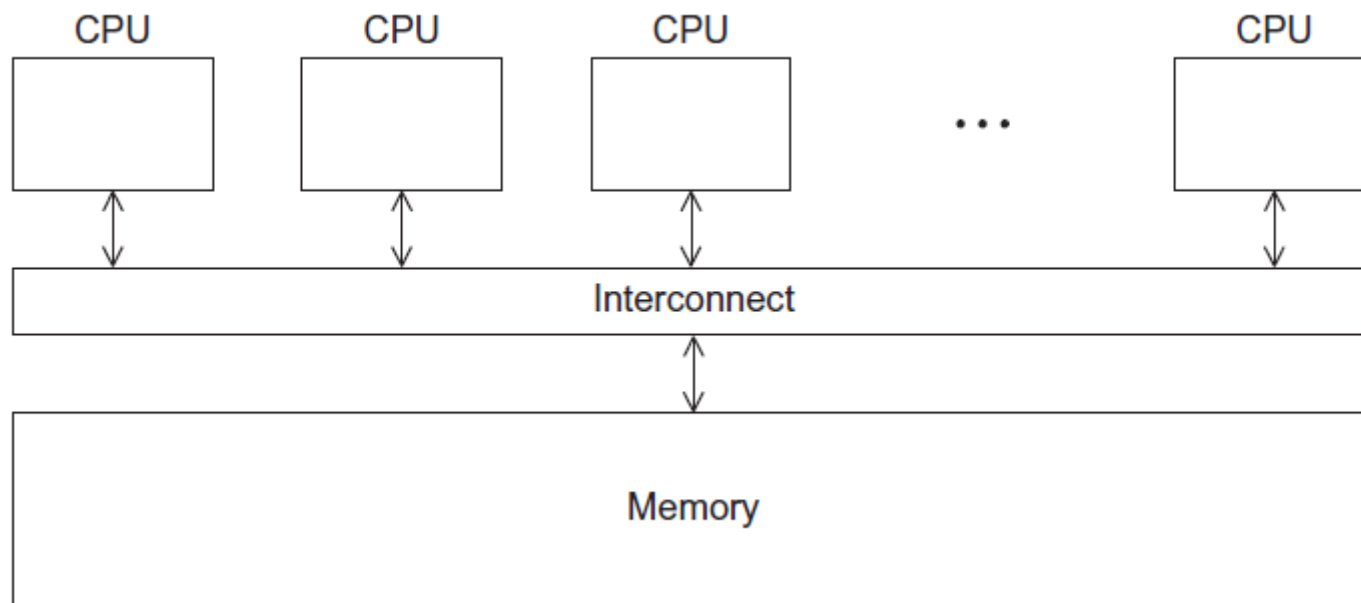


Principles of Systems Programming

A Shared Memory System



Threads

- What is a thread?
 - A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler
- The implementation of threads and processes differs between operating systems
 - in most cases a thread is a component of a process
- Multiple threads can exist within one process, executing concurrently and share resources such as memory, while different processes do not share these resources.
- A way of executing parallel parts of an application

Threads

- 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"
- Threads use and exist within process resources, yet are able to be scheduled by the operating system and run as independent entities largely because they duplicate only the bare essential resources that enable them to exist as executable code.

Threads

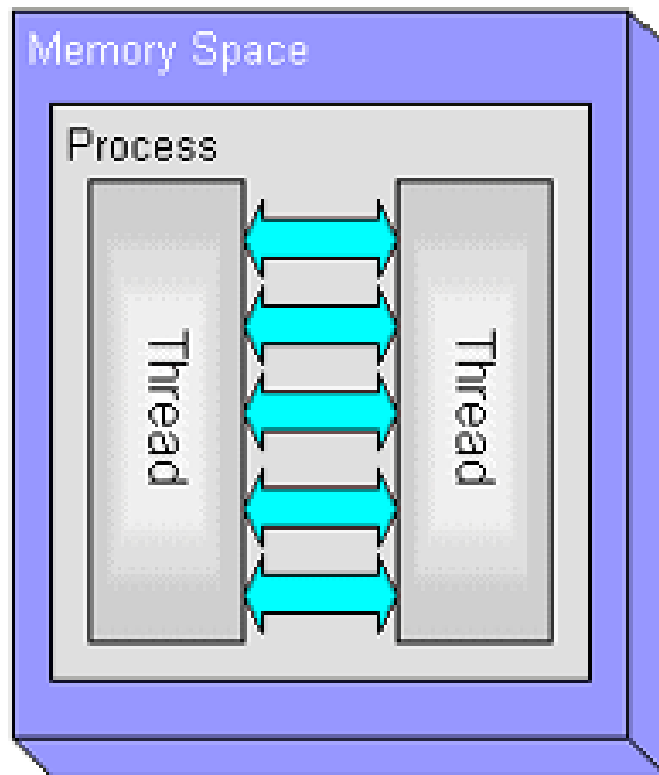
- So, in summary, in the UNIX environment a thread:
 - Exists within a process and uses the process resources
 - Has its own independent flow of control as long as its parent process exists and the OS supports it
 - Duplicates only the essential resources it needs to be independently schedulable
 - May 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.
- 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.

