6.S081: Networking

abelay@mit.edu

Today’s lecture

- Networking basics
- OS networking abstractions
- Receive livelock

Logistics

- Due this Thursday:
  - Mmap lab assignment
  - Project status reports (if you're registered for 6.828)
- Networking lab (next assignment) is posted today

Networks

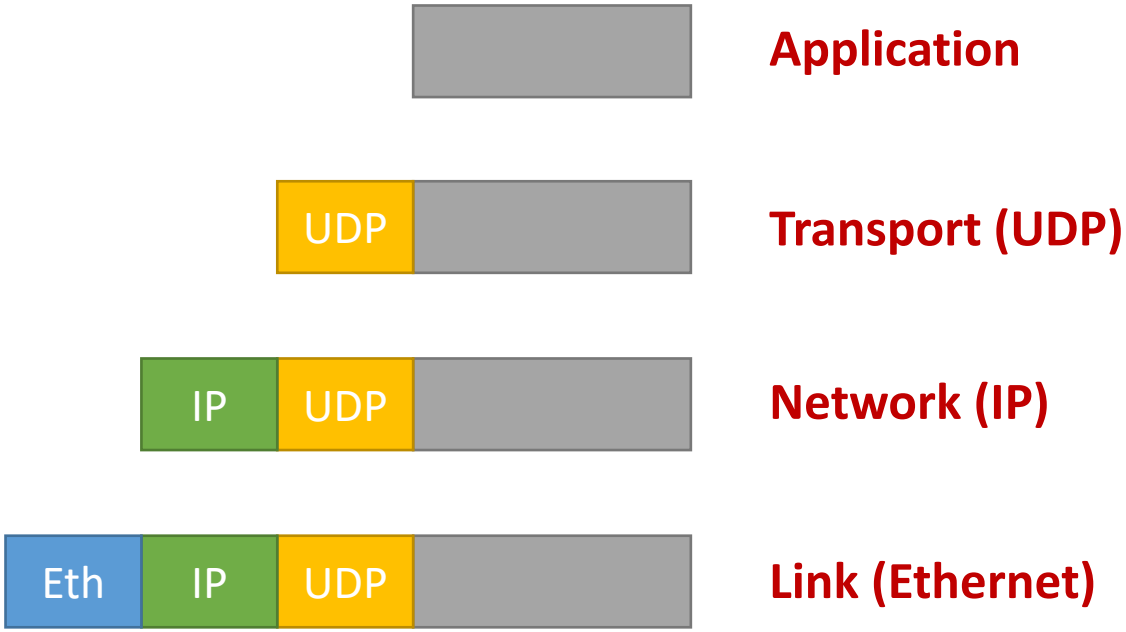
- What is a network?
  - A system of channels that interconnect nodes
  - E.g. railroads, highways, plumbing, telephones
- What about computer networks?
  - A communication network that moves information
  - The nodes are computers!
- Computer networks are powerful networks
  - Interaction between nodes and network is programmable
  - Today's networks can move incredible amounts of information very quickly

Bandwidth-delay product

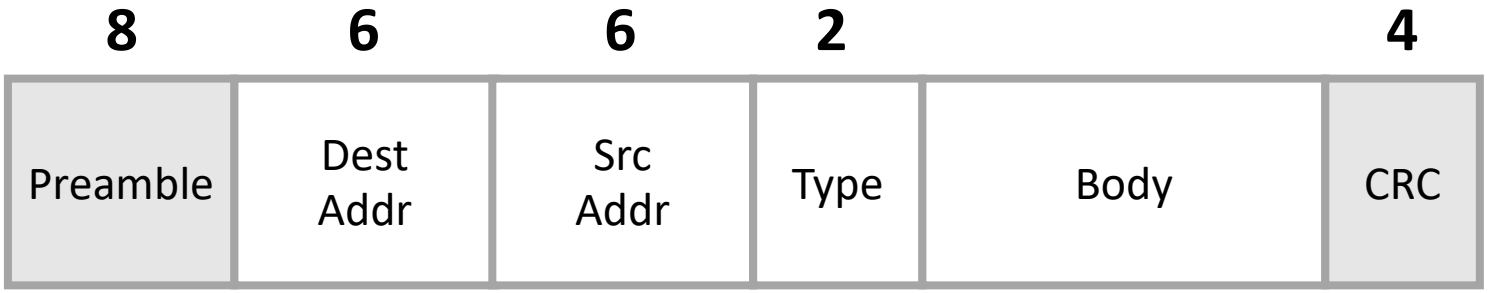
- Can view network as a pipe
- For full utilization:
  - # bytes in flight ≥ bandwidth * delay
  - But too much in flight can cause collapse
- What if protocol doesn't bulk transfer? (e.g. RPC)
  - Concurrency needed for higher utilization

