Dublin City University

Electronic and Computer Engineering

EE562 Network Stack Implementation

Notes

Author

Michael Lenehan michael.lenehan4@mail.dcu.ie

Student Number: 15410402

Contents

1 Intro

1.1 Userspace vs Kernel

- Modern processors have 2 modes of operation
  - Intel - Ring 0/3
  - ARM - Supervisor/User
- Code in Ring 0 > Priority than Ring 3
  - More instructions from set
  - More resource access
- Allow for improved security
- System code = kernel

1.2 Monolithic vs Microkernel

- Monolithic
  - 1 multi-tasking prog. implements OS funcitons
    * Add new functionality requires kernel rebuild
    * Linux, FreeBSD
- Microkernel
  - Most services run in userlan, min. code is privileged
  - Enhances modularity/maintainability at expense of performance
  - Mach, GNU Hurd, Minix 3
- Hybrid Kernel/Macrokernel
  - Most services run on microkernel on Ring 0
  - Windows > NT

1.3 Loadable Modules

- Dynamically loaded at runtime
- Access kernel structures/methods which have been exported

1.4 OSI Model Limitations

- OSI Model is approx. guide
  - Session/Presentation not in most networked apps
  - Tunneling can result in multi. stacked L3 protocols
  - Ethernet Swithces nominally operate at L2, supposed to be "data-link" protocol layer.
  - Control plane does not feature in model

1.5 Bit Rates and Packet Rates

Packet Rates depend on:

- Physical Bit Rate
- OH
  - Extra bits + by lower layers/protocol features
- Pkt length
- Physical BR vary from V.Low to V.High
- Useable BR < Nominal BR

– To inc. data transfer rate of X from Transport to higher layers, bit rate at phy layer must be $\gg X$

- Net SW must be written so net equip can operate at wire speed
  - HW and SW in net device rd/wr 2 pkts from link w/o dead time

1.6 Headers/Trailers

- Each pkt comprises of Protocol Data Unit and new layers head/tail
- Exception: Multiplexing, Fragmentation/Re-Assembly

1.7 Buffer Management

- Memory space ("socket buffer") req. to store data varies between layers
- Solutions:
  - Allocate new buffer
    * Buffer contents PBV, cp to larger buffer (with offset for header)
      · Lots of cp ops.

  - Oversized Linear Buffers (Linux)
    * How big skt buff is at lowest level implemented
    * Allocate buffer of this size, offset pointer to it by aggregate size of lower level headers
    * Good performance, breaches layering principle
  - Linked Minibuffers (BSD Unix)
    * Header/PDU/Tailer stored as linked list
    * Head/Tail + by inserting minibuffers at start/end of list
    * Memory-efficient, lower performance

2 Kernel Issues

2.1 Kernel Programming Issues

- Library functions not available
- Limited Stack - Small, fixed-size stack for each proc
  - kmalloc args 1=Size of block, 2=how kmalloc works
  - GFP_Kernel - kmalloc puts curnt. proc to sleep, waits for page when called in low mem situations
  - Funciton using GFP must be reentrant, otherwise use GFP_Atomic
- Kernel functions must be rentrant
- Linux kernel fully preemptible
  - kernel proc can be preempted by > priority proc
  - Should avoid global static vars in network code.
  - Need to use spinlock/semaphores to protect access to global structs
- Linux supports Symmetric Multiprocessor (SMP) architectures
  - Mutli. processors share common mem, potential issues
  - Harder debugging
  - Bugs cause catastrophic failure, require reboot

2.2 Memory Allocation in the Kernel

- Slabs ...

2.3 Kernel Scheduling

- Multi. Scheduling Classes
- Various policies within class
- Higher priority class served first
- Tasks can migrate between classes, policies, CPUs
- Completely Fair Scheduler
  - Decides which process is executed next
  - Uses red-black tree to decide on proc.
  - Implements Weighted Fair Queuing

Class	Policies
Stop	none
Deadline	SCHED_DEADLINE
Real Time (RT)	SCHED_FIFO, SCHED_RR
Fair (CFS)	SCHED_NORMAL, SCHED_BATCH, SCHED_IDLE
Idle	none

Class	Features
Stop	SMP-only, cannot be preempted, preemts all
Deadline	For periodic RT tasks
Real Time (RT)	for Posix "real-time" (low latency) tasks
Fair (CFS)	All other system tasks
Idle	Nothing to do - enter low power mode

2.4 Symmetric Multiprocessing (SMP)

- Like parallel processing
- Multi resources share single bus with multi. processors
- Contention for resources
  - N cores not N times faster than 1
  - Need to lock shared resources when in use

2.5 Reentrancy and Preemption

Preemption

- Stopping code to run other code

Reentrancy

- Preemting process might call preemted code
  - Preemnted code must support multi. execution instances

2.6 Writing reentrant code

- Avoid global vars, use automatic vars/kmalloc instead
- Use spinlock/semaphors to ensure global var access is atomic
- Do not call other functions, unless they are reentrant
- Accessing HW causes issues, use spinlock/semaphores to lock out other proc.
  - Causes bad RT performance
  - Fix - Disable interrupts
  - Most interrupt handlers do this