Threads vs. processes

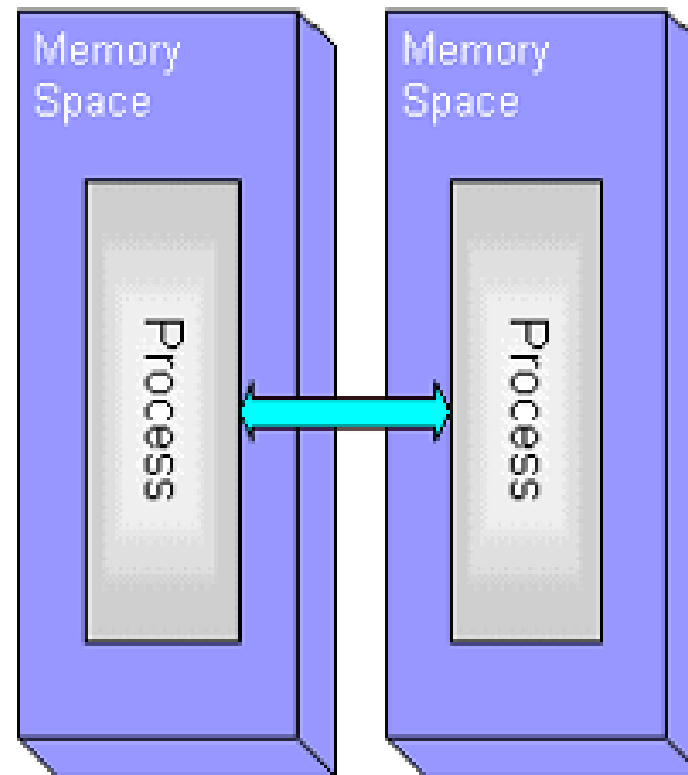
- Threads differ from traditional multitasking operating system processes in that:
 - processes are typically independent, while threads exist as subsets of a process
 - processes carry considerably more state information than threads, whereas multiple threads within a process share process state as well as memory and other resources
 - processes have separate address spaces, whereas threads share their address space
 - processes interact only through system-provided inter-process communication mechanisms
 - context switching between threads in the same process is typically faster than context switching between processes.

Threads vs. processes

Threads



Forked Processes



What are Pthreads?

