System and Device Programming

Thread Synchronization (part A)

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Introduction

- Semaphores
  - Are a very specific form of Inter-Process Communication
  - They correspond to a counter protected by a lock (i.e., a binary semaphore or lock) with a waiting queue
  - To implement a semaphore, manipulation functions must be atomic operations
    - For this reason, semaphores are normally implemented inside the kernel

Introduction

- Semaphores are a quite sophisticated strategy for synchronization
  - On UNIX-like system threads it is possible to use synchronization strategies faster and less expensive than semaphores
    - These strategies can be used with threads, as multiple threads share the same memory space

Introduction

- For thread synchronization it is possible to adopt the following alternatives
  - Volatile objects
  - Mutexes
  - Reader-Writer Locks
  - Condition Variables
  - Spin Locks

OS' protected objects

Simple and fast binary semaphores

Allow shared (R) and exclusive (W) locks

Used to wait for a specific condition to happen

Mutexes with busy-waiting on a spin-lock

Volatile variables

- When a variable is modified, a thread may hold its value in a register
  - Informs the compiler to not optimize it, as the variable can change at any time
  - The ANSI C **volatile** quantifier ensures that
    - The variable will be **always fetched** from memory before use
    - The variable will be **always stored** to memory after modification
  - Volatile variables must be declare as
    - **volative** int var;

i++;
→
register = i
register++
i = register

Volatile variables

- Unfortunately,
  - Volatile variable can negatively effect performance
    - The process will access its vaue afresh every time
  - Even if a variable is **volatile** a processor may hold its value into the **cache** memory
    - In multi-core architectures each **core** has its **own cache** (level 1 and level 2) memory
    - Each thead may copy the variable into its own cache before committing it into the main memory
    - There is **no assurance** that the new value (even if the object is volatile) **will be visible to threads running on other cores**

Pthread mutex

- A **mutex** is basically a lock that we
  - Set (lock) before accessing a shared resource
    - While it is set, any other thread that tries to set it will block until we release it
  - Release (unlock) when we are done
    - If more than one thread is blocked when we unlock the mutex, then all threads blocked on the lock will be made runnable, and the first one to run will be able to set the lock

```
while (TRUE) {
  ...
  reservation code
  Critical Section
  release code
  ...
  non critical section
}
```

Classical Critical Section protocol

Pthread mutex

- Mutexes are essentially
  - **Binary**, **local** and **fast** semaphores
- A mutex
  - Is a variable represented by the **pthread_mutex_t** data type
  - It can be managed using the following system calls
    - pthread_mutex_init
    - pthread_mutex_lock
    - pthread_mutex_trylock
    - pthread_mutex_unlock
    - pthread_mutex_destroy

A mutex is less general than semaphores (i.e., it can assume only the two values 0 or 1)

pthread_mutex_init

- Before we can use a mutex variable, we must first initialize it by
  - Either setting it to the constant PTHREAD_MUTEX_INITIALIZER, for statically allocated mutexes only and default attributes
  - Calling **pthread_mutex_init**, if we allocate the mutex dynamically (e.g., by calling **malloc**) or we want to set specific attributes
    - If we dinamically allocate it, then we need to call **pthread_mutex_destroy** before freeing the memory (i.e., by calling **free**)

pthread_mutex_init

```
int pthread_mutex_init (
  pthread_mutex_t *mutex,
  const pthread_mutexattr_t *attr
);
```

Always include pthread.h

- Initializes the mutex referenced by **mutex** with attributes specified by **attr**
  - To initialize a mutex with the default attributes, we set **attr** to NULL
- Return value
  - The value, 0 on success
  - An error code, otherwise

pthread_mutex_lock

```
int pthread_mutex_lock (
  pthread_mutex_t *mutex
);
```

- Lock a mutex, i.e., control its value and
  - Blocks the caller if the mutex is locked
  - Acquire the mutex lock if the mutex is unlocked
- Return value
  - The value 0, on success
  - An error code, otherwise

pthread_mutex_trylock

- If a thread can't afford to block, it can use pthread_mutex_trylock to lock the mutex conditionally
  - If the mutex is unlocked at the time **pthread_mutex_trylock** is called, then pthread_mutex_trylock will lock the mutex without blocking and return 0
  - Otherwise, **pthread_mutex_trylock** will fail, returning EBUSY without locking the mutex

pthread_mutex_trylock

```
int pthread_mutex_trylock (
  pthread_mutex_t *mutex
);
```

