

Introduction to Operating Systems

Troels Henriksen

What we will learn today

- Why do we have operating systems?
- How do interrupts allow the operating system kernel to take control?
- What is the purpose of system calls?
- What are processes, and how do we switch between them?
- What is the purpose of virtual memory?

The amount of technical complexity in this area is staggering, so we will focus on concepts and just a few technical details.

The purpose of operating systems

Kernel and Interrupts

Processes

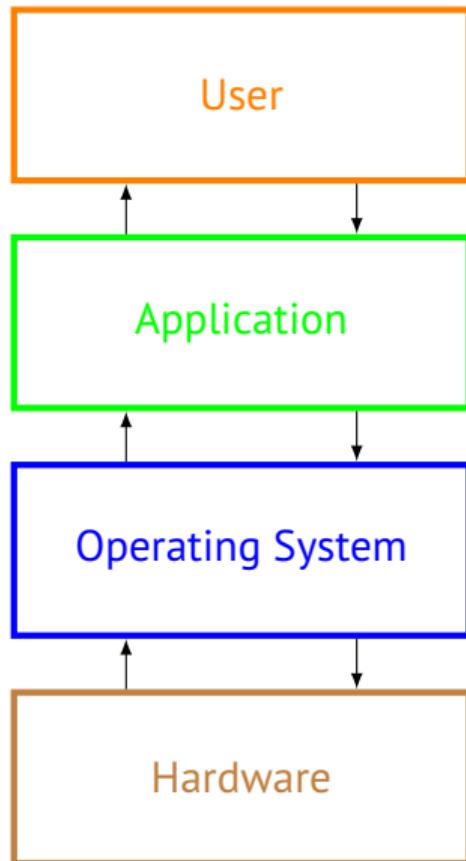
System calls

Virtual memory

Process management in Unix

Main takeaways

What is an operating system?



OS is middleware between applications and hardware.

- Provides standardized interface to resources.
- Manages hardware.
- Orchestrates currently executing processes.
- Responds to resource access requests.
- Handles access control.

Why study operating systems?

Inspirational

- One of the most potent *abstractions* in computing.
 - ▶ Each process thinks it has machine to itself.
 - ▶ Controlled communication.
 - ▶ Abstracts over hardware differences.

Practical

- You almost always use an operating system.
- Its performance characteristics are important to understand.
- It often determines what is fundamentally possible.

They are where the magic happens.

In the old days

Each brand of machine would have its own operating system.



IBM System/360 running OS/360 (man not part of computer)



DEC PDP-10 running TENEX



VAX 11/780 running VMS



- PDP-11 running early UNIX, written in C.
- Developed at AT&T, who were banned from selling UNIX.
- Shared it (almost) freely with others, who *ported* UNIX to every machine under the sun.
- Unix was *popular, good enough, and cheap.*

What is Unix

- | | |
|--------------|--|
| What? | <ul style="list-style-type: none">▪ Unix is a family of operating systems derived from original UNIX developed in the 1970s by Ken Thompson and Dennis Ritchie.▪ Most modern operating systems are heavily influenced by Unix (even Windows).▪ Many operating systems are <i>direct descendants</i>: Linux, iOS, macOS, the BSDs, etc. |
| Why? | <ul style="list-style-type: none">▪ We teach Unix because it is <i>simple</i> and <i>representative</i> of modern systems.▪ We use Unix designs for all examples.▪ ...does not mean Unix is always <i>good</i> design. |

The purpose of operating systems

Kernel and Interrupts

Processes

System calls

Virtual memory

Process management in Unix

Main takeaways

The kernel

- Technically, *operating systems* encompass lots of parts: shell, GUI, C library, maybe bundled applications (mail reader etc).
- In this course, when we say *operating system* we really mean the **kernel**.

The kernel

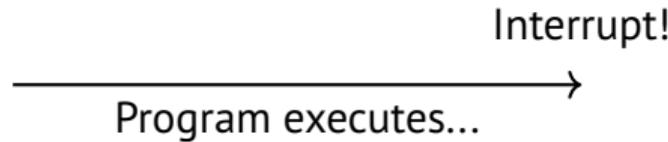
- Technically, *operating systems* encompass lots of parts: shell, GUI, C library, maybe bundled applications (mail reader etc).
- In this course, when we say *operating system* we really mean the **kernel**.

Kernel

Always-resident code that services requests from the hardware and manages processes.

The kernel takes control!

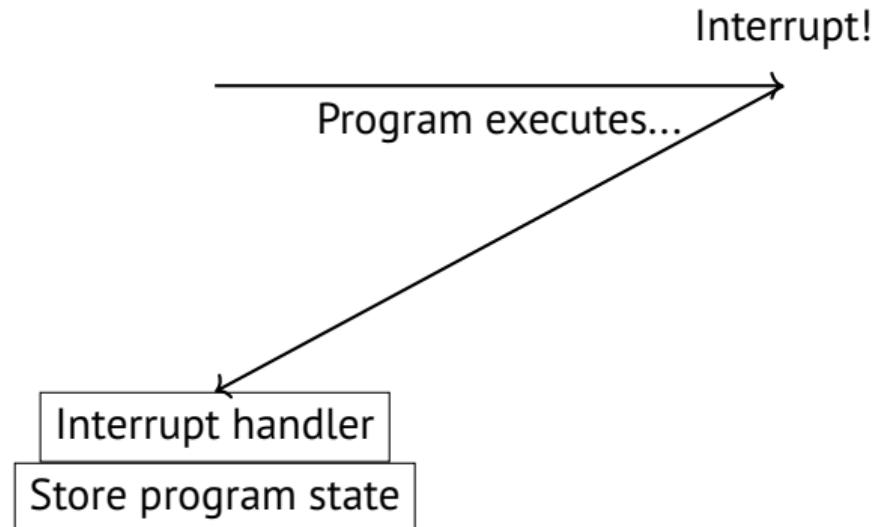
- The CPU executes program code **until an interrupt happens.**



Boxed parts are kernel code.

The kernel takes control!