Layering

- Networks are built on a series of layered standards
- Each layer has a *header*, a metadata preamble
- Each layer *encapsulates* the next layer



Example layer: Ethernet



- Used by just the local network (not across the Internet)
- Vendors give each device a unique 48-bit MAC address
  - Specifies which node should receive a packet
  - OS can tells NIC which MAC address to accept
  - FF:FF:FF:FF:FF:FF broadcasts to entire network fabric
- Extra fields on wire (in grey) but stripped by NIC hardware
  - Preamble helps device recognize packets
  - CRC detects corrupted packets
- Type aids in encapsulation (IPv4, IPv6, ARP, etc.)

Example layer: Internet protocol

- Used to connect different networks together (i.e. the internet)
- IP addresses (e.g. 8.8.8.8)
  - Like street addresses, describe a location in network
  - Routers use address to determine path through network
  - 127.0.0.1 is special, loopback to the local machine
- Header is much more complex than Ethernet
  - But still has source, destination, and type
  - Usually encapsulates UDP and TCP

Address resolution protocol (ARP)

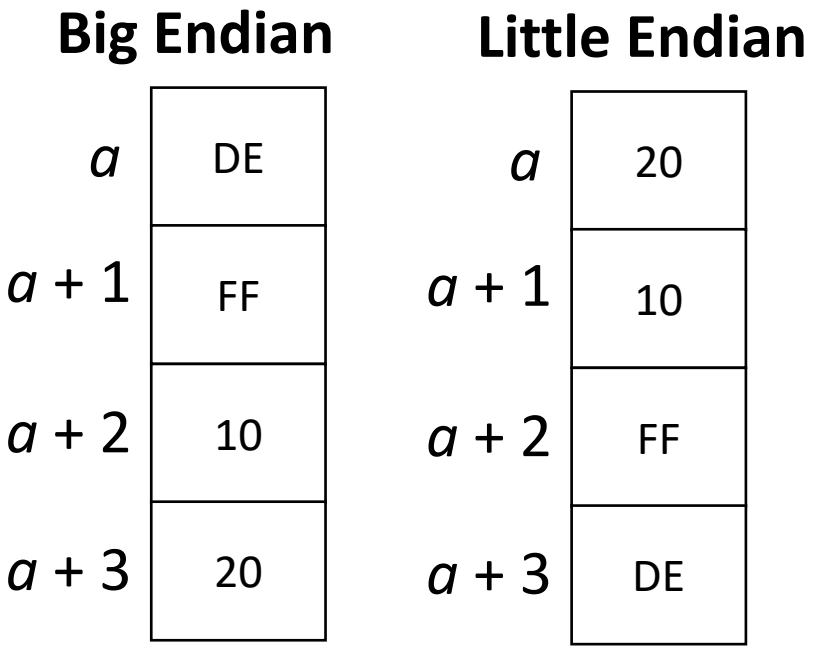
- Goal: Want to send to an IP address.
- Problem: Which MAC address to use?
- If destination is in same link network, use its MAC address, otherwise use MAC of gateway
- ARP:
  - Broadcast request for MAC address of IP address
  - Everybody learns requester's MAC and IP
  - Target machine responds with its MAC address
- OS maintains cache of ARP mappings

Endianness

- The order of bytes within a short or int
- RISC-V is little endian, but network is big endian
- To convert, use ntohs/ntohl (network order -> host order) and htons/htonl (host order -> network order)