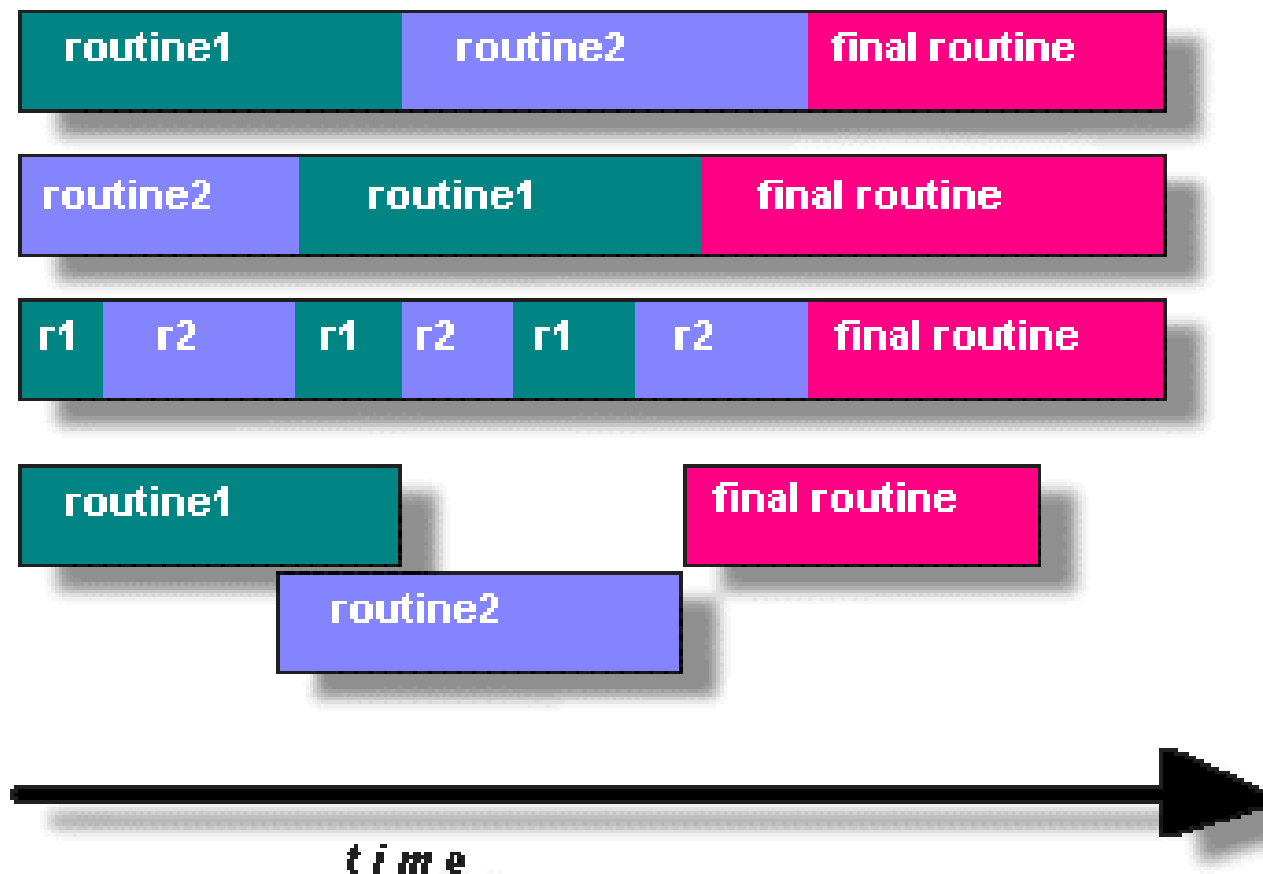
- Historically, hardware vendors have implemented their own proprietary versions of threads
- In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required
 - For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995)
 - Implementations adhering to this standard are referred to as POSIX threads, or Pthreads
 - Most hardware vendors now offer Pthreads in addition to their proprietary API's.
- The POSIX standard has continued to evolve and undergo revisions, including the Pthreads specification.

Designing Threaded Programs

- Parallel Programming:
 - On modern, multi-core machines, pthreads are ideally suited for parallel programming, and whatever applies to parallel programming in general, applies to parallel pthreads programs.
 - 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.

Designing Threaded Programs

- Parallel Programming:

