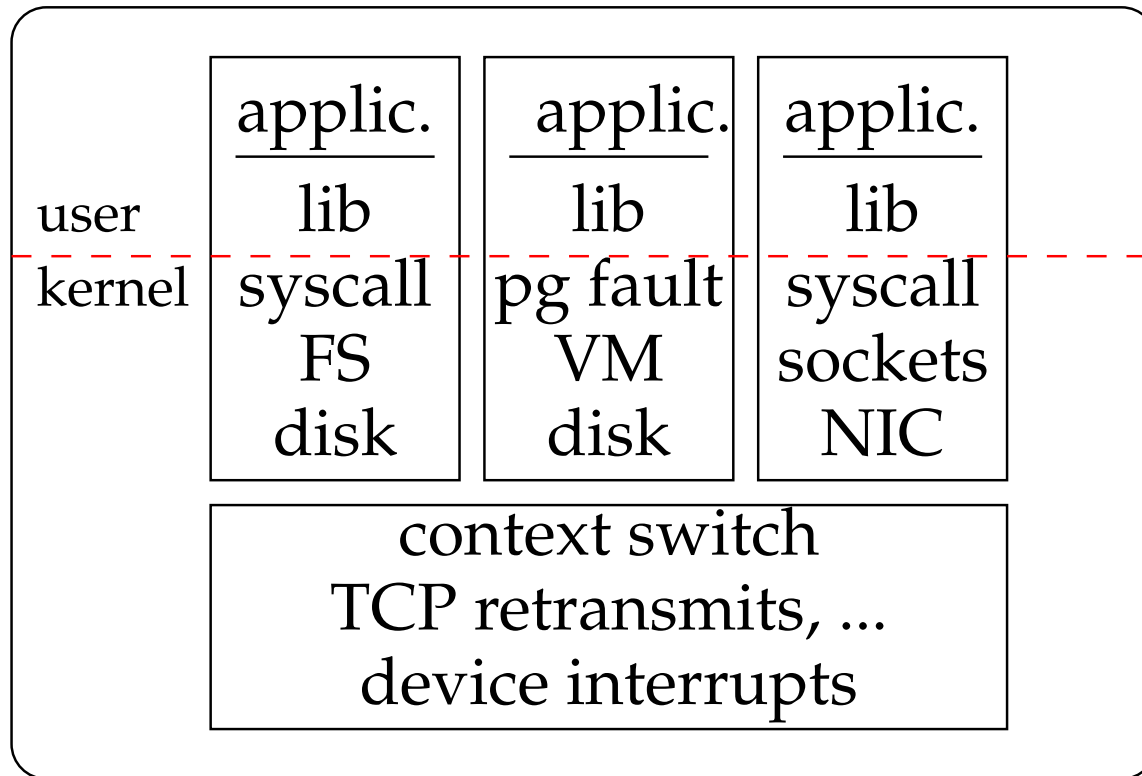


# OS Interfaces

- **Between applications and the OS.**
  - read(), write(), etc.
  - sendfile(), IO Lite
- **Between the OS and hardware.**
  - How is data moved? (PIO, DMA)
  - How is operation synchronized? (interrupts, polling)
- **We'll start at the bottom.**

# 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**

# Kernel execution contexts

- **Top half** of kernel
  - 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.)
- **Device interrupt – Network, disk, timer...**
  - External hardware causes CPU to jump to OS entry point
- **Software interrupt – Finishes the work implied by a hardware interrupt. *E.g.* TCP/IP processing.**
- **Context switch code**
  - Switches current process (thread switch for top half)

## Transitions between contexts

- **User → top half: syscall, page fault**
- **User/top half → device/timer interrupt: hardware**
- **Top half → user/context switch: return**
- **Top half → context switch: sleep**
- **Context switch → user/top half**

# Context switches are expensive

- **Modern CPUs very complex**
  - Deep pipelining to maximize use of functional units
  - Dynamically scheduled, speculative & out-of-order execution
- **Context switches flush the pipeline**
  - *E.g.*, after memory fault, can't execute further instructions
- **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.

