POSIX `pthreads` and Concurrency Management

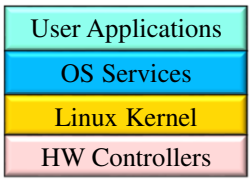
Raj Rajkumar

Lecture #6

Outline

- The IEEE POSIX Standard for threads: POSIX
- POSIX Threads and Attributes
- POSIX Mutexes and Attributes
- POSIX Condition Variables and Attributes

Major Linux Subsystems



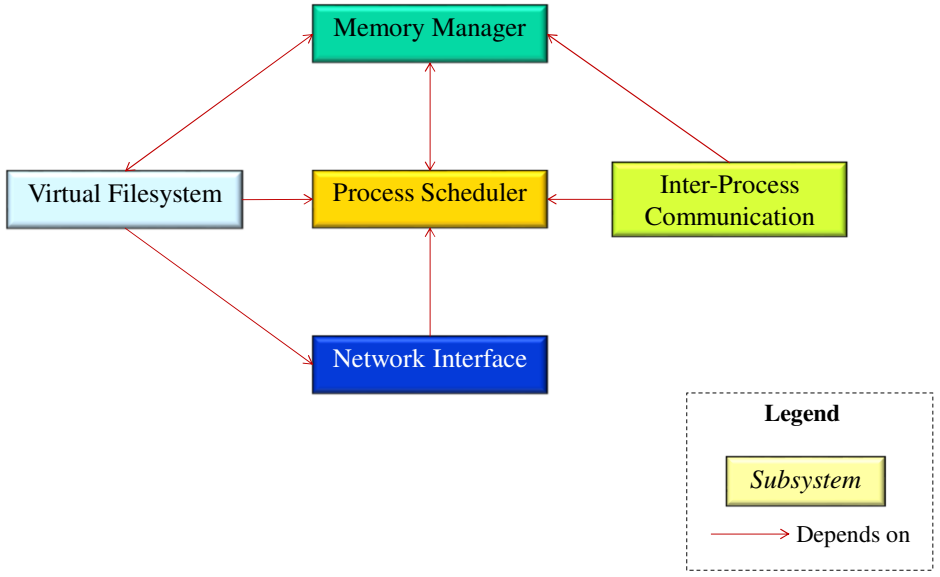
- **User Applications** -- the set of applications in use on a particular Linux system
- **O/S Services** -- services that are typically considered part of the operating system (a windowing system, command shell, etc.);
- also, the programming interface to the kernel (compiler tool and library) is included in this subsystem.
- **Linux Kernel** -- the kernel abstracts and mediates access to the hardware resources, including the CPU.
- **Hardware Controllers** -- this subsystem is comprised of the possible physical devices in a Linux installation; for example, the CPU, memory hardware, hard disks, and network hardware are all members of this subsystem

Linux Kernel Subsystems

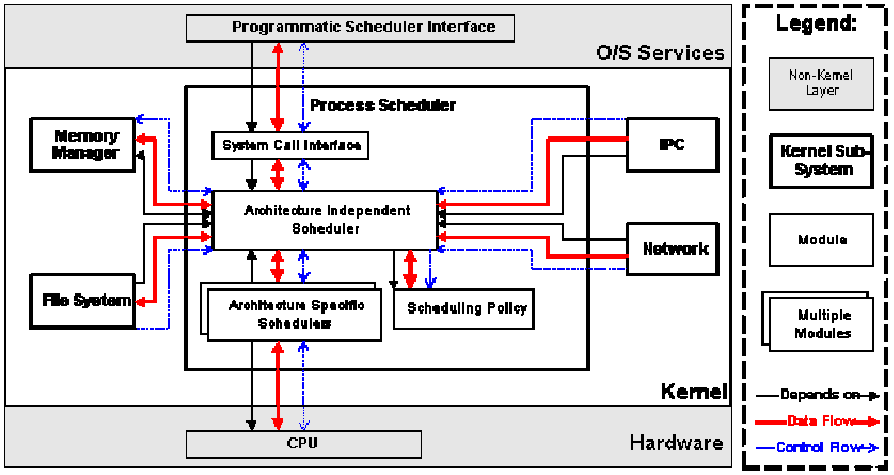
The Linux kernel is composed of five main subsystems:

