Sistemas Operativos: Concurrency in the Kernel

Pedro F. Souto (pfs@fe.up.pt)

April 23, 2014

Agenda

Agenda

Facets of Kernel Synchronization

Implementation of the synchronization mechanisms

- Many synchronization mechanisms have to be implemented by the kernel

Synchronization in the kernel itself

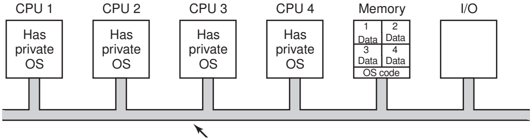
- Several processes/threads can make system calls concurrently
  - The kernel code implementing the system calls may modify kernel data structures
- Interrupt handlers need to access kernel data structures that may be accessed by other parts of the kernel code
- The concurrent execution of kernel code by different processes/threads and of interrupt handlers requires synchronization to prevent race conditions

Nonpreemptive Kernels

- Solution used mostly with uniprocessors
- A process/thread running in kernel mode is never preempted. A process/thread in kernel mode runs until it:
  - Exits the kernel (the system call returns)
  - Blocks
  - Voluntarily yields the CPU
- By careful programming, it is possible to avoid race conditions between processes/threads running in kernel mode
- Race conditions with interrupt handlers can be avoided by:
  - Disabling interrupts when accessing shared data structures
  - Often, the HW allows to selectively inhibiting interrupts
    - This allows the system to be more responsive

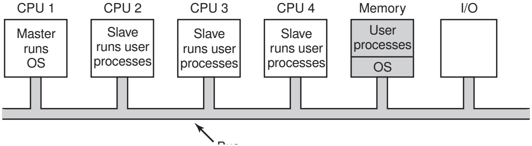
Synchronization on Multiprocessors (1/2)

OS instance per CPU



- When a process makes a system call it is handled by its own CPU
- The issue here is I/O and memory, which are shared

OS on Master CPU also called Master-Slave

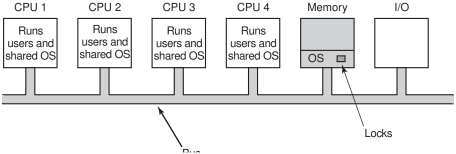


- System calls are redirected to the master CPU

- OS synchronization can be done mostly as on uniprocessors
  - Races can be avoided using nonpremptive kernel

Synchronization on Multiprocessors (2/2)

Single OS which can be run by any CPU



Problem Race conditions

Solution Several:

Single lock whole kernel in the same critical section

- At any time only one process can be inside the kernel
- Requires minimal changes from uniprocessor code

Multiple locks OS components are independent

- However, there are some data structures, such as the process table, that are accessed by otherwise independent parts of the kernel
- Access to multiple data structures may lead to **deadlock**

Agenda

Interrupt Handlers

- Some devices generate HW interrupts to notify the OS of the occurrence of events
  - E.g. the press of a key, the tick of a clock or the arrival of a network packet
- Occurrence of an HW interrupt usually leads to the suspension of the currently running thread, and the execution of an interrupt handler (IH)
  - IHs can also run upon occurrence of a SW interrupt, i.e. the execution of a special ISA instructions
- The way the kernel switches from the interrupted thread to the IH is HW dependent
  - The kernel may have to perform more or less tasks depending on the level of support from the HW

Interrupt Context

- IH execute asynchronously wrt standard OS threads
- The kernel saves the context of the interrupted thread
  - Possibly with the help of the HW
- Often the IH uses the kernel stack of the interrupted thread
  - The kernel stack must be sized accordingly
  - An alternative is to use a dedicated kernel stack for interrupts
- The IH cannot block or sleep
  - The IH has no `struct task`, and therefore is not schedulable
  - It cannot call functions that may block, e.g. `kmalloc()`
- Cannot copy data from/to user-level
  - The interrupted process may not be the sender/destination of the data

Kernel Interrupt Handling

Top-half (Linux) which performs the actions required by the device

- E.g. read a frame from the network card
- It must be as short as possible
  - Sometimes the interrupts are disabled in the top-half
  - Usually, no new interrupts are generated by the device
  - Lower priority interrupts are delayed

Bottom-half (Linux) which performs less urgent actions

- Response to the interrupt can be faster
- Handling can be performed in a more convenient context

