Scheduling of threads and Synchronization in linux

Rakesh Basavaraju

Agenda

- ✓Thread Scheduling
- ✓Synchronization Overview
- ✓ Mutex and join
- ✓ Condition variables
- ✓ Read/write locks
- ✓ Spinlocks
- ✓ Barriers
- ✓ Semaphore

Thread Scheduling Overview

Linux Scheduler

- The scheduler is the kernel component that decides which runnable thread will be executed by the CPU next
- Policy is the behavior of the scheduler that determines what runs when. A scheduler's policy often determines the overall feel of a system and is responsible for optimally utilizing processor time.
- A common type of scheduling algorithm is *priority-based* scheduling. The idea is to rank processes based on their worth and need for processor time. Processes with a higher priority run before those with a lower priority, whereas processes with the same priority are scheduled round-robin (one after the next, repeating). On some systems, Linux included, processes with a higher priority also receive a longer timeslice. The runnable process with timeslice remaining and the highest priority always runs.
- Both the user and the system may set a process's priority to influence the scheduling behavior of the system
- Linux provides ***dynamic priority-based*** scheduling. This concept **begins with an initial base priority** and then **enables the scheduler to increase or decrease the priority dynamically to fulfill scheduling objectives**. For example, a process that is spending more time waiting on I/O than running is clearly **I/O bound** hence it receives an elevated dynamic priority. As a counterexample, a process that continually uses up its entire timeslice is **processor bound**, it would receive a **lowered dynamic priority**
- The Linux kernel implements two separate priority ranges
  - The first is the *nice* value, a number from -20 to +19 with a default of 0. Larger nice values correspond to a lower priority. Processes with a lower nice value (higher priority) run before processes with a higher nice value (lower priority). The nice value also helps determine how long a timeslice the process receives. A process with a nice value of -20 receives the maximum possible timeslice, whereas a process with a nice value of 19 receives the minimum possible timeslice. Nice values are the standard priority range used in all Unix systems
  - The second range is the real-time priority. The values are configurable, but by default range from 0 to 99. All real-time processes are at a higher priority than normal processes. Linux implements real-time priorities in accordance with POSIX standards on the matter. Most modern Unix systems implement a similar scheme.

Linux Scheduler – An Example

The Scheduling Policy in Action

- Consider a system with two runnable tasks: a text editor and a video encoder. **The text editor is I/O-bound because it spends nearly all its time waiting for user key presses** (no matter how fast the user types, it is not *that* fast). Despite this, when the text editor does receive a key press, **the user expects the editor to respond *immediately***. Conversely, the **video encoder is processor-bound**. Aside from reading the raw data stream from the disk and later writing the resulting video, the **encoder spends all its time applying the video codec to the raw data**, easily using 100% of the processor. The **video encoder does not have any strong time constraints on when it runs if it started running now or in half a second**, the user could not tell and would not care. Of course, the sooner it finishes the better, but latency is not a primary concern.
- In this scenario example, ideally **the scheduler gives the text editor a higher priority** and **larger timeslice than the video encoder** receives **because the text editor is interactive**. This ensures that the text editor has plenty of timeslice available. Furthermore, because the text editor has a higher priority, it is capable of preempting the video encoder when needed, say, the instant the user presses a key. This guarantees that the text editor is capable of responding to user key presses immediately. This is to the detriment of the video encoder, but because the text editor runs only intermittently, when the user presses a key, the video encoder can monopolize the remaining time. This optimizes the performance of both applications.

Scheduling General

- The basic data structure in the scheduler is the **runqueue**. The **runqueue is the list of runnable processes on a given processor**; there is **one runqueue per processor**. **Each runnable process is on exactly one runqueue**.
- Each thread has an associated scheduling policy and a static scheduling priority, sched_priority. The scheduler makes its decisions based on knowledge of the scheduling policy and static priority of all threads on the system
- For threads scheduled under one of the normal scheduling policies (SCHED_OTHER, SCHED_IDLE, SCHED_BATCH), **sched_priority is not used in scheduling decisions** (it must be specified as 0)
- Processes scheduled under one of the real-time policies (SCHED_FIFO, SCHED_RR) have a sched_priority value in the range 1 (low) to 99 (high). (As the numbers imply, real-time threads always have higher priority than normal threads.)
- Conceptually, the scheduler maintains a list of runnable threads for each possible sched_priority value. In order to determine which thread runs next, the scheduler looks for the nonempty list with the highest static priority and selects the thread at the head of this list
- A thread's scheduling policy determines where it will be inserted into the list of threads with equal static priority and how it will move inside this list
- All scheduling is preemptive: if a thread with a higher static priority becomes ready to run, the currently running thread will be preempted and returned to the wait list for its static priority level. The scheduling policy determines the ordering only within the list of runnable threads with equal static priority

Scheduling General

Contention Scope

- There are two possible contention scopes. PTHREAD_SCOPE_SYSTEM and PTHREAD_SCOPE_PROCESS. They can be set with pthread_attr_setscope(). The scope of a thread can only be specified before the thread is created.

