Operating System & Virtual Memory

Kunchang Guo

May 22, 2025

Overview

1. An Overview of OS
2. Process & CPU Management
3. Virtual Memory

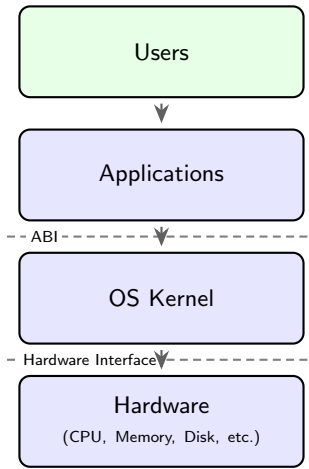
What does an Operating System (OS) do?

A Layered View:

- Manages hardware resources.
- Provides abstractions to applications.
- Offers a consistent interface.

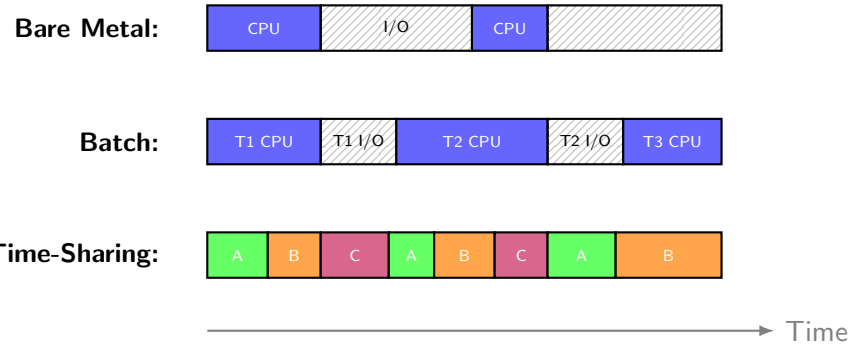
Key Roles of an OS:

1. **Abstraction**: Hides hardware complexity (e.g., files vs. disk blocks).
2. **Resource Management**: Shares CPU, memory, I/O devices fairly and efficiently.
3. **Protection & Isolation**: Prevents interference between users/programs.
4. **Standardized Interface** (e.g., System Calls, ABI): Simplifies application development.



(Conceptual OS Layers)

Computing paradigms



Key Idea

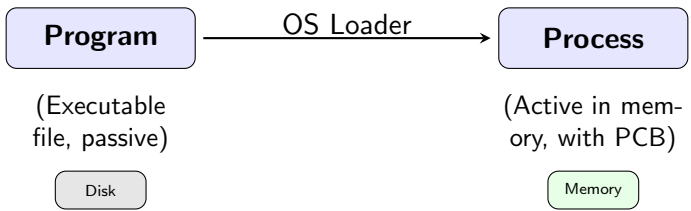
The evolution aims for better tradeoff between resource utilization and performance.

Defining a "Task": The Process

In a time-sharing system, the OS manages multiple concurrent tasks.

Program vs. Process:

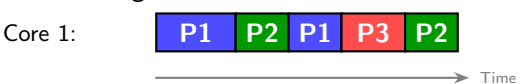
- **Program**: Passive entity stored on disk (e.g., executable file)
- **Process**: Active entity with:
  - Address space (code, data, stack, heap)
  - Execution state (CPU registers like `pc`, `sp`, GPRs)
  - Resources (files, sockets)
  - Managed via PCB (Process Control Block)



Concurrency vs. Parallelism

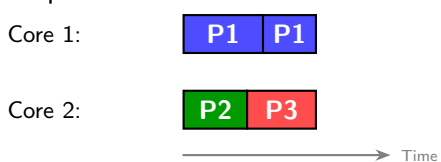
Concurrency
(Illusion of Simultaneousness)

- Multi-processes make progress over time
- On a single core: Processes are interleaved



Parallelism (Actual Simultaneousness)

- Multiple processes (or threads) execute truly at the same instant
- Requires multi-cores.



Modern OS supports both, but concurrency is fundamental even on single core.

System Calls: The Gateway to the Kernel

- A system call is the programmatic way a user program requests a service from the OS kernel.
- Acts as the Application Programming Interface (API) to the kernel.
- Triggers a mode switch from *user mode* to *kernel mode*.

