From Programs to Virtual Memory

How an Operating System Builds Useful Illusions

Discussion 14

June 4, 2026

Today's Route

1. The OS as an Abstraction Builder
2. Programs and Processes
3. System Calls and OS Objects
4. Address Spaces
5. Virtual Memory Mechanics
6. Scheduling and Time-Sharing
7. Discussion Wrap-up

Not a Definition, but a Point of View

A useful working model

An operating system is the program that turns raw hardware into a world where applications can run comfortably.

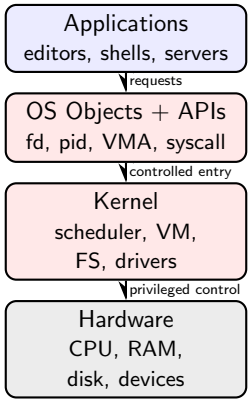
From the application side

- Processes, files, sockets, pipes
- Stable APIs: `fork`, `execve`, `read`, `mmap`
- A private world for each process

From the hardware side

- CPU only executes instructions
- Memory is just bytes
- Devices expose registers and interrupts

The Boundary the OS Builds

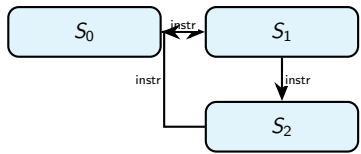


Key idea

The OS is mostly about boundaries: who can see what, who can touch what, and how requests cross the boundary.

Program: A State Machine Description

- Program text describes states and transitions.
- At runtime, the CPU repeatedly applies one transition: execute the next instruction.
- There is no magic: the machine state changes step by step.



Useful question

If the CPU only executes instructions, how does a program stop, print, read a file, or create another program?

What `main()` Hides

```c
#include <stdio.h>

int main() {
    printf("hello\n");
    return 0;
}
```

- This is how most courses start: write `main()`, call a library function, return.
- But the CPU does not know `main`; the executable has an entry point.
- The C runtime calls `main`, then turns its return value into process termination.

Discussion

If `main()` returns, who receives the return value?

Demo Bridge: A Tiny C Program Is Already an OS Client

```c
#include <stdio.h>

int main(void) {
    printf("Hello, world!\n");
    return 0;
}
```

What looks simple

- print one line
- return zero

What is actually happening

- runtime enters `main`
- libc eventually performs `write`
- process eventually performs `exit`

Teaching point

Use this demo to shift the class from "a C function runs" to "a process keeps asking the OS to make effects happen".

The Smallest Useful Program Still Needs the OS

With libc

```c
int main() {
    printf("hello\n");
    return 0;
}
```

Without libc (RISC-V Linux)

```
.global _start
.equ MSG_LEN, 18
_start:
    li a0, 1          # stdout
    la a1, msg        # buffer
    li a2, MSG_LEN    # length
    li a7, 64         # write
    ecall

    li a0, 0          # status
    li a7, 93         # exit
    ecall
msg: .ascii "Hello, RISC-V OS!\n"
```

- `printf` eventually asks the OS to `write`.
- `return 0` eventually asks the OS to `exit`.
- On RISC-V, `ecall` is the special instruction that crosses the OS boundary.

Live Demo: Compare Two `strace` Outputs

Assembly hello

```
execve("./hello_aarch64_linux",
  ["./hello_aarch64_linux"],
  ...) = 0
write(1, "Hello, world!\n", 14)
  = 14
exit(0) = ?
+++ exited with 0 +++
```

musl C hello

```
execve("./hello_musl_dbg",
  ["./hello_musl_dbg"], ...) = 0
set_tid_address(0x...) = 141
ioctl(1, TIOCGWINSZ, ...) = 0
writev(1, [{"Hello, world!", 13},
  {"\n", 1}], 2) = 14
exit_group(0) = ?
+++ exited with 0 +++
```

Key observation

The two programs print the same output, but the libc version crosses extra layers: startup code and stdio.