- Similar to pthread_mutex_lock, but returns without blocking the caller if the mutex is locked
- Return value
  - The value 0, if the lock has been successfully acquired
  - **EBUSY** error if the mutex was already locked by another thread

pthread_mutex_unlock

```
int pthread_mutex_unlock (
  pthread_mutex_t *mutex
);
```

- Release (unlock) the mutex lock (typically at the end of a sritical section)
- Return value
  - The value 0, on success
  - An error code, otherwise

pthread_mutex_destroy

```
int pthread_mutex_destroy (
  pthread_mutex_t *mutex
);
```

- Free **mutex** memory
  - Used for dynamically allocated mutexes
  - The mutex cannot be used anymore
- Return value
  - The value 0, on success
  - An error code, otherwise

Example

Use a dynamically allocated mutex to protect a CS

```
pthread_mutex_t *lock;
...
lock = malloc (1 * sizeof (pthread_mutex_t));
if (lock == NULL) ... error ...
if (pthread_mutex_init (lock, NULL) != 0) ... error ..

...
pthread_mutex_lock (lock);
CS
pthread_mutex_unlock(lock);
...

pthread_mutex_destroy (lock)
free (lock);
```

Reader-Writer Locks

- Reader–writer locks are similar to mutexes, but they allow for higher degrees of parallelism
  - With a mutex, the state is either locked or unlocked, and only one thread can lock it at a time
  - With a reader–writer lock, three states are possible
    - Locked in read mode
    - Locked in write mode
    - Unlocked
- Reader–writer locks are well suited for situations in which data structures are read more often than they are modified

Reader-Writer Locks

- Only **one** thread at a time can hold a reader–writer lock in write mode, but **multiple** threads can hold a reader–writer lock in read mode at the same time
  - When a reader–writer lock is write-locked
    - All threads attempting to lock block until the lock is unlocked
  - When a reader–writer lock is read-locked
    - All threads attempting to lock it in read mode are given access
    - Any threads attempting to lock it in write mode block until all the threads have released their read locks

Reader-Writer Locks

- Reader-writer locks are also called shared–exclusive locks
  - When a lock is read locked, it is said to be locked in **shared** mode
  - When a lock is write locked, it is said to be locked in **exclusive** mode
- What about Reader-Writer precedence?
  - Although implementations vary, reader–writer locks usually block additional readers if a lock is already held in read mode and a thread is blocked trying to acquire the lock in write mode
    - This prevents a constant stream of readers from starving waiting writers

phread_rwlock_init

Always include pthread.h

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

- Locks must be initialized before usage
  - Use the constant **PTHREAD_RWLOCK_INITIALIZER** for statically defined locks with default attributes
  - Use **pthread_rwlock_init** for dynamically allocated lock or when attributes are not the default one
    - The default attribute is **NULL**

phread_rwlock_*lock

```
int pthread_rwlock_rdlock (pthread_rwlock_t *rwlock);

int pthread_rwlock_wrlock (pthread_rwlock_t *rwlock);

int pthread_rwlock_unlock (pthread_rwlock_t *rwlock);
```

- These functions are used to lock a reader-writer lock and to unlock it
  - In read mode
  - In write mode
- Return value
  - The value 0, on success
  - An error code, otherwise

phread_rwlock_try*lock

```
int pthread_rwlock_tryrdlock(
  pthread_rwlock_t *rwlock);

int pthread_rwlock_trywrlock(
  pthread_rwlock_t *rwlock);
```