Examples (Unix-like):

- `fork()`: Create a new child process.
- `exec()`: Replace current process image with a new program.
- `pipe()`: Create a data channel for Inter-Process Communication (IPC).
- `read()/write()`: File I/O.
- `sbrk()/mmap()`: Memory management.
- `exit()`: Terminate process.

User Mode vs. Kernel Mode

User Mode (Unprivileged)

- Applications run here
- Restricted memory access
- Cannot execute privileged instructions
- Attempts trigger CPU exceptions
- Protected from each other

Kernel Mode (Privileged / Supervisor)

- OS Kernel runs here
- Full access to memory and hardware
- Can execute all CPU instructions

Why this separation?

Provides isolation that prevents user programs from crashing the entire system.

The Trap: Software Interrupts

How does a user program transition to kernel mode to execute OS code?

- Via the RISC-V `ECALL` instruction (Environment Call).

The RISC-V Trap Mechanism:

1. User program executes `ECALL` (e.g., after library call like 'read()').
   - System call number placed in a7 (x17) register.
   - Arguments placed in a0-a6 (x10-x16) registers.
2. CPU switches from User Mode to Supervisor Mode.
3. CPU saves current program state in `sepc` (Supervisor Exception Program Counter), and other CSRs like `scause`, `stval` record cause/details.
4. CPU jumps to the address specified in `stvec` (Supervisor Trap Vector).
5. Kernel's trap handler executes the requested service.
6. OS executes `SRET` (Supervisor Return) instruction.
7. CPU switches back to User Mode.
8. CPU restores saved program counter from `sepc`.
9. User program resumes execution after the ECALL.

The Illusion of Many CPUs on RISC-V

On a single-core processor, how does the OS run multiple processes "simultaneously"?

- By rapidly switching the CPU between processes – **Context Switching**.
- Each process gets a small time slice to run.

What triggers a context switch on RISC-V?

- **Voluntary**:
  - Process makes an `ECALL` that blocks (e.g., `read()` from empty pipe).
  - Process explicitly yields the CPU using `sched_yield()` system call.
  - Process terminates (via `exit`).
- **Involuntary (Preemption)**:
  - **Timer Interrupt**: When a timer interrupt comes, decided by corresponding handler.
  - Higher priority process becomes ready in scheduler queue.

Context Switching: The Mechanism

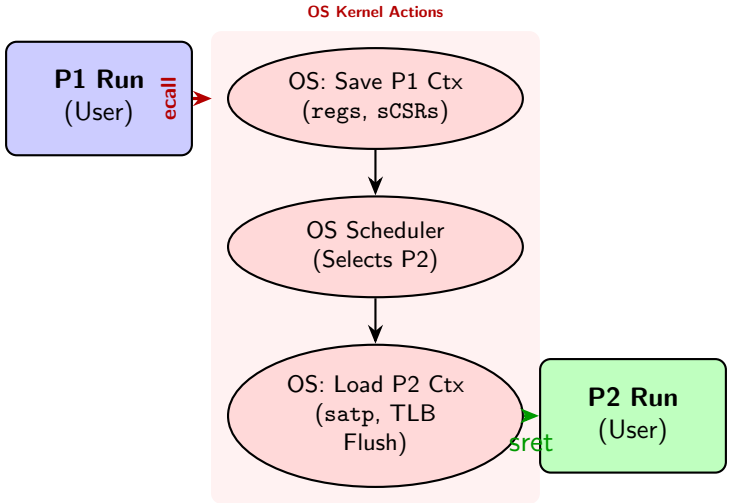
What is saved and restored? The process's entire execution context.

1. **Save Context of Current Process (P1):**
   - RISC-V general-purpose registers (x1-x31) $\rightarrow$ P1's PCB.
   - Save supervisor CSRs (sstatus, sepc, etc.) if necessary.
   - Update P1's state in PCB (e.g., from Running to Ready/Waiting).
2. **Select Next Process (P2):**
   - OS Scheduler chooses P2 from the Ready Queue based on scheduling policy.
3. **Restore Context of Next Process (P2):**
   - Load P2's saved registers from its PCB $\rightarrow$ CPU registers.
   - *Crucially: Update* **satp** *register to point to P2's page table.*
   - Flush TLB entries.
   - Update P2's state in PCB (from Ready to Running).
4. **Resume P2:** Execute SRET instruction or jump to P2's saved program counter.

