Thread & Synchronization

Why Talk About This Subject

- **A thread of program execution**
  - How a program start and end its execution
  - waiting for an event or a resource, delay a period, etc.
- **For concurrent operations : multiple threads of program execution**
- **How can we make this happen?**
  - support for program execution
  - sharing of resources
  - scheduling
  - communication between threads

Thread and Process

- **process:**
  - an entity to which system resources (CPU time, memory, etc.) are allocated
  - an address space with 1 or more threads executing within that address space, and the required system resources for those threads
- **thread:**
  - a sequence of control within a process and shares the resources in that process
- **lightweight process (LWP):**
  - LWP may share resources: address space, open files, …
  - clone or fork – share or not share address space, file descriptor, etc.
  - In Linux kernel, threads are implemented as standard processes (LWP) that shares certain resources with other processes, and there is no special scheduling semantics or data structures to represent threads

Why Threads

- **Advantages:**
  - the overhead for creating a thread is significantly less than that for creating a process
  - multitasking, i.e., one process serves multiple clients
  - switching between threads requires the OS to do much less work than switching between processes
- **Drawbacks:**
  - not as widely available as the process features
  - writing multithreaded programs require more careful thought
  - more difficult to debug than single threaded programs
  - for single processor machines, creating several threads in a program may not necessarily produce an increase in performance (only so many CPU cycles to be had)

POSIX Thread

- IEEE's POSIX Threads (Pthread) Model:
  - programming models for threads in a UNIX platform
  - pthreads are included in the international standards
- pthreads programming model:
  - creation of threads
  - managing thread execution
  - managing the shared resources of the process
- main thread:
  - initial thread created when main() is invoked
  - has the ability to create daughter threads
  - if the main thread returns, the process terminates even if there are running threads in that process
  - to explicitly avoid terminating the entire process, use pthread_exit()

Linux task_struct

```
/* Linux/include/linux/sched.h    */

struct task_struct {
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage;
    unsigned int flags;  /* per process flags, defined below */
    unsigned int ptrace;

    int lock_depth;      /* BKL (big kernel lock) lock depth */

    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    ……………..
    struct mm_struct *mm, *active_mm;
        struct thread_struct thread; /* CPU-specific state of this task */
        struct fs_struct *fs;    /* filesystem information */
        struct files_struct *files;  /* open file information */
```