32-bit Integer

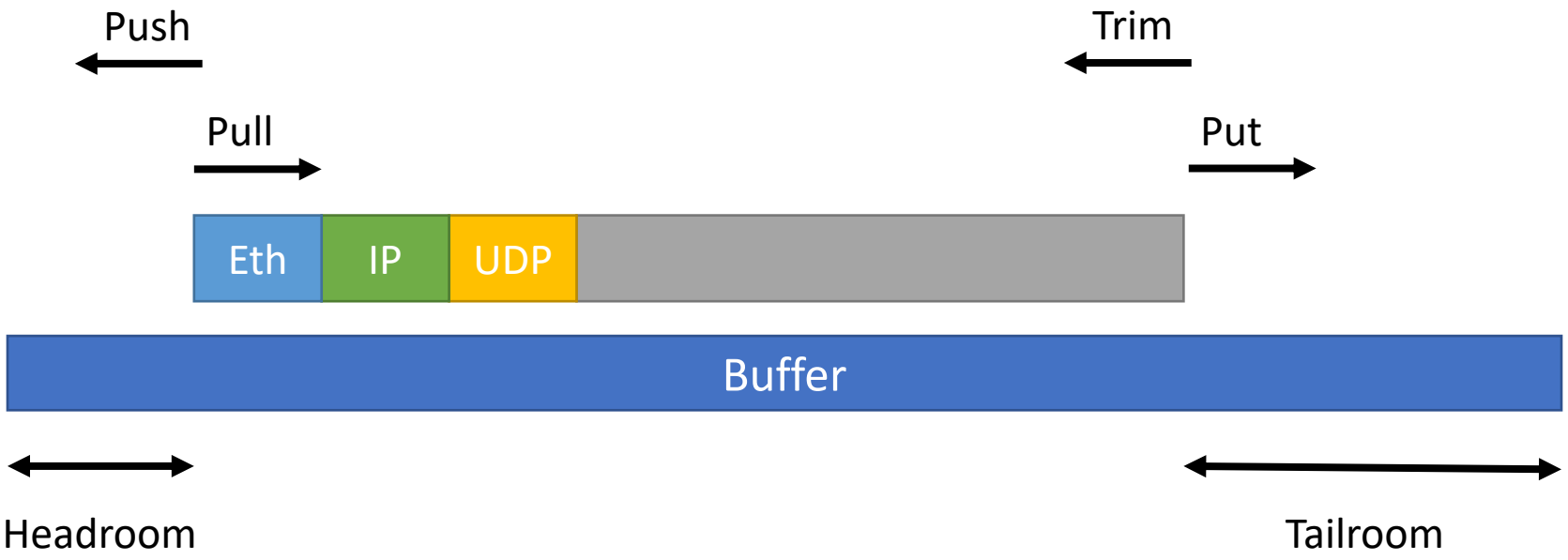
0xDEFF1020



Mbufs

- Can’t store packets in contiguous memory
  - Moving data to make room for new headers is slow
- Need an abstraction to support layering efficiently
  - E.g. add UDP header to data, add IP header to UDP, etc.
  - E.g. Remove ethernet header so IP code doesn’t see it
- Solution mbufs! (original idea from BSD)
  - Linux uses sk_buffs, similar idea
  - Simple mbuf provided in lab assignment
  - Mbufs can also store metadata about packets (e.g. checksum, arrival time, etc.)

Mbuf layout



- Buffer is larger than largest possible packet
- Headroom and tailroom leave extra space for headers

