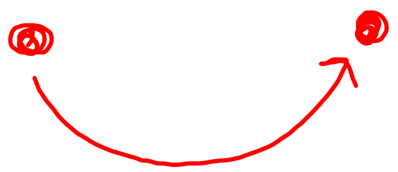
Design and Engineering of Computer Systems

Lecture 19: Network I/O Implementation

Mythili Vutukuru

IIT Bombay

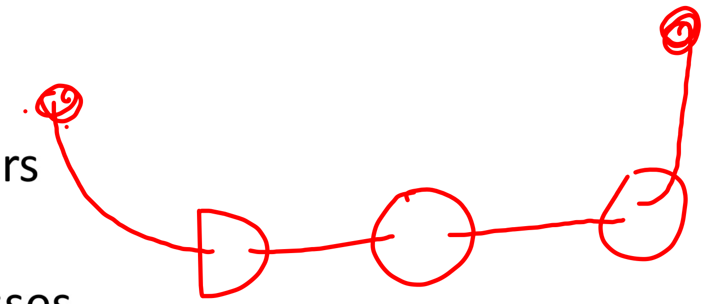
Recap: Socket API



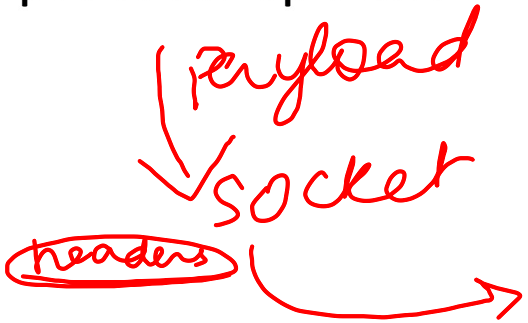
- Socket abstraction to communicate between two processes
  - Unix domain sockets for inter-process communication on same machine
  - TCP (connection-oriented) or UDP (connection-less) sockets for communication across machines
- Socket API
  - Opening a socket, binding it to an address (IP address+port number)
  - Connecting (client) and accepting new connections (server)
  - Sending and receiving data, for connection-less and connection-based
  - Event-driven APIs for managing multiple sockets at a time
  - Other: converting data from host format to network format and vice versa, …
- In this lecture: how are socket system calls implemented in OS?

Overview of network communication

- Data is exchanged on a network in units of packets = sequence of bytes
- Communicating processes have unique network addresses
  - Computers on Internet have IP (Internet Protocol) addresses
  - Multiple processes at one IP address are differentiated by port numbers

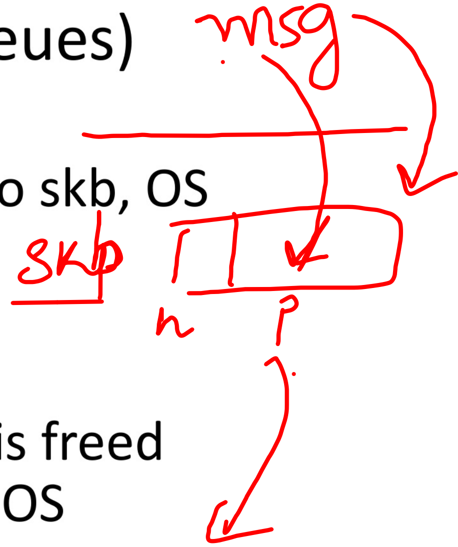


- Every packet has sender/receiver IP addresses, port numbers
  - Network routes packets from source to destination using these addresses
- Machines on a network communicate using multiple protocols
  - Protocol specifies which message to exchange with each other, format of messages
  - Example: TCP+IP protocols used for reliable connection-oriented communication
  - Example: UDP+IP protocols used for unreliable connection-less communication
- Packet sent over network = payload (actual data sent by users) + protocol-specific headers (IP address, port numbers, other metadata)
  - User program reads/writes payload using socket API
  - Protocol processing, encapsulating/decapsulating headers done by OS

