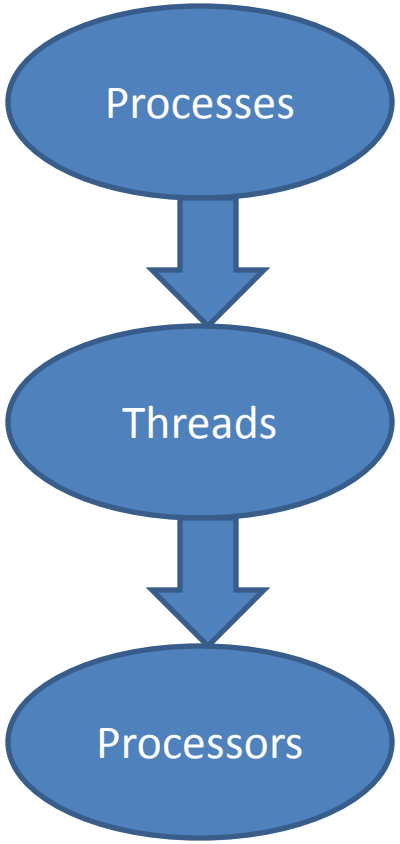
NYU
New York University

CSCI-GA.3033-010
Multicore Processors: Architecture & Programming

Lecture 5: Threads … Pthreads

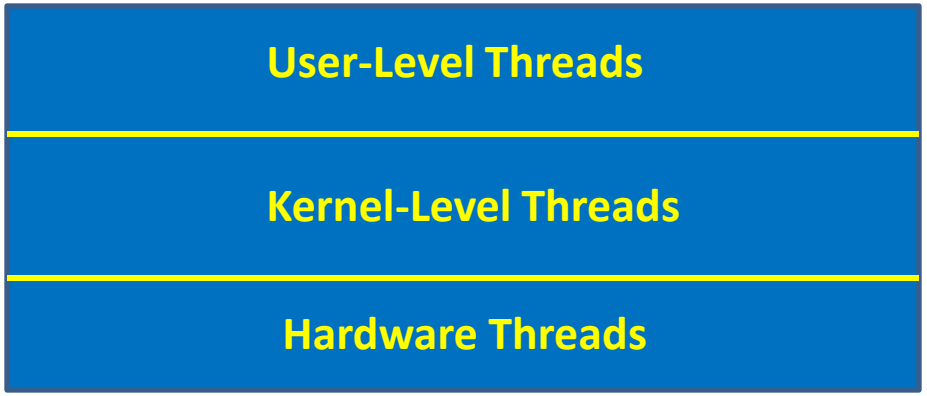
Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Multithreading



A Thread?

- **Definition**: sequence of related instructions executed independently of other instruction sequences
- A thread can create another thread
- Each thread maintains its current machine state



A Thread?

- Relationship between user-level and kernel-level threads
  - 1:1
    - user-level thread maps to kernel-level thread
    - e.g. win32, Linux (original C-library), windows 7, FreeBSD
  - N:1 (user-level threads)
    - Kernel is not aware of the existence of threads
    - e.g. Early version of Java, Solaris Green Thread
  - M:N
- Threads share same address space but have their own private stacks
- Thread states: ready, running, waiting (blocked), or terminated

Why Pthreads?

- To realize potential 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

POSIX Threads (Pthreads)

- Low-level threading libraries
- Native threading interface for Linux now
- Use kernel-level thread (1:1 model)
  - starting from kernel 2.6
- developed by the IEEE committees in charge of specifying a Portable Operating System Interface (POSIX)
- Shared memory

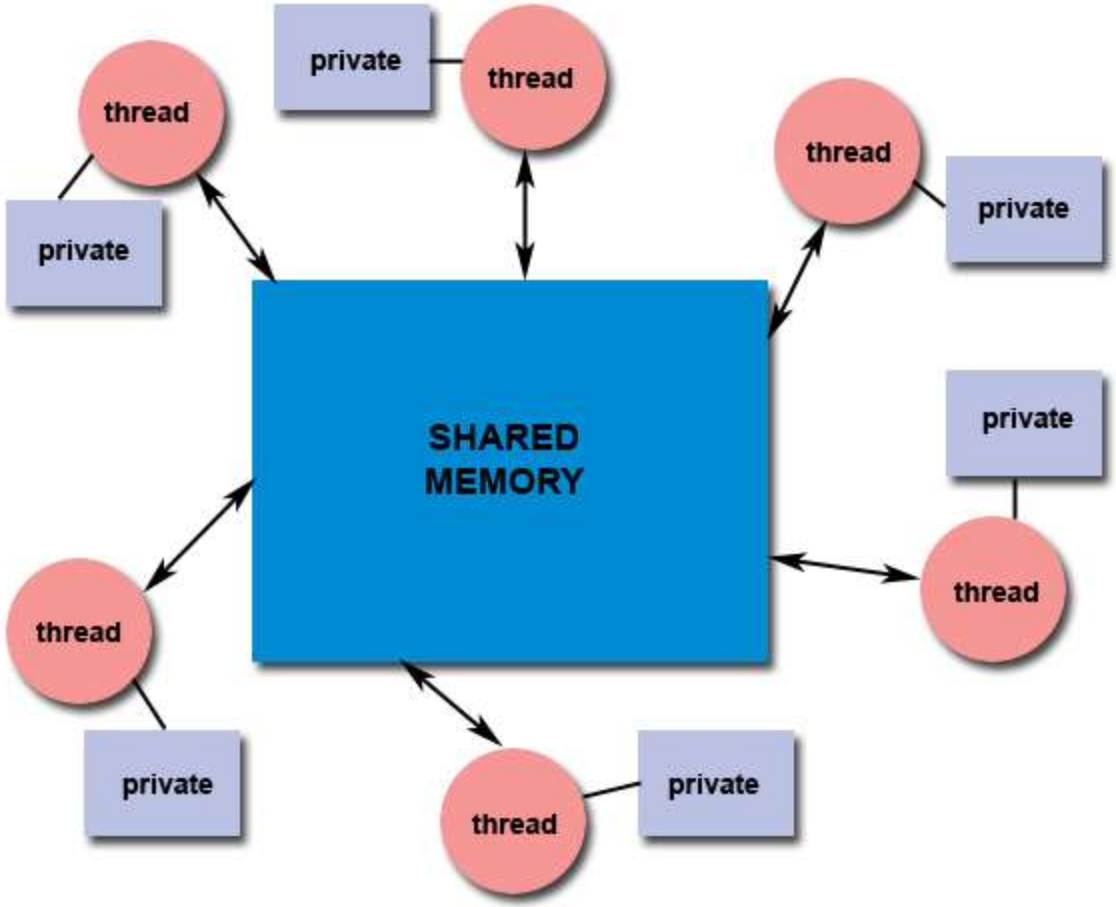
POSIX Threads (Pthreads)

