Concurrency and Race Conditions
(Linux Device Drivers, 3rd Edition (www.makelinux.net/ldd3))

Concurrency refers to the situation when the system tries to do more than one thing at once

Concurrency-related bugs are some of the easiest to create and some of the hardest to find

In early Linux kernels, there were relatively few sources of concurrency

Symmetric multiprocessing (SMP) systems were not supported by the kernel, and the only cause of concurrent execution was the servicing of *hardware interrupts*

The Linux kernel has evolved to a point where many more things are going on simultaneously

This provides far greater performance and scalability, but unfortunately, significantly complicates the task of kernel programming

Concurrency and Race Conditions

Your DD code must handle concurrency and you must have a good understanding of the facilities provided by the kernel for concurrency management

From the *write* method of the *scull* code, there is a check to determine if memory has been allocated or not:

```
if (!dptr->data[s_pos])
   {
   dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
   if (!dptr->data[s_pos])
      goto out;
   }
```

Suppose that two processes, "A" and "B", are independently attempting to write to the same offset within the same scull device

And each process reaches the first *if stmt* at the same time

If the pointer in question is NULL, each process will decide to allocate memory and each will assign the resulting pointer to *dptr->data[s_pos]*

Since both processes are assigning to the same variable, the second process wins

Concurrency and Race Conditions

For example, if process "A" assigns first, *scull* will 'loose' the memory allocated by "A", and will result in a memory leak -- an ever increasing kernel size

We refer to this situation as a **race condition**, where an 'unlikely' access pattern results in corrupting system state

Race conditions result from *uncontrolled* access to *shared data*

Here we have a memory leak, but other situations can lead to system crashes, corrupt data and security problems

Concurrency and its management:

In Linux systems, there are multiple sources of concurrency:

- Multiple *user-space* processes are running and can access DD code in many different possible ways
- Processes running on SMP systems can be executing your DD code SIMULTANEOUSLY
- Kernel code is now preemptible
- Device interrupts are asynchronous events, allowing concurrent access to DD code

Concurrency and Race Conditions

In an environment where anything can happen at any time, how does a DD programmer manage the possibility of absolute chaos?

Turns out that most race conditions can be avoided

- Through careful programming
- Through the use of kernel concurrency control primitives
- And through application of some basic principles.

Note that race conditions result from *shared access to resources*

First rule of thumb is to minimize the amount of shared resources (no sharing, no race conditions, it's that simple)

Avoid the use of *global* variables

Unfortunately, avoiding shared resources is not always possible since hardware resources, by their nature, are shared

Bear in mind that sharing also occurs when you pass a pointer to another part of the kernel, so it's difficult to avoid

Concurrency and Race Conditions

Main rule associated with resource sharing:

If any time that a hardware or software resource *is shared* beyond a single thread of execution,

And the possibility exists that one thread can have an *inconsistent view* of that resource (in *scull*, process "B" was unaware that "A" had allocated memory)

Then YOU MUST explicitly manage access to that resource

In the *scull* example, we must control access to the *scull* data structure

The usual mechanism for access management is called **locking** or **mutual exclusion**, which ensure that only one thread can manipulate a shared resource at any time

Semaphores and Mutexes

Mechanisms to make operations on a shared resource **atomic**

This is accomplished by setting up **critical sections**: code that can be executed by only one thread at any given time

Concurrency and Race Conditions

Critical sections differ in their requirements, so the kernel provides different primitives for different needs:

- Do all accesses happen within a process context, or are some access by asynchronous process interrupts?
- Are there any response time requirements, i.e., limits on how long a process can 'hold' a critical section?
- Does the process hold any other critical sections?

Care must be taken on the last one b/c the system can **deadlock** if it is true

Sleep: When a Linux process reaches a point where it cannot make any further processes, it yields the processor to another process (happens frequently with I/O)

There are situations where a process cannot sleep, as we shall see

For *scull*, none of the above are relevant so we can use a **semaphore** that puts the process to sleep

Note that requesting memory through *kmalloc* can put the process to sleep

Concurrency and Race Conditions

Semaphores:

A semaphore is a single integer variable combined with a *pair of functions*, called *P* and *V*

A process wishing to enter a critical section calls *P* on the semaphore

- If the semaphore's value is > 0, the value is decremented by 1 and the process continues
- If the semaphore's value is <= 0, the process must wait for another process to release the semaphore, which is accomplished by calling *V*

Calling *V* increments the semaphore and, if necessary, wkes up processes that are waiting to enter the critical section

When semaphores are used for *mutual exclusion*, their initial value is set to 1 (in this case, they are called *mutex*)

This is by far the most common case

Concurrency and Race Conditions

Linux Semaphores (must include <asm/semaphore.h>)

• First step is to initialize a semaphore -- for mutex types, the following will initialize the semphore to 1 and 0, resp.

```
void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);
```

• To obtain access to a share resource, a process calls one of the following:

```
void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
int down_trylock(struct semaphore *sem);
```

down decrements the value of the semaphore and waits (possibly forever)

