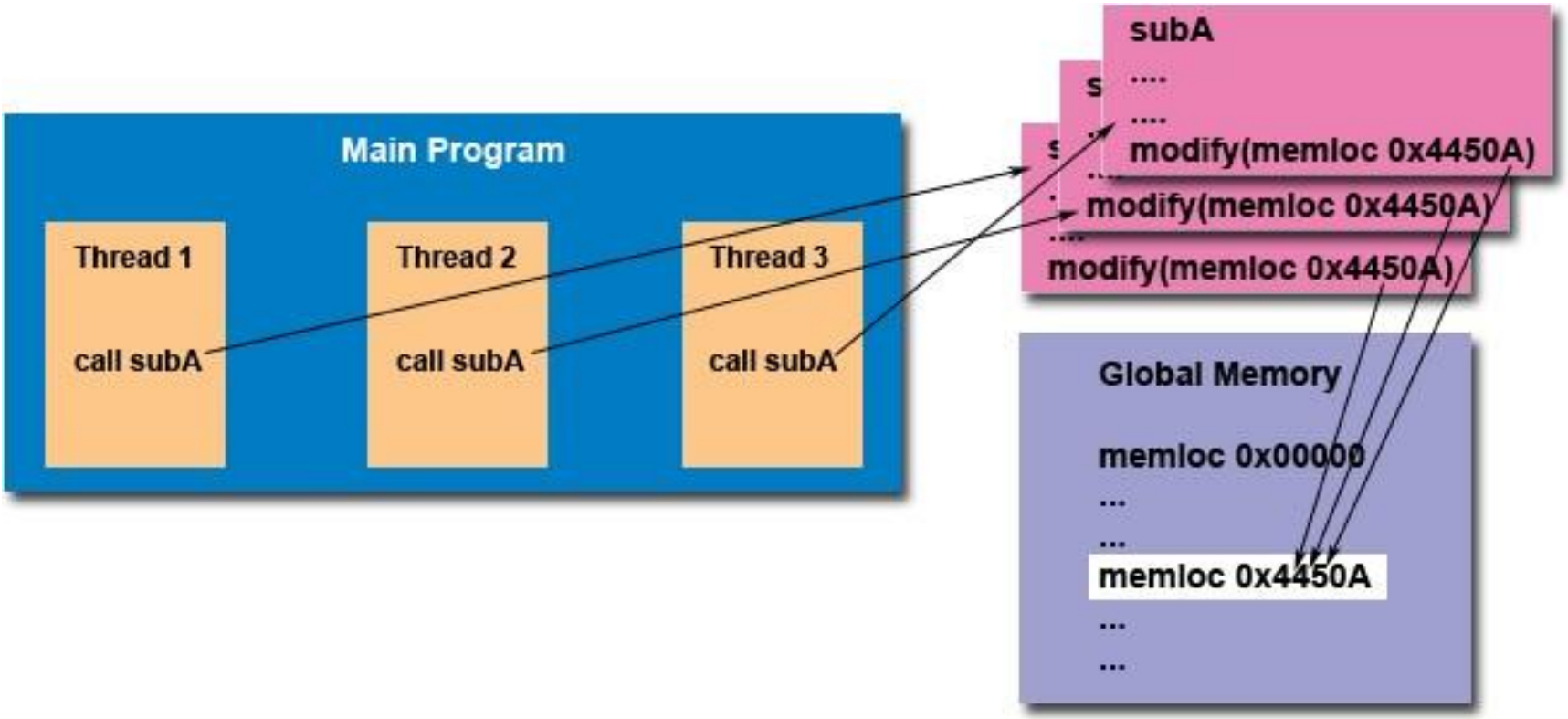


Thread-safeness

- 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.

Thread-safeness



Thread-safeness

- **The implication to users of external library routines:**
- 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.