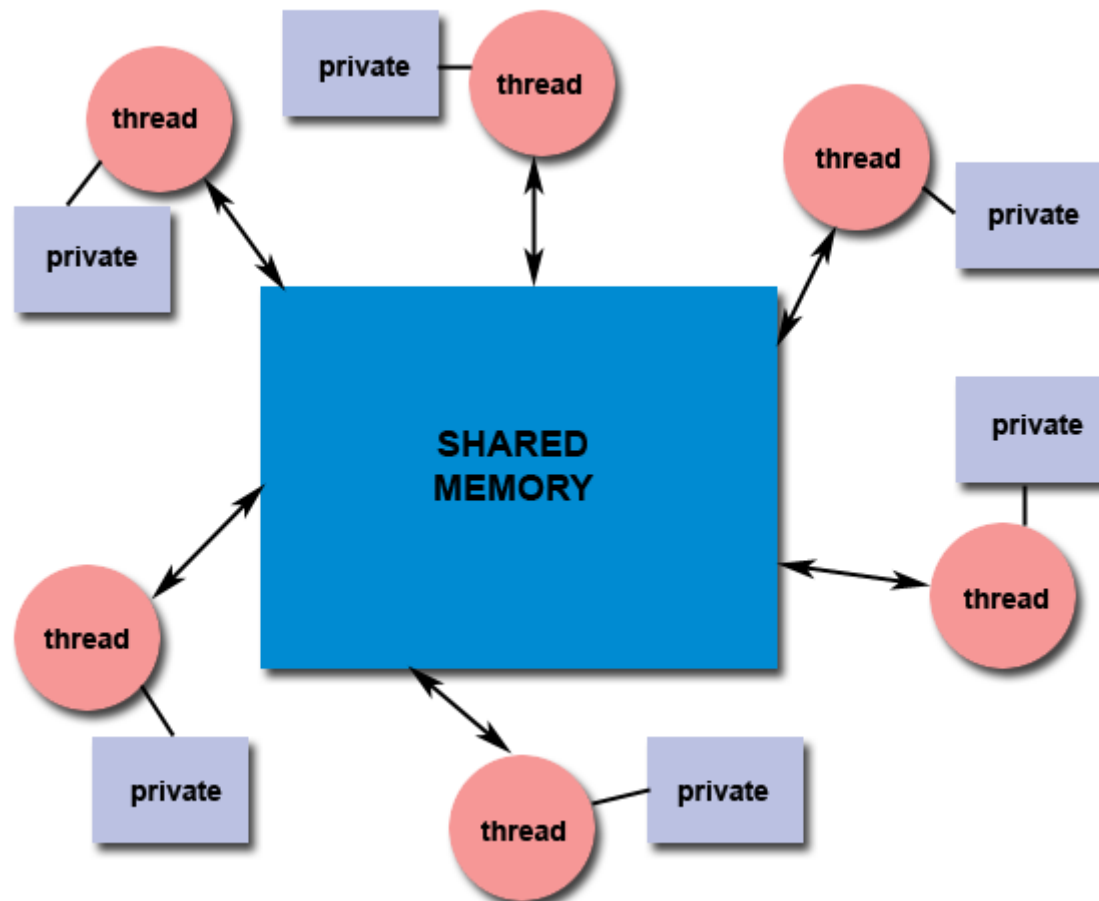


Designing Threaded Programs

- 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.

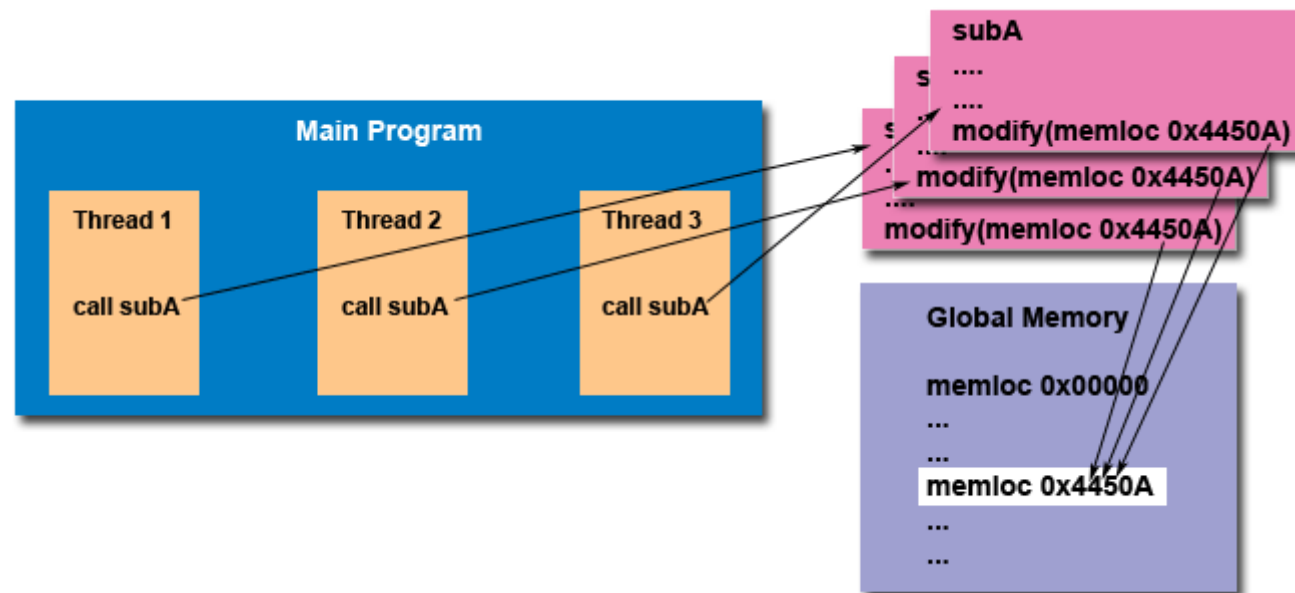
Designing Threaded Programs

- Shared Memory Model:



Designing Threaded Programs

- 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.



Designing Threaded Programs

- Thread Limits:
 - Although the Pthreads API is an ANSI/IEEE standard, implementations can, and usually do, vary in ways
 - Because of this, a program that runs fine on one platform, may fail or produce wrong results on another platform
 - For example, the maximum number of threads permitted, and the default thread stack size are two important limits to consider when designing your program

POSIX[®] Threads

- Also known as Pthreads.
- A standard for Unix-like operating systems.
- A library that can be linked with C programs.
- Specifies an application programming interface (API) for multi-threaded programming.