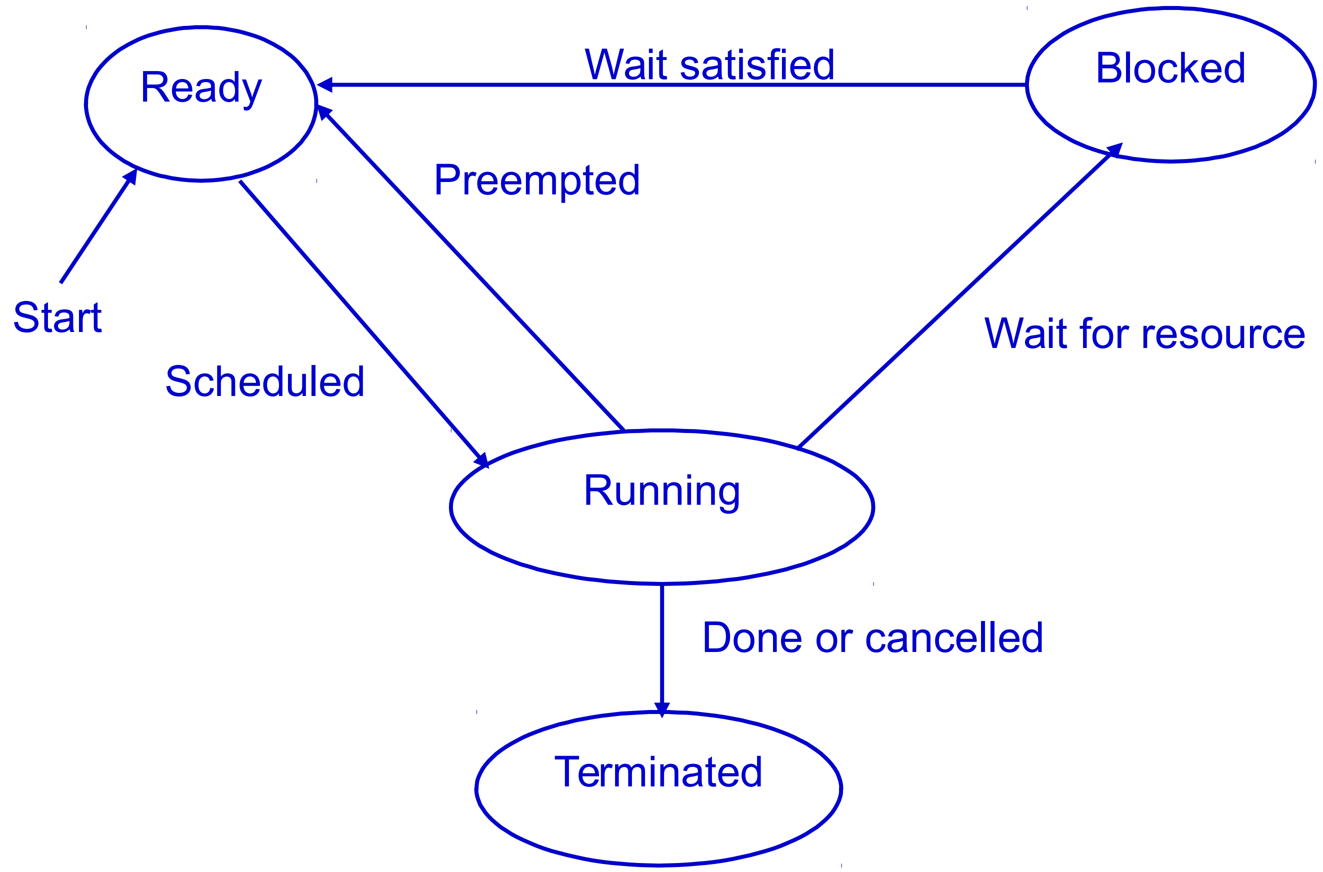
Process: *task_struct* data structure

- **state: process state**
  - TASK_RUNNING: executing
  - TASK_INTERRUPTABLE: suspended (sleeping)
  - TASK_UNINTERRUPTABLE: (no process of signals)
  - TASK_STOPPED (stopped by SIGSTOP)
  - TASK_TRACED (being monitored by other processes such as debuggers)
  - EXIT_ZOMBIE (terminated before waiting for parent)
  - EXIT_DEAD
- **thread_info: low-level information for the process**
- **mm: pointers to memory area descriptors**
- **tty: tty associated with the process**
- **fs: current directory**
- **files: pointers to file descriptors**
- **signal: signals received ………….**

Linux Processor State

```
/* This is the TSS  (task State Segment) defined by the hardware and
   saved in stack. */
struct x86_hw_tss {
    unsigned short      back_link, __blh;
    unsigned long       sp0;
    unsigned short      ss0, __ss0h;
    unsigned long       sp1;
    /* ss1 caches MSR_IA32_SYSENTER_CS: */
    unsigned short      ss1, __ss1h;
    unsigned long       sp2;
    unsigned short      ss2, __ss2h;
    unsigned long       __cr3;
    unsigned long       ip;
    unsigned long       flags;
    unsigned long       ax;
    unsigned long       cx;
    unsigned long       dx;
    unsigned long       bx;

/*  For ARM, Linux/arch/arm/include/asm/thread_info.h.,
```

Linux Thread State Transition



Pthread APIs

- **pthread_create()**
- **pthread_detach()**
- **pthread_equal()**
- **pthread_exit()**
- **pthread_join()**
- **pthread_self()**
- **pthread_cancel()**
- **pthread_mutex_init()**
- **pthread_mutex_destroy()**
- **pthread_mutex_lock()**
- **pthread_mutex_trylock()**
- **pthread_mutex_unlock()**
- **sched_yield()**

