Encapsulation and Tunneling

- *encapsulation* describes the process of placing an IP datagram inside a network packet or frame
- encapsulation refers to how the network interface uses packet switching hardware
  - two machines communicating across IEEE 802.3 using IP encapsulates each datagram in a single Ethernet packet for transmission
  - the encapsulation standard for TCP/IP specifies:
    - that an IP datagram occupies the data portion of the IEEE 802.3 packet
    - the IEEE 802.3 packet type must be set to IP

Tunneling

- by contrast, the term *tunneling* refers to the use of a high level transport service to carry packets or messages from another service
- the key difference between tunneling and encapsulation lies in whether IP transmits datagrams in hardware packets or uses a high level transport service
- *IP encapsulates each datagram in a packet when it uses hardware directly*
- *it creates a tunnel when it uses a high level transport delivery service to send datagrams from one point to another*

Performance implications of data transmission on high speed network devices

- 10 Gbits/sec Ethernet has been standardised
  - and 10 Gbits/sec cards are becoming available and are supported on
  - Linux/Windows/FreeBSD ⟨`http://www.myri.com/Myri-10G/10gbe_solutions.html`⟩
    - the IEEE committee is now investigating 100 GBit/sec!
- the 10 Gbits/sec card above has a 2 MB buffer and uses the PCI-Express bus
  - claims to run at line speed

Performance implications of data transmission on high speed network devices

- effectively 10 bits transmitted every nano second
- huge strain on the microprocessor and memory system as it must move ˜1 GByte ram/sec across to the card - for sustained performance
- or put another way, assuming a single core is clocking at 3.33 GHz and that a core can execute an instruction every cycle then
- every instruction used to configure the next transmitted packet will incur a 3 bit latency delay
- what we are seeing is the network speed becoming faster than CPU speed
  - not seen this for 20 years
- the free lunch is over! ⟨`http://www.gotw.ca/publications/concurrency-ddj.htm`⟩
  - over the last 30 years we have seen microprocessor speeds increase

Microprocessors and their transistor counts

Processor	Transistor count	Date	Manufacturer
Intel 4004	2,300	1971	Intel
Intel 8008	3,500	1972	Intel
MOS Technology 6502	3,510	1975	MOS Technology
Intel 8080	4,500	1974	Intel
Intel 8088	29,000	1979	Intel
Intel 80286	134,000	1982	Intel
Intel 80386	275,000	1985	Intel
Intel 80486	1,180,000	1989	Intel

Microprocessors and their transistor counts

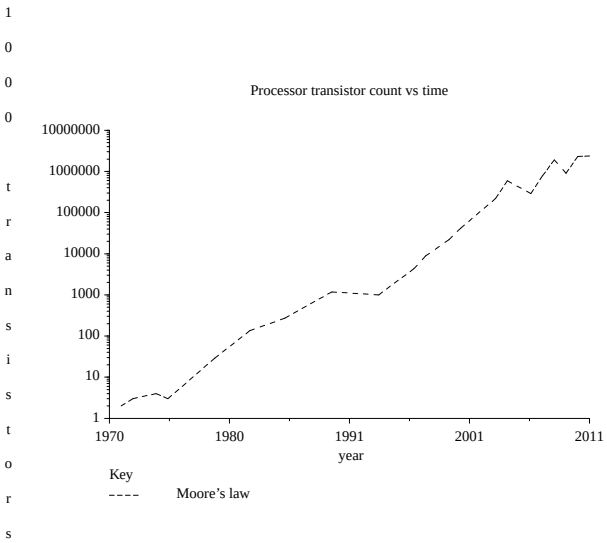
Processor	Transistor count	Date	Manufacturer
Atom	47,000,000	2008	Intel
Barton	54,300,000	2003	AMD
AMD K8	105,900,000	2003	AMD
Itanium 2	220,000,000	2003	Intel
Cell	241,000,000	2006	Sony/IBM/Toshiba
Core 2 Duo	291,000,000	2006	Intel
AMD K10	463,000,000	2007	AMD
AMD K10	758,000,000]	2008	AMD

Microprocessors and their transistor counts

Processor	Transistor count	Date	Manufacturer
Itanium 2 with 9MB cache	592,000,000	2004	Intel
Core i7 (Quad)	731,000,000	2008	Intel
POWER6	789,000,000	2007	IBM
Six-Core Opteron 2400	904,000,000	2009	AMD
Six-Core Core i7	1,170,000,000	2010	Intel
POWER7	1,200,000,000	2010	IBM
z196	1,400,000,000	2010	IBM
Dual-Core Itanium 2	1,700,000,000]	2006	Intel
Six-Core Xeon 7400	1,900,000,000	2008	Intel
Quad-Core Itanium Tukwila	2,000,000,000	2010	Intel
8-Core Xeon Nehalem-EX	2,300,000,000	2010	Intel
16-Core Sparc T3	2,400,000,000	2011	Oracle

