Devices and the Hardware/Software Interface

CS 571

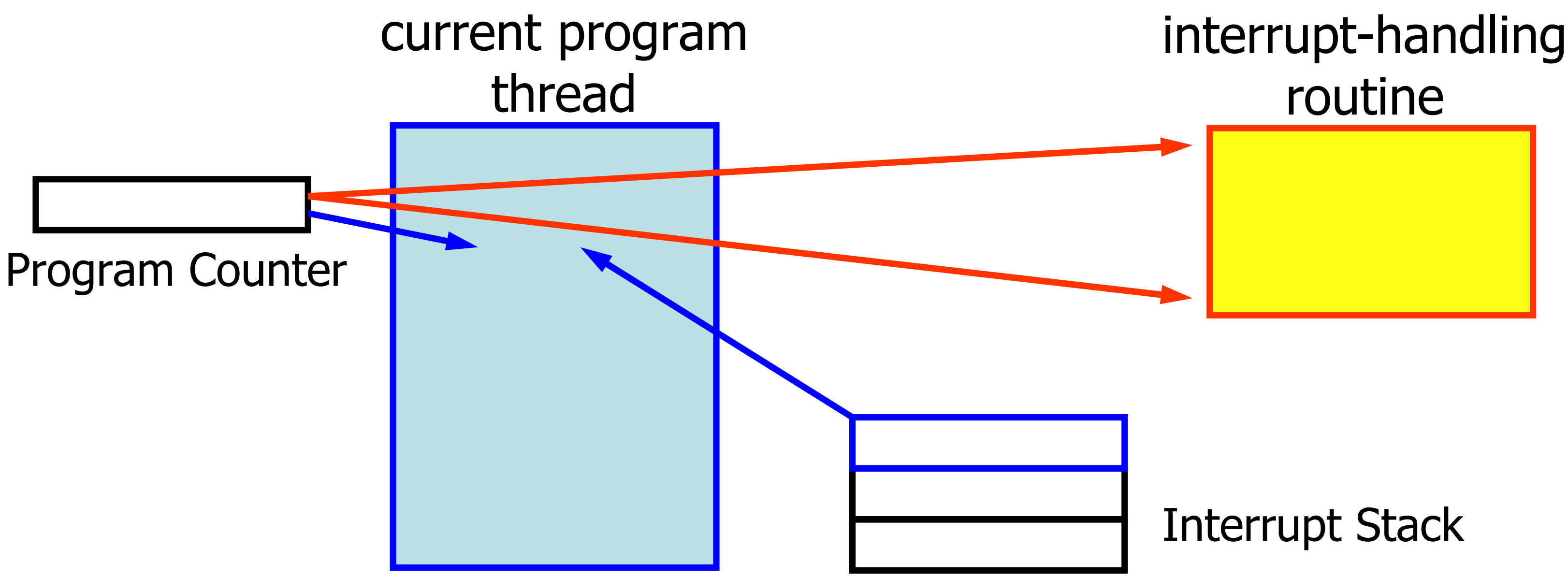
Fall 2006

What Network Hardware Does

- Transmit:
  - Add framing information
  - Serialize data from memory
  - Gain access to channel via MAC Protocol (if applicable)
  - Modulate signal to transmit symbols per physical protocol
  - Implement Error Detection protocol (if applicable)
  - Inform software (via interrupt) when transmission is complete
- Receive:
  - Derive symbols from physical signal
  - Recognize station address (if applicable)
  - Strip framing, de-serialize into memory
  - Perform error-detection checks (if applicable), signaling errors
  - Inform software (via interrupt) when frame is received

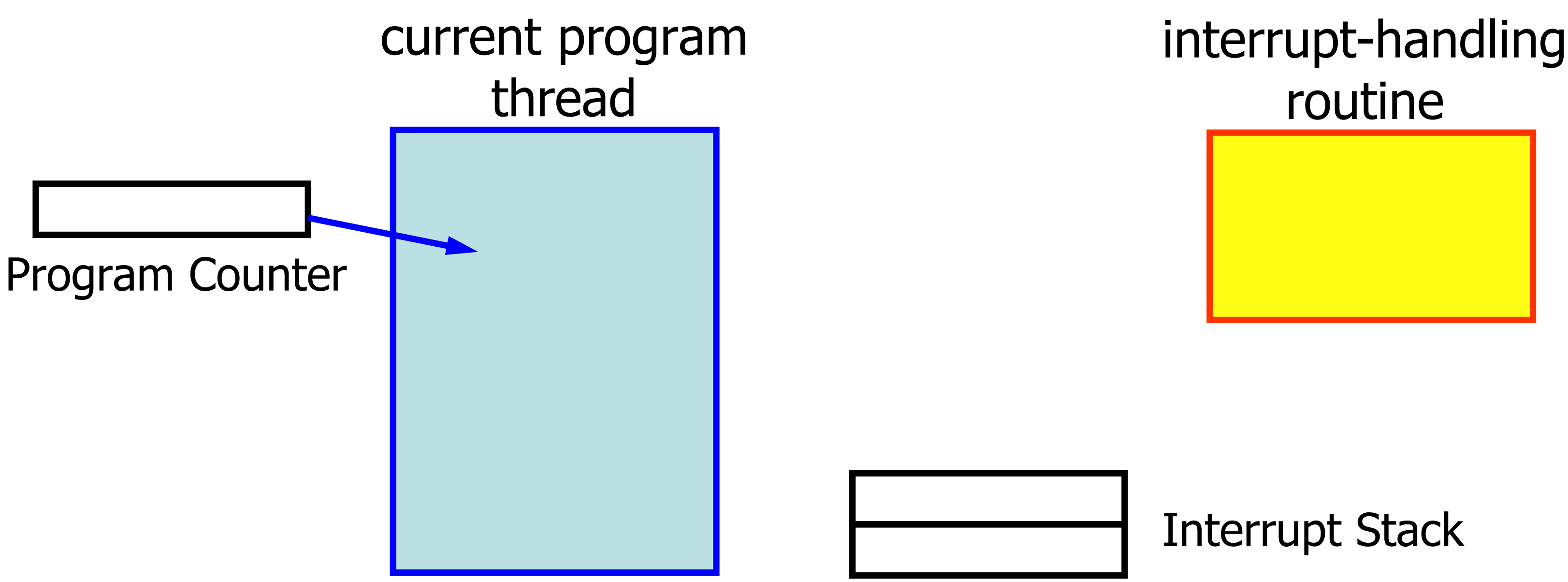
Background: Threads of Control

- At any time, the CPU is executing instructions of at most one <u>thread of control</u>
  - Part of a: user program, OS, or device driver,
- When an interrupt occurs, CPU begins executing instructions of a special thread
  - Interrupt-handling routine of device driver



