# Data Races, Synchronization Primitives - Spin locks, Semaphores, Barriers

# 1.Data Races

# *Data Races*

Data races are most common error found in parallel code.

It occurs when multiple threads use the same data item and one or more of those threads are updating it.

**Listing 2.1 Updating the value at an Address**

```
Void update(int *a)
{
        *a = *a + 4;
}
```

**Listing 2.2 SPARC Disassembly for Incrementing a variable helid in Memory**

```
ld [%o0], %o1          // Load *a
add %o1, 4, %o1        // Add 4
st %o1, [%o0]          // Store *a
```

# Data Races

- Suppose this code occurs in a multithreaded application and 2 threads try to increment the same variable at the same time.

| Value of variable a = 10 | |
|---|---|
| **Thread 1** | **Thread 2** |
| ld [%o0], %o1 **// Load %o1 = 10** | ld [%o0], %o1 **// Load %o1 = 10** |
| add %o1, 4, %o1 **// Add %o1 = 14** | add %o1, 4, %o1 **// Add %o1 = 14** |
| st %o1, [%o0] **// Store %o1** | st %o1, [%o0] **// Store %o1** |
| **Value of variable a = 14** | |

- Each thread adds 4 to the variable, but because they do it at exactly the same time, the value 14 ends up being stored into the variable. If 2 threads had executed the code at different times, then the variable would have ended up with the value of 18

# Data Races

- This is the situation where both threads run simultaneously.
- This illustrates a common kind of data race and possibly the easiest one to visualize.
- Another situation : When one thread is running, but the other thread has been context switched off of the processor.
- Imagine that the first thread has loaded the value of the variable a and then gets context switched off of the processor.
- When it eventually runs again, the value of the variable a will have changed and the final store of the restored thread will cause the value of the variable a to regress to an old value.

# Using Tools to Detect Data Races

```
#include <pthread.h>
int counter = 0;
void * func(void *params)
{
    counter++;
}
void main()
{
    pthread_t thread1, thread2;
    pthread_create(&thread1,0,func,0);

    pthread_create(&thread2,0,func,0);
    pthread_join(thread1,0);
    pthread_join(thread2,0);
}
```

Both threads will attempt to increment the variable counter. We can compile this code with GNU gcc and then use Helgrind, which is part of the Valgrind1 suite, to identify the data race.

**Valgrind** is a tool that enables an application to be instrumented and its runtime behavior examined.

**Helgrind** tool uses this instrumentation to gather data about data races.

# *Listing 2.4 Using Helgrind to detect Data Races*

```
$ gcc –g race.c –lpthread
$ valgrind –tool=helgrind ./a.out
…
==4742==
==4742== Possible data race during write of size 4 at
   0x804a020 by thread #3
==4742== at 0x402A89B:
   mythread_wrapper(hg_intercepts.c:194)
…
```

- The output from Helgrind shows that there is a potential data race between 2 threads, both executing line 7 in the file race.c
- This is the anticipated result, but it is pointed out that the tools will find some false positives.
- The programmer may write the code where different threads access the same variable, but the programmer may know that there is an enforced order that stops an actual data race.
- The tools, however, may not be able to detect the enforced order and will report the potential data race.

# Avoiding the Data Races

Although it can be hard to identify data races, avoiding them can be very simple.

- Make sure that only one thread can update the variable at a time
- The easiest way to do this is to place a synchronization lock around all accesses to that variable and ensure that before referencing the variable, the thread must acquire the lock.

**Listing 2.6 Code Modified to Avoid Data Races**

```
Void* func(void *params)
{
        pthread_mutex_lock(&mutex);
        counter++;
         pthread_mutex_unlock(&mutex);
}
```

# *2.Synchronization Primitives*

# Synchronization Primitives

Synchronization is used to coordinate the activity of multiple threads.