Microprocessors and their transistor counts

- the free lunch is over! ⟨`http://www.gotw.ca/publications/concurrency-ddj.htm`⟩
  - over the last 30 years we have seen microprocessor speeds increase



Moore's Law

- interestingly it states the no of transistors will double every 18 months
  - actually fairly true if altered 24 months
- does ***not*** say that performance doubles every 18 months!
  - although this has been a by product of transistors doubling, so far..
- Moore's Law continues to hold, but microprocessor manufacturers are using transistors in different ways
  - they cannot keep increasing clock speed as was done over the last 30 years
  - thus multicore microprocessors are available (currently 16 core)
  - large cache and maybe GPU on chip
  - expect no. of cores to double every 24 months

Moore's Law

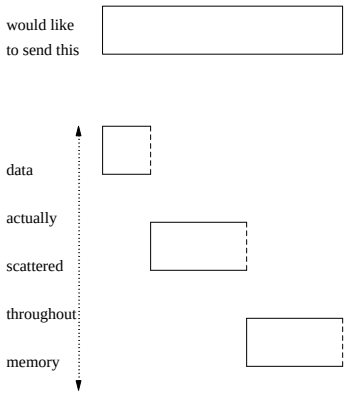
- implications for the Computer Science community are immense
  - strange irony that one of the oldest scripting languages will make harness these multicores with no extra user level programming
    - bash!
- we need to ensure that we reduce the amount of data copying within the operating system to a minimum
  - it is critical to keep to a minimim the number of instructions executed when configuring the device driver hardware to transmit/receive the next packet
- finally extra reading - check the conclusion ⟨`http://www.hep.man.ac.uk/u/rich/net/nic/GE_FGCS_v18.doc`⟩

System and user organization of frames, packets, headers and data

- network programming often requires disjoint data to be transmitted in a single unit

System and user organization of frames, packets, headers and data

-



User level scatter/gather technique

- could copy data - but excess copying is *slow*
- a common user level technique is to use an `iovec` and `readv`, `writev`
-

```
#include <sys/types.h>
#include <sys/uio.h>

int writev (int fd, struct iovec iov[],
            int iovcount);
int readv  (int fd, struct iovec iov[],
            int iovcount);
```

- similar mechanism exists at the user and system level in UNIX
  - see `iovec`, `readv`, `writev`

User level scatter/gather technique

- the iovec is defined under linux as:
-

```
struct iovec {
   void *iov_base;
   int iov_len;
};
```


iovec example

- consider a function that needs to transmit header and data

iovec example

-

```
int write_with_hdr (int fd, void *buff, int nBytes)
{
   struct hdr header;

   /* set up header ... as required */

   if (write(fd, &header, sizeof(header)) !=
       sizeof(header)) {
      return( -1 );
   }
   if (write(fd, buff, nBytes) == nBytes) {
      return( nBytes );
   } else {
      return( -1 );
   }
}
```

iovec example

- requires two write system calls, again slow.
  - could copy - but again slow
- remove two calls to write and the need to copy by:

Using writev and iovec

```
int write_with_hdr (int fd, void *buff, int nBytes)
{
   struct hdr header;
   struct iovec iov[2];

   /* set up header ... as required */

   iov[0].iov_base = (void *)&header;
   iov[0].iov_len  = sizeof(header);
   iov[1].iov_base = buff;
   iov[1].iov_len  = nBytes;

   if (writev(fd, &iov, 2) ==
       sizeof(header) + nBytes) {
      return( nBytes );
   } else {
      return( -1 );
   }
}
```

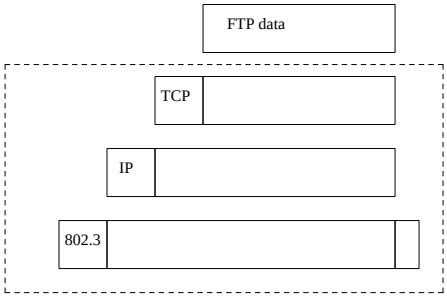

User and System interface

- user level iovecs remove redundant copies and multiple writes
  - useful to improve speed
  - maybe necessary if data must be written atomically
    - ie to a network device
- what happens when the system call occurs and the operating system needs to add further headers?
  - could copy iovecs and create a new iovec
    - not a good idea as outgoing data (writes) may need a number of headers
  - require an efficient method of adding and removing headers and data

Internal organization of frames, packets, headers and data

- one aspect seldom covered in network books
  - protocol implementation
  - difficult and large subject
- specific topics are useful to examine as an overall aid to understanding
  - how is data passed between protocol layers?
  - in particular remember that the data component of lower layers contains headers (control information) of the higher

Protocol encapsulation



- only concerned with packets/frames within dotted box
  - going up the layers we need to strip off the headers (receiving a packet)
  - going down the layers we need to add headers! (might be harder!)

