Chapter 4d: Virtual Memory and the Page Table

Gene Cooperman

Copyright ©2026 Gene Cooperman, gene@ccs.neu.edu
This text may be copied as long as the copyright notice remains and no text is modified.

Contents

4d Virtual Memory and the Page Table **1**
4d.1 Caching: From CPU to RAM to Disk/SSD . . . 1
4d.2 Pages: The Data Blocks of Virtual Memory . . . 2
4d.3 Virtual Addresses and Physical Addresses . . . 3
4d.4 The Page Table as a Direct-Mapped Cache . . . 4
4d.5 Address Translation: Page Hits and Page Faults . . . 6
4d.6 Hardware Support: The MMU and the TLB . . . 8
4d.7 Comparison: Cache vs. Page Table . . . 10
4d.8 Summary . . . 11

THIS IS STILL A WORK IN PROGRESS.

4d Virtual Memory and the Page Table

4d.1 Caching: From CPU to RAM to Disk/SSD

Chapters 4a, 4b, and 4c described three ways to build a cache: the *fully associative*, *direct-mapped*, and *set-associative* caches. In each one, the cache is a key-value store: the key comes from a memory address, and the value is an atomic data block. This chapter applies the same idea one level further down the memory hierarchy.

4d.1.1 The memory hierarchy

A computer's memory is organized as a **memory hierarchy**. Each level is larger and slower than the level above it:

```
CPU  ⟶  CPU caches (L1, L2, L3)  ⟶  RAM  ⟶  disk or SSD
fastest, smallest                          slowest, largest
```

Each level keeps copies of recently used data from the level below it. In the earlier chapters, RAM was large, but it was slow compared with the CPU. So, the solution to the problem of slow RAM is a small, fast cache.

4d.1.2 The same problem, one level down

One level down, we find the same problem. The disk (or SSD) is large but slow, and RAM is relatively small and fast. 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). In our simplified model, the swapfile holds 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.

This is **virtual memory**: the operating system creates the illusion that each process has a large, private address space, even though only part of that space is resident in RAM at any time. The data structure that makes this possible is the **page table**. As we will see, the page table works on the same principles as the direct-mapped cache of Chapter 4b. If you understand the direct-mapped cache, you already understand most of the page table.

4d.2 Pages: The Data Blocks of Virtual Memory

4d.2.1 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.

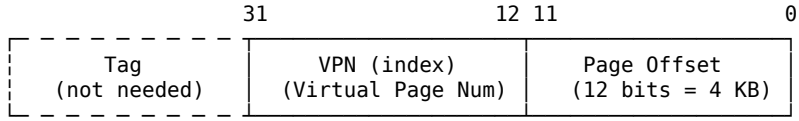
4d.2.2 Decomposing the virtual address for RAM-to-disk caching

In Chapter 4b, we decomposed the address from the CPU into three fields:



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 **VPN** plays the role of the **index**: it selects one entry in the page table (described below). The dotted box on the left shows where a **tag** would be, but it contains no bits. In a CPU cache, the cache has far fewer cache lines than there are data blocks in RAM. So the index uses only some of the bits above the offset, and the remaining high bits form the tag. The page table is different: it includes one entry for *every* virtual page. So the index (the VPN) uses all of the bits above the page offset, and no bits are left over for a tag.

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 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: it chooses a victim page frame, writes it back to the swapfile if it is dirty (M=1), and then loads 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:

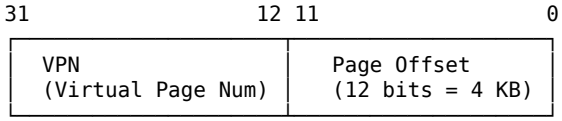
- **Unmodified (M=0):** the page frame holds an *identical* copy of some page from the swapfile.
- **Modified (M=1):** the page frame holds a copy of some page from the swapfile, but its data may have been modifed (for example, due to an `sw` assembly instruction).

4d.3 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. Each virtual address is translated 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 64 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 the **index** from the direct-mapped cache. No tag is needed, as explained above.
- **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).

4d.4 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)**, which points 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.

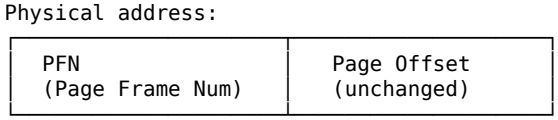
There is also an important difference between a Page Table and a CPU cache. The Page Table contains an entry for *every* possible page in the virtual address space. However, in a 64-bit address space, the operating system will only map into the virtual address space a limited number of memory segments (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, a 32-bit address space has room for 4 GB, and a 64-bit address space has room, in principle, for 16 exabytes (16 EB). 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.