PTHREAD_SCOPE_SYSTEM

- A thread that has a scope of PTHREAD_SCOPE_SYSTEM will content with other processes and other PTHREAD_SCOPE_SYSTEM threads for the CPU. That is if there is one process P1 with 10 threads with scope PTHREAD_SCOPE_SYSTEM and a single threaded process P2, P2 will get one time slice out of 11 and every thread in P1 will get one time slice out of 11. I.e. P1 will get 10 time more time slices than P2.

PTHREAD_SCOPE_PROCESS

- All threads of a process that have a scope of PTHREAD_SCOPE_PROCESS **will be grouped together and this group of threads contents for the CPU**. If there is a process with 4 PTHREAD_SCOPE_PROCESS threads and 4 PTHREAD_SCOPE_SYSTEM threads, then each of the PTHREAD_SCOPE_SYSTEM threads will get a fifth of the CPU and the **other 4 PTHREAD_SCOPE_PROCESS threads will share the remaining fifth of the CPU i.e. the these 4 threads will compete in this one fifth of CPU time that is less CPU compared with other system scope threads**. How the PTHREAD_SCOPE_PROCESS threads share their fifth of the CPU among themselves is determined by the scheduling policy and the thread's priority.
- If there are other processes running, then every PTHREAD_SCOPE_SYSTEM and every group of PTHREAD_SCOPE_PROCESS threads (i.e. every process with PTHREAD_SCOPE_PROCESS threads) will be handled like a separate process by the system scheduler.

Scheduling Policy

Priorities and Scheduling Policy

- A PTHREAD_SCOPE_PROCESS thread has a priority. Whenever a thread is runnable and no other thread (of this process) has a higher priority the thread will get the CPU. Note that this might lead to starvation of other threads
- When two or more runnable threads have the same priority and no other runnable thread has a higher priority, then the scheduling policy will determine which of these highest priority threads to run
- The priority is assigned statically with pthread_setschedparam(). The scheduler will not change the priority of a thread
- The scheduling policy can either be SCHED_FIFO or SCHED_RR. FIFO is a first come first serve policy. RR is a round robin policy that might preempt threads. But again, the policy only effects threads that have the same priority
- A more extensive description of priorities and policies can be found in [1] and [2]. Note that these documents discuss process scheduling, but the principle is the same.
- **Note:** The priority and scheduling policy settings are meaningless when a thread has scope PTHREAD_SCOPE_SYSTEM.

Nice values

- The nice value of a process also influences the scheduling behavior. A process (and the threads therein) with a lower nice value (i.e. higher priority) will get a higher share of the CPU time. Starting a program with nice works as expected

Synchronization Overview

Critical Section

Creating threads is easy. Hard part is to share the data amongst them. Basic problems is threads use the same memory space.

Critical Section

Code paths that access and manipulate shared data are called critical regions (critical sections).

If it is possible for two threads of execution to be simultaneously executing within the same critical region. When this does occur, we call it a *race condition*, so-named because the threads *raced* to get there first.

Consider a Single shared variable in critical region, int var; and var is getting incremented.

var++

Steps involved in incrementing the var

1. Get the current value of var and copy it in register – get var
2. Add one to the value stored in register – increment var
3. Write back to memory the new value of var – write var

Now assume that there are two threads of execution and both enter this critical region.

Expectation and Reality

Expectation

Thread 1	Thread 2
get var(7)	--
increment var(7 -> 8)	--
Write back var(8)	--
--	get var(8)
--	increment var(8 -> 9)
--	Write back var(9)

Reality

Thread 1	Thread 2
get var(7)	get var(7)
increment var(7 -> 8)	--
--	increment var(7 -> 8)
write back var(8)	--
	write back var(8)
--	
--	

Thread Synchronization

Ensuring that threads access shared data in an orderly and controlled way is called synchronization.

Synchronization methods

- ✓ Mutex
- ✓ Join
- ✓ Condition variables
- ✓ Reader/writer Locks
- ✓ Semaphores

Mutex and join

Mutex

- Mutual exclusion lock: Block access to variables by other threads. This enforces exclusive access by a thread to a variable or set of variables.

A mutex is a lock that guarantees three things:

- Atomicity - Locking a mutex is an atomic operation, meaning that the operating system (or threads library) assures you that if you locked a mutex, no other thread succeeded in locking this mutex at the same time.

- Singularity - If a thread managed to lock a mutex, it is assured that no other thread will be able to lock the thread until the original thread releases the lock.

- Non-Busy Wait - If a thread attempts to lock a thread that was locked by a second thread, the first thread will be suspended (and will not consume any CPU resources) until the lock is freed by the second thread.

Pthread mutex

Create and initialize a mutex:

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);

pthread_mutex_t variable required to operate on as the first argument.
Attributes for the mutex can be given through the second parameter. To specify default attributes, pass NULL as the second parameter.

Alternatively, mutexes can be initialized to default values through a convenient macro rather than a function call:

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread mutex

API	Description
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);	initialises the mutex referenced by mutex with attributes specified by attr. If attr is NULL, the default mutex attributes are used;
int pthread_mutex_destroy (pthread_mutex_t *mutex);	destroys the mutex object referenced by mutex; the mutex object becomes, in effect, uninitialised.
int pthread_mutex_lock(pthread_mutex_t *mutex);	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.
int pthread_mutex_unlock (pthread_mutex_t *mutex);	unlock a mutex variable. An error is returned if mutex is already unlocked or owned by another thread.
int pthread_mutex_trylock (pthread_mutex_t *mutex);	attempt to lock a mutex or will return error code if busy. Useful for preventing deadlock conditions.