2.7 Spinlock

- Crude method to protect data struct
- Single int field as lock
  - Proc withing to access protected region must check lock val
  - If 1: proc. retries (spins) in tight loop of code
  - If 0: proc. enters region, sets value to 1
- Accessing mem location must be atomic - cannot be interrupted
- Winning proc is random/arbitrary

2.8 Semaphores

- Protect critical regions of code/data structs
- Queueing mechanism to give order to access
- Count
  - Num of proc waiting for resource
  - +: resource available
  - -/0: procs waiting
  - Initially set to 1
  - Requesting resource dec. count
  - Proc finished - inc. count
  - Abs val of count = num of sleepers
- Wait
  - Queue in which sleeping procs stored

2.9 HW Interrupts

- I/O Interrupts
  - Keyboard Press, USB bus has data, pkt arrives in NIC
- Timer Interrupts
  - Maintaining System Clock, Executing sched alg
- Interprocesspr Interrupts (IPI)
  - Only in SMP systems
  - On ARM processors
    * IPI_RESCHEDULE “Rescheduling interrupts“
    * IPI_CALL_FUNC “Function call interrupts“
    * IPI_CPU_STOP “CPU stop interrupts“
    * IPI_CPI_CRASH_STOP “CPU stop (for crash dump) interrupts“
    * IPI_TIMER “Timer broadcast interrupts”
    * IPI_IRQ_WORK “IRQ work interrupts”
    * IPI_WAKEUP “CPU wake-up interrupts”
    * Used for e.g. power management, moving procs between CPUs

2.10 Timer Interrupts

- Triggered by Programmable Interval Timer/High Precision Event Clock
  - ISR (interrupt service routine) implement jiffy clock, invoke proc scheduling
- SMP systems have local CPU timers
  - Used for profiling kernel code, timing procs.

3 Weighted Fair Queueing

3.1 Max-Min Fairness

- Purpose - Allocate fair share of resource among competing req.
- Suppose:
  - R units of resource
  - n requests
  - Fair allocation R/n?
  - Some requests for less?

  - How to apportion excess?
- Assume n requests for $d_1, d_2, \dots d_n$, with $D = \sum_{i=1}^{n} d_i$ and $d_j < d_j + 1$
- Assume $D > R$
  - $r_1 = min(D_1, \frac{R}{n})$
- Now remaining level of resource is $R - r_1$, allocate:
  - $r_2 = min(d_2, \frac{R-r_1}{n-1})$ etc.
- Two important Features:
  - Max num of req. met in full, starting with min request
  - Users which req more of resource can be allocaed by alg to receive same allocation
- Equivalent Algorithm:
  - Divide R evenly among req.
  - For those req. which have received more than required:
    * Divide excess evenly among outstanding req.
  - Repeat until no excess resources to redistribute

3.2 Quality of Service Guarantees

- FCFS doesn't guarantee BW share to flow
- Use Per-Flow queueing
- Principle - Generalised Processor Sharing
  - Assume traffic on flow is a fluid, can be continuously subdivided, is possible to offer service to infinitesimal quantity of traffic on a flow
  - Can devise a scheme to give exactly req. proportion of BW to each flow
  - Practical systems, pkts indivisible, can only approximate

3.3 Generalised Processor Sharing

- Aka Fluid-Flow Fair Queuing
- Offers max-min fiar share of BW to each flow
- When all flows busy, each gets req. share of avail. BW
- When only some flows busy BW share of idle flows allocated in max-min fair way to backlogged flows
- Let $S_i(t, t+\delta t)$ be amnt data from flow i served in interval $[t, t+\delta t]$
- If flow i backlogged during interval, for any flow j:
  - $\frac{S_i(t,t+\delta t)}{S_j(t,t+\delta t)} \geq \frac{\delta_i}{\delta_j}$

3.4 Weigthed Fair Queuing

- Round Time units - bits served per flow
- As no. backlogged flows changes, actual time taken to serce bit changes in inverse proportion
- So far giving each flow equal BW share
- Let $F_{i,k}$ and L_i, k be the finish/length resp. of the k-th pkt to arrive on flow i
- Let $R_{i,k}$ be the round in which it arrived. Evidently:
  - $F_{i,k} = max(F_{i,k-1}, R_{i,k}) + L_{i,k}$
- Problem: calculation of $R_{i,k}$
- Need to know relationshiop between R(t) and t when pkt arrives
- In principle:
  - $R(t_\delta t) = R_(t) + \frac{\delta t}{B_(t)}$
  - if B(t) is constant in interval $[t, t + \delta t]$
  - Need to keep track of when B(t) changes
    - When pkts arrive/depart
- Let E(t) be time of last ecent before time t (arrival/depart)
- R(E(t)) is round no. when last event occurred.
- Suppose that at time t, pkt arrives, then:
  - $R(t) < -R(E(t)) + \frac{t-E(t)}{B(t)}$ //Update value of R(t)
  - If pkt arrives into empty buffer:
  - $B(t) < -B(E(t)) + 1$ // Now 1 more backlogged flow
  - Else
  - $B(t) < -B(E(t)) endif$

