NYU
New York University

Operating Systems

Concurrency - II

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

Some slides of this lecture are from:
- Peter S. Pacheco book

Today's Lecture:

A small glimpse on threading APIs

POSIX Threads

- **P**ortable **O**perating **S**ystem **I**nterface
- Is an IEEE standard
- API
- Maintain compatibilities among OSes
- Pthreads → a POSIX standard for threads

POSIX Threads (Pthreads)

- Low-level threading libraries
- Native threading interface for Linux now
- Use kernel-level thread (1:1 model)
- 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)

- Implemented with a pthread.h header
  - i.e. #include <pthread.h>
- 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.

Hello World!

```c
#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 */
```

```c
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

. /helloworld <number of threads>

. /helloworld 1

```
Hello from the main thread
Hello from thread 0 of 1
```

. /helloworld 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
```

A closer look

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

If you want to keep the default, just use NULL.

Allocate before calling.

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

The function that the thread is to run.

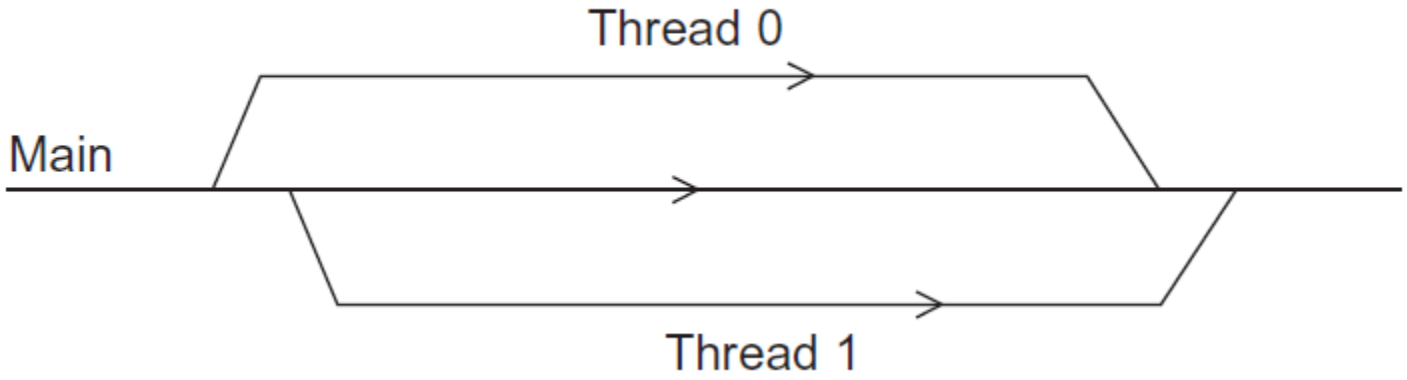
Function started by pthread_create

- 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.

Stopping the Threads

- We call the function pthread_join once for **each thread**.
- Blocking function.

```
int pthread_join(
      pthread_t   thread;          /* in */
      void * *      ret_val; /*out*/ );
```

Estimating π

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots\right)$$

```
double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i++, factor = -factor) {
    sum += factor/(2*i+1);
}
pi = 4.0*sum;
```

A thread function for computing π

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)  /* my_first_i is even */
      factor = 1.0;
   else  /* my_first_i is odd */
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      sum += factor/(2*i+1);
   }

   return NULL;
}  /* Thread_sum */
```

Using a dual core processor

	n			
	10^5	10^6	10^7	10^8
π	3.14159	3.141593	3.1415927	3.14159265
1 Thread	3.14158	3.141592	3.1415926	3.14159264
2 Threads	3.14158	3.141480	3.1413692	3.14164686

Note that as we increase n, the estimate with one thread gets better and better!!

Reason: Race Condition in updating sum by more than one thread.

Solution: Busy Waiting

flag initialized to 0 by main thread