Mutex example

```
#include <pthread.h>

void *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main()
{
   int rc1, rc2;
   pthread_t thread1, thread2;

   if( (rc1=pthread_create( &thread1, NULL, &functionC, NULL)) )  {
      printf("Thread creation failed: %d\n", rc1);
   }

   if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) )  {
      printf("Thread creation failed: %d\n", rc2);
   }
   pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);
   exit(EXIT_SUCCESS);
}

void *functionC()
{
   pthread_mutex_lock( &mutex1 );
     counter++;
   printf("Counter value: %d\n",counter);
   pthread_mutex_unlock( &mutex1 );
}
```

```
# ./a.out
Counter value: 1
Counter value: 2
```

Deadlocks

- Deadlocks
  - In POSIX, the lock can be created as a recursive lock which does not go into a deadlock when a thread tries to lock the same mutex twice as in the previous example
  - There are other situations when deadlocks may arise: there is a possibility of deadlock in the following scenario
    - *thread 1 code*

      Some executable statements;

      pthread_mutex_lock (&mLock);

      Some more executable statements;

      pthread_mutex_lock (&pLock);

    - *thread 2 code*

      Some executable statements;

      pthread_mutex_lock (&pLock);

      Some more executable statements;

      pthread_mutex_lock (&mLock);

Join

- A join is performed when one wants to wait for a thread to finish.
- A thread calling routine may launch multiple threads then wait for them to finish to get the results. One waits for the completion of the threads with a join.

int pthread_join(pthread_t th, void **thread_return);

th - thread suspended until the thread identified by th terminates, either by calling pthread_exit() or by being cancelled.

thread_return - If thread_return is not NULL, the return value of th is stored in the location pointed to by thread_return.

pthread_join() suspends the calling thread to wait for successful termination of the thread specified as the first argument pthread_t thread with an optional data passed from the terminating thread's call to pthread_exit().

Join example

```
#include <pthread.h>

#define NTHREADS 10
void *thread_function(void *);
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main()
{
   pthread_t thread_id[NTHREADS];
   int i, j;

   for(i=0; i < NTHREADS; i++)    {
     pthread_create( &thread_id[i], NULL, thread_function, NULL );
   }

   for(j=0; j < NTHREADS; j++)    {
     pthread_join( thread_id[j], NULL);
   }
   printf("Final counter value: %d\n", counter);
}

void *thread_function(void *dummyPtr)
{
   printf("Thread number %ld\n", pthread_self());
   pthread_mutex_lock( &mutex1 );
   counter++;
   pthread_mutex_unlock( &mutex1 );
}
```

```
# ./a.out
Thread number 139745550309120
Thread number 139745491560192
Thread number 139745533523712
Thread number 139745525131008
Thread number 139745516738304
Thread number 139745508345600
Thread number 139745499952896
Thread number 139745541916416
Thread number 139745483167488
Thread number 139745474774784
Final counter value: 10
```

Condition variables

Where to use conditional variables

- A producer-consumer situation is a good example. Let's say you have a bunch of worker threads that dequeue jobs from a queue and execute them. If there are no jobs, the threads sleep and are woken up when there is work. Let's call these threads workers. Let's assume there is another thread - the producer - that queues up these jobs for the workers to execute.

  Clearly, since the queue is shared, you need a lock of some sort on it. A mutex would be one option. But a mutex alone will not suffice. ***If there was only a mutex, every worker would try to grab the mutex, and in the case of there being no work, release the mutex and try to acquire it again, repeat the check and so on. This leads to a lot of CPU utilization even though little work is getting done. Further, the more CPU the workers consume, the less CPU the producer gets, thus affecting the throughput of the system***. It also makes the queue lock very heavily contended which again affects the ability of the producer to enqueue jobs, again affecting the throughput of the system.

  A better design* would be to have the workers wait on a condition variable and be woken up when there is work.
  Courtesy: Quora

Conditional variables

- Do look at following: http://pages.cs.wisc.edu/~remzi/OSTEP/threads-cv.pdf
- A condition variable is a mechanism that allows threads to wait (without wasting CPU cycles) for some event to occur.
- Several threads may wait on a condition variable, until some other thread signals this condition variable (thus sending a notification). At this time, one of the threads waiting on this condition variable wakes up, and can act on the event.
- It is possible to also wake up all threads waiting on this condition variable by using a broadcast method on this variable.
- Note that a condition variable does not provide locking. Thus, a mutex is used along with the condition variable, to provide the necessary locking when accessing this condition variable.
- A condition variable is a variable of type pthread_cond_t
- pthread condition variables are created through the following function call or initializer macro

int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

To specify defaults, either use the initializer macro or specify NULL in the second parameter to the call to pthread_cond_init().

