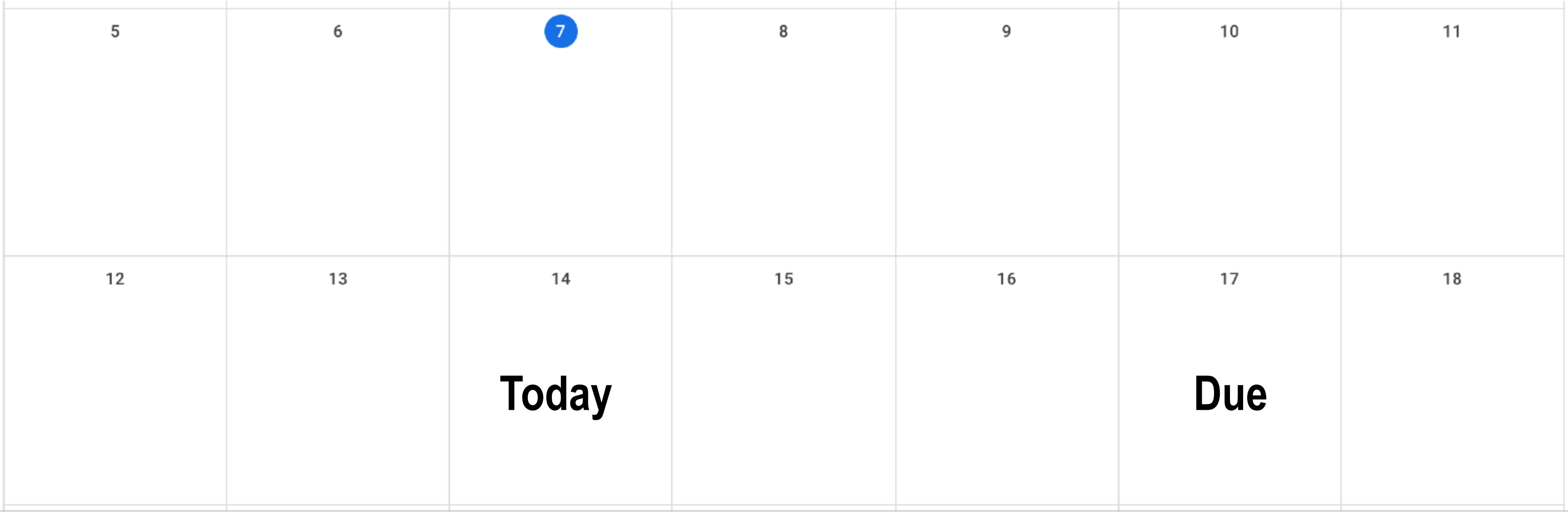
CSC 252: Computer Organization
Spring 2020: Lecture 21

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Announcement

- Programming assignment 4 is out
  - Details: https://www.cs.rochester.edu/courses/252/spring2020/labs/assignment4.html
  - Due on **Apr. 17**, 11:59 PM
  - You (may still) have 3 slip days



5	6	7	8	9	10	11
12	13	14 **Today**	15	16	17 **Due**	18

Another Unsafe Signal Handler Example

Another Unsafe Signal Handler Example

- Assume a program wants to do the following:
  - The parent creates multiple child processes
  - When each child process is created, add the child PID to a queue
  - When a child process terminates, the parent process removes the child PID from the queue

Another Unsafe Signal Handler Example

- Assume a program wants to do the following:
  - The parent creates multiple child processes
  - When each child process is created, add the child PID to a queue
  - When a child process terminates, the parent process removes the child PID from the queue
- One possible implementation:
  - An array for keeping the child PIDs
  - Use a loop to fork child, and add PID to the array after fork
  - Install a handler for SIGCHLD in parent process
  - The SIGCHLD handler removes the child PID