- Conditional versions (not blocking functions) to lock a reader-writer lock
  - In read mode
  - In write mode

phread_rwlock_timed*lock

```
int pthread_rwlock_timerdlock(
  pthread_rwlock_t *restrict rwlock,
  const struct timespec *restrict tsptr);

int pthread_rwlock_timedwrlock(
  pthread_rwlock_t *restrict rwlock,
  const struct timespec *restrict tsptr);
```

Always include pthread.h and time.h

- Reader-writer locks with timeouts
  - Lock with a timeout to give applications a way to avoid blocking indefinitely
- Return value
  - The value 0, on success
  - An error code, otherwise

phread_rwlock_destroy

```
int pthread_rwlock_destroy (
  pthread_rwlock_t *rwlock
);
```

- Locks must be destroyed after use
  - Call **pthread_rwlock_destroy** before freeing them up

Condition Variables

- Condition variables provide a place for threads to rendezvous
- Condition variables allow threads to wait in a race-free way for arbitrary conditions to occur
  - The condition itself is protected by a mutex
  - A thread must first lock the mutex to change the condition state
  - Other threads will not notice the change until they acquire the mutex, because the mutex must be locked to be able to evaluate the condition

pthread_cond_init

```
int pthread_cond_init (
  pthread_cond_t *restrict cond,
  const pthread_condattr_t *restrict attr
);
```

Always include pthread.h

- A condition variable is represented by the pthread_cond_t data type
- Before a condition variable is used, it must first be initialized

pthread_cond_init

- A condition variable can be initialized in two ways
  - We can assign the constant PTHREAD_COND_INITIALIZER to a statically allocated condition variable
  - We can use **pthread_cond_init** to initialize it when the condition variable is allocated dynamically
    - The parameter **attr** may be set to NULL

```
int pthread_cond_init (
  pthread_cond_t *restrict cond,
  const pthread_condattr_t *restrict attr
);
```

pthread_cond_wait

```
int pthread_cond_wait (
  pthread_cond_t *restrict cond,
  pthread_mutex_t *restrict mutex
);
```

Condition variable and **mutex**

- Function pthread_cond_wait waits for the condition cond to be true
- The mutex mutex protects the condition
  - It must be locked when the condition is checked
  - It will be released when the operation is done

pthread_cond_wait

- The general procedure is as follows
  - A thread obtains the mutex and it tests a predicate
  - If the predicate is
    - **Not satisfied**, the thread executes **pthread_cond_wait** which releases the mutex and it awaits on the condition variable
      - The mutex must be released to allow other threads to check the condition
      - When the condition variable is signaled, the thread wakes up and the predicate is checked again
    - **Satistied** the thread goes on and it unlock the mutex

```
int pthread_cond_wait (
  pthread_cond_t *restrict cond,
  pthread_mutex_t *restrict mutex
);
```

pthread_cond_timedwait

```
int pthread_cond_timedwait (
  pthread_cond_t *restrict cond,
  pthread_mutex_t *restrict mutex,
  const struct timespec *restrict tsptr
);
```

- This variant of pthread_cond_wait has a timeout
  - It returns an error code if the condition hasn't been satisfied in the specified amount of time

pthread_cond_signal & broadcast

```
int pthread_cond_signal (pthread_cond_t *cond);

int pthread_cond_broadcast (pthread_cond_t *cond);
```

- These functions are used to notify threads that a condition has been satisfied
  - **thread_cond_signal** will wake up at least one thread waiting on a condition
  - **pthread_cond_broadcast** will wake up all threads waiting on a condition

pthread_cond_destroy

```
int pthread_cond_destroy (
  pthread_cond_t *cond
);
```

- We can use the pthread_cond_destroy function to deinitialize a condition variable before freeing its underlying memory

Example

- Run a thread and wait for it
  - Suppose pthread_join does not exist
  - Use a condition variable to wait for a thread

```
pthread_t p;
void *status;

pthread_create (&p, NULL, child, NULL);

pthread_join (p, &status);
```

