Review: Hardware user/kernel boundary

- Processor can be in one of two modes
  - **user** mode – application software & libraries
  - **kernel** (supervisor/privileged) mode – for the OS kernel

- *Privileged* instructions only available in kernel mode
Execution contexts

- **User** – executing unprivileged application code
- **Top half** of kernel (a.k.a. “bottom half” in Linux)
  - Kernel code acting on behalf of current user-level process
  - System call, page fault handler, kernel-only process, etc.
  - This is the only kernel context where you can sleep (e.g., if you need to wait for more memory, a buffer, etc.)
- **Software interrupt**
  - Code not acting on behalf of current process
  - TCP/IP protocol processing
- **Device interrupt**
  - External hardware causes CPU to jump to OS entry point
  - Network interface cards generate these, disks too, …
- **Timer interrupt** (hardclock)
- **Context switch code** – change top half thread
Transitions between contexts

- **User → top half**: syscall, page fault
  - E.g., Read or write to a socket

- **User/top half → device/timer interrupt**: hardware
  - E.g., network transmission complete or a packet has arrived

- **Top half → user**: syscall return
  - Note syscall return can also go to context switch (e.g., if packet has arrived and made sleeping process runnable)

- **Top half → context switch**: sleep
  - E.g., User called read on a TCP socket, but no data yet

- **Context switch → user/top half**: return (to new ctx)
- Kernel gives each program its own context

- Isolates processes from each other
  - So one buggy process cannot crash others
Virtual memory

• Fault isolation between processes requires memory isolation

• Want each process to have its own view of memory
  - Otherwise, pain to allocate large contiguous structures
  - Processes can consume more than available memory
  - Dormant processes (waiting for event) still have core images

• Solution: Virtual Memory
  - Give each program its own address space—i.e., Address 0x8000 goes to different physical memory in $P_1$ and $P_2$
  - CPU must be in kernel mode to manipulate mappings
  - Isolation between processes is natural
Paging

- Divide memory up into small *pages*
- Map virtual pages to physical pages
  - Each process has separate mapping
- Allow OS to gain control on certain operations
  - Read-only pages trap to OS on write
  - Invalid pages trap to OS on write
  - OS can change mapping and resume application
- Other features sometimes found:
  - Hardware can set “dirty” bit
  - Control caching of page
Example: Paging on x86

- Page size is 4 KB (in most generally used mode)
- Page table: 1024 32-bit translations for 4 Megs of Virtual mem
- Page directory: 1024 pointers to page tables
- %cr3—page table base register
- %cr0—bits enable protection and paging
- INVLPDG – tell hardware page table modified
*32 bits aligned onto a 4–KByte boundary

1024 PDE × 1024 PTE = 2^{20} Pages
<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Positions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Base Address</td>
<td>0-31</td>
</tr>
<tr>
<td>Available</td>
<td>31</td>
</tr>
<tr>
<td>Global Page</td>
<td>30</td>
</tr>
<tr>
<td>Page Table Attribute Index</td>
<td>29-28</td>
</tr>
<tr>
<td>Dirty</td>
<td>27-20</td>
</tr>
<tr>
<td>Accessed</td>
<td>19-16</td>
</tr>
<tr>
<td>Cache Disabled</td>
<td>15-12</td>
</tr>
<tr>
<td>Write-Through</td>
<td>11-8</td>
</tr>
<tr>
<td>User/Supervisor</td>
<td>7-6</td>
</tr>
<tr>
<td>Read/Write</td>
<td>5-4</td>
</tr>
<tr>
<td>Present</td>
<td>3-0</td>
</tr>
</tbody>
</table>
• **Hardware has 64-entry TLB**
  - References to addresses not in TLB trap to kernel

• **Each TLB entry has the following fields:**
  Virtual page, Pid, Page frame, NC, D, V, Global

• **Kernel itself unpaged**
  - All of physical memory contiguously mapped in high VM
  - Kernel uses these pseudo-physical addresses

• **User TLB fault hander very efficient**
  - Two hardware registers reserved for it
  - utlb miss handler can itself fault—allow paged page tables
Memory and I/O buses

- CPU accesses physical memory over a bus
- Devices access memory over I/O bus with DMA
- Devices can appear to be a region of memory
Realistic Pentium architecture

- CPU
- North Bridge
  - AGP bus
  - PCI bus
  - USB
- South Bridge
  - ISA bus
- Main memory
  - Advanced Programmable Interrupt Controller (APIC)
  - PCI IRQs
What is memory

• **SRAM – Static RAM**
  - Like two NOT gates circularly wired input-to-output
  - 4–6 transistors per bit, actively holds its value
  - Very fast, used to cache slower memory

