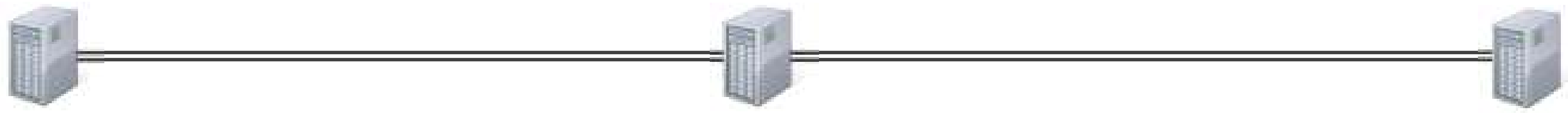
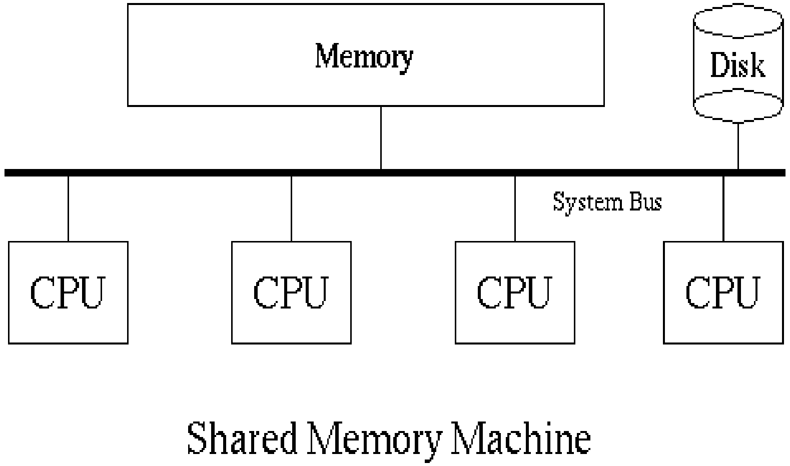
Operating Systems

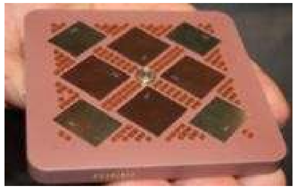
(pthreads)



- **Symmetric MultiProcessor**

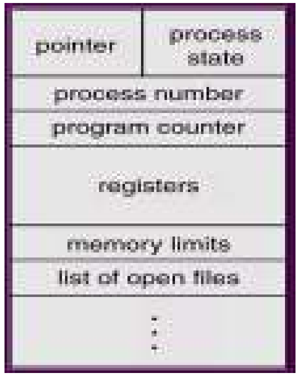


Shared Memory Machine

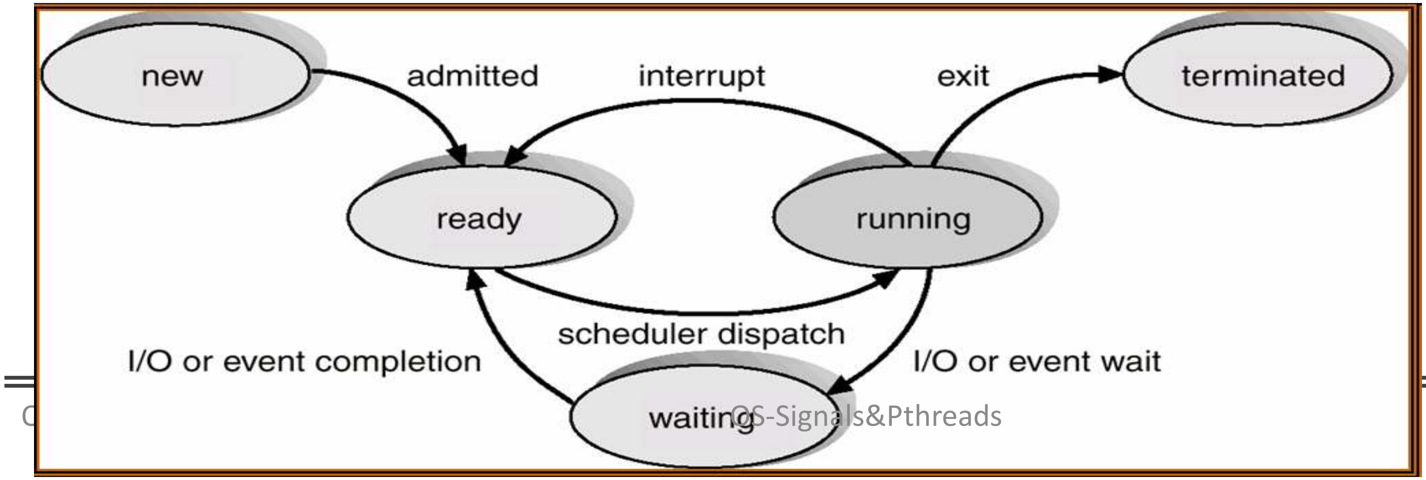


What is Process?

