Linux Network software

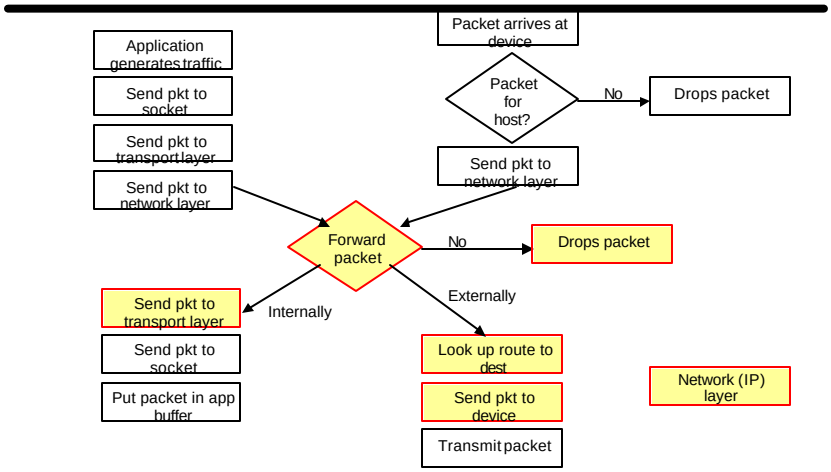
Acknowledgements due to Stephen Hailes, Saleem Bhatti, Cecilia Mascolo from UCL CS

Unix system introduction

- We will be dealing with the way that Unix works (most particularly those Unixes with net code derived from BSD e.g. SunOS 4.x, SVR4, AIX 3.2)

- At a user level this is through the *socket* interface (actually, there is an alternative – TLI aka XTI X/Open Transport Interface)

Linux overview



Network drivers

- For a long time, OS have provided a standard abstraction/interface for classes of device.
- Unix traditionally divides devices into 2 classes
  - Chararcter (low rate, interactive, serial line typically)
  - Block (Disk, Display, etc)
- Its possible to *squeeze* network devices into the block mode paradigm, but it's messy
- Linux adds a 3rd type of device - network.

Device API

- Typically, device has name to place it in the file namespace, but also has identifier – unix has major/minor numbers
- Driver is a structure (class?) with a set of entry points (functions/methods)
- At boot (or module load) time, the device is initialised by calling its init() function – this resets the device, and installs any relevant interrupt handlers and so on….it then registers with the OS…
- Rest of time, we manage i/o with device with open, close, queue_xmit, and interrupts/notifications

Device files are found in the */dev* directory. Each device is assigned a major and minor device number. The major device number identifies the type of device, i.e. all SCSI devices would have the same number as would all the keyboards. The minor device number identifies a specific device, i.e. the keyboard attached to *this* workstation.

Device files are created using the *mknod* command.

Internals

- Device driver manages specifics like
  - Bus interface/memory/I/o address of device registers
  - DMA and timer chip use
  - IRQs, etc
- Notice asymmetry of input and output – output is requested, whereas input arrives unexpectedly
- Input results in packets being queued, and netif_rx() called to find out which higher level protocol function to dispatch

Bridge, Route, Filter

- What if packet is “not for us”?
- Basically, will either bridge, route, or discard
  - Bridge is intensive (requires promiscuous ether interface – expensive in packet discard!)
  - Route is part of linux and bsd unix – requires forwarding table, and prob. 1 routing protocol process to build and maintain it
  - Discard – most common case! Requires efficient handling – lots of good work on efficient filtering (berkeley packet filter – see papers!)

Book: Network implementation

Jon Crowcroft & Iain Philips

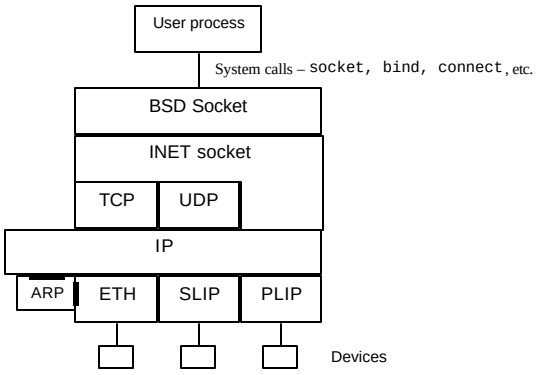
TCP/IP & Linux Protocol Implementation:
Systems Code for the Linux Internet