Background: Threads of Control

- At any time, the CPU is executing instructions of at most one thread of control
  - Part of a: user program, OS, or device driver,
- When an interrupt occurs, CPU begins executing instructions of a special thread
  - Interrupt-handling routine of device driver



Background: Threads of Control

- At any time, the CPU is executing instructions of at most one <u>thread of control</u>
  - Part of a: user program, OS, or device driver,
- When an interrupt occurs, CPU begins executing instructions of a special thread
  - Interrupt-handling routine of device driver
    - Code may be anywhere
    - Pointer to code ("<u>interrupt vector</u>") is stored in a low memory location associated with that device
  - Hardware automatically...
    1. Saves current state (on the interrupt stack)
    2. Begins loading instructions from that location

    ...when that device raises an interrupt
- Devices may have different interrupts for receiving and transmitting

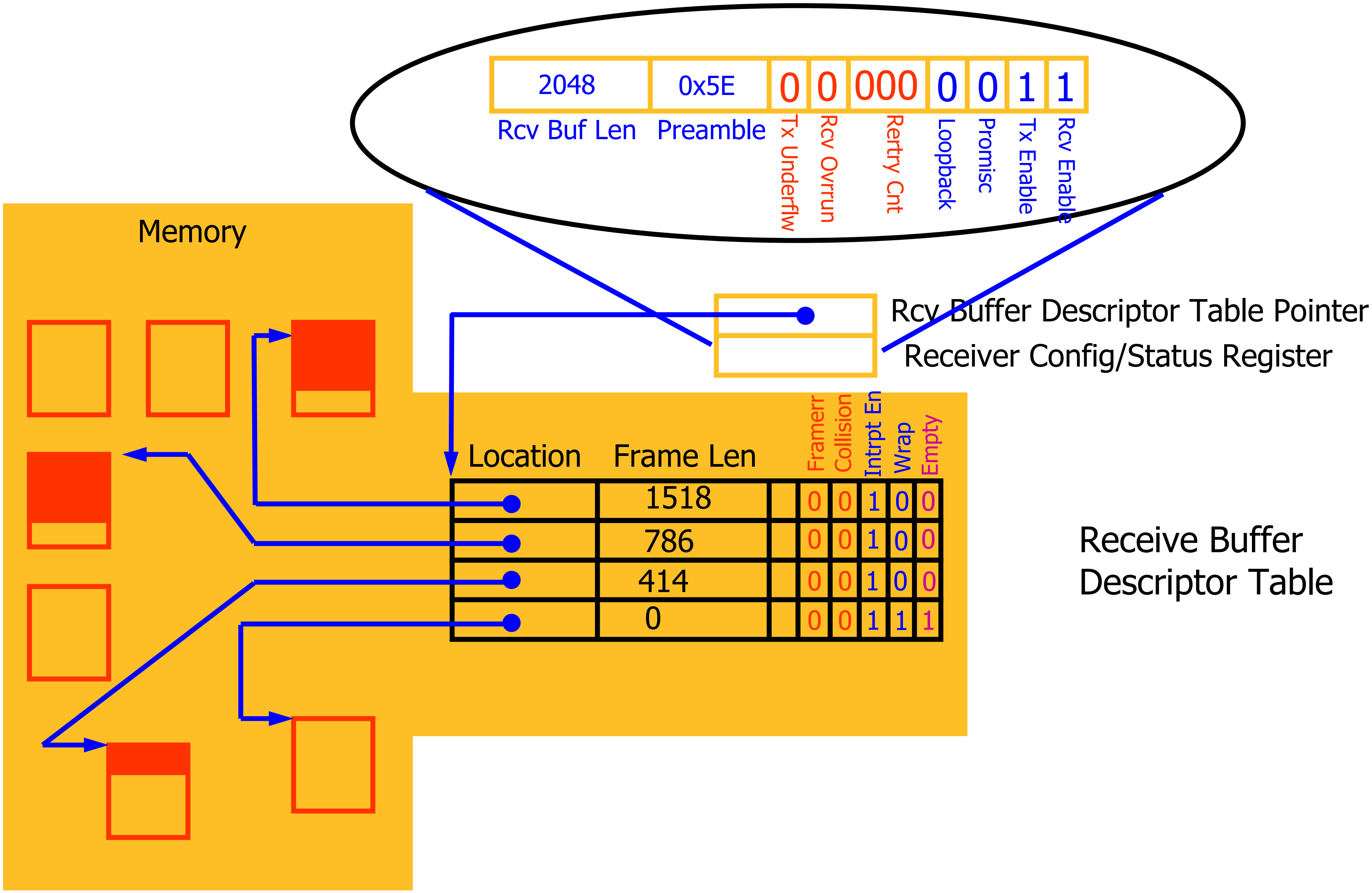
What Software Does

- Inform hardware of buffer (memory) locations
  - Where incoming frames should be placed
  - Where frames to be transmitted are located
  - Buffer descriptors: pointers to memory areas for hardware use
  - For small frames (e.g. single characters), data is passed directly via data registers
- Inform hardware when buffers are ready to use
  - Receive buffers: empty
  - Transmit buffers: full, ready to send
  - Implemented by setting/clearing a bit in the buffer descriptor
- Add addressing information if applicable
  - E.g. Ethernet
- Deal with any errors signaled by device
  - E.g. framing errors, checksum errors

How Do S/W & H/W Communicate?

