COMSW4118-12-1

10/12/2017

Linux Synchronization Primitives (cont.)

- Atomic types
- spinLocks = kernel/sched/core.c (see all uses of spin_lock)
  - spin_lock_irqsave & spin_lock_irqrestore for when structure can be modified from an interrupt.

DEFINE_SPINLOCK(name) is the static way to initialize

More or Less implemented as:

```
while (test_and_set(x));
```

But with some additions to resolve cases when you grab a lock and then someone preempts you and just sits there spinning and waiting for your lock, wasting time:

```
spinlock(x) {
    disable_preemption();
    while (test-and-set(x)) {
        enable_preemption();   <- to give a chance to be preempted
        disable_preemption();
    }
    enable_preemption();   <- in actual Linux interpretation this is probably not here, i.e. you can't be preempted while holding a spinlock.
}
```

To make this an uninterruptible, just disable interrupts at start and re-enable or restore them before returning.

- Locks only work if you use them correctly

Monitors are meant to make use easier:

```
monitor {
    f1(){...}; f2(){...}; f3(){---};
}
```

By guaranteeing mutual exclusion b/n f1...f3 w/in the monitor.

Example:

```
monitor {
    float balance;
    void credit(float a){ balance+=a;};   <- only one function can be executing at a time.
    void debit(float a){ balance-=a;}
}
```

- **Condition Variables** - wait for events within the monitor:

```
ConditionVariable {
    void wait()    <- wait till someone signals (and get out of the monitor) meanwhile
    void signal()  <- signal the event
}
```

NOTE: this allows threads to do things that depend on one another

Suppose t_0 is in wait() and t_1 does signal(), now t_1 is in the monitor but t_0 wants to get in - which one do we run? Either way can be implemented. Hoare says run t_0 immediately & suspend t_1. Hansen says to keep running t_1. There is no right or wrong answer.

▶ Monitors don't exist as a construct in the Linux kernel.

But the idea of a critical section with signaling (condition variables) goes beyond monitors, for example:

- Critical section implemented with a lock
- Within it wait for a condition/event, giving up the lock at the same time
- When done waiting, grab the lock again & continue

Basically, the core idea of using a condition variable w/in a critical section.

● **Waitqueues** are Linux's mechanism for going from running to wait for some event and becoming non runnable

▶ High-level constructs are wait_event(queue, condition) and wake_up(queue) ← wake up waiters in queue ← wait for condition

wait_event(queue, condition) {

DEFINE_WAIT(wait);
while(1) {
  lock(lock) put on queue for wait
  prepare_to_wait(queue, wait, task_state);
  if(condition) break;
  unlock(lock);
  schedule();
  lock(lock);
finish_wait(queue, wait);
unlock(lock)

TASK_INTERRUPTIBLE
TASK_UNINTERRUPTIBLE
determine if pulled off of wait queue if signaled

release lock, then wait and reacquire lock

schedule() says run something else

The scheduler is the part of the OS that decides what to run on the CPU (typically, triggered by a timer interrupt).

/kernel/sched/core.c → __schedule(void)

You keep sleeping till someone calls wake_up(), which will find jobs on the wait queue and change their state to TASK_RUNNING (meaning task is runnable, Linux does not distinguish b/n running & runnable). Then the scheduler picks it up from there. Processes that were waiting pick up from schedule() onwards.

COMSW4118-12-3 10/12/2017

Linux has a **Runqueue** (a List of runnable/running tasks). Tasks that are waiting are on a **waitqueue**, but when are signaled or woken (as TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE) up, they get moved to the runqueue w/ status TASK_RUNNABLE. The scheduler only looks at the runqueue.

- wakeup_all() wakes up all waiters, wakeup() wakes up one
- wait_event() is uninterruptible, wait_event_interruptible() is interruptible

▶ For more complex wait checks (i.e. when a simple condition is not sufficient), we can just code sth. similar to the **wait_event()** implementation. or search for examples how **prepare_to_wait()** gets used. ↓

pretty much what we have on prev. page. ← include/linux/wait.h

- Use add_wait_queue() after DEFINE_WAIT(wait) to add the wait together w/ other waits in the same queue (optional?)
- usually wait in a loop to re/check condition and sleep again, if necc.
- prepare_to_wait() is to make yourself not running (in a blocked state) and add yourself to the queue.
- **Reference Counting** - check LKD book for description
  similar to **RCU** in that you keep track of users for sth. you need to release.
- **Kfifo** - kernel FIFO buffer
- **IDR** - integer id management mechanism

- **Scheduler** - A thread runs, at some point, because of an interrupt (HW or syscall) you get into the kernel. Till you get to the kernel, this thread will always be running. Once you are in the kernel code, it will check if it needs to run something else. That happens because somewhere in the OS, **schedule()** gets called:
  - either directly, like in wait_event() to release processor
  - or indirectly by setting a schedule flag in interrupt code (for. ex.) since we don't want to call schedule (a complex call) there, and then (after interrupt is done) checking it and calling schedule() if necessary.