Abstract

In shared memory multiprocessor architectures, threads can be used to implement parallelism. Historically, hardware vendors have implemented their own proprietary versions of threads, making portability a concern for software developers. For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard. Implementations that adhere to this standard are referred to as POSIX threads, or Pthreads.

The tutorial begins with an introduction to concepts, motivations, and design considerations for using Pthreads. Each of the three major classes of routines in the Pthreads API are then covered: Thread Management, Mutex Variables, and Condition Variables. Example codes are used throughout to demonstrate how to use most of the Pthreads routines needed by a new Pthreads programmer.

1. SOME BASICS OF THREADS

1.1 What is a Thread?

- Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system. But what does this mean?
- 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" program.

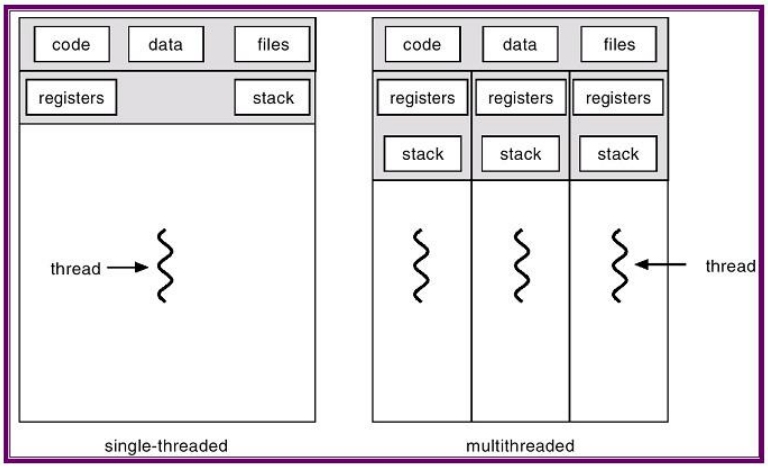
1.2 UNIX Process and Threads?

- Before understanding a thread, one first needs to understand a UNIX process. A process is created by the operating system, and requires a fair amount of "overhead". 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
    - Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

- Threads use and exist within these 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.

This independent flow of control is accomplished 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.



- 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.

2. PTHREADS OVERVIEW

2.1 What are Pthreads?

- Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.

- 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.

2.2. Why Pthreads?

- Light Weight:

    - When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.
    - For example, the following table compares timing results for the **`fork()`** subroutine and the **`pthread_create()`** subroutine. Timings reflect 50,000 process/thread creations, were performed with the `time` utility, and units are in seconds, no optimization flags.

      Note: don't expect the sytem and user times to add up to real time, because these are SMP systems with multiple CPUs/cores working on the problem at the same time. At best, these are approximations run on local machines, past and present.

- Efficient Communications/Data Exchange:

    - The primary motivation for considering the use of Pthreads in a high performance computing environment is to achieve optimum performance. In particular, if an application is using MPI for on-node communications, there is a potential that performance could be improved by using Pthreads instead.
    - MPI libraries usually implement on-node task communication via shared memory, which involves at least one memory copy operation (process to process).
    - For Pthreads there is no intermediate memory copy required because threads share the same address space within a single process. There is no data transfer, per se. It can be as efficient as simply passing a pointer.
    - In the worst case scenario, Pthread communications become more of a cache-to-CPU or memory-to-CPU bandwidth issue. These speeds are much higher than MPI shared memory communications.
    - For example: some local comparisons, past and present, are shown below:

- Other Common Reasons:

    - Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways:
        - 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.

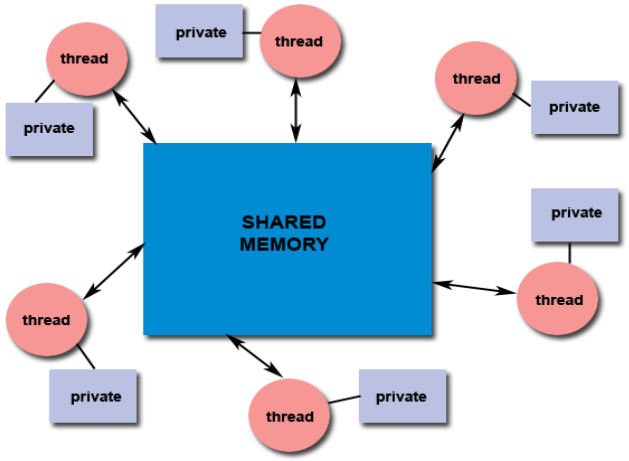
- A perfect example is the typical web browser, where many interleaved tasks can be happening at the same time, and where tasks can vary in priority.
- Another good example is a modern operating system, which makes extensive use of threads. A screenshot of the MS Windows OS and applications using threads is shown below.