What This Demo Proves

1. The same user-visible result can correspond to different syscall traces.
2. The assembly program talks to the kernel almost directly: `write`, then `exit`.
3. The libc program goes through extra layers such as startup code and stdio, so `strace` shows `set_tid_address`, `ioctl`, and `writev`.
4. `strace` reveals not just what a program does, but which layer is doing the work.

One sentence to say out loud

Two programs can print the same line, but one speaks almost directly to the kernel while the other brings libc along for the conversation.

Mini Section: Debugging libc with `gdb`

What `strace` can show

- `execve`
- `writev`
- `exit_group`

What `strace` cannot show

- `printf`
- `vfprintf`
- `__stdio_write`

Key transition

If `strace` shows the kernel boundary, then `gdb` shows the libc path before the boundary.

Live Demo: Follow `printf` into libc

```
$ gdb ./hello_musl_dbg
(gdb) break main
(gdb) break printf
(gdb) run
(gdb) step
(gdb) bt
```

- Start from `main`.
- Step into `printf`.
- Use `bt` to show that libc code stands between user code and the syscall.

Tool boundary

`strace` answers "which syscall happened?"; `gdb` answers "which library code led us there?"

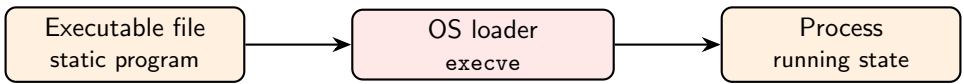
Process: A Running State Machine

Program

- Static description
- Stored as an executable file
- Does not change by itself

Process

- Active runtime instance
- Registers, memory, file descriptors
- Managed by the OS



Demo Bridge: Process = Executable + Live Machine State

```c
static jmp_buf env;

WRITE_SETJMP_MARKER;
ret = setjmp(env);

if (ret == 0) {
    WRITE_LONGJMP_MARKER;
    longjmp(env, 1);
}
```

- `setjmp` saves part of the current execution context.
- `longjmp` restores it and resumes from there.

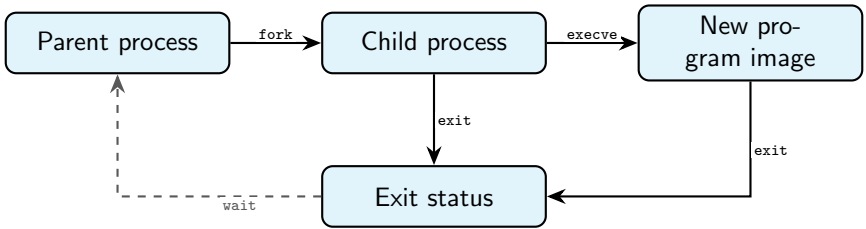
Why this matters

- A process is not just code on disk.
- It includes registers, stack, and control position.
- Context can be saved and restored.

Teaching point

This is a user-space preview of the same idea the OS relies on during context switching.

Process Lifecycle: The UNIX Answer



Mental model

`fork` copies a running state machine; `execve` resets it to a new initial state; `exit` destroys it.

Discussion: What Does `fork` Copy?

```c
int x = 1;
pid_t pid = fork();
x++;
printf("pid=%d, &x=%p, x=%d\n", getpid(), &x, x);
```

What students often say

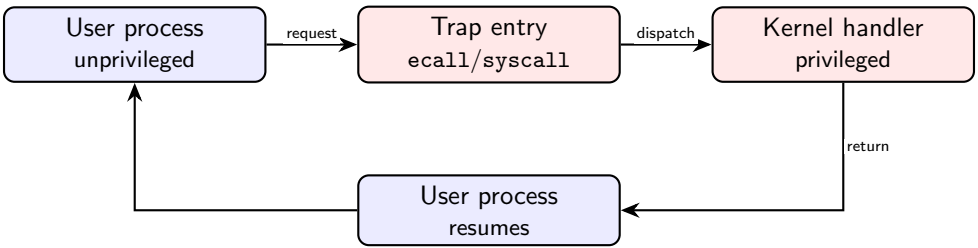
- Parent and child share the same address.
- So they must share the same variable.

