Bern University of Applied Sciences
Engineering and Information Technology

Module I-7410 Advanced Linux

Kernel Thread Management

By Franz Meyer
Version 1.2
March 2013

Introduction to Linux Kernel Threads (1)

Agenda

- Kernel Thread API
- Kernel Thread Example
- Race Conditions
- Concurrency Management
- Atomic Variables
- Semaphores and Mutexes
- Wait Queues
- Completions
- Spinlocks

Introduction to Linux Kernel Threads (2)

- Kernel threads are similar to POSIX style user threads but they only exist in the kernel
- You create a kernel thread in a kernel loadable module
- Internally the Linux kernel uses a **`task_struct`** to represent a process, a user thread, and a kernel thread.
- All kernel threads are decendents of the kernel thread **`kthreadd`**
- The code of a kernel thread is given (as in a user thread) by the thread function:

```
int threadfn(void *data)

data: pointer to a data package passed to that
      thread
      NULL if no data are passed
```

Introduction to Linux Kernel Threads (3)

- More kernel thread API functions: **`kthread_run()`** usually called in the module init function (from **`<linux/kthread.h>`**)

```
/**
 * kthread_run - create and wake a thread.
 * @threadfn: the function to run until signal_pending
 * @data: data ptr for @threadfn.
 * @fmt:  printf-style name for the thread.
 * Description: Convenient wrapper for kthread_create()
 */
#define kthread_run(threadfn, data, fmt, ...) \
({                                            \
   struct task_struct *__k                    \
     = kthread_create(threadfn, data, fmt, ## __VA_ARGS__); \
    if (!IS_ERR(__k))                         \
                wake_up_process(__k);         \
    __k;                                      \
})
```

Introduction to Linux Kernel Threads (4)

- More kernel thread API functions (from **<linux/kthread.h>**):
  - **kthread_create:** do not use directly, use wrapper

```
struct task_struct *kthread_create(
   int (*threadfn)(void *data),/*the thread function*/
   void data,                  /*passed to the thr fct*/
   const char fmt[],...)       /*variable arguments*/
        __attribute__((format(printf, 3, 4)));
```

  - **kthread_stop:** stop a thread given by a ptr to **task_struct *ts**
    usually called in module exit function

```
int kthread_stop(struct task_struct *ts)
```

  - **kthread_should_stop:** returns true if the calling thread should stop (return); usually called in the task function

```
int kthread_should_stop(void)
```

Introduction to Linux Kernel Threads (5)

- A kernel thread example: Create a simple kernel thread when the module is loaded, destroy it when the module is unloaded. The thread periodically writes on the console and also in the syslog.
  - The kernel thread function: **kthread_fct()**

```
#define DPRINT(fmt,args...) \    /*debug macro*/
        printk(KERN_INFO "%s,%i:" fmt "\n", \
        __FUNCTION__, __LINE__,##args);

int kthread_fct(void *data)
{
         while(1) {
                 DPRINT("in kernel thread");
                 mdelay(500); msleep(1);
                 if (kthread_should_stop()) return 0;
         }
}
```

Introduction to Linux Kernel Threads (6)

- A kernel thread example, cont.
  - The module init/exit function: **kthr_init/exit:**

```
static struct task_struct *ts;

int __init kthr_init(void)
{
        DPRINT("init_module() called\n");
        ts=kthread_run(kthread_fct,NULL,"kthread");
        return 0;
}

void __exit kthr_exit(void)
{
        DPRINT("exit_module() called\n");
        kthread_stop(ts);
}
```

Introduction to Linux Kernel Threads (7)

- Kernel Thread Lab-1
  - Exercise 1
    Write a module with the example code given on the previous slides.
    Put the code in a file called **`kthread0.c`** and write an appropriate **`make`** file.
    Load the module using the **`insmod`** or **`modprobe`** program. Then find the module in the **`ps`** list. Which **`ps`** option(s) do you need to specify to see thread info?
  - Exercise 2
    Study the kernel thread related code in the Linux kernel source. How do the functions like **`kthread_create()`**, **`kthread_stop()`**, **`kthread_should_stop()`** etc work? What does the kernel thread daemon (kthreadd) do?
    Hint: The kernel thread stuff is stored in two files: **`<linux/kthread.h>`** and **`/usr/src/linux<-ver>/kernel/kthread.c`**.