- Through <u>device registers</u>
  - Special memory locations
  - Readable/writable by (privileged) software
- H/W → S/W: Registers indicate:
  - When a buffer is ready for s/w to service (by setting a bit)
    - Why isn't just interrupting sufficient?
  - When an error has occurred
  - Device status (ready, synchronizing, ...)
- S/W → H/W: Registers control:
  - Where buffers are
  - Device configuration
    - E.g. for async: # bits/frame, # stop bits
  - When a buffer is ready for h/w to service (by setting a bit)

Example Hardware Interface



Example Ethernet Config/Status Register

Bits	17-32	9-16	8	7	4-6	3	2	1	0
	0x800	0x5D	0	0	000	0	0	1	1
	Receive Buffer Length	Start Frame Delimiter	Transmit Underrun	Receive Overrun	Rertry Count	Loopback Mode	Promiscuous Mode	Transmitter Enable	Receiver Enable

Software Bit-Diddling

- Accessing registers:
  - Set a pointer (to the appropriate word size) to the (fixed!) address of the register
    - This only works <u>in the kernel</u>!
    - Can be troublesome if the device control register addresses are not fixed! (Pre-Plug-n-Play PC devices)
  - Read/write indirectly via the pointer

```
#define ETHER_MCSR  0xfff78420
unsigned int regValue, *csr;
csr = ETHER_MCSR;
regValue = *csr;
```

Software Bit-Diddling

- Setting individual bits:
  - Get the current value
  - OR in the desired bit
  - E.g., to turn on Loopback mode (bit 3):

```
#define LOOPBACK_FLAG  0x8 // or (1<<3)
*csr |= LOOPBACK_FLAG;
```

- Clearing individual bits:
  - Get the current value
  - AND with the complement of the desired bit
  - To turn off Loopback mode:

```
*csr &= ~LOOPBACK_FLAG;
```

Software Bit-Diddling

- Complementing individual bits
  - XOR with the desired bit

```
*csr ^= LOOPBACK_FLAG;  // invert the flag!
```

- Reading groups of bits as a number:
  - AND the register value with the desired bits
  - Shift to proper magnitude
  - E.g., to check number of retries (bits 4-6):

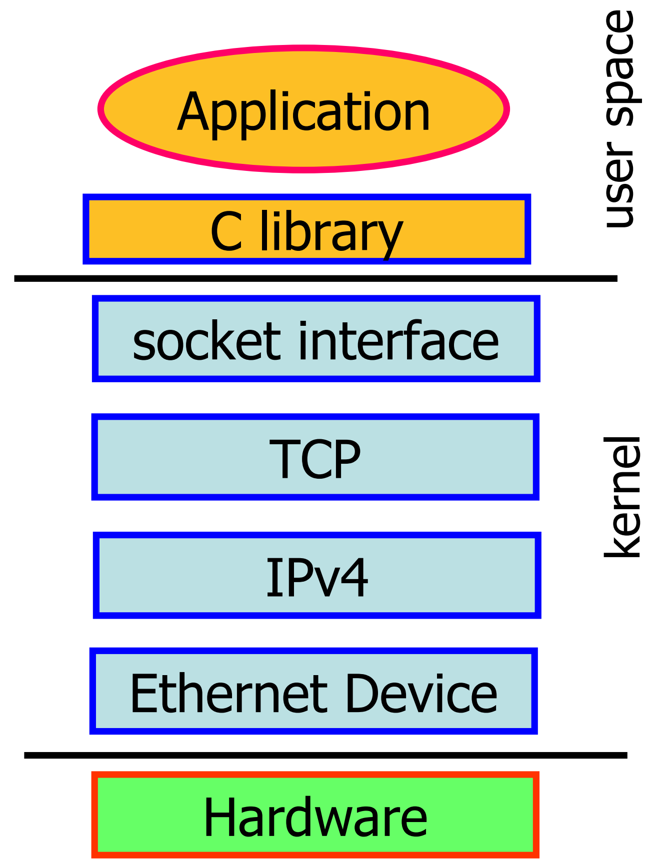
```
#define RETRYCOUNT_SHIFT  4
#define RETRYCOUNT_MASK   0x70
numRetries = *csr & RETRYCOUNT_MASK;
numRetries >>= RETRYCOUNT_SHIFT;
```

Anatomy of a Packet Transmission

Assumptions:

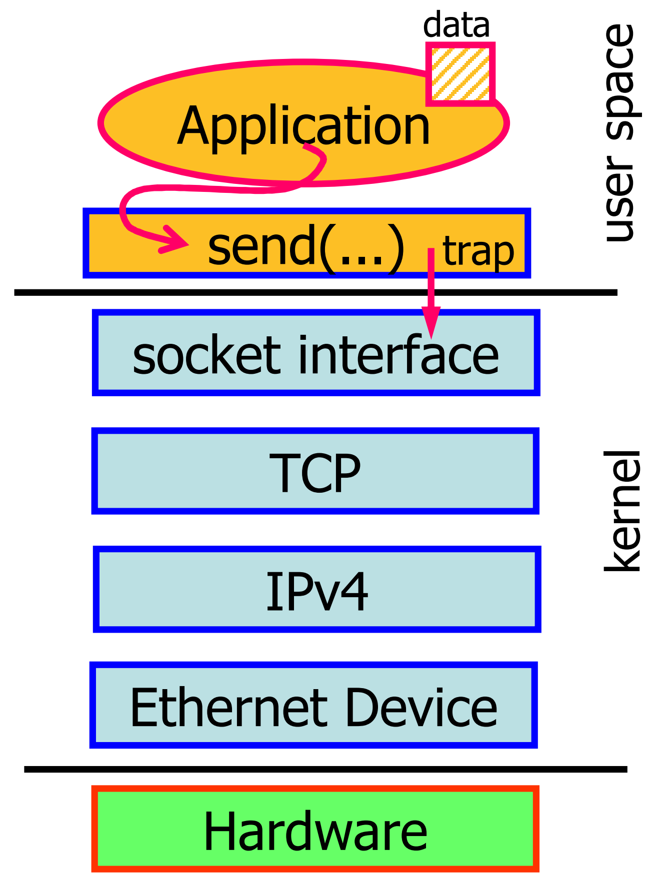
- User-space C program using TCP via "sockets" interface
- Sending a 500-byte message
- Machine connected to an Ethernet
- Modern Operating System
- No prior messages sent

• N.B. This is generic and greatly simplified!