Overhead

Register saves/restores (32+ registers), TLB flushes, and cache pollution.

Context Switching: The Mechanism



(Context switch: P1 → OS → P2)

Context Switching: Policies

The "mechanism" is how switching happens. The "policy" decides when and to whom to switch. **CPU Scheduling Policies (Examples - not exhaustive):**

- **First-Come, First-Served (FCFS)**: Simple, but can lead to convoy effect.
- **Shortest Job First (SJF)**: Optimal average turnaround, but needs future knowledge.
- **Priority Scheduling**: Important jobs run first; risk of starvation.
- **Round Robin (RR)**: Fair, good for time-sharing. Time slice (quantum) is critical.
- **Multilevel Feedback Queues (MLFQ)**: Adaptive, widely used.

The Programmer's Ideal Memory View

From an application programmer's perspective, memory should be:

- **Large**: Enough space for code, data, stack, heap.
- **Contiguous**: A single, unbroken range of addresses.
- **Private/Isolated**: My program's memory is mine alone, safe from others.
- **Starts at 0 (or a known address)**: Simplifies linking and loading.

Challenges of Direct Physical Memory Access

If programs directly accessed physical RAM:

- **Limited Capacity**: Physical RAM is finite. How to run many/large programs?
- **Address Relocation/Positioning**:
  - If multiple programs, where does each start in RAM?
  - Addresses compiled into program might conflict or be invalid.
- **Fragmentation**:
  - **External**: Unused memory holes too small for new programs.
  - **Internal**: Allocated block larger than needed, waste within block.
- **Protection/Isolation**:
  - No way to prevent one program from corrupting another or the OS.
  - A single bug can crash the entire system.

Virtual Memory

Core Idea: OS gives each process its own private, virtual address space.

- Appears large, contiguous, starting from 0.
- OS + MMU translate virtual addresses to physical addresses dynamically.

Key Benefits:

- **Illusion of More Memory**: Can run programs larger than physical RAM.
- **Protection**: Processes isolated in their own VAS.
- **Simplified Programming**: Programmers use simple, consistent virtual addresses.
- **Efficient Sharing**: Controlled sharing of physical memory (e.g., libraries).
- **Flexibility**: Programs can be loaded anywhere in physical RAM.

Paging

Observation: Programs often don't use their entire address space at once.

- Only a small "working set" of memory is actively used.

Paging Idea:

- Divide **virtual address space** into fixed-size blocks – **pages** (e.g., 4KB, 2MB).
- Divide **physical memory** into same-sized blocks – **page frames**.
- OS manages mapping from virtual pages to physical page frames.
  - A page can be in RAM or on disk (swapped out).

Advantages of Paging:

- **Solves External Fragmentation**: Any free frame can be used for any page.
- **Efficient for Sparse Address Spaces**: Only allocate frames for pages in use.
- **Demand Paging**: Load pages into RAM only when accessed (on page fault).

Note: Internal fragmentation can still occur within the last page of a segment.

Virtual to Physical Address Translation

The MMU (Memory Management Unit) translates *virtual addresses* (VAs) to *physical addresses* (PAs).

The Page Table

A per-process data structure, pointed to by a root register (e.g., `satp` in RISC-V), used by the MMU.

- Contains mappings from Virtual Page Numbers (VPNs) to Physical Page Numbers (PPNs).
- Page Table Entries (PTEs) also hold control bits (e.g., Valid, R/W, U, D, A).

A virtual address is typically split:

```
Virtual Address = [Virtual Page Number (VPN)] [Page Offset]
```

The MMU uses the VPN to find the PPN via the Page Table, then:

```
Physical Address = [Physical Page Number (PPN)] [Page Offset]
```

Page Table Entries

A RISC-V PTE (e.g., for SV39/SV48) contains more than just the PPN:

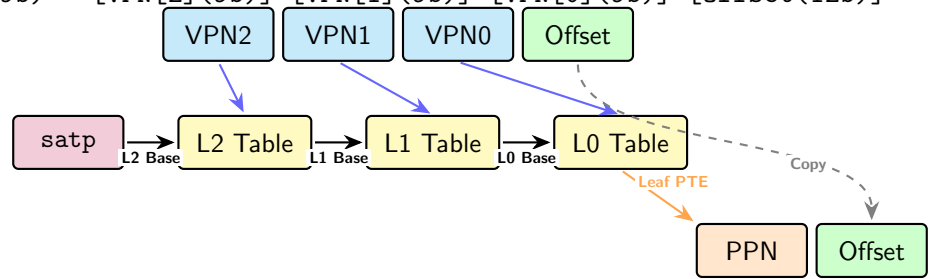
- **Valid bit (V)**: Is this PTE valid and the page present in a physical frame?
- **Read bit (R)**: Is reading from this page allowed?
- **Write bit (W)**: Is writing to this page allowed?
- **Execute bit (X)**: Is executing code from this page allowed?
- **User Mode Accessible bit (U)**: Can user-mode code access this page?
- **Accessed bit (A)**: Has this page been accessed since A was last cleared?
- **Dirty bit (D)**: Has this page been written to since D was last cleared?
- **RSW (Reserved for Supervisor Software)**: 2 bits for OS use.
- **Physical Page Number (PPN)**: If V=1 and R/W/X bits don't indicate a pointer PTE, this is the frame number.
- *Note: If R=0, W=0, X=0, the PTE is a pointer to the next-level page table. Otherwise, it's a leaf PTE.*

The MMU checks these bits on every memory access

Enforcing protection and triggering faults if necessary.

RISC-V SV39 Page Walk

```
VA (39b) = [VPN[2](9b)] [VPN[1](9b)] [VPN[0](9b)] [Offset(12b)]
```



Translation Lookaside Buffer (TLB)

TLB: A small, fast, hardware-managed cache on the CPU chip.

- Stores recent **VPN** $\rightarrow$ **PPN mappings** (and protection bits).
- Modern RISC-V implementations typically have hardware-managed TLBs.
- May include ASID tags to distinguish between address spaces.

Translation Process with TLB:

1. CPU generates VA. Extract VPN.
2. **Check TLB**
   - **TLB Hit**: Get PPN from TLB, form PA = PPN + offset.
   - **TLB Miss**:
     2.1 Performs page table walk.
     2.2 If PTE valid: Load translation into TLB, form physical address.
     2.3 If PTE invalid: Generate a page fault exception.

Memory Protection with Paging

Private Page Tables

- Each process has its own page table structure.

RISC-V PTE Protection Bits:

- **R/W/X bits**: Control read/write/execute permissions.
- **U bit**: When 0, only supervisor mode can access the page.
- Attempt to write to a read-only page → Store Page Fault.
- User mode attempt to access a supervisor page → Load/Store/Instruction Page Fault.
- Attempt to execute from non-executable page → Instruction Page Fault.

PMP (Physical Memory Protection)

- Additional RISC-V feature that can restrict even supervisor access to physical memory regions.

Memory Sharing with Paging

Shared Libraries:

- Multiple processes map the same physical frames into their different VAS.
- Typically set with R-X permissions (readable, executable, but not writable).

Inter-Process Communication:

- OS maps same physical data page(s) with R-W permissions for multiple processes.

Copy-on-Write for `fork()`:

- Initially parent and child share all pages with R– permissions.
- Write attempt triggers page fault; OS makes a private copy and sets R-W permissions.

Context Switching: Memory & TLB in RISC-V

When switching from Process P1 to Process P2:

1. **Save/Load Process Context** - CPU registers, etc.
2. **Switch Address Spaces:**
   - Update `satp` register with P2's root page table physical address and its ASID.
3. **Handle the TLB:**
   - Execute `SFENCE.VMA` instruction to flush TLB entries.
   - **With ASIDs**:
     - TLB entries include ASID tags.
     - `SFENCE.VMA` can target specific ASID.

Further Exploration

Many important topics are not covered, such as:

- File Systems (Organization, Implementation, Performance)
- I/O Device Management and Drivers
- Deadlocks
- Security features beyond basic protection
- And much more...

Recommended Reading for Deeper Understanding:

- **Operating Systems: Three Easy Pieces** (OSTEP) by Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau.
  - Website: `http://pages.cs.wisc.edu/~remzi/OSTEP/`
- **xv6: a simple, Unix-like teaching operating system** (and its accompanying book).
  - Website: `https://pdos.csail.mit.edu/6.828/2024/xv6.html`
  - Book: `https://pdos.csail.mit.edu/6.828/2024/xv6/book-riscv-rev4.pdf`