1st edition (October 15, 2001)
John Wiley & Sons; ISBN: 0471408824

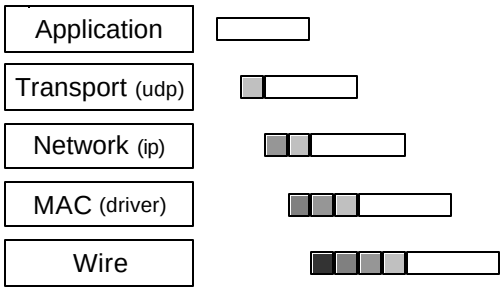
Introduction

- Now we're going to look at system level details of UNIX networking.
  - Assume Net/3 – like approach e.g. BSD sockets
  - However, code will be from Linux – kernel version 2.4.14) – there are some differences in implementation.
- Socket data structures
- sk_buf (Linux) (? mbuf (Net/3)) and a brief look at transmission.
- Routing (forwarding) DS & code

Layering


User process
System calls – socket, bind, connect, etc.
BSD Socket
INET socket
TCP
UDP
IP
ARP
ETH
SLIP
PLIP
Devices

Application to wire (and v.v.)



User level code

See sheet 1

Overview -- output

- Send-type routines are normally blocking
  - Data gets passed to the appropriate lower level transport code, based on the fd.
    - See net/socket.c::sock_sendmsg, net/ipv4/af_inet.c::inet_sendmsg
  - This runs the state machine for that protocol and then passes code on to IP level
    - See e.g. net/ipv4/udp.c::udp_sendmsg
  - This deals with routing, fragementation, etc. adds appropriate IP header and queues for output
    - See net/ipv4/ip_output.c::ip_build_xmit
    - See net/ipv4/ip_output.c::ip_fragment
    - See net/ipv4/ip_output.c::ip_queue_xmit
  - Actually these may be deferred to allow better use of resources– need a network scheduler (or actually several levels of scheduling)

Overview -- input

- Receive involves coordinating a synchronous call and an asynchronous packet arrival
  - Hardware determines if packet is for us, and generates interrupt if it is.
  - ISR in device driver is called, pulls packet off device and determines which type of packet it is.
  - Network level – check input, perform reassembly, determine whether to reroute, etc.
    - net/ipv4/ip_input.c::ip_rcv
    - net/ipv4/route.c::ip_route_input
    - net/ipv4/ip_input.c::ip_local_deliver
  - Transport level – check checksums, update local state machine, and demux to individual socket.
    - net/ipv4/udp.c::udp_recvmsg

Important files – so far

- There are lots and lots of important ones, but for now….
- .h files
  - include/linux/[net.h, udp.h, tcp.h]
  - include/net/[socket.h, sock.h, udp.h, tcp.h]
- .c files
  - net/socket.c
  - net/ipv4/[af_inet.c, udp.c tcp.c tcp_output.c tcp_input.c, tcp_ipv4.c tcp_timer.h]
  - net/core/sock.c

struct socket

See sheet 2

sock structure

include/net/sock.h

- struct sock is messy:
  - Bits of it are to do with TCP – in fact the whole of the networking code is a bit of a jumble, with TCP data appearing at the network layer.
  - Since we don't have time to look at TCP, figuring this part of it out is an exercise for the reader.
  - It is likely to be tidied up in future versions of Linux (and is now a lot better than it was in earlier versions)

See sheet 3

struct sk_buff

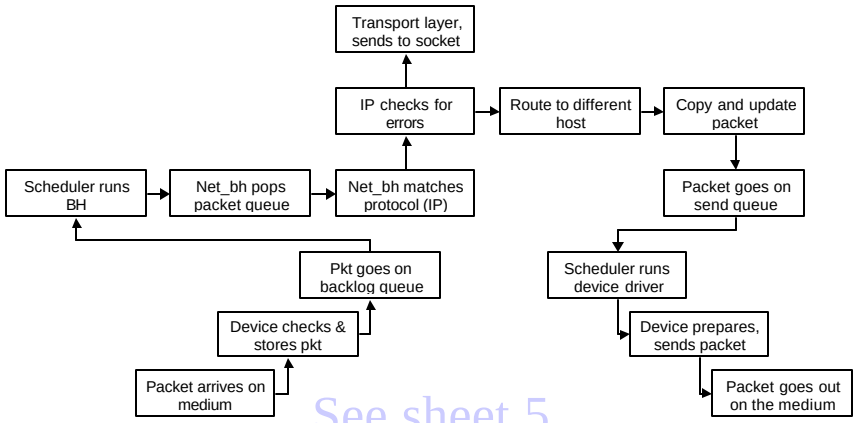
- The task of the sk_buff is to manage individual packets, their payloads and their headers. You must understand it to understand the networking code.
  - (actually it does more than this, but we'll ignore that for now)
- They have an equivalent in Net/3 code, the mbuf, which is described in Stevens, but they are different.
- There is a producer-consumer chain where the buffer is allocated by the producer (be this the driver for input or the transport for output) and freed by the consumer.
- There is only one copy of the buffer ever in existence