This might be to ensure that

- Shared resources are not accesses by multiple threads simultaneously or
- that all work on those resources is complete before new work starts

- Mutexes and Critical Sections
  - Simplest form of synch is a mutually exclusive (mutex) lock.
  - Only 1 thread at a time can acquire a mutex lock
  - So they can be placed around a data structure to ensure that the data structure is modified by only one thread at a time

# *Contended mutex*

**Listing 2.7 Placing Mutex locks Around Accesses to Variables**

```
int counter;
mutex_lock mutex;

void Increment()
{
        acquire(&mutex);
        counter++;
        release(&mutex);
}

void Decrement()
{
        acquire(&mutex);
        counter--;
        release(&mutex);
}
```

# Contended mutex

- In the example, the 2 routines Increment() and Decrement() will either increment or decrement the variable counter.
- To modify the variable, a thread has to first acquire the mutex lock.
- Only one thread at a time can do this; all the other threads that want to acquire the lock need to wait until the thread holding the lock releases it.
- Both routines use the same mutex; consequently, only one thread at a time can either increment or decrement the variable counter.
- If multiple threads are attempting to acquire the same mutex at the same time, then only one thread will succeed, and the other threads will have to wait. This situation is called **contended mutex**.

# Critical region

- The region of code between the acquisition and release of a mutex lock is called a critical section or critical region.
  - Code in this region will be executed by only one thread at a time.

**Listing 2.8 Placing a Mutex Lock Around a Region of Code**

```
Void* threadSafeMalloc(size_t size)
{
        acquire(&mallocMutex);
        void *memory = malloc(size);
        release(&mallocMutex);
        return memory;
}
```

- Example: Imagine that an operating system does not have an implementation of malloc() that is thread-safe, or safe for multiple threads to call at the same time.
- One way to fix this is to place the call to malloc() in a critical section by surrounding it with a mutex lock, as shown in Listing 2.8.

- If all the calls to malloc() are replaced with the threadSafeMalloc() call, then only one thread at a time can be in the original malloc() code, and the calls to malloc() become thread-safe.
- Threads block if they attempt to acquire a mutex lock that is already held by another thread.
  - **Blocking** means that the threads are sent to sleep either immedietly or after a few unsuccessful attempts to acquire the mutex.
- Problem with this approach:
  - It can serialize a program.
  - If multiple threads simultaneously call threadSafeMalloc(), only one thread at a time will make progress.
  - This causes the multithreaded program to have only a single executing thread, which stops the program from taking advantage of multiple cores.

# *2.1 Spin Locks*

# Spinlocks

- Are a Simple single-holder lock.
- If process attempts to acquire a lock and it is not available, the process will keep trying to acquire the lock (spinning) until the lock is available.

  Once the lock is acquired you can process the critical region and release the lock

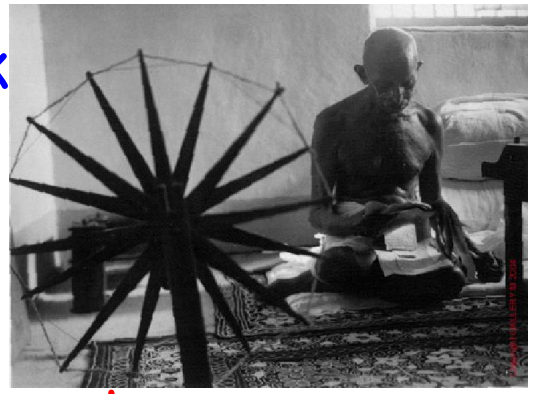
**SMP**

```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;
unsigned long flags;
spin_lock_irqsave(&mr_lock, flags);
/* critical section ... */
spin_unlock_irqrestore(&mr_lock, flags);
```

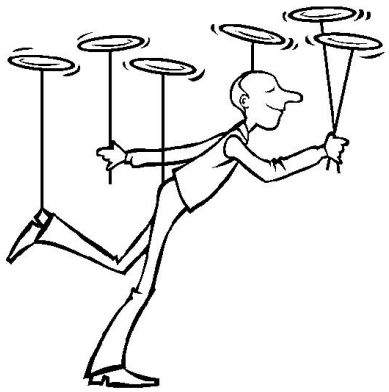
**Unicore**

```
unsigned long flags;
save_flags(flags);
cli();
/* critical section ... */
restore_flags(flags);
```