What the OS actually gives

- Same virtual address.
- Different process address spaces.
- Physical pages may be shared until copy-on-write.

System Calls: Controlled Boundary Crossing

- A syscall is not just a function call.
- It is a controlled transition from user mode into kernel mode.
- The kernel validates the request, updates OS objects, and returns a result.



User Mode vs. Kernel Mode

User mode

- Applications execute here.
- Cannot touch arbitrary memory or privileged registers.
- Bad accesses become faults or trapped requests.

Kernel mode

- The OS executes here.
- Can manage hardware, page tables, and scheduling state.
- Enforces protection on behalf of every process.

Why the split matters

Without this boundary, a buggy user program could overwrite another process or the whole kernel.

The Trap Path on RISC-V

1. User code places syscall number and arguments in registers.
2. `ecall` traps from user mode into supervisor mode.
3. The CPU records trap state such as `sepc` and cause registers.
4. Control jumps to the kernel trap handler.
5. The kernel validates the request, performs work, and prepares a return value.
6. `sret` returns control to user mode.

Connection to earlier demos

The assembly hello showed a single explicit `ecall`; `strace` later shows an ordinary process crossing this trap path again and again.

OS Objects: The Application's World

Applications do not manipulate kernel data structures directly.

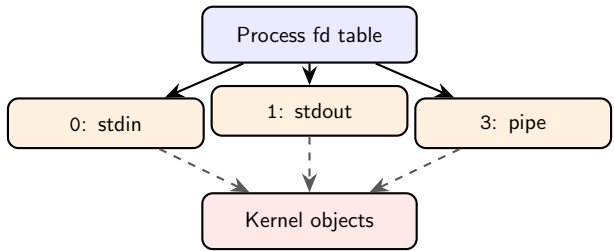
Object	What the application sees	Common APIs
Process	`pid`, exit status	`fork`, `wait`, `kill`
File-like object	file descriptor	`open`, `read`, `write`, `close`
Address range	pointer / mapped region	`mmap`, `munmap`, `mprotect`
Pipe	two file descriptors	`pipe`, `read`, `write`

Core abstraction

Handles such as `pid` and `fd` are pointers into the OS-managed world, not raw memory pointers.

File Descriptors: Pointers into the Kernel World

- A process has a file descriptor table.
- Each `fd` names an OS object.
- The same API works for files, terminals, sockets, pipes, and many pseudo-files.



Live Demo: Record a Trace File

```
$ strace ./hello 2> hello.strace
$ less hello.strace
```

- The program output still goes to `stdout`.
- The trace is redirected into a file.
- That file becomes a reusable OS case study.

Why keep this file

This trace is a timeline of what the OS had to do in order to support one tiny program.

Read `hello.strace` from the OS Side

Representative lines

```
execve("./hello", ...) = 0
openat(..., "/etc/ld.so.cache", ...) = 3
openat(..., "/lib/.../libc.so.6", ...) = 3
read(3, "\177ELF...", 832) = 832
mmap(..., PROT_READ|PROT_EXEC, ...) = ...
mprotect(..., PROT_READ) = 0
fstat(1, {...}) = 0
write(1, "Hello, world!\n", 14) = 14
exit_group(0) = ?
```

OS-side reading

- `execve`: create a new process image
- `openat` + `read`: find and load libraries
- `mmap` + `mprotect`: build address space
- `fstat(1)`: check what stdout refers to
- `write`: finally perform visible I/O
- `exit_group`: terminate the process cleanly

Why this trace matters later

This tiny trace already contains process startup, library loading, address-space construction, file-descriptor inspection, and final output.