- **Program in execution is called a process the address space map,**
  - the current status of the process,
  - the execution priority of the process,
  - the resource usage of the process,
  - the current signal mask,
  - the owner of the process.

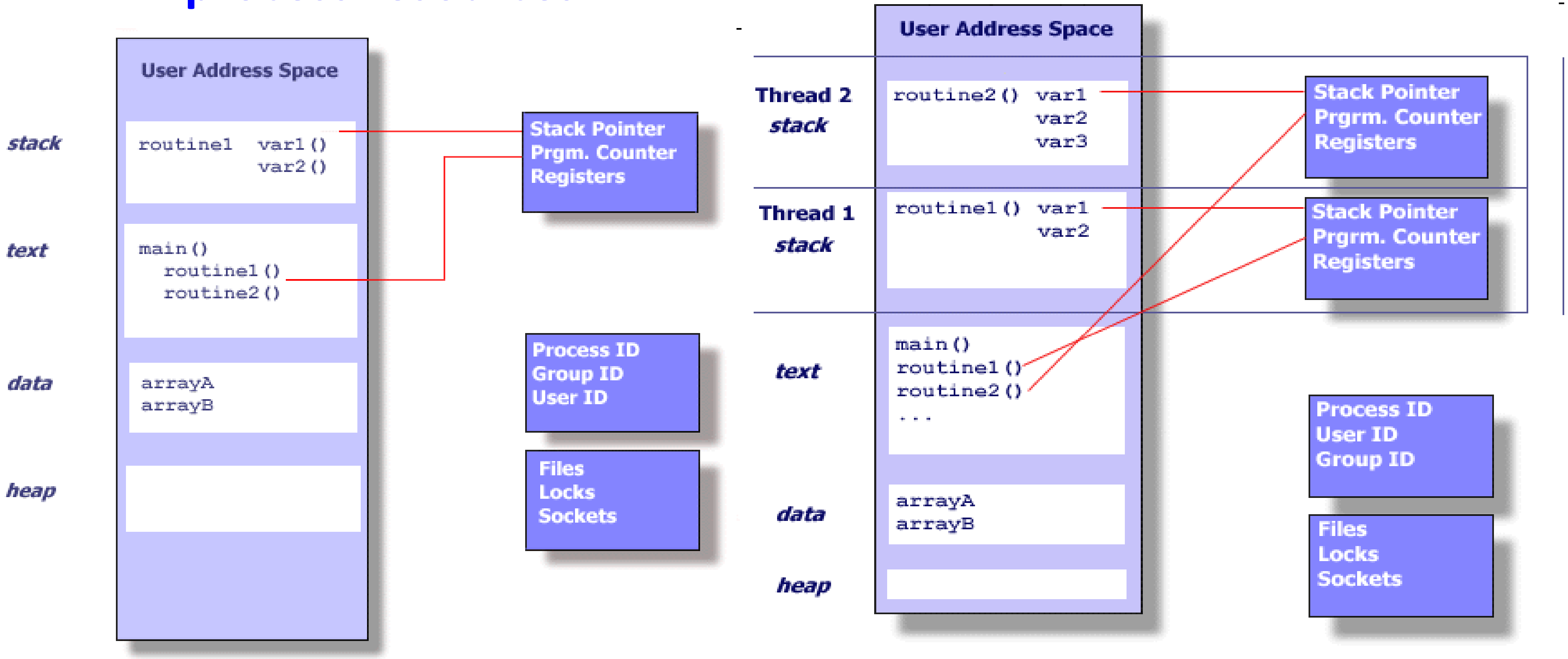


pointer | process state
process number
program counter
registers
memory limits
list of open files
⋮



What is Thread?

- **Is an independent /different stream of control that can execute its instructions independently and can use the process resources.**