Race Conditions (1)

- A race condition occurs if several threads access uncontrolled to shared data. Without protection (locking) a race condition produces unexpected or incorrect results.
- Examples of race conditions
  - Increment/decrement of a shared counter: on most processors **`count++`** or **`count--`** result in three machine instructions: (1) load count (2) inc/dec count (3) store count. After a sequence of in **`count++`** in thread T1 and **`count--`** in thread T2 the variable count should have the same value as before. Because of race conditions, however, the value could differ by one.
    Counter-measure: atomic operations
  - Lazy memory allocation of shared buffers: A memory buffer is only allocated when it is used. If two threads see a memory buffer pointer of NULL and will decide to allocate memory. Without protection, the assignment of the second thread will modify the buffer pointer of the first thread. Memory leakages is the result.
    Counter-measure: locking

Concurrency Management (1)

- A modern Linux kernel is preemptible. A kernel thread can lose the CPU at any time.
- Device interrupts are asynchronous events that can cause concurrent execution of kernel code.
- Interrupts are often handled in two phases: fast and slow (deferred) handling. If shared data is modified you will need to protect it
- Several modern devices are hot-pluggable: A device can disappear while you working with it
- First rules of thumb: Avoid shared data structure whenever possible. If not keep sharing on a minimum. Avoid global variables.
- Of course, sharing is often required. In this case use the technique of **critical sections** for access management. Recall that critical sections include **mutual exclusion** and **fair access**.

Concurrency Management: Atomic Variables (1)

- Often a shared resource is a simple variable like an integer value
- Recall that a simple operation like **`i++`** would require locking.
- Some processors might execute simple arithmetic operation atomically, but you cannot count on it.
- Thus the architecture-dependent part of the Linux kernel provides a bunch of macros that produce the correct code for many processors.
- An **`atomic_t`** holds an **`int`** value (32-bit) on all supported architectures. (**`atomic64_t`** are available on 64 bit machines)
- The full integer range may not be available but you can count on 24 bits.
- The macros are SMP safe and written to generate the fastest code that the architecture allows.

Concurrency Management: Atomic Variables (2)

- Some atomic operations (defined in **<asm.atomic>**)

```
void atomic_set(atomic_t *v,int i);set v to value I
Initializer Macro: ATOMIC_INIT(val);
int atomic_read(atomic_t *v);       return *v
void atomic_inc(atomic_t *v);       atomic *v++
void atomic_dec(atomic_t *v) ;      atomic *v--
void atomic_add(int i,atomic_t *v); atomic *v += i
void atomic_sub(int i,atomic_t *v); atomic *v -= i
int  atomic_inc_and_test(atomic_t *v);perform operation then
int  atomic_dec_and_test(atomic_t *v); test result,if (0)
int  atomic_sub_and_test(int i,atomic_t *v); (*v-=i)==0?
int  atomic_add_negative(int i,atomic_t *v); (*v+=i)<0?
```

Concurrency Management: Atomic Variables (3)

- Some atomic operations, cont (defined in **<asm.atomic>**)
  - It is strongly recommended to manipulate data items through above macros
  - Operations in **atomic_t** variables work only when the quantity is atomic. Operations across multiple atomic functions are not atomic.

    Consider the typical banking account code: from checking to saving

```
atomic_sub(amount, &checking_balance);
atomic_add(amount, &saving_balance);
```

  - Both operations are indeed atomic but it is possible that code runs between those two operations that could create trouble. Some other form of locking must be used.

Concurrency Management: Atomic Bit Operations (1)

- Some atomic bit operations (defined in **<asm.bitops>**)
  - Atomic bit operations are safe also on SMP machines

```
void set_bit(int i,unsigned long *addr); set bit #i in *addr
void clear_bit(int i,unsigned long *addr); clear bit #i
void clear_bit(int i,unsigned long *addr); toggle bit #i
int test_bit(int i,unsigned long *addr);   return bit #i
```

The following operations atomically perform the requested operation and also return the previous value of the bit number i

```
int test_and_set_bit(int i,unsigned long *addr);
int test_and_clear_bit(int i,unsigned long *addr);
int test_and_change_bit(int i,unsigned long *addr);
```

Concurrency Management: Atomic Bit Operations (2)

- A very simple form of a spin lock used to access a shared data item in a critical section may look like this:

```
#include <asm/bitops.>
int lock = 0;
/*entry code: try to acquire the lock*/
while (test_and_set_bit(0, &lock) != 0)
   wait_a_while();

/* in critical section: do your critical work */

/*exit code: release the lock, and check */
if (test_and_clear_bit(0, &lock) == 0)
   report_error();
```

Concurrency Management: Semaphores (1)

- The critical section problem
  - A critical section is a segment of code in which a kernel thread performs an operation on a shared object (common variable, data structure, etc). No other kernel thread may execute in its critical section at this time (mutual exclusion). It means that those other kernel threads must wait. When the kernel thread leaves its critical section another kernel thread may enter its critical section. The fairness condition says that all kernel threads must have a chance to enter its critical section. In other words the waiting time of a kernel thread to enter its critical section must be bounded.
- A semaphore is an object that consists of (1) a counter (2) a queue of waiting tasks. Besides the constructor and destructor two operations are defined:
  - down operation (corresponds with the classic P operation):
    - Atomically decrement the counter
    - If the counter becomes zero then (1) block the calling kernel thread (2) insert its task structure to the queue of this semaphore, and (3) schedule another task