ELF: Blueprint for Initial Process State

- The executable is not just code bytes; it is a structured ELF description.
- ELF tells the loader the entry point, required interpreter, and loadable segments.
- Those segments explain why ‘strace‘ later shows ‘mmap‘, anonymous zero-fill, and ‘mprotect‘.

What ELF specifies

- entry point
- ‘INTERP‘
- ‘LOAD‘ segments
- permission expectations

What the OS/loader does

- ‘execve‘ starts construction
- libraries are located and opened
- regions are mapped and protected
- control eventually reaches the entry point

Bridge to the next section

‘readelf‘ describes the target shape of a process; ‘strace‘ shows the OS constructing that shape into an address space.

Live Demo: Make the Boundary Visible with `strace`

```
$ strace -f -e trace=process,file,read,write ./a.out
```

- Every line is an application crossing the OS boundary.
- `openat`, `read`, `write`, `mmap`, `execve` reveal hidden assumptions.
- This is the fastest way to turn "OS concepts" into observable behavior.

Connection to the previous demos

The `ecall` demo shows one boundary crossing from the instruction level; `strace` shows that an ordinary process spends its whole life crossing that boundary.

Question

If every program is "just a state machine", why does `strace` show so many OS interactions?

Address Space: The Process's Private World

Definition for this discussion

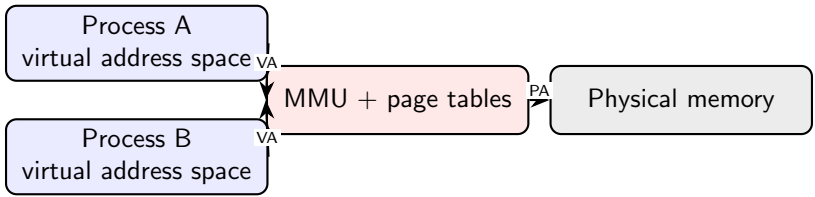
An address space is the set of virtual addresses a process can use, plus the meaning assigned to those addresses.

- After `execve`, the OS uses ELF-driven mappings to build this world.
- Pointers are meaningful only inside an address space.
- Two processes can print the same pointer value and still refer to different physical memory.
- The OS can add, remove, and protect regions using APIs such as `mmap` and `mprotect`.

From ELF to address space

The `mmap` and `mprotect` calls we saw in `strace` were the OS constructing and tightening this address space from the ELF blueprint.

A Process Does Not See Physical Memory



Mental model

The MMU is the VR headset the OS puts on every process.

- Virtual addresses solve placement, protection, and sharing problems.
- Without this layer, programs would fight over RAM and one bug could corrupt everything.

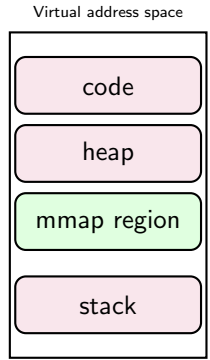
What `mmap` Really Changes

Before `mmap`:

- The address may be unmapped.
- Any access triggers a fault.

After `mmap`:

- A new mapped region appears.
- Page table entries may be lazy.
- Permissions are part of the mapping.
- Physical pages may appear on first access.



ELF loading connection

The loader uses ELF `LOAD` segments to define mapped code, data, and zero-filled regions.

Why Address Spaces Matter

For programmers

- A large, simple pointer world
- A mostly contiguous, stable layout
- Files can be mapped into memory
- Private execution that feels isolated

For the OS

- Enforce protection
- Share pages deliberately
- Implement lazy allocation
- Support copy-on-write

Discussion

Is virtual memory mainly a performance feature, a safety feature, or a programming convenience?

Virtual Memory: The Contract

Core idea

The OS gives each process a virtual address space that looks large, mostly contiguous, private, and stable enough to program against.