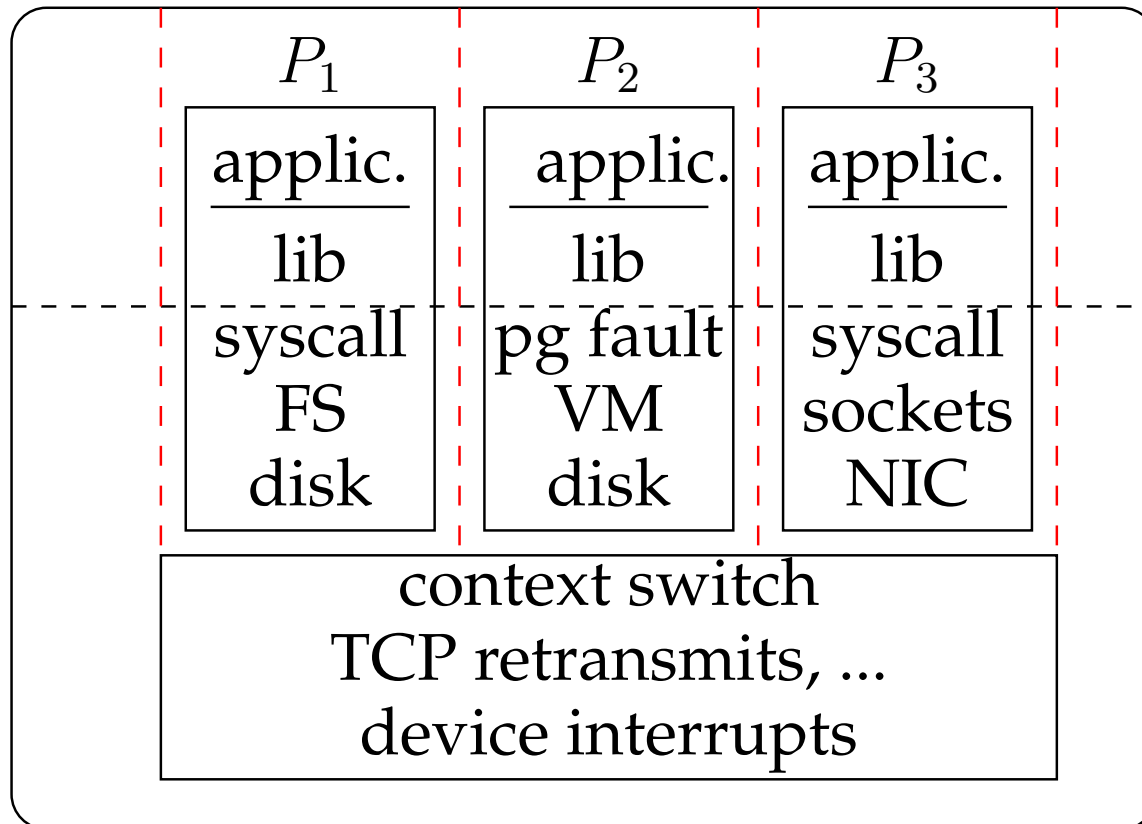
# Top/bottom half synchronization

- **Top half kernel procedures can mask interrupts**

```
int x = splhigh ();           /* Set Priority Level High */
/* atomic operation */
splx (x);                    /* Return to previous level */
```

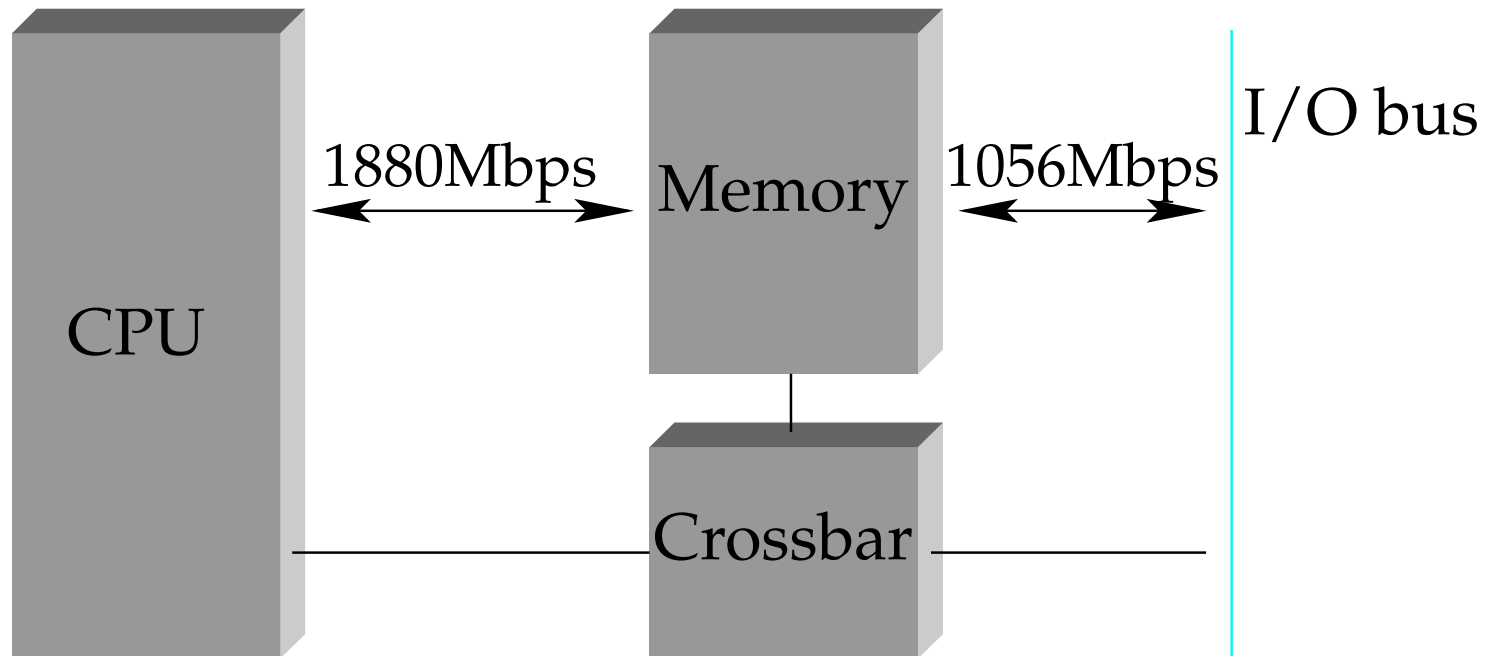
- **splhigh disables all interrupts, but also have splnet, splbio, splsoftnet, ...**
  - Hardware interrupt handler sets mask automatically
- **Example: splnet implies splsoftnet**
  - Ethernet packet arrives, driver code flags TCP/IP needed
  - Upon return from handler, TCP code invoked at softnet
- **Masking interrupts in hardware can be expensive**
  - Optimistic implementation – set in-memory flag on splhigh, check flag before executing any interrupt handler.