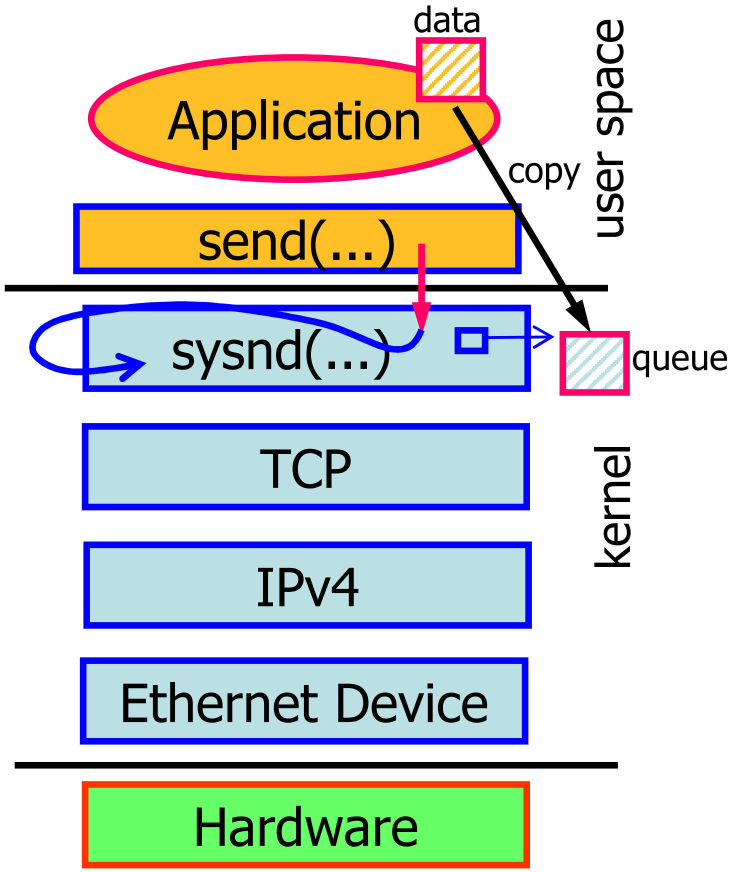
Anatomy of a Packet Transmission

1. Application calls "send(sock#, bufPtr, 500)"
   - Run-time C library implementation of send() pushes arguments on stack
   - Implementation executes a "system call trap" instr.
   - Address of kernel trap svcing routine loaded into PC
   - Processor changes to privileged mode



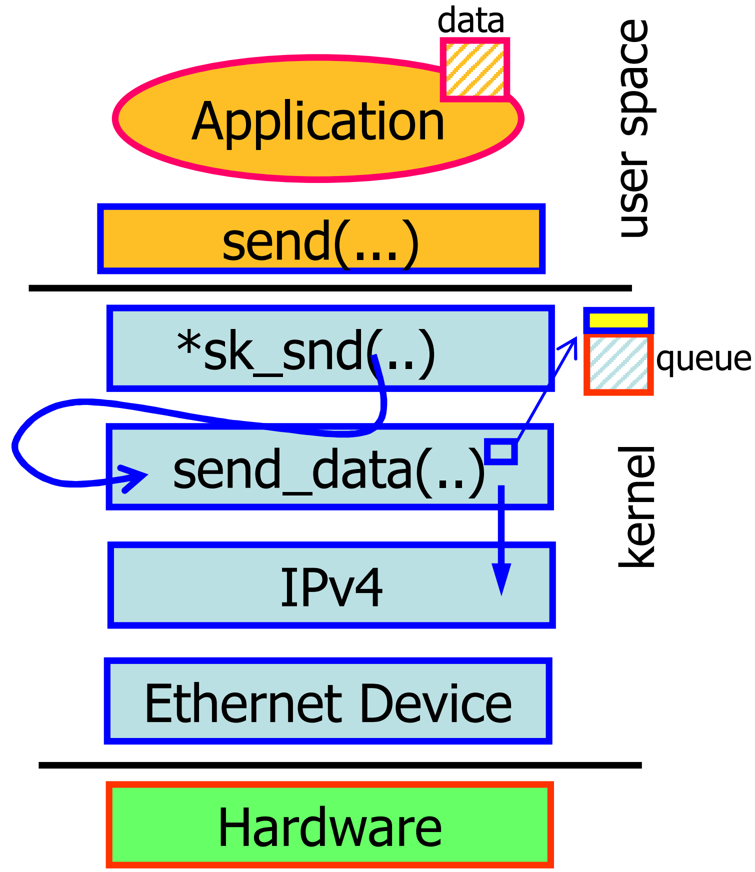
Anatomy of a Packet Transmission

2. Trap handler invokes kernel implementation of send( ) system call
   - Validates arguments (e.g., pointer is in the proc's address space)
   - Copies user data into kernel address space, adds buffer header
   - Locates the state data structure for the socket
   - Verify the socket state is OK to transmit
   - Appends the data to the socket's send queue (assumed empty)



