Chapter 4c: Virtual Memory and the Page Table

Gene Cooperman

THIS IS STILL A WORK IN PROGRESS.

1 Lecture Notes: Virtual Memory and the Page Table

1.1 Motivation: Why Virtual Memory?

In the previous two chapters, we studied how a CPU cache hides the slowness of RAM from the CPU. There is an analogous problem one level further down the memory hierarchy: **RAM is finite**, but a running process may need more memory than physically fits in RAM. The solution is **virtual memory**: the operating system creates the illusion that each process has a large, private address space, even if only part of that space is currently resident in RAM. The rest is stored on disk in a file called the **swapfile** (also called the swap partition or paging file).

The hardware mechanism that makes this possible is the **page table**. As you will see, the page table works on exactly the same principles as the direct-mapped cache you studied in Chapter 4b. If you understand direct-mapped cache, you already understand most of the page table.

1.2 Review: From Fully Associative Cache to Direct-Mapped Cache

1.2.1 The fully associative cache as a key-value store

Recall from Chapter 4a that a fully associative cache is a **key-value store**:

- The **key** is the memory address.
- The **value** is the **cache line**, which consists of four fields: `(address/tag, data block, Valid bit, Modified bit)`.

A key-value store treats each value as **atomic**: the value is either fully present and valid, or it is not used at all. We never work with a partially defined value. In the cache, this atomicity applies to the **data block**: when the cache loads data from RAM, it transfers the entire data block as a single unit. The CPU never sees a half-loaded block. This is what makes it meaningful to speak of a cache hit or miss – the block is either fully there (V=1) or it is not (V=0).

Within the cache line, the address/tag serves as the **key**, and the data block is the **atomic value**. The V and M bits are bookkeeping metadata that support the key-value semantics (validity) and the write-back policy (modification tracking).

1.2.2 The cost of a fully associative cache

A fully associative cache places **one comparator next to every cache line**. When the CPU issues an address, all comparators fire simultaneously to check whether any cache line holds that address. This parallelism is what makes the lookup fast regardless of how many lines the cache has.

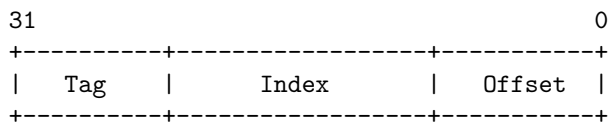
However, this design does not scale well to large caches. Every additional cache line requires an additional comparator, and comparators are expensive in two ways:

- **Chip real estate:** Each comparator occupies physical space on the silicon die. A cache with thousands of lines requires thousands of comparators, leaving less room for other components.
- **Power consumption:** Every comparator draws current on every cache access, even the ones that produce a MISS. With thousands of comparators switching simultaneously, the energy cost becomes significant.

1.2.3 The direct-mapped cache: one comparator for any number of cache lines

The goal of the direct-mapped cache is to reduce the number of comparators to **exactly one**, no matter how many cache lines the cache has. With only one comparator, both the chip area cost and the power cost become constant – they do not grow as the cache grows.

The key insight is: if the address from the CPU can tell us *which single cache line to inspect*, we only need to compare against that one line. We achieve this by decomposing the CPU address into three fields:



- The **Index** bits are used directly to select one specific cache line – the only line where this address could possibly be stored.
- The **Offset** bits identify the specific byte within the data block of that cache line.
- The **Tag** bits are simply the leftover bits: whatever remains after the index and offset fields are carved out of the address.

On a cache lookup, the hardware goes directly to the cache line at the given index, reads the tag stored there, and uses the **single comparator** to check whether that stored tag matches the tag from the CPU address. If they match and V=1, it is a hit. Otherwise, it is a miss. No other cache line is consulted.

The tag is necessary because many different addresses share the same index bits. Two addresses that differ only in their tag bits map to the same cache line slot, and the tag is what distinguishes them. Storing the tag in the cache line lets the single comparator determine whether the line currently holds the right address or some other address that happens to share the same index.

As we will see, the page table for virtual memory uses the same idea of decomposing an address into multiple field, applied to a much larger granularity.

1.3 From CPU-to-RAM Caching to CPU-to-Disk/Swapfile Caching

1.3.1 What the CPU cache does

In Chapters 4a and 4b, both the fully associative cache and the direct-mapped cache served the same purpose: translating between an address held in a CPU register (as used by `lw` or `sw`) and a location in RAM. The cache sits between the CPU and RAM, and when it holds a copy of the needed data, it allows the CPU to avoid the slow trip to RAM entirely. The atomic unit transferred between RAM and the cache is the **data block**.