# Process context



- Kernel gives each program its own context
- Isolates processes from each other
  - So one buggy process cannot crash others

## Memory and I/O buses

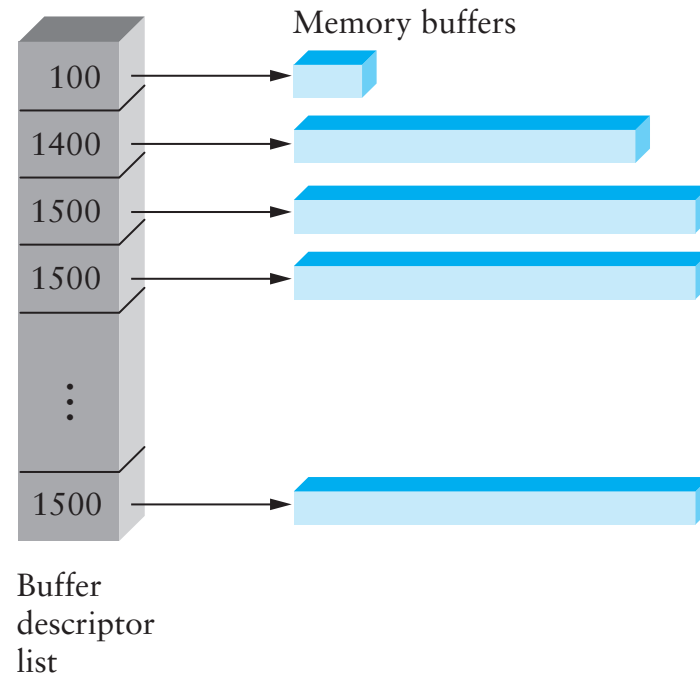


- CPU accesses physical memory over a bus
- Devices access memory over I/O bus with DMA
- Devices can appear to be a region of memory