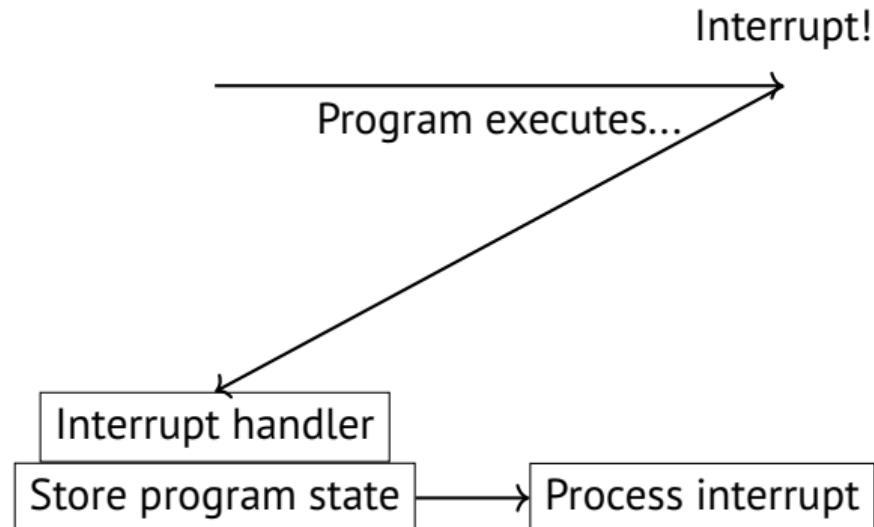
- The CPU executes program code **until an interrupt happens.**



Boxed parts are kernel code.

The kernel takes control!

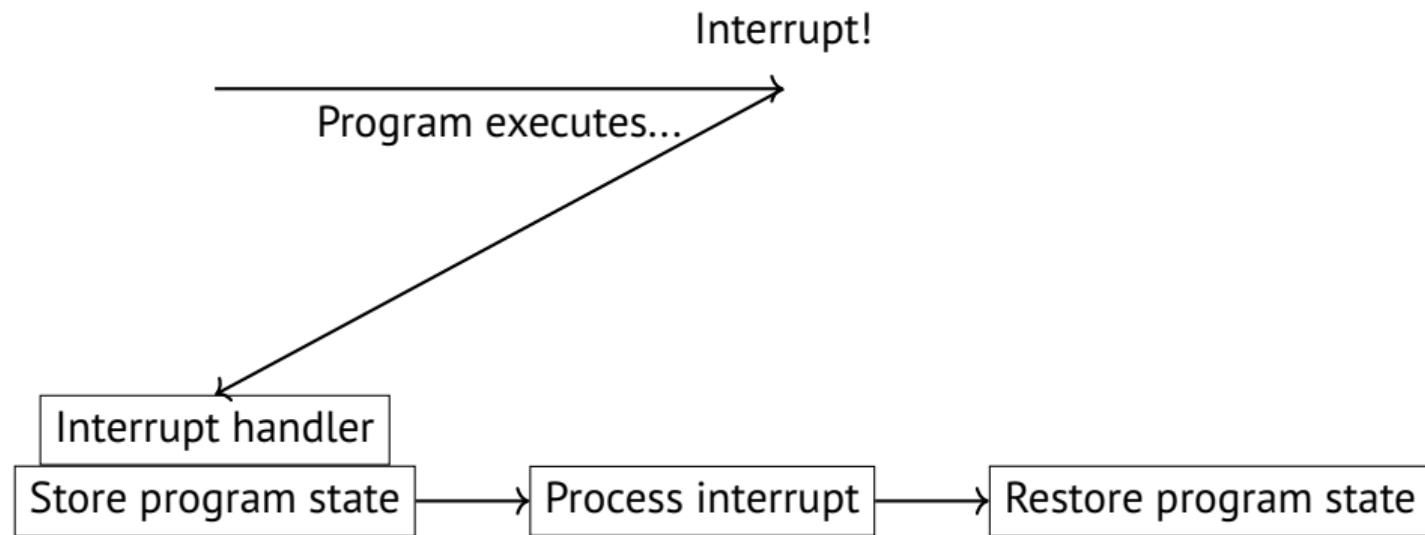
- The CPU executes program code **until an interrupt happens.**



Boxed parts are kernel code.

The kernel takes control!

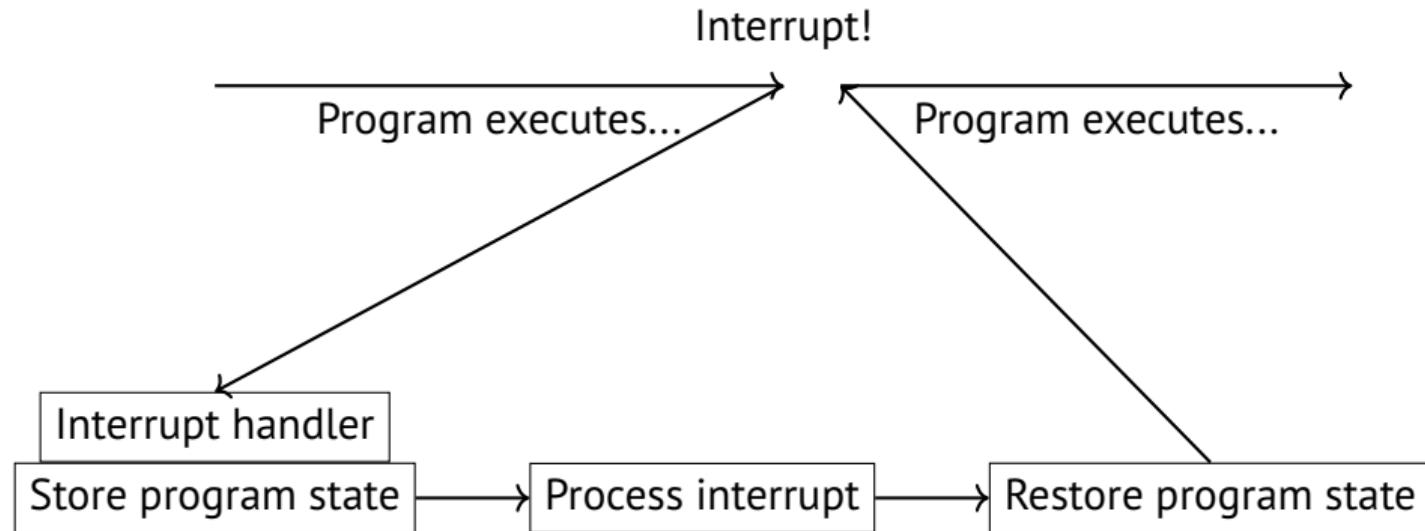
- The CPU executes program code **until an interrupt happens.**



Boxed parts are kernel code.

The kernel takes control!

- The CPU executes program code **until an interrupt happens.**



Boxed parts are kernel code.

The purpose of operating systems

Kernel and Interrupts

Processes

System calls

Virtual memory

Process management in Unix

Main takeaways

Processes contra programs

Program: is a file containing code. Stateless and **dead**.

Process: a running *instance* of a program. The same program can be running in multiple instances. Stateful and **alive**.