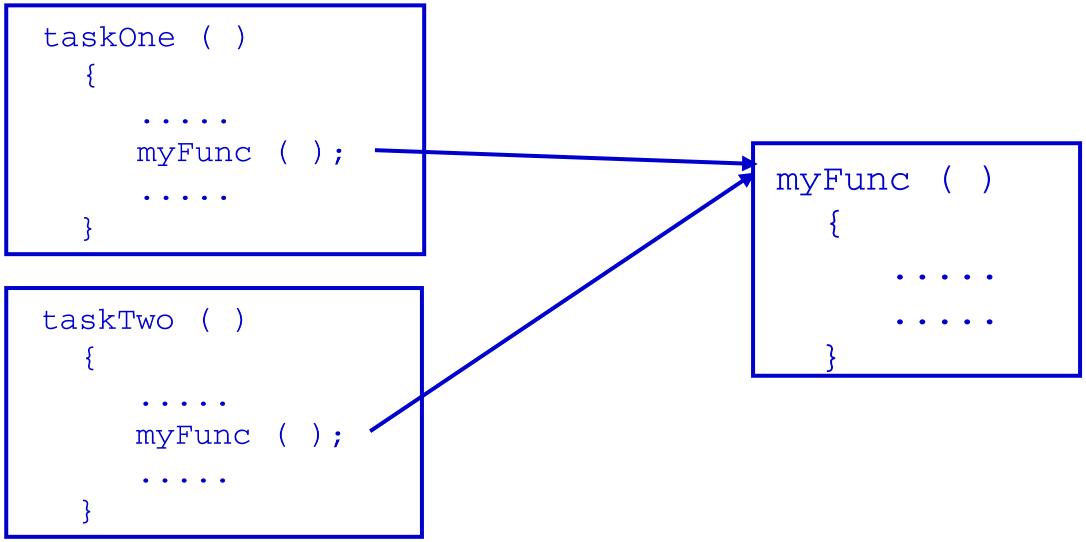
```
int pthread_create(
       pthread_t *tid,              // Thread ID returned by the system
       const pthread_attr_t  *attr,    // optional creation attributes
       void *(*start)(void *),     // start function of the new thread
       void *arg               // Arguments to start function
   );
```

Example of Thread Creation

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

void *thread_routine(void* arg){
          printf("Inside newly created thread \n");
}

void main()
{
    pthread_t thread_id;      // threat handle
    void *thread_result;

    pthread_create(&thread_id, NULL,
                   thread_routine, NULL );

  printf("Inside main thread \n");
  pthread_join(thread_id,  &thread_result);
}
```

Shared Code and Reentrancy

- **A single copy of code is invoked by different concurrent tasks must reentrant**
  - pure code
  - variables in task stack (parameters)
  - guarded global and static variables (with semaphore or taskLock)
  - variables in task content (taskVarAdd)



Thread Synchronization: Mutex

- **Mutual exclusion (mutex):**
  - guard against multiple threads modifying the same shared data simultaneously
  - provides locking/unlocking critical code sections where shared data is modified
- **Basic Mutex Functions:**

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

  - data type named pthread_mutex_t is designated for mutexes
  - the attribute of a mutex can be controlled by using the pthread_mutex_init() function

Example: Mutex

```
#include <pthread.h>
...
pthread_mutex_t my_mutex;         // should be of global scope
...
int main()
{
     int tmp;
     …
     tmp = pthread_mutex_init( &my_mutex, NULL ); // initialize the mutex
     ...
     // create threads
     ...
     pthread_mutex_lock( &my_mutex );
             do_something_private();
     pthread_mutex_unlock( &my_mutex );
     ...
     return 0;
}
```

Thread Synchronization: Semaphore

- **creating a semaphore:**

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

  - initializes a semaphore object pointed to by sem
  - pshared is a sharing option; a value of 0 means the semaphore is local to the calling process
  - gives an initial value to the semaphore

- **terminating a semaphore:**

  `int sem_destroy(sem_t *sem);`

- **semaphore control:**

  `int sem_post(sem_t *sem);`

  `int sem_wait(sem_t *sem);`

  - sem_post atomically increases the value of a semaphore by 1,
  - sem_wait atomically decreases the value of a semaphore by 1; but always waits until the semaphore has a non-zero value first

Example: Semaphore