1.3.2 The same problem, one level down

RAM itself has a capacity limit. A modern computer may run dozens of processes simultaneously, and the total memory demanded by all those processes often exceeds the amount of physical RAM installed. The solution is to extend the same caching idea down one level in the memory hierarchy:

- Just as RAM is slow relative to the CPU, **disk (or SSD) is slow relative to RAM**.
- Just as we used a cache to avoid going to RAM on every instruction, we use **RAM itself as a cache** to avoid going to disk on every memory access.

The disk (or SSD) holds a special file called the **swapfile** (called the **pagefile** on Microsoft Windows). The swapfile is the authoritative store of all memory for all processes. RAM holds only the recently used subset of that memory – exactly as a CPU cache holds only the recently used subset of RAM.

1.3.3 Pages and page frames are atomic

Just as a CPU cache transfers data in atomic **data blocks**, the RAM-to-disk level transfers data in atomic **pages** (also called **page frames** when resident in RAM). A page is a fixed-size, aligned chunk of memory – typically 4096 bytes. The entire page is transferred as a unit between RAM and the swapfile; there is no such thing as a partial page transfer.

When a page is sitting in RAM, we call it a **page frame**. When it is on disk, it is a **page** in the swapfile. The terminology shifts depending on where the data currently lives, but the size and contents are the same.

The key-value framing still applies:

- The **key** is the address (as seen by the CPU).
- The **value** is the atomic page in the swapfile.

RAM is the cache. A page frame in RAM is a cached copy of a page in the swapfile, just as a data block in a CPU cache is a cached copy of data in RAM.

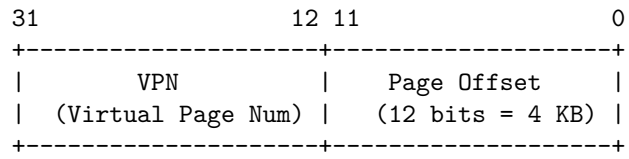
1.3.4 Decomposing the virtual address for RAM-to-disk caching

In Chapters 4a and 4b, we decomposed the virtual address from the CPU into three fields:

```
+----------+------------------+-----------+
|   Tag    |      Index       |   Offset  |
+----------+------------------+-----------+
```

The index selected a cache line; the offset selected a byte within the data block of that line; the tag distinguished between different addresses that share the same index.

When we move down one level – from CPU-to-RAM caching to RAM-to-disk caching – we decompose the virtual address differently:



The **page offset** plays the same role as the block offset before: it selects a specific byte within the page, and it passes through translation unchanged. Pages are typically **4096 bytes** – much larger than a typical CPU cache data block of 32 or 64 bytes. A 4096-byte page requires a 12-bit offset (since 2^12 = 4096), so the lower 12 bits of the virtual address are the page offset, and the remaining upper bits are the VPN.

Just as a CPU cache can be thought of as an array of data blocks indexed by the cache index field, **RAM can be thought of as an array of page frames**, where the **PFN (Page Frame Number)** is the index into that array. If RAM contains 4 GB of physical memory and each page frame is 4096 bytes, then there are 4 GB / 4 KB = 1,048,576 page frames, numbered PFN 0 through PFN 1,048,575.

The actual, authoritative copy of each page lives on disk in the swapfile. A page frame in RAM is simply a cached copy of one of those disk pages. Just as a cache line carries a valid bit, each disk page can be thought of as either:

- **Occupied/Present (P=1):** a page frame in RAM holds a copy of the disk page from the swapfile.
- **Unoccupied/Not Present (P=0):** no page frame holds a copy of the disk page.

When all page frames are occupied and a new page must be brought in from disk, the OS must evict one – choosing a victim page frame, writing it back to the swapfile if it is dirty (M=1), and then loading the new page into the freed frame. This is exactly the same eviction and write-back logic as in the CPU cache.

And finally, as with a CPU cache, each page will be marked as either:

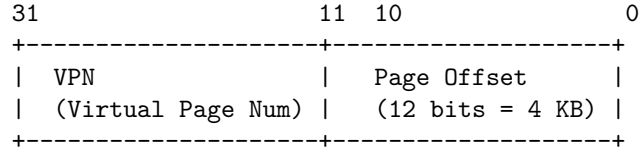
- **Unmodified (M=0):** the page frame holds an *identical* copy of some page from the swapfile.
- **Modified (M=1):** the page frame holds a copy of some page from the swapfile, but its data may have been modifed (for example, due to an `sw` assembly instruction).