Anatomy of a Packet Transmission

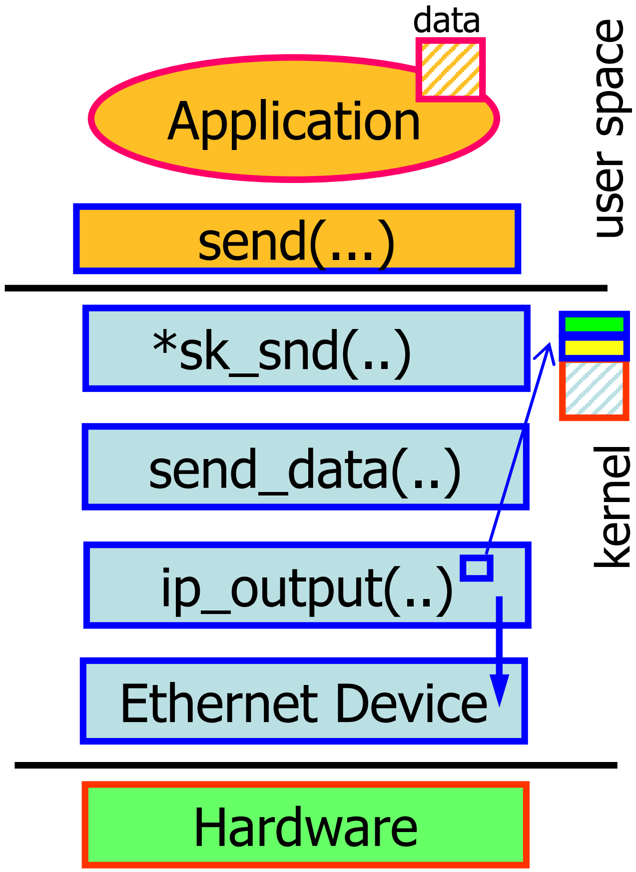
3. System call invokes (indirectly) socket's sk_send function
   - Invokes TCP "send_data()" function, which:
     - Retrieves the relevant TCP state info
     - Checks whether it is possible to send anything (flow ctl)
     - Constructs 20-byte TCP header, prepends to message
   - TCP send_data invokes "ip_output( )" with packet



Anatomy of a Packet Transmission

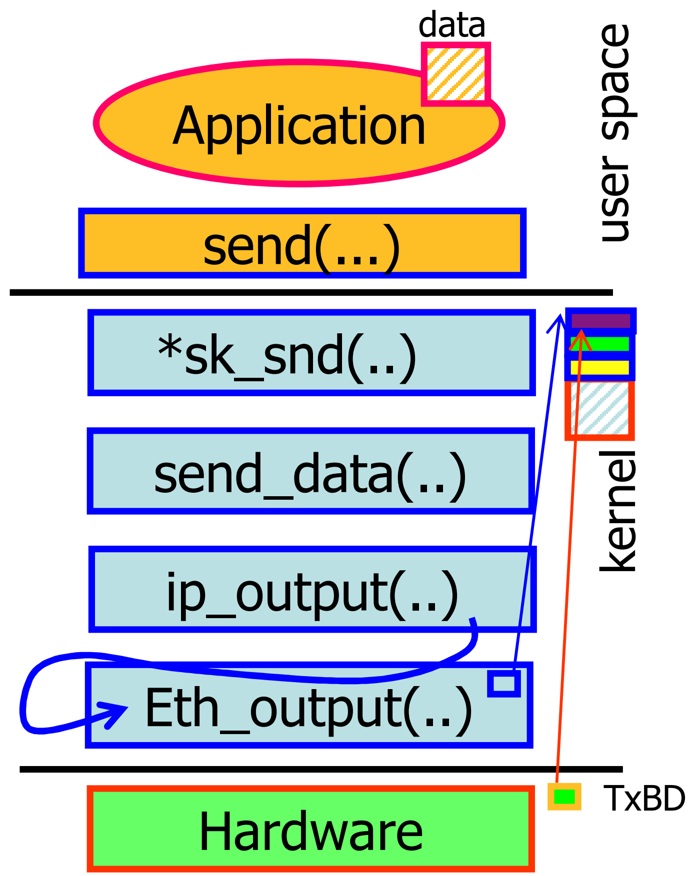
4. ip_output(...)
   - Gets destination IP address from TCP state data structure (layering violation)
   - Looks up that address in forwarding table to get a route

     (= logical interface + next hop IP address)
   - Prepends 20-byte IP header to TCP packet
   - Invokes the interface's output routine, bound to Eth_output( )



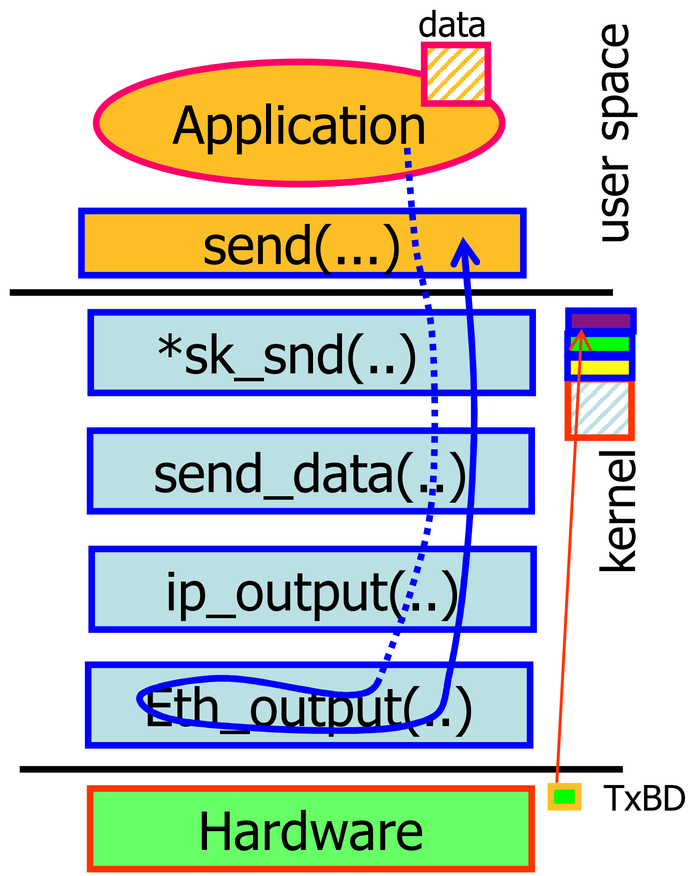
Anatomy of a Packet Transmission

4. Eth_output(...)
   - Resolves next-hop IP address to Ethernet address via ARP (may queue)
   - Prepend 14-byte Ethernet header (incl. dest addr)
   - If there is an available TxBufDescriptor, make it point to packet data
   - If necessary, start the hardware device
   - Free the kernel buffer hdr



Anatomy of a Packet Transmission

5. Control returns up the stack
   - Success indication returned to application program
6. Hardware eventually transmits packet per Ethernet protocol