- The **Process Scheduler** (SCHED) is responsible for controlling process access to the CPU. The scheduler enforces a policy that ensures that processes will have fair access to the CPU, while ensuring that necessary hardware actions are performed by the kernel on time.
- The **Memory Manager** (MM) permits multiple process to securely share the machine's main memory system. In addition, the memory manager supports virtual memory that allows Linux to support processes that use more memory than is available in the system. Unused memory is swapped out to persistent storage using the file system then swapped back in when it is needed.
- The **Virtual File System** (VFS) abstracts the details of the variety of hardware devices by presenting a common file interface to all devices. In addition, the VFS supports several file system formats that are compatible with other operating systems.
- The **Network Interface** (NET) provides access to several networking standards and a variety of network hardware.
- The **Inter-Process Communication** (IPC) subsystem supports several mechanisms for process-to-process communication on a single Linux system.

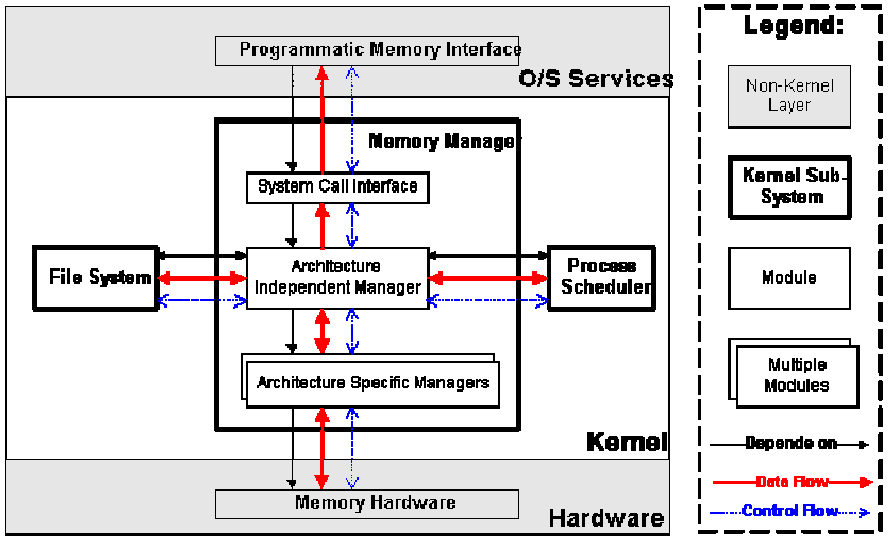
Linux Kernel Subsystems



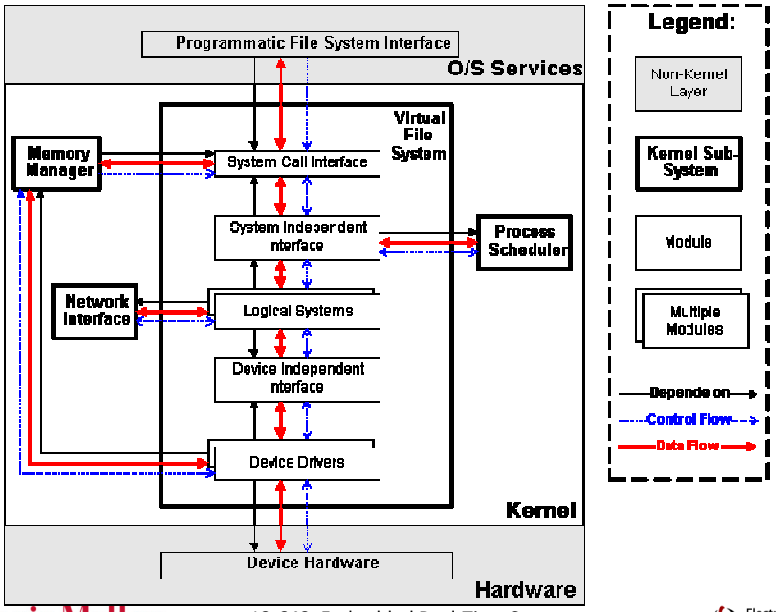
Process Scheduler Subsytem



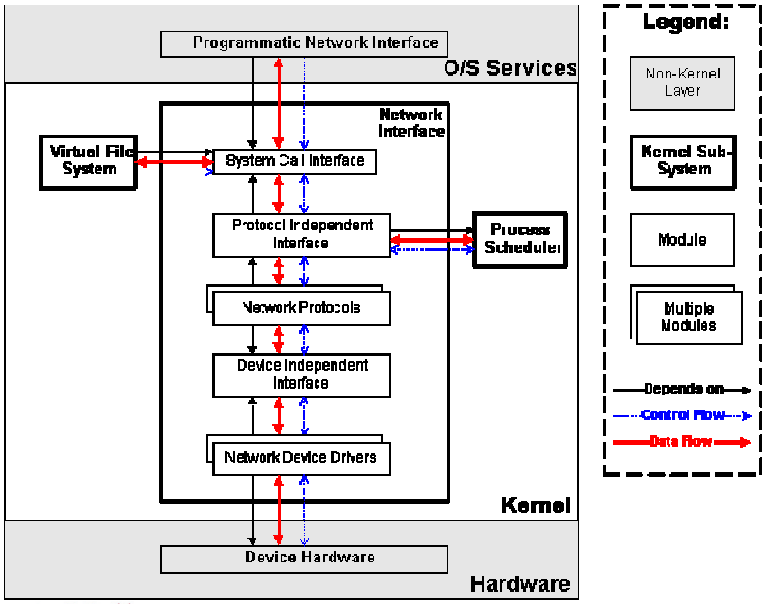