See sheet 4

Routing

- Two main functions:
  - Forwarding
    - Carried out on every packet – look in forwarding table to determine destination and output interface.
  - Routing
    - Build and maintain forwarding table. Done asynchronously, usually by a user space process.

Forwarding block structure



See sheet 5

Forwarding in Linux

- There are 3 structures of interest:
  - The neighbour table
    - include/net/neighbour.h::neigh_table
    - In effect, this is an ARP cache:
      - It only contains information for machines that are physically connected to ours
      - That info eventually vanishes, unless hardwired by an admin.
  - The FIB table
    - This is the main routing table, which contains details of how we forward packets to any address. More later.
  - The routing cache – smaller and faster.
    - Caches info obtained from recently routed packets.
    - The info times out if not used.

Class based addresses

- Before we look at routing in detail, we need to understand something about addressing, subnetting and aggregation.
- Back to basics:
- Class A `0NNN NNNN HHHH HHHH HHHH HHHH HHHH HHHH`
  `0.0.0.0 to 127.255.255.255`
- Class B `10NN NNNN NNNN NNNN HHHH HHHH HHHH HHHH`
  `128.0.0.0 to 191.255.255.255`
- Class C `110N NNNN NNNN NNNN NNNN NNNN HHHH HHHH`
  `192.0.0.0 to 223.255.255.255`
- Class D `1110 MMMM MMMM MMMM MMMM MMMM MMMM MMMM`
  `224.0.0.0 to 239.255.255.255`
- Class E `1111 0XXX XXXX XXXX XXXX XXXX XXXX XXXX`
  `240.0.0.0 to 247.255.255.255`

…and their problems

- `network.host` form is
  - too inflexible
  - Wasteful – e.g. class A addresses have 2^{24} hosts on a single network!
- We want multiple levels of hierarchy

Subnetting

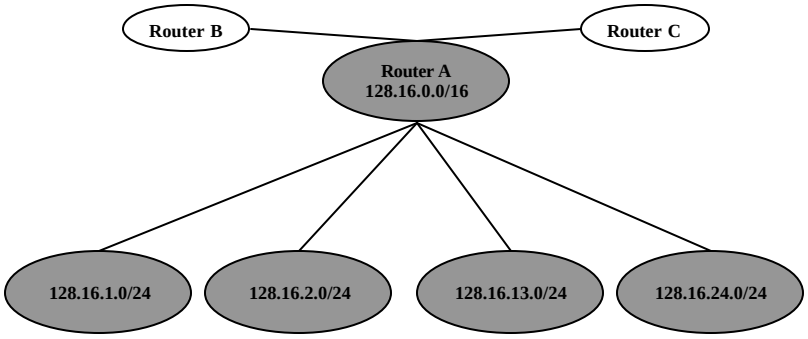
- All very well, but what happens when you want to split up your address allocation amongst smaller administrative components.
  - E.g Take a Class B address 128.16.0.0
  - We could split this up into a number of class C networks
  - We would have, in effect, addresses of the form:

```
......NETWORK...... ........HOST.......      CLASS B ADDRESS
1000 0000 0001 0000 SSSS SSSS HHHH HHHH      BUT WE USE SUBNETS
NNNN NNNN NNNN NNNN NNNN NNNN HHHH HHHH      IN EFFECT
   255   .   255   .   255   .    0          SUBNET MASK OR /24
```

  - NB the first subnet address is the net identifier, the last is for broadcast. First usable address is normally router.
  - Could do others, e.g. /20 gives subnets of 4094 machines

Aggregation

- We do not have to advertise each subnet individually: B and C only need one route.