Networking lab code walkthrough

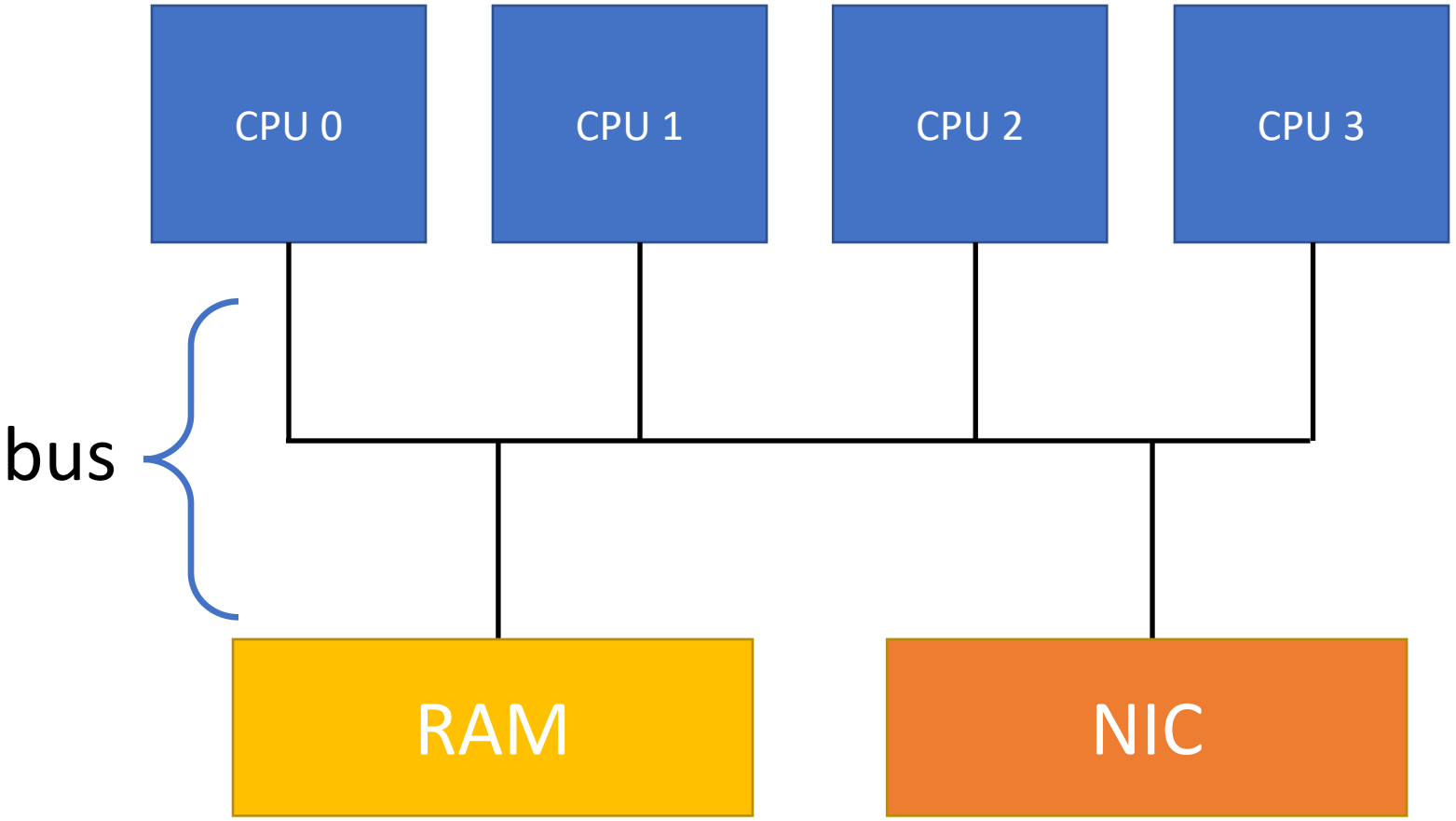
Sockets

- Abstraction for communicating between machines
- **Datagram sockets**: Unreliable message delivery
  - E.g. UDP
  - Messages may be reordered or lost
  - Reads return the full message (if req len is large enough)
- **Stream sockets**: Bi-directional pipes
  - E.g. TCP
  - Bytes written on one end, are read on the other
  - Reads may not return the full amount requested