2.3. Designing Threaded Programs

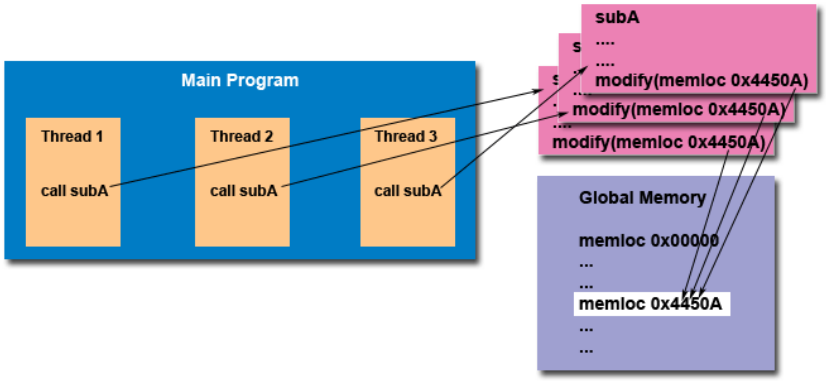
- ***Parallel Programming:***
  - On modern, multi-core machines, pthreads are ideally suited for parallel programming, and whatever applies to parallel programming in general, applies to parallel pthreads programs.
  - There are many considerations for designing parallel programs, such as:
    - What 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
    - ...
  - Covering these topics is beyond the scope of this tutorial,

  - 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)

- ***Shared Memory Model***
  - All threads have access to the same global, shared memory
  - Threads also have their own private data
  - Programmers takes care of 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.
  - For example, suppose that your 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.



  - The implication to users of external library routines is that 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.

- ***Thread Limits:***
  - Although the Pthreads API is an ANSI/IEEE standard, implementations can, and usually do, vary in ways not specified by the standard.

- 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.
- Several thread limits are discussed in more detail later in this tutorial.

2.4. Pthreads API and Programming Constructs

- The Pthreads API contains around 100 subroutines. This tutorial will focus on a subset of these - specifically, those which are most likely to be immediately useful to the beginning Pthreads programmer.

- The subroutines which comprise the Pthreads API can be informally grouped into four major groups:

    1. **Thread management:** Routines that work directly on threads - creating, detaching, joining, etc. They also include functions to set/query thread attributes (joinable, scheduling etc.)
    2. **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. These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
    3. **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. Functions to set/query condition variable attributes are also included.

2.4.1. Naming Conventions

- Naming conventions: All identifiers in the threads library begin with **pthread**_. Some examples are shown below.

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

- The concept of opaque objects pervades the design of the API. The basic calls work to create or modify opaque objects - the opaque objects can be modified by calls to attribute functions, which deal with opaque attributes.

- For portability, the `pthread.h` header file should be included in each source file using the Pthreads library.

2.4.2. Compiling Threaded Programs

- Several examples of compile commands used for pthreads codes are listed in the table below.

Compiler / Platform	Compiler Command	Description
INTEL Linux	`icc -pthread`	C
	`icpc -pthread`	C++
PGI Linux	`pgcc -lpthread`	C
	`pgCC -lpthread`	C++
GNU Linux, Blue Gene	`gcc -pthread`	GNU C
	`g++ -pthread`	GNU C++
IBM Blue Gene	`bgxlc_r / bgcc_r`	C (ANSI / non-ANSI)
	`bgxlC_r, bgxlc++_r`	C++