```
#include <pthread.h>
#include <semaphore.h>
void *thread_function( void *arg )
{
   sem_wait( &semaphore ); perform_task(); pthread_exit( NULL );
}
sem_t semaphore; // also a global variable just like mutexes
int main()
{
    int tmp = sem_init( &semaphore, 0, 0 );  // initialize the semaphore
    pthread_create( &thread[i], NULL, thread_function, NULL );
    while (still_has_something_to_do()) {
        sem_post( &semaphore );
        …
    }
    pthread_join(thread[i], NULL);
    sem_destroy(&semaphore);
    Return 0;
}
```

Condition Variables

- **A variable of type** *pthread_cond_t*
- **Use condition variables to atomically block threads until a particular condition is true.**
- **Always use condition variables together with a mutex lock.**

```
pthread_mutex_lock();
      While (condition_is_false)
         pthread_cond_wait();
pthread_mutex_unlock();
```

- **Use** *pthread_cond_wait()* **to atomically release the mutex and to cause the calling thread to block on the condition variable**
- **The blocked thread can be awakened by** *pthread_cond_signal(), pthread_cond_broadcast(),* **or when interrupted by delivery of a signal.**

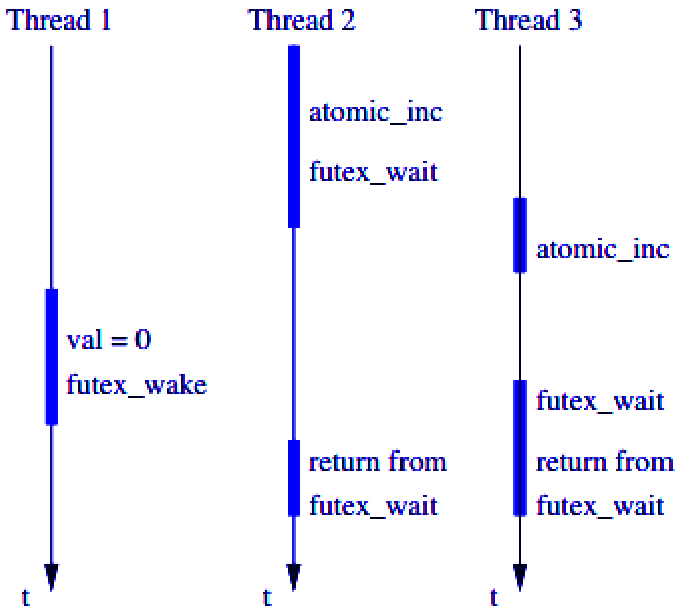
Mutex in Linux

- **Two states: locked and unlocked.**
  - if locked, wait until it is unlocked
  - only the thread that locked the mutex may unlock it
- **Various implementations for performance/function tradeoffs**
  - Speed or correctness (deadlock detection)
  - lock the same mutex multiple times
  - priority-based and priority inversion
  - forget to unlock or terminate unexpectedly
- **Available types**
  - normal
  - fast
  - error checking
  - recursive: owner can lock multiple times (couting)
  - robust: return an error code when crashes while holding a lock
  - RT: priority inheritance

Pthread Futex

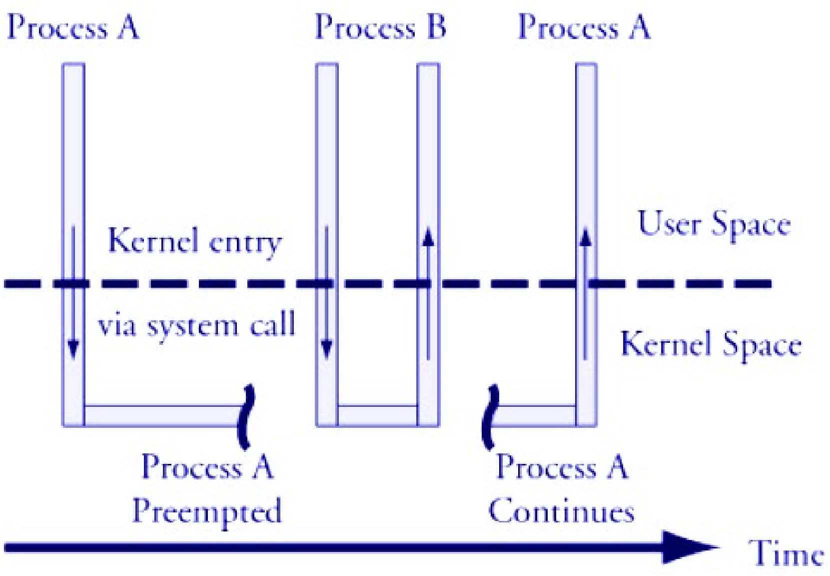
- **Fast mutex: Lightweight and scalable**
- **In the noncontended case can be acquired/released from userspace without having to enter the kernel.**
  - lock is a user-space address, e.g. a 32-bit lock variable field.
  - “uncontended” and “waiter-pending”
  - kernel provides futex queue, and sys_futex system call
  - invoke sys_futex only when there is a need to use futex queue
  - need atomic operations in user space
  - race condition: atomic update of *ulock* and system call are not atomic