• **DRAM – Dynamic RAM**
  - A capacitor + gate, holds charge to indicate bit value
  - 1 transistor per bit – extremely dense storage
  - Charge leaks—need slow comparator to decide if bit 1 or 0
  - Must re-write charge after reading, or periodically refresh

• **VRAM – “Video RAM”**
  - Dual ported, can write while someone else reads
What an is I/O bus? E.g., PCI
Communicating with a device

- **Memory-mapped device registers**
  - Certain *physical* addresses correspond to device registers
  - Load/store gets status/sends instructions – not real memory

- **Device memory** – device may have memory OS can write to directly on other side of I/O bus

- **Special I/O instructions**
  - Some CPUs (e.g., x86) have special I/O instructions
  - Like load & store, but asserts special I/O pin on CPU
  - OS can allow user-mode access to I/O ports with finer granularity than page

- **DMA** – place instructions to card in main memory
  - Typically then need to “poke” card by writing to register
  - Overlaps unrelated computation with moving data over (typically slower than memory) I/O bus
• Include list of buffer locations in main memory
• Card reads list then accesses buffers (w. DMA)
  - Allows for scatter/gather I/O
• Link interface talks to wire/fiber/antenna
  - Typically does framing, link-layer CRC

• FIFOs on card provide small amount of buffering

• Bus interface logic uses DMA to move packets to and from buffers in main memory
Context switches are expensive

- Modern CPUs very complex
  - Deep pipelining to maximize use of functional units
  - Dynamically scheduled, speculative & out-of-order execution

- Most context switches flush the pipeline
  - E.g., after memory fault, can’t execute further instructions

- Reenabling hardware interrupts expensive

- Kernel must set up its execution context
  - Cannot trust user registers, especially stack pointer
  - Set up its own stack, save user registers

- Switching between address spaces expensive
  - Invalidates cached virtual memory translations
    (discussed in a few slides…)
…can affect networking performance

- Most OSes evolved from systems w/o network cards
- Context switch w. disk device not such a big deal
  - Lucky to complete 1,000 requests per second
  - Request rate proportional to application progress
- Network events much more frequent
  - E.g., 100,000s of packets per second quite possible
  - Could be non-flow controlled non-UDP traffic
  - So machine slowing down won’t slow down packets
Minimizing context switch overhead

- **TCP Push bit gives hint to OS**
  - Sender calls `write (tcpfd, buf, 0x10000);`
  - 64KB message broken into ∼ 45 TCP segments
  - Only the last one has TCP Push bit set
  - Don’t preempt another proc. on receiver until Push packet

- **Send packets in batches**
  - If NIC not busy, send packet immediately
  - If busy, queue up packets in OS, but don’t give to NIC
  - Send new batch of packets only when previous batch sent
  - Note: also reduces overhead of “poking” card

- **Receive packets in batches**
  - Check for next packet before returning from receive intr
Minimizing latency

• Don’t want optimizations to increase latency

• Recall TCP throughput depends on latency
  \[ \text{Rate} = \frac{W}{\text{RTT}} \approx \sqrt{\frac{8}{3p}} \]  for fixed loss rate \( p \)
  - RTT includes time to process packet & generate ACK

• Also want to recover fast from lost packets
  - Timeout is estimated RTT + 4 \cdot \text{deviation}
  - Lots of variance from receiver scheduler is bad idea

• So ACK received packets as soon as possible
  - Absolute priority to incoming packets at driver
  - “Softnet” fake interrupt level processes TCP/IP at higher priority than whatever process is running
  - So generates TCP ACKs w/o switching to receiving process
Driver architecture

- **Device driver provides several entry points to kernel**
  - Reset, output, interrupt, …

- **How should driver synchronize with card?**
  - Need to know when transmit buffers free or packets arrive

- **One approach: *Polling***
  - Sent a packet? Loop asking card when buffer is free
  - Waiting to receive? Keep asking card if it has packet

- **Disadvantages of polling**
  - Can’t use CPU for anything else while polling
  - Or schedule poll if future and do something else, but then high latency to receive packet
Interrupt driven devices

- Instead, ask card to interrupt CPU on events
  - Interrupt handler runs at high priority
  - Asks card what happened (xmit buffer free, new packet)
  - This is what most general-purpose OSes do

- Problem: Bad for high-throughput scenarios
  - Interrupts are very expensive (context switch)
  - Interrupts handlers have high priority
  - In worst case, can spend 100% of time in interrupt handler and never make any progress – *receive livelock*

- Best: Adaptive algorithm that switches between interrupts and polling
Socket implementation: buffering

• Need to be able to encapsulate data easily
  - E.g., add UDP header to data
  - Add IP header to UDP packet
  - Add Ethernet header to IP packet