# Communicating with a device

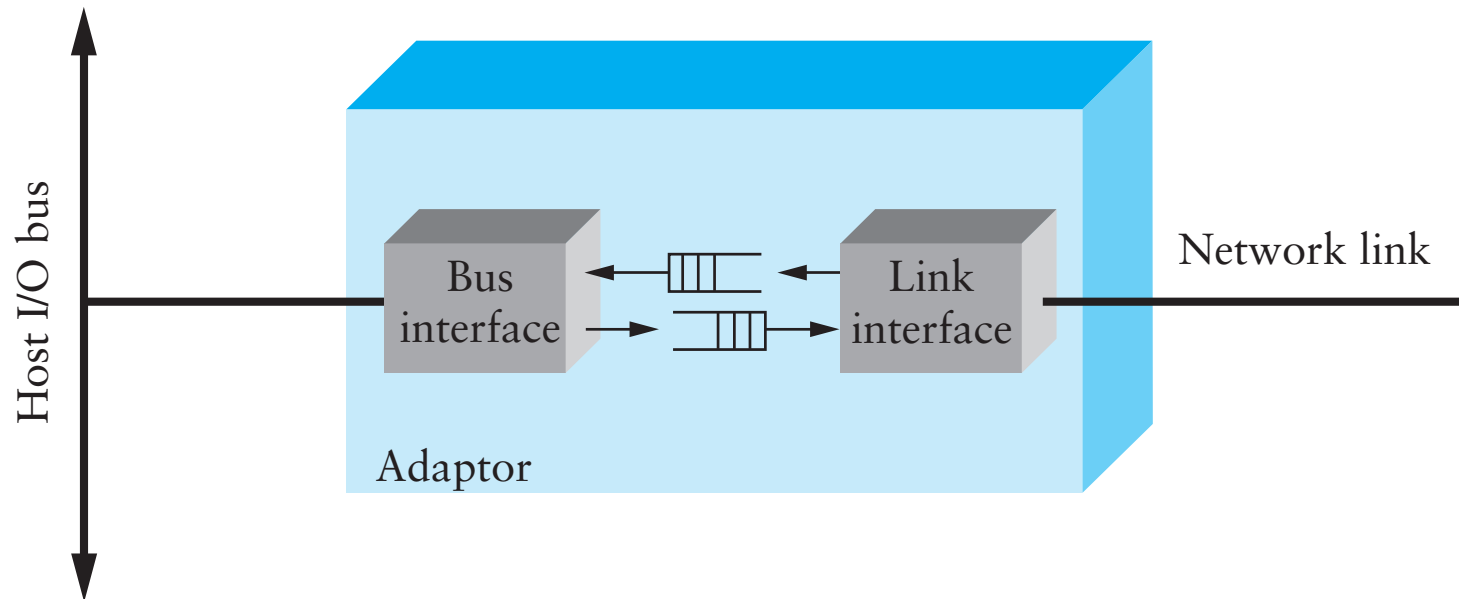
- **Memory-mapped device registers**
  - Certain *physical* addresses correspond to device registers
  - Load/store gets status/sends instructions – not real memory
- **Device memory – device may have memory OS can write to directly on other side of I/O bus**
- **Special I/O instructions**
  - Some CPUs (*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**
  - Then “poke” card by writing to register.

# DMA buffers



- **Include list of buffer locations in main memory**
- **Card reads list then accesses buffers (with DMA)**
  - Allows for scatter/gather I/O

# Anatomy of a Network Interface Card



- **Link interface talks to wire/fiber/antenna**
  - Typically does framing, link-layer CRC
- **FIFOs on card provide small amount of buffering**
- **Bus interface logic moves packets to/from memory or CPU**

# Driver architecture

- **Device driver provides several entry points to kernel**
  - Reset, output, interrupt, ...
  - An “object-oriented” approach, even in C.
  - Each driver supplies a table of function pointers.
- **How should driver synchronize with card?**
  - Driver tells card to transmit, when is it done?
  - When do new packets arrive?

# Interrupt driven devices

- **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
  - Interrupts handlers have high priority
  - What happens when packets arrive too fast?

# Typical Interface: Interrupt driven DMA

- **Tx\_ready (host→card)**
  - Here's a buffer address, transmit it.
- **Tx\_free (card→host)**
  - I'm done with this buffer you gave me.
- **Rx\_ready (card→host)**
  - There's now a packet in this buffer.
- **Rx\_free (host→card)**
  - Here's a free buffer to put a received packet in.

# Livelock

- Livelock is like deadlock but work is done constantly (yet nothing useful is accomplished).
- *Receive Livelock*
  - Interrupts arrive so fast that repeated calls to the interrupt handler consume the CPU.
  - Soft interrupts have no time to run.
  - CPU is exhausted before *real* packet processing can occur.
  - Turn off interrupts?

# Device Polling

- **Another approach: *Polling***
- **Sent a packet? Loop, asking when buffer is free.**
- **Waiting to receive? Loop, asking for packet.**
- **Advantages of polling**
  - No Livelock: Device can't interrupt processing.
  - Batching possible: One poll for several packets.
- **Disadvantages of polling**
  - If busy-waiting, can't use CPU for anything else.
  - If periodically checking, high latency to receive packet.
- **Hybrid schemes get best of both.**

# Application to OS

- **Typical Send**

- Application makes send() system call.
- Data copied from user space into kernel buffers, Tx\_ready
- Data eventually transmitted, Tx\_free

- **Typical Receive**

- Card receives data: Rx\_ready
- Data copied from card buffer to kernel buffers: Rx\_free
- Application calls recv() (at some point)
- Data copied from kernel buffers to user-space

# Data copying matters

- **Recall similarity of Memory and I/O bandwidth**
- **$n$  copies or scans reduce bandwidth by factor of  $n$ .**
- **“Zero-copy” interfaces improve upon read/write**
  - Sendfile() – Sends data from one fd to another, usually disk to network.
  - IO Lite – Use data handles to allow user-space management without moving data to user-space.
  - Exokernel – Expose a device-driver like interface to applications
  - What about checksums?

## Coming up

- **Tomorrow, 4pm: cs168\_handin ip-milestone**
- **Tue 24th: Routing (larger-scale)**
- **Thu 26th: DNS**
- **Fri 27th, IP due**