Threads Features

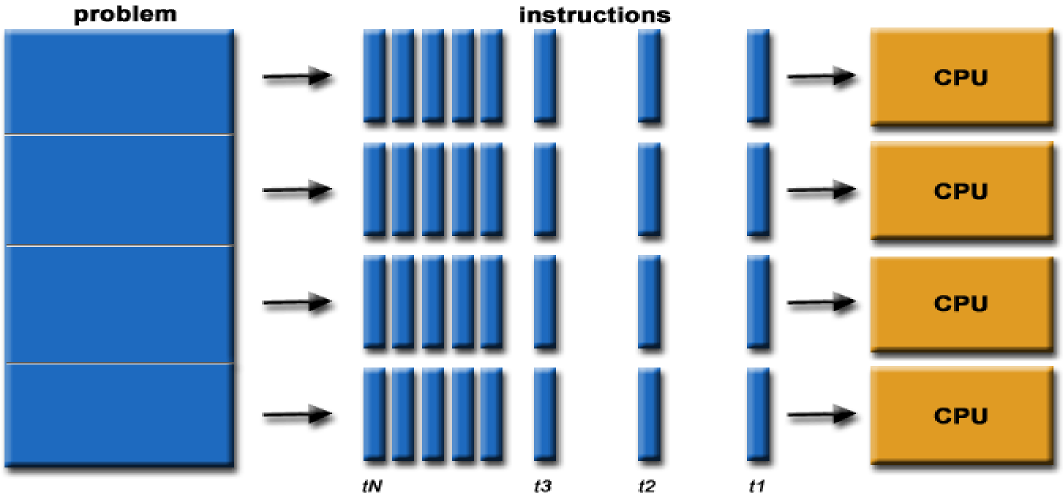
- Exists within a process and uses the process resources
- Has its own independent flow of control as long as its parent process exists and the OS supports it
- Duplicates only the essential resources it needs to be independently schedulable
- May share the process resources with other threads that act 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.

Disadvantages

- **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.

When to use threads?

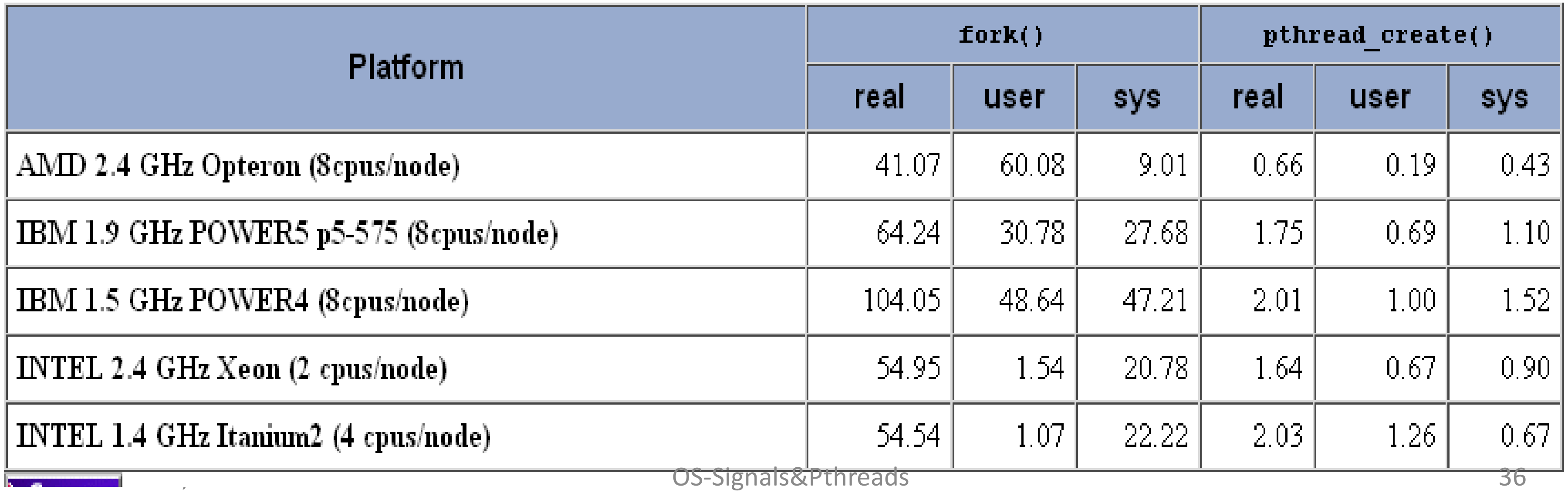
- Independent tasks
- Servers
- Repetitive tasks
- Asynchronous events



- **Considerations For Thread Programming**
  - Problem partitioning and complexity
  - Load balancing
  - Data dependencies
  - Synchronization and race conditions
  - Data communications
  - Memory, I/O issues