1.4 Virtual Addresses and Physical Addresses

Every address a process uses in its instructions (`lw`, `sw`, function calls, etc.) is a **virtual address**. The process never directly names a location in RAM. The CPU's **Memory Management Unit (MMU)** hardware translates each virtual address into a **physical address** – a real location in RAM – before the memory access takes place.

The unit of translation is a **page**: a fixed-size, aligned chunk of memory. A typical page size is **4096 bytes (4 KB)**. Just as a cache data block was 32 bytes and addresses within a block were block-aligned, pages are 4096-byte chunks and their base addresses are 4096-byte aligned.

The virtual address is split into two fields:



- **VPN (Virtual Page Number):** Identifies which virtual page is being accessed. This plays the role of both the **index** and the **tag** from the direct-mapped cache – as explained in the next section.

- **Page Offset:** The byte position within the page.

Recall from the section on direct-mapped CPU caches, the virtual address is decomposed into (tag, index, offset). The offset referred to the offset in the data block of a cache line. The page offset now plays the role of the **block offset** from the CPU cache. Crucially, the page offset passes through the translation unchanged – the offset within a page is the same in both the virtual address and the physical address (see the "->" in the diagram below).

1.5 The Page Table as a Direct-Mapped Cache

Here is the central analogy of this chapter:

> **The page table is a direct-mapped cache in which:**
>
> - The **index** is the **Virtual Page Number (VPN)**.
> - The **data block** is a **Page Frame Number (PFN)** – a pointer to the physical page in RAM – or an indication that the page is not currently resident in RAM.

The **Page Table** can be thought of as an array of **Page Table Entry (PTE)** elements. The **key** in our key-value store continues to be the virtual address, or more specifically the VPN (the index into our array). The **value** is now the **Page Table Entry (PTE)**, which includes the PFN, a pointer to the physical page in RAM.

There is also an important difference between a Page Table and a CPU cache. The Page Table contains an entry for *every* possible page in the virtual address space. However, in a 64-bit address space, the operating system will only map into the virtual address space a limited number of memory segments (using the `mmap()` system call). The Page Table reflects this, by having large "holes" where the pages have not been mapped into our virtual address space.

NOTE: Where do these large holes in the Page Table come from?
In Linux, in a virtual address space of 4 GB (for a 32-bit addresses) or 16 exabytes (16 EB) (for a 64-bit address space). When a new program is loaded into an address space, the address space is initially empty (unmapped). The linking loader of Linux then uses mmap to create mapped memory segments with text, data, stack and other memory segments. Anything not mapped becomes a "hole". To see an example, execute `cat /proc/self/maps` in Linux. an example of

1.5.1 Page Frame Number (PFN)

Just as RAM is divided into virtual pages from the process's perspective, it is also divided into **page frames** from the hardware's perspective. Each page frame is a 4096-byte aligned chunk of physical RAM. A **PFN (Page Frame Number)** is simply the index of one of these physical frames.

To form the final physical address, the MMU concatenates the PFN with the original page offset:

```
Physical address:
+---------------------+--------------------+
|  PFN                |   Page Offset      |
|  (Page Frame Num)   |   (unchanged)      |
+---------------------+--------------------+
```

1.5.2 Structure of the Page Table

The page table is an array in RAM, indexed directly by the VPN. Entry number `i` describes virtual page `i`. Each entry is called a **Page Table Entry (PTE)** and carries:

- **P (Present bit):** 1 if the page is currently present in RAM; 0 if not.
- **PFN:** The physical frame number where this page lives in RAM (meaningful only when P=1).
- **Additional flags:** Read/write/execute permissions, dirty bit, etc. (discussed later).

```
Page Table (indexed by VPN):

VPN  +---+-------------------+
  0  | P |  PFN (unless P=0) |
     +---+-------------------+
  1  | P |  PFN (unless P=0) |
     +---+-------------------+
  2  | P |  PFN (unless P=0) |
     +---+-------------------+
 ...      ...
     +---+-------------------+
  N  | P |  PFN (unless P=0) |
     +---+-------------------+
```

Because the VPN is used directly as an array index, **no comparator is needed** to find the right entry. The MMU simply computes the address of the PTE as:

```
PTE address = page_table_base_register + (VPN * sizeof(PTE))
```

This is precisely the direct-mapped property: the address (here, the VPN) selects exactly one entry, with no search required.