Memory Manager Subsystem



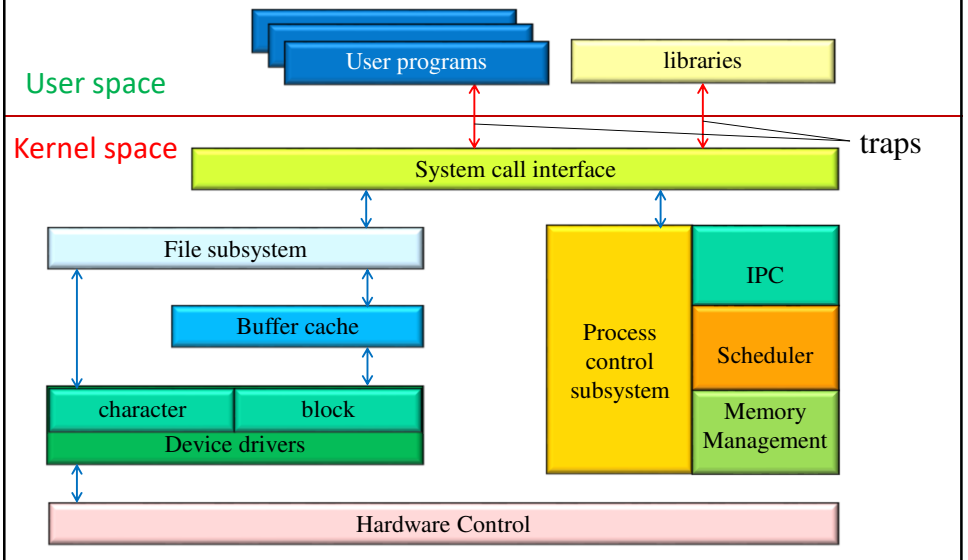
Virtual File System Architecture



Network Interface Architecture



A Different View of the Kernel



Why `pthreads`?

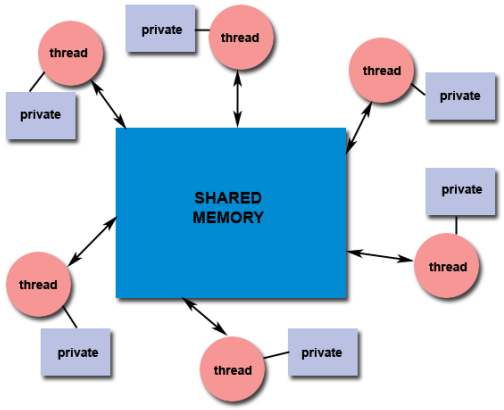
- The primary motivation for using `pthreads` is to realize potential program performance gains.
- When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.
  - A `fork()` is much costlier than a `pthread_create()`
- 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)
  - `pthreads` can also be used for serial applications, to emulate parallel execution. A perfect example is the typical web browser, which for most people, runs on a single cpu desktop/laptop machine. Many things can "appear" to be happening at the same time.

Common Models for Threaded Programs

- ***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 and dynamic worker pool.
- ***Pipeline***
  - A task is broken into a series of suboperations, each of which is handled in series, but concurrently, by a different thread.
  - An automobile assembly line best describes this model.
- ***Peers***
  - Similar to the manager/worker model, but after the main thread creates other threads, it participates in the work.

The Shared Memory Model of Threads

- All threads in a process 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.