- 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

POSIX Threads (Pthreads)

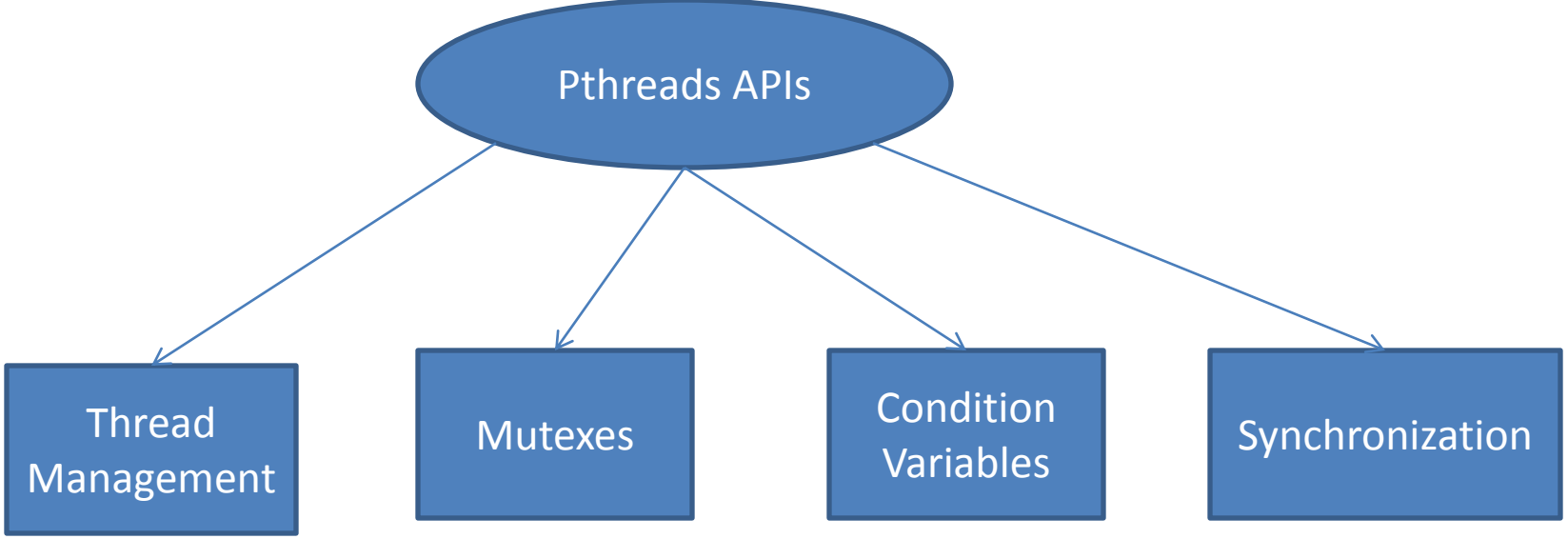
- C language programming types and procedure calls
- implemented with a pthread.h header
- To compile with GNU compiler, 2 methods:
  - gcc/g++ *progname* -lpthread
  - gcc/g++ -pthread *progname*
- Programmers are responsible for synchronizing access (protecting) globally shared data.
- Capabilities like thread priority are not part of the core pthreads library.

POSIX Threads (Pthreads)

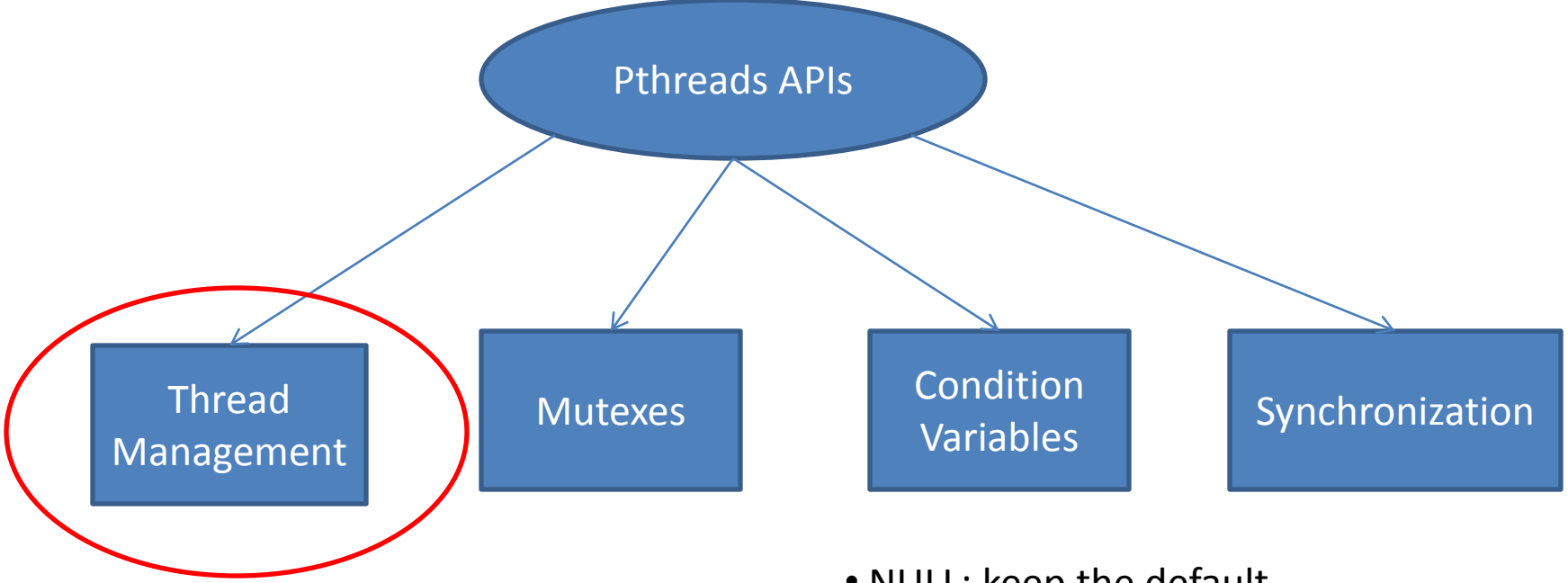


Source: https://computing.llnl.gov/tutorials/pthreads/

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

const int MAX_THREADS = 64;