They are not the same thing, although we informally often say *program* when we really mean *process*.

Processes contra programs

Program: is a file containing code. Stateless and **dead**.

Process: a running *instance* of a program. The same program can be running in multiple instances. Stateful and **alive**.

They are not the same thing, although we informally often say *program* when we really mean *process*.

- Operating systems manage *processes*.
 - ▶ Switching between multiple *concurrent* processes.
 - ▶ Handling process termination.
 - ▶ Starting new processes (perhaps from a given *program file*).

The kernel

- The kernel uses the same CPU, memory, and other hardware as ordinary code.
- When running process code, the CPU is in *unprivileged state*, and many operations are restricted (e.g. access to hardware devices).
 - ▶ When an *interrupt* happens, the CPU switches to *privileged state* and jumps to kernel code, which handles it and then resumes the previously running process.
 - ▶ Think of it like a sudden and unplanned procedure call.
 - ▶ Interrupts can be outside events (keyboard press, network traffic) or special instructions (invalid memory accesses, *system calls*).

Virtualising the CPU

Goal Give each process the illusion of exclusive CPU access.

Reality: the CPU is a shared resource.

Virtualising the CPU

Goal Give each process the illusion of exclusive CPU access.

Reality: the CPU is a shared resource.

Solution: **context switching**

- Only one process gets to run at a time.
- ...but we regularly switch between available processes.
- Doing this often and rapidly creates the illusion of simultaneous execution.

Context switching

Intuition Pausing a process, saving its entire *state*, then resuming some other process based on its saved state.

Context switching

Intuition Pausing a process, saving its entire *state*, then resuming some other process based on its saved state.

So what do we need to save?

1. All registers, including control registers.
2. Contents of memory.

Context switching

Intuition Pausing a process, saving its entire *state*, then resuming some other process based on its saved state.

So what do we need to save?

1. All registers, including control registers.
2. Contents of memory.

So when do we do this?

- Regular *timer interrupts* transfer control to the kernel, whose *scheduler* decides the next process to run.
 - ▶ Scheduling is a big and interesting topic that we don't have time to go into.

The purpose of operating systems

Kernel and Interrupts

Processes

System calls

Virtual memory

Process management in Unix

Main takeaways

System calls

- Only the kernel has direct access to hardware and system memory.
- Whenever we want to do IO we have to perform a *system call*.

System calls

A request by a process that the kernel carries out some operation on its behalf.

- Much like a function call, but implemented very differently.

System calls in RISC-V

- The `ecall` (*environment call*) instruction transfers control to the kernel.
 - ▶ Kernel then inspects registers (mostly `a0-a7`) to see what it has been asked to do.
 - ▶ Specific interpretation varies between operating systems.
 - ▶ System call identified by a number.

System calls in RISC-V

- The `ecall` (*environment call*) instruction transfers control to the kernel.
 - ▶ Kernel then inspects registers (mostly `a0-a7`) to see what it has been asked to do.
 - ▶ Specific interpretation varies between operating systems.
 - ▶ System call identified by a number.

In RARS

<https://github.com/TheThirdOne/rars/wiki/Environment-Calls>

- System call number passed in `a7`.
- Excerpt:

Human name	Number	Description	Reads	Writes
PrintInt	1	Prints int to console.	a0	
ReadInt	5	Reads int from console.		a0

System calls in C

- C exposes system calls as functions.
 - ▶ Internally use `ecall` instruction or architecture-specific equivalent.
- Not observably distinct from other C functions, except typically more primitive and tedious to use.

System calls in C

- C exposes system calls as functions.
 - ▶ Internally use `ecall` instruction or architecture-specific equivalent.
- Not observably distinct from other C functions, except typically more primitive and tedious to use.

Example

```
// system calls
int open(const char *pathname, int flags);
ssize_t write(int fd, const void *buf, size_t count);

// stdio functions
FILE *fopen(const char *pathname, const char *mode);
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream);
```

File descriptors

- ```
int open(const char *pathname, int flags);

ssize_t write(int fd, const void *buf, size_t count);
```
- The `int` returned by `open` is a *file descriptor*.
  - Has no significance in itself, but allows the kernel to recognise the open file when passed to other system calls.
    - ▶ Typically an index into some kernel-side table.
    - ▶ Such values are known as *handles*.
  - Passing complex data structures or pointers between *kernel space* and *user space* is annoying and fragile, so we usually use numeric identifiers instead.