Caveat

- The Pthreads API is only available on POSIXR systems — Linux, MacOS X, Solaris, HPUX, ...



The Pthreads API

- The subroutines which comprise the Pthreads API can be informally grouped into four major groups:
- Thread management: Routines that work directly on threads - creating, detaching, joining, etc.
- 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.
- 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.
- Synchronization: Routines that manage read/write locks

The Pthreads API

- Naming conventions: All identifiers begin with pthread_. Examples:

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous 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
pthread_rwlock_	Read/write locks
pthread_barrier_	Synchronization barriers

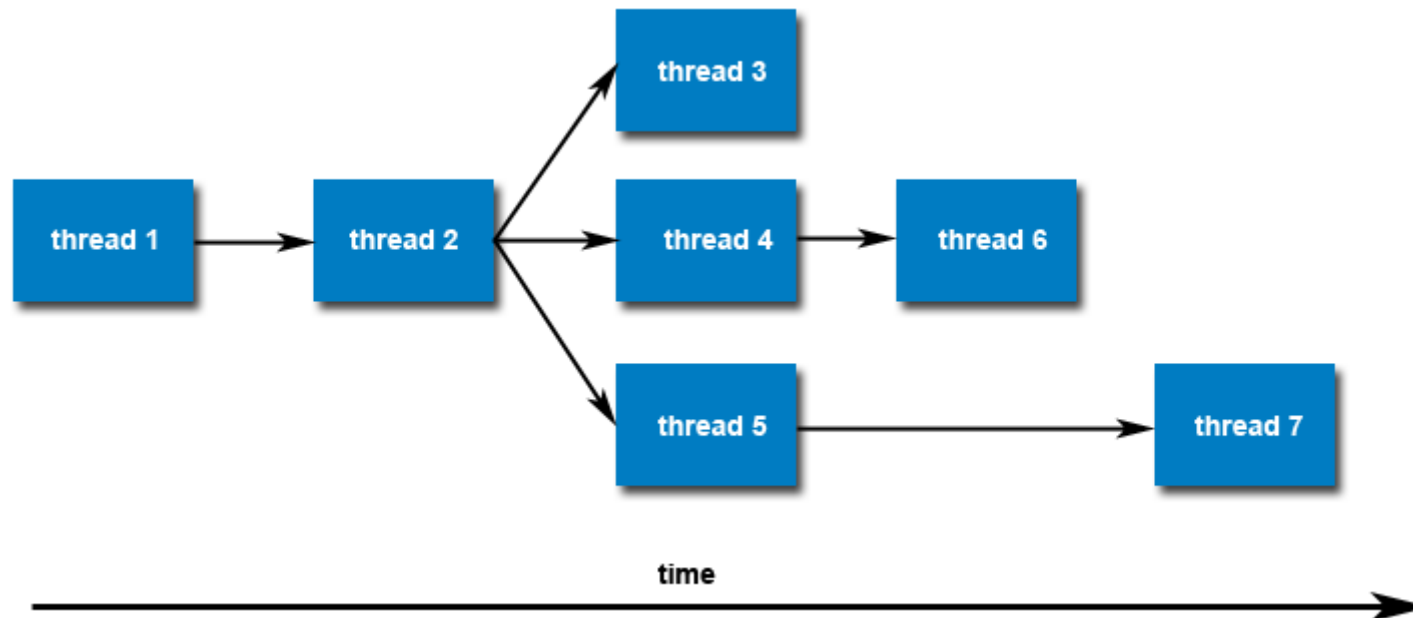
Starting the Threads

- Processes in MPI are usually started by a script.
- In Pthreads the threads are started by the program executable.

Thread Management

`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.

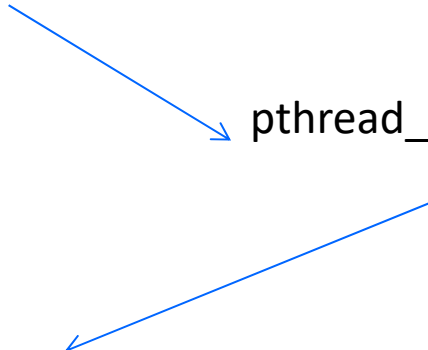


Starting the Threads

pthread.h

*One object for
each thread.*

pthread_t



```
int pthread_create (
    pthread_t* thread_p /* out */,
    const pthread_attr_t* attr_p /* in */,
    void* (*start_routine) ( void ) /* in */,
    void* arg_p /* in */ );
```

pthread_t objects

- **Opaque**
- The actual data that they store is system-specific.
- Their data members aren't directly accessible to user code.
- However, the Pthreads standard guarantees that a pthread_t object does store enough information to uniquely identify the thread with which it's associated.