No pthread_join

Example

Solution with spin-lock
(never use it)

```
void *child(void *arg) {
  done = 1;
  return NULL;
}
int main(int argc, char *argv[]) {
  pthread_t p;
  pthread_create (&p, NULL, child, NULL);
  while (done == 0)
    ; // spin
  return 0;
}
```

pthread_join

Simplest solution, grossly inefficient as it wastes CPU cycles (and in some cases may be incorrect)

Solution with CV

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;

int done = 0;
```

Example

```
void *child(void *arg) {
  pthread_mutex_lock (&m);
  done = 1;
  pthread_cond_signal (&cv);
  pthread_mutex_unlock (&m);
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t p;
  pthread_create (&p, NULL, child, NULL);
  pthread_mutex_lock (&m);
  while (done == 0)
    pthread_cond_wait (&cv, &m);
  pthread_mutex_unlock(&m);
  return 0;
}
```

pthread_join

Does it work?

Example

```
void *child(void *arg) {
  pthread_mutex_lock (&m);
  done = 1;
  pthread_cond_signal (&cv);
  pthread_mutex_unlock (&m);
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t p;
  pthread_create (&p, NULL, child, NULL);
  pthread_mutex_lock (&m);
  while (done == 0)
    pthread_cond_wait (&cv, &m);
  pthread_mutex_unlock(&m);
  return 0;
}
```

The parent runs first:

1. It will acquire the lock, check done, and as done=0, it will go to sleep releasing the lock
2. The child will run, set done to 1, signal the cv, release the mutex, and quit
3. The parent will be woken-up by the signal with the mutex locked, unlock the mutex, and return

In this case the "job" is done by the **cv** on which the parent waits

Example

```
void *child(void *arg) {
  pthread_mutex_lock (&m);
  done = 1;
  pthread_cond_signal (&cv);
  pthread_mutex_unlock (&m);
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t p;
  pthread_create (&p, NULL, child, NULL);
  pthread_mutex_lock (&m);
  while (done == 0)
    pthread_cond_wait (&cv, &m);
  pthread_mutex_unlock(&m);
  return 0;
}
```

The child runs first:

1. It will set done to 1, signal the cv, unlock the mutex, and terminate. When the mutex is unlocked there is no one waiting. Thus, the signal **has no effect**
2. The parent will get to the critical section, lock the mutex, as done==1 it will skop the wait, unlock the mutex, and terminate

In this case the "job" is done by the variable done as the parent never does a wait

Example

Is the variable **done** required?

```
void *child(void *arg) {
  pthread_mutex_lock (&m);
  pthread_cond_signal (&cv);
  pthread_mutex_unlock (&m);
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t p;
  pthread_create (&p, NULL, child, NULL);
  pthread_mutex_lock (&m);
  pthread_cond_wait (&cv, &m);
  pthread_mutex_unlock(&m);
  return 0;
}
```

The code is broken. In fact, **iff** the child runs first:

1. It will signal the cv but as there is no one waiting, the signal has no effect
2. The parent will get to the critical section, lock the mutex, and wait on cv forever

Then, variable **done** records the status the threads are interested in knowing

Example

Is the **mutex** m required?

```
void *child(void *arg) {
  done = 1;
  pthread_cond_signal (&cv);
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t p;
  pthread_create (&p, NULL, child, NULL);
  while (done == 0)
    pthread_cond_wait (&cv);
  return 0;
}
```

The code is broken. There is a subtle **race condition**:

1. The parent runs first, and it checks done. As done==0 it is ready to go to sleep on the wait, but before going on the wait the child runs
2. The child set done to 1 and it signal cv. But at this point the parent is not waiting, thus the signal is lost
3. The parent will go on the wait and wait forever

This is not a correct implementation, because there is no mutex. Let us suppose it is correct just for the sake of the example

Using the mutex may not be always required around the signal but it is **always** required around the wait

Example

Is the **while** required or we can use a if?