```
y = Compute(my_rank);
while (flag != my_rank);
x = x + y;
flag++;
```

Executed by all threads

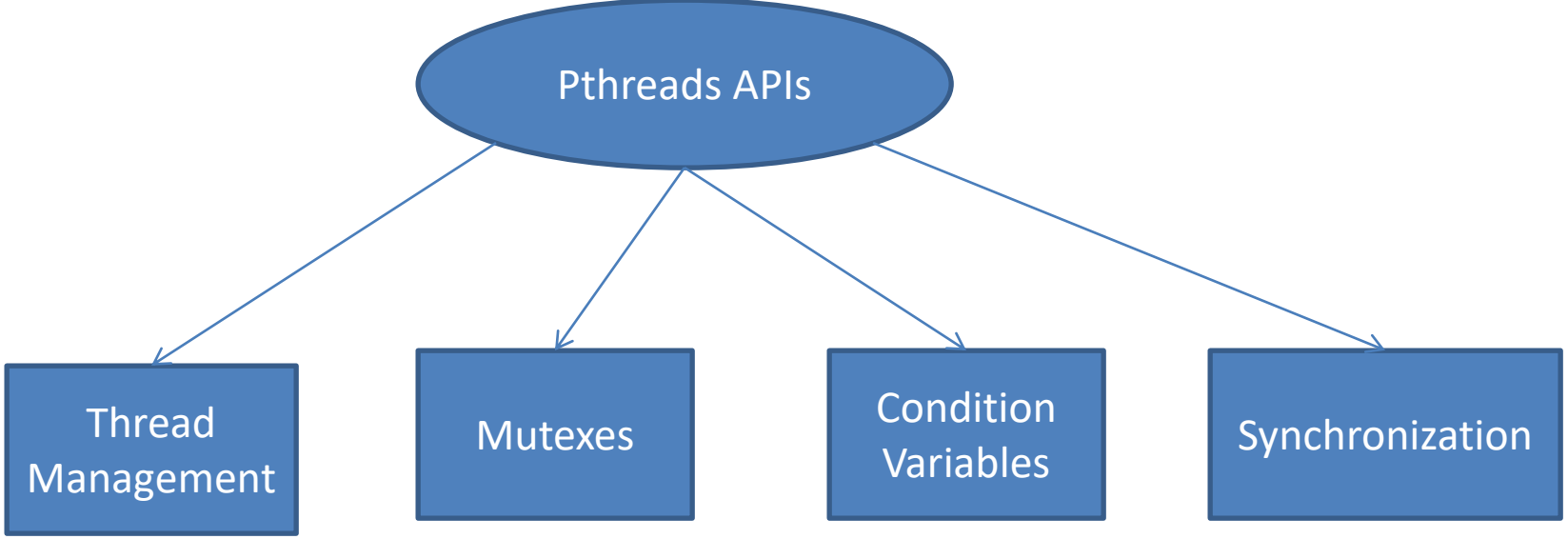
- This will ensure no race condition … But:
  - Busy doing nothing
  - serialization
  - optimizing compilers can mess with it!

Pthreads global sum with busy-waiting

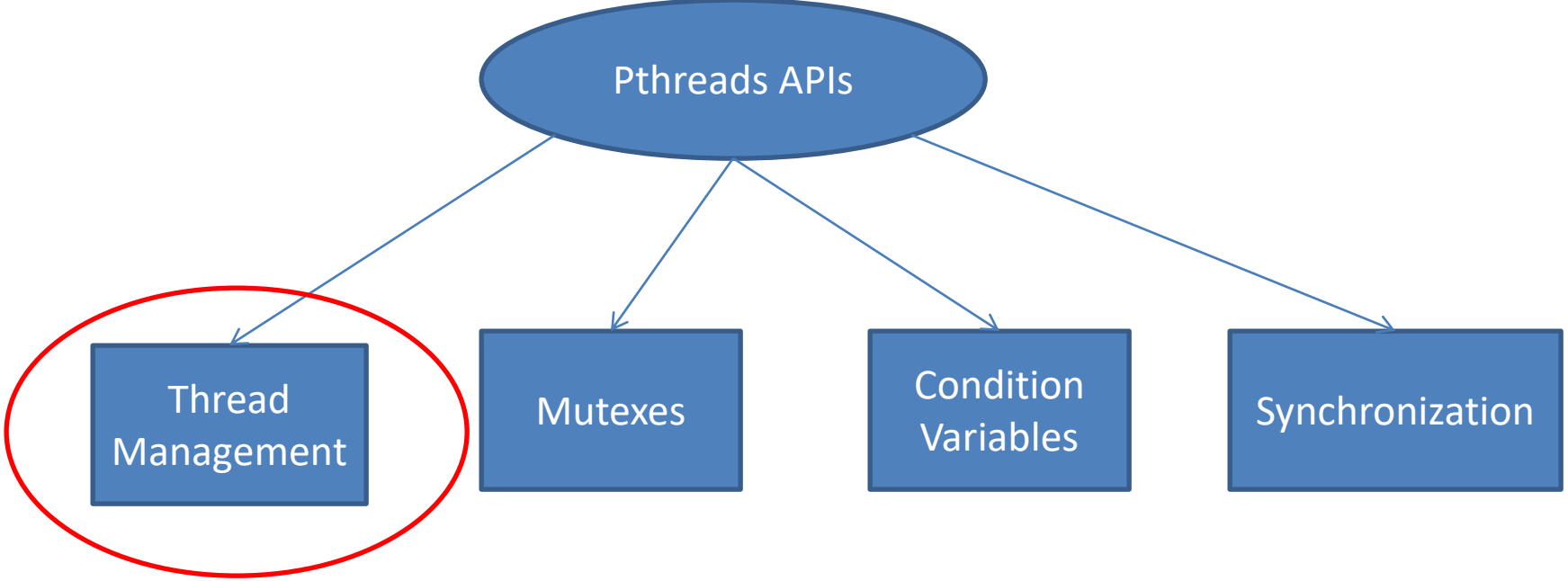
```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      while (flag != my_rank);
      sum += factor/(2*i+1);
      flag = (flag+1) % thread_count;
   }