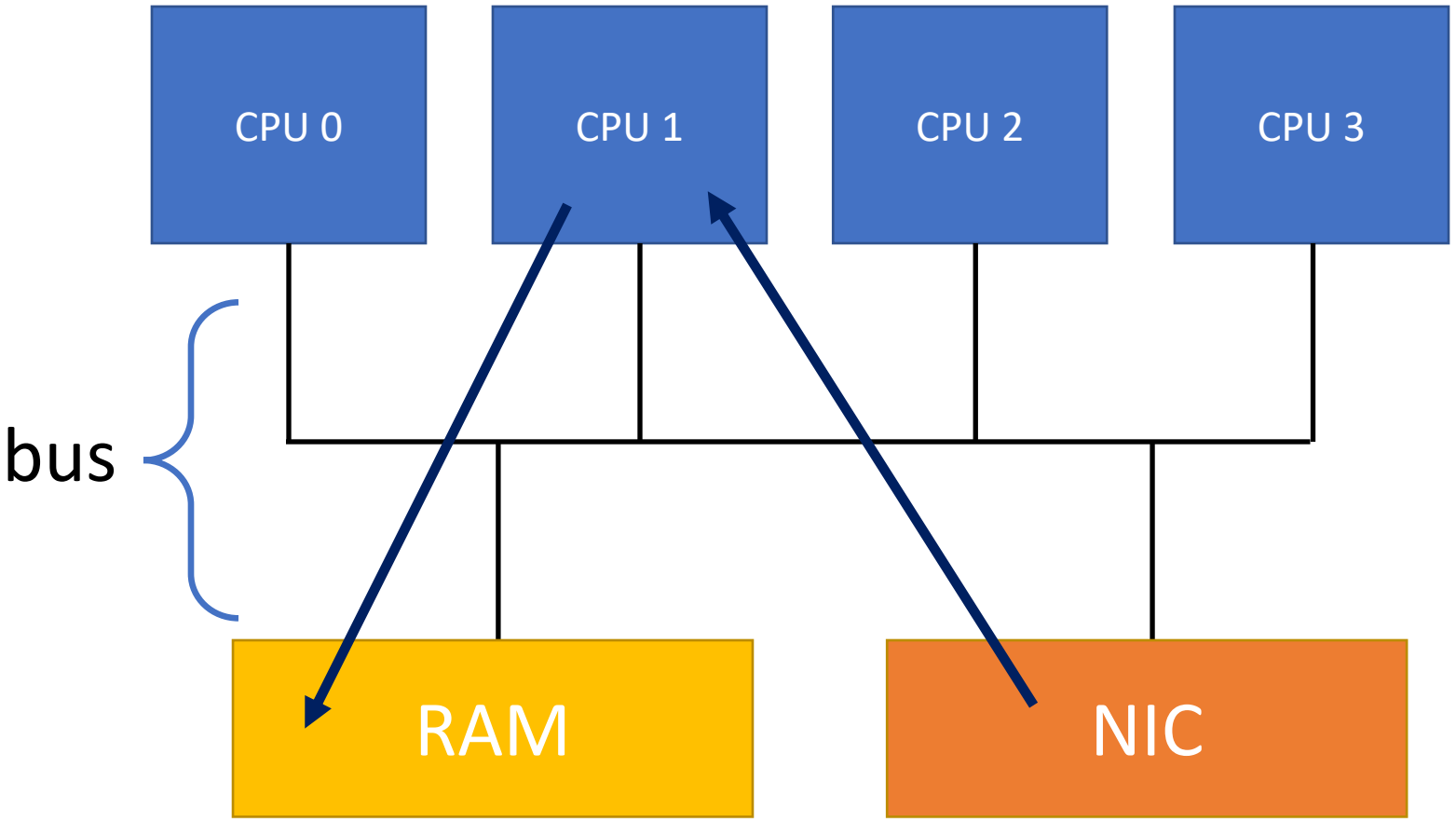
Socket implementation

- Both TCP and UDP name connection endpoints
  - 32-bit IP address specifies machine
  - 16-bit Port number demultiplexes within host
- Thus, a connection is named by 5 components
  - Protocol (UDP), local IP, local Port, remote IP, remote Port (called a 5-tuple)
- OS keeps connection state in PCB structures
  - Keep all PCBs in a hash table
  - When packet arrives, use 5-tuple to find PCB and use PCB to determine what to do with packet