The Pthreads API

- To compile using GNU on Linux:

```
gcc -pthread
```

- Pthread routines

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

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)`

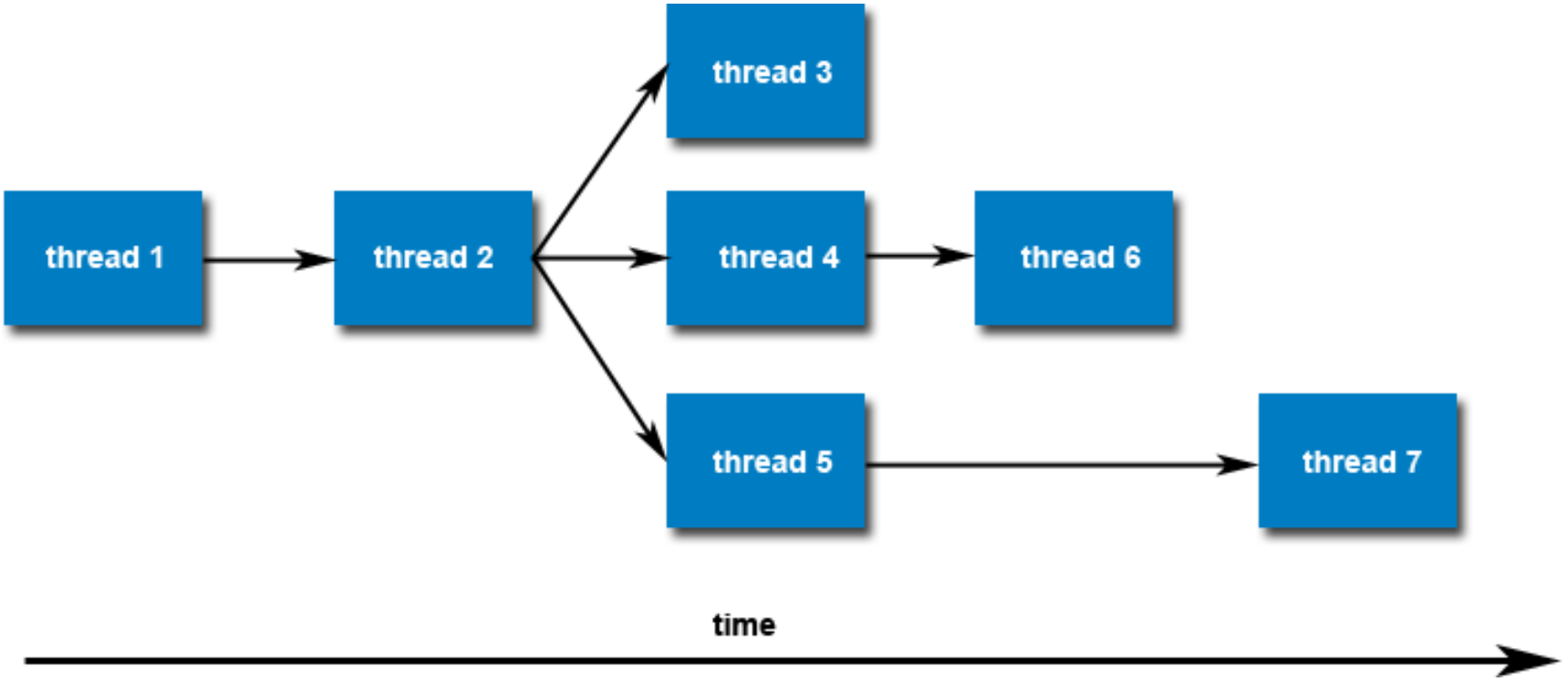
- **`pthread_create`** creates a new thread and makes it executable
- This routine can be called any number of times from anywhere within your code

Pthread_create

- `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 Croutine 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.