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

void Usage(char* prog_name);
void *Hello(void* rank);  /* Thread function */

int main(int argc, char* argv[]) {
   long       thread;
   pthread_t* thread_handles;

   /* Get number of threads from command line */
   if (argc != 2) Usage(argv[0]);

   thread_count = strtol(argv[1], NULL, 10);
   if (thread_count <= 0 || thread_count >
      MAX_THREADS)
      Usage(argv[0]);

   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;

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

   return NULL;
}  /* Hello */


void Usage(char* prog_name) {
   fprintf(stderr, "usage: %s <number of threads>\n",
      prog_name);
   fprintf(stderr, "0 < number of threads <= %d\n",
      MAX_THREADS);
   exit(0);
}  /* Usage */
```



More than 100 subroutines!



- NULL: keep the default
- specified only at thread creation time
- The main steps in setting attributes:
  - pthread_attr_t tattr
  - pthread_attr_init(&tattr)
  - pthread_attr_*(&tattr,SOME_ATRIBUTE_VALUE_PARAMETER)

```
pthread_create(pthread_t *,
               const pthread_attr_t *,
               void *(*start_routine)(void*),
               void * arg)
```

Threads Cheaper than Processes

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

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

void *print_message_function( void *ptr );

main()
{
    pthread_t thread1, thread2;
    char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    int  iret1, iret2;

    iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
    iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    printf("Thread 1 returns: %d\n",iret1);
    printf("Thread 2 returns: %d\n",iret2);
    exit(0);
}

void *print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
}
```

Wait until thread returns

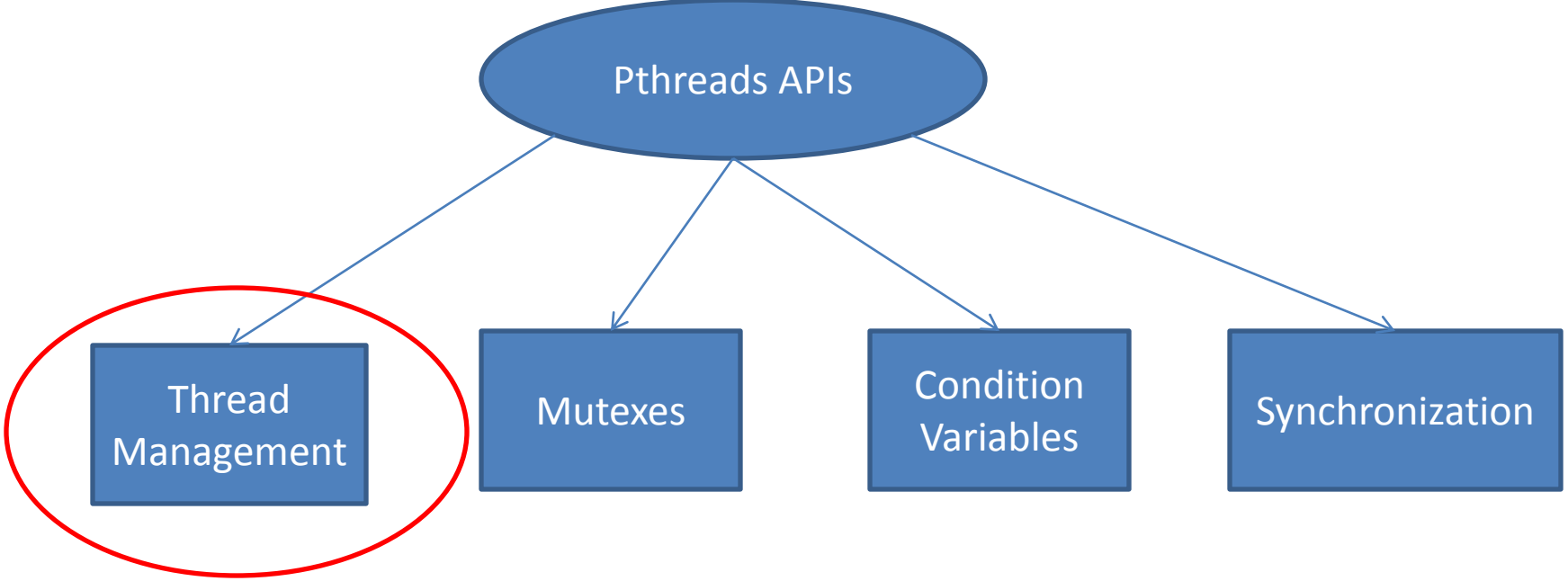
Threads terminate by:

- explicitly calling pthread_exit
- letting the function return
- a call to the function exit which will terminate the process including any threads.
- canceled by another thread via the pthread_cancel routine

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

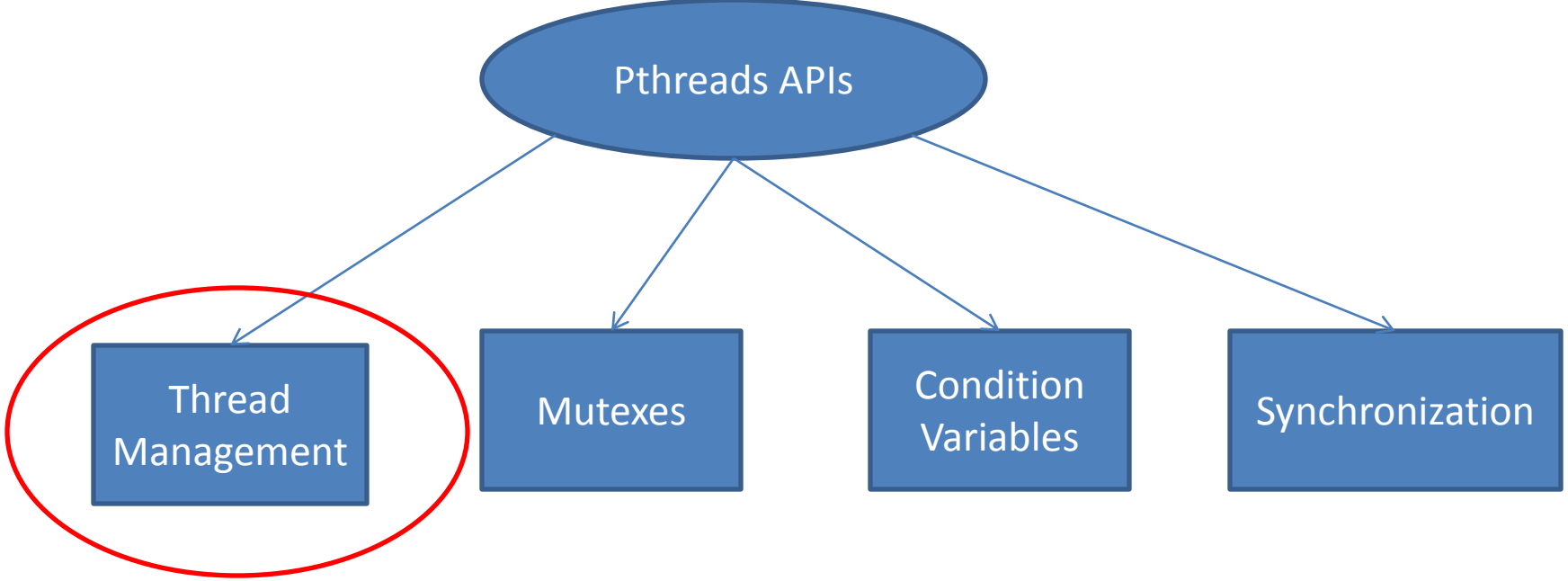
By having main() explicitly call pthread_exit() as the last thing it does, main() will block and be kept alive to support the threads it created until they are done.