Concurrency Management: Semaphores (2)

- Semaphore Operations, cont
  - up operation (corresponds with the classic V operation):
    - If the semaphore queue is not empty, pick a task and make it runnable
    - Otherwise, increment the counter
- Semaphores defined like this can be used as general synchronization object
  - To solve the critical section problem, use a semaphore in the following way
    - Create semaphore with its counter initially set to **one**
    - Call the **down** operation at entry to the critical section
    - Call the **up** operation at exit of the critical section
  - To solve the task A precedes task B problem (task B waits for task A), use a semaphore in the following way
    - Create semaphore with its counter initially set to **zero**
    - Call the **down** operation at the beginning of task B
    - Call the **up** operation at the end of task A

Concurrency Management: Semaphores (3)

- Semaphores in Linux are used to protect critical sections. A semaphore used in this mode is sometimes called a binary semaphore or a **`mutex`**.
- Although the concept of the semaphore is well understood it is dangerous in practice. The most frequent errors are:
  - A developer mixes up down/up at entry/exit of a critical section.
  - The up operation is not execution when a critical section is left.
  - If there are several critical sections in a module it is possible that a developer executes an operation on the wrong semaphore. That produces synchronization errors that are hard to find and debug.
- To overcome the weakness of a semaphore a **`mutex`** was introduced into the Linux kernel. A **`mutex`** is different but similar to a semaphore. It is a sort of “enhanced mutex semaphore” with a more strict semantics. For example, a task acquires the **`mutex`** and becomes its **owner**. Only the owner may release the **`mutex`**. Recursive locking is not permitted because it may be deadlock prone.

Concurrency Management: Semaphores (4)

- Semaphore implementation in Linux
  - A semaphore is represented by a **`struct semaphore`** (<linux/semaphore.h>)

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

  - There are useful macros/functions to create or initialize a semaphore

```
DECLARE_MUTEX(name) –   define semaphore given by name
                        with initial value set to 1
void sema_init(struct semaphore *s, int val);
init_MUTEX(name)            – same as sema_init(name,1)
init_MUTEX_LOCKED(name)     – same as sema_init(name,0)
```

Concurrency Management: Semaphores (5)

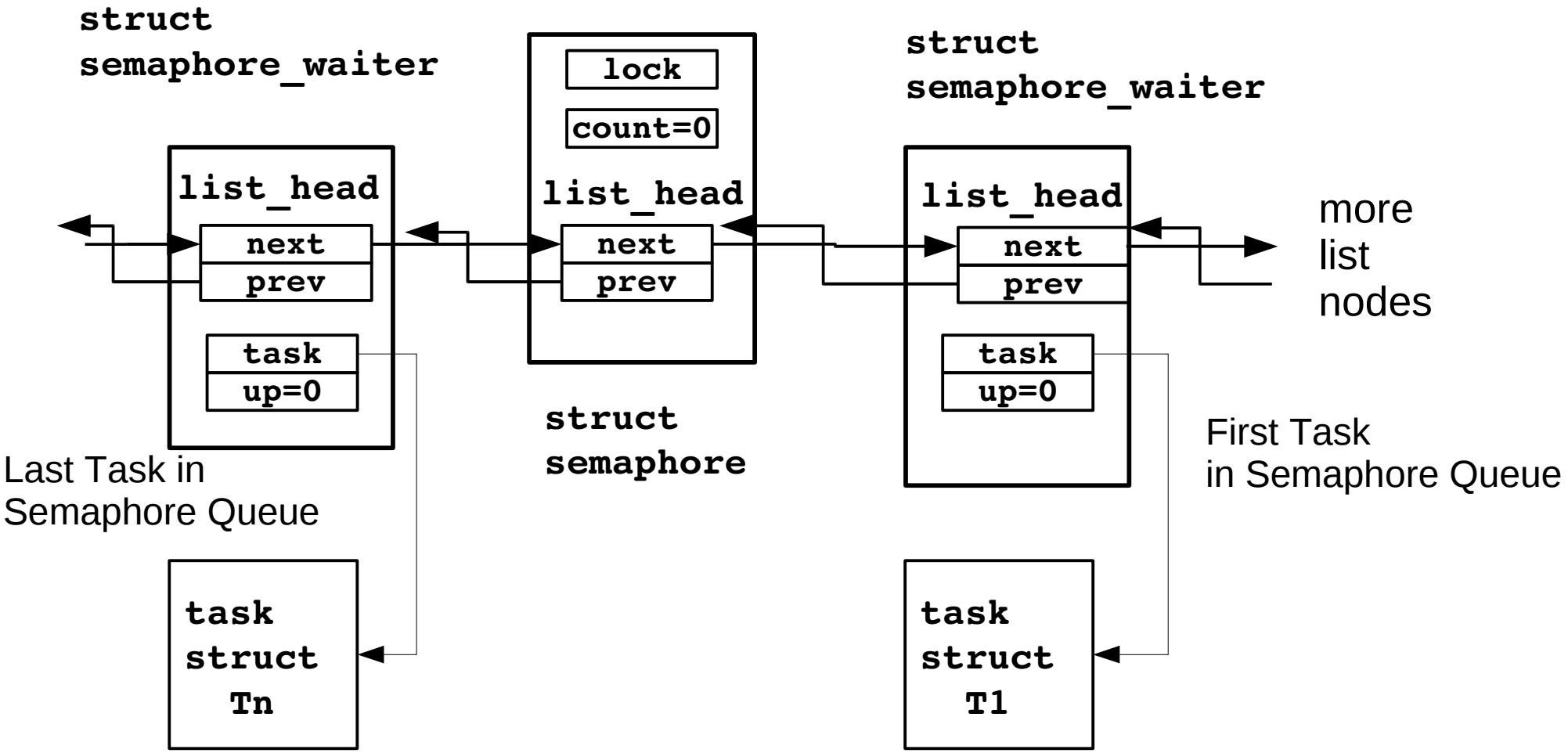
- Semaphore implementation in Linux, cont.
  - The down operation: **`void down(struct semaphore *sem)`**