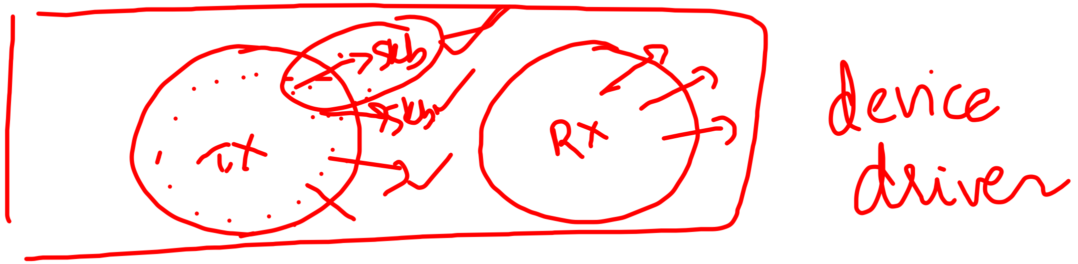


Socket buffers and queues

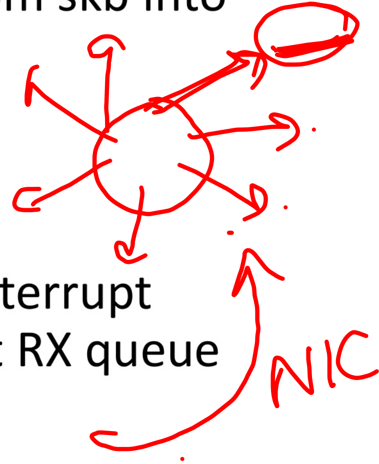
- Opening a socket returns socket file descriptor
  - Index into file descriptor array of process, which points to open file table
  - Open file table contains pointers to inode for files, socket structures for sockets, …
- Socket buffer (skb or sk_buff in Linux) is a kernel data structure to store network packets (payload + headers)
- Every socket has two socket buffer queues (transmit and receive queues)
- Send/write data into socket
  - New skb is added to socket TX queue, payload copied from user memory into skb, OS processes packet (adds headers) and transmits via device driver
  - When packet transmitted over network and no longer needed, skb is freed
- Receive/read from socket
  - Dequeue skb from RX queue, payload copied from skb to user memory, skb is freed
  - If socket RX queue empty, process blocks until: data received from network, OS processes packet, adds skb to socket RX queue



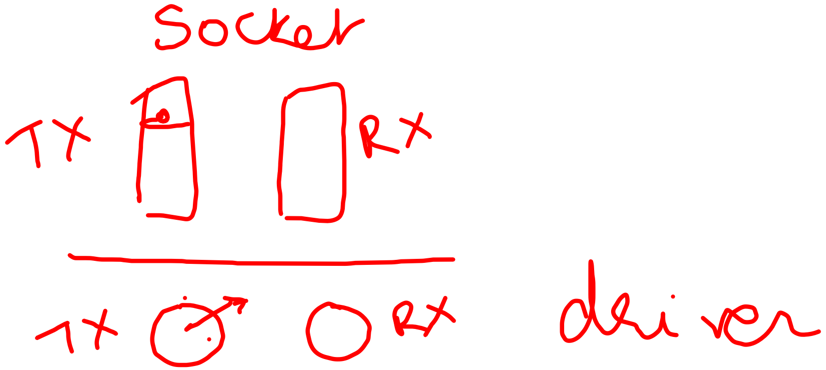
Network device driver



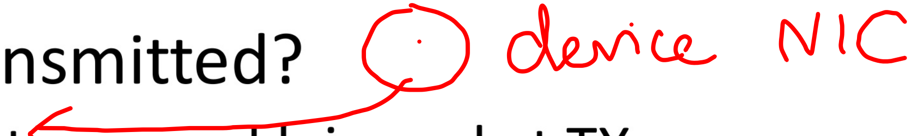
- Device driver talks to network interface card (NIC) to exchange packets
- Device driver maintains TX/RX rings of packet descriptors in main memory
  - Ring = circular array (loop back to start upon reaching end)
  - Packet descriptor = pointer to socket buffer and other info about packet
  - Head/Tail = pointers to start/end of occupied slots in ring
  - NIC knows locations of TX/RX rings and can access it in memory
- TX ring is queue of skb waiting to be transmitted
  - Device driver adds skb to TX ring when packet is ready to be sent
  - When NIC free to transmit, NIC reads address of next skb from TX ring, DMA packet from skb into device hardware
  - When transmit complete, NIC raises interrupt, skb in TX ring is freed
- RX ring is queue of empty skb waiting to be filled by received packets
  - Device driver initializes RX ring with pointers to empty skb
  - NIC receives packet, find empty skb on RX ring, DMA received packet into skb, raises interrupt
  - OS handles interrupt, processes received packet, hands it over to corresponding socket RX queue
  - When received packet dequeued from RX ring, new empty skb is replenished by OS



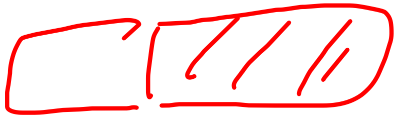
Packet transmission



- Summary: TX/RX queues at socket, TX/RX rings at device driver
- How is a network packet transmitted?