1.5.3 Why There Is No Separate Tag?

In a direct-mapped CPU cache, the stored tag is needed because many different addresses share the same index – the cache has far fewer lines than there are possible addresses. The tag distinguishes which of those many addresses is actually stored in a given line.

The page table has one entry **per virtual page**, so the VPN uniquely identifies the entry. There is no ambiguity about which virtual page an entry belongs to – entry number `i` always and only describes virtual page `i`. This is why no tag is stored inside the PTE itself. (The tag is implicit: it equals the entry's own index.)

1.5.4 Many processes: the key is now pid+address

In Chapters 4a and 4b, there was only one address space: the CPU issued an address, and that address uniquely identified a location in RAM. With multiple processes running simultaneously, this is no longer true. Two different processes may both be executing instructions that reference the same address – for example, both may have a variable at virtual address `0x0000_8000`. From each process's point of view, that address is perfectly well-defined, but they refer to completely different data.

This means the key that uniquely identifies a page is no longer just the address. It is the pair:

```
(process ID/pid, address)
```

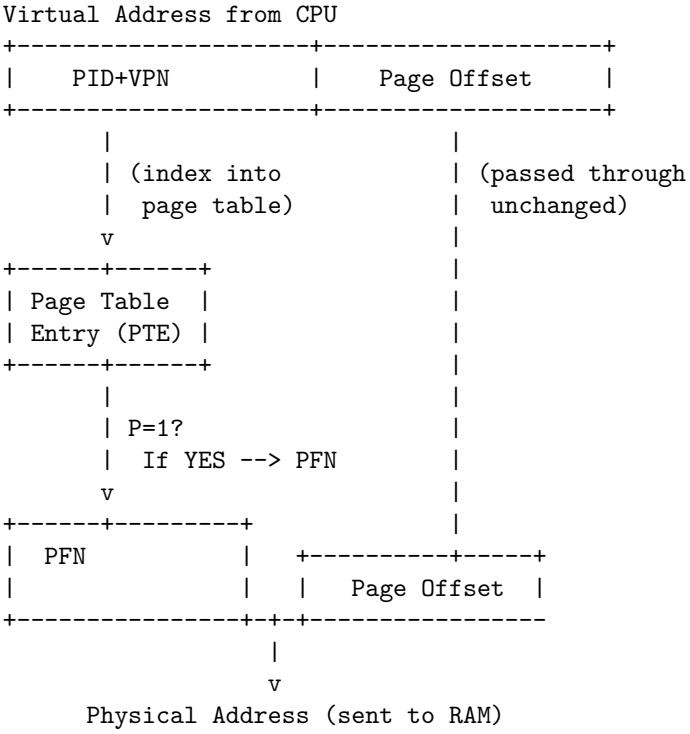
If process PID1 and process PID2 both access virtual address `addrX`, RAM will (when both pages are resident) contain two separate page frames:

```
|  PID   | Address |   Page Frame (4096 bytes)  |
+--------+---------+----------------------------+
|  PID1  |  addrX  |  [data belonging to PID1]  |
+--------+---------+----------------------------+
|  PID2  |  addrX  |  [data belonging to PID2]  |
+--------+---------+----------------------------+
```

These are two distinct page frames with distinct contents, even though the address `addrX` is the same in both. The process ID is what separates them. The same principle applies in the swapfile: every page stored there is identified by `(pid, address)`, not by address alone.

1.6 Address Translation: Step by Step

Here is what the address translation hardware does on every memory access:



1. Split the virtual address into VPN and page offset.
2. Use VPN to index into the page table and retrieve the PTE.
3. If `P=1` (page frame is present): extract the PFN and concatenate it with the page offset to form the physical address. Proceed normally.
4. If `P=0` (page frame is **not** present): the MMU raises a **page fault** exception. The CPU transfers control to the operating system's **page fault handler**.

1.7 Page Faults and the Swapfile

1.7.1 The Swapfile

When RAM is full and the OS needs to make room for a new page, it **evicts** a page from RAM – analogous to evicting a cache line when the cache is full. The evicted page is written to disk into a special file called the **swapfile** (on Linux) or **page file** (on Windows). The corresponding PTE is updated: `P` is set to 0, and the PFN field is used to tell Linux which device (and which swapfile) to look at.