    (1) get the semaphore spin lock
    (2) check if **`sem->count`** is positive, if so decrement is and leave
    (3) otherwise (**`sem->count = 0`**) create a waiter structure and append it at the semaphore queue given by **`sem->wait_list`**. The calling thread enters **uninterruptible** sleep (does not react on signals).
  - The operation **`int down_interruptible(struct semaphore *sem)`** enters **interruptible** sleep (honors signals).
  - A node in the waiting queue of a semaphore is represented by a **`struct semaphore_waiter`** that looks like this

```
struct semaphore_waiter {
   struct list_head    list;
   struct task_struct *task;
   int up; /*flag:=0-> task is sleeping*/
};         /*     #0-> task should wakeup*/
```

Concurrency Management: Semaphores (6)

- Example of a semaphore with n waiting tasks



Concurrency Management: Semaphores (7)

- Semaphore implementation in Linux, cont.
  - The up operation: **`void up(struct semaphore *sem)`**

    (1) get the semaphore spin lock
    (2) check if the semaphore queue given by **`sem->wait_list`** is empty, if so increment **`sem->count`** and leave
    (3) otherwise (**`sem->count = 0`**) remove the first waiter structure from the semaphore queue given by **`sem->wait_list`**, and wake up the task specified in the waiter structure
- Semaphores and Mutual Exclusion
  - Protecting a critical section with a semaphore mutex needs three steps
    1. Declare the semaphore: **`DECLARE_MUTEX(mutex)`**
    2. Critical section entry: **`down(&mutex)`**
    3. Critical section exit: **`up(&mutex)`**
  - Do not mix up semaphores with mutexes treated later.

Concurrency Management: Semaphores (8)

- Lab-2 (Semaphores): Synchronize kernel threads with semaphores

  Write a kernel module that contains two kernel threads called **`ping`** and **`pong`**. Synchronize those two threads using semaphores so that they alternatively write messages in the syslog (/var/log/syslog) using the **`printk()`** function. The kernel thread **`ping`** should begin. Prevent the kernel threads from writing too many log messages. You may want to use kernel functions like **`mdelay()`** or **`msleep()`**. Find out the difference.

  When you have loaded the module then use the **`ps`** utility and find the kernel threads in its output.

  When you unload the module the kernel threads should stop. Verify that by using the **`ps`** utility. Analyze the log messages displayed by the **`dmesg`** utility. Especially, check the time stamps of the messages.

Concurrency Management: Mutexes (1)

- Mutexes are similar to semaphores with some improvements
  - A mutex is associated with an owner: The thread that acquires the mutex becomes its owner. Only one task is allowed to hold a mutex
  - Only the owner is allowed to unlock (rlease) the mutex.
  - Multiple and recursive locking is not allowed (deadlock prevention)
- Mutex API (**<linux/mutex.h>**)
  - Mutex structure:

```
struct mutex {
        atomic_t              count;
        spinlock_t            wait_lock;
        struct list_head      wait_list;
        struct thread_info *owner;
};
```

  - Define a mutex: DEFINE_MUTEX(mutex_name);
  - Lock a mutex: `void mutex_lock(struct mutex_lock *lock);`
  - Unlock a mutex: `void mutex_unlock(struct mutex_lock *lock);`