4d.4.1 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 PFN is concatenated with the original page offset:



4d.4.2 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-1   | 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 address of the PTE is simply:

```
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.

> **NOTE:** *Why is there no separate tag?*
> Recall the empty, dotted tag box in the earlier diagram. Since the page table includes one entry for every virtual page, entry number `i` always describes virtual page `i`. So, unlike a CPU cache line, a PTE never needs to store a tag.

The formula above uses a special CPU register, the **page table base register**, which points to the 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 the page table base register to point to the new process's page table. This gives each process its own private virtual address space.

Because each process has its own page table, the page table never needs to store a process ID. The process ID is implicit in which page table the base register points to. Conceptually, each PTE still describes the page `(Process ID, VPN)`: the base register supplies the process ID, and the table index supplies the VPN.

> **NOTE:** *Multi-level page tables.*
> Real operating systems do not store the page table as one flat array. A flat array needs one PTE for every virtual page, even for the large holes in the address space. With 32-bit addresses and 4 KB pages, that is 2^{20} PTEs (4 MB, at 4 bytes per PTE) per process; with 64-bit addresses, a flat array would be far too large to store at all. So real systems use a **multi-level page table**. In a two-level page table, the upper bits of the VPN index a small top-level table, whose entries point to second-level tables. A second-level table is created only for a region of the address space that is actually in use, so a hole costs only one empty entry in the top-level table. (Modern 64-bit CPUs typically use three to five levels, depending on the CPU and on the size of the virtual address space.) Conceptually, a multi-level page table still behaves like one large array indexed by the VPN.

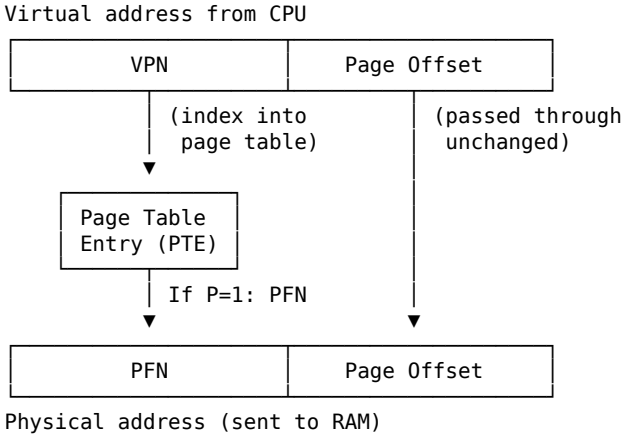
4d.5 Address Translation: Page Hits and Page Faults

For now, think of address translation as a short procedure that is carried out before every `lw` and `sw`. Later, we will see that the hardware performs these same steps.

In cache terms, this lookup can have two outcomes. If `P=1`, the page is already in RAM: this is a **page hit**, just like a cache hit. If `P=0`, the page must first be brought into RAM from the swapfile: this is a **page fault**, the equivalent of a cache miss. (It is called a "fault" rather than a "miss" because the access cannot be completed until the operating system steps in.)

4d.5.1 Translating an address

Here is the procedure:



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): translation stops, and a **page fault** occurs. Control transfers to the operating system's **page fault handler**.

4d.5.2 Evicting a page to the swapfile

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

When the process later accesses that page again, translation finds `P=0`, and a page fault occurs.

4d.5.3 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 page.

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 translation 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
  Translation: P=0 in PTE
          |
          v
        PAGE FAULT
          |
          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
  Translation: P=1 --> PFN --> physical address
          |
          v
    "lw" assembly instruction completes normally
```

4d.5.4 Going Deeper: Many Processes, But One CPU Cache

This subsection returns briefly to the CPU cache of the earlier chapters, to answer a question that often comes up: when many processes share one CPU cache, how does the cache keep their data apart? The short answer is that the CPU cache is keyed by the **physical** address, not the virtual address.

In Chapter 4a, we assumed for simplicity that the CPU runs just a single process. Now that we have virtual and physical addresses, we can refine that picture.

With many processes, two processes may use the same virtual address for completely different data. For example, both may have a variable at virtual address `0x0000_8000`. So a virtual address alone no longer identifies a location in RAM. To identify a virtual page uniquely, we need the pair `(process ID, VPN)`. The page table handles this by giving each process its own page table, as we saw earlier. The swapfile handles it differently, as we will see in the next subsection, Going Deeper: Where the Process ID Is Kept.