Mutex and Condition Variable

- Condition variable always along with the mutex variable
  - A condition predicate must be protected by a mutex. When waiting for a condition, the wait subroutine (either the pthread_cond_wait or pthread_cond_timedwait subroutine) atomically unlocks the mutex and blocks the thread. When the condition is signaled, the mutex is relocked and the wait subroutine returns
  - While a mutex lets threads synchronize by controlling their access to data, a condition variable lets threads synchronize on the value of data
  - This allows a thread to block for some event to happen
  - A condition variable is always associated with a particular mutex
  - A particular condition variable is always used in conjunction with the same mutex and its data
  - The mutex associated with a condition variable protects the shared data

Conditional variables

API	Description
int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr);	Initialize the condition variable referenced by cond with attributes referenced by attr. If attr is NULL, the default condition variable attributes shall be used
int pthread_cond_destroy(pthread_cond_t *cond);	Destroy the given condition variable specified by cond
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);	Puts the current thread to sleep. It requires a mutex of the associated shared resource value it is waiting on.
int pthread_cond_timedwait(pthread_cond_t cond, pthread_mutex_t mutex, const struct timespec abstime);	Place limit on how long it will block. The pthread_cond_timedwait() function shall be equivalent to pthread_cond_wait(), except that an error is returned if the absolute time specified by abstime passes before the condition cond is signaled or broadcasted, or if the absolute time specified by abstime has already been passed at the time of the call.
int pthread_cond_signal(pthread_cond_t *cond);	Signals *one* thread out of the possibly many sleeping threads to wakeup.
int pthread_cond_broadcast(pthread_cond_t *cond);	Signals *all* threads waiting on the cond condition variable to wakeup.

Conditional variables example

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

pthread_mutex_t count_mutex     = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  condition_var   = PTHREAD_COND_INITIALIZER;

void *functionCount1();
void *functionCount2();
int  count = 0;
#define COUNT_DONE  10
#define COUNT_HALT1  3
#define COUNT_HALT2  6

main()
{
  pthread_t thread1, thread2;

  pthread_create( &thread1, NULL, &functionCount1, NULL);
  pthread_create( &thread2, NULL, &functionCount2, NULL);

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

  printf("Final count: %d\n",count);

  exit(EXIT_SUCCESS);
}
```

```
// Write numbers 1-3 and 8-10 as permitted by functionCount2()
void *functionCount1() {
    for(;;)    {
    // Lock mutex and then wait for signal to relase mutex
    pthread_mutex_lock( &count_mutex );
    // Wait while functionCount2() operates on count
    // mutex unlocked if condition varialbe in functionCount2()
signaled.
    pthread_cond_wait( &condition_var, &count_mutex );
    count++;
    printf("Counter value functionCount1: %d\n",count);
    pthread_mutex_unlock( &count_mutex );
    if(count >= COUNT_DONE) return(NULL);
    }
}
// Write numbers 4-7
void *functionCount2() {
    for(;;)    {
    pthread_mutex_lock( &count_mutex );
    if( count < COUNT_HALT1 || count > COUNT_HALT2 )
    {
    // Condition of if statement has been met.
    // Signal to free waiting thread by freeing the mutex.
    // Note: functionCount1() is now permitted to modify "count".
    pthread_cond_signal( &condition_var );
    }
    else
    {
    count++;
    printf("Counter value functionCount2: %d\n",count);
    }
    pthread_mutex_unlock( &count_mutex );
    if(count >= COUNT_DONE) return(NULL);
    }
}
```

Conditional variables

```
# ./a.out
Counter value functionCount1: 1
Counter value functionCount1: 2
Counter value functionCount1: 3
Counter value functionCount2: 4
Counter value functionCount2: 5
Counter value functionCount2: 6
Counter value functionCount2: 7
Counter value functionCount1: 8
Counter value functionCount1: 9
Counter value functionCount1: 10
Final count: 10
```

- The pthread_cond_wait and the pthread_cond_broadcast subroutines must not be used within a signal handler.

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

Read/Write locks

Read/Write Locks

- In many situations data is read more often than it is modified or written.
- It is desirable to allow threads to read concurrently while holding the lock and allow only one thread to hold the lock when data is modified.
- A read-write lock is acquired either for reading or writing, and then is released.
- The thread that acquires the read or write lock must be the one that releases it.

Creating and Destroying Read Write Locks

*int pthread_rwlock_init(pthread_rwlock_t *rwlock, const pthread_rwlockattr_t *attr);*

It initializes a new read/write lock with the specified attributes for use.

If subroutine fails, the rwlock object is not initialized and the contents are undefined.

*int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);*

It destroys the read-write lock object referenced by rwlock and releases any resources used by the lock.

Read/Write Locks

API	Description
int pthread_rwlock_init(pthread_rwlock_t *rwlock, const pthread_rwlockattr_t *attr);	Initializes a new read/write lock with the specified attributes for use. If attr is specified as NULL, all attributes are set to the default read/write lock attributes for the newly created read/write lock.
int pthread_rwlock_rdlock (pthread_rwlock_t *rwlock);	Shall apply a read lock to the read-write lock referenced by rwlock. The calling thread acquires the read lock if a writer does not hold the lock and there are no writers blocked on the lock.
int pthread_rwlock_tryrdlock (pthread_rwlock_t *rwlock);	Shall apply a read lock as in the pthread_rwlock_rdlock() function, with the exception that the function shall fail if the equivalent pthread_rwlock_rdlock() call would have blocked the calling thread. In no case shall the pthread_rwlock_tryrdlock() function ever block; it always either acquires the lock or fails and returns immediately.

Read/Write Locks

API	Description
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);	Shall apply a write lock to the read-write lock referenced by rwlock. The calling thread acquires the write lock if no other thread (reader or writer) holds the read-write lock rwlock. Otherwise, the thread shall block until it can acquire the lock. The calling thread may deadlock if at the time the call is made it holds the read-write lock (whether a read or write lock).
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);	Shall apply a write lock like the pthread_rwlock_wrlock() function, with the exception that the function shall fail if any thread currently holds rwlock (for reading or writing).
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);	Shall release a lock held on the read-write lock object referenced by rwlock. Results are undefined if the read-write lock rwlock is not held by the calling thread.
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);	Shall destroy the read-write lock object referenced by rwlock and release any resources used by the lock.

Read-Write Example