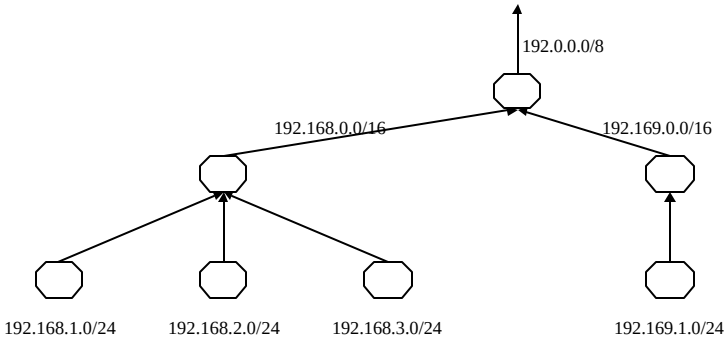
…cont

- In older routing protocols e.g. RIPv1, routing updates do not include subnet masks.
  - Thus a router must assume that the subnet mask it has been configured with is valid for all subnets. i.e. a single mask must be used for all subnets within a network.
- No longer true – since mid 1993 we've had Classless Interdomain Routing (CIDR).
  - Newer routing protocols (e.g. RIPv2, OSPFv2, BGPv4, etc) can deal with this.
  - *FORGET EVERYTHING I JUST SAID ABOUT THE (CLASS-BASED) 'NETWORK' AND 'HOST' SEPARATION*
    - a routing table entry is indexed on a combination of address and mask
  - Not only can we break networks into subnets, but we can combine networks into *supernets*, so long as they have a common network prefix.

CIDR

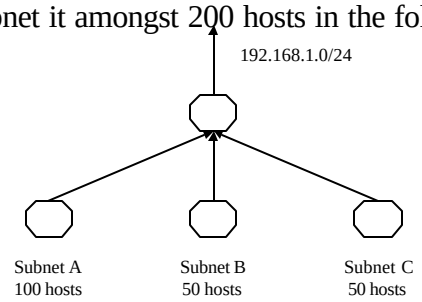
(RFCs 1518, 1519, 1466, 1447)

- If you summarise any block of routes with a subnet mask smaller than the matching class of the address, you are *supernetting*.



Variable Length Subnet Masks

- This goes hand-in-hand with variable length submasks (actually VLSM preceeded CIDR).
- Assume we have a class C address: 192.168.1.x and we want to subnet it amongst 200 hosts in the following way:



VLSM cont

- Our problem is that our possible masks are:
  - /25 giving 2 subnets with 126 hosts in each
  - /26 giving 4 subnets with 62 hosts in each
- Neither is any good.
- We need to use different masks for each subnet
  - Use /25 for subnet A
  - Use /26 for subnets B and C

  - A = 192.168.1.0/25
  - B = 192.168.1.128/26
  - C = 192.168.1.192/26

CIDR vs VLSM

- CIDR and VLSM are essentially the same thing, since each is about allowing a portion of the IP address space to be repeatedly divided into smaller and smaller pieces (aka recursion).
  - Both approaches require that the extended network prefix information be provided with each route advertisement.
  - The key difference between VLSM and CIDR is a matter of where recursion is performed:
    - In VLSM the subdivision of addresses is done after the address range is given to the user.
    - In CIDR the subdivision of addresses is done by the Internet authorities and ISP before the user receives the addresses.
  - Both approaches use longest matching for addresses

Longest match

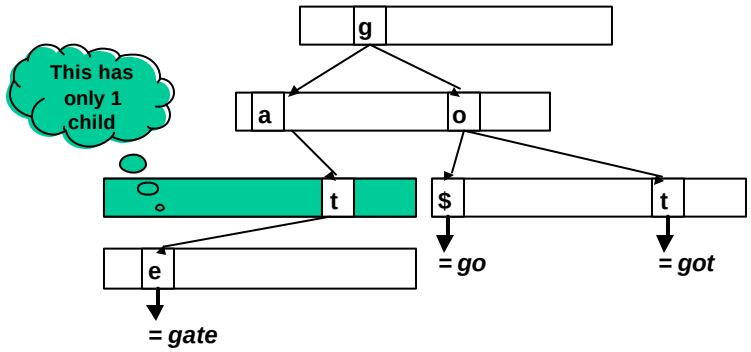
- We have a situation in which we have variable length masks in a routing table.
- Pick the routing table entry that is closest to the address we want => need a longest match algorithm
- e.g.
  - 128.0.0.0/8 via route A
  - 128.1.0.0/16 via route B
  - 128.1.1.0/24 via route C
- Where do we send
  - 128.1.0.1
  - 128.1.1.1
  - 128.2.1.1
- Note that e.g. 128.1.1.1 matches all three rules but it MUST be accessible via route C, else it will never get any packets => need to assign addresses with care.