# Spinlocks



- Spinlocks should only be used when the lock not going to be held very long. Or other processes will spin doing nothing.
- Do not use spin locks on processes that will go to sleep. Or else a deadlock could occur.
- spin_lock_irqsave() and spin_unlock_irqrestore() are atomic operations that acquire the lock and disable interrupts with one atomic operation
- we need to both acquire the lock and disable interrupts in one operation or else we could have a race condition

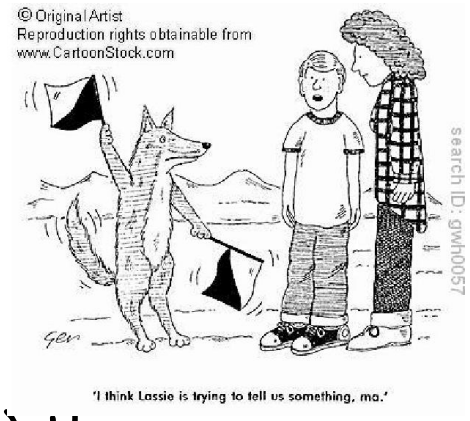


# *2.1 Semaphores*

# Semaphores

- Semaphores are sleeping locks because they can cause a task to sleep on contention instead of spin. They are used when you are going to take hold of a lock for a long time
- There is overhead for putting task to sleep and waking task up. This should not be used for locks that are held for a short amount of time.
- Semaphores have a queue for how many threads are waiting. If the number is positive, this is the amount of threads that are waiting on the queue.
- If negative, the semaphore is unavailable and the absolute value is the usage count

# Semaphore




- Semaphores manipulate 2 methods up(), down() they increment/decrement the wait queue
- up_interruptible() the calling process is added to the wait queue and blocked
- down_interruptible() the process obtains the semaphore
- up and down increment of semaphores have to atomic operations

```
struct semaphore mr_sem;
sema_init(&mr_sem, 1);        /* usage count is 1 */
if (down_interruptible(&mr_sem))
  /* semaphore not acquired; received a signal ... */
/* critical region (semaphore acquired) ... */
up(&mr_sem);
```

# Reader/Writer Locks

- It is safe for multiple threads to read data concurrently. As long as nothing modifies the data R/W locks allow multiple concurrent readers but only a single writer.
- If data is access, it naturally divides into clear reading and writing patterns especially with greater amount of reading time, then writing time. If so R/W locks are preferred.

  R/W spinlock is called rwlock, similar to spinlock with the exception of separate R/W locking.

  R/W locks can give appreciable optimization

  Attempting to acquire exclusive access while holding reader access will deadlock

# *reader/writer locks*

- Data Races : shared data is modified
- Multiple threads reading the shared data do not present the problem.
- Read-only data does not , therefore need protection with some kind of lock.

- A writer cannot acquire the write lock until all the readers have released their reader locks.
  - This lock stops allowing further readers to enter.
  - This action causes the number of readers holding the lock to diminish and will eventually allow the writer to get exclusive access to the lock.

# *reader/writer locks : Example 1*

**Listing 2.9 Using a Readers-Writer Lock**

```
int readData(int cell1, int cell2)
{
        acquireReaderLock(&lock);
        int result = data[cell1] + data[cell2];
        releaseReaderLock(&lock);
        return result;
}

int writeData(int cell1, int cell2, int value)
{
        acquireWriterLock(&lock);
        data[cell1] += value;
        data[cell2] -= value;
        releaseWriterLock(&lock);
}
```

# *reader/writer locks*

- The code shows how a readers-writer lock might be used. Most threads will be calling the routine readData() to return the value from a particular pair of cells.
- Once a thread has a reader lock, they can read the value of the pair of cells, before releasing the reader lock.
- To modify the data, a thread needs to acquire a writer lock.
- This will stop any reader threads from acquiring a reader lock.
- When all reader threads have released their lock, and only at that point does the writer thread actually acquire the lock and is allowed to update the data.