Buffer management

- incoming packets must be placed in memory and passed to the appropriate protocol software for processing
- applications generate output which must be stored in packets and passed to software and hardware devices for transmission
- ultimately the efficiency of protocol software depends on how it manages memory
- a good design allocates space quickly and avoids copying data

Outgoing traffic implementation issues - method 1

- create frame header
  - could copy frame data
  - generally considered bad as copying data is slow
  - advantage, clear and simple
- raises two questions
  - how do we allocate buffers
  - surely there is a better way!

Buffer allocation

- ideally a system could efficiently allocate buffers by using buffers of the same size
  - difficult to choose the optimum packet size
  - 1 computer might be attached to two networks
    - each has a different optimum size of packet
  - may wish to add connections to a computer without changing system buffer size
  - IP may need to store datagrams larger than underlying network packet size (reassembly)
  - applications may wish to send arbitrary sized messages

Large buffer solution - method 2

- could choose buffers capable of storing largest packets
  - advantage, it works and is simple
  - disadvantage IP datagrams can be 64k
  - large datagrams are rare
    - large amounts of memory are wasted
- in practice such a solution is often adopted but at a size of 4 or 8k + size of physical network layer header
- no problem in our dotted boxed system but we would have to copy data to/from the user
  - ie out of system buffers into user space

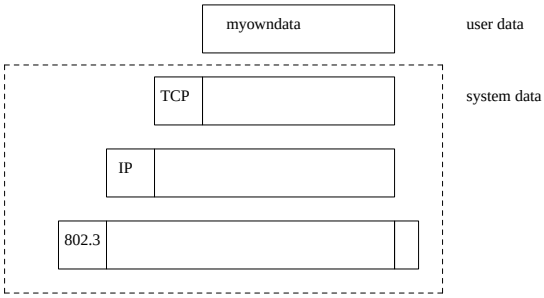
Problem with large buffer solution

- maybe a better solution can be found which avoids this
-

```
    char myowndata[lengthOfData] ;

    nbytes = write(tcpSocket, &myowndata,
                   lengthOfData);
}
```

-



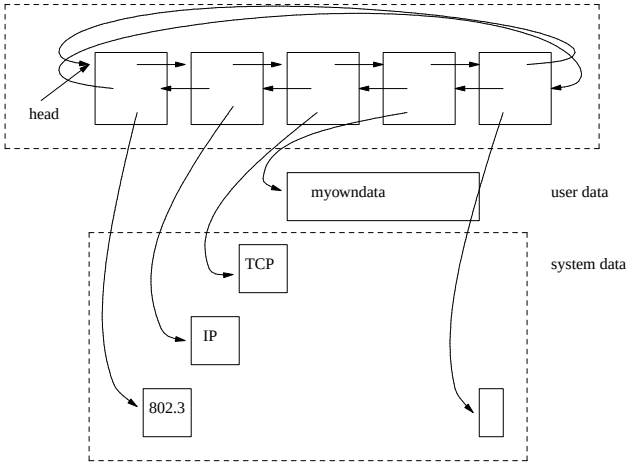
Linked list solution - the mbuf - method 3

- the alternative in to use linked lists of smaller buffers
  - buffers may be of fixed or variable size usually small between 128..1024 bytes
  - BSD Unix uses 128 bytes in a structure called (`mbuf`)
- individual buffers do not need to be full of data
  - they contain a very short header which define the
    - length of data buffer
    - amount of real data
    - where the data exists (address of data)

Linked list diagram

- permitting data on the linked list to contain partial data
- allows quick encapsulation without copying

Linked list diagram



Linked list diagram

- it is likely that you do not need the trailer as hardware *may* automatically generate it

Requirements and advantages

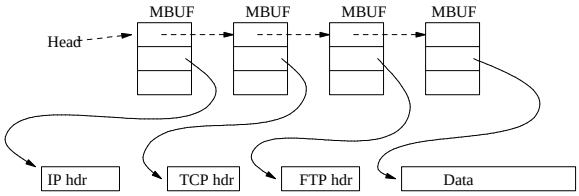
- need to make all protocol layer implementation understand the linked list mechanism (`mbuf`)
- some devices can write or read data in non contiguous blocks
  - called *scatter read scatter write*
  - linked list fits neatly with this hardware mechanism
- UNIX must translate MBUFs into iovecs and visa-versa

Incoming packets using MBUFs

- 2 cases
- case (i) either receiver is waiting for the packet
  - eager reader, or
- case (ii) packet is waiting for receiver
  - lazy reader

Case (i)

- can build empty MBUF list for incoming frame



Data needs to be large enough to contain largest data

- note we need an extra field in each MBUF indicating *# of bytes used*
  - especially for the data component

Case (ii)

- here the packet comes in before we are ready to consume data
- need to house packet until the receiver is ready
  - use a single large packet buffer
  - large enough for largest packet
- when receiver is finally ready to consume packet
  - we copy packet into our MBUF structured buffer