```
typedef struct ulock_t {
    long status;
} ulock_t;
```



Synchronization in Linux Kernel

- **The old Linux system ran all system services to completion or till they blocked (waiting for IO).**
  - When it was expanded to SMP, a lock was put on the kernel code to prevent more than one CPU at a time in the kernel.
- **Kernel preemption**
  - a process running in kernel mode can be replaced by another process while in the middle of a kernel function
  - In the example, process B may be waked up by a timer and with higher priority
  - Why – dispatch latency



(Christopher Hallinan,"Embedded Linux Primer: A Practical Real-World Approach".)

Linux Kernel Thread

- **A way to implement background tasks inside the kernel**

```
static struct task_struct *tsk;
static int thread_function(void *data) {
    int time_count = 0;
    do {
        printk(KERN_INFO "thread_function: %d times", ++time_count);
        msleep(1000);
        }while(!kthread_should_stop() && time_count<=30);
    return time_count;
}

static int hello_init(void)  {
    tsk = kthread_run(thread_function, NULL, "mythread%d", 1);
    if (IS_ERR(tsk)) {  …. }
}
```

WorkQueues

- **To request that a function be called at some future time.**
  - tasklets execute quickly, for a short period of time, and in atomic mode
  - workqueue functions may have higher latency but need not be atomic
- **Run in the context of a special kernel process (worker thread)**
  - more flexibility and workqueue functions can sleep.
  - they are allowed to block (unlike deferred routines)
  - No access to user space
- ***A workqueue (*** *workqueue_struct)* ***must be explicitly created***
- **Each workqueue has one or more dedicated "kernel threads", which run functions submitted to the queue via *queue_work()*.**
  - work_struct structure to submit a task to a workqueue

    *DECLARE_WORK(name, void (*function)(void *), void *data);*
- **The kernel offers a predefined work queue called *events*, which can be freely used by every kernel developer**

Example of Work Structure and Handler

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/workqueue.h>
MODULE_LICENSE("GPL");

static struct workqueue_struct *my_wq; // work queue
typedef struct {                       // work
    struct work_struct  my_work;
    int x;
} my_work_t;

my_work_t  *work, *work2;

static void my_wq_function( struct work_struct *work)  // function to be call
{
    my_work_t  *my_work = (my_work_t *)work;
    printk( "my_work.x %d\n", my_work->x );
    kfree( (void *)work );
    return;
}
```

(http://www.ibm.com/developerworks/linux/library/l-tasklets/index.html)

Example: Work and WorkQueue Creation

```
int init_module( void )
{
     int ret;
     my_wq = create_workqueue("my_queue"); // create work queue
     if (my_wq) {
         work = (my_work_t *)kmalloc(sizeof(my_work_t), GFP_KERNEL);
         if (work) {                 // Queue work (item 1)
         INIT_WORK( (struct work_struct *)work, my_wq_function );
         work->x = 1;
         ret = queue_work( my_wq, (struct work_struct *)work );
         }

         work2 = (my_work_t *)kmalloc(sizeof(my_work_t), GFP_KERNEL);
         if (work2) {         // Queue work (item 2)
         INIT_WORK( (struct work_struct *)work2, my_wq_function );
         work2->x = 2;
         ret = queue_work( my_wq, (struct work_struct *)work2 );
         }
     }
     return 0;
}
```