A closer look (1)

```
int pthread_create (  
    pthread_t* thread_p /* out */,  
    const pthread_attr_t* attr_p /* in */,  
    void* (*start_routine) ( void ) /* in */,  
    void* arg_p /* in */ );
```

We won't be using, so we just pass NULL.

Allocate before calling.

A closer look (2)

```
int pthread_create (  
    pthread_t* thread_p /* out */,  
    const pthread_attr_t* attr_p /* in */,  
    void* (*start_routine) ( void ) /* in */,  
    void* arg_p /* in */ );
```

Pointer to the argument that should
be passed to the function *start_routine*.

The function that the thread is to run.

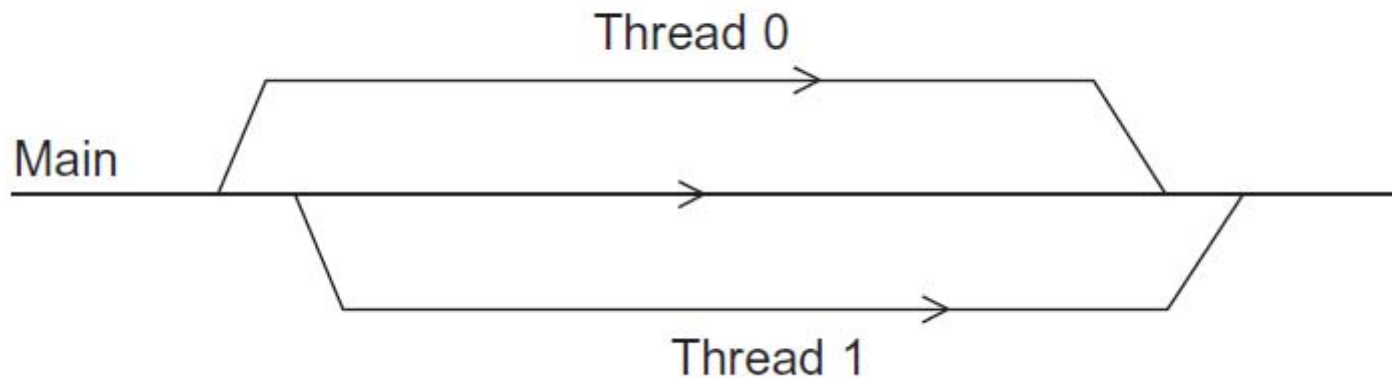
Function started by pthread_create

- Prototype:

```
void* thread_function ( void* args_p );
```

- Void* can be cast to any pointer type in C.
- So args_p can point to a list containing one or more values needed by thread_function.
- Similarly, the return value of thread_function can point to a list of one or more values.

Running the Threads



Main thread forks and joins two threads.

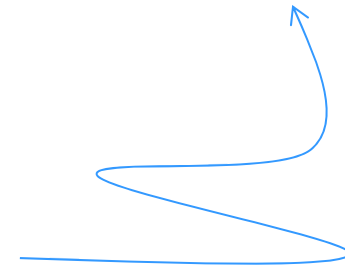
Terminating Threads

- There are several ways in which a thread may be terminated:
 - The thread returns normally from its starting routine. Its work is done
 - The thread makes a call to the *pthread_exit* subroutine - whether its work is done or not.
 - The thread is canceled by another thread via the *pthread_cancel* routine.
 - The entire process is terminated due to making a call to either the *exec()* or *exit()*
 - If *main()* finishes first, without calling *pthread_exit* explicitly itself

Compiling a Pthread program

```
gcc -g -Wall -o pthread_hello pthread_hello.c -lpthread
```

link in the Pthreads library



```
#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);
}

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);
        }
    }

    /* Last thing that main() should do */
    pthread_exit(NULL);
}
```

```

#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);
}

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);
        }
    }

    /* Last thing that main() should do */
    pthread_exit(NULL);
}

```

Output:

```

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

```

Thread Management

- The *pthread_create()* routine permits the programmer to pass one argument to the thread start routine.
- What about multiple arguments?
 - Create 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 *)

```
#define NUM_THREADS    8

char *messages[NUM_THREADS];

void *PrintHello(void *threadid)
{
    long taskid;

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

int main(int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    long 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: Zdravstvuyte, mir!";
    messages[6] = "Japan: Sekai e konnichiwa!";
    messages[7] = "Latin: Orbis, te saluto!";
```



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

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

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!
```

```
#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];

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);
}
```

```
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!";

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);
}
```

```

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!";

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);
}

```

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

