Kernel Module Programming - 2

Alessandro Barenghi

Dipartimento di Elettronica e Informazione
Politecnico di Milano

barenghi - at - elet.polimi.it

June 7, 2012

Recap

By now , you should be familiar with...

- Programming with sockets employing different protocols
- System programming, synchronization primitives and IPC
- System administration skills , as far as the local host and network monitoring go
- Network administration and filtering, tunnels and NAT

Lesson contents

Overview

- Circular Buffers
- Read-Copy-Updates (RCUs)
- DebugFS
- A character device

Lock- and Wait- freedom

Overview

- In synchronization mechanisms, a key issue is preventing deadlocks: a deadlock is a state of the computation where the access to the resources is prevented due to a circular dependence in the access
- If a mechanism warrants that every entity will access a protected region, it is called lock-free
- In case the access will necessarily happen within a bounded number of steps, it is also defined as wait-free
- Lock-freedom warrants that a system will not hang, wait-freedom that noone will starve (i.e. that access to a resource is possible in a bounded amount of time)
- Only a few wait free algorithms are known in literature: we will tackle circular buffers and read-copy-update mechanisms

Circular buffers

Overview

- Circular buffers are a memorisation structure which can be accessed in a lockless, wait-free fashion
- The key idea is that a memory buffer is thought of as circular instead of the common linear form
- This implies that writing beyond the end of the buffer starts writing back from the beginning
- The most common implementation involves two cursors, one pointing to the beginning of the valid data, the other to the end
- Key element: can be implemented even without atomic variables

Circular buffers

Typical actions

- Only one reader or writer is admitted to the structure; the structure is lock free as no possible deadlocks can happen
- **Reader**: the reader accesses the buffer reading the pointers first. Once the boundaries are known, the read access will be safe.
- **Writer**: the writer reads the boundaries, performs the writing action and finally updates the end pointer.
- Deletion from the buffer is managed moving forward the start pointer (no explicit need to blank the memory cells)

Circular buffers

Distinguishing between full and empty

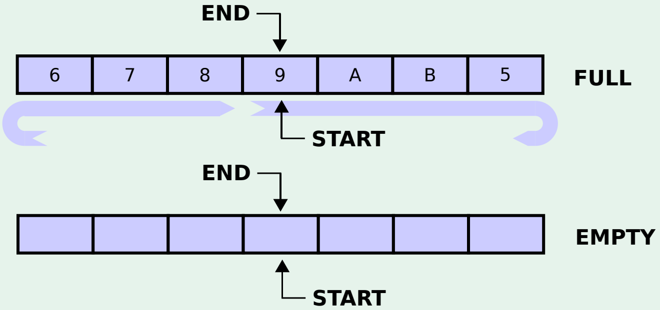


Figure: It is not possible to distinguish between a full and an empty buffer

Circular buffers

Issues and solutions

- Possible solutions to distinguish a full from an empty buffer are:
  - Use integer indexes instead of pointers: no extra variables needed, but each access to the structure costs a *modulo*[a] operation as the indexes are constantly incremented
  - Use a fill counter: requires greater care when the write operations wrap around the buffer, but saves a variable (end pointer) and simplifies fullness test
  - Always keep one cell open: never fill up the last free cell and declare the buffer full before: loses a little space at the cost of no computational/space overhead (chosen in Linux kernel implementation)

[a] This reduces to a bitwise mask if the length of the buffer is 2^n

Circular buffers

Linux Kernel implementation

- Implementing a circular buffer is rather straightforward, you can cook your own soup (although this is not advised)
- Linux kernel offers a standard three pointer structure to uniform the implementation in `circ_buf.h`
- The header also includes a couple of helper macros
  - `CIRC_CNT` : returns the used space in the buffer
  - `CIRC_SPACE` : returns the free space in the buffer
  - `CIRC_CNT_TO_END` : returns the used slot count up to the (linear) end of the buffer
  - `CIRC_SPACE_TO_END` : return the space count up to the (linear) end of the buffer

Read-Copy-Update

Overview

- Fully wait-free reads (with multiple readers) and wait-free write (one writer only) is achievable via Read-Copy-Update constructs
- RCUs are a relatively recent (2006) strategy to avoid update conflicts on a shared variable
- They are now implemented in both the Linux kernel and as a user space available library `liburcu` and their use is advised whenever a variable is shared among many readers, while being updated by a few writers
- The key idea is to decouple the writing phase from the removal of the old data, avoiding syncronization issues