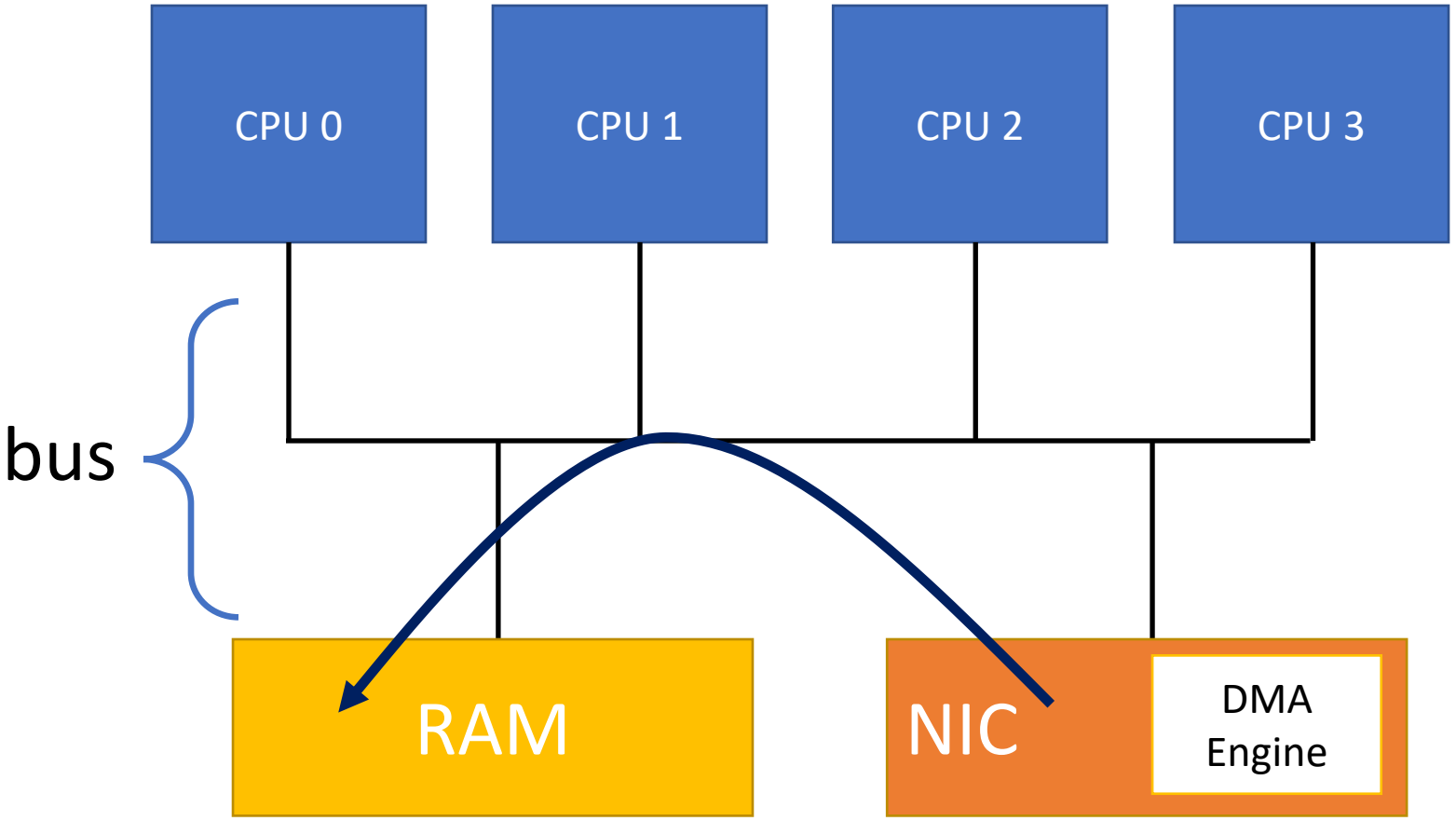
How to transfer pkts to/from NIC?