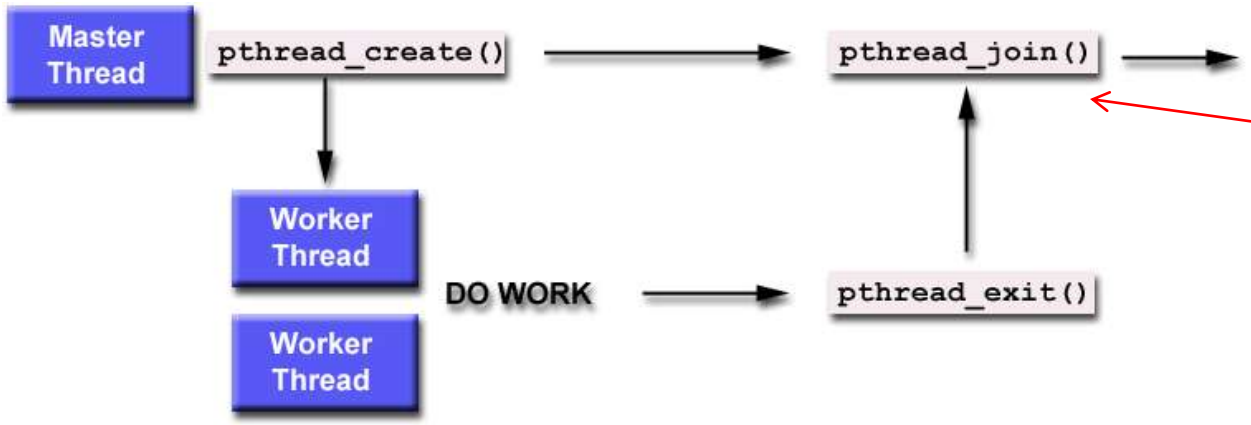
If you don't call `pthread_exit()` explicitly, when main() completes, the process (and all threads) will be terminated



What is wrong about the following code?

```
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);
    ... }
```





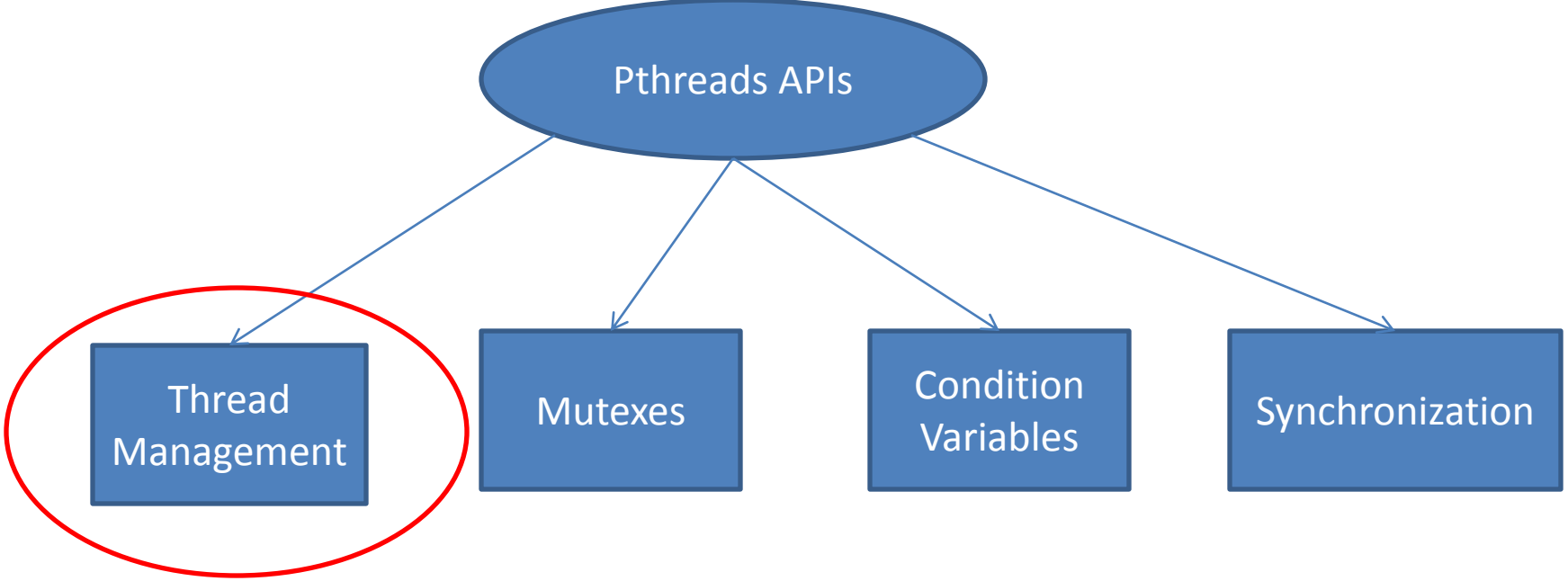
The pthread_join() subroutine blocks the calling thread until the specified thread id thread terminates.