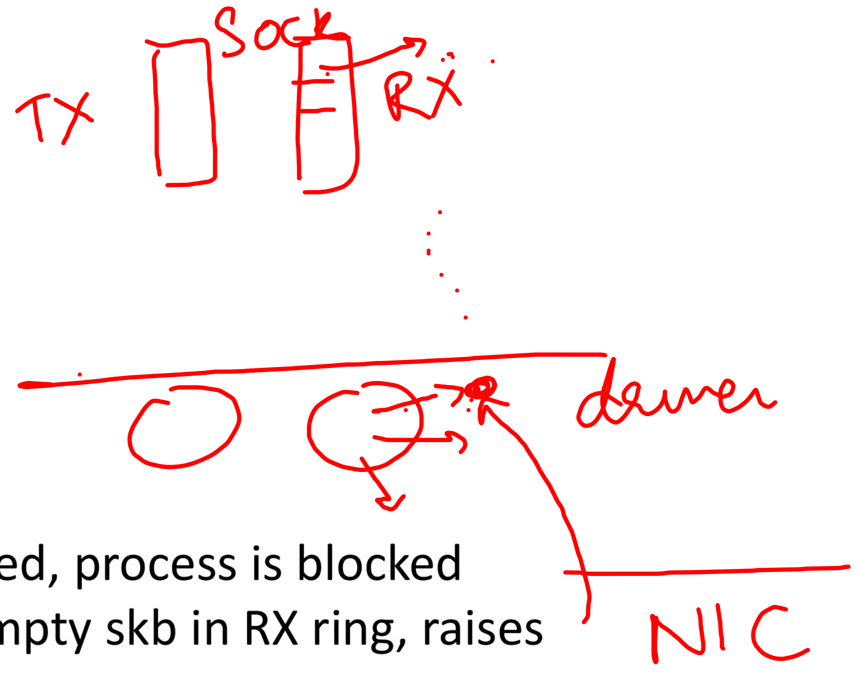


  - Write/send system call allocates new skb in socket TX queue, copies data from user memory into skb
  - OS performs network protocol processing, adds headers to skb

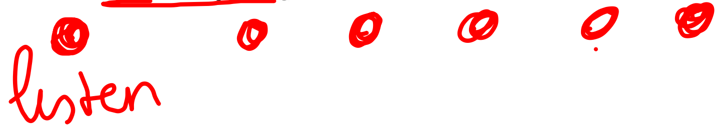


  - When network protocol decides to send packet, OS adds skb to TX ring (note that some network protocols may slow down transmission for congestion control and other reasons)
  - When device is free to transmit, DMA packet from TX ring into device, raise interrupt when transmission complete
  - OS handles interrupt, frees up skb in TX ring

Packet reception

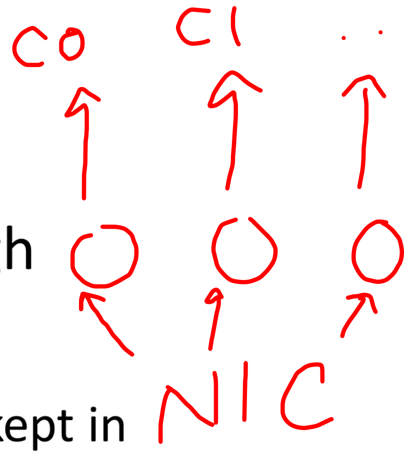


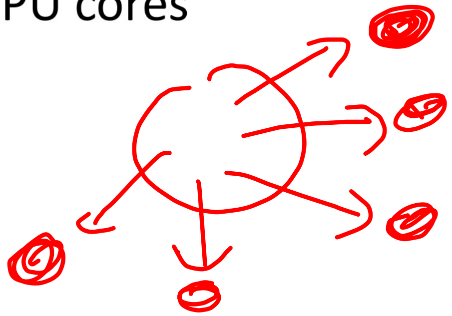
- How is a network packet received?
  - Device driver populates device RX ring with empty skb
  - If process makes read/recv system call before data is received, process is blocked
  - When packet received, NIC performs DMA of packet into empty skb in RX ring, raises interrupt
  - OS handles interrupt, performs minimal processing (e.g., acknowledge interrupt) in interrupt handler (top half), schedules another interrupt handler process (bottom half) to run when CPU is free
  - Bottom half interrupt handler removes skb from RX ring, replenishes RX ring with fresh skb, processes received packet, adds received skb in socket RX queue
  - When recv system call returns, data copied from skb into user memory, skb freed up
- How is socket identified based on received packet?
  - For connection-less sockets, receiver IP address/port number uniquely identifies socket
  - For connected sockets, sender/receiver IP address/port number (4-tuple) identifies socket