The purpose of operating systems

Kernel and Interrupts

Processes

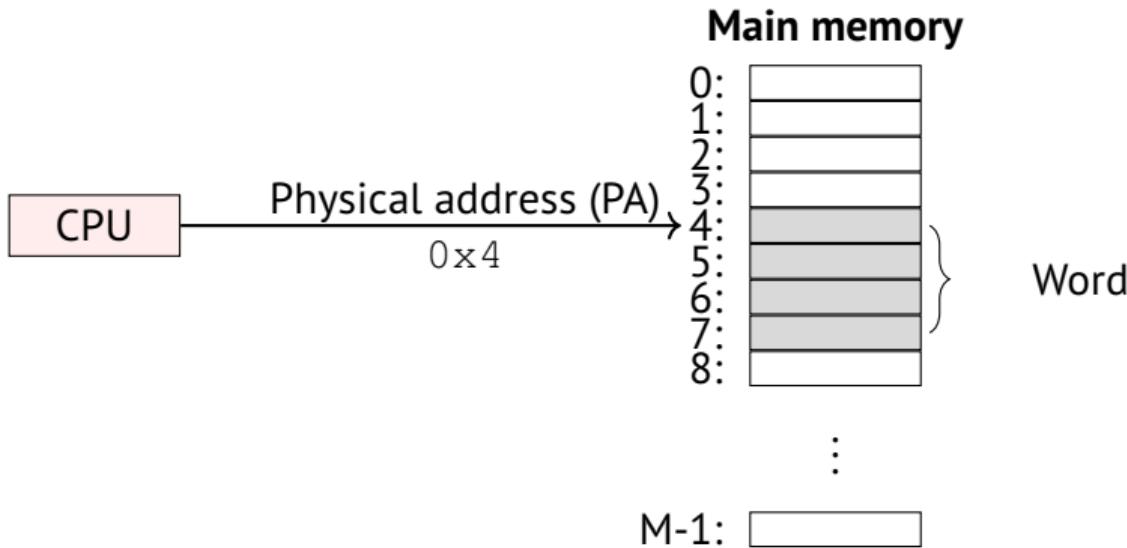
System calls

**Virtual memory**

Process management in Unix

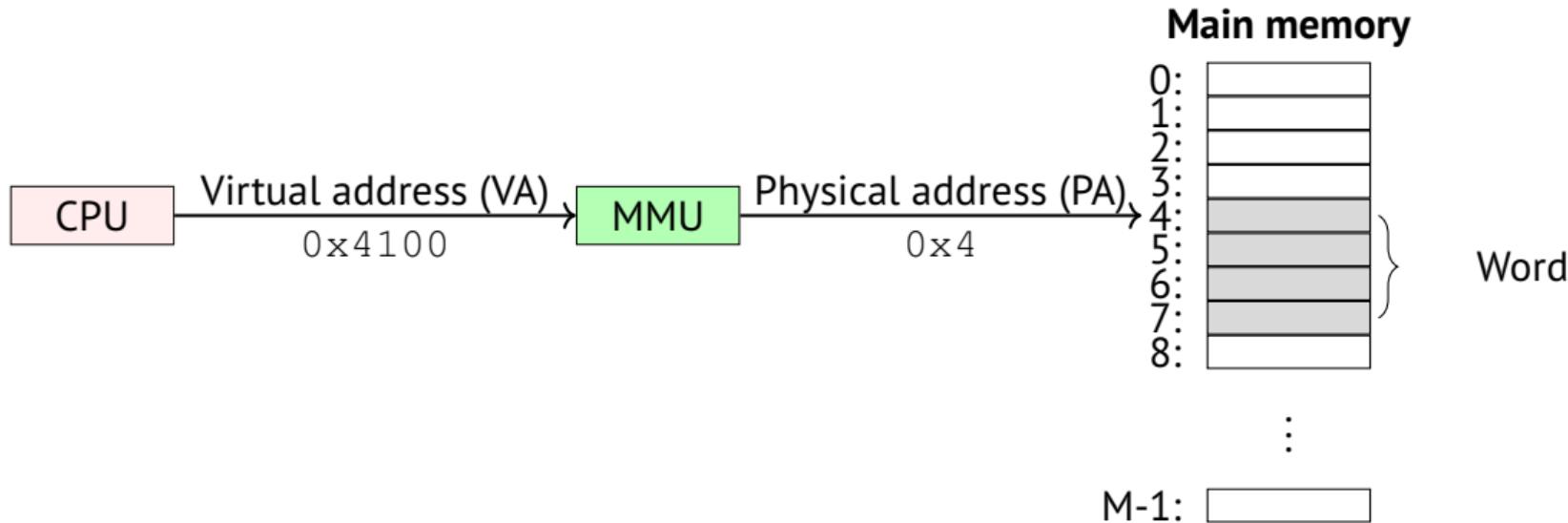
Main takeaways

# Physical addressing – fetching a 4-byte word



- Addresses fetched by program correspond immediately to physical memory locations.
- Caches not part of the diagram, but still present.
- **Only very small computers work this way.**

# Virtual addressing – fetching a 4-byte word



- **MMU** (*memory management unit*) interposed between CPU and memory, rewrites *virtual addresses* to *physical addresses*.
- MMU essentially a mechanism to inject behaviour for *every memory access*.
- Caching happens *after* address translation (on physical addresses).

# Address spaces

## Address space

Ordered set of contiguous non-negative integer addresses, some of which may be invalid.

We consider memory, but a file could also be seen as an address space.

## Physical address space

Set of  $N = 2^n$  addresses.

## Virtual address space

Set of  $M = 2^m$  addresses.

## Address translation

Converting between addresses in different address spaces, for our purposes this is **virtual to physical**.

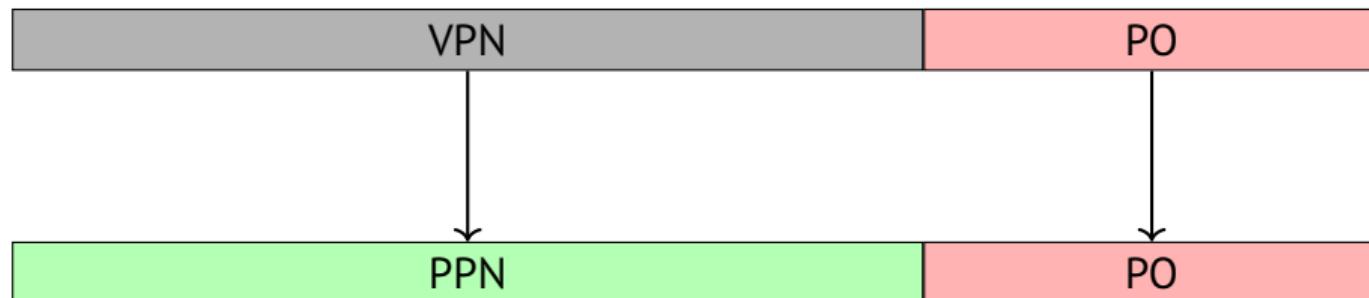
# Address translation in virtual memory

We divide memory into *pages* (similar to how we divide it into *blocks* for caching).

## Page

A block of memory of some fixed size, often 4KiB on current computers.

- For 4KiB pages, the least significant 12 bits (because  $2^{12} = 4096$ ) of an address constitute *page offset* (PO), and the remainder are the *virtual page number* (VPN)
- Address translation replaces the VPN with a *physical page number* (PPN), leaving the PO unchanged in the resulting physical address.



# Why virtual memory?

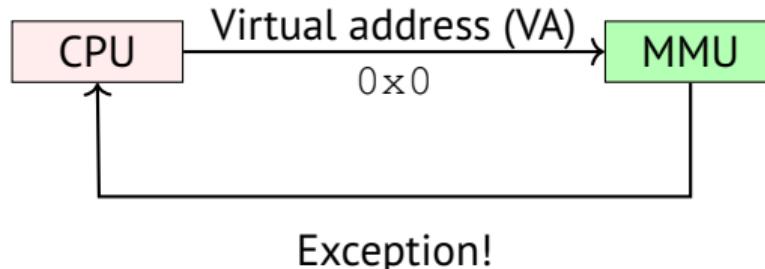
**Economy:** We can treat RAM as a cache for a virtual address space larger than physical memory.