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

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);
}
```

Important!

- Call **pthread_join**() or **pthread_detach**() for every thread that is created joinable
  - so that the system can reclaim all resources associated with the thread
  - int pthread_detach(pthread_t threadid); indicates that system resources for the specified thread should be reclaimed when the thread ends

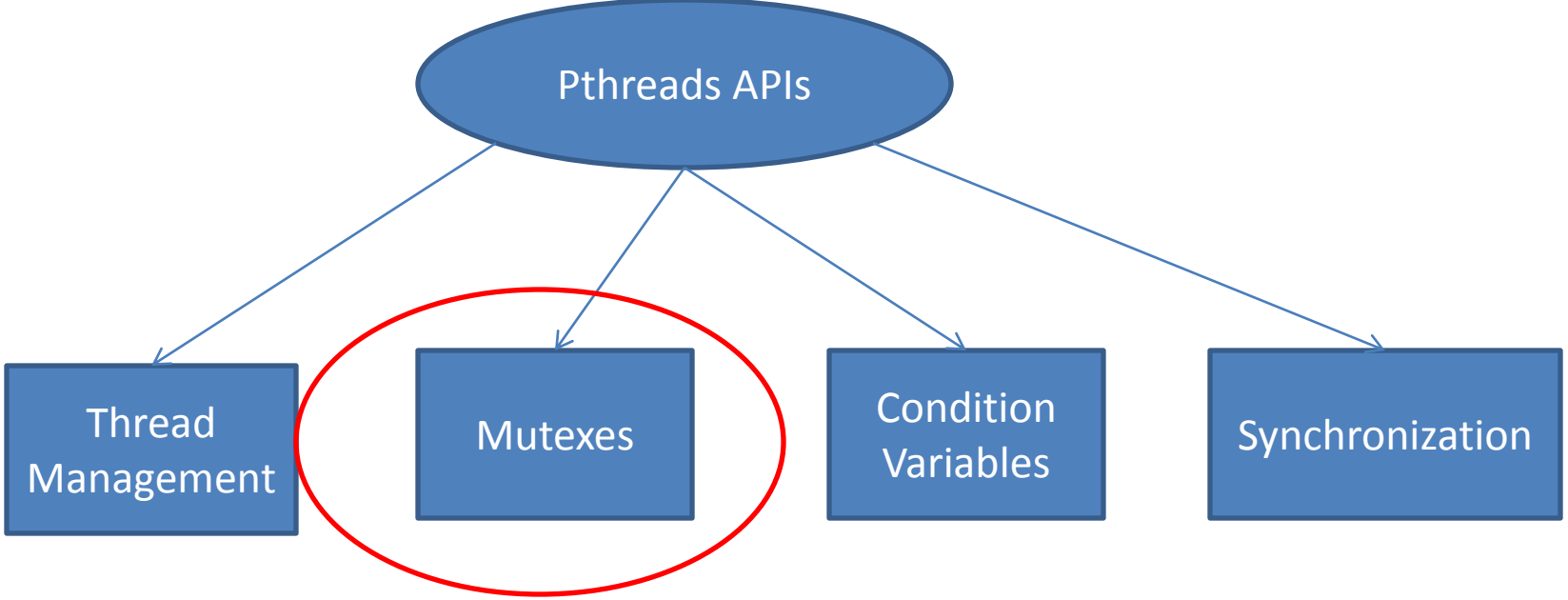
- Failure to join or to detach threads →memory and other resource leaks until the process ends



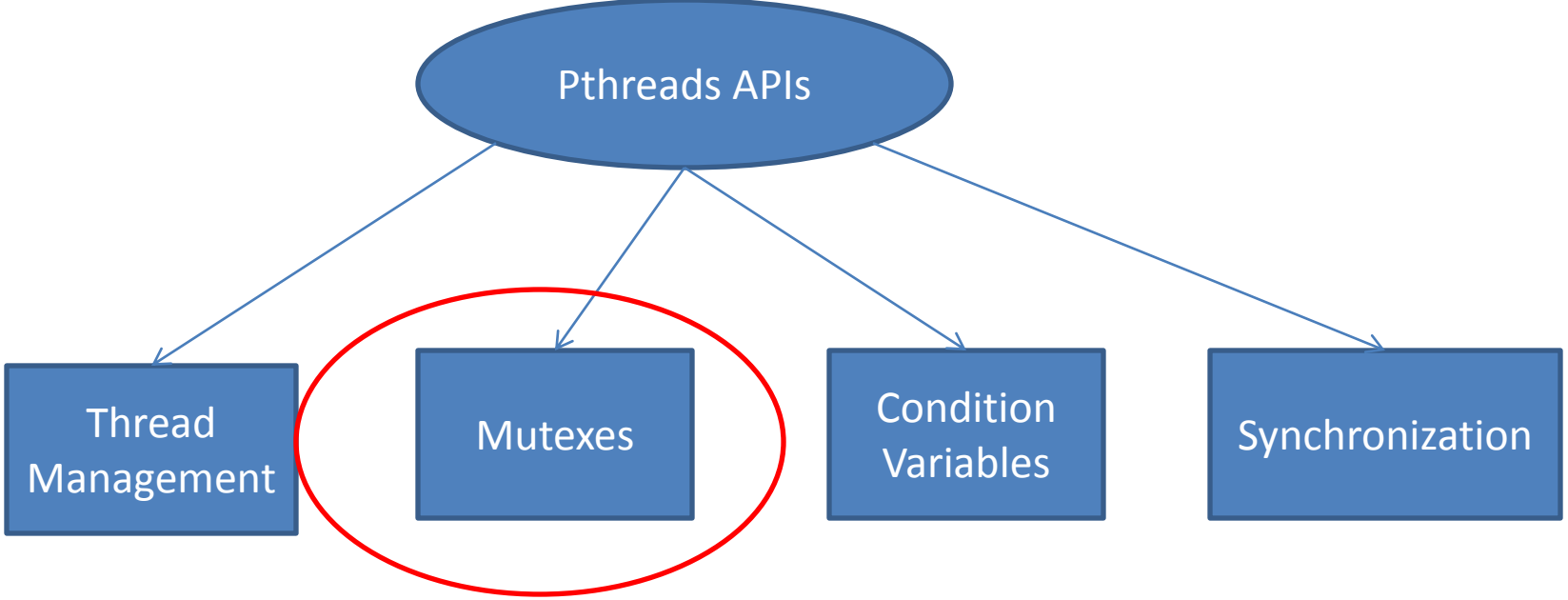
How about the stack?

- Default thread stack size varies greatly.
- Safe and portable programs do not depend upon the default stack limit