```

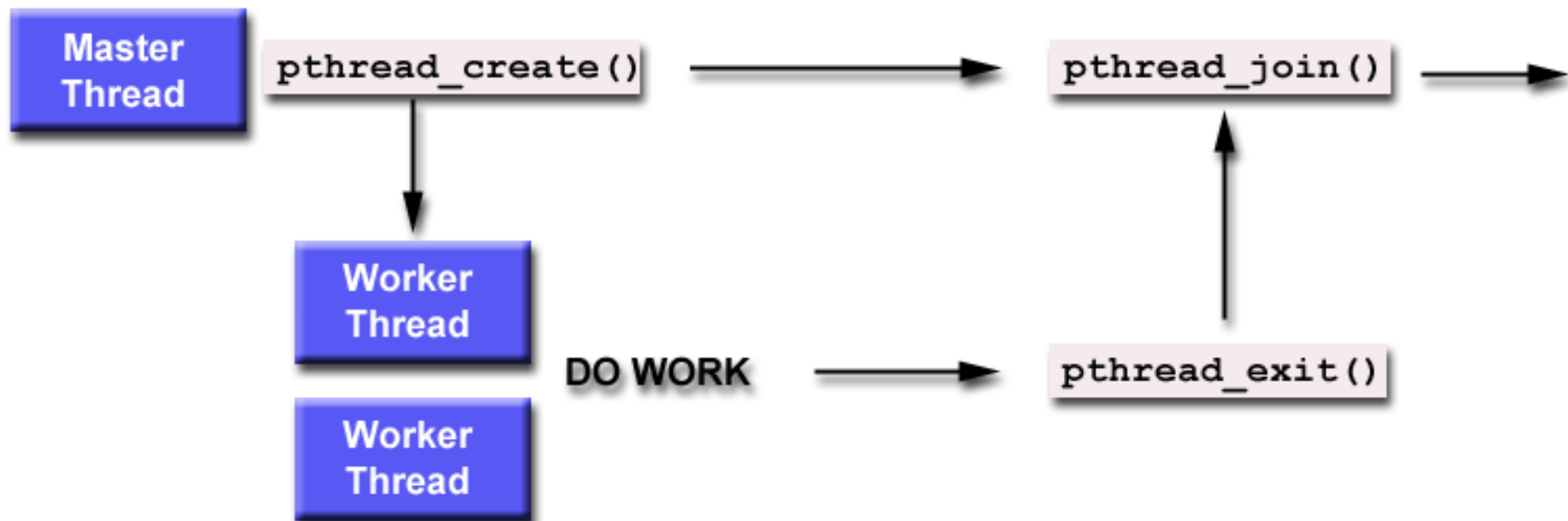
Thread Management

```
int pthread_attr_init(pthread_attr_t *attr);  
int pthread_attr_destroy(pthread_attr_t *attr);
```

- The `pthread_attr_init()` function initializes the thread attributes object pointed to by `attr` with default attribute values.
- When a thread attributes object is no longer required, it should be destroyed using the `pthread_attr_destroy()` function.

Thread Management

- Joining and detaching threads
 - "Joining" is one way to accomplish synchronization between threads



Thread Management

`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 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.

Thread Management

`pthread_detach(status)`

- What should I do when my thread doesn't return anything useful, and I don't have to wait for its completion?
 - Do I have to call `pthread_join()` anyway just for clean-up purpose?
- Pthreads offers a mechanism to tell the system: I am starting this thread, but I am not interested about joining it. Please perform any clean-up action for me, once the thread has terminated

Thread Management

- Create a detached thread when you know you won't want to wait for it with `pthread_join()`. The only performance benefit is that when a detached thread terminates, its resources can be released immediately instead of having to wait for the thread to be joined before the resources can be released.
- It is 'legal' not to join a joinable thread; but it is not usually advisable because (as previously noted) the resources won't be released until the thread is joined, so they'll remain tied up indefinitely (until the program exits) if you don't join it.

Thread Management

- Joinable or not?
 - When a thread is created, one of its attributes defines whether it is joinable or detached.
 - Threads are made joinable by default.
 - Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.

Thread Management

- 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 (attr, detachstate)
 - PTHREAD_CREATE_DETACHED
 - PTHREAD_CREATE_JOINABLE
 - When done, free library resources used by the attribute with pthread_attr_destroy().