```
#include <pthread.h>
#include <stdio.h>
pthread_rwlock_t       rwlock;
void *rdlockThread(void *arg)
{
  int rc;
  printf("Entered thread, getting read lock\n");
  rc = pthread_rwlock_rdlock(&rwlock);
  printf("got the rwlock read lock\n");
  sleep(5);
  printf("unlock the read lock\n");
  rc = pthread_rwlock_unlock(&rwlock);
 printf("Secondary thread unlocked\n");
  return NULL;
}

void *wrlockThread(void *arg)
{
  int rc;
  printf("Entered thread, getting write lock\n");
  rc = pthread_rwlock_wrlock(&rwlock);
 printf("Got the rwlock write lock, now unlock\n");
  rc = pthread_rwlock_unlock(&rwlock);
  printf("Secondary thread unlocked\n");
  return NULL;
}
```

```
int main(int argc, char **argv)
{
  int                rc=0;
  pthread_t          thread, thread1;
  printf("Main, initialize the read write lock\n");
  rc = pthread_rwlock_init(&rwlock, NULL);
  printf("Main, grab a read lock\n");
  rc = pthread_rwlock_rdlock(&rwlock);
  printf("Main, grab the same read lock again\n");
  rc = pthread_rwlock_rdlock(&rwlock);
  printf("Main, create the read lock thread\n");
  rc = pthread_create(&thread, NULL, rdlockThread, NULL);
  printf("Main - unlock the first read lock\n");
  rc = pthread_rwlock_unlock(&rwlock);
  printf("Main, create the write lock thread\n");
  rc = pthread_create(&thread1, NULL, wrlockThread, NULL);
  sleep(5);
  printf("Main - unlock the second read lock\n");
  rc = pthread_rwlock_unlock(&rwlock);
  printf("Main, wait for the threads\n");
  rc = pthread_join(thread, NULL);
  rc = pthread_join(thread1, NULL);
  rc = pthread_rwlock_destroy(&rwlock);
  printf("Main completed\n");
  return 0;
}
```

Read-Write Example

```
Main, initialize the read write lock
Main, grab a read lock
Main, grab the same read lock again
Main, create the read lock thread
Main - unlock the first read lock
Main, create the write lock thread
Entered thread, getting read lock
got the rwlock read lock
Entered thread, getting write lock
Main - unlock the second read lock
Main, wait for the threads
unlock the read lock
Secondary thread unlocked
Got the rwlock write lock, now unlock
Secondary thread unlocked
Main completed
```

spinlock

spinlock

- Spin locks are a low-level synchronization mechanism suitable primarily for use on shared memory multiprocessors.
- It is usually implemented as a single bit in an integer value. Code wishing to take out a particular lock tests the relevant bit
  - If the lock is available, the "locked" bit is set and the code continues into the critical section
  - If, instead, the lock has been taken by somebody else, the code goes into a tight loop where it repeatedly checks the lock until it becomes available. This loop is the "spin" part of a spinlock
  - The "test and set" operation **must be done in an atomic manner (cmpxchg)** so that only one thread can obtain the lock, even if several are spinning at any given time
- When the calling thread requests a spin lock that is already held by another thread, the second thread spins in a loop to test if the lock has become available.
- When the lock is obtained, it should be held only for a short time, as the spinning wastes processor cycles
- Callers should unlock spin locks before calling sleep operations to enable other threads to obtain the lock
- Spin locks might have lower overall overhead for very short-term blocking

spinlock

API	Description
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);	allocate resources required to use a spin lock, and initialize the lock to an unlocked state.
int pthread_spin_lock(pthread_spinlock_t *lock);	The calling thread acquires the lock if it is not held by another thread. Otherwise, the thread does not return from the pthread_spin_lock() call until the lock becomes available. The results are undefined if the calling thread holds the lock at the time the call is made.
int pthread_spin_trylock(pthread_spinlock_t *lock);	lock a spin lock and fail immediately if the lock is held by another thread
int pthread_spin_unlock(pthread_spinlock_t *lock);	release a locked spin lock.
int pthread_spin_destroy(pthread_spinlock_t *lock);	destroy a spin lock and release any resources used by the lock.

Spinlock & Mutex

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <errno.h>
#include <sys/time.h>

#include <list>

#define LOOPS 10000000

using namespace std;

list<int> the_list;

#ifdef USE_SPINLOCK
pthread_spinlock_t spinlock;
#else
pthread_mutex_t mutex;
#endif

pid_t gettid() { return syscall( __NR_gettid ); }
```

```
void *consumer(void *ptr)
{
 int i;
 printf("Consumer TID %lu\n", (unsigned long)gettid());
 while (1)
 {
   #ifdef USE_SPINLOCK
   pthread_spin_lock(&spinlock);
   #else
   pthread_mutex_lock(&mutex);
   #endif

   if (the_list.empty())
   {
       #ifdef USE_SPINLOCK
       pthread_spin_unlock(&spinlock);
       #else
       pthread_mutex_unlock(&mutex);
       #endif
       break;
       }
       i = the_list.front();
       the_list.pop_front();

       #ifdef USE_SPINLOCK
       pthread_spin_unlock(&spinlock);
       #else
       pthread_mutex_unlock(&mutex);
       #endif
  }
 return NULL;
}
```