Idea: Have CPU copy to RAM

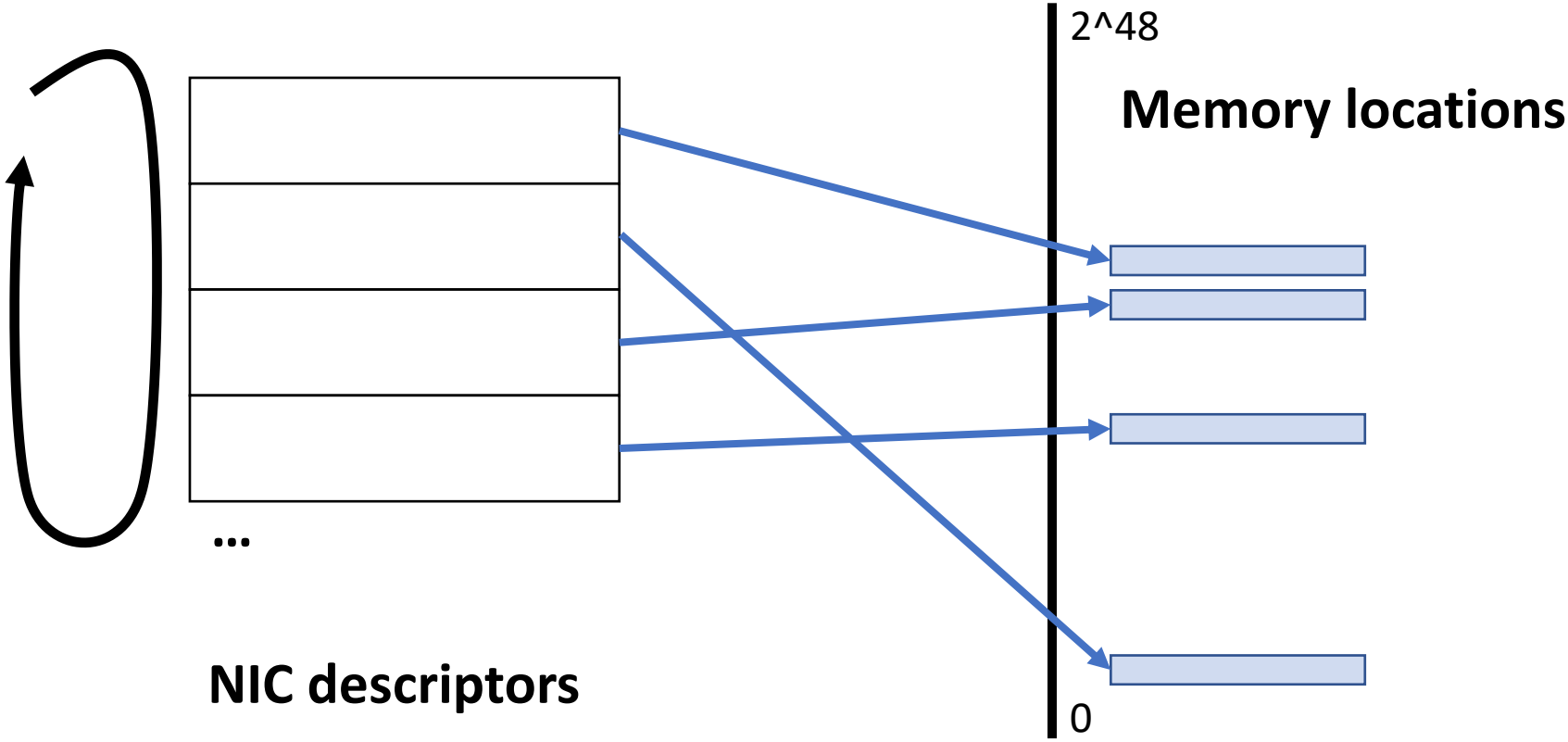


Better: Have NIC copy to RAM
Called direct memory access (DMA)



OS programs DMA engine

- Circular array of descriptors (fetched from memory by NIC)
- Each descriptor describes location to put packet in memory



DMA details

- OS provides NIC with locations to copy packet data
- NIC provides OS with notifications of finishing
  - Mechanism #1: Writes done flag to descriptor
  - Mechanism #2: Sends interrupts
- OS recycles descriptors
  - Gives previous buffer to networking stack (or frees it)
  - Then allocates and programs new buffer into descriptor
- Descriptors often contain flags and metadata about how to receive or transmit a packet
- Separate descriptor rings for receive and transmit

Don’t starve the DMA engine

- Need to keep descriptor rings full!
- What if receive ring goes empty?
  - NIC drops packets!
- What if transmit ring goes empty?
  - NIC wastes bandwidth! (doesn’t send)
- OS has to constantly monitor descriptors!
  - Can poll, by checking them periodically
  - Or can program NIC to send interrupts

What’s better… Interrupts or polling?

Polling vs. interrupts

- When is polling good?
  - When tasks are predictable, polling works better
  - Task can cooperatively decide when to poll
  - E.g. software routers are great at polling
- When are interrupts good?
  - When tasks are unpredictable or uncooperative
  - E.g. what if the service time of requests is variable
  - E.g. what if one task doesn’t care about performance of other tasks on same machine

Both polling and interrupts can waste the CPU

- Polling wasteful when it is used to wait for new packets
  - Could do other work instead!
- Interrupts are wasteful when they're frequent
  - Each interrupt has much higher overhead than a poll
  - Can easily dominate the CPU
- Solution: Use both to minimize waste

Receive livelock

- Let’s assume an interrupt is delivered for each received packet
  - OS refills descriptor ring and processes each packet
  - Maintains low latency even if app isn’t cooperating
- Now assume the packet arrival rate is much higher than the rate the OS can handle packets
- OS will spend nearly all time processing interrupts
  - Lower priority functions like transmitting and running application logic will fail to make progress
  - Result: System performs no useful work! (livelock)

How to fix livelock?