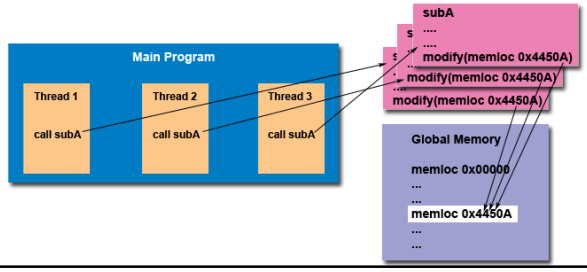


Thread Safety (1 of 2)

- Thread safety refers to 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 location in memory.
    - As each thread calls this routine, it is possible that many threads 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.

Thread Safety (2 of 2)

- If you are *not* 100% certain functions in a library which you use are thread-safe, then you take your chances with problems that could arise
  - at the most inopportune time – *Murphy*
    - Be careful if your application uses libraries or other objects that do not explicitly guarantee thread-safety.
    - If in doubt, assume that they are not thread-safe until proven otherwise.
    - "Serialize" calls to the potentially "unsafe" routine, etc.



Compiler/Linker Command in Linux

```
prompt> gcc -pthread
```

POSIX API Categories

- ***Thread management functions:***
  - work directly on threads - creating, detaching, joining, etc.
  - include functions to set/query thread attributes (joinable, scheduling etc.)
- ***Mutexes:***
  - deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion".
  - functions provide for creating, destroying, locking and unlocking mutexes.
  - supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
- ***Condition variables:***
  - address communications between threads that share a mutex.
  - based upon programmer-specified conditions.
  - includes functions to create, destroy, wait and signal based upon specified variable values.
  - functions to set/query condition variable attributes are also included.

POSIX Threads (pthreads)

What are `pthreads`?

- A standardized programming interface for programming threads.
- Historically, hardware vendors implemented their own proprietary versions of threads.
  - These implementations differed substantially from each other.
- For UNIX systems, the POSIX standard interface has been specified by the IEEE POSIX 1003.1c standard
  - Implementations which adhere to this standard are referred to as POSIX threads, or pthreads.
- pthreads are defined as a set of C language programming types and procedure calls, implemented with a `pthread.h` header/include file and a thread library - though this library may be part of another library, such as `libc`.

POSIX

- ***Thread management:***
  - This class of functions work directly on threads - creating, detaching, joining, etc.
  - Include functions to set/query thread attributes (joinable, scheduling etc.)
- ***Mutexes:***
  - This class of functions deals with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion".
  - Mutex functions provide for creating, destroying, locking and unlocking mutexes.
  - Supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
- ***Condition variables:***
  - This class of functions address communications between threads that share a mutex.
  - Based upon programmer-specified conditions.
  - Includes functions to create, destroy, wait and signal based upon specified variable values.
  - Functions to set/query condition variable attributes are also included.

POSIX

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

POSIX Thread Management

Creating and Terminating Threads

Routines:

- `pthread_create (thread, attr, start_routine, arg)`
- `pthread_exit (status)`
- `pthread_attr_init (attr)`
- `pthread_attr_destroy (attr)`

Example Code - Pthread Creation & Termination

```
#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);
           }
   }
   pthread_exit(NULL);
}
```

Thread Argument Passing (Example 1)

```
long *taskids[NUM_THREADS];

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


Thread Argument Passing (Example 2)

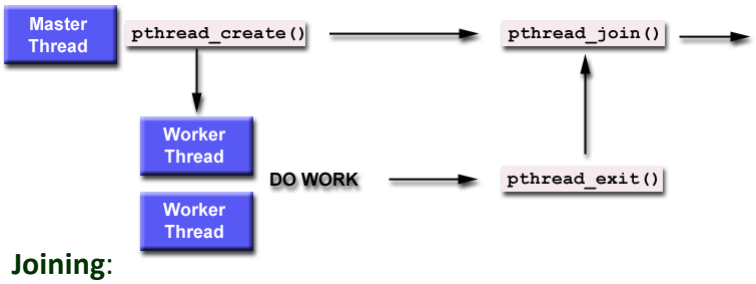
This example shows how to setup/pass multiple arguments via a structure.

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

Joining and Detaching Threads

- Routines:

```
pthread_join(threadid, status)
pthread_detach(threadid, status)
pthread_attr_setdetachstate(attr, detachstate)
pthread_attr_getdetachstate(attr, detachstate)
```



- **Joining**:
  - One way to accomplish synchronization between threads.
  - The `pthread_join()` subroutine blocks the calling thread until the specified `threadid` thread terminates.

pthread Joining

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.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/set thread detached attr */
   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 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);
}
```

Stack Management

- Routines:

```
pthread_attr_getstacksize (attr, stacksize)
pthread_attr_setstacksize (attr, stacksize)
pthread_attr_getstackaddr (attr, stackaddr)
pthread_attr_setstackaddr (attr, stackaddr)
```

- Preventing Stack Problems:
  - The POSIX standard does *not* dictate the size of a thread's stack.
  - This is implementation-dependent and varies.