  Tasklets are not schedulable
  - Cannot block

  Workqueues are schedulable
  - But do not execute in the context of any user process

Serial port DD No need for bottom-half

- The top-half handles the device and wakes up the user-thread, if any
- The user-level thread copies the data to user-level

Linux IH-related API (`<linux/interrupt.h>`

```
irqreturn_t handler(int irq, void *arg)
```

where:

- `irq` is the interrupt request line (IRQ) that caused execution of the IH
- `arg` is a data structure that must have been registered together with the handler

```
int request_irq(unsigned int irq,
        irq_handler_t handler, unsigned long flags,
        const char *dev_name, void *arg)
```

where:

- `irq` is the interrupt request line (IRQ)
- `handler` name of the IH (function)
- `flags` e.g. whether interrupts should be disabled,
- `dev_name` for `/proc/interrupts`
- `arg` address of data structure to pass as 2nd argument to IH (similar to `pthread_create()`)

Agenda

Concurrency in the Linux Kernel

- Linux supports multiprogramming and kernel-level threads
  - Linux uses preemptive scheduling
    - Although, on occasion the kernel disables preemptions
- Linux supports also several kinds of ephemerous-threads, i.e. threads that
  - Do not execute in the context of a process
  - Perform short duration computations in response to events

  such as:
  - Interrupt handlers
  - Timers, tasklets and workqueues
    - Thread-like entities for deferred execution

  all of which run asynchronously wrt other threads

Linux Kernel Concurrency Control Mechanisms

- Semaphores (can be used as mutexes, or locks)
- Read-write locks (called read-write semaphores)
- Spinlocks
  - Reader-writer spinlocks
- Lock-free data-structures
- Atomic variables
- Other mechanisms
  - seqlocks, based on optimistic locking techniques
  - RCU (read-copy-update), akin to multi-version CC

  both for cases where reads are common and writes rare

CC Implementations

- Some of these mechanisms have different implementations, e.g.
  - For use with data-structures accessed only by user-threads
  - For use with data-structures also accessed by IH, and other "asynchronous threads"
- The reasons for this are two-fold

  Correctness some "threads", e.g. IHs, cannot block nor be preempted (they do not have an execution context)

  Efficiency

Linux Kernel Semaphore API (according to LDD3)

Use: `#include <asm/semaphore.h>`

```
void sema_init(struct semaphore *sem, int val)
```

- Use the appropriate value for `val`, if you want a lock
- But there is also an interface for mutexes, for convenience

```
void down(struct semaphore *sem)
```

```
int down_interruptible(struct semaphore *sem)
```

Should use this one instead of `down()`, as otherwise your thread may get stuck inside the kernel

```
int down_trylock(struct semaphore *sem)
```

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

Linux Kernel Reader/Writer Semaphore API

Use: `#include <linux/rwsem.h>`

`void init_rwsem(struct rw_semaphore *sem)`

Should be called a lock instead of semaphore

`void down_read(struct rw_semaphore *sem)`

No interruptible version :(

`int down_read_trylock(struct rw_semaphore *sem)`

`void up_read(struct rw_semaphore *sem);`

`void down_write(struct rw_semaphore *sem)`

`int down_write_trylock(struct rw_semaphore *sem)`

`void up_write(struct rw_semaphore *sem);`

`void downgrade_write(struct rw_semaphore *sem)`

Should use two-phase locking to prevent deadlock

`IMPORTANT` Writers have priority. From LDD3:

- "as soon as a writer tries to enter the critical section, no readers will be allowed in until all writers have completed their work."

Serial Port DD and Semaphores

Suggestion Implement the DD as a monitor

- Execute each operation of the `struct fileops` in mutual exclusion
- Add a semaphore to the device struct of the serial port DD

Issue This prevents concurrent execution by several user threads, but often the performance is acceptable

- Anyway, this is much better than one lock for the entire kernel

Problem What about the IH?