   return NULL;
} /* Thread_sum */
```

In dual core
with n = 10^8
serial → 2.8s
2 threads → 19.5s!

Busy waiting is not the best solution if we need performance.



More than 100 subroutines!




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

```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

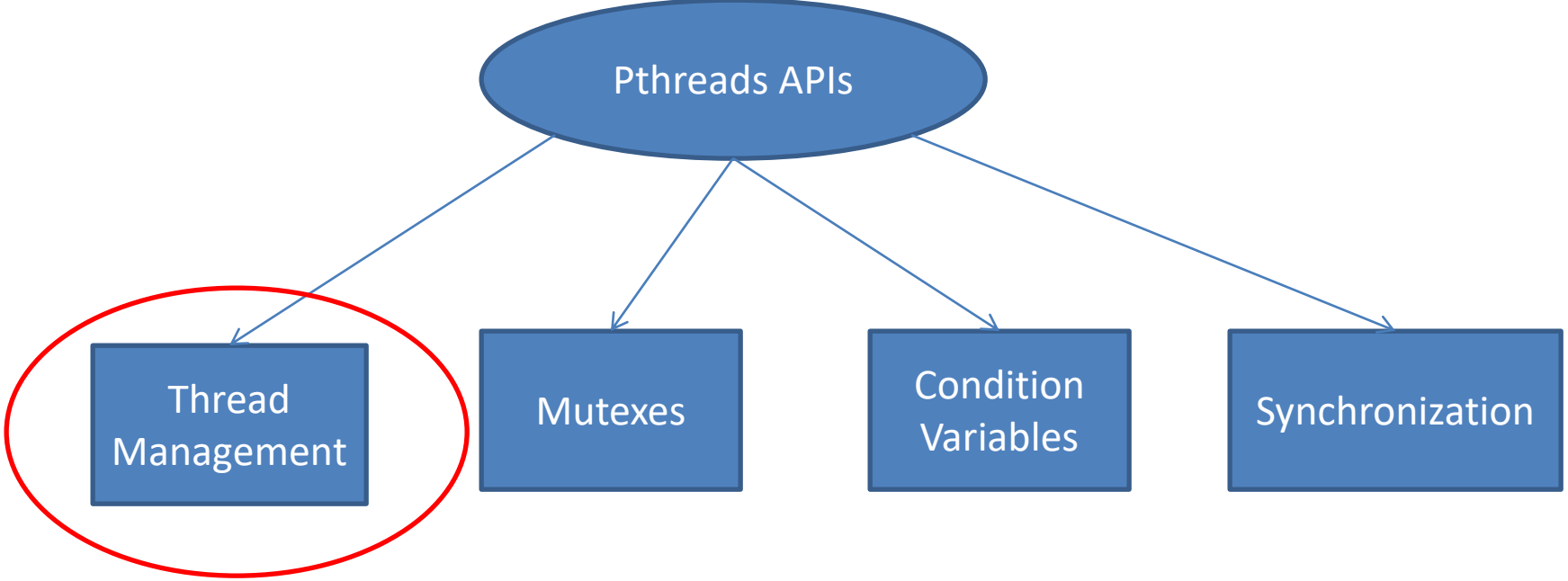
Threads terminate by:

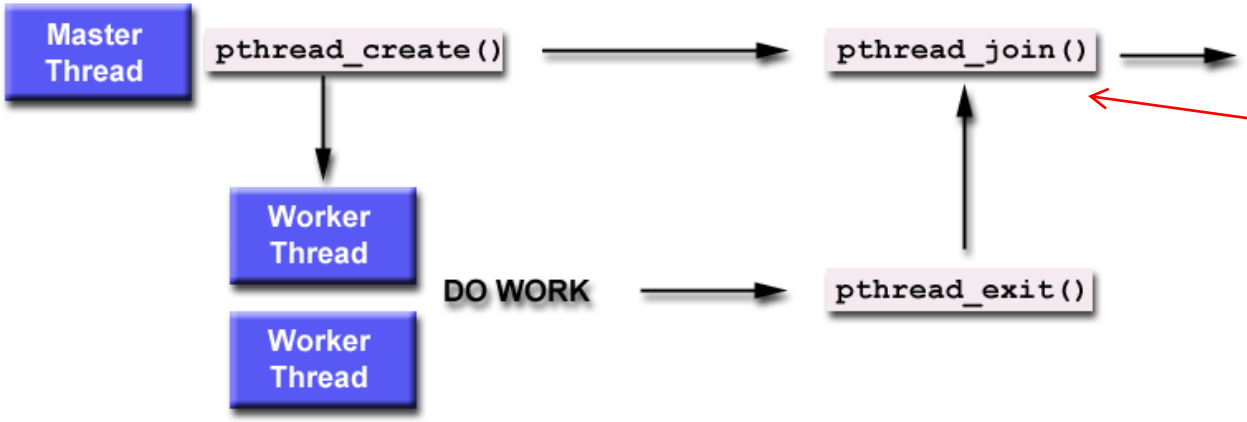
- explicitly calling pthread_exit
- letting the function return
- a call to the function exit which will terminate the process including any threads.
- canceled by another thread via the pthread_cancel routine

```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);
}
```