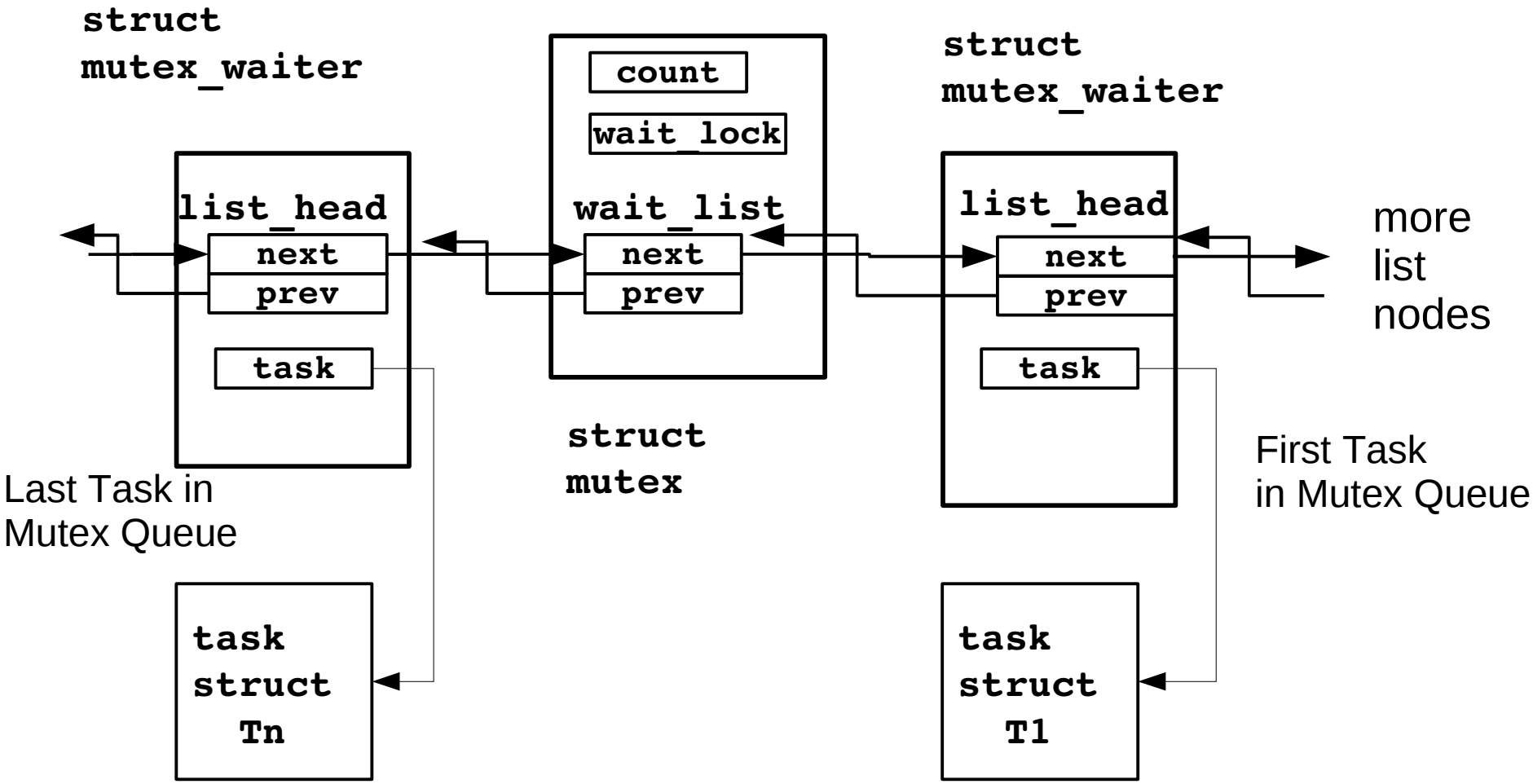
Concurrency Management: Mutexes (2)

- Mutex implementation in Linux
  - More on strict mutex semantics
    - A mutex must be initilized by calling the DEFINE_MUTEX macro
    - A task must not exit when holding a mutex
    - A mutex held by a task must not be re-initialized
    - A mutex must not be used in a hardware or software interrupt context
  - The **`mutex_lock()`** function is based on the atomic exchange (swap) instruction between the mutex count and zero. (count is rather a lock state variable than a counter). When the variable count changes from one to zero then the calling task becomes the owner of the lock. Otherwise, the calling task is queued in the per mutex waiting list.
  - A task in the waiting list is kept in a

```
struct mutex_waiter {
        struct list_head    list;
        struct task_struct *task;
};
```

Concurrency Management: Mutexes (3)