Spinlock & Mutex

```
int main()
{
        int i;
        pthread_t thr1, thr2;
        struct timeval tv1, tv2;
        #ifdef USE_SPINLOCK
        pthread_spin_init(&spinlock, 0);
        #else
        pthread_mutex_init(&mutex, NULL);
        #endif

        for (i = 0; i < LOOPS; i++)
        the_list.push_back(i);
        gettimeofday(&tv1, NULL);
        pthread_create(&thr1, NULL, consumer, NULL);
        pthread_create(&thr2, NULL, consumer, NULL);
        pthread_join(thr1, NULL);
        pthread_join(thr2, NULL);
        gettimeofday(&tv2, NULL);
        if (tv1.tv_usec > tv2.tv_usec)
        {
        tv2.tv_sec--;
        tv2.tv_usec += 1000000;
        }
        printf("Result - %ld.%ld\n", tv2.tv_sec - tv1.tv_sec,
        tv2.tv_usec - tv1.tv_usec);
        #ifdef USE_SPINLOCK
        pthread_spin_destroy(&spinlock);
        #else
        pthread_mutex_destroy(&mutex);
        #endif
        return 0;
}
```

```
# g++ -DUSE_SPINLOCK -Wall -
pthread spin.c
# ./a.out
Consumer TID 19734
Consumer TID 19735
Result - 3.350809
```

```
# g++ -pthread spin.c
# ./a.out
Consumer TID 19779
Consumer TID 19778
Result - 3.77880
```

barriers

barrier

- When multiple threads are working together it might be required that the threads wait for each other at a certain point or event in the program before proceeding ahead.

- Let us say we have four threads, each of which is going to initialize a global variable. The 4 variables in turn might be used by all the four threads. Thus it would be feasible that all the threads wait for each other to finish the initialization of the variables before proceeding.

- Such operations can be implemented by adding a barrier in the thread. **A barrier is a point where the thread is going to wait for other threads and will proceed further only when predefined number of threads reach the same barrier in their respective programs.**

- int pthread_barrier_init(pthread_barrier_t *barrier, pthread_barrierattr_t *barrier_attr, unsigned int count);

barrier: The variable used for the barrier

attr: Attributes for the barrier, which can be set to NULL to use default attributes

count: Number of threads which must wait call pthread_barrier_wait on this barrier before the threads can proceed further.

- pthread_barrier_t barrier = PTHREAD_BARRIER_INITIALIZER(count);

barrier

API	Description
int pthread_barrier_init(pthread_barrier_t *barrier, const pthread_barrierattr_t *restrict attr, unsigned count);	Allocate resources for a barrier and initialize its attributes.
int pthread_barrier_wait(pthread_barrier_t *barrier);	Synchronize threads at a specified barrier. The calling thread blocks until the required number of threads have called pthread_barrier_wait() specifying the barrier. The number of threads is specified in the pthread_barrier_init() function
int pthread_barrier_destroy(pthread_barrier_t *barrier);	Destroy the barrier referenced by barrier and release any resources used by the barrier.

barrier example

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

#define THREAD_COUNT 4

pthread_barrier_t mybarrier;

void* threadFn(void *id_ptr) {
  int thread_id = *(int*)id_ptr;
  int wait_sec = 1 + rand() % 5;
  printf("thread %d: Wait for %d seconds.\n", thread_id, wait_sec);
  sleep(wait_sec);
  printf("thread %d: I'm ready...\n", thread_id);

  pthread_barrier_wait(&mybarrier);

  printf("thread %d: going!\n", thread_id);
  return NULL;
}
```

```
int main() {
  int i;
  pthread_t ids[THREAD_COUNT];
  int short_ids[THREAD_COUNT];

  srand(time(NULL));
  pthread_barrier_init(&mybarrier, NULL, THREAD_COUNT + 1);

  for (i=0; i < THREAD_COUNT; i++) {
    short_ids[i] = i;
    pthread_create(&ids[i], NULL, threadFn, &short_ids[i]);
  }

  printf("main() is ready.\n");

  pthread_barrier_wait(&mybarrier);

  printf("main() is going!\n");

  for (i=0; i < THREAD_COUNT; i++) {
    pthread_join(ids[i], NULL);
  }

  pthread_barrier_destroy(&mybarrier);