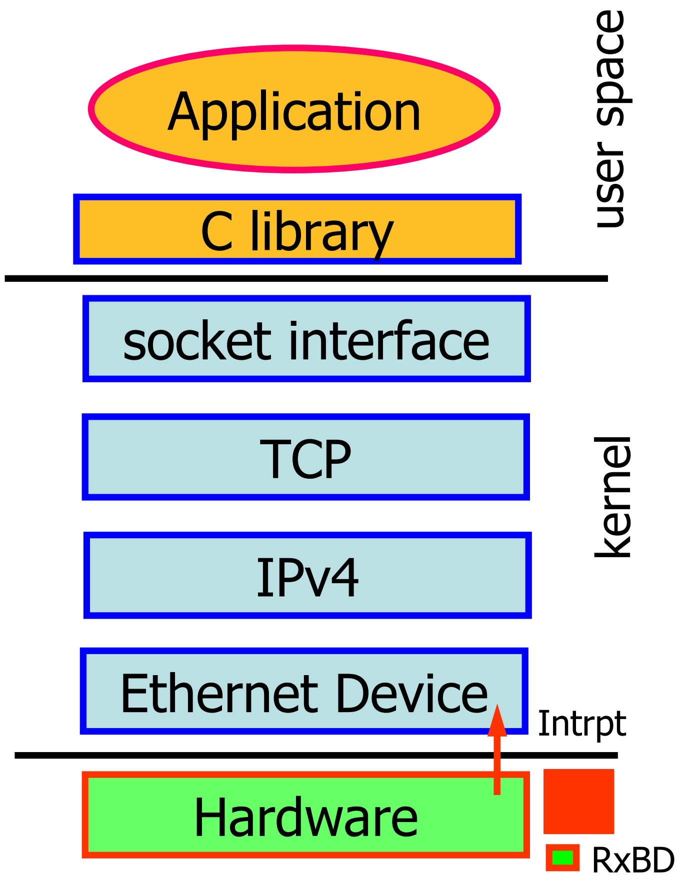


Packet Transmission: Highlights

- Message queued in at least two places:
  - Socket transmit queue
    - May wait if socket is flow-controlled at transport level
  - (Maybe) ARP queue
    - Waiting for reply from target
  - Device output queue
    - May wait if channel is busy
- Packet processing happens in single thread of control all the way to device driver
- When send( ) returns, message may or may not have been transmitted

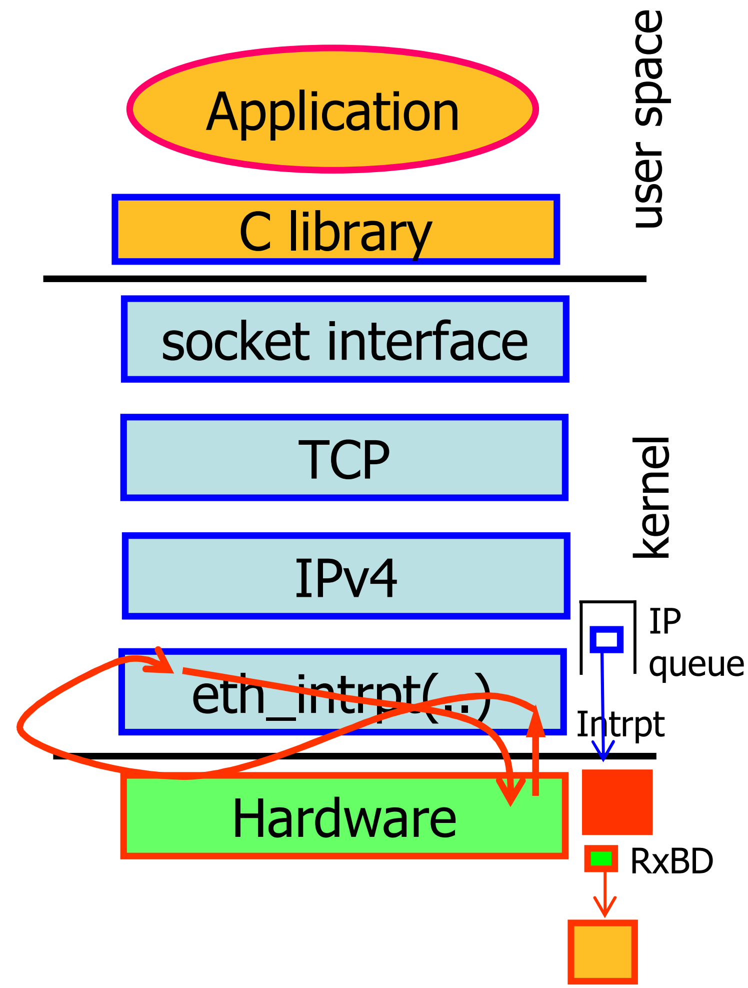
Anatomy of a Packet Reception

1. Hardware recognizes frame addressed to this station
   - Places packet into memory per next free RxBD
   - Hardware generates device interrupt