Stack Management Example

```
#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);
}
```

Miscellaneous Routines

- `pthread_self()`
  - returns the unique, system assigned thread ID of the calling thread.
- `pthread_equal(thread1, thread2)`
  - compares two thread IDs. If the two IDs are different 0 is returned, otherwise a non-zero value is returned.
  - do not use == for comparing thread ids!
- `pthread_once(once_control, init_routine)`
  - 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.

Mutexes

- **Mutex** is an abbreviation for "mutual exclusion".
  - A mutex variable is one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur.
- A mutex serves as a "**lock**" protecting access to shared data or resource.
- Only one thread can lock (or own) a mutex variable at any given time. Even if several threads try to lock a mutex, only one thread succeeds. No other thread can own the mutex until the owning thread unlocks that mutex.
  - Threads must "take turns" accessing protected data.

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 variable `Balance` while a thread is using this shared data resource.
- The shared variable(s) being updated belong to a "**critical section**".

Creating and Destroying Mutexes

Routines:

- `pthread_mutex_init(mutex, attr)`
  - *`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*: Not all implementations may provide the three optional mutex attributes.
- `pthread_mutex_destroy(mutex)`
  - Used to free a mutex object which is no longer needed.
- `pthread_mutexattr_init(attr)`
  - Used to create mutex attribute objects
- `pthread_mutexattr_destroy(attr)`
  - Used to destroy mutex attribute objects

Locking and Unlocking Mutexes

- `pthread_mutex_lock(mutex)`
  - 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(mutex)`
  - will attempt to lock a mutex. However, if the mutex is already locked, the routine will return immediately with a "busy" error code.
- `pthread_mutex_unlock(mutex)`
  - 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.

Typical Mutex Usage 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.

Example

```
#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. */
```

```
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);
}
```

Example: Using Mutexes

```
/* 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);
}
```


Condition Variables

- Condition variables provide another way for threads to synchronize.
  - Mutexes implement synchronization by controlling thread access to data
  - Condition variables allow threads to synchronize based upon the actual value of data.

Main Thread
- Declare and initialize global data/variables which require synchronization (such as "count")
- Declare and initialize a condition variable object
- 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. 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

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

Main Thread
Join / Continue

Creating and Destroying Condition Variables

`pthread_cond_init (condition,attr)`

- Condition variables must be declared with type `pthread_cond_t`, and must be initialized before use. 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. The ID of the created condition variable is returned to the calling thread through the *condition* parameter.
  - 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.

`pthread_cond_destroy (condition)`

- used to free a condition variable that is no longer needed.

`pthread_condattr_init (attr)`

- used to create condition variable attribute objects.

`pthread_condattr_destroy (attr)`

- used to destroy condition variable attribute objects.

- Note that *not* all implementations may provide the process-shared attribute.
- The `pthread_condattr_init()` and `pthread_condattr_destroy()` routines are for init and destruction.

Waiting and Signaling on Condition Variables

- **`pthread_cond_wait(condition, mutex)`**
  - 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.
- **`pthread_cond_signal (condition)`**
  - 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.
- **`pthread_cond_broadcast (condition)`**
  - Used instead of `pthread_cond_signal()` if more than one thread is in a blocking wait state.

It can be a logical error to call `pthread_cond_signal()` before calling `pthread_cond_wait()`.

- Proper locking and unlocking of the associated mutex variable is essential when using these routines.
  - Failing to lock the mutex before calling `pthread_cond_wait()` may cause it NOT to block.
  - Failing to unlock the mutex after calling `pthread_cond_signal()` may not allow a matching `pthread_cond_wait()` routine to complete (and it will remain blocked).