  return 0;
}
```

Barrier example

```
# ./a.out
thread 1: Wait for 1 seconds.
thread 0: Wait for 2 seconds.
thread 2: Wait for 4 seconds.
main() is ready.
thread 3: Wait for 5 seconds.
thread 1: I'm ready...
thread 0: I'm ready...
thread 2: I'm ready...
thread 3: I'm ready...
main() is going!
thread 3: going!
thread 1: going!
thread 0: going!
thread 2: going!
```

Semaphore

Semaphore

- Semaphores are another type of synchronization primitive

- Two flavors - binary and counting.
- Binary semaphores act much like simple mutexes,
- Counting semaphores can be initialized to any arbitrary value which should depend on how many resources you have available for that particular shared data.

- Many threads can obtain the lock simultaneously until the limit is reached. This is referred to as *lock depth*.
  Semaphores are counters for resources shared between threads. The basic operations on semaphores are: increment the counter atomically, and wait until the counter is non-null and decrement it atomically.

- Semaphores have a maximum value past which they cannot be incremented. The macro SEM_VALUE_MAX is defined to be this maximum value. In the GNU C library, SEM_VALUE_MAX is equal to INT_MAX

Semaphore

- All POSIX semaphore functions and types are prototyped or defined in semaphore.h. To define a semaphore object, use   sem_t sem_name;

Initialize a semaphore

*int sem_init(sem_t *sem, int pshared, unsigned int value);*

sem points to a semaphore object to initialize

pshared is a flag indicating whether or not the semaphore should be shared with fork()ed processes.

value is an initial value to set the semaphore to

On success sem_init returns 0. On failure it returns -1 and sets errno to one of the following values:

EINVAL - value exceeds the maximal counter value SEM_VALUE_MAX

ENOSYS - pshared is not zero. LinuxThreads currently does not support process-shared semaphores. (This will eventually change.)

Example: sem_init(&sem_name, 0, 10);

Semaphore

API	Description
int sem_init (sem_t *sem, int pshared, unsigned int value)	Initializes the semaphore object pointed to by sem. The count associated with the semaphore is set initially to value.
int sem_destroy (sem_t * sem)	Destroys a semaphore object, freeing the resources it might hold. If any threads are waiting on the semaphore when sem_destroy is called, it fails and sets errno to EBUSY.
int sem_wait (sem_t * sem)	Suspends the calling thread until the semaphore pointed to by sem has non-zero count. It then atomically decreases the semaphore count. sem_wait is a cancellation point. It always returns 0.
int sem_trywait (*sem_t * sem*)	Non-blocking variant of sem_wait. If the semaphore pointed to by sem has non-zero count, the count is atomically decreased and sem_trywait immediately returns 0. If the semaphore count is zero, sem_trywait immediately returns -1 and sets errno to EAGAIN.
int sem_post (*sem_t * sem*)	Atomically increases the count of the semaphore pointed to by sem. This function never blocks. sem_post always succeeds and returns 0, unless the semaphore count would exceed SEM_VALUE_MAX after being incremented. In that case sem_post returns -1 and sets errno to EINVAL.
int sem_getvalue (sem_t * sem, int * sval)	Stores in the location pointed to by sval the current count of the semaphore sem. It always returns 0.

Semaphore example

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>

sem_t semaphore;
void threadfunc() {
   while (1) {
       sem_wait(&semaphore);
       printf("I am the thread!\n");
       sem_post(&semaphore);
       sleep(1);
   }
}
int main(void) {
   sem_init(&semaphore, 0, 1);
   pthread_t *mythread;
   mythread = (pthread_t *)malloc(sizeof(*mythread));
   printf("Starting thread, semaphore is unlocked.\n");
   pthread_create(mythread, NULL, (void*)threadfunc, NULL);
   getchar();
   sem_wait(&semaphore);
   printf("Semaphore locked.\n");
       // do stuff with whatever is shared between threads
   getchar();
   printf("Semaphore unlocked.\n");
   sem_post(&semaphore);
   getchar();
   return 0;
}
```