- Cannot use the semaphore, because in Linux IHs cannot block, they are not schedulable entities, i.e. standard threads
- Need to use another mechanism
  - For accessing the data structure used in the communication between the IH and the user thread
  - The IH does not need to access the device struct

Linux Kernel Spinlock API (1/2)

Use: `#include <linux/spinlock.h>`

`void spin_lock_init(spinlock_t *lock)`

`void spin_lock(spinlock_t *lock)`

- The thread does not block, but rather keeps spinning trying to acquire the lock
- `spin_lock()` disables preemption, so the scheduler will not take the processor away from a thread while it is inside a CS protected by spin locks
  - Why?
- CS protected by spin locks must be as short as possible
  - They cannot make calls to functions that may block/sleep

`void spin_unlock(spinlock_t *lock)`

Issue What if a thread holding a spin lock is interrupted and the IH tries to acquire that spin lock?

Linux Kernel Spinlock API (2/2)

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

- Disables HW interrupts on the local processor only before locking. The previous interrupt state is stored in `flags` (it is not a pointer: `spin_lock_irqsave()` is a macro)

```
void spin_lock_bh(spinlock_t *lock)
```

- Disables SW interrupts on the local processor only before locking, but leaves HW interrupts enabled

```
void spin_lock_irqrestore(spinlock_t *lock,
                          unsigned long flags)
```

- `flags` should be the value returned from `spin_lock_irqsave()`

```
void spin_unlock_bh(spinlock_t *lock)
```

Note These functions should be used when a spinlock can be taken in the context of handling an interrupt (either HW or SW)

kfifo (<linux/kfifo.h>) (1/3)

A circular buffer like the one we used to study concurrency

```
struct kfifo {
    unsigned char *buffer; /* the buffer for data */
    unsigned int size; /* the size of the buffer */
    unsigned int in; /* data is added at (in % size) */
    unsigned int out; /* data is fetched from (out % size) */
    spinlock_t *lock; /* protects concurrent changes */
};
/* kfifo_init - allocates FIFO using a preallocated buffer
 *    lock must have beee previously initialized        */
struct kfifo *kfifo_init(unsigned char *buffer,
   unsigned int size, gfp_t gfp_mask, spinlock_t *lock)

/* kfifo_alloc - allocates FIFO and its internal buffer
 *    lock must have beee previously initialized
 * The size will be rounded-up to a power of 2. */
struct kfifo *kfifo_alloc(unsigned int size,
              gfp_t gfp_mask, spinlock_t *lock)

/* kfifo_free - frees the FIFO, including the buffer */
void kfifo_free(struct kfifo *fifo)
```

kfifo (<linux/kfifo.h>) (2/3)

```
/**
 *  kfifo_put - puts some data into the FIFO
 * @buffer: the data to be added.
 * @len: the length of the data to be added. */
static inline unsigned int kfifo_put(struct kfifo *fifo,
     unsigned char *buffer, unsigned int len) {
    unsigned long flags;
    unsigned int ret;
    spin_lock_irqsave(fifo->lock, flags);
    ret = __kfifo_put(fifo, buffer, len);
    spin_unlock_irqrestore(fifo->lock, flags);
    return ret;
}

/**
 * kfifo_get - gets some data from the FIFO
 * @buffer: where the data must be copied.
 * @len: the size of the destination buffer. */
static inline unsigned int kfifo_get(struct kfifo *fifo,
     unsigned char *buffer, unsigned int len)
```

kfifo (<linux/kfifo.h>) (3/3)

From kernel/kfifo.c

```
unsigned int __kfifo_put(struct kfifo *fifo,
                unsigned char *buffer, unsigned int len)
{
    unsigned int l;
    len = min(len, fifo->size - fifo->in + fifo->out);
    /* Ensure that we sample the fifo->out index -before- we
     * start putting bytes into the kfifo. */
    smp_mb();
    /* first put the data starting from fifo->in to buffer end *
    l = min(len, fifo->size - (fifo->in & (fifo->size - 1)));
    memcpy(fifo->buffer + (fifo->in & (fifo->size - 1)),
           buffer, l);
    /* then put the rest (if any) at the beginning of the buffer
    memcpy(fifo->buffer, buffer + l, len - l);
    /* Ensure that we add the bytes to the kfifo -before-
     * we update the fifo->in index. */
    smp_wmb();
    fifo->in += len;
    return len;
}
```

Atomic Variables/Bits

Arithmetic operations on integer type `atomic_t`. E.g.:

```
void atomic_set(atomic_t *v, int i)
int atomic_read(atomic_t *v)
void atomic_add(int i, atomic_t *v)
void atomic_inc(int i, atomic_t *v)
```

Bit operations on a memory position. E.g.:

`void set_bit(int nr, void *addr)`

`void clear_bit(int nr, void *addr)`

`void change_bit(int nr, void *addr)` toggles bit

`void test_and_set_bit(int nr, void *addr)` as well as for the other operations (clear and change)

- Some instruction set architectures (ISA) provide instrutions that perform these operations
  - The Linux API allows to develop portable code
  - That will work even if the ISA of the processor being used does not provide the operation being invoked

Waitqueues

Issue `kfifo`'s use spinlocks:

- + IH cannot block
- \- User threads must busy wait

How can user threads avoid busy waiting?