What the program feels

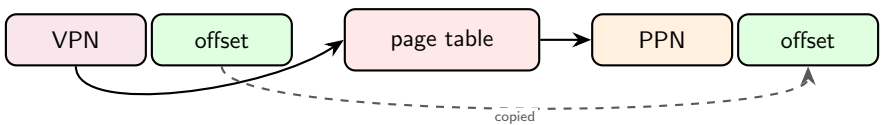
- simple pointers
- private memory
- familiar regions: code, heap, stack, mappings

What the OS and MMU do

- translate virtual to physical addresses
- enforce permissions
- share or delay allocation when useful

Paging: Split the Illusion into Fixed-Size Pieces

- Virtual address space is divided into pages.
- Physical memory is divided into page frames.
- A page table maps virtual pages to physical frames.
- The illusion stays contiguous even when the physical frames are scattered.



Why this mechanism exists

Paging avoids large contiguous-allocation problems, supports sparse address spaces, and enables demand paging.

Virtual to Physical Address Translation

Address split

```
VA = [Virtual Page Number] [Page Offset]
```

- The MMU uses the virtual page number to consult page tables.
- The page table yields a physical page number plus permission bits.
- The page offset is copied unchanged.

```
PA = [Physical Page Number] [Page Offset]
```

Page Tables Are OS Data for Hardware

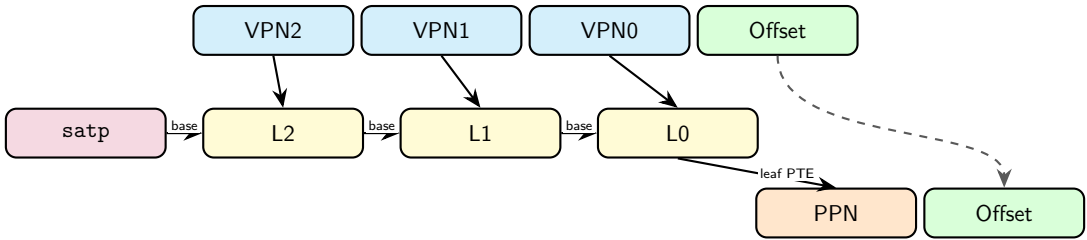
- The OS constructs page tables.
- The MMU consumes page tables on every memory access.
- A special register points to the root page table.
- On RISC-V, this register is `satp`.

Important distinction

Page tables are not an application data structure. They are the contract between the kernel and the MMU.

RISC-V SV39 Page Walk

```
VA = [VPN2] [VPN1] [VPN0] [Offset]
```



PTE Bits: Translation plus Policy

A page table entry carries both an address and rules.

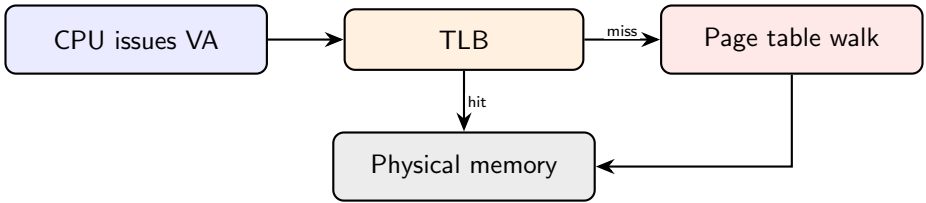
Bit / field	Meaning
PPN	Physical page number
V	Whether the entry is valid
R / W / X	Read, write, execute permission
U	Whether user mode can access this page
A / D	Accessed and dirty tracking
RSW	Reserved for supervisor software

Every memory access asks a question

Does this virtual address translate, and is this access allowed?

TLB: Caching the Illusion

- Walking page tables on every memory access would be too expensive.
- The TLB caches recent VPN-to-PPN translations.
- Context switches may require changing `satp` and invalidating stale translations.



Protection, Sharing, and Copy-on-Write

Protection

- R/W/X bits
- user vs supervisor
- faults on violation

Sharing

- shared libraries
- shared memory
- mapped files

Copy-on-write

- initially share pages
- mark read-only
- copy on write fault

Unifying idea

These are not separate tricks. They are different policies encoded in page tables.