- Idea 1: Drop requests as early as possible when overloaded
  - Minimizes wasted work if they’re going to be dropped anyway
- Idea 2: Process requests to completion
  - Don’t start a new request until old requests finish
  - Guarantees progress

Solution details

- Using interrupts only to initiate polling.
- Using round-robin polling to fairly allocate resources among event sources.
- Temporarily disabling input when feedback from a full queue, or a limit on CPU usage, indicates that other important tasks are pending.
- Dropping packets early, rather than late, to avoid wasted work. Once decided to receive a packet, try to process it to completion.

Livelock results

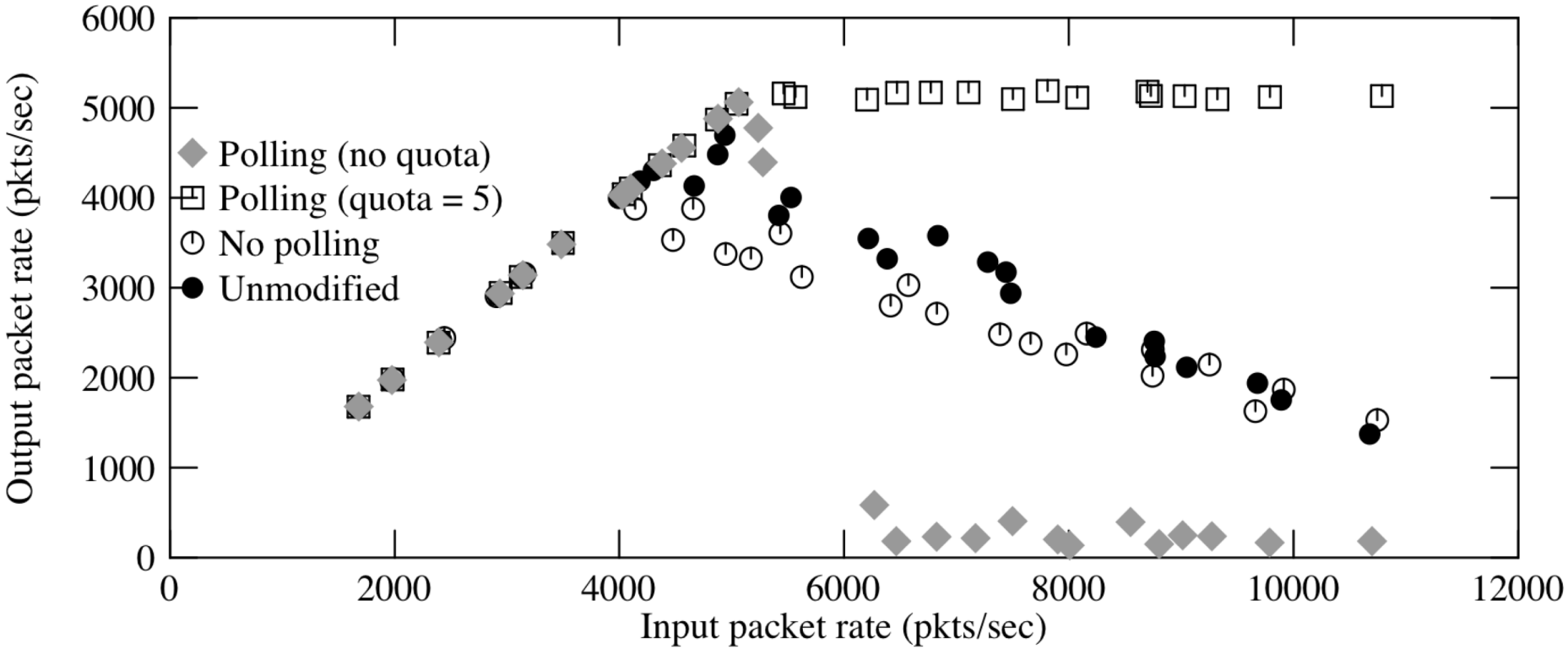


Figure 6-3: Forwarding performance of modified kernel, without using *screend*

Conclusion

- Computer networks require well-behaved OSes
  - OS decides how and when to process packets
  - Every detail matters for good performance
- Drop requests as early as possible if you can't handle them
- In most cases, both interrupts and polling are needed
- Mbufs and sockets are powerful abstractions for OS networking