The CPU cache solves this problem more simply, by using the **physical address** as its key. In other words, the tag

stored in each cache line comes from the physical address, *not* from the virtual address. By the time an address reaches the CPU cache, it has already been translated. The same virtual address in two different processes is translated to two different page frames, and so to two different physical addresses:

```
PID1:  virtual 0x0000_8000  ⟶  PFN 0x2A  ⟶  physical 0x0002_A000
PID2:  virtual 0x0000_8000  ⟶  PFN 0x51  ⟶  physical 0x0005_1000
```

The two physical addresses have different tags. So the cache never confuses the data of the two processes. The CPU cache does not need the process ID at all. This has three consequences:

1. **No flush on a context switch.** The cache does not need to be cleared when the CPU switches to another process. The old process's data blocks simply stay in the cache until LRU evicts them.
2. **The kernel is no different.** Kernel code and data also live at physical addresses in RAM. So they use the cache in exactly the same way.
3. **Sharing works naturally.** When two processes share a page (for example, the code of a shared library), both page tables map it to the same page frame. Both processes then use the same cache lines, which is exactly what we want.

4d.5.5 Going Deeper: Where the Process ID Is Kept

Since each process has its own page table, the page table never needs to store a process ID. The swapfile is different: it is **shared across all processes**. If two processes, PID 3 and PID 7, each have a page with VPN=5, the swapfile may hold both pages. So the swapfile records the pair `(Process ID, VPN)` for each stored page. Note that in the diagram below, PID 3 and PID 7 each have a page with VPN=5, stored as two distinct entries:

```
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]

4d.6 Hardware Support: The MMU and the TLB

4d.6.1 The MMU: translation in hardware

The procedure in Address Translation: Page Hits and Page Faults runs before every `lw` and `sw`. It could not really be carried out by software: that software would itself need its addresses translated. In a real CPU, these steps are carried out in hardware, by the **MMU (Memory Management Unit)**, a hardware unit on the CPU chip. The MMU splits each virtual address into VPN and page offset, reads the PTE, and forms the physical address. When it finds `P=0`, it raises a hardware **exception**, which transfers control to the operating system's page fault handler. This is why it is called a page *fault*: a fault is an exception that stops an instruction partway through, so that the operating system can fix the problem and then restart the same instruction.

NOTE: *Page hits and page faults in practice.*
The term "page hit" is used in some textbooks, by analogy with a cache hit. More commonly, operating systems say that the page is **resident**, meaning that it is currently in RAM. The term "page fault" is also broader than "cache miss": a page fault can occur for other reasons too, for example, when a process tries to write to a virtual page whose permissions mark it as read-only.

Even in hardware, translation has a cost. Before a `lw` can complete, the MMU 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.

An MMU that worked this way would still be correct. It would simply be slow. The next subsection describes the hardware optimization that makes translation nearly free.

4d.6.2 The TLB: a hardware optimization

The key to speeding up translation is the observation from Chapter 4a: programs exhibit **temporal locality**: they tend to reuse 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.

So the MMU keeps a small cache of recently used PTEs, called the **TLB (Translation Lookaside Buffer)**. In its simplest form, a TLB is a **fully associative cache**, exactly like the cache of Chapter 4a. (Modern CPUs often add a larger, second-level TLB, which is set-associative, like the caches of Chapter 4c.) In either case:

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

Notice that the TLB caches *translations*, not data. The value is a PTE, rather than a copy of data from RAM.

A fully associative level-one TLB checks all of its entries 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.

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. (On some CPUs, such as MIPS, a TLB miss is handled in software instead: the MMU traps to the kernel, and a short kernel routine reads the PTE from the page table and loads it into the TLB.)

Note that the CPU cache of the earlier chapters 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.

> **NOTE:** Like the swapfile, the TLB must keep track of the process ID. Unlike the CPU cache, the TLB is consulted *before* translation, using the VPN. So VPN 5 of PID 3 must not be confused with VPN 5 of PID 7. Older CPUs solved this problem simply, by flushing the entire TLB on every context switch. Many current CPUs instead include a short process identifier in the key of each TLB entry, so that the key becomes `(Process ID, VPN)`. Then the TLB can hold entries for several processes at once, and a process that runs again soon may still find its translations in the TLB.

4d.7 Comparison: Cache vs. Page Table

The table below summarizes the parallel between the concepts in the earlier chapters and this chapter.

CPU Cache Concept	Virtual Memory Equivalent
Cache	Page Table
Cache line	Page Table Entry (PTE)
Data block (64 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	Approximate LRU (e.g., the clock algorithm) for evicting a page frame
Fully associative cache	TLB (cache of recently used PTEs)

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).

4d.8 Summary

Concept	Description
Virtual address	Address used by the process; not a real RAM location
Physical address	Real RAM location, produced by the MMU
MMU	Hardware unit on the CPU chip that performs address translation
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 the P bit + PFN (or non-resident indicator) for one virtual page
Multi-level page table	Real-world page table structure that avoids storing PTEs for holes in the address space
Page hit (resident page)	The page is already in RAM (P=1); often described as *resident*
Page fault	Exception raised when P=0 (among other causes); the OS loads the page from the swapfile
Swapfile	Disk file that, in our simplified model, holds all memory of all processes
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
TLB	Small cache of recently used PTEs, in its simplest form fully associative; key: VPN, value: PTE