Pthreads Overview

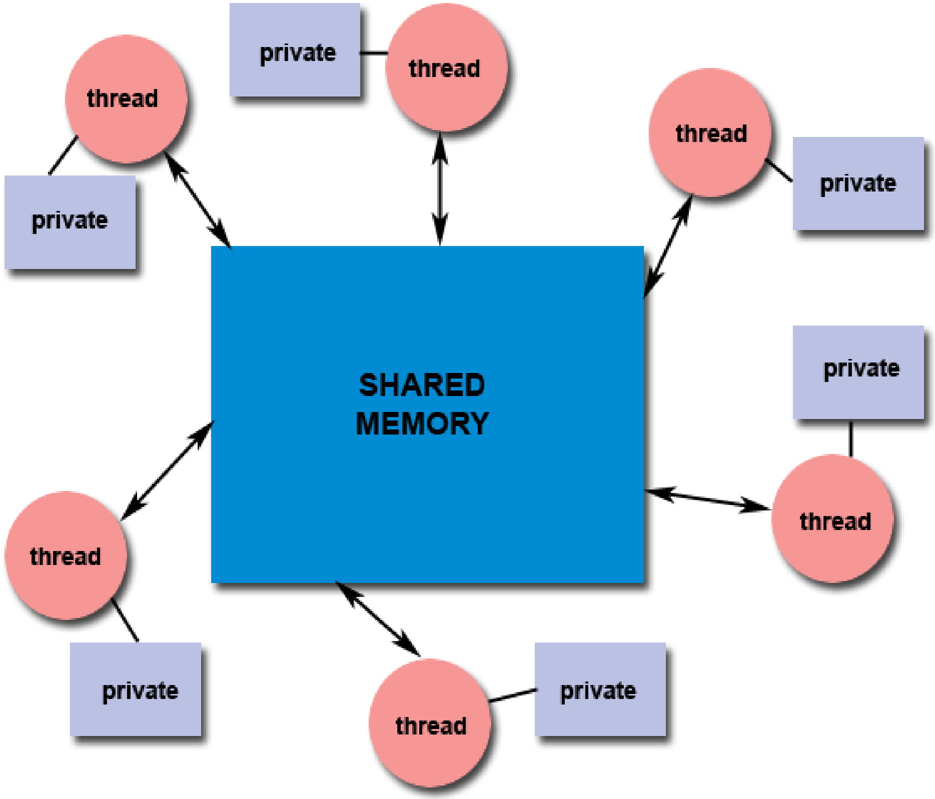
- **What are Pthreads.**
  - specified by the IEEE POSIX 1003.1c standard (1995).
  - set of C programming types & procedure calls, implemented with a pthread.h header file and a thread library.
- **Why Pthreads.**
  - Eg: 5000 threads/process creation



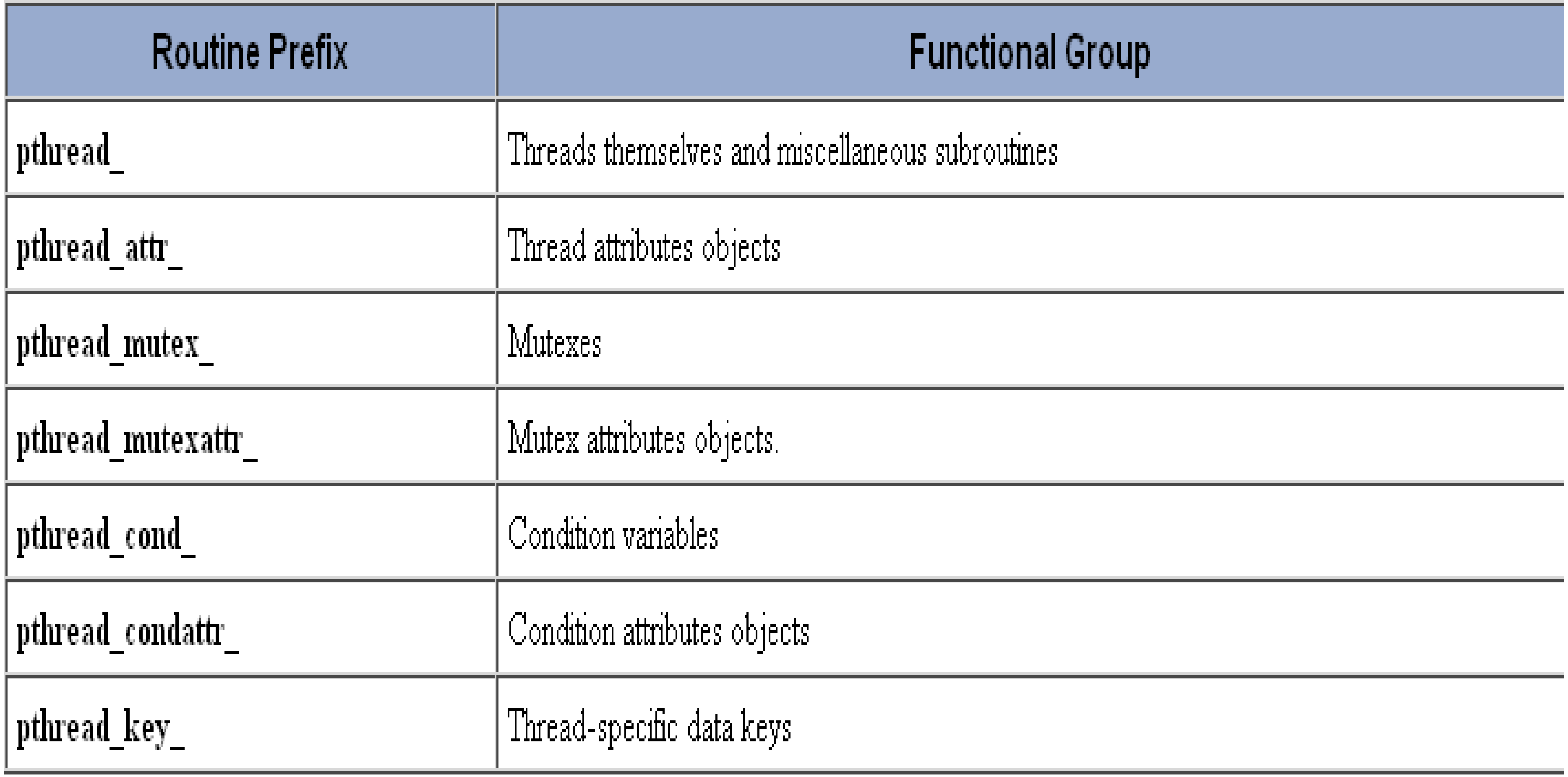
Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
AMD 2.4 GHz Opteron (8cpus/node)	41.07	60.08	9.01	0.66	0.19	0.43
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.24	30.78	27.68	1.75	0.69	1.10
IBM 1.5 GHz POWER4 (8cpus/node)	104.05	48.64	47.21	2.01	1.00	1.52
INTEL 2.4 GHz Xeon (2 cpus/node)	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