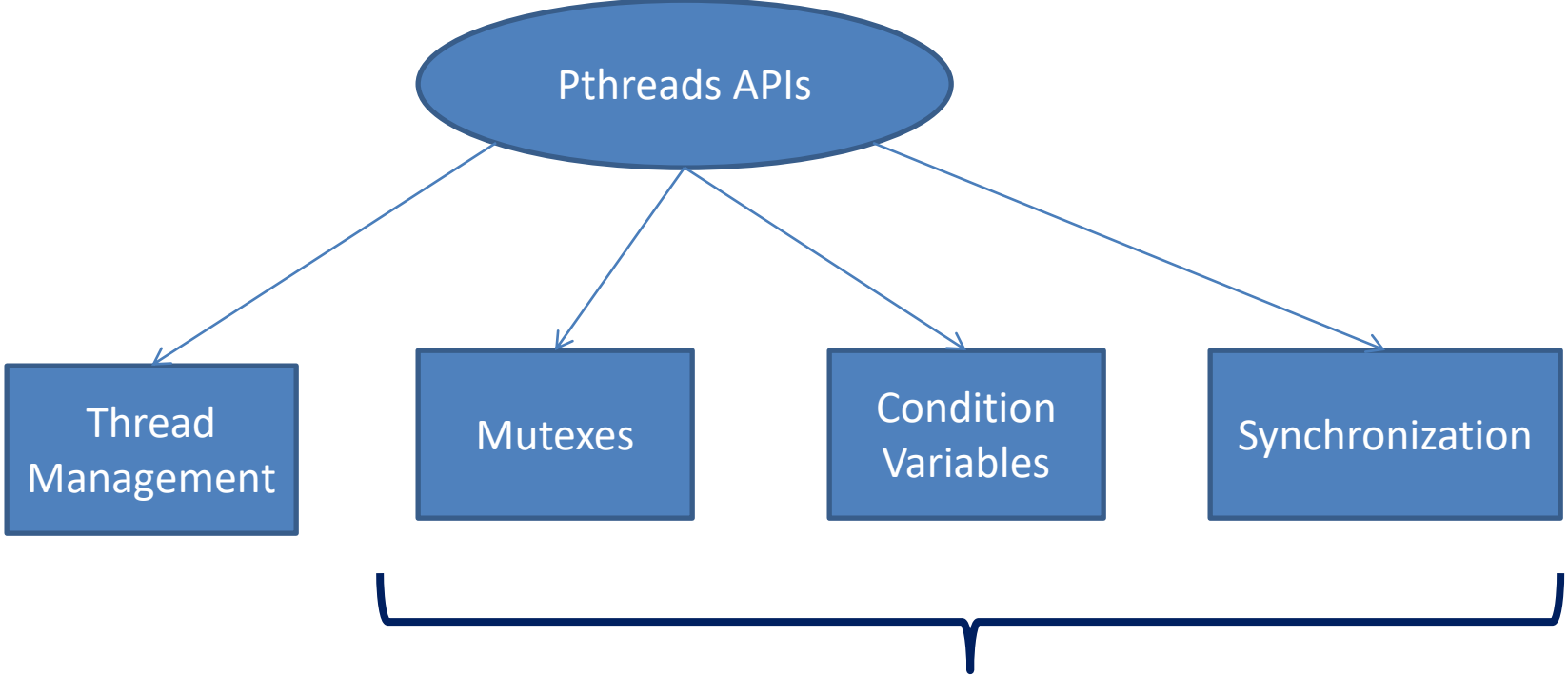
By having main() explicitly call pthread_exit() as the last thing it does, main() will block and be kept alive to support the threads it created until they are done.

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





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



We will briefly look at some of the APIs of these categories when we start discussing deadlocks, race conditions, etc.

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