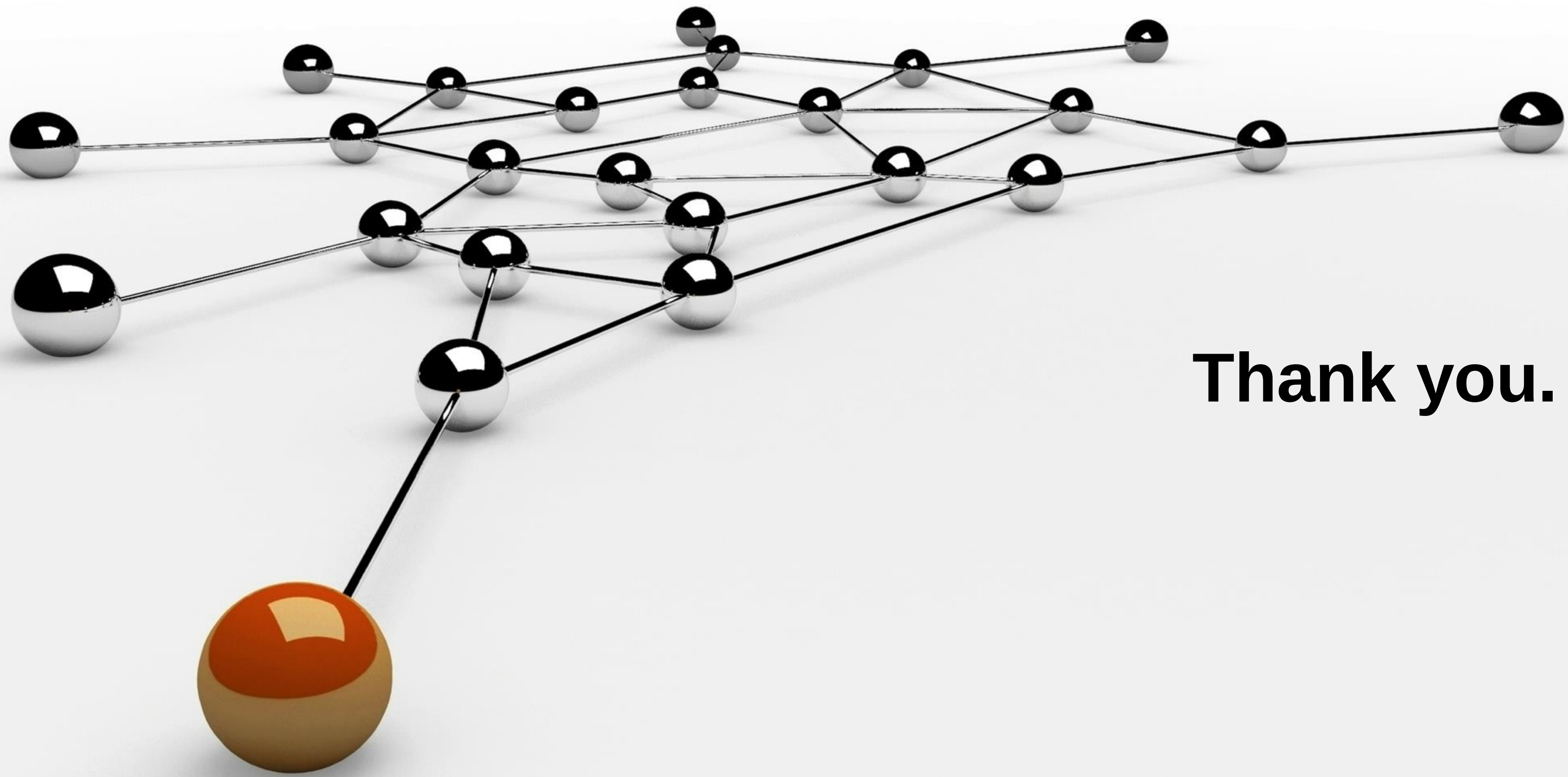
```
# ./a.out
Starting thread, semaphore is
unlocked.
I am the thread!
I am the thread!
I am the thread!
I am the thread!
I am the thread!

Semaphore locked.

Semaphore unlocked.
I am the thread!
I am the thread!
I am the thread!
^C
```

Aricent®

Engineering excellence. Sourced.



Thank you.

www.aricent.com

Additional slides

Multithreaded example

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
pthread_rwlock_t       rwlock =
PTHREAD_RWLOCK_INITIALIZER;

void *wrlockThread(void *arg)
{
  int            rc;
  int            count=0;

  printf("%.8x: Entered thread, getting write
lock\n",
        pthread_self());
  Retry:
  rc = pthread_rwlock_trywrlock(&rwlock);
  if (rc == EBUSY) {
    if (count >= 10) {
      printf("%.8x: Retried too many times,
failure!\n",
            pthread_self());
      exit(EXIT_FAILURE);
    }

    ++count;
    printf("%.8x: Go off an do other work, then
RETRY...\n",
          pthread_self());
    sleep(1);
    goto Retry;
  }
```

```
 printf("%.8x: Got the write lock\n", pthread_self());
 sleep(2);
 printf("%.8x: Unlock the write lock\n",
        pthread_self());
  rc = pthread_rwlock_unlock(&rwlock);
  printf("%.8x: Secondary thread complete\n",
        pthread_self());
  return NULL;
}

int main(int argc, char **argv)
{
  int                rc=0;
  pthread_t          thread, thread2;
  printf("Main, get the write lock\n");
  rc = pthread_rwlock_wrlock(&rwlock);
  printf("Main, create the timed write lock threads\n");
  rc = pthread_create(&thread, NULL, wrlockThread, NULL);
  rc = pthread_create(&thread2, NULL, wrlockThread, NULL);
  printf("Main, wait a bit holding this write lock\n");
  sleep(1);
  printf("Main, Now unlock the write lock\n");
  rc = pthread_rwlock_unlock(&rwlock);
  printf("Main, wait for the threads to end\n");
  rc = pthread_join(thread, NULL);
  rc = pthread_join(thread2, NULL);
  rc = pthread_rwlock_destroy(&rwlock);
  printf("Main completed\n");
  return 0;
}
```

Multithreaded example

```
Main, get the write lock
Main, create the timed write lock threads
Main, wait a bit holding this write lock
00000102: Entered thread, getting write lock
00000102: Go off an do other work, then RETRY...
00000203: Entered thread, getting write lock
00000203: Go off an do other work, then RETRY...
Main, Now unlock the write lock
Main, wait for the threads to end
00000102: Got the write lock
00000203: Go off an do other work, then RETRY...
00000203: Go off an do other work, then RETRY...
00000102: Unlock the write lock
00000102: Secondary thread complete
00000203: Got the write lock
00000203: Unlock the write lock
00000203: Secondary thread complete
Main completed
```