- Mutex Implementation: Example of a mutex with n waiting tasks



Concurrency Management: Mutexes (4)

- Mutex implementation, cont.
  - There are variants of the mutex_lock function
    - **`mutex_lock_interruptible()`**
      Works like **`mutex_lock()`** but in case the calling thread is blocked is blocking state is WAIT_INTERRUTIBLE. It means that a signal for a blocked thread unblocks it. The function returns -EINTR.
    - **`mutex_lock_killable()`**
      Similar to **`mutex_lock()`** but in case the calling thread is blocked is blocking state is WAIT_KILLABLE.
  - The function **`mutex_try_lock()`** tries to acquire the lock but does not block if the lock has been taken by another task. It returns one if the mutex lock has been successfully acquired, and zero otherwise.
  - The function **`mutex_unlock()`** releases a mutex lock previously acquired. In the current implementation it is **not** checked if the caller is the owner of the lock.

Concurrency Management: Mutexes (5)

- Lab-3 (Mutexes): Synchronize kernel threads with mutexes

  Write a kernel module that creates a mutex and a given number of instances of a kernel thread. The kernel thread goes repeatedly through a critical section protected by the mutex. It executes a loop with the following actions:

  (1) Acquire the mutex lock → enter the critical section.

  (2) Burn some CPU cycles then decide to relinquish the CPU in the critical section using the **`schedule()`** function. This is necessary on a single CPU machine so that another kernel thread happens to wait. On a SMP machine this may not be necessary because the kernel threads are progressing in parallel on different CPUs.

  (3) Release the mutex lock → exit the critical section.

  (4) Burn some CPU cycles in the non-critical section.

  You may want to use kernel functions like **`mdelay()`** or **`msleep()`**.

  In principle, you need not load the kernel module permanently. Do your work in the init routine of the module and return from it with an error. In this case the kernel module is unloaded and the **`modprobe`** utility reports an error to ignore.

Concurrency Management: Wait Queues (1)

- If a task (user or kernel) waits for an event like a hardware operation, a data transfer, a message from another task, etc, it is blocked.
- In Linux a blocked process is waiting in a **`wait_queue`**. A task in a blocked state is not selected by the CPU scheduler. When the event occurs the task unblocks: Its state changes to runnable. Now the task is eligible by the CPU scheduler to run.
- A blocked task is expecting that some condition will become true
- In Linux a wait queue is managed by a wait queue head and wait queue nodes.

Concurrency Management: Wait Queues (2)

- The wait queue API (defined in **<linux/wait>**)
  - Define a wait queue by means of a wait queue header
  - Statically

```
DECLARE_WAIT_QUEUE_HEAD(name);
```

  - Dynamically

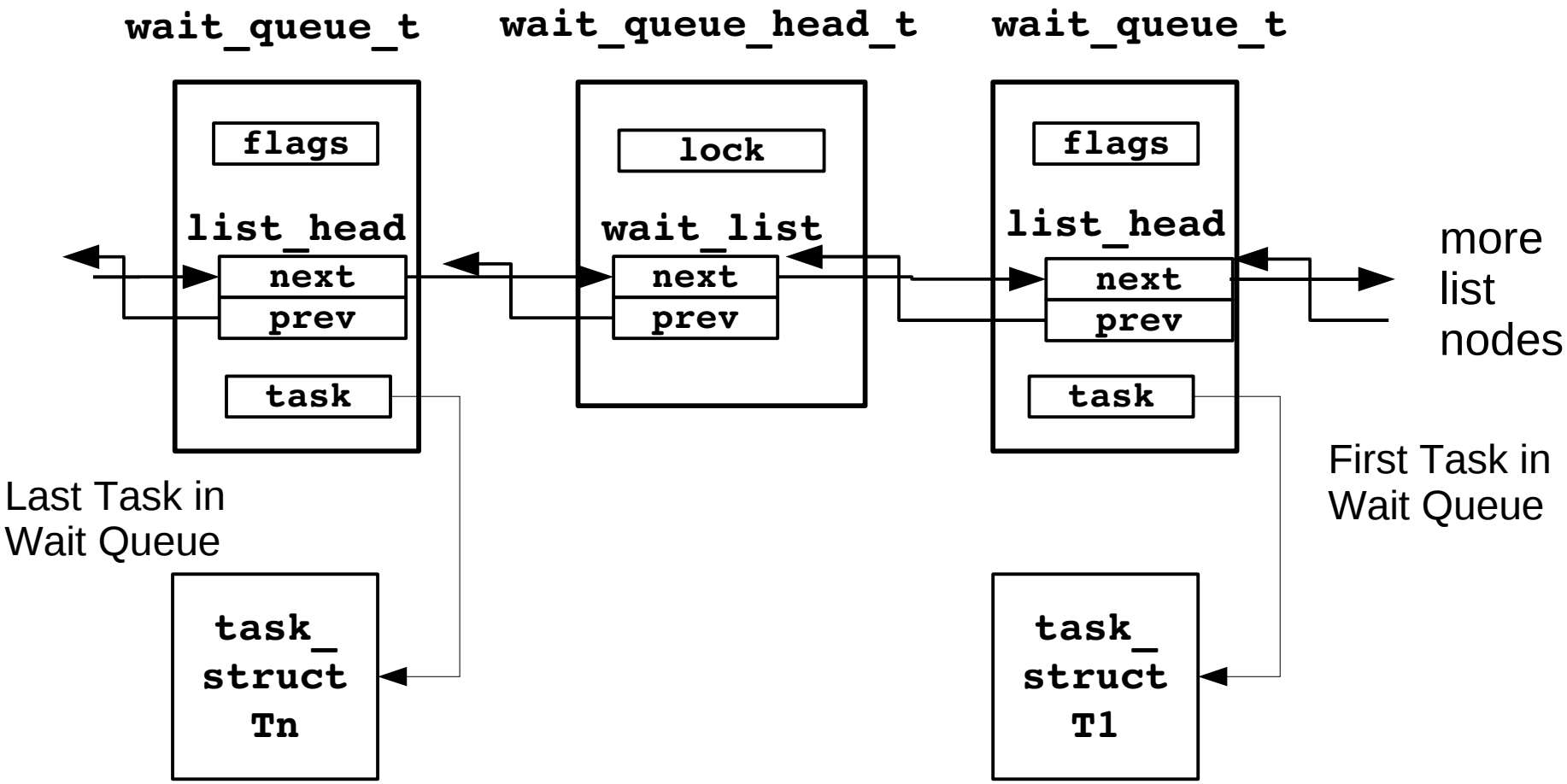
```
wait_queue_head_t  wqh;
init_wait_queue_head(&wqh);
```