– E(t) <- t

- Suppose that at time t, pkt departs GPS discipline then:
  - $R(t) < -R(E(t)) + \frac{t-E(t)}{B(t)}$
  - If pkt leaves behind empty buffer then:
  - $B(t) < -B(E(t)) - 1$
  - else
  - B(t) <- B(E(t))
  - end if
  - E(t) <- t
- If all calc done when pkt arrives, alg is:

1. Calc tentative value for R(t) (assuming no departures since E(t)
2. Find pkt with min finish time F_{min} and check if $F_{min} < R(t)$

- If so:
  - Calc its time of departure t as $E(t) + B(E(t))(F_{min} - R(E(t)))$
  - Process departure as described earlier ($R(t) = F_{min}$)
  - $E(t) < -t$
  - Go back to step 1
- Otherwise:
  - Process newly arrived cell as described earlier.

3.5 Weighting the Queues

- Redefine B(t) as:
  - $B(t) = \sum_{allFlows(i),activeAtTime(t)} N\phi_i$
- Service time in rounds of pkt is now $\frac{L_{i,k}}{N\phi_i}$
- Finish time calc as:
  - $F_{i,k} = max(F_{i,k-1}, R_{i,k}) + \frac{L_{i,k}}{N\phi_i}$
- Since N acting as scaling factor, can set to unity:
  - $B(t) = \sum_{allFlows(i),activeAtTimet} \phi_i$
  - $F_{i,k} = max(F_{i,k-1}, R_{i,k}) + \frac{L_{i,k}}{\phi_i}$

4 Socket Buffers

4.1 Socket Buffers in Linux

- sk_buff struct defines doubly-linked list of socket buffers with extra struct to allow head of list to be found quickly
- Various functions available to manage socket buffers
- Data space created in Linux sk_buff comprises:
  - Headroom
  - Data
  - Tailroom
- Idea - Pkt being tx, amnt space reserced eq. that req. at L2, even though sk_buff being created in Transport Layer

- Extra head/tailroom reserved for anticipated req. of lower-layer headers and trailers.
- When pkt rx from higher layer, head/tailroom encroached upon to create extra mem. space required.
- When created by alloc_skb(), sk_buff is "all tail".
- skb_reserve() is called in to reserve headroom
- Space for data created using skb_push() (bite into headroom) or skb_put() (bite into tailroom)
- sk_buff will be queued in doubly-linked list by protocol handler
- Following atomic func. avvailable to manipulate such lists (each list assoc. with socket):
  - skb_queue_head() - Places buffer at start of list
  - skb_queue_tail() - Places buffer at end of list
  - skb_dequeue() - Takes first buffer from list, returns pointer to sk_buff or Null if queue empty
  - skb_insert(), skb_append() - Place sk_buff before/after specific buffer in list. Useful for pkt resequencing in TCP
  - kfree_skb() - Releases a buffer
  - skb_unlink() - Extracts sk_buff from the list in which it is stored (w/o needing to know list identity) does not free it.
  - skb_clone(), skb_copy() - Makes a copy of an sk_buff
    * Former doesn't copy data area (thus has much lower OH than skb_copy())
    * Disadvantage is that data area in cloned buffer is read-only

4.2 Non-Linear Socket Buffers in Linux

- Two forms of non-linear buffers supported:
  - Paged Buffers
    * Large buffers may be neededif L2 supports e.g. 64kB frames, or if using some HW acceleration methods
  - Fragmented Buffers
    * When pkt is being reassembled from its fragments, data is assembled using linked list, to avoid buffer copying
- data_len records pages in socket buffer, and so is 0 for "ordinary" linear buffers

4.3 Hack for 64-bit Systems

- In moving from v2.6.21 -> .22 of kernel, hack used to reduce size of skb_tail and skb_end to 4 from 8 bytes.
  - Change never documented.
- Tail and End are now offsets, specifically the no. of bytes between head and tail of bff.
- transport_header, network_header, and mac_header fields in struct now offsets, rather than pointers.
- No longer safe in versions > 2.6.22 to directly access these fields
  - Inline functions defined to compute relevant values safely

5 NIC Drivers

5.1 Linux Network Device Drivers

5.2 Initialisation

5.3 Packet Reception

5.4 eth_type_trans()

5.5 Reaching Layer 3 in Linux

6 L2 -> L3

6.1 Network RX softirq

6.2 Offload to the IP Packet Handler

6.3 Read-Copy-Update (RCU)

6.4 Bridge Handling

6.5 NAPI Drivers

7 Netfilter

7.1 Packet Transmission

7.2 Netfilter

7.3 IP Stack in Linux Kernel

7.4 Using the netfilter hooks

8 Netfilter Hooks

8.1 The IP Stack Interface to Netfilter

9 Packet Processing

9.1 The Forwarding Information Base

9.2 Aside: MPLS

9.3 Longest Prefix Match and LC Tries