First Attempt

```
void handler(int sig)
{
    pid_t pid;

    while ((pid = wait(NULL)) > 0) { /* Reap child */
        /* Delete the child from the job list */
        deletejob(pid);
    }
}

int main(int argc, char **argv)
{
    int pid;

    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        if ((pid = Fork()) == 0) { /* Child */
            Execve("/bin/date", argv, NULL);
        }
        /* Add the child to the job list */
        addjob(pid);
    }
    exit(0);
}
```

First Attempt

```
void handler(int sig)
{
    pid_t pid;

    while ((pid = wait(NULL)) > 0) { /* Reap child */
        /* Delete the child from the job list */
        deletejob(pid);
    }
}

int main(int argc, char **argv)
{
    int pid;

    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        if ((pid = Fork()) == 0) { /* Child */
            Execve("/bin/date", argv, NULL);
        }
        /* Add the child to the job list */
        addjob(pid);
    }
    exit(0);
}
```

The following can happen:

First Attempt

```
void handler(int sig)
{
    pid_t pid;

    while ((pid = wait(NULL)) > 0) { /* Reap child */
        /* Delete the child from the job list */
        deletejob(pid);
    }
}

int main(int argc, char **argv)
{
    int pid;

    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        if ((pid = Fork()) == 0) { /* Child */
            Execve("/bin/date", argv, NULL);
        }
        /* Add the child to the job list */
        addjob(pid);
    }
    exit(0);
}
```

The following can happen:

- Child runs, and terminates

First Attempt

```
void handler(int sig)
{
    pid_t pid;

    while ((pid = wait(NULL)) > 0) { /* Reap child */
        /* Delete the child from the job list */
        deletejob(pid);
    }
}

int main(int argc, char **argv)
{
    int pid;

    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        if ((pid = Fork()) == 0) { /* Child */
            Execve("/bin/date", argv, NULL);
        }
        /* Add the child to the job list */
        addjob(pid);
    }
    exit(0);
}
```

The following can happen:

- Child runs, and terminates
- Kernel sends SIGCHLD

First Attempt

```
void handler(int sig)
{
    pid_t pid;

    while ((pid = wait(NULL)) > 0) { /* Reap child */
        /* Delete the child from the job list */
        deletejob(pid);
    }
}

int main(int argc, char **argv)
{
    int pid;

    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        if ((pid = Fork()) == 0) { /* Child */
            Execve("/bin/date", argv, NULL);
        }
        /* Add the child to the job list */
        addjob(pid);
    }
    exit(0);
}
```

The following can happen:

- Child runs, and terminates
- Kernel sends SIGCHLD
- Context switch to parent, but before it can run, kernel has to handle SIGCHLD first

First Attempt

```
void handler(int sig)
{
    pid_t pid;

    while ((pid = wait(NULL)) > 0) { /* Reap child */
        /* Delete the child from the job list */
        deletejob(pid);
    }
}

int main(int argc, char **argv)
{
    int pid;

    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        if ((pid = Fork()) == 0) { /* Child */
            Execve("/bin/date", argv, NULL);
        }
        /* Add the child to the job list */
        addjob(pid);
    }
    exit(0);
}
```

The following can happen:

- Child runs, and terminates
- Kernel sends SIGCHLD
- Context switch to parent, but before it can run, kernel has to handle SIGCHLD first
- The handler deletes the job, which does nothing

First Attempt

```
void handler(int sig)
{
    pid_t pid;

    while ((pid = wait(NULL)) > 0) { /* Reap child */
        /* Delete the child from the job list */
        deletejob(pid);
    }
}

int main(int argc, char **argv)
{
    int pid;

    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        if ((pid = Fork()) == 0) { /* Child */
            Execve("/bin/date", argv, NULL);
        }
        /* Add the child to the job list */
        addjob(pid);
    }
    exit(0);
}
```

The following can happen:

- Child runs, and terminates
- Kernel sends SIGCHLD
- Context switch to parent, but before it can run, kernel has to handle SIGCHLD first
- The handler deletes the job, which does nothing
- The parent process resumes and adds a terminated child to job list