  - A wait queue header consists of a spin lock to protect queue operations and a list node:

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

Concurrency Management: Wait Queues (3)

- Wait queue example: Wait queue with n waiting tasks



Concurrency Management: Wait Queues (4)

- The wait queue API, cont. (defined in **<linux/wait>**)
  - Make task wait until some condition will become true. Use one of the following macros, where **queue** specifies the wait queue header and **condition** takes a condition by value like **flag!=0** :

```
wait_event(queue, condition)
wait_event_interruptible(queue,condition)
wait_event_timeout(queue,condition,timeout)
wait_event_interruptible_timeout(queue,condition,timeout)
```

  - The first variant puts a task in an un-interruptible sleep. Be careful.
  - The second variant allows the kernel to wake up a task by sending a signal to it. Often the function returns **-ERESTARTSYS**. Prefer this form.
  - The timeout period is given in **jiffies**.

Concurrency Management: Wait Queues (5)

- The wait queue API, cont. (defined in **<linux/wait>**)
  - To wake up a task there are two functions:

```
void wake_up(wait_queue_head_t *queue);
void_wake_up_interruptible(wait_queue_head_t *queue);
```

  - The **wake_up()** function wake up **all** tasks in the specified queue.
  - The **wake_up_interruptible()** function wake up **all interruptible** tasks in the specified queue.

Concurrency Management: Wait Queues (6)

- Lab -4 (Wait Queues): Synchronize kernel threads with wait queues

  Revisit the module you developed in the semaphore lab: two kernel threads called **`ping`** and **`pong`**. Synchronize those two threads using wait queues and conditions so that they alternatively write messages in the syslog file (/var/log/syslog) using the **`printk()`** function. Make sure that kernel thread **`ping`** begins the game. Prevent the kernel threads from writing too many log messages. You may want to use kernel functions like **`mdelay()`** or **`msleep()`**. Find out the difference. Which of the functions is to prefer on a SMP machine?

  When you have loaded the module then use the **`ps`** utility and find the kernel threads in its output.

  When you unload the module the kernel threads should stop. Verify that by using the **`ps`** utility. Analyze the log messages displayed by the **`dmesg`** utility. Especially, check the time stamps of the messages.

Concurrency Management: Completions (1)

- A common pattern in kernel programming is to start a new kernel thread to perform an action in parallel, and later to wait for that activity to complete (in principle the join problem).
- As you know a semaphore could do the job, however, semaphores in Linux are optimized for the critical section problem.
- It is suggested to use the completion API (<linux/completion.h>)
  - Create a completion
    - Statically

      ```
      DECLARE_COMPLETION(name);
      ```

    - Dynamically

      ```
      struct completion  comp;
      init_completion(&comp);
      ```

Concurrency Management: Completions (2)

- The completion API, cont. (<linux/completion.h>)
  - Waiting for a completion, called by the creator

```
void wait_for_completion(struct completion *comp);
```

    - This function calls in un-interruptible wait. The problem with this is that if no one ever completes the task, the result will be a hanging task.
  - Complete, called by the created kernel thread

```
void complete(struct completion *comp);
void complete_all(struct completion *comp);
```

    - The first form wakes up only one of the threads waiting for the completion. This is the normal case because there is a single waiter.
    - The second form is useful if more than one thread is waiting for the same completion.