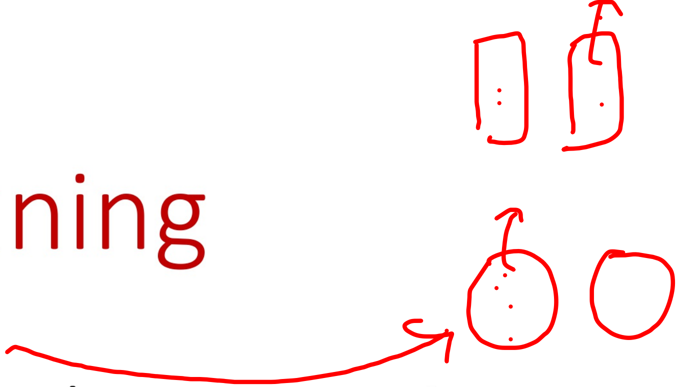
Optimizations to packet reception

- Receive-side processing of network packets split into top half and bottom half interrupt handlers: why?
  - Top half does minimal processing, to avoid disruption to interrupted process
  - Bottom half (soft IRQ) is separate process that is scheduled when CPU is free, does processing related to various network protocols
- One CPU core may not be able to keep up with interrupt processing on high speed network cards (~100Gbps today)
  - Receive side scaling (RSS) feature in NICs allows NIC to have multiple RX/TX rings
  - NIC splits received packets among multiple RX rings (packets of one connection are kept in the same ring, use hash of connection 4-tuple to pick RX ring)
  - Each RX ring is assigned to separate core, interrupt handling distributed to CPU cores
- Another optimization: NAPI (new API) to reduce interrupt load
  - Once interrupt is raised, all future interrupts disabled till bottom half runs
  - Bottom half polls all packets received until then, reenables interrupts





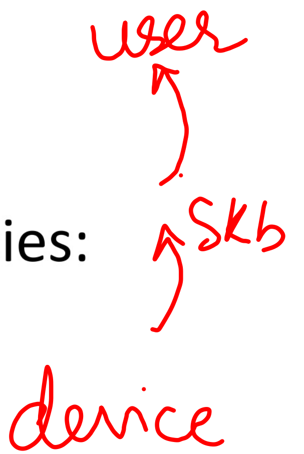
Performance tuning

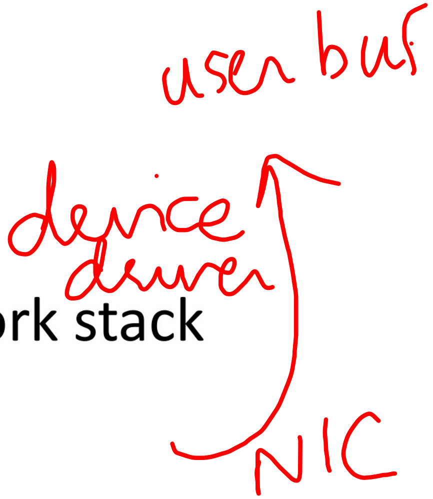


- Multiple queues: socket TX/RX queues, device TX/RX rings (finite size)
- Mismatch in speed of network, NIC, OS, application can lead to queues building up and overflowing, packet drops, poor performance
  - If packets arriving at very high rate on NIC, but OS not handling interrupts fast enough, device RX ring can overflow, packets dropped by device
  - If application reading packets very slowly from socket, socket RX queue hits maximum value, packets dropped
  - If device too slow in transmitting, device TX ring and socket TX queue become full, send/write into socket can block
- Best performance achieved when speeds of all components and queue sizes are matched (more on this topic later)
  - Sender must adjust sending speed based on capacity of network and receiver
  - Socket queue size and device driver ring size must be tuned for optimal performance

Kernel bypass techniques

- I/O subsystem in OS not designed for high speed network I/O, has inefficiencies:
  - Interrupts, system calls for every packet, leads to expensive traps and context switches
  - Dynamic skb allocation for every packet, adds overhead
  - Packet data copied twice: device to skb, skb to user memory
- For high speed network I/O (~ 100Gbps), modern computer systems employ kernel bypass techniques, e.g., Data Plane Development Kit (DPDK)
  - Uses special device driver to access NIC, regular kernel processing fully bypassed
  - Packets DMA directly into userspace memory (zero copy)
  - Preallocate packet buffers in huge pages (efficient memory access)
  - Avoid interrupts, application itself checks RX ring periodically (polling)
  - Access multiple packets at a time from RX ring (batching)
- Disadvantages of kernel bypass: kernel isolation mechanisms and network stack processing fully bypassed, regular kernel networking tools do not work
  - Mainly useful in applications that process very high speed I/O





Summary

- In this lecture:
  - Socket API implementation in OS
  - Performance bottlenecks and optimizations
- Find out the size of RX/TX rings, default size of transmit/receive socket buffer queues in your system, and how to tune them.
  - Tools and commands exist in Linux and other OS to configure these queues