**Simplicity:** each process receives a distinct, uniform, and linear virtual memory space.

**Security:** we can enforce rules on how memory is accessed.

**For time reasons we will look only superficially at these.**

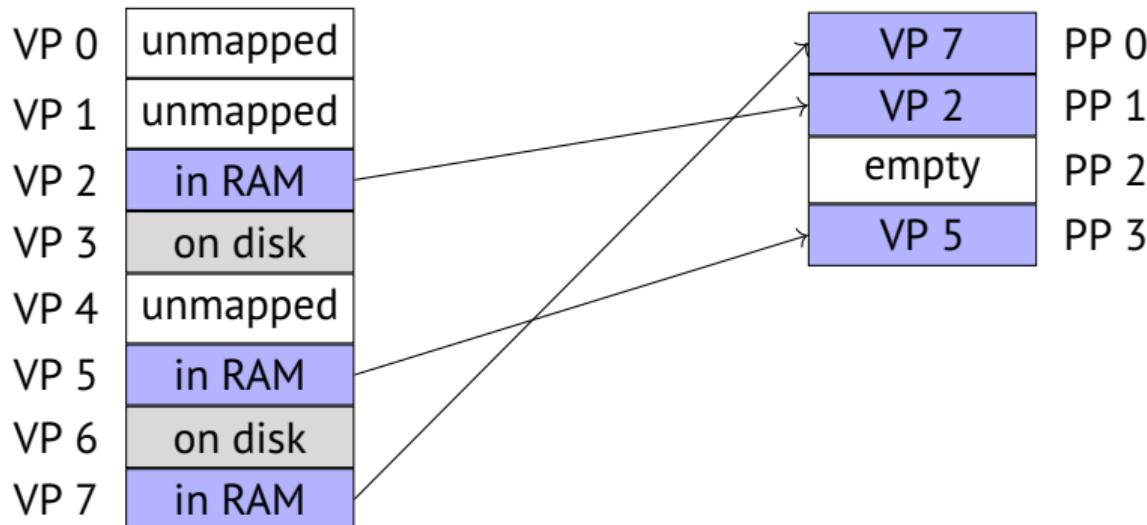
# A memory exception



- An **invalid address** causes an **exception** that triggers control to the kernel.
  - ▶ Similar to a system call.
- The kernel can then (in software) handle the memory exception in whatever way it wants.
- **Important point:** an invalid address does not mean the program is *wrong*, it just means the address needs *special treatment*.

# VM as a tool for caching via paging

- Memory is split into pages.
- Each page is either in physical memory or on disk.
- Pages moved into physical memory when accessed, and put on disk when necessary.



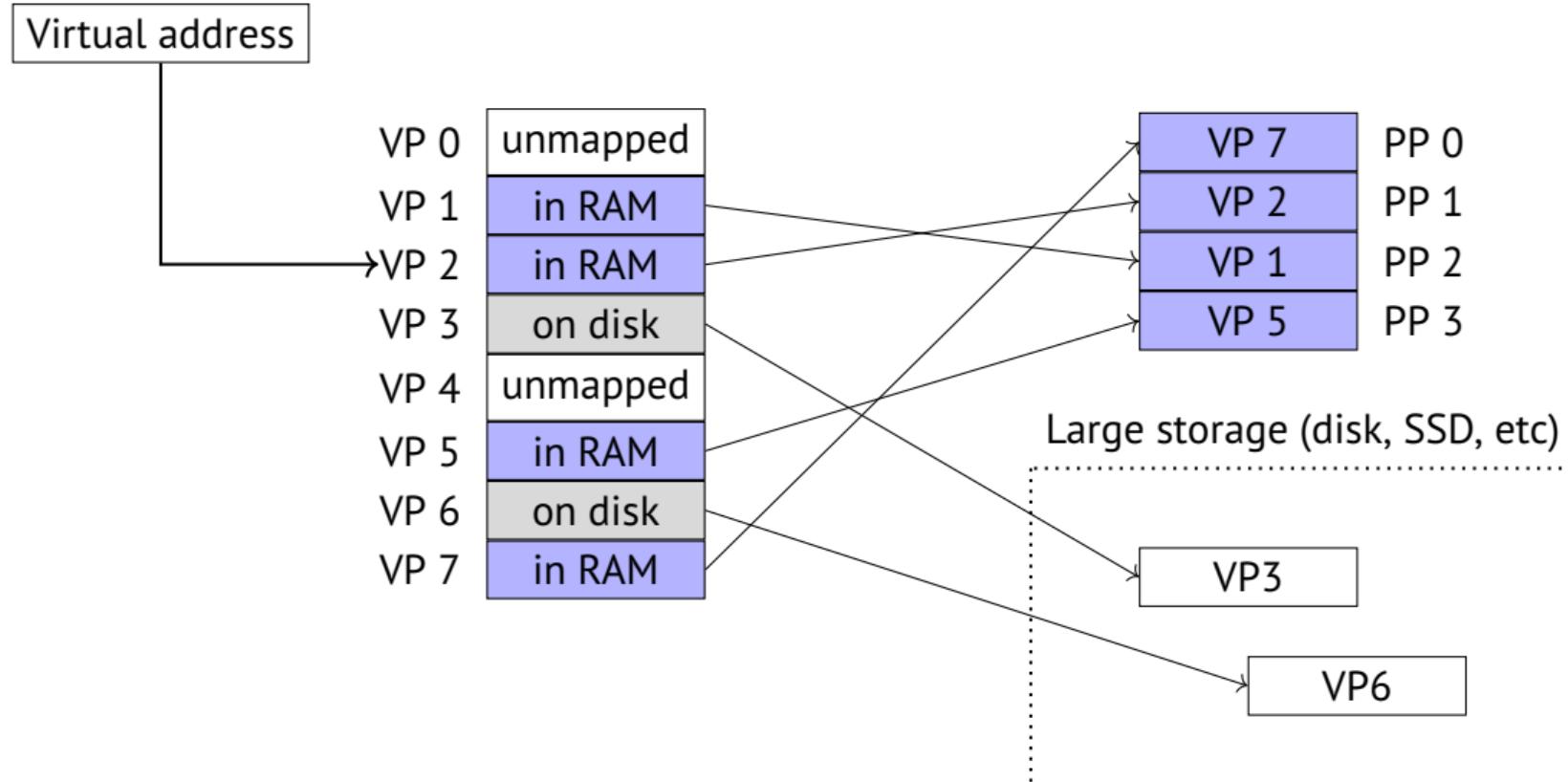
The **page table** is a data structure that records this mapping and which is accessed by the MMU.