Do not use **non**interruptible operations unless there is no alternative -- Non-interruptible operations are a good way to create *unkillable processes* (the dreaded "D state" seen in ps), and annoy your users.

down_interruptible does the same, but the operation is interruptible

This one allows a *user-space* process that is waiting on a semaphore to be interrupted by the user, which causes the semaphore to return a nonzero value

Concurrency and Race Conditions

down_trylock never sleeps -- if the semaphore is not available at the time of the call, *down_trylock* returns immediately with a nonzero return value

If 'down' succeeds, the process is said to have *taken out* the semaphore

The Linux equivalent to *V* is *up*

```
void up(struct semaphore *sem);
```

Note that if an error occurs while the semaphore is held, the process must *release* the semaphore before returning an error status to the caller

In *scull*, the *struct scull_dev* data structure contained a semaphore:

```
struct scull_dev {
    struct scull_qset *data;  /* Pointer to first quantum set */
    int quantum;              /* the current quantum size */
    int qset;                 /* the current array size */
    unsigned long size;       /* amount of data stored here */
    unsigned int access_key;  /* used by sculluid and scullpriv */
    struct semaphore sem;     /* mutual exclusion semaphore     */
    struct cdev cdev;         /* Char device structure      */
};
```

Concurrency and Race Conditions

This provided a 'per data structure' mutual exclusion mechanism

A global mechanism can also be used but would limit concurrency since each *scull* data structure is independent of the others

The initialization code initialized the semaphores:

```
for (i = 0; i < scull_nr_devs; i++) {
    scull_devices[i].quantum = scull_quantum;
    scull_devices[i].qset = scull_qset;
    init_MUTEX(&scull_devices[i].sem);
    scull_setup_cdev(&scull_devices[i], i);
}
```

It is important to note that this initialization occurs BEFORE the device is made available with *scull_setup_cdev* (described earlier)

The code for *scull_write* (see on-line text) begins by acquiring the semaphore:

```
if (down_interruptible(&dev->sem))
    return -ERESTARTSYS;
```

If the user 'interrupts', the nonzero return value causes the kernel to restart the call or return an error to the user

Concurrency and Race Conditions

SpinLocks: (must include <linux/spinlock.h>)

Semaphores are useful but most locking uses a 'spinlock' instead

Spinlocks provide mutual exclusion by providing two values, "locked" and "unlocked", that is usually implemented using a single bit in an integer value

Code wishing to 'take out' a particular lock tests the relevant bit

If lock is available, the 'locked' bit is set and the code continues into the critical section

If not available, the code goes into a tight loop where it repeatedly checks the lock (it spins)

The primitive is implemented such that the 'test and set' operation is atomic

Note that these only work on SMP systems and preemptive kernels, otherwise deadlock occurs

spinlocks on uniprocessor systems without preemption are coded to do nothing

Concurrency and Race Conditions

Initialization:

```
void spin_lock_init(spinlock_t *lock);
```

Before the critical section:

```
void spin_lock(spinlock_t *lock);
```

After the critical section:

```
void spin_unlock(spinlock_t *lock);
```

Note that *spinlocks* are NOT noninterruptible and there are variants to the above (as with semaphores)

Deadlock can happen easily and needs to be avoided:

- Your code looses the processor by calling *copy_from_user* (gets put to sleep), or perhaps preemption kicks in and a higher priority process runs
- Since you are still holding the lock, other processes with spin, potentially forever

The core rule is that code holding a *spinlock* **must be atomic** and must not loose the processor except to service interrupts

Concurrency and Race Conditions

The kernel preemption case is handled by the spinlock code itself

Any time kernel code holds a spinlock, preemption is disabled on the relevant processor

The bigger challenge is to **avoiding sleep** while holding a lock

Many kernel functions can sleep, and this is not always well documented

- Copying data to or from user space is an obvious example b/c of the possiblity of swapping
- Same is true of memory allocation, e.g., *kmalloc*, b/c of the possiblity that the system call sleeps until memory becomes available (this can be disabled)
- A critical section in DD code has taken the lock and an interrupt occurs, and the ISR requires the lock (fix here is to disable interrupts)

You need to examine every function that you call in the critical section to ensure 'sleeping' is not a possibility

You should also minimize the amount of time a *spinlock* is held b/c higher priority processes are locked out

Concurrency and Race Conditions

One of the alternatives:

```
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
```

spin_lock_irqsave disables interrupts (on the local processor only), with previious interrupt state stored in *flags*

You must use the *irq* version in ISRs themselves to avoid deadlock, and both the *irqsave* and *irqrestore* calls must appear in the same ISR function

There are other challenges with avoiding deadlock with *spinlocks*

In more complex code, a *spinlock* will be assigned to specific structures

If you call two functions that request the same *spinlock*, the code deadlocks -- fix is to remove the *spinlocks* from the individual functions and require the caller to acquire it (which MUST be documented!)

If you call several functions that each access different *spinlocks*, then deadlock can be avoided if all processes follow the same order to acquire/release them

On-line text describes some other, lighter weight alternative locking schemes