Operating Systems II

Interprocess Communication

IPC Support

- IPC is used to coordinate and share information between tasks.
- Resource sharing
- Synchronization
- Data exchange (connectionless and connection oriented)
  - This can be in the form of one-to-one or one-to-many.

Kernel Synchronization

- The kernel manages system resources and access to these resources must be synchronized.
- To provide the synchronization the kernel controls who can run during a system call.
- Other processes get “scheduled” only in three different cases...

Kernel Synchronization (cont.)

1. When the system call invokes the `schedule()` method.
2. When the system call invokes a method that will *suspend* the process.
3. If pre-emptive scheduling is in force, when an interrupt occurs that affects scheduling.

Kernel Synchronization (cont.)

- So, the kernel provides synchronization internally by:
  - Only calling schedule() when it will not affect synchronization.
  - Same for a suspending function.
  - Turn off interrupts when performing critical regions.
    - Simple, but does slow down the system…

Multiprocessor Systems

- Unfortunately, the approach of enabling/disabling interrupts does not work for multiprocessor systems.
- So how does it perform synchronization?
  - Using a spinlock (mutex).

Spinlocks

- `typedef struct {int lock;} spinlock;`
- Just like a regular mutex, unlocked it contains a 1, locked it contains a 0.
- System calls try to acquire the lock by decrementing the value. Release the lock by incrementing the value.
- Requires processor test&set atomic operation.

Spinning the Tires...

- Unlike an OS mutex/semaphore these locks have a busy loop!!!
- In the kernel, you cannot put the system call or ISR into a waiting state. So you just spin the loop waiting for it to be unlocked.
- No problem ... the other processor is expected to release the lock.
- Defined to be empty on single processor systems.

Assessment of Spinlocks

- Can be used in interrupt service routines on multiprocessor systems.
- Quick
- ... and dirty. Wasted clock cycles of execution.

Read/Write Locks

- Same as spinlocks, but ...
  - Allow multiple readers of a resource.
  - Only one writer with no readers.
- `typedef struct {int lock; }rwlock;`
- Lock field starts with `RW_LOCK_BIAS`
  - For example 0x01000000

Read/Write Locks (cont.)

- Readers try to decrement the lock and the result being a positive number.
- Writers try to decrement the lock by `RW_LOCK_BIAS` and get a 0.
- Also carried out by a busy loop.
- More costly to perform than spinlocks
  - Use them more wisely!

Spinlock Macros

- There are several macros one can provide to assist in using spinlocks & read/write locks.
- `SPIN_LOCK_UNLOCKED & RW_LOCK_UNLOCKED`
  - Initialization of a spinlock or read/write lock.
- `spin_lock() & spin_unlock()`
  - Lock and unlock without any affect on interrupts.
- `read_lock() & read_unlock()`
- `write_lock() & write_unlock()`
  - Lock and unlock without any affect on interrupts.

Spinlock Macros (cont.)

- `spin_lock_irq()` & `spin_unlock_irq()`
  - Lock/Unlock, but also allow interrupt manipulation for current processor. Sets interrupt mask.
- `spin_lock_irqsave()` & `spin_unlock_irqrestore()`
  - Lock/Unlock, but also allow interrupt manipulation for current processor. Restores interrupt mask.

Task synchronization

- Application processes (tasks) are synchronized by the use of `wait_queues`.
- As previously discussed, queues are often implemented as doubly linked lists.

Queue macros

- Queue management only!!!
- `DECLARE_WAITQUEUE()`
  - Declares and initializes a wait_queue_t structure.
- `add_wait_queue()`
  - Adds the task to the queue
- `remove_wait_queue()`
  - Removes the task from the queue
- `wait_queue_active()`
  - Anything in the queue?

Synchronization Methods

- `sleep_on()`
  - eternal sleep in the queue.
- `interruptible_sleep_on()`
  - sleep, but can be awoken by an interrupt.
- `sleep_on_timeout()`
  - sleep for a fixed amount of time.
- `interruptible_sleep_on_timeout()`
  - sleep for a fixed amount of time AND can be interrupted from sleep.

Synchronization Methods (cont.)

- `wake_up()`
  - Wake the task from the queue.
- `wake_up_nr()`
  - Wake a given number of tasks from the queue
- `wake_up_sync()`
  - Wake up tasks, but do not schedule until next **regular** scheduling period.
- `interruptible wake_up() variants`
  - Wakes up ONLY tasks that were put to sleep with interruptible status.

sleep_on()

```
Sleep_on(struct wait_queue **p){
     struct wait_queue wait;
     current->state = TASK_UNINTERRUPTIBLE;
     wait.task = current;
     add_wait_queue(p, &wait);
     schedule();
     remove_wait_queue(p, &wait);
}
```

Semaphores

```
Struct semaphore {
        int count;
        int sleepers;
        wait_queue_head_t *wait;
}
```

- count - The lock! Atomic for test&set in one operation. Supported by hardware architecture.
- sleepers – sleepers + count = correct value for semaphore.
- wait - The list of tasks that are sleeping.

Semaphores (cont.)

- up() – increment count and wake ALL tasks if <= 0; wake a process if count = 1.
- down() – decrement count; modify sleepers; and sleep if count < 0.
- down_iterruptible() – task is interruptible if it does go to sleep on trying to lock.
- down_trylock() – does not block task if lock cannot be acquired.

Communication via Files

- Oldest mechanism of exchanging data.
- In a multitasking environment you require synchronization to ensure correctness with the file.
- Most kernels provide file locking…
  - You can lock the full file or just areas of the file.
  - You can have *mandatory* or *advisory* locking.

Implementation Details

- The kernel maintains a linked list of `file_lock` structures for each lock.
- The list is per file!
- Different information stored in the structure include
  - next pointer, list of waiting processes, file indicator, start & end of lock, lock type, ...

File Control

- `sys_fcntl(int fd, int cmd, void* arg);`

- `fd` – file descriptor
- `cmd` – command to perform on the file
- `arg` – argument(s) needed to perform the command.

Locking Files

- cmd can be one of F_GETLK, F_SETLK, or F_SETLKW.
- arg – flock structure

```
Struct flock {
 short l_type; /* F_RDLCK,F_WRLCK,F_UNLCK */
 short l_whence; // SEEK_SET,SEEK_CUR,SEEK_END
 off_t l_start; // offset relative to l_whence
 off_t l_len; /* length of area to lock */
 pid_t l_pid; /* returned with F_GETLK */
}
```

Locking Files (cont.)

- `F_GETLK` – tests whether the lock is possible, if not attempted lock returned.
- `F_SETLK` – sets the lock specified, returns either way.
- `F_SETLKW` – sets the lock specified, but blocks if lock cannot be set.

Pipes

- Pipes are used to share data between two distinct processes.
- The pipe uses a go between `inode` (file system block) to store the information.
- Once created, processes use `read()` and `write()` to interact with the pipe.

Pipes (cont.)

- Creating a pipe involves
  - allocating an `inode` for storing the data.
  - generating a reader and writer file descriptor for the `inode`.
- Accessing the pipe is the same as for a file.

`pipe_inode_info` structure

```
wait_queue_head_t wait; /* wait queue for processes */
char *base; /* address of buffer */
int start; // amount written and yet to be read
int len; /* current amount that is unread. */
int readers; /* # of processes reading */
int writers; /* # of processes writing */
int waiting_readers; // # blocked readers in Q
int waiting_writers; // # blocked writers in Q
int r_counter; // # read processes that opened
int w_counter; // # write processes that opened
```