```
void *child(void *arg) {
  pthread_mutex_lock (&m);
  done = 1;
  pthread_cond_signal (&cv);
  pthread_mutex_unlock (&m);
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t p;
  pthread_create (&p, NULL, child, NULL);
  pthread_mutex_lock (&m);
  if (done == 0)
    pthread_cond_wait (&cv, &m);
  pthread_mutex_unlock(&m);
  return 0;
}
```

The code is broken. Let us suppose there are two entities doing the wait

1. The first "main" waits on the wait and the second one has to start
2. The child runs and signal the cv
3. The first main in released from the wait but before it really starts, the second main enters the critical section and as done==1 it skips the wait, and runs
4. The same does the first main

Signaling a thread wakes-it up but there is **no** guarantee that when it runs the **state** will still **be the same. The while is required.**

Condition Variables: Conclusions

- To summarize
  - The mutex is used to protect the condition variable
  - The mutex must be locked before waiting
  - The wait will "atomically" unlock the mutex, allowing others access to the condition variable
  - When the condition variable is signalled (or broadcast to) one or more of the threads on the waiting list will be woken-up and the mutex will be magically locked again for that thread
- This is just the way condition variables are implemented

Condition Variables: Conclusions

- Condition variables allow a thread to notify other threads when something needs to happen
- Thus, a condition variable relieves the user of the burden of polling some condition and waiting for the condition without wasting resources

Spin Locks

- A spin lock is like a mutex but instead of blocking a process by sleeping, the process is blocked by busy-waiting (spinning) until the lock can be acquired
  - A spin lock could be used when
    - Locks are held for very short periods of times
    - Threads do not want to incur the cost of being descheduled
- Spin locks are often used as low-level primitives to implement other types of locks

pthread_spin_init

```
int pthread_spin_init (
  pthread_spinlock_t *lock, int pshared
);
```

Always include pthread.h

- We can initialize a spin lock with function pthread_spin_init

pthread_spin_*lock

```
int pthread_spin_lock (pthread_spinlock_t *lock);
int pthread_spin_trylock (pthread_spinlock_t *lock);
int pthread_spin_unlock (pthread_spinlock_t *lock);
```

- To lock the spin lock, we can call
  - pthread_spin_lock, which will spin until the lock is acquired
  - pthread_spin_trylock, which will return the EBUSY error if the lock can't be acquired immediately
    - Note that pthread_spin_trylock doesn't spin
- Regardless of how it was locked, a spin lock can be unlocked by calling pthread_spin_unlock

pthread_spin_destroy

```
int pthread_spin_destroy (
  pthread_spinlock_t *lock
);
```

- To deinitialize a spin lock, we can call the pthread_spin_destroy function

Conclusions

- Volatile variables
  - According to C++20 most uses of the volatile keyword are deprecated
    - New standards do no appropriately define its meaning
  - Many of these uses are related to ensuring atomic access of variables
    - To grant an atomic access to an object use another strategy

Conclusions

- Mutexes
  - Are used for mutual exclusion problems
    - They cannot be used for general synchronization
  - They are efficient
    - Use them as long as possible for simple critical section problems

Conclusions

- Spin-locks
  - Are an alternatives to mutexes
  - Use them only in exceptional cases, such as the one in which it is guarantted that waiting times will be really short
- Read-write locks
  - They can be used in some peculiar situation
    - The access can be given to more "readers" in shared mode
    - Plese remind the Readers and Writers schemes

Conclusions

- Condition variables
  - Are generally used to avoid busy waiting

```
while (done == 0);
```

```
pthread_mutex_lock (&m);
while (done == 0)
  pthread_cond_wait (&cv, &m);
pthread_mutex_unlock(&m);
```

- Used when one or more threads are waiting for a specific condition to come true

Conclusions

- Semaphores
  - Are used for generic synch schemes
    - Used when there is a shared resource that can be available or unavailable based onsome integer number of things
    - Remind the producer/consumer problem
  - Semaphores are very general and sophisticated
    - They are expensive
    - There are many cases in which they can do the same thing of a condition variable
      - A condition variable is essentially a wait-queue (but it needs a mutex)
      - A semaphore is essentially a counter + a mutex + a wait queue