Thread Shared 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.

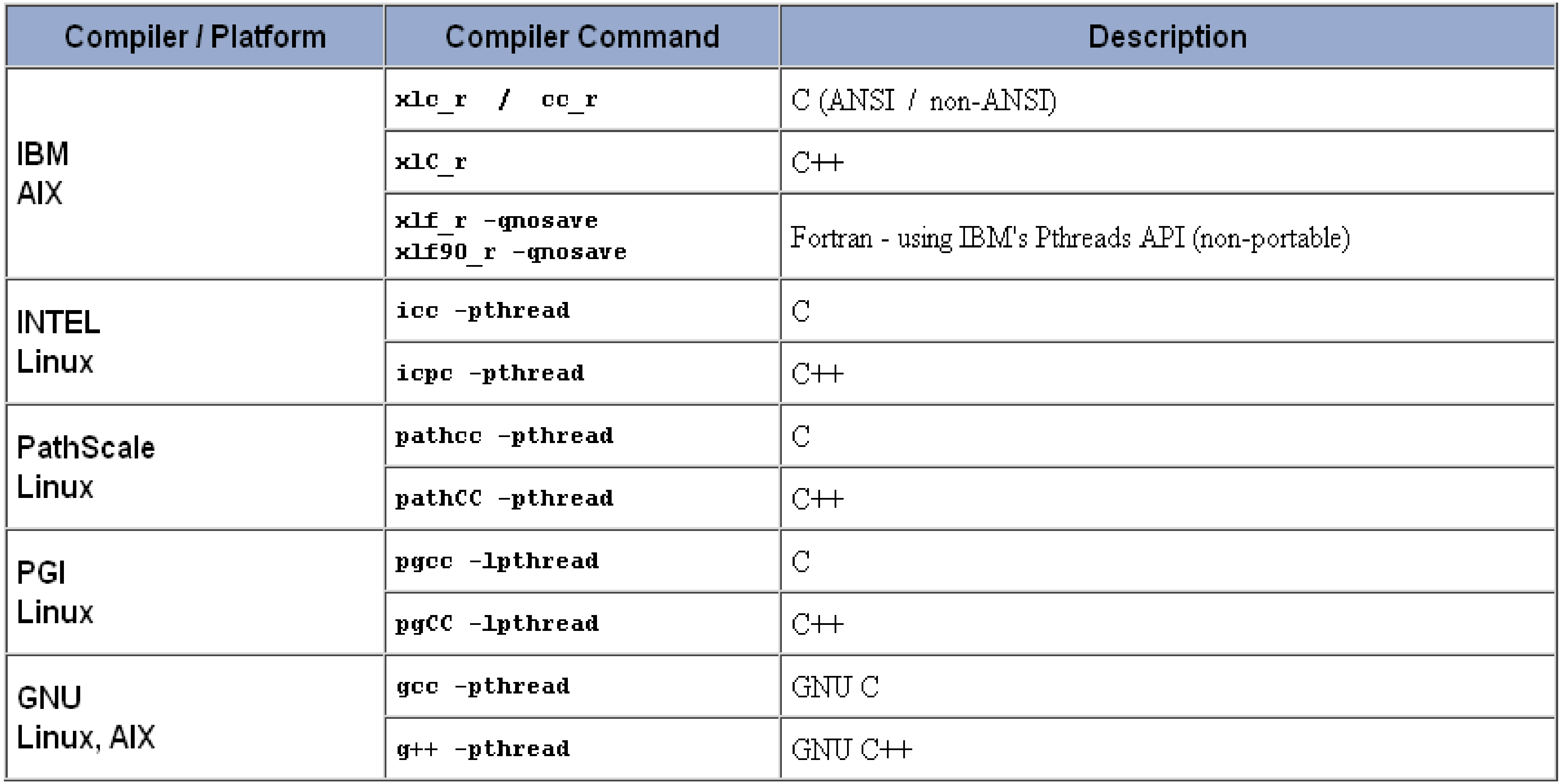


Naming convention



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

Compilation



Compiler / Platform	Compiler Command	Description
IBM AIX	`xlc_r / cc_r`	C (ANSI / non-ANSI)
	`xlC_r`	C++

	`xlf_r -qnosave`
	`xlf90_r -qnosave`
		Fortran - using IBM's Pthreads API (non-portable)

INTEL Linux	`icc -pthread`	C
	`icpc -pthread`	C++
PathScale Linux	`pathcc -pthread`	C
	`pathCC -pthread`	C++
PGI Linux	`pgcc -lpthread`	C
	`pgCC -lpthread`	C++
GNU Linux, AIX	`gcc -pthread`	GNU C
	`g++ -pthread`	GNU C++

Concept

- Concept of opaque objects pervades the design of API.
- Pthreads has over 100 subroutines
- For portability, pthread.h header file should be used for accessing pthread library.
- POSIX standard defined only for C language
- Once threads are created they are peers and may create other threads.
- Maximum number of threads created is implementation dependent.

1. Thread Management

- pthread_create (thread,attr,start_routine,arg)
- pthread_exit (status)
- pthread_attr_init (attr)
- pthread_attr_destroy (attr)
- pthread_join (threadid,status)
- pthread_detach (threadid,status)