Concurrency Management: Completions (3)

- Notice: In some example code about kernel threads you see that completions are used to join threads. However, completions are hidden in the new kernel thread API presented above. In other words use the macro **`kthread_run()`** that in turn calls the function **`kernel_thread_create()`** instead of the older API function **`kernel_thread()`**together with a completion.

Concurrency Management: Completions (4)

- The older kernel thread API using completion explicitly
- Example: Kernel Thread based on the old style kernel thread creation function and a completion object for thread join.
  - Static variables

```
static int    working; /*flag: true while active*/
static structure completion on_exit;
```

  - The kernel thread function

```
static int kthread_ftc(void *arg)
{
        while(working) {
                kprint("kthread loop...\n");
        }
        complete(&on_exit);
}
```

Concurrency Management: Completions (5)

- Kernel thread example, cont.
  - Module init function

```
static int __init kthr_init(void)
{
        init_completion(&on_exit);
        working = 1;    /* allow thread fct to run*/
        kernel_thread(kthread_fct,NULL,0); /*create*/
}
```

  - Module exit function

```
static void __exit kthr_exit(void)
{
        working = 0;        /*stop thread*/
        wait_for_completion(&on_exit);
}
```

Concurrency Management on SMPs: Spin Locks (1)

- Spin locks are the proper mechanism to protect **very short** critical sections on an SMP machine.
- Spin locks use busy wait to avoid context switches. Thus, a thread in a critical section protected by spin locks **MUST NOT** sleep.
- Spin locks are useful in the interrupt context.
- The spin lock macros disable/re-enable interrupts on a uni-processor (UP) machine.

Concurrency Management on SMPs: Spin Locks (2)

- The spinlock API (<linux/spinlock.h>)
  - Spinlock type

```
spinlock_t slock;
```

  - Compile time initialization of a spinlock

```
slock = SPIN_LOCK_UNLOCKED;
```

  - Run time initialization of a spinlock

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

Concurrency Management on SMPs: Spin Locks (3)

- The spinlock API, cont (<linux/spinlock.h>)
  - Lock a spin lock. Recall that a spinlock is un-interruptible by its nature. Consequently, a thread must not sleep in the critical section protected by the spin lock

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

  - There are (at least) two variants of the spin lock function that include interrupts on the local processor. Both disable interrupts on the local processor, befor they acquire the spin lock. The first one saves the interrupt status in the variable flags before. Use the second form **only if** you are sure the interrupts are enabled before you call it.

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

Concurrency Management on SMPs: Spin Locks (4)

- The spinlock API, cont (<linux/spinlock.h>)
  - Release a spinlock. The form you use to release a spinlock must correspond to the function you used to acquire it

```
void spin_unlock(spinlock_t *lock);

void spin_unlock_irqrestore(spinlock_t *lock,
                            unsigned long flags);

void spin_unlock_irq(spinlock_t *lock);
```

  - The second form that store the interrupt state is the safest method perhaps not the most efficient one

Concurrency Management on SMPs: Spin Locks (5)

- The spinlock API, cont (<linux/spinlock.h>)
  - **Important**: Spin locks are NOT recursive: If you attempt to acquire a spin lock you are already holding, you will spinning forever. --> A deadlock is the result.
  - Example: A simple critical section

```
DEFINE_SPINLOCK(slock);
unsigned long flags;

/*uncritical section*/
spin_lock_irqsave(&slock, flags); /*entry*/
/*critical section*/
spin_unlock_irqrestore(&slock, flags);
/*uncritical section*/
```

Synchronization Lab

- Lab-5: Synchronization Analysis

In the previous Labs you learned how to protect critical sections by means of semaphores and mutexes. It is interesting to know how many threads are waiting to enter the critical section that is held by another thread.

Write a module that does the following things

- Create a given number of kernel threads each runs through a critical section. The number is specified by a module parameter.
- When one of your threads has acquired the semaphore/mutex then scan the wait queue and count the number of waiting kernel threads. Of course, you need to scan the queue by using its spinlock.
- Use a flag to indicate a thread to give up its CPU inside the critical section. This may be necessary to obtain the situation of waiting threads. On a SMP machine (The machines in the LAB are such machines) it is not necessary.
- At the end of your test output various statistic fields.