Thread Management - Recommendations

- Even though the threads are created as joinable by default, 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 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 so that they can be joined later.

```
#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);
}

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);
    }
}

/* 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);
}

```


Output:

```
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.
```

Get number of threads from command line

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

/* Global variable: accessible to all threads */
int thread_count;

void* Hello(void* rank); /* Thread function */

int main(int argc, char* argv[]) {
    long thread; /* Use long in case of a 64-bit system */
    pthread_t* thread_handles;

    /* Get number of threads from command line */
    thread_count = strtol(argv[1], NULL, 10);

    thread_handles = malloc (thread_count*sizeof(pthread_t));

    for (thread = 0; thread < thread_count; thread++)
        pthread_create(&thread_handles[thread], NULL,
            Hello, (void*) thread);

    printf("Hello from the main thread\n");

    for (thread = 0; thread < thread_count; thread++)
        pthread_join(thread_handles[thread], NULL);

    free(thread_handles);
    return 0;
} /* main */

void* Hello(void* rank) {
    long my_rank = (long) rank
        /* Use long in case of 64-bit system */

    printf("Hello from thread %ld of %d\n", my_rank,
        thread_count);

    return NULL;
} /* Hello */
```

Running a Pthreads program

```
./ pth_hello <number of threads>
```

```
./ pth_hello 1
```

Hello from the main thread

Hello from thread 0 of 1

```
./ pth_hello 4
```

Hello from the main thread

Hello from thread 0 of 4

Hello from thread 1 of 4

Hello from thread 2 of 4

Hello from thread 3 of 4

Global variables

- Can introduce subtle and confusing bugs!
- Limit use of global variables to situations in which they're really needed.
 - Shared variables.



a_{00}	a_{01}	\cdots	$a_{0,n-1}$
a_{10}	a_{11}	\cdots	$a_{1,n-1}$
\vdots	\vdots		\vdots
a_{i0}	a_{i1}	\cdots	$a_{i,n-1}$
\vdots	\vdots		\vdots
$a_{m-1,0}$	$a_{m-1,1}$	\cdots	$a_{m-1,n-1}$

$\begin{matrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{matrix}$
 $=$

y_0
y_1
\vdots
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
\vdots
y_{m-1}

MATRIX-VECTOR MULTIPLICATION IN PTHREADS

Serial pseudo-code

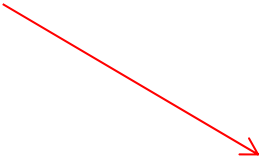
```
/* For each row of A */  
for (i = 0; i < m; i++) {  
    y[i] = 0.0;  
    /* For each element of the row and each element of x */  
    for (j = 0; j < n; j++)  
        y[i] += A[i][j]* x[j];  
}
```

$$y_i = \sum_{j=0}^{n-1} a_{ij}x_j$$

- We want to parallelize this by dividing the work among the threads.
- Divide the iterations of the outer loop among the threads.
- Each thread will compute some of the components of y.

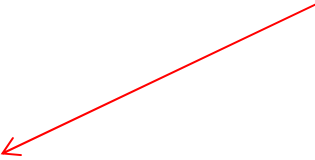
Using 3 Pthreads

Thread	Components of y
0	y[0], y[1]
1	y[2], y[3]
2	y[4], y[5]



```
y[0] = 0.0;
for (j = 0; j < n; j++)
    y[0] += A[0][j]*x[j];
```

thread 0



```
y[i] = 0.0;
for (j = 0; j < n; j++)
    y[i] += A[i][j]*x[j];
```

general case

Comments (1)

- Each thread will access every element of row i of A and every element of x .
- Make A and y shared.
 - Dangerous (why?) but useful
- If A and y are global, the main thread can easily initialize all of A
- Assume that both m and n are evenly divisible by t .
- Thread 0 gets the first m/t , thread 1 gets the next m/t , and so on
- The formulas for the components assigned to thread q are:

$$\text{first component: } q \times \frac{m}{t} \qquad \text{last component: } (q + 1) \times \frac{m}{t} - 1$$

Pthreads matrix-vector multiplication

```
void* Pth_mat_vect(void* rank) {
    long my_rank = (long) rank;
    int i, j;
    int local_m = m/thread_count;
    int my_first_row = my_rank*local_m;
    int my_last_row = (my_rank+1)*local_m - 1;

    for (i = my_first_row; i <= my_last_row; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i][j]*x[j];
    }

    return NULL;
} /* Pth_mat_vect */
```

Comments (2)

- In MPI chapter, it took more work to write a matrix-vector multiplication program using MPI.
 - data structures were necessarily distributed
 - each MPI process only has direct access to its own local memory.
 - In MPI code, we need to explicitly *gather* all of x into each process' memory.
- Writing shared-memory programs is easier than writing distributed-memory programs.
- However, there are situations in which shared-memory programs can be more complex in order to assure correctness.