# Structure of a page table

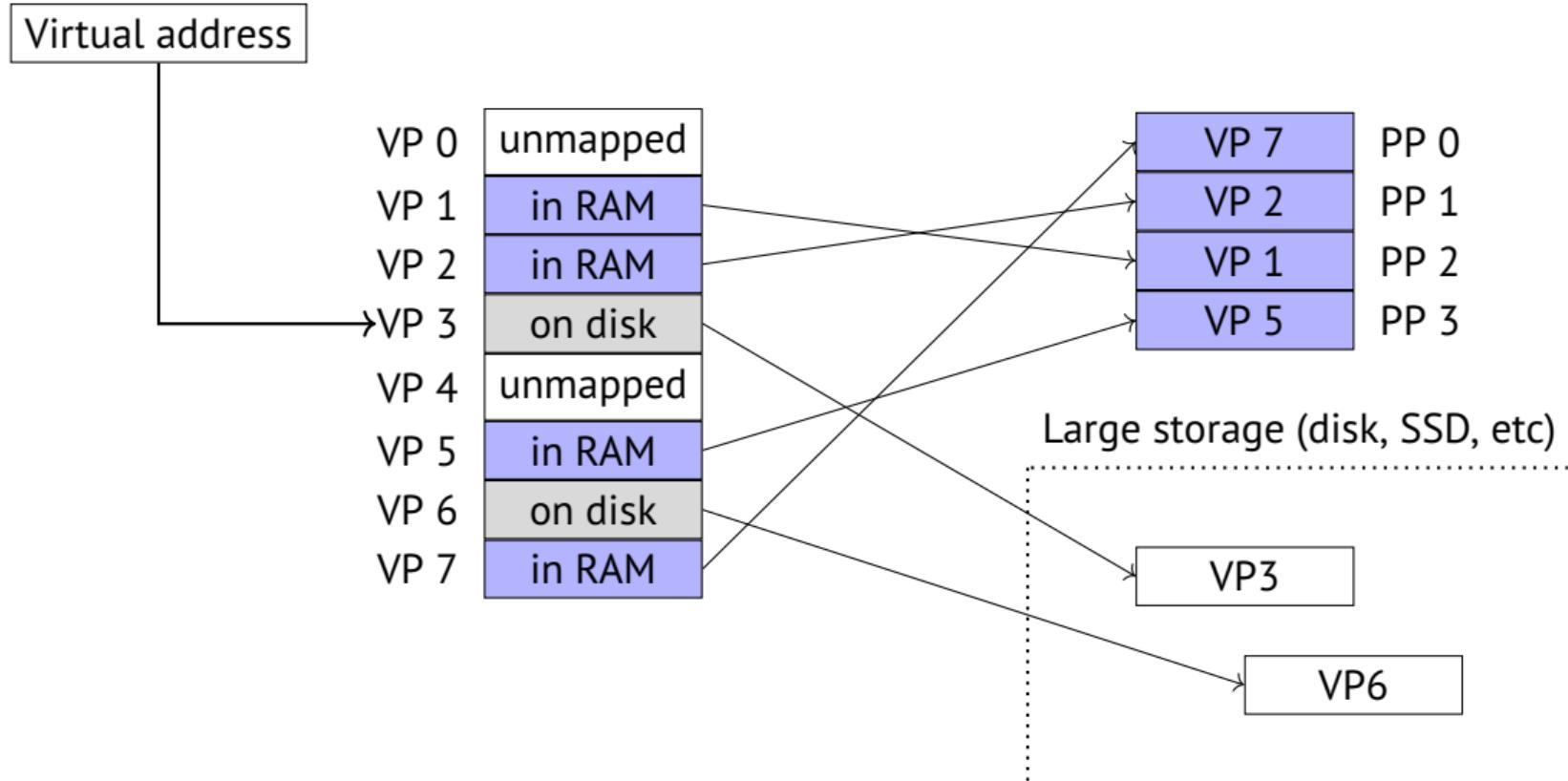
| VPN | PPN | Valid | Flags |
|-----|-----|-------|-------|
| 0   |     |       |       |
| 1   |     |       |       |
| 2   |     |       |       |
| 3   |     |       |       |
| 4   |     |       |       |
| :   |     |       |       |

- Showing a flat page table for simplicity.
- Actually a multi-level page table.
- Page table is accessed by MMU (hardware) so its structure is **fixed** and kept **simple**.
- Either a page is *there* or *it's not*.
  - ▶ No “page is on disk” information.
  - ▶ No “demand-paging” information.
  - ▶ If we access a non-valid page: **page fault!**
- Page faults are handled by **software** (kernel code), meaning we have **flexibility**.
  - ▶ Page fault *handler* can update the page table based on kernel data and policy.