2.4.3. Thread Management

➢ *Routines:*

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

pthread_exit (status)

pthread_cancel (thread)

pthread_attr_init (attr)

pthread_attr_destroy (attr)
```

➢ ***Creating Threads***

- Initially, your `main()` program comprises a single, default thread. All other threads must be explicitly created by the programmer.
- `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` 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 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.
  - The maximum number of threads that may be created by a process is implementation dependent. Programs that attempt to exceed the limit can fail or produce wrong results.

- ***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. Attributes include:
    - Detached or joinable state
    - Scheduling inheritance
    - Scheduling policy
    - Scheduling parameters
    - Scheduling contention scope
    - Stack size
    - Stack address
    - Stack guard (overflow) size

- ***Terminating Threads & pthread_exit():***
  - 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
  - The `pthread_exit()` routine allows the programmer to specify an optional termination *status* parameter. This optional parameter is typically returned to threads "joining" the terminated thread (covered later).
  - In subroutines that execute to completion normally, you can often dispense with calling `pthread_exit()` - unless, of course, you want to pass the optional status code back.
  - Cleanup: the `pthread_exit()` routine does not close files; any files opened inside the thread will remain open after the thread is terminated.
  - **Discussion on calling `pthread_exit()` from main():**
    - There is a definite problem if main() finishes before the threads it spawned if you don't call `pthread_exit()` explicitly. All of the threads it created will terminate because main() is done and no longer exists to support the threads.
    - 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.

- ***Example Programs:***

A. Pthread Creation and Termination

- This code creates 5 threads with the `pthread_create()` routine. Each thread prints a "Hello World!" message, and then terminates with a call to `pthread_exit()`.

```
#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);
 }
```

B. 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, this limitation is easily overcome by creating 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 *).

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.

```
long taskids[NUM_THREADS];

for(t=0; t<NUM_THREADS; t++)
{
   taskids[t] = t;
   printf("Creating thread %ld\n", t);
   rc = pthread_create(&threads[t], NULL, PrintHello, (void *) taskids[t]);
   ...
}
```

Example 2 - 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]);
   ...
}
```

Example 3 - Thread Argument Passing (Incorrect): This example performs argument passing incorrectly. It passes the *address* of variable `t`, which is shared memory space and visible to all threads. As the loop iterates, the value of this memory location changes, possibly before the created threads can access it.

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

2.4.4. Joining and Detaching Threads

- ***Routines:***

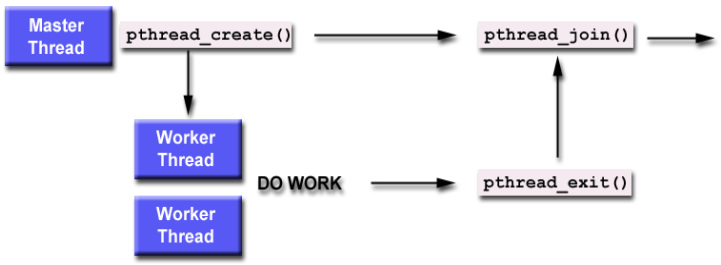
pthread_join (threadid,status)

pthread_detach (threadid)

pthread_attr_setdetachstate (attr,detachstate)

pthread_attr_getdetachstate (attr,detachstate)

- "Joining" is one way to accomplish synchronization between threads. For example:



- 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.
- Two other synchronization methods, mutexes and condition variables, will be discussed later.

➢ ***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.
- 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()`

➢ ***Detaching:***

- 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.

➢ ***Example Programs: 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.

```
#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);
 }
```

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
```

2.4.5. Miscellaneous Routines

```
pthread self ()

pthread equal (thread1,thread2)
```

- `pthread_self` returns the unique, system assigned thread ID of the calling thread.
- `pthread_equal` compares two thread IDs. If the two IDs are different 0 is returned, otherwise a non-zero value is returned.
- Note that for both of these routines, the thread identifier objects are opaque and can not be easily inspected. Because thread IDs are opaque objects, the C language equivalence operator `==` should not be used to compare two thread IDs against each other, or to compare a single thread ID against another value.