Alternatives for IP lookups

- Hardware – Content Addressable Memory (CAM)
  - Present e.g. IP destination and get back next hop
  - Like a TLB. Expensive.
- Protocol-based approaches
  - IP and tag/layer 3 switching (e.g. MPLS)
  - Similar to VCID in circuit switched nets (and may use it!)
  - Requires separate label distribution protocol to specify address/tag mapping
  - Basically, use IP pkts and IP routing as signalling for circuit set-up
  - Faster algorithms call this into question.
- Software…

Data structures -- tries

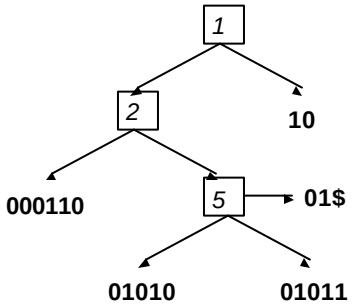
- Tries: an m-ary tree structure. e.g. 26 chars + ‘end of word’



- Very heavy on space for sparse keyspace where most nodes have only 1 descendant

Patricia trees (4.3 Berkeley Reno)

- Binary trie, but with *'path compression'*



- See http://www.cs.berkeley.edu/~sklower/routing.ps

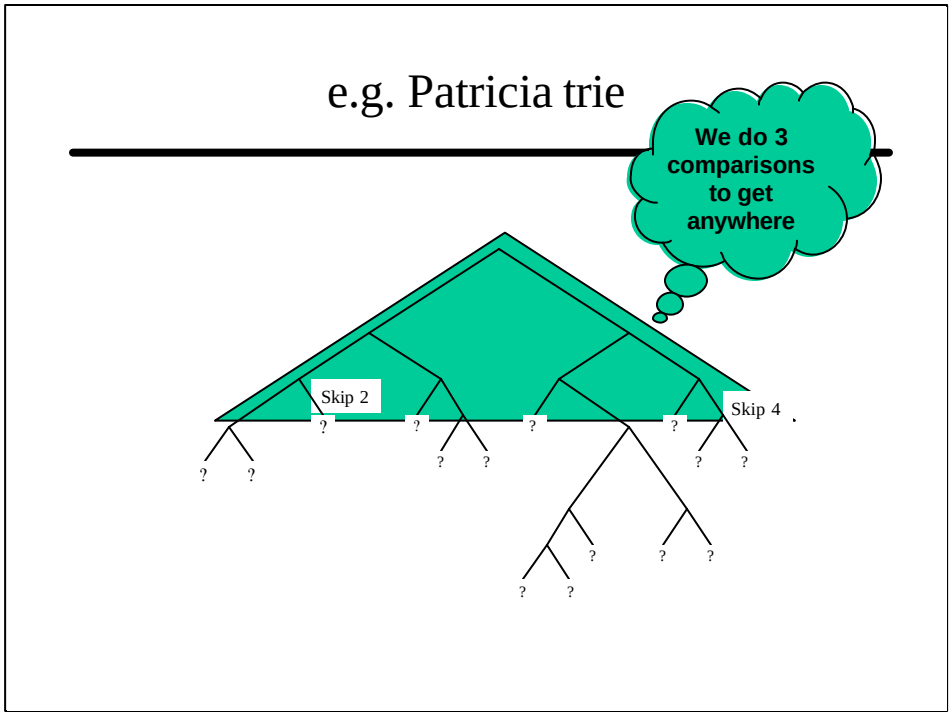
LC tries

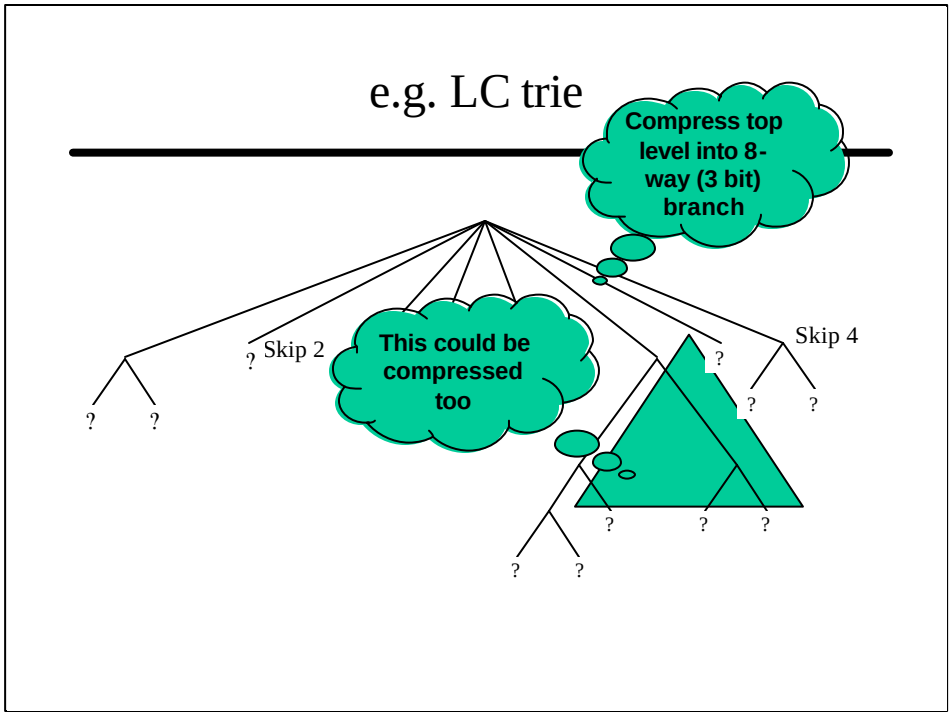