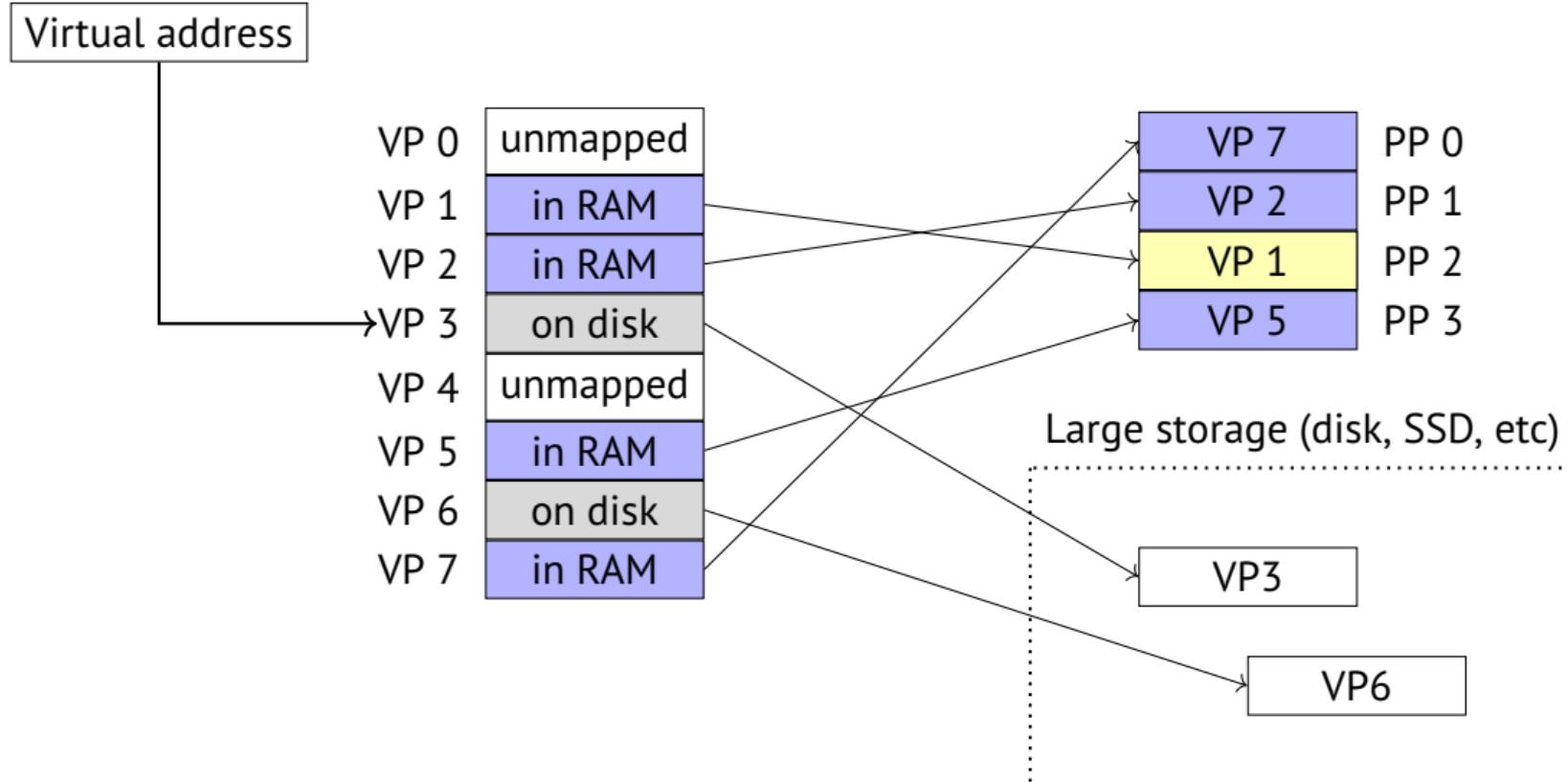
# Page hit



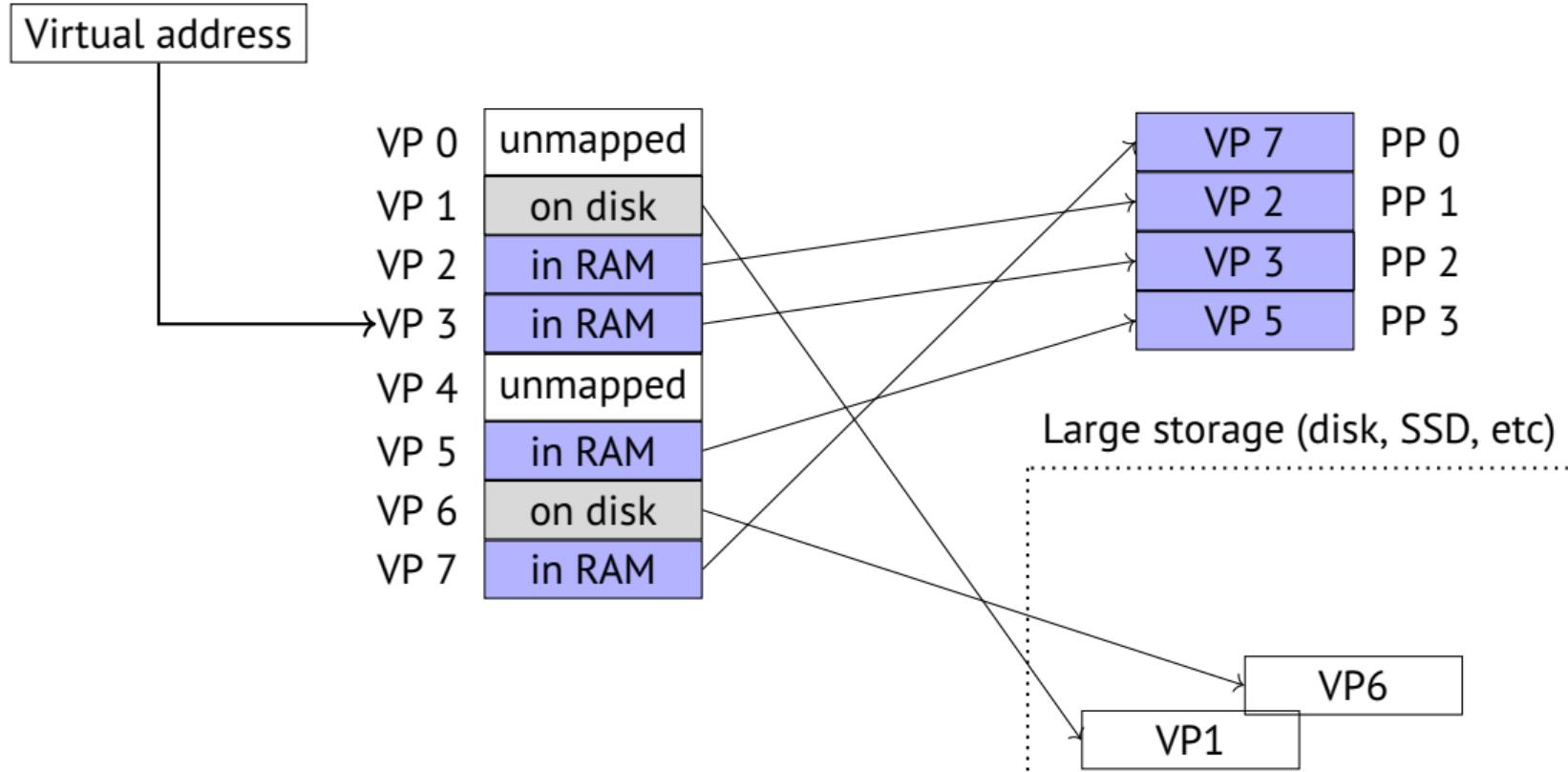
# Page fault



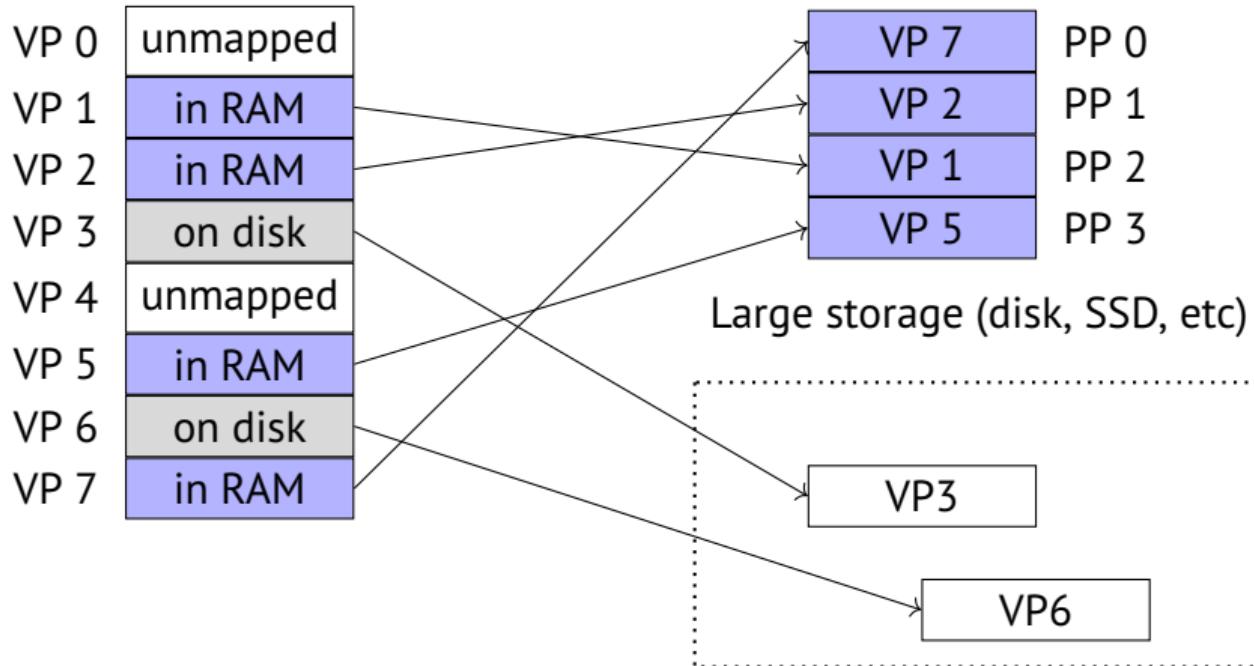
# Page fault



# Page fault

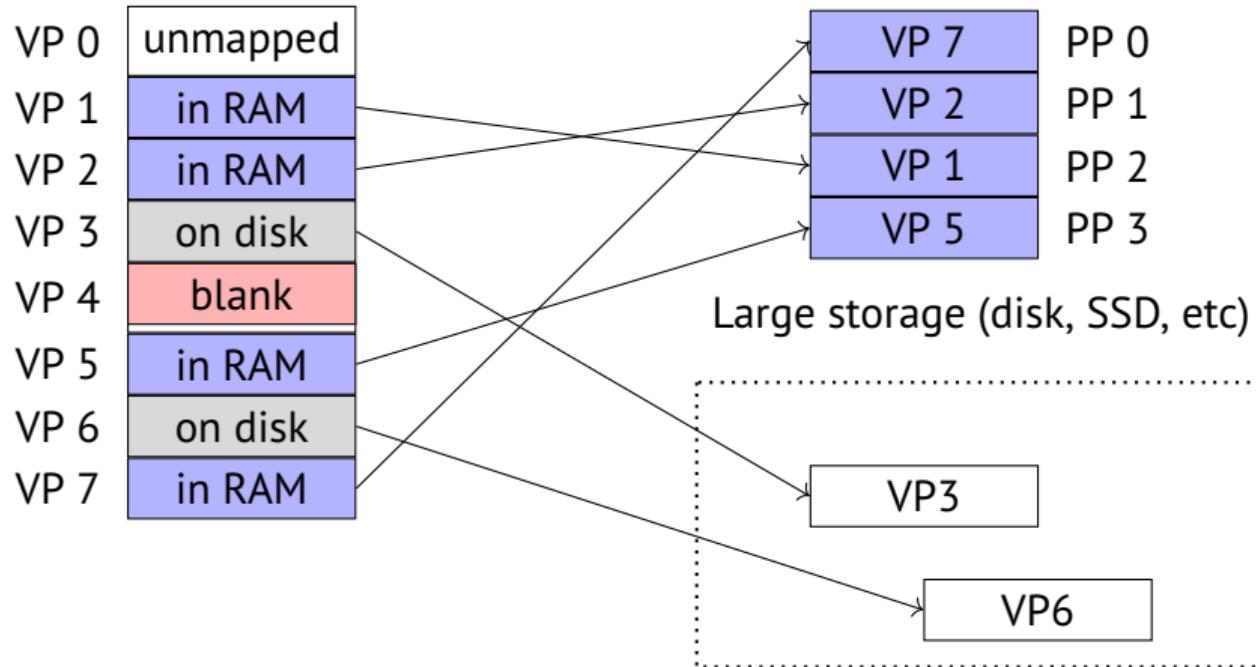


# Allocating memory with demand paging



On first access to blank page, we have a *page fault* and the page is then actually put in memory and initialised with zeroes.

# Allocating memory with demand paging



On first access to blank page, we have a *page fault* and the page is then actually put in memory and initialised with zeroes.

# Virtual memory for memory protection

- Page table entries have **permission bits**.
- When CPU does an address lookup, it also says **what it will be used for**.
- Checked by MMU during page translation.

| Process <i>i</i> | Read | Write | Exec | PPN  |
|------------------|------|-------|------|------|
| VP 0             | Yes  | No    | Yes  | PP 6 |
| VP 1             | Yes  | Yes   | No   | PP 4 |
| VP 2             | Yes  | No    | No   | PP 1 |

| Process <i>j</i> | Read | Write | Exec | PPN  |
|------------------|------|-------|------|------|
| VP 0             | Yes  | Yes   | No   | PP 9 |
| VP 1             | Yes  | No    | Yes  | PP 6 |
| VP 2             | Yes  | Yes   | No   | PP 1 |

Can also be used to **share memory between processes in controlled manner**.

The purpose of operating systems

Kernel and Interrupts

Processes

System calls

Virtual memory

**Process management in Unix**

Main takeaways

# Basic principles

- Each process in Unix has a *process ID* (PID).
- Each process has a *parent*.
  - ▶ ...except the *initial process* (`init`) with PID 1.
- A process may have multiple *children*.
- Implies processes are organised as a *tree* (`pstree` command shows it).
- **Creating processes:** `fork()`.
- **Terminating current process:** `exit()`.
- **Loading program code from disk into current process:** `exec()`.
- **Waiting for a specific child to die:** `waitpid()`.
- **Getting PID of running process:** `getpid()`.

The purpose of operating systems

Kernel and Interrupts

Processes

System calls

Virtual memory

Process management in Unix

**Main takeaways**

# Main takeaways

- Hardware provides *mechanisms* such as interrupts, privileged/unprivileged mode, and *virtual memory* (next week).
  - ▶ Kernels implement *policy* and *abstractions* on top.
- Processes are a *purely virtual concept*—CPU has no idea what they are.
- Processes are *isolated* from each other.
- Processes can only directly interact with the outside world through *system calls*, mediated by the kernel.