Read Copy Update

Roles

- Key Idea: the writer makes a copy of the value he wants to update updates the copy which is added to the structure in a second time
- The readers are provided a lock on the last, fully updated, copy of the data, no risks of read hazards are possible
- In the regular working of RCUs there are three key roles :
  - Reader: The reader is pointed to the last stable version of the data, this data is not deleted until the reader has finished reading
  - Updater: The updater needs to change the data: it is allowed to do so on a shadow copy which is linked to the structure in a second time
  - Reclaimer: The reclaimer is in charge of swapping the old data with the fresh ones only when there are no longer any readers locking the old

Read Copy Update

Pros and Cons

- RCUs provide a very fast, lockless, read access to many readers, even in concurrency to a pointer based structure
- It is critical that only a single updater at a time acts on it
- The updater can immediately write the update on his personal shadow copy, so the action will finish in a limited amount of time (wait-free)
- The whole structure can be implemented without the use of atomic variables
- On non-preemptible kernels, the reader lock of the RCU does not need to be performed (the compiler does not emit any code for the lock function): all the read actions are completed within the time quantum

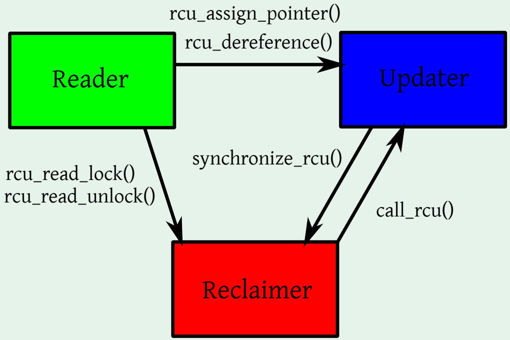
Read Copy Update

Linux Kspace RCU

- The Linux kernel offers a full fledged, simple RCU API:
  - `rcu_read_lock()` / `rcu_read_unlock()` allow the readers to assert a lock on a specific version of the data
  - `rcu_dereference()` and `rcu_assign_pointer()` allow the updater to access properly the data to be updated
  - `synchronize_rcu()` Allows to wait until all the pre-existing RCU read critical sections have completed
  - `call_rcu()` Sets up a callback function to be invoked when all the read locks expire : this allows the updater to move on with other tasks leaving the RCU reclaimer safely in background
- The same APIs are available in both garden variety and soft IRQ blocking flavour via adding a `_bh` suffix to the call name

Read Copy Update

Linux Kspace RCU Visual summary



Communications with the real world

Devices

- In order to expose a unified interface for communication with the hardware, the kernel exposes devices
- Following the UNIX philosophy, the devices are seen in userspace as simple files
- It is possible to either expose a real device via a block/character interface (e.g. /dev/sda)
- Or to build a mockup device which may be useful (/dev/zero)
- A simpler alternative, if there is only the need to communicate between userspace and kernelspace is the debug filesystem

Communications with the real world

Quick debugging I/O

- Originally, the proc filesystem served as both a quick debugging interface and to expose a parameter passing interface to the kernel parts
- In the current Linux Kernels, these two roles have been split and implemented in the DebugFS and SysFS respectively
- It is thus possible to obtain a quick, file based communication interface through creating a file in DebugFS
- The read/write callbacks must be implemented by the module developer and handle the common read/write operations on the file
- A directory structure can be easily created via the exposed API to organize the output

Communications with the real world

A real device

- A real character device needs to implement all the possible operations which can be performed on it
- Moreover, it is required to handle the number of stakeholders which are actually using the device to avoid improper removal of the module
- The devices are accessible from the userspace via a peculiar filesystem entry, which does not have any actual space reserved on disk known as device node
- Real devices are split into :
  - Character devices: minimum unit for access : single character (one byte), usually unbuffered
  - Block devices: minimum unit for access : a block of data (a contiguous chunk in the kB size range), usually buffered

Communications with the real world

Device implementation

- We will see the implementation of a mockup character device[a]
- A character device needs to implement at least four key primitives : `open`,`read`,`write` and `release`
- It also needs to take into account whether someone is using the device in order to prevent premature module removal
- The transferral of the data from kernel to user address space is managed by the `put_user` primitive

[a] Block devices go the same way, just with more functionalities

Communications with the real world

Node setup

- A device node can be created via the `mknod` utility and needs three parameters
    - The type of the device (block or character device)
    - The major number, i.e. a unique, kernel assigned, identifier for the device
    - The minor number, a sub-index handled by the module answering for that device in kernelspace
- A list of all the devices exported by the kernel is available via `/proc/devices`
- It is possible also to avoid static devices via the udev filesystem, which is automatically populated by the kernel[a]

[a] say, the partitions of a hard disk