- LC tries are really Patricia trees with *'level compression'*
  - Path compression helps compress parts of the tree which are sparsely populated.
  - Level compression helps with parts of the tree that are densely populated. It's a bit like going back to standard m-ary tries for parts of the structure.
- Instead of having a binary tree, make it a m-ary tree (m is a power of 2) for some levels in the true, where this helps.
- http://citeseer.nj.nec.com/nilsson98fast.html

Example

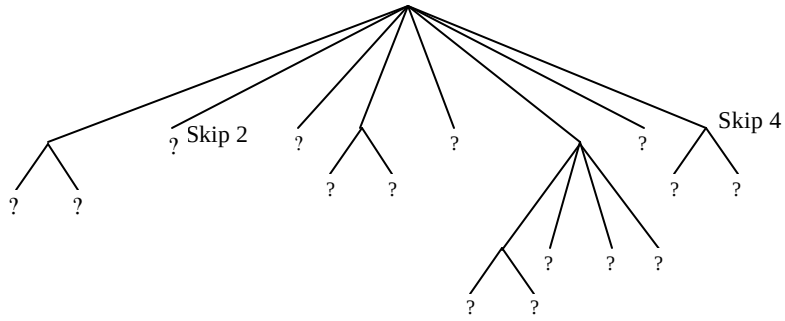
- So, imagine we have the following strings to enter:

? 0000	? 0110	? 101001	? 110
? 0001	? 0111	? 10101	? 11101000
? 00101	? 100	? 10110	? 11101001
? 010	? 101000	? 10111	


e.g. Patricia trie
We do 3 comparisons to get anywhere
Skip 2
Skip 4
?


e.g. LC trie
Compress top level into 8-way (3 bit) branch
Skip 2
This could be compressed too
Skip 4
?

So, we get to…



In table form:

branch = 5 bits, skip = 7 bits, ptr =20 bits – 1 word per entry.