Anatomy of a Packet Reception

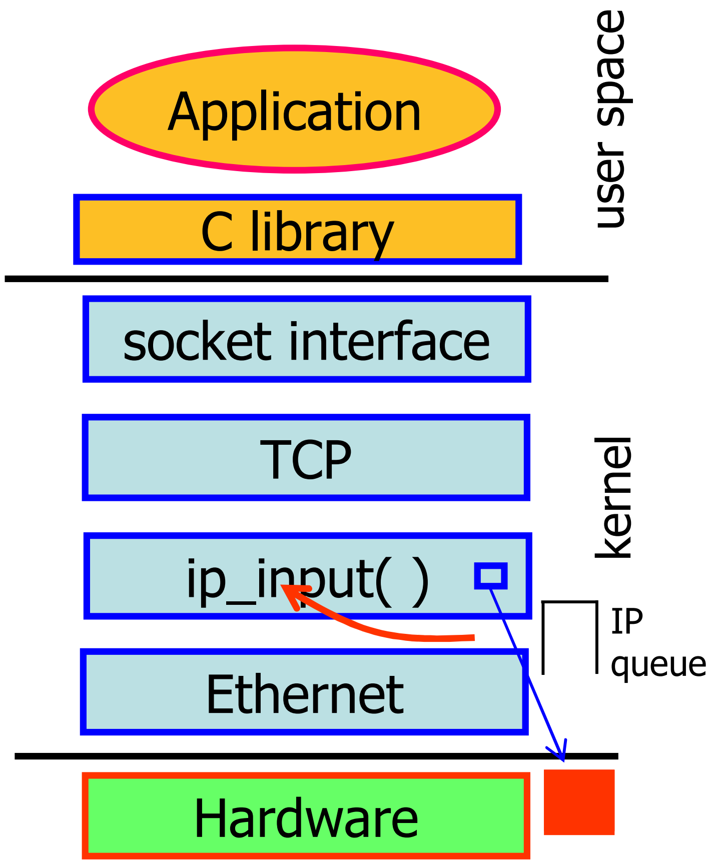
2. Current thread is interrupted; Ethernet interrupt service routine runs
   - Checks device status for errors
   - Verifies dest. address matches device address
   - Allocates kernel buffer hdr for packet data
   - Determines next protocol (IP)
   - Strips Ethernet header
   - Places buffer header in that protocol's input queue
   - Make RxBD point to a fresh buffer
   - Schedule the kernel net service thread to run
   - Return from interrupt; scheduler runs highest priority thread



Anatomy of a Packet Reception

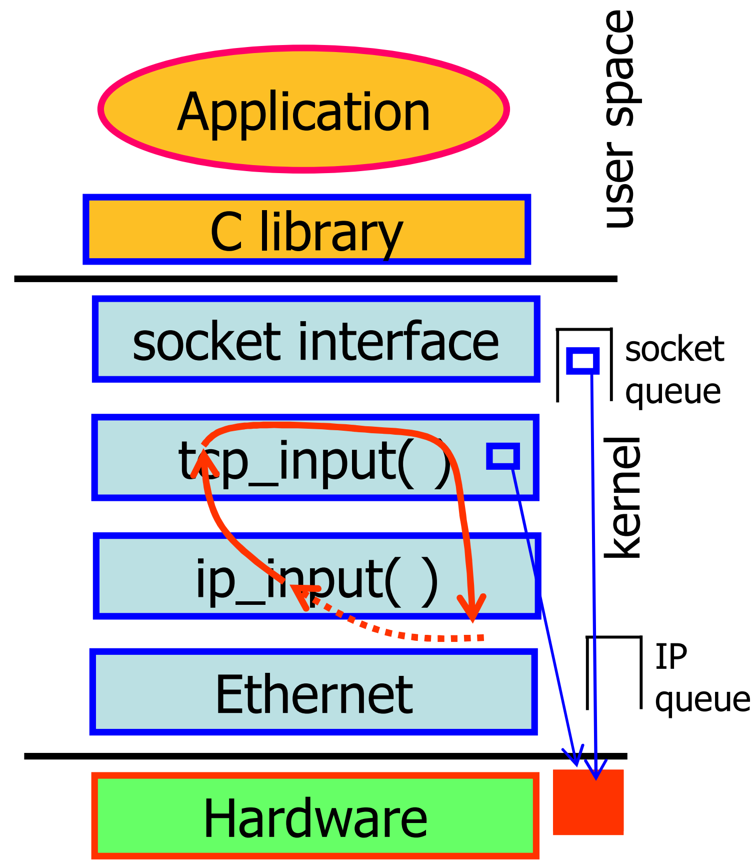
3. Net service thread detects nonempty queue, calls ip_input(), which:
   - Dequeues packet
   - Sanity-checks IP header
   - Checks that packet's destination IP address = one of this device's addresses
   - Determines next-higher protocol
   - Invokes that protocol's input routine indirectly via "switch table"

   In this case, the actual routine is
   tcp_input( )



Anatomy of a Packet Reception

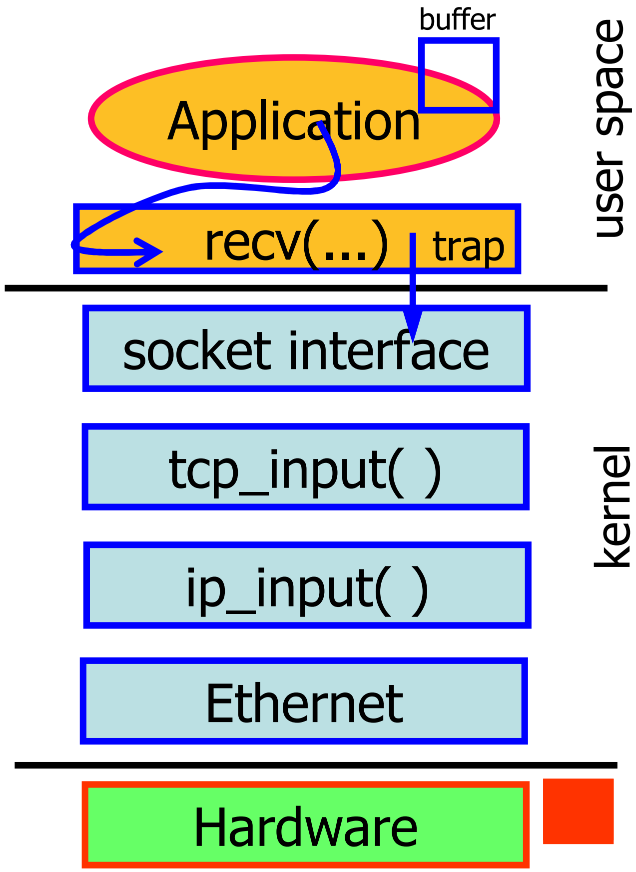
4. tcp_input( )
   - Retrieves relevant protocol state, using both IP and TCP headers
   - Determines if data is acceptable per TCP sliding window protocol
   - If so:
     - Strips IP + TCP headers by advancing buffer pointer
     - Retrieves associated socket state
     - Places packet payload in socket receive queue
     - If any application process is blocked on the queue, make it runnable
   - Returns
5. Net service thread blocks if no packet in net-level (IP) queue