```c
#include <pthread.h>
#include <stdio.h>
#define NTHREADS 4
#define N 1000
#define MEGEXTRA 1000000

pthread_attr_t attr;

void *dowork(void *threadid)
{
   double A[N][N];
   int i,j;
   long tid;
   size_t mystacksize;

   tid = (long)threadid;
   pthread_attr_getstacksize (&attr, &mystacksize);
   printf("Thread %ld: stack size = %li bytes \n", tid, mystacksize);
   for (i=0; i<N; i++)
     for (j=0; j<N; j++)
      A[i][j] = ((i*j)/3.452) + (N-i);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
   pthread_t threads[NTHREADS];
   size_t stacksize;
   int rc;
   long t;

   pthread_attr_init(&attr);
   pthread_attr_getstacksize (&attr, &stacksize);
   printf("Default stack size = %li\n", stacksize);
   stacksize = sizeof(double)*N*N+MEGEXTRA;
   printf("Amount of stack needed per thread = %li\n",stacksize);
   pthread_attr_setstacksize (&attr, stacksize);
   printf("Creating threads with stack size = %li bytes\n",stacksize);
   for(t=0; t<NTHREADS; t++){
      rc = pthread_create(&threads[t], &attr, dowork, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   printf("Created %ld threads.\n", t);
   pthread_exit(NULL);
}
```



- Mutex = Mutual Exclusion
- One of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur.
- acts like a lock protecting access to a shared data resource
- only one thread can lock (or own) a mutex variable at any given time.



A typical sequence in the use of a mutex is as follows:

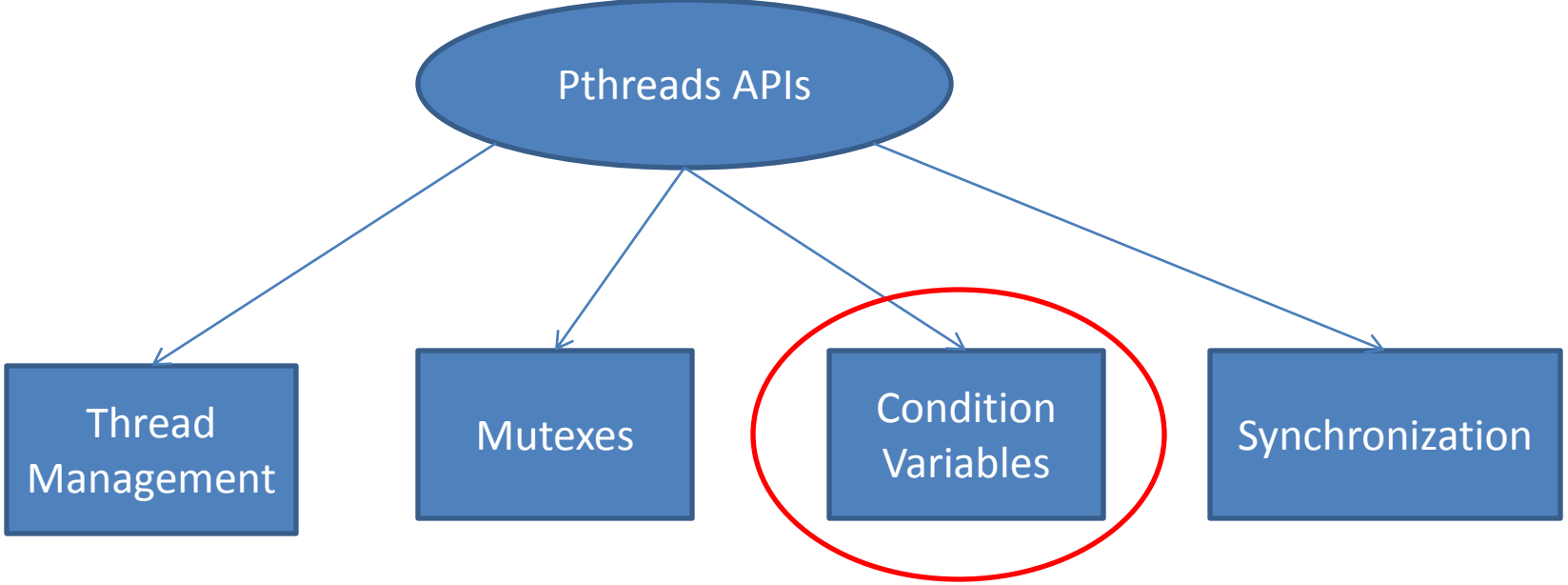
- Create and initialize a mutex variable
- Several threads attempt to lock the mutex
- Only one succeeds and that thread owns the mutex
- The owner thread performs some set of actions
- The owner unlocks the mutex
- Another thread acquires the mutex and repeats the process
- Finally the mutex is destroyed

It is up to the code writer to insure that the necessary threads all make the the mutex lock and unlock calls correctly.

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

/*
The following structure contains the necessary information
to allow the function "dotprod" to access its input data and
place its output into the structure.
*/

typedef struct
 {
   double      *a;
   double      *b;
   double     sum;
   int     veclen;
 } DOTDATA;

/* Define globally accessible variables and a mutex */

#define NUMTHRDS 4
#define VECLEN 100
   DOTDATA dotstr;
   pthread_t callThd[NUMTHRDS];
   pthread_mutex_t mutexsum;
```

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

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

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

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

   /* Assign storage and initialize values */
   a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
   b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));

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

   dotstr.veclen = VECLEN;
   dotstr.a = a;
   dotstr.b = b;
   dotstr.sum=0;

   pthread_mutex_init(&mutexsum, NULL);