(http://www.ibm.com/developerworks/linux/library/l-tasklets/index.html)

When Synchronization in Necessary

- **A race condition can occur when the outcome of a computation depends on how two or more interleaved kernel control paths are nested**
- **To identify and protect the critical regions in exception handlers, interrupt handlers, deferrable functions, and kernel threads**
  - On single CPU, critical region can be implemented by disabling interrupts while accessing shared data
  - If the same data is shared only by the service routines of system calls, critical region can be implemented by disabling kernel preemption (interrupt is allowed) while accessing shared data
- **How about multiprocessor systems (SMP)**
  - Different synchronization techniques are necessary for data to be accessed by multiple CPUs
- **Note that interrupts can be nested, but they are non-blocking, not preempted by system calls.**

Atomic Operations

- **Atomic operations provide instructions that are**
  - executable atomically;
  - without interruption
  - Not possible for two atomic operations by a single CPU to occur concurrently
- **Atomic 80x86 instructions**
  - Instructions that make zero or one aligned memory access
  - Read-modify-write instructions (inc or dec)
  - Read-modify-write instructions whose opcode is prefixed by the lock byte (0xf0)
- **In RISC, load-link/store conditional (ldrex/strex)**
  - store can succeed only if no updates have occurred to that location since the load-link.
- **Linux kernel**
  - two sets of interfaces for atomic operations: one for integers and another for individual bits

Linux Atomic Operations

- **Uses atomic_t data type**
- **Atomic operations on integer counter in Linux**

Function	Description
atomic_read(v)	Return *v
atomic_set(v,i)	set *v to i
atomic_add(i,v)	add i to *v
atomic_sub(i,v)	subtract i from *v
atomic_sub_and_test(i,v)	subtract i from *v and return 1 if result is 0
atomic_inc(v)	add 1 to *v
atomic_dec(v)	subtract 1 from *v
atomic_dec_and_test(v)	subtract 1 from *v and return 1 if result is 0
atomic_inc_and_test(v)	add 1 to *v and return 1 if result is 0
atomic_add_negative(i,v)	add i to *v and return 1 if result is negative

- **A counter to be incremented by multiple threads**
- **Atomic operate at the bit level, such as**

```
unsigned long word = 0;
set_bit(0, &word); /* bit zero is now set (atomically) */
```

Spinlock

- **Ensuring mutual exclusion using a busy-wait lock.**
  - if the lock is available, it is taken, the mutually-exclusive action is performed, and then the lock is released.
  - If the lock is not available, the thread busy-waits on the lock until it is available.
  - it keeps spinning, thus wasting the processor time
  - If the waiting duration is short, faster than putting the thread to sleep and then waking it up later when the lock is available.
  - really only useful in SMP systems
- **Spinlock with local CPU interrupt disable**

```
spin_lock_irqsave(&my_spinlock, flags);
  /* critical section */
spin_unlock_irqrestore(&my_spinlock, flags);
```

- **Reader/writer spinlock – allows multiple readers with no writer**

Semaphore

- **Kernel semaphores**
  - struct semaphore: count, wait queue, and number of sleepers

```
void sem_init(struct semaphore *sem, int val);
// Initialize a semaphore's counter sem->count to given value

inline void down(struct semaphore *sem);
//try to lock the critical section by decreasing sem->count

inline void up(struct semaphore *sem); // release the semaphore
```

- **blocked thread can be in TASK_UNINTERRUPTIBLE or TASK_INTERRUPTIBLE (by timer or signal)**
- **Special case – mutexes (binary semaphores)**

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

- **Read/Write semaphores**

Spinlock vs Semaphore

- **Only a spinlock can be used in interrupt context,**
- **Only a semaphore can be held while a task sleeps.**

Requirement	Recommended Lock
Low overhead locking	Spinlock
Short lock hold time	Spinlock
Long lock hold time	Semaphore
Need to lock from interrupt context	Spinlock
Need to sleep while holding lock	Semaphore

- **Other mechanisms:**
  - Completion: synchronization among multiprocessors
  - The global kernel lock (a.k.a big kernel lock, or BKL)
    - *Lock_kernel(), unlock_kernel()*
  - RCU – read-copy update, for mostly-read access