Pthreads

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



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
- 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 <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

```
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

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 *)

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

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

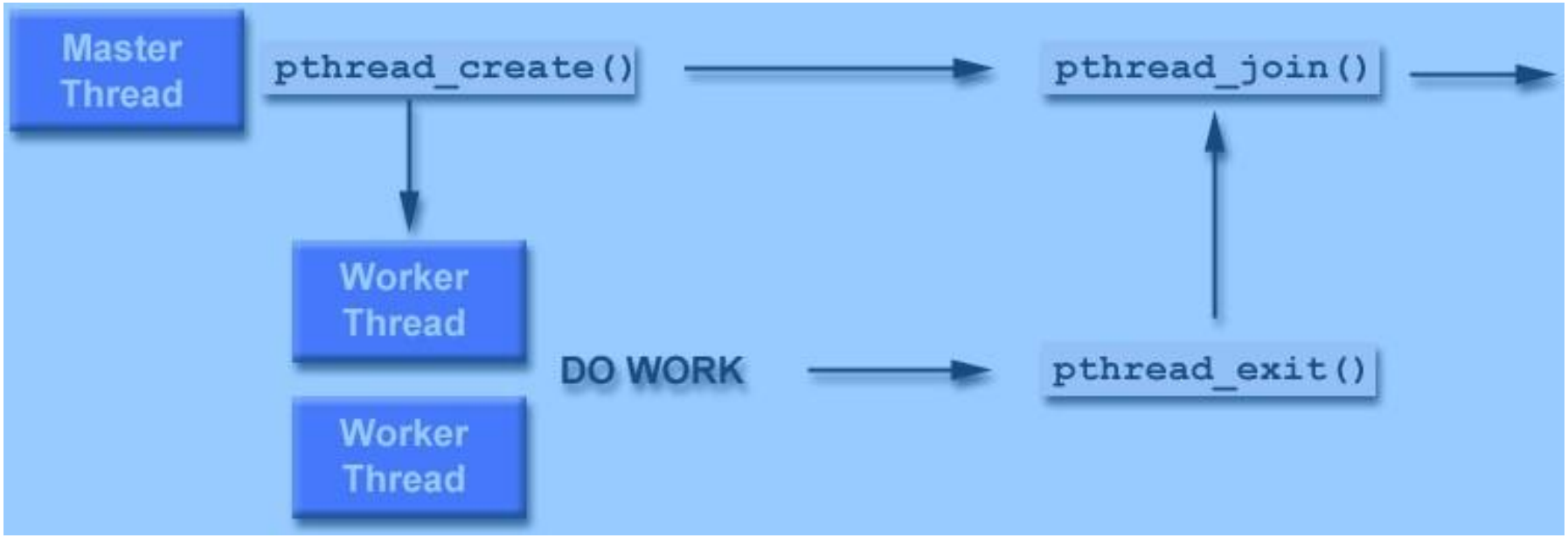
```
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Thread 0: English: Hello World!      Sum=0
Thread 1: French: Bonjour, le monde!
          Sum=1
Thread 2: Spanish: Hola al mundo     Sum=3
```

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.

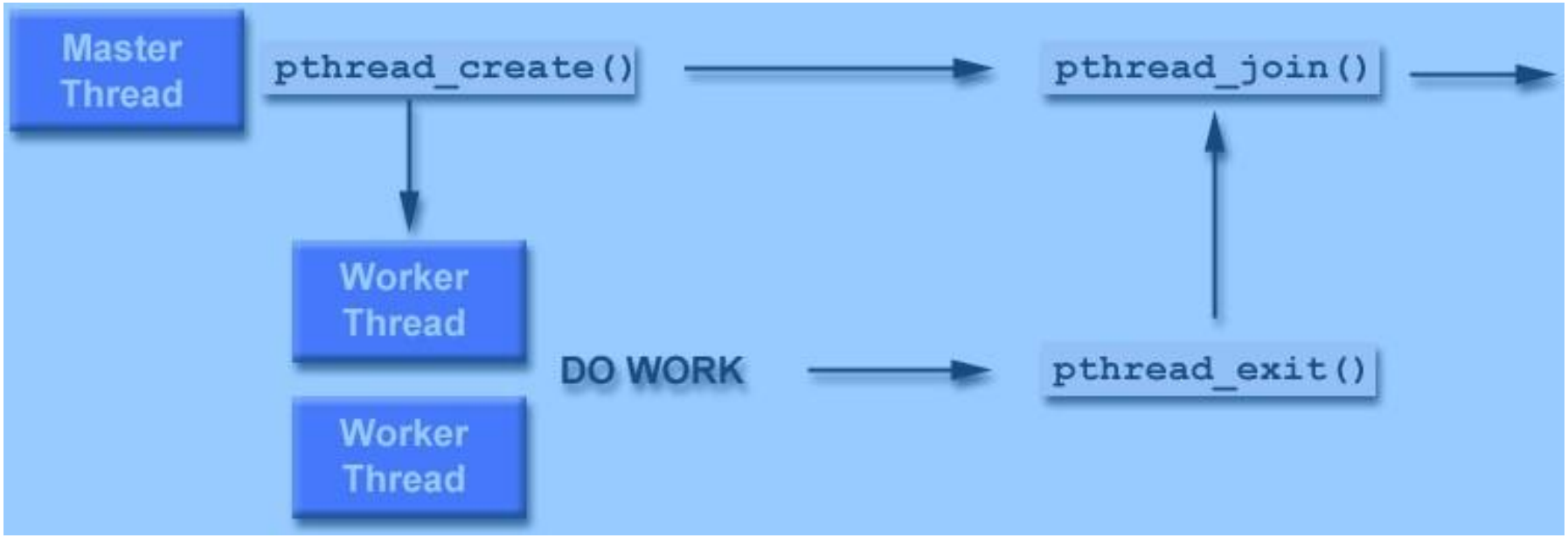
Joining

- "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

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.