           -----------

    pthread_mutex_destroy(&mutexsum);
    pthread_exit(NULL);
}
```

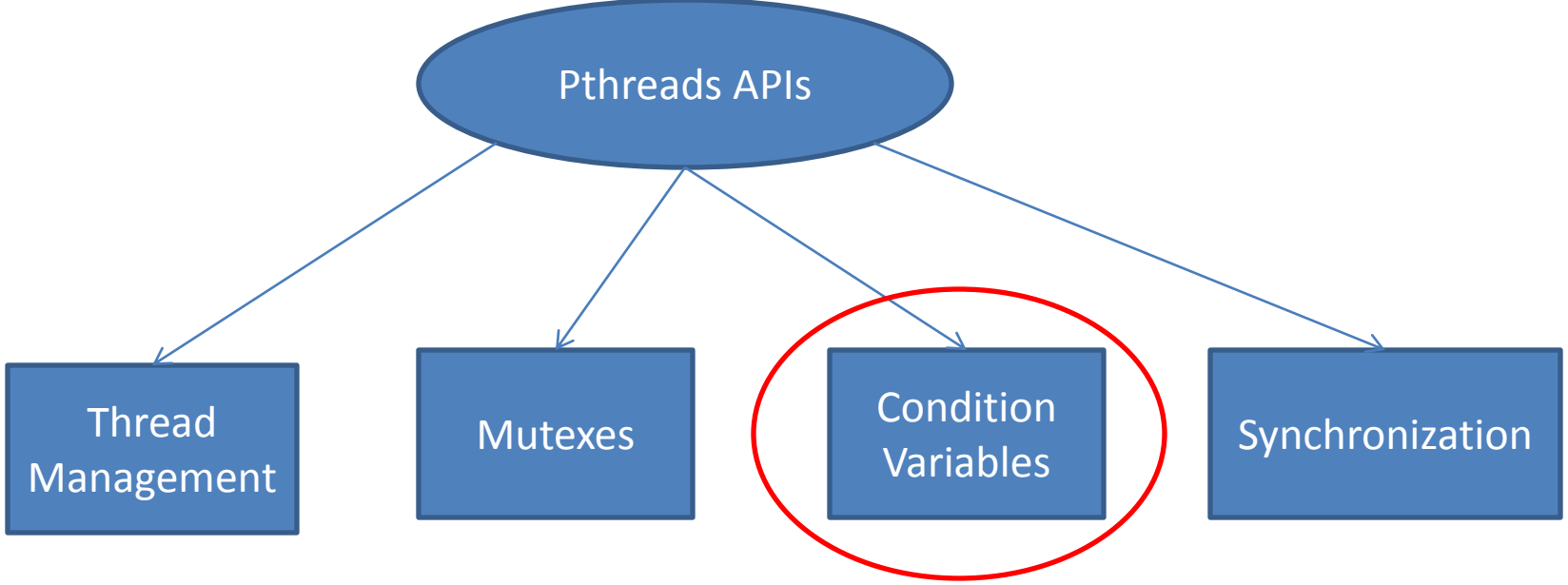


- While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
- A condition variable is always used in conjunction with a mutex lock.
- Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. This is very resource consuming since the thread would be continuously busy in this activity.

A condition variable is a way to achieve the same goal without polling

Steps for Using Condition Variables

- Main thread:
  - Declare and initialize global data/variables which require synchronization
  - Declare and initialize a condition variable
  - Declare and initialize an associated mutex
  - Create threads A and B to do work
- Thread A
  - Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)
  - Lock associated mutex and check value of a global variable
  - Call pthread_cond_wait() to perform a blocking wait for signal from Thread-B.
    - a call to pthread_cond_wait() automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.
  - When signalled, wake up. Mutex is automatically and atomically locked.
  - Explicitly unlock mutex
  - Continue
- Thread B
  - Do work
  - Lock associated mutex
  - Change the value of the global variable that Thread-A is waiting upon.
  - Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.
  - Unlock mutex.
  - Continue

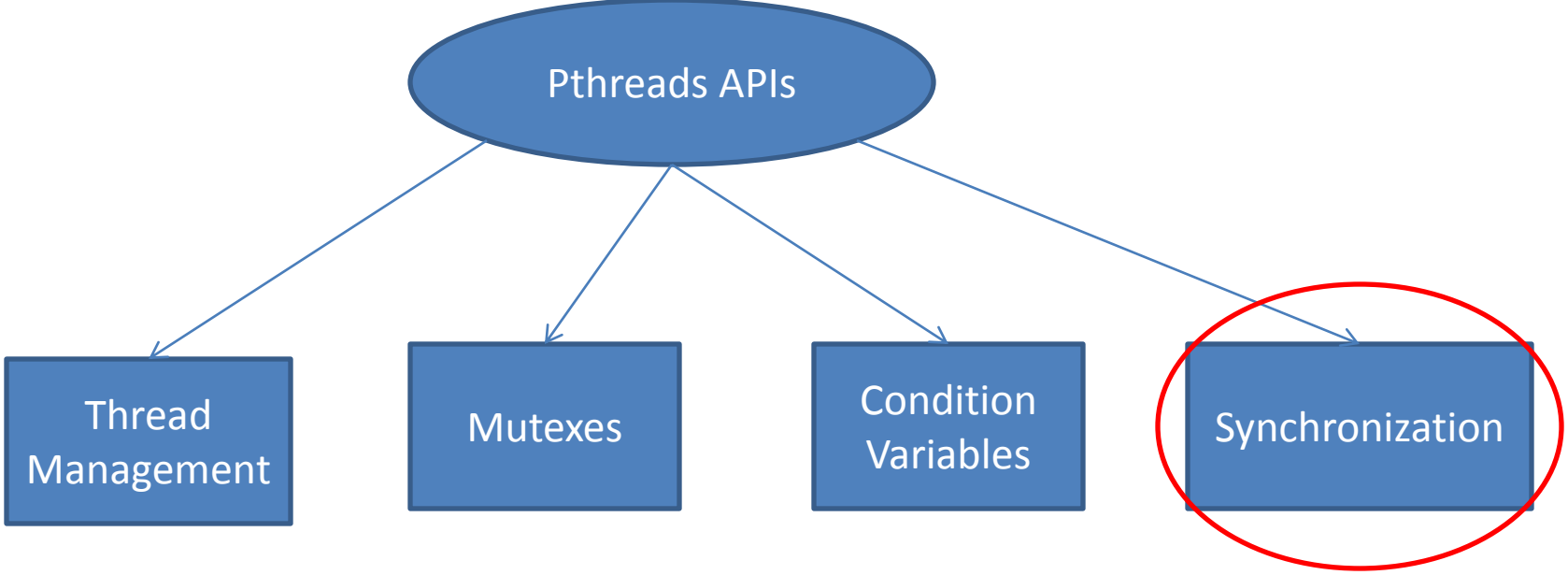


- Condition variables must be declared with type pthread_cond_t
- must be initialized before they can be used.
- There are two ways to initialize a condition variable:
  - Statically, when it is declared. For example:
    pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;
  - Dynamically, with the pthread_cond_init() routine.
- pthread_cond_destroy() should be used to free a condition variable that is no longer needed.

