Kernel Module Programming

Alessandro Barenghi

Dipartimento di Elettronica e Informazione
Politecnico di Milano

barenghi - at - elet.polimi.it

June 27, 2011

Recap

By now , you should be familiar with...

- System administration and userspace system programming
- Network administration and userspace network programming
- Basic Linux kernel module programming

Lesson contents

Overview

- Wait-free synchronization mechanisms
- Netfilter internal structure
- A new match rule for Netfilter
- A new target for Netfilter

Lock- and Wait- freedom

Overview

- In synchronization mechanisms, a key issue is preventing deadlocks
- In case a mechanism warrants that every entity accessing the protected region will gain access eventually, it is called lock-free
- In case the access will necessarily happen within a bounded number of steps, it is also wait-free
- Lock-freedom warrants that a system will not hang, wait-freedom that noone will starve
- 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 represented as circular instead of linear
- 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

Issues and solutions

- Only one reader or writer is admitted to the structure
- There is an issue when the buffer is completely full as start and end pointers will be in the same position as the empty buffer
- Some viable solutions are:
  - Use indices instead of pointers: no extra variables, costs a *modulo* operation each access[a]
  - Use a fill counter: needs only an additional variable but is a pain to track it properly
  - Always keep one cell open: lose an element, without any other overhead (chosen in Linux kernel implementation)

[a] may not be that slow if the modulo is 2^n

Circular buffers

Linux Kernel implementation

- Implementing a circular buffer is rather straightforward, any plain implementation will work
- 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 lockless read for many readers and wait-free write 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`
- The key idea is to decouple the writing phase from the removal of the old data

Read Copy Update

Roles

- In the regular working of RCUs there are three key roles :
  - Reader: The reader needs to access the latest, fully written data: it is the one effectively locking the data
  - Updater: The updater needs to change the data: it is allowed to do so on a shadow copy
  - 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
- As the readers are provided a lock on the last, fully updated, copy of the data, no risks of read hazards are possible

Read Copy Update

Pros and Cons

- RCUs provide a very fast, lockless, read access to many readers, even in concurrency,
- It is critical that only a single updater acts at a time
- The locking taken by the updater is no big deal, since the update is warranted to be wait free
- The whole structure can be implemented without the use of atomic variables

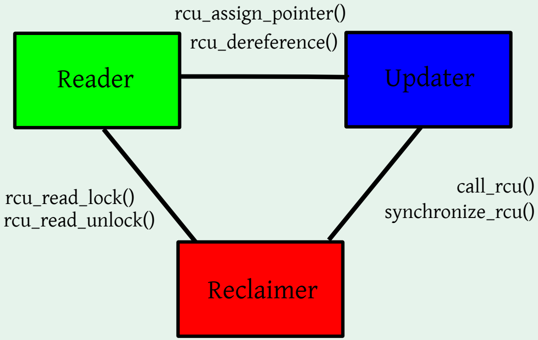
Read Copy Update

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 update 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



Netfilter inner structure

Firewalling from the other side

- On kernel side, the netfilter structure is practically represented by five hooks
- Each hook corresponds to one of the five fundamental tables we have seen in the firewalling and NAT lessons
- It is possible to directly bind to one of the five fundamental hooks...
- or to modularly add a match or a target rule (more flexible, more reusable)

Netfilter inner structure

Adding matches and targets

- In order to add a new match or a new target, the new Netfilter tables provides proper registration/deregistration functions
- All the functions related to the newest Netfilter tables are prefixed with the `xt_` prefix
- These functions act on IPv4 and IPv6 likewise , as on any other transport protocol which will be implemented in future
- The full description of the new match/target is provided via a static structure which must be filled prior to registering the module

Netfilter inner structure

Roles

- After the last reengineering, there is a strict splitting among the roles of matches and targets
- Matching rules should only check if a particular condition is true or false and return the result without affecting the packet
- Target rules are only allowed to perform actions on a packet (mangle it, derive informations, log, blink leds) but should act on any packet buffer passed to them
- The packets are handled in the form of a C union `sk_buff` which is passed by reference to both matchers and targets
- Each rule added to a hook invokes at first the matching function and, if it returns true, it calls the corresponding target function

Netfilter inner structure

Rule Codes

- In order to perform actions on the packets, other that mangling, the target main function should return one of the following rule codes:
  - `NF_ACCEPT`: Accept the packet and send it further up (or down) the network stack
  - `NF_DROP`: drop the packet instantly
  - `NF_REPEAT`: repeat the hook function from scratch
  - `NF_STOLEN`: similar to NF_DROP but the packet effectively vanishes from the counters[a]
  - `NF_QUEUE`: queue the packet to userspace via Netlink
  - `XT_CONTINUE`: continue to the next rule in the hook

[a] it is assumed that the programmer takes care of the packet memory area from there onwards

A matching rule

Registering the matcher

- A new matching rule can be registered and unregistered via the `xt_register_match` and the `xt_unregister_match` functions
- Both functions accept a `xt_match` structure containing:
  - `name`: the string which will be recognised after the `-m` option of the `iptables` command
  - `revision`: the version of the matcher
  - `family`: the family of protocols on which the matcher acts (`NFPROTO_UNSPEC`)
  - `match`: the function pointer to the matching function
  - `matchsize` : the size of the matching function
  - `me`: field set to the macro `THIS_MODULE` if the match is intended to be compiled as such

A new target

Structure

- A new target function should be able to handle packets from every protocol it is registered for
- The main role of a target function is usually to either mangle the packet (f.i. TTL modifications) or to collect statistics
- It is a good praxis to register targets with a name fully in capitals, in order to distinguish them from the matching modules
- Remember to recompute checksums in case the packet has been mangled or it will not be considered valid afterwards

A new target

Registering the target

- A new targeting rule can be registered and unregistered via the `xt_register_targets` and the `xt_unregister_targets` functions
- Both functions accept a `xt_target` structure containing:
  - `name`: the string which will be recognised after the `-m` option of the `iptables` command
  - `revision`: the version of the matcher
  - `family`: the family of protocols on which the matcher acts (`NFPROTO_UNSPEC`)
  - `target`: the function pointer to the target function
  - `targetsize` : the size of the target function
  - `checkentry` : a function pointer to a sanity checker for target parameters
  - `me`: field set to the macro `THIS_MODULE` if the match is intended to be compiled as such