	Branch	Skip	Ptr
0	3	0	1
1	1	0	9
2	0	2	2
3	0	0	3
4	1	0	11
5	0	0	6
6	2	0	13
7	0	0	12
8	1	4	17
9	0	0	0

10	0	0	1
11	0	0	4
12	0	0	5
13	1	0	19
14	0	0	9
15	0	0	10
16	0	0	11
17	0	0	13
18	0	0	14
19	0	0	7
20	0	0	8

- Start at 0, start input at skip bits, take branch bits of it, and add these to ptr. If we get to an entry with a branch of 0, then it's a leaf.
- Stop & do full comparison

Other tree based algs

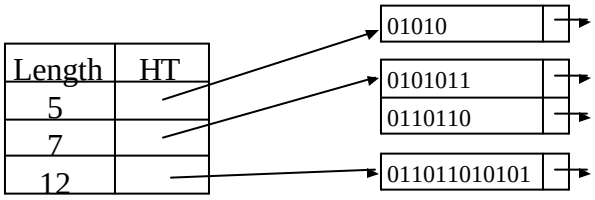
- Generalised level-compressed tree.
  - See e.g. 'Optimal Routing Table Design for IP Address Lookups Under Memory Constraints' -- Gene Cheung and Steve McCanne.
  - http://citeseer.nj.nec.com/267395.html

Hashing

- See ‘Scalable High Speed IP Routing Lookups’ by Marcel Waldvogel et al.
  - http://citeseer.nj.nec.com/waldvogel97scalable.html
- It is possible to find hash functions whose computation is lower cost than a memory access – can we exploit this?
  - Note that access to a trie requires a number of accesses, depending on the amount of level and path compression.
- We’ll increase complexity gradually.

Linear search of hash tables

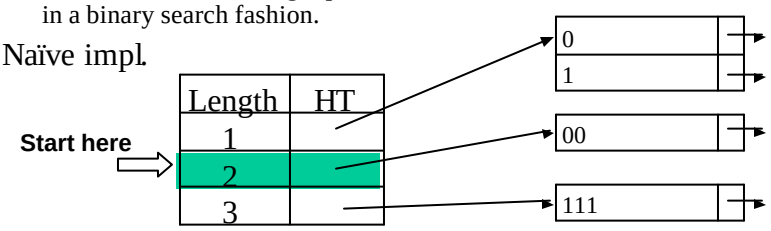
- First, examine linear search of hashing tables:
  - Have a series of hash tables, one for each network prefix length we know about.
  - In the worse case for IPv4 this will be 32, for IPv6 it'll be 128.



  - Lookup in longest length prefix table (i.e. 12) on a key that's the first 12 bits of the address. If a match, OK.
  - If not, pick next longest (i.e. 7) and try again with a 7-bit key

Binary search of hash tables

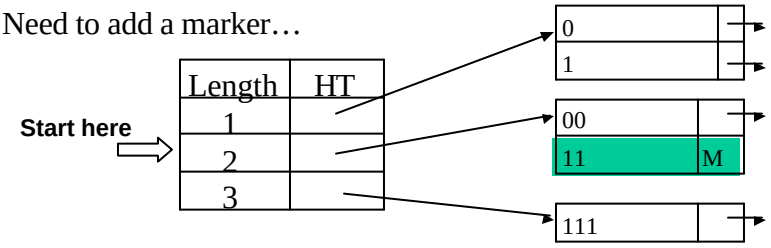
- General idea:
  - Start somewhere in the middle of the table (or, perhaps, with the most popular prefix length)
  - If we match, search longer prefixes. If we fail, search shorter ones in a binary search fashion.
- Naïve impl.



- Search for 111. Problem – no match.
- We don't know that we should search bottom half of table, so…

Binary search of hash tables

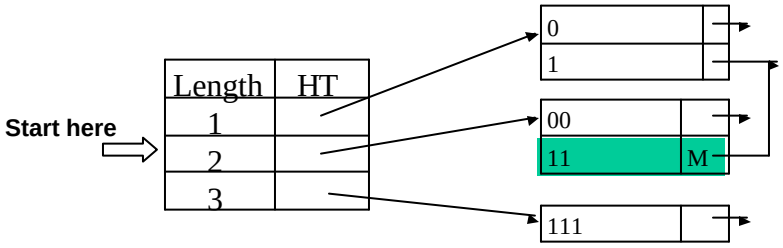
- Need to add a marker…



- Searching for 111, we find the marker, which tells us to search bottom half of table, then we find what we want.
- But what if we're searching for 110x xxxx xxxx xxxx etc.?
  - We find the marker and search bottom half, which is wrong.
  - Our match is 1
  - Need to backtrack – messy

Binary searching of hash tables: precomputation

- When marker is inserted into table, tag it with the value of the best matching prefix of marker M already in the table.



- Remember best matching prefix so far – when we search for 110x, find marker and remember pointer to HT for '1'.
- Search lower half and don't find 110 ? return stored value