• Need to de-encapsulate as well
  - Strip off headers before sending data up the layer stack

• Solution: Don’t store packets in contiguous memory

• BSD solution: mbufs
  - Small, fixed-size (256 byte) structures
  - Makes allocation/deallocation easy (no fragmentation)

• Mbufs working example for this lecture
  - Linux uses sk_buffs, which are similar idea
mbuf details

- Pkts made up of multiple mbufs
  - Chained together by m_next
  - Such linked mbufs called chains

- Chains linked w. m_nextpkt
  - Linked chains known as queues
  - E.g., device output queue

- Most mbufs have 108 data bytes
  - First in chain has pkt header

- Cluster mbufs have more data
  - ext header points to data
  - Up to 2 KB not collocated w. mbuf
  - m_dat not used

- m_flags or of various bits
  - E.g., if cluster, or if pkt header used
Adding/deleting data w. mbufs

- m_data always points to start of data
  - Can be m_dat, or ext.buf for cluster mbuf
  - Or can point into middle of that area

- To strip off a packet header (e.g., TCP/IP)
  - Increment m_data, decrement m_len

- To strip off end of packet
  - Decrement m_len

- Can add data to mbuf if buffer not full

- Otherwise, add data to chain
  - Chain new mbuf at head/tail of existing chain
mbuf utility functions

- mbuf *m_copym(mbuf *m, int off, int len, int wait);
  - Creates a copy of a subset of an mbuf chain
  - Doesn’t copy clusters, just increments reference count
  - wait says what to do if no memory (wait or return NULL)

- void m_adj(struct mbuf *mp, int len);
  - Trim |len| bytes from head or (if negative) tail of chain

- mbuf *m_pullup(struct mbuf *n, int len);
  - Put first len bytes of chain contiguously into first mbuf

- Example: Ethernet packet containing IP datagram
  - Trim Ethernet header w. m_adj
  - Call m_pullup (n, sizeof (ip_hdr));
  - Access IP header as regular C data structure
Socket implementation

- Each socket fd has associated socket structure with:
  - Send and receive buffers
  - Queues of incoming connections (on listen socket)
  - A protocol control block (PCB)
  - A protocol handle (struct protosw *)

- PCB contains protocol-specific info. E.g., for TCP:
  - Pointer to IP TCB w. source/destination IP address and port
  - Information about received packets & position in stream
  - Information about unacknowledged sent packets
  - Information about timeouts
  - Information about connection state (setup/teardown)
protosw structure

- **Goal: abstract away differences between protocols**
  - In C++, might use virtual functions on a generic socket struct
  - Here just put function pointers in protosw structure

- **Also includes a few data fields**
  - *type, domain, protocol* – to match socket syscall args, so know which protosw to select
  - *flags* – to specify important properties of protocol

- **Some protocol flags:**
  - **ATOMIC** – exchange atomic messages only (like UDP, not TCP)
  - **ADDR** – address given w. messages (like unconnected UDP)
  - **CONNREQUIRED** – requires connection (like TCP)
  - **WANTRCVD** – notify socket of consumed data (e.g., so TCP can wake up a sending process blocked by flow control)
protosw functions

• pr_slowtimo – called every 1/2 sec for timeout processing
• pr_drain – called when system low on space
• pr_input – takes mbuf chain of data to be read from socket
• pr_output – takes mbuf chain of data written to socket
• pr_usrreq – multi-purpose user-request hook
  - Used for bind/listen/accept/connect/disconnect operations
  - Used for out-of-band data
  - Various other control operations
Network interface cards

• Each NIC driver provides an `ifnet` data structure
  - Like `protosw`, tries to abstract away the details

• Data fields:
  - Interface name (e.g., “eth0”)
  - Address list (e.g., Ethernet address, broadcast address, …)
  - Maximum packet size
  - Send queue

• Function pointers
  - `if_output` – prepend header, enqueue packet
  - `if_start` – start transmitting queued packets
  - Also `ioctl`, `timeout`, `initialize`, `reset`
Optimizations

- **Computing TCP/UDP checksum has overhead**
  - Fold checksum computation into copy
  - Fancy NICs can compute in hardware (layer violation)

- **Copying data is expensive**
  - Blows out your processor cache
  - Not that many times faster than really fast network
    (E.g., 8 GB/sec in L1, 3 GB/sec in L2, 1 GB/sec to mem)
  - Optimize by keeping to 1 or even 0 copies of data

- **Several research systems change network interface**
  - Have NIC DMA directly into application memory (changes alignment requirements, or have hardware demultiplex)

- **Optimize for the common case**
  - Structure code so path for next expected TCP packet shortest