When the process later accesses that page again, the MMU sees `P=0` and triggers a page fault.

1.7.2 The Page Fault Handler

The page fault handler is a routine in the OS kernel. When a page fault occurs, the handler:

1. **Finds the page in the swapfile.** The swapfile is organized as a sequence of page-sized slots. The handler looks up the swapfile location for the faulting PPN.
2. **Finds a free page frame in RAM.** If RAM is full, the OS evicts a page using a policy such as LRU (exactly as with cache eviction). If the evicted page is dirty (modified), it is written to the swapfile first – this is the same write-back policy you saw in Chapter 4a.
3. **Reads the page from the swapfile into the free frame.** The disk read copies 4096 bytes into RAM.
4. **Updates the PTE:** sets `P=1` and records the new PFN.
5. **Restarts the faulting instruction.** The process resumes as if nothing happened, and this time the MMU finds `P=1`.

From the process's point of view, the page fault is invisible (aside from a brief delay). This is the same transparency that the CPU cache provides: the program does not know whether data came from cache or from RAM.

```
Page Fault Sequence:

CPU issues "lw" assembly instruction for some virtual address
          |
          v
       MMU: P=0 in PTE
          |
          v
    PAGE FAULT EXCEPTION
          |
          v
   OS Page Fault Handler:
   1. Locate page in swapfile
   2. Evict a page frame (write-back if dirty)
   3. Read page from swapfile --> RAM frame
   4. Update PTE: P=1, PFN = new frame
          |
          v
   Restart faulting instruction
          |
          v
       MMU: P=1 --> PFN --> physical address
          |
```

```
        v
"lw" assembly instruction completes normally
```

1.8 The MMU and the TLB

1.8.1 The cost of naive address translation

Consider what happens when the CPU executes `lw` with a virtual address. Before the load can complete, the address translation hardware must:

1. Compute the address of the PTE in RAM: `page_table_base_register + (VPN * sizeof(PTE))`.
2. Read the PTE from RAM to obtain the PFN.
3. Form the physical address and read the actual data word from RAM.

Step 2 requires a RAM access just to find out *where* in RAM the data lives. Then step 3 requires a second RAM access to fetch the data itself. Every single `lw` or `sw` would require two trips to RAM instead of one. Since RAM accesses already take 50-100 nanoseconds, doubling that cost on every memory instruction would be unacceptable.

1.8.2 The MMU: bringing translation onto the CPU chip

The solution is the **MMU (Memory Management Unit)**, a hardware unit that is part of the CPU chip itself. The MMU performs virtual-to-physical address translation in hardware, fast enough that it does not become a bottleneck.

The key to making the MMU fast is the observation from Chapter 4a: programs exhibit **temporal locality** – they tend to re-use the same addresses repeatedly. This means a running process tends to access the same small set of pages over and over. If the MMU can remember the translations for those pages, it rarely needs to go to RAM to look up a PTE.

The MMU therefore contains a small, fast cache of recently used PTEs. This cache is called the **TLB (Translation Lookaside Buffer)**. The TLB is a **fully associative cache** (exactly as described in Chapter 4a), in which:

- The **key** is the VPN.
- The **value** is the corresponding PTE (including the PFN and the V bit).

Because the TLB is fully associative, all entries are checked in parallel using one comparator per entry. Because it lives on the CPU chip – not in RAM – a TLB hit takes only 1-2 nanoseconds, the same as an ordinary CPU cache access.

1.8.3 How the MMU and TLB work together

When the CPU issues a virtual address for `lw`, the MMU proceeds as follows:

```
Virtual address from CPU
       |
       v
MMU extracts VPN
       |
       v
```

```
TLB lookup (fully associative, on-chip):
  HIT?  --> PFN found immediately (no RAM access needed)
  MISS? --> read PTE from page table in RAM,
            load PTE into TLB,
            extract PFN
        |
        v
Form physical address: PFN + page offset
        |
        v
Access RAM (or cache) at physical address
to fetch the single data word for "lw"
```

On a TLB hit – which is the common case – the entire translation happens on-chip without any RAM access. The only RAM access is the one the `lw` instruction itself needs: fetching the single data word. Address translation becomes effectively free.

On a TLB miss, one additional RAM access is needed to read the PTE. The MMU loads that PTE into the TLB so that subsequent accesses to the same page will hit. This is the same principle as loading a cache line on a cache miss.