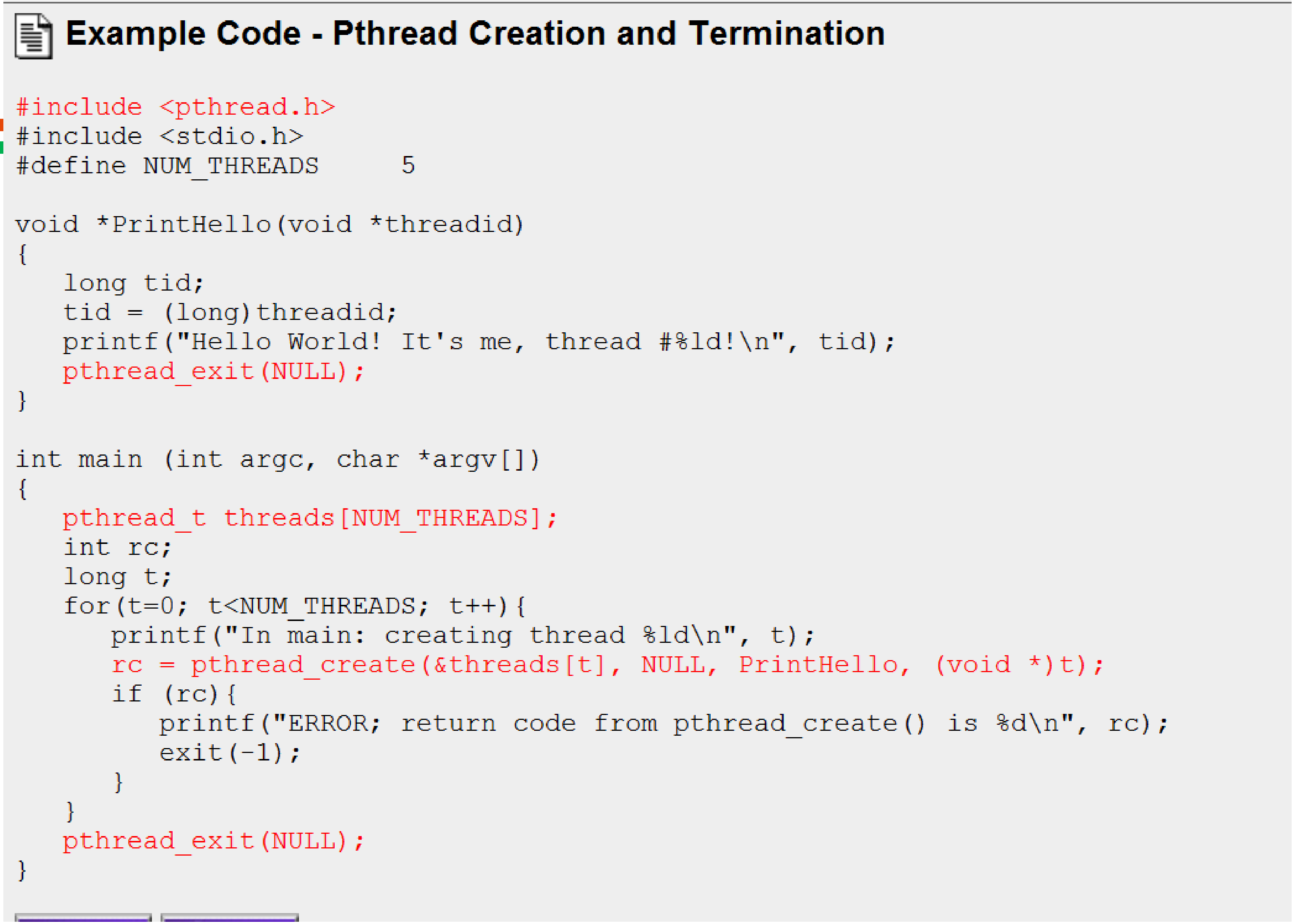
Pthread_create

- **pthread_create (thread, attr, start_routine, arg)**
  - creates a new thread and makes it executable.

- **thread: An unique identifier for the new thread returned by the subroutine.**
- **attr: An attribute object that may be used to set thread attributes. NULL for the default values.**
- **start_routine: the C routine 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.**