Anatomy of a Packet Reception

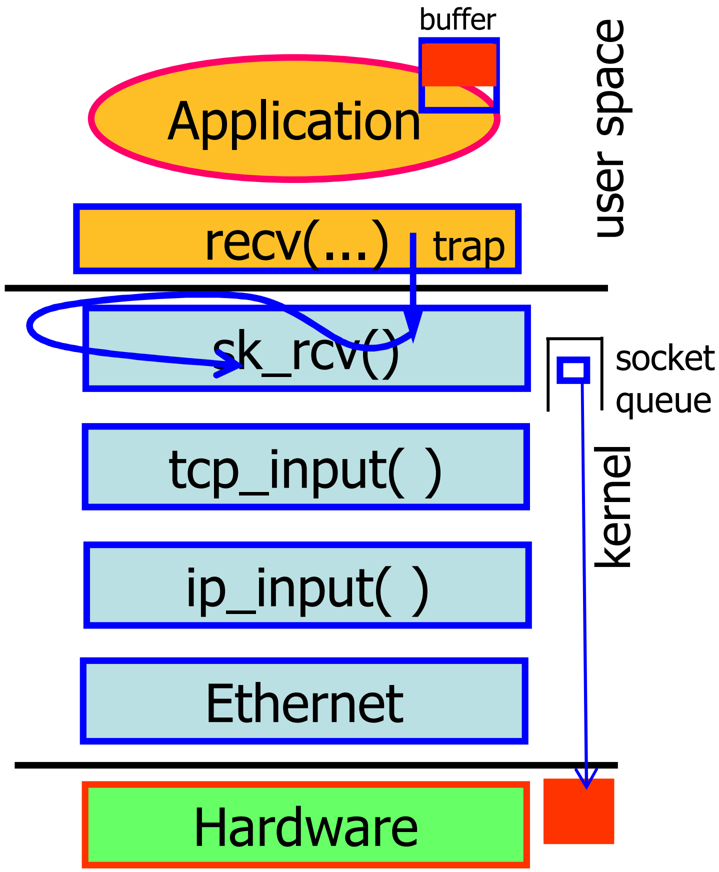
6. Application calls "recv(sock#, buffer, 1000")
   - Run-time C library implementation of recv() pushes arguments on stack
   - Implementation executes a "system call trap" instr.
   - Address of kernel trap svcing routine loaded into PC
   - Processor changes to privileged mode

Note: this step may happen before previous steps



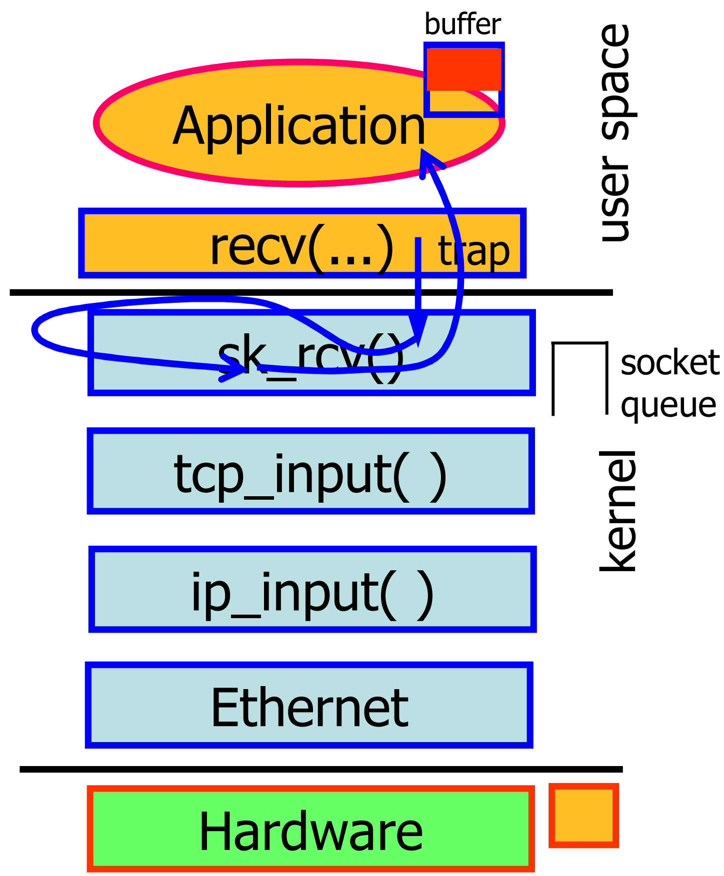
Anatomy of a Packet Reception

7. Trap handler invokes kernel implementation of recv( ) system call
   - Validates arguments (e.g., pointer is in the proc's address space)
   - Invokes socket's recv( ) function
   - Locates the state data structure for the socket
   - Verifies the socket state is OK to receive
   - If there is data in the socket queue
     - Copy it into the user's buffer; free kernel buffer header
     - Return
   - Else block until data arrives



Anatomy of a Packet Reception

7. Trap handler invokes kernel implementation of recv( ) system call
    - Validates arguments (e.g., pointer is in the proc's address space)
    - Invokes socket's recv( ) function
    - Locates the state data structure for the socket
    - Verifies the socket state is OK to receive
    - If there is data in the socket queue
        - Copy it into the user's buffer; free kernel buffer header and buffer
        - Return
    - Else block until data arrives



Packet Reception Highlights

- Control flows from the bottom up
- Three different threads of control
  - Hardware interrupt
    - Must run very fast because it blocks everything else
  - High-priority "network service" thread
    - Processes data via function calls upward through stack
  - User program
- Data must queue somewhere between hardware and user program
  - There exists an "Asynchronous-synchronous" interface
- In this example, there are two queues
  - IP input
  - User input

  (What happens when these queues get full?)

Packet Reception Highlights

- Some protocol layers have to determine which next-higher layer to invoke by looking at their <u>own</u> header information
  - Examples: Ethernet, IP
- Typically this is done <u>indirectly</u>, via a table of protocol functions
  - Header field value used as index into protocol table