Example: Using Condition Variables (1 of 2)

```
#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);
}
```


Example: Using Condition Variables (2 of 2)

```
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);
  if (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 & 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 Library Routines (1 of 6)

Pthread Functions	
Thread Management	`pthread_create pthread_exit pthread_join pthread_once pthread_kill pthread_self pthread_equal pthread_yield pthread_detach`
Thread-Specific Data	`pthread_key_create, pthread_key_delete, pthread_getspecific, pthread_setspecific`
Thread Cancellation	`pthread_cancel, pthread_cleanup_pop, pthread_cleanup_push, pthread_setcancelstate, pthread_getcancelstate, pthread_testcancel`
Thread Scheduling*	**`pthread_getschedparam, pthread_setschedparam`**
Signals	`pthread_sigmask`

*SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC

Pthread Library Routines (2 of 6)

Pthread Attribute Functions	
Basic Management	`pthread_attr_init, pthread_attr_destroy`
Detachable or Joinable	`pthread_attr_setdetachstate, pthread_attr_getdetachstate`
Specifying Stack Information	`pthread_attr_getstackaddr, pthread_attr_getstacksize, pthread_attr_setstackaddr, pthread_attr_setstacksize`
Thread Scheduling Attributes	`pthread_attr_getschedparam, pthread_attr_setschedparam, pthread_attr_getschedpolicy, pthread_attr_setschedpolicy, pthread_attr_setinheritsched, pthread_attr_getinheritsched, pthread_attr_setscope, pthread_attr_getscope`

Pthread Library Routines (3 of 6)

Mutex Functions	
Mutex Management	`pthread_mutex_init, pthread_mutex_destroy, pthread_mutex_lock, pthread_mutex_unlock, pthread_mutex_trylock`
Priority Management	`pthread_mutex_setprioceiling pthread_mutex_getprioceiling`

Pthread Library Routines (4 of 6)

Mutex Attribute Functions	
Basic Management	`pthread_mutexattr_init pthread_mutexattr_destroy`
Sharing	`pthread_mutexattr_getpshared, pthread_mutexattr_setpshared`
Protocol Attributes*	`pthread_mutexattr_getprotocol pthread_mutexattr_setprotocol`
Priority Management	`pthread_mutexattr_setprioceiling pthread_mutexattr_getprioceiling`

`*PTHREAD_PRIO_NONE, PTHREAD_PRIO_INHERIT, PTHREAD_PRIO_PROTECT`

Pthread Library Routines (5 of 6)

Condition Variable Functions	
Basic Management	`pthread_cond_init`, `pthread_cond_destroy`, `pthread_cond_signal`, `pthread_cond_broadcast`, `pthread_cond_wait`, `pthread_cond_timedwait`,

Pthread Library Routines (6 of 6)

Condition Variable Attribute Functions	
Basic Management	`pthread_condattr_init`, `pthread_condattr_destroy`
Sharing	`pthread_condattr_getpshared`, `pthread_condattr_setpshared`

POSIX Real-Time Extensions

- POSIX 1003.1b, as well as 1003.1d and 1003.1j, define extensions useful for development of real-time systems.
- Bulk of the features defined in POSIX 1003.1b:
  - **Timers**: periodic timers, delivery is accomplished using POSIX signals
  - **Priority scheduling**: fixed priority preemptive scheduling with a minimum of 32 priority levels
  - **Real-time signals**: additional signals with multiple levels of priority
  - **Semaphores**: named and memory counting semaphores
  - **Memory queues**: message passing using named queues
  - **Shared memory**: named memory regions shared between multiple processes
  - **Memory locking**: functions to prevent virtual memory swapping of physical memory pages

POSIX Thread Extensions

- **Thread control**: creation, deletion, and management of individual threads
- **Priority scheduling**:
  - POSIX real-time scheduling extended to include scheduling on a per-thread basis;
  - the scheduling scope is either done globally across all threads in all processes, or performed locally within each process
- **Mutexes**:
  - used to guard critical sections of code;
  - mutexes also include support for priority inheritance and priority ceiling protocols to help prevent priority inversions
- **Condition variables**: used in conjunction with mutexes, condition variables can be used to create a monitor synchronization structure
- **Signals**: ability to deliver signals to individual threads

Summary

- POSIX is a standard set of interfaces for operating systems, standardized by the IEEE (IEEE POSIX 1003.1)
- `pthreads` is the set of POSIX interfaces for threads (IEEE 1003.1c)
- Supports interfaces for
  - Threads management, thread attributes
  - Mutex management, mutex attributes
  - Condition variable management, condition variable attributes
- Real-time scheduling policies to support RMS/RMA are included
- Real-time synchronization policies to support basic priority inheritance and priority ceiling protocols are included