Blocking Mechanism in Linux Kernel

- **ISR can wake up a block kernel thread**
  - which is waiting for the arrival of an event
- **Wait queue**
- **Wait_for_completion_timeout**
  - specify “completion” condition, timeout period, and action at timeout
  - “complete” to wake up thread in wait queue
  - wake-one or wake-many

```
struct semaphore {
    raw_spinlock_t lock;
    unsigned int count;
    struct list_head wait_list;
};
```

```
struct completion {
    unsigned int done;
    wait_queue_head_t wait;
};
```

```
struct __wait_queue_head {
    spinlock_t lock;
    struct list_head task_list;
};
```

Wait_for_Completion Example

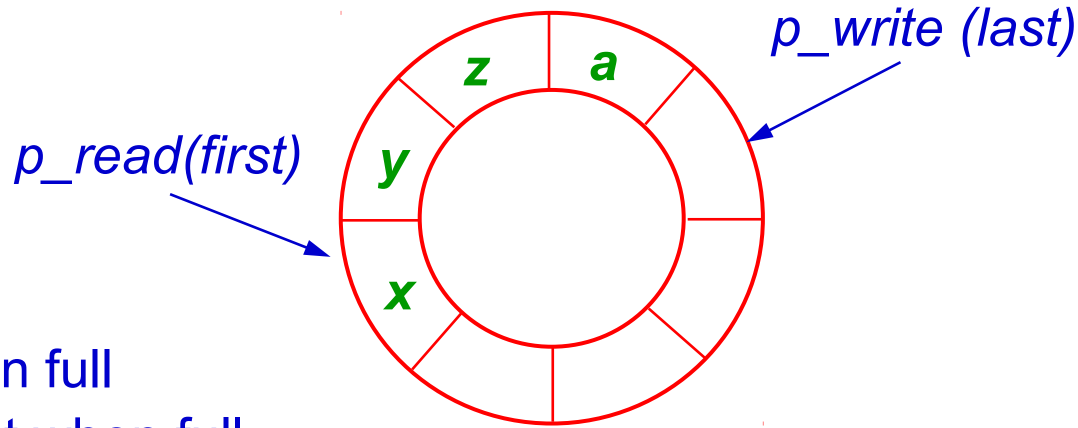
- In i2c-designware-core.c
- Threads call `i2c_dw_xfer` will do
  - `INIT_COMPLETION(dev->cmd_complete);`
  - `i2c_dw_xfer_init(dev);`
  - `ret = wait_for_completion_interruptible_timeout(&dev->cmd_complete, HZ);`
- In *i2c_dw_xfer_init,* interrupt get enabled
- In *i2c_dw_isr,* when xfer is done
  - `complete(&dev->cmd_complete);`

Reader/Writer: ISR and Buffering

- **Input: single producer (ISR) and single consumer (thread)**
- **If a read is initialed by the thread**
  - calls "*read*" with a buffer of *n* bytes
  - initiate IO operation, enable interrupt
  - ISR reads input and store in the buffer.
  - If done, single the completion
- **Blocking or nonblocking**
  - in thread context (vxWorks): semaphore, lock
  - in kernel context (Linux): wait queue
- **Guarded access**
  - Lock (mutex) and interrupt lock (disable)

Ring Buffer

- if `p_read == p_write`, empty
  if `(p_write + 1) % size == p_read`, full
- Invariant: *p_write* never incremented up to *p_read*
- Thread safe if memory accesses are ordered
  - no write concurrency



- **Queue operation**
  - New data is lost when full
  - overwrite old element when full
- **Multiple consumers & producers**

Thread Safe Producer Consumer Queue

Writing elements

```
bool WriteElement(Type &Element)
{
    int next = (p_Write + 1) % Size;
    if(next != p_Read)
    {
        Data[p_Write] = Element;
        p_Write = next;
        return true;
    }
    return false;
}
```

Reading elements

```
bool ReadElement(Type &Element)
{
    if(p_Read == p_Write)
        return false;

    int next= (p_Read + 1) % Size;
    Element = Data[p_Read];
    p_Read = next;
    return true;
}
```