Kernel driver prog. day 3

Presented by

Hans de Goede

Multitasking/Threading

- Cooperative: Tasks voluntary give up the cpu by calling into the OS themselves
  - They can do so at a convenient time removing the need for locking with multi-threading
- Pre-emptive: The OS takes the CPU away from the task when its timeslot is up
  - This can happen at any time → need locking
  - Stuck processes cannot kill the entire system

Linux

- Uses pre-emptive task switching when a task is executing userspace code
- Traditionally uses cooperative task switching when a task is executing kernel code
- Also supports a semi-realtime mode where it uses pre-emptive task switching for tasks executing kernel code too
- This new semi-realtime mode is often the default

Kernel entry points

- On boot the first cpu core starts executing kernel
- When a task makes a system call the cpu core running that task starts executing kernel code
- On a hardware interrupt the cpu core which receives this interrupt starts executing kernel code

Kernel contexts

- On boot and on a system call the kernel code being run runs in process context.
- In process context the code may call into the scheduler to schedule another task while it waits for some event, this is called sleeping and is a coorperative task switch

Kernel contexts

- On a hardware interrupt the kernel code being run is in atomic context
- In atomic context the code cannot sleep since it is impossible to schedule another task and later go back to executing the interrupt handler
- An interrupt handler must finish in one go, hence the name atomic
- An interrupt handler must clear the source of the interrupt

fedora

Locking

- Given hardware interrupt handling, multiple cpu cores and kernel pre-emption, any kernel code can be running at the same time as any other kernel code, including itself
- This means that the kernel must make extensive use of locking to avoid race conditions
- This locking is often fine grained to avoid slowdowns due to other tasks waiting for the same lock (lock starvation)

Lock types

- Mutexes are the standard kernel locks, these sleep while waiting to aquire a lock and thus can only be used in process context
- Spinlocks are locks for use in atomic context, these use a busy loop waiting for the lock, hence the name spinlocks
  - Code sections protected by spinlocks must be short both in amount of code and executing time
  - Taking a spinlock in process context switches to atomic context until the lock is released
  - No sleeping while holding a spinlock!

fedora™

Locking (2)

- Most Linux subsystem take care of locking for you, but you must always be aware of the locking being done, and in some cases you may need to take a subsys lock yourself
- To figure out the locking rules for a subsystem you've to read the subsystem code, there is no comprehensive and uptodate documentation

Lock ordering

- You must always take locks in the same order
- If you've a code-path taking first lock b and then lock a, then ALL your code paths taking BOTH lock a and b must first take lock b and then lock a
- Not following this rule will lead to deadlocks which causes hanging systems and unhappy users (which is not good ™)

Questions?

Contact:

hdegoede@redhat.com