NON-Solution Use *semaphores*

- Operations on semaphores may block and IH cannot

Solution Use *waitqueues*

Def. A wait queue is a queue of threads waiting for some event

Each queue is defined by a queue header of type `wait_queue_head_t`:

```
wait_queue_head_t wqueue;
init_waitqueue_head(&wqueue);
```

Waitqueue Hi-level API (`<linux/wait.h> macros`)

Waiting for an event

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

where:

`queue` is the header of the waitqueue
`condition` is an arbitrary C boolean expression
`timeout` is the duration of a timeout in *jiffies*, which is given by `1/HZ` (the vbox image is using a value of 100 for `HZ`)

Waking up

```
void wake_up(wait_queue_head_t *queue);
void wake_up_interruptible(wait_queue_head_t *q);
```

wait_queue()

```
#define wait_event(wq, condition)   \
    do { \
        if (condition)    \
            break; \
        __wait_event(wq, condition); \
    } while (0)
#define __wait_event(wq, condition)   \
    do { \
        DEFINE_WAIT(__wait); \
        for (;;) { \
            prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE);
            if (condition) \
                break; \
            schedule(); \
        } \
        finish_wait(&wq, &__wait); \
    } while (0)
#define DEFINE_WAIT(name) \
    wait_queue_t name = { \
        .private = current, \
        .func = autoremove_wake_function, \
        .task_list = LIST_HEAD_INIT((name).task_list), \
    }
```

Waitqueue Low-level API

- `prepare_to_wait()` and `finish_wait()` belong to a low-level API
  - The use of this API is tricky
- In the serial port DD use the hi-level API instead:

```
typedef struct {
    struct cdev cdev;
    struct semaphore mutex;
    struct kfifo *rxfifo;   // receiver fifo
    wait_queue_head_t rxwq; // for IH synchron.
    [...]
} seri_dev_t;
```

- If you wish to use the low-level API beware of the "lost-wakeup" bug

Agenda

O Problema do *lost wakeup()* (1/2)

- O *thread dispatcher* do servidor de *Web* poderia incluir o seguinte código:

```
lock();
while(bbuf_p->cnt == BUF_SIZE) { /* Busy wait */
    unlock();
    lock();
}
enter(bbuf_p, (void *)req_p); /* Enter request *
unlock();                     /* in buffer */
```

- Para evitar *espera activa*, o SO pode oferecer o par de chamadas ao sistema: `sleep()` e `wakeup()`.

O Problema do *lost wakeup()* (2/2)

- Para evitar desperdiçar o tempo do CPU, poderia usar-se:

```
lock();
while(bbuf_p->cnt == BUF_SIZE) {
    unlock();
    sleep(bbuf_p);          /* Block thread */
    lock();
}
enter(bbuf_p, (void *)req_p);
unlock();
```

- Para desbloquear o *dispatcher*, os *worker threads* executariam:

```
req_p = (req_t *)remove(bbuf_p);
if(bbuf_p->cnt == BUF_SIZE - 1) /* Buffer was full
    wakeup(bbuf_p);      /* Wakeup dispatcher thread
```

- Este código tem uma *race condition* (lost wakeup) entre a aplicação e o SO, que pode bloquear o *dispatcher* para sempre. Qual é?

Agenda

Additional Reading

Sistemas Operativos

- Section 5.6

Modern Operating Systems, 2nd. Ed.

- Section 8.1

J. Corbet, A. Rubini, and G. Kroah-Hartman, "Linux Device Drivers", 3rd Ed., O'Reilly

Ch. 5: Concurrency and Race Conditions
Ch.10: Interrupt Handling
Ch. 7: Time, Delays and Deferred Work
Ch. 6: Advance Char Driver Operations