Note that the CPU cache (Chapter 4a/4b) and the TLB are complementary: the CPU cache eliminates repeated RAM accesses for data that has been read recently; the TLB eliminates the extra RAM access that would otherwise be needed to translate every virtual address.

1.9 Process ID: Sharing the Page Table Infrastructure

1.9.1 Each Process Has Its Own Page Table

So far we have spoken as if there is just one page table. In reality, the OS maintains **one page table per process**. When the OS switches from one process to another (a context switch), it updates a special CPU register – the **page table base register** – to point to the new process's page table. This gives each process its own private virtual address space.

1.9.2 Why the Process ID Must Be Part of the Key

Consider two processes, A and B, each with a virtual page numbered VPN=5. These are completely different pages: they belong to different processes, occupy different page frames, and may contain entirely different data. Yet their VPNs are identical.

This creates a bookkeeping problem in any data structure that stores information about pages – in particular:

- The **page table** itself, and
- The **swapfile**.

For both structures, the key that uniquely identifies a page is not the VPN alone. It is the pair:

```
(Process ID, VPN)
```

1.9.3 Process ID in the Page Table

The page table is process-private, so the process ID is implicit: the OS knows which process's page table it is reading because the page table base register was set during the last context switch. The process ID does not need to be stored explicitly in each PTE. However, **conceptually**, each PTE describes a page identified by `(process_id, VPN)` – the process ID is carried by the base register, and the VPN is carried by the table index.

1.9.4 Process ID in the Swapfile

The swapfile, on the other hand, is **shared across all processes**. Multiple processes may have pages swapped out simultaneously. The swapfile must therefore record the `(process_id, VPN)` pair explicitly for each stored page, so that the page fault handler can locate the correct page when a fault occurs.

```
Swapfile layout (conceptual):

| Process ID | VPN |   Page Data (4096 bytes)   |
+------------+-----+----------------------------+
|    PID 3   |  5  |  [byte 0] ... [byte 4095]  |
+------------+-----+----------------------------+
|    PID 7   |  5  |  [byte 0] ... [byte 4095]  |
+------------+-----+----------------------------+
|    PID 3   | 12  |  [byte 0] ... [byte 4095]  |
+------------+-----+----------------------------+
|    PID 7   | 22  |  [byte 0] ... [byte 4095]  |
+------------+-----+----------------------------+
```

Notice that PID 3 and PID 7 both have a page with VPN=5 in the swapfile. They are distinct entries because the process ID differs. Without the process ID, the swapfile could not distinguish between them.

1.10 Comparison: Cache vs. Page Table

The table below summarizes the parallel between the concepts in Chapters 4a/4b and this chapter.

CPU Cache Concept	Virtual Memory Equivalent
Cache	Page Table
Cache line	Page Table Entry (PTE)
Data block (32 bytes)	Page Frame (4096 bytes); found at PFN
Data Block offset	Page offset
Tag (direct-mapped)	Not used; PTE has no tag
Index (direct-mapped)	VPN as an index into the page table
Valid bit (V)	Present bit (P) (Is PFN present in RAM?)
Modified/dirty bit (M)	Dirty bit in PTE
Cache miss	Page fault
Eviction + write-back to RAM	Eviction + write-back to swapfile
RAM	swapfile on disk
Cache holds copy of RAM	RAM holds copy of pages in swapfile
LRU eviction policy	Clock algorithm for eviction of PFN

The memory hierarchy now has three levels: **cache → RAM → disk**. Each level is slower and larger than the one above it, and each level uses the same fundamental idea – keep recently used data close to the CPU, and fetch from the next level only on a miss (or fault).

1.11 Summary

Concept	Description
Virtual address	Address used by the process; not a real RAM location
Physical address	Real RAM location, produced by the MMU
VPN	Virtual Page Number; indexes into the page table
PFN	Page Frame Number; identifies a physical page in RAM
Page offset	Byte position within a page; unchanged by translation
Page Table Entry (PTE)	Stores V bit + PFN (or non-resident indicator) for one virtual page
Page fault	Exception raised when P=0; OS must load the page from disk
Swapfile	Disk file holding pages evicted from RAM
Page fault handler	OS routine that reads a page from swapfile into RAM and updates the PTE
Write-back (pages)	Dirty pages are written to swapfile only when evicted from RAM
Process ID in swapfile	The key for a swapfile entry is `(process_id, VPN)`, not VPN alone
Page table base register	CPU register pointing to the current process's page table; updated on context switch