```
int count = 0;
int thread_ids[3] = {0,1,2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *t) {
        int i;
        long my_id = (long)t;

        for (i=0; i<TCOUNT; i++) {
                pthread_mutex_lock(&count_mutex);
                count++;
                if (count == COUNT_LIMIT)
                                pthread_cond_signal(&count_threshold_cv);
                pthread_mutex_unlock(&count_mutex);

                /* Do some "work" so threads can alternate on mutex lock */
                sleep(1); }
pthread_exit(NULL);
}

void *watch_count(void *t) {
                long my_id = (long)t;
                pthread_mutex_lock(&count_mutex);

            while (count<COUNT_LIMIT) {
                                pthread_cond_wait(&count_threshold_cv, &count_mutex);
                        count += 125;
                 }
                pthread_mutex_unlock(&count_mutex);
                pthread_exit(NULL);
}
```

unblocks threads blocked for that condition variable

block on the condition variable



Definition: Synchronization is an enforcing mechanism used to impose constraints on the order of execution of threads, in order to coordinate thread execution and manage shared data.

By now you must have realized that we have 3 synchronization mechanisms:

- Mutexes
- Condition variables
- joins

Semaphores

- permit a limited number of threads to execute a section of the code
- similar to mutexes
- should include the semaphore.h header file
- semaphore functions have sem_ prefixes

Basic Semaphore functions

- creating a semaphore:
  - int sem_init(sem_t *sem, int pshared, unsigned int value)
  - initializes a semaphore object pointed to by sem
  - pshared is a sharing option; a value of 0 means the semaphore is local to the calling process
  - gives an initial value value to the semaphore
- terminating a semaphore:
  - int sem_destroy(sem_t *sem)
  - frees the resources allocated to the semaphore sem
  - usually called after pthread_join()

Basic Semaphore functions

- int sem_post(sem_t *sem)
  - atomically increases the value of a semaphore by 1, i.e., when 2 threads call sem_post simultaneously, the semaphore's value will also be increased by 2 (there are 2 atoms calling)
- int sem_wait(sem_t *sem)
  - atomically decreases the value of a semaphore by 1
  - If the value is 0 then the thread will block waiting it to become 1

Semaphores

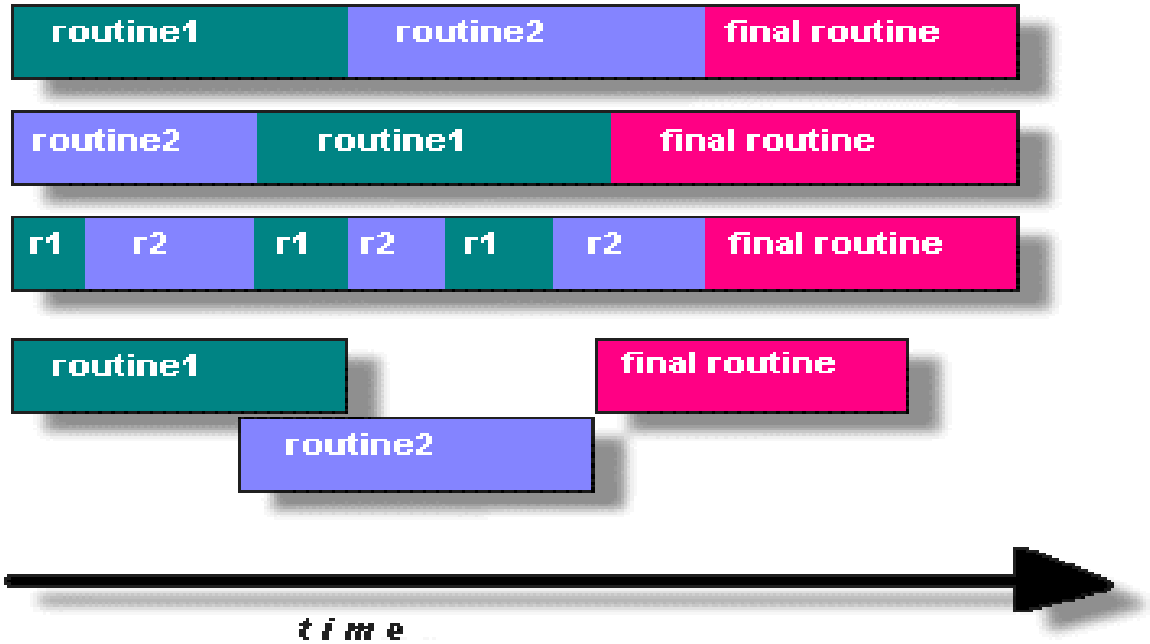
- Only positive values (or 0)
- Its operations are atomic
- Cannot read it except at initializations
- Main usage
  - mutual exclusion
  - synchornization

```
#include <pthread.h>
#include <semaphore.h>
...
void *thread_function( void *arg );
...
sem_t semaphore;        // also a global variable just like mutexes
...
int main()
{
        int tmp;
        ...
        // initialize the semaphore
        tmp = sem_init( &semaphore, 0, 0 );
        ...
        // create threads
        pthread_create( &thread[i], NULL, thread_function, NULL );
        ...
        while ( still_has_something_to_do() )
        {
             sem_post( &semaphore );
             ...
        }
        ...
        pthread_join( thread[i], NULL );
        sem_destroy( &semaphore );
        return 0;
}
```

```
void *thread_function( void *arg )
{
        sem_wait( &semaphore );
        perform_task_when_sem_open();
        ...
        pthread_exit( NULL );
}
```

Parallel Programming

- To take advantage of Pthreads, a program must be able to be organized into discrete, independent tasks which can execute concurrently
- If routine1 and routine2 can be interchanged, interleaved and/or overlapped in real time, they are candidates for threading.



Parallel Programming Model

- 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:
    - static worker pool
    - dynamic worker pool

Parallel Programming Model

- 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

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

The Problem With Threads

- Paper by Edward Lee, 2006
- The author argues:
  - "From a fundamental perspective, threads are seriously flawed as a computation model"
  - "Achieving reliability and predictability using threads is essentially impossible for many applications"
- The main points:
  - Our abstraction for concurrency does not even vaguely resemble the physical world.
  - Threads are dominating but not the best approach in every situation
  - Yet threads are suitable for embarrassingly parallel applications

The Problem With Threads

- The logic of the paper:
  - Threads are nondeterministic
  - Why shall we use nondeterministic mechanisms to achieve deterministic aims??
  - The job of the programmer is to prune this nondeterminism.
  - This leads to poor results

Do you agree or disagree with the author ?

Conclusions

- Processes → threads → processors
- User-level threads and kernel-level threads are not the same but they have direct relationship
- Pthreads assume shared memory