• **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.)

- **Software interrupt – TCP/IP protocol processing**

- **Device interrupt – Network, disk, …**
  - External hardware causes CPU to jump to OS entry point

- **Timer interrupt (hardclock)**

- **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

- 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…)
Top/bottom half synchronization

- Top half kernel procedures can mask interrupts

```c
int x = splhigh();
/* ... */
splx(x);
```

- 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 mask flag on splhigh, check interrupted flag on splx
Kernel gives each program its own context

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

- Need fault isolation between processes
- 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
- INVLPG – tell hardware page table modified
*32 bits aligned onto a 4-KByte boundary
Page-Table Entry (4-KByte Page)

<table>
<thead>
<tr>
<th>31</th>
<th>12</th>
<th>11</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avail</td>
<td>G</td>
<td>P</td>
<td>T</td>
<td>D</td>
<td>A</td>
<td>P</td>
<td>C</td>
<td>D</td>
<td>P</td>
<td>W</td>
<td>T</td>
</tr>
</tbody>
</table>

Available for system programmer’s use
Global Page
Page Table Attribute Index
Dirty
Accessed
Cache Disabled
Write-Through
User/Supervisor
Read/Write
Present
Example: MIPS

- **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
• 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
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
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
DMA buffers

- Include list of buffer locations in main memory
- Card reads list then accesses buffers (w. 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, …

- 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**