Computing Paradigms

Bare metal

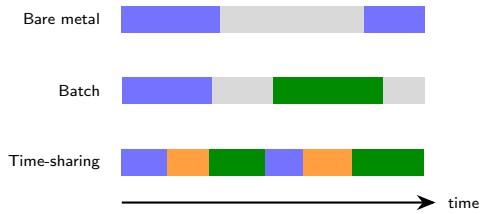
- One program owns the machine.
- I/O often leaves the CPU idle.

Batch

- The OS queues jobs and runs them one after another.
- Better utilization, weak interactivity.

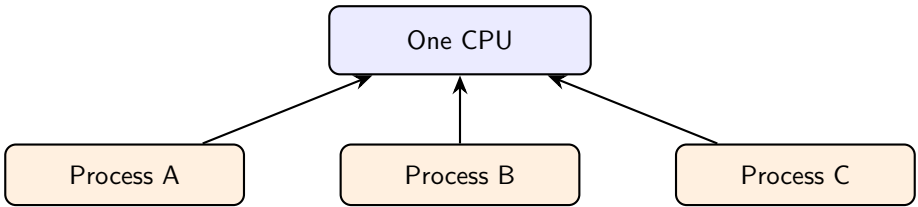
Time-sharing

- The OS multiplexes one CPU across many active processes.
- This is where process, syscall, and virtual-memory abstractions become central.



How One CPU Feels Like Many

- On one core, processes do not really run at the same instant.
- The OS keeps switching the CPU between runnable processes.
- Blocking syscalls, timer interrupts, and process exit are common switch points.



Illusion

Time-sharing makes multiple state machines feel simultaneously alive even when the hardware has only one execution stream.

Context Switching: What the OS Must Move

1. Save the current process's CPU state.
2. Choose another runnable process.
3. Restore its registers and execution point.
4. Resume as if that process had been running all along.

Saved state

- program counter
- general registers
- kernel bookkeeping

Cost

- save/restore overhead
- cache disruption
- TLB and address-space switching

Mechanism vs. Policy

Mechanism

- Trap into the kernel
- Save state
- Restore another state
- Return with `sret`

Policy

- Who runs next?
- For how long?
- Who should wait?
- How do fairness and latency trade off?

Examples of policy

Round robin emphasizes fairness; priorities and feedback queues try to balance responsiveness and throughput.

Context Switches Also Switch Address Spaces

1. Save the current process's CPU state.
2. Load the next process's saved registers.
3. Update `satp` to point at the next process's page-table root.
4. Ensure stale translations do not survive incorrectly, often with TLB invalidation such as `SFENCE.VMA`.

Why this matters

Switching processes is not just "who gets the CPU". It also changes which virtual world every future memory access lives in.

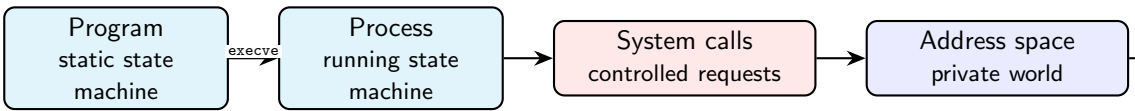
Three Questions to Check Understanding

1. Two processes print the same pointer value. Are they accessing the same memory?
2. After `fork`, why can parent and child appear to share memory without immediately copying all pages?
3. Why does changing `satp` during a context switch affect every future memory access?

Rule of thumb

When confused, ask: which state machine, which OS object, which address space?

One Story



Takeaway

The OS course is not a list of mechanisms. It is a way to understand how a simple hardware state machine can host many isolated, useful, interactive worlds.

Suggested Further Reading

- JYY OS 2026: course website; application view; programs/processes; address spaces; OS objects.
- OSTEP: book website.
- xv6: RISC-V book; source code.
- uCore: tutorial guide; source and labs.