```
pthread once (once_control, init_routine)
```

- `pthread_once` executes the `init_routine` exactly once in a process. The first call to this routine by any thread in the process executes the given `init_routine`, without parameters. Any subsequent call will have no effect.
- The init_routine routine is typically an initialization routine.
- The `once_control` parameter is a synchronization control structure that requires initialization prior to calling `pthread_once`. For example:

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

2.5. MUTEX VARIABLE

2.5.1. Overview

- 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.

- A mutex variable acts like a "lock" protecting access to a shared data resource. The basic concept of a mutex as used in Pthreads is that only one thread can lock (or own) a mutex variable at any given time. Thus, even if several threads try to lock a mutex only one thread will be successful. No other thread can own that mutex until the owning thread unlocks that mutex. Threads must "take turns" accessing protected data.
- Mutexes can be used to prevent "race" conditions. An example of a race condition involving a bank transaction is shown below:

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**

- In the above example, a mutex should be used to lock the "Balance" while a thread is using this shared data resource.
- Very often the action performed by a thread owning a mutex is the updating of global variables. This is a safe way to ensure that when several threads update the same variable, the final value is the same as what it would be if only one thread performed the update. The variables being updated belong to a "critical section".
- 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
- When several threads compete for a mutex, the losers block at that call - an unblocking call is available with "trylock" instead of the "lock" call.
- When protecting shared data, it is the programmer's responsibility to make sure every thread that needs to use a mutex does so.

2.5.2. Creating and Destroying Mutexes

➢ *Routines:*

```
pthread_mutex_init (mutex,attr)

pthread_mutex_destroy (mutex)

pthread_mutexattr_init (attr)

pthread_mutexattr_destroy (attr)
```

- *Usage:*
  - Mutex variables must be declared with type `pthread_mutex_t`, and must be initialized before they can be used. There are two ways to initialize a mutex variable:
    1. Statically, when it is declared. For example:
       `pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;`
    2. Dynamically, with the `pthread_mutex_init()` routine. This method permits setting mutex object attributes, *attr*.

    The mutex is initially unlocked.

  - The *attr* object is used to establish properties for the mutex object, and must be of type `pthread_mutexattr_t` if used (may be specified as NULL to accept defaults). The Pthreads standard defines three optional mutex attributes:
    - Protocol: Specifies the protocol used to prevent priority inversions for a mutex.
    - Prioceiling: Specifies the priority ceiling of a mutex.
    - Process-shared: Specifies the process sharing of a mutex.

  - Note that not all implementations may provide the three optional mutex attributes. The pthread_mutexattr_init() and pthread_mutexattr_destroy() routines are used to create and destroy mutex attribute objects respectively. pthread_mutex_destroy() should be used to free a mutex object which is no longer needed.

2.5.3. Locking and Unlocking Mutexes

- *Routines:*

```
pthread_mutex_lock (mutex)

pthread_mutex_trylock (mutex)

pthread_mutex_unlock (mutex)
```

- *Usage:*
  - The `pthread_mutex_lock()` routine is used by a thread to acquire a lock on the specified *mutex* variable. If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked.

  - `pthread_mutex_trylock()` will attempt to lock a mutex. However, if the mutex is already locked, the routine will return immediately with a "busy" error code. This routine may be useful in preventing deadlock conditions, as in a priority-inversion situation.

  - `pthread_mutex_unlock()` will unlock a mutex if called by the owning thread. Calling this routine is required after a thread has completed its use of protected data if other threads are to acquire the mutex for their work with the protected data. Error will be returned if:
    - If the mutex was already unlocked
    - If the mutex is owned by another thread

  - There is nothing "magical" about mutexes...in fact they are akin to a "gentlemen's agreement" between participating threads. It is up to the code writer to insure that the necessary threads all make the mutex lock and unlock calls correctly. The following scenario demonstrates a logical error:

Thread 1	Thread 2	Thread 3
Lock	Lock	
A = 2	A = A+1	A = A*B
Unlock	Unlock	

- ***Example: Using Mutexes***

  - This example program illustrates the use of mutex variables in a threads program that performs a dot product.
  - The main data is made available to all threads through a globally accessible structure.
  - Each thread works on a different part of the data.
  - The main thread waits for all the threads to complete their computations, and then it prints the resulting sum.