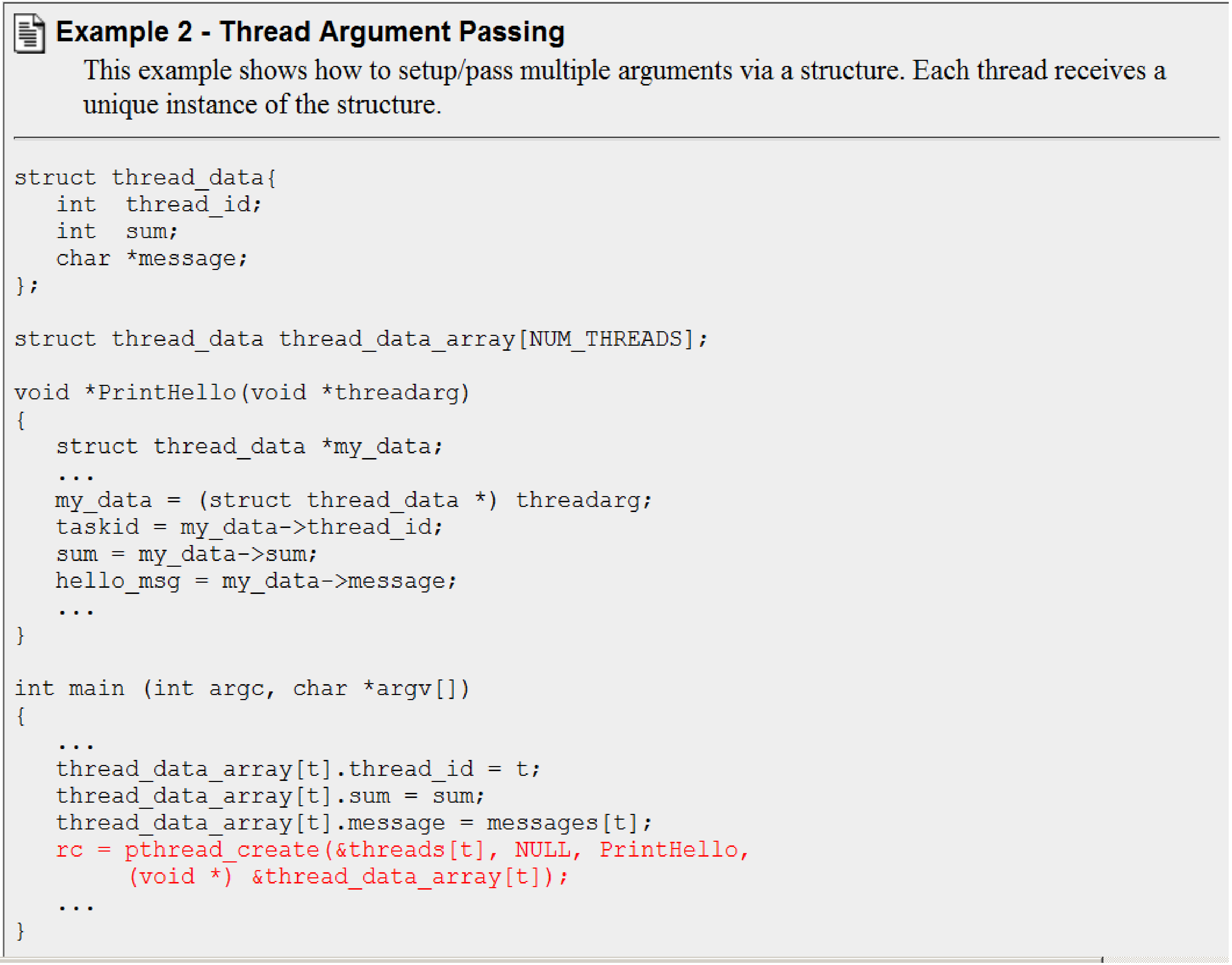
Termination

- Thread returns from main routine.
- Thread calls pthread_exit (status) . This is used to explicitly exit a thread
- the pthread_exit() routine does not close files; any files opened inside the thread will remain open after the thread is terminated.
- Thread is cancelled by other thread – pthread_cancel()
- Entire process is terminated.



Example Code - Pthread Creation and Termination

```
#include <pthread.h>
#include <stdio.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);
      }
   }
   pthread_exit(NULL);
}
```



Example 2 - Thread Argument Passing

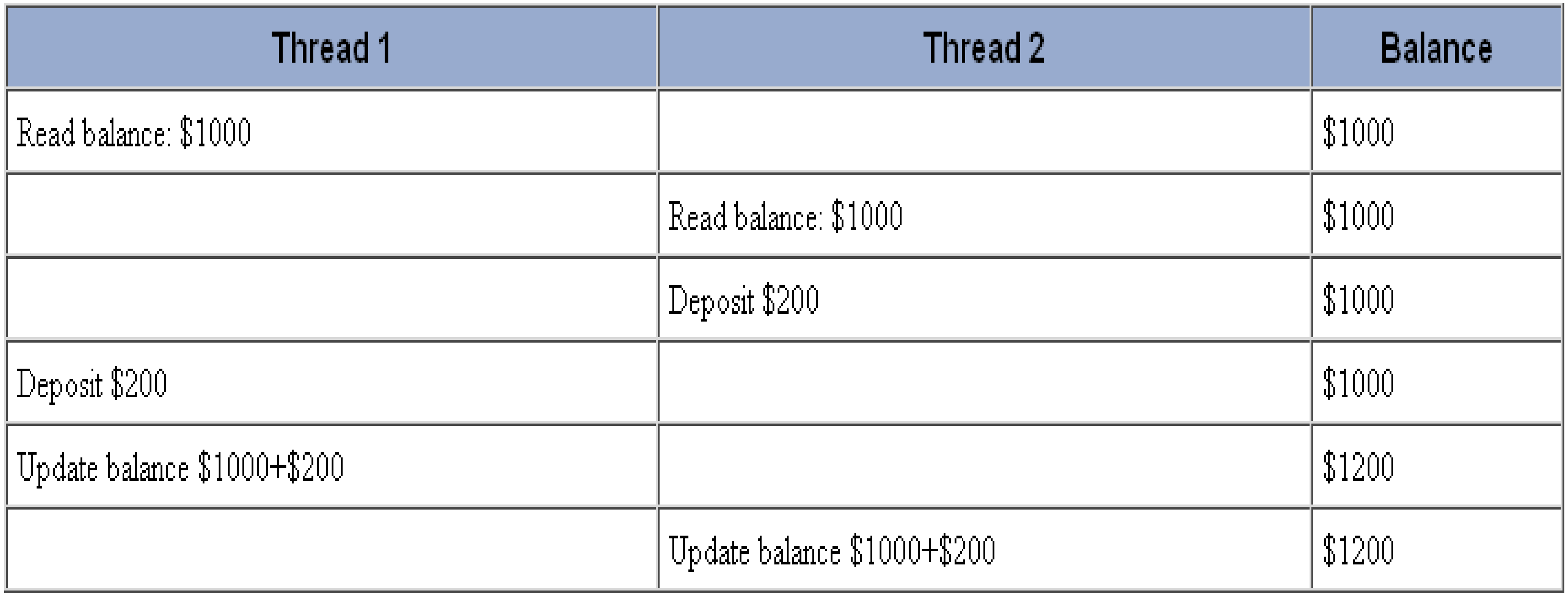
This example shows how to setup/pass multiple arguments via a structure. Each thread receives a unique instance of the structure.

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

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
   struct thread_data *my_data;
   ...
   my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
   hello_msg = my_data->message;
   ...
}

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

- **pthread_attr_getstacksize (attr, stacksize)**
- **pthread_attr_setstacksize (attr, stacksize)**
- **pthread_attr_getstackaddr (attr, stackaddr)**
- **pthread_attr_setstackaddr (attr, stackaddr)**

Mutex

- 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.
- Mutexes can be used to prevent "race" conditions.



Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200

Sequence

- 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

Mutex Routines

- pthread_mutex_init (mutex,attr)
- pthread_mutex_destroy (mutex)
- pthread_mutexattr_init (attr)
- pthread_mutexattr_destroy (attr)
- pthread_mutex_lock (mutex)
- pthread_mutex_trylock (mutex)
- pthread_mutex_unlock (mutex)

Thank You