```
#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;

 /*
 The function dotprod is activated when the thread is created.
 All input to this routine is obtained from a structure
 of type DOTDATA and all output from this function is written into
 this structure. The benefit of this approach is apparent for the
 multi-threaded program: when a thread is created we pass a single
 argument to the activated function - typically this argument
 is a thread number. All  the other information required by the
 function is accessed from the globally accessible structure.
 */

 void *dotprod(void *arg)
 {

    /* Define and use local variables for convenience */

    int i, start, end, len ;
```

```
   long offset;
   double mysum, *x, *y;
   offset = (long)arg;

   len = dotstr.veclen;
   start = offset*len;
   end   = start + len;
   x = dotstr.a;
   y = dotstr.b;

   /*
   Perform the dot product and assign result
   to the appropriate variable in the structure.
   */

   mysum = 0;
   for (i=start; i<end ; i++)
    {
      mysum += (x[i] * y[i]);
    }

   /*
   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);
}

/*
The main program creates threads which do all the work and then
print out result upon completion. Before creating the threads,
the input data is created. Since all threads update a shared structure,
we need a mutex for mutual exclusion. The main thread needs to wait for
all threads to complete, it waits for each one of the threads. We specify
a thread attribute value that allow the main thread to join with the
threads it creates. Note also that we free up handles when they are
no longer needed.
*/

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);

    /* Create threads to perform the dotproduct  */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    for(i=0; i<NUMTHRDS; i++)
    {
    /*
    Each thread works on a different set of data. The offset is specified
    by 'i'. The size of the data for each thread is indicated by VECLEN.
    */
    pthread_create(&callThd[i], &attr, dotprod, (void *)i);
    }

    pthread_attr_destroy(&attr);

    /* Wait on the other threads */
    for(i=0; i<NUMTHRDS; i++)
       {
       pthread_join(callThd[i], &status);
       }

    /* After joining, print out the results and cleanup */
    printf ("Sum =  %f \n", dotstr.sum);
    free (a);
    free (b);
    pthread_mutex_destroy(&mutexsum);
    pthread_exit(NULL);
}
```

2.6. CONDITION VARIABLES

2.6.1. Overview

- Condition variables provide yet another way for threads to synchronize. While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
- 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 can be 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.
- A condition variable is always used in conjunction with a mutex lock.
- A representative sequence for using condition variables is shown below.

Main Thread o Declare and initialize global data/variables which require synchronization (such as "count") o Declare and initialize a condition variable object o Declare and initialize an associated mutex o Create threads A and B to do work	
Thread A	Thread B

○ 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. Note that 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	○ 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
Main Thread Join / Continue	

2.6.2. Creating and Destroying Condition Variables

➢ *Routines:*

```
pthread_cond_init (condition,attr)

pthread_cond_destroy (condition)

pthread_condattr_init (attr)

pthread_condattr_destroy (attr)
```

➢ *Usage:*

- Condition variables must be declared with type `pthread_cond_t`, and must be initialized before they can be used. There are two ways to initialize a condition variable:
  1. Statically, when it is declared. For example:
     `pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;`
  2. Dynamically, with the `pthread_cond_init()` routine. The ID of the created condition variable is returned to the calling thread through the *condition* parameter. This method permits setting condition variable object attributes, *attr*.
- The optional *attr* object is used to set condition variable attributes. There is only one attribute defined for condition variables: process-shared, which allows the condition variable to be seen by threads in other processes. The attribute object, if used, must be of type `pthread_condattr_t` (may be specified as NULL to accept defaults).

  Note that not all implementations may provide the process-shared attribute.

- The `pthread_condattr_init()` and `pthread_condattr_destroy()` routines are used to create and destroy condition variable attribute objects.
- `pthread_cond_destroy()` should be used to free a condition variable that is no longer needed.

2.6.3. Waiting and Signaling on Condition Variables

➢ *Routines:*

```
pthread_cond_wait (condition,mutex)

pthread_cond_signal (condition)

pthread_cond_broadcast (condition)
```

➢ *Usage:*

- `pthread_cond_wait()` blocks the calling thread until the specified *condition* is signalled. This routine should be called while *mutex* is locked, and it will automatically release the mutex while it waits. After signal is received and thread is awakened, *mutex* will be automatically locked for use by the thread. The programmer is then responsible for unlocking *mutex* when the thread is finished with it.

  Recommendation: Using a WHILE loop instead of an IF statement (see watch_count routine in example below) to check the waited for condition can help deal with several potential problems, such as:

  - If several threads are waiting for the same wake up signal, they will take turns acquiring the mutex, and any one of them can then modify the condition they all waited for.
  - If the thread received the signal in error due to a program bug
  - The Pthreads library is permitted to issue spurious wake ups to a waiting thread without violating the standard.
- The `pthread_cond_signal()` routine is used to signal (or wake up) another thread which is waiting on the condition variable. It should be called after *mutex* is locked, and must unlock *mutex* in order for `pthread_cond_wait()` routine to complete.
- The `pthread_cond_broadcast()` routine should be used instead of `pthread_cond_signal()` if more than one thread is in a blocking wait state.
- It is a logical error to call `pthread_cond_signal()` before calling `pthread_cond_wait()`.

➢ ***Example: Using Condition Variables***

- This simple example code demonstrates the use of several Pthread condition variable routines.
- The main routine creates three threads.
- Two of the threads perform work and update a "count" variable.
- The third thread waits until the count variable reaches a specified value.

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

 #define NUM_THREADS  3
 #define TCOUNT 10
 #define COUNT_LIMIT 12
```

```
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++;

    /*
    Check the value of count and signal waiting thread when condition is
    reached.  Note that this occurs while mutex is locked.
    */
    if (count == COUNT_LIMIT) {
      pthread_cond_signal(&count_threshold_cv);
      printf("inc_count(): thread %ld, count = %d  Threshold reached.\n",
             my_id, count);
      }
    printf("inc_count(): thread %ld, count = %d, unlocking mutex\n",
           my_id, count);
    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;

  printf("Starting watch_count(): thread %ld\n", my_id);

  /*
  Lock mutex and wait for signal.  Note that the pthread_cond_wait
  routine will automatically and atomically unlock mutex while it waits.
  Also, note that if COUNT_LIMIT is reached before this routine is run by
  the waiting thread, the loop will be skipped to prevent pthread_cond_wait
  from never returning.
  */
  pthread_mutex_lock(&count_mutex);
  while (count<COUNT_LIMIT) {
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %ld Condition signal received.\n", my_id);
    count += 125;
    printf("watch_count(): thread %ld count now = %d.\n", my_id, count);
    }
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  int i, rc;
  long t1=1, t2=2, t3=3;
  pthread_t threads[3];
```

```
  pthread_attr_t attr;

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);

  /* For portability, explicitly create threads in a joinable state */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, watch_count, (void *)t1);
  pthread_create(&threads[1], &attr, inc_count, (void *)t2);
  pthread_create(&threads[2], &attr, inc_count, (void *)t3);

  /* Wait for all threads to complete */
  for (i=0; i<NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited on %d  threads. Done.\n", NUM_THREADS);

  /* Clean up and exit */
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit(NULL);

}
```

Output

```
Starting watch_count(): thread 1
inc_count(): thread 2, count = 1, unlocking mutex
inc_count(): thread 3, count = 2, unlocking mutex
watch_count(): thread 1 going into wait...
inc_count(): thread 3, count = 3, unlocking mutex
inc_count(): thread 2, count = 4, unlocking mutex
inc_count(): thread 3, count = 5, unlocking mutex
inc_count(): thread 2, count = 6, unlocking mutex
inc_count(): thread 3, count = 7, unlocking mutex
inc_count(): thread 2, count = 8, unlocking mutex
inc_count(): thread 3, count = 9, unlocking mutex
inc_count(): thread 2, count = 10, unlocking mutex
inc_count(): thread 3, count = 11, unlocking mutex
inc_count(): thread 2, count = 12  Threshold reached. Just sent signal.
inc_count(): thread 2, count = 12, unlocking mutex
watch_count(): thread 1 Condition signal received.
watch_count(): thread 1 count now = 137.
inc_count(): thread 3, count = 138, unlocking mutex
inc_count(): thread 2, count = 139, unlocking mutex
inc_count(): thread 3, count = 140, unlocking mutex
inc_count(): thread 2, count = 141, unlocking mutex
inc_count(): thread 3, count = 142, unlocking mutex
inc_count(): thread 2, count = 143, unlocking mutex
inc_count(): thread 3, count = 144, unlocking mutex
inc_count(): thread 2, count = 145, unlocking mutex
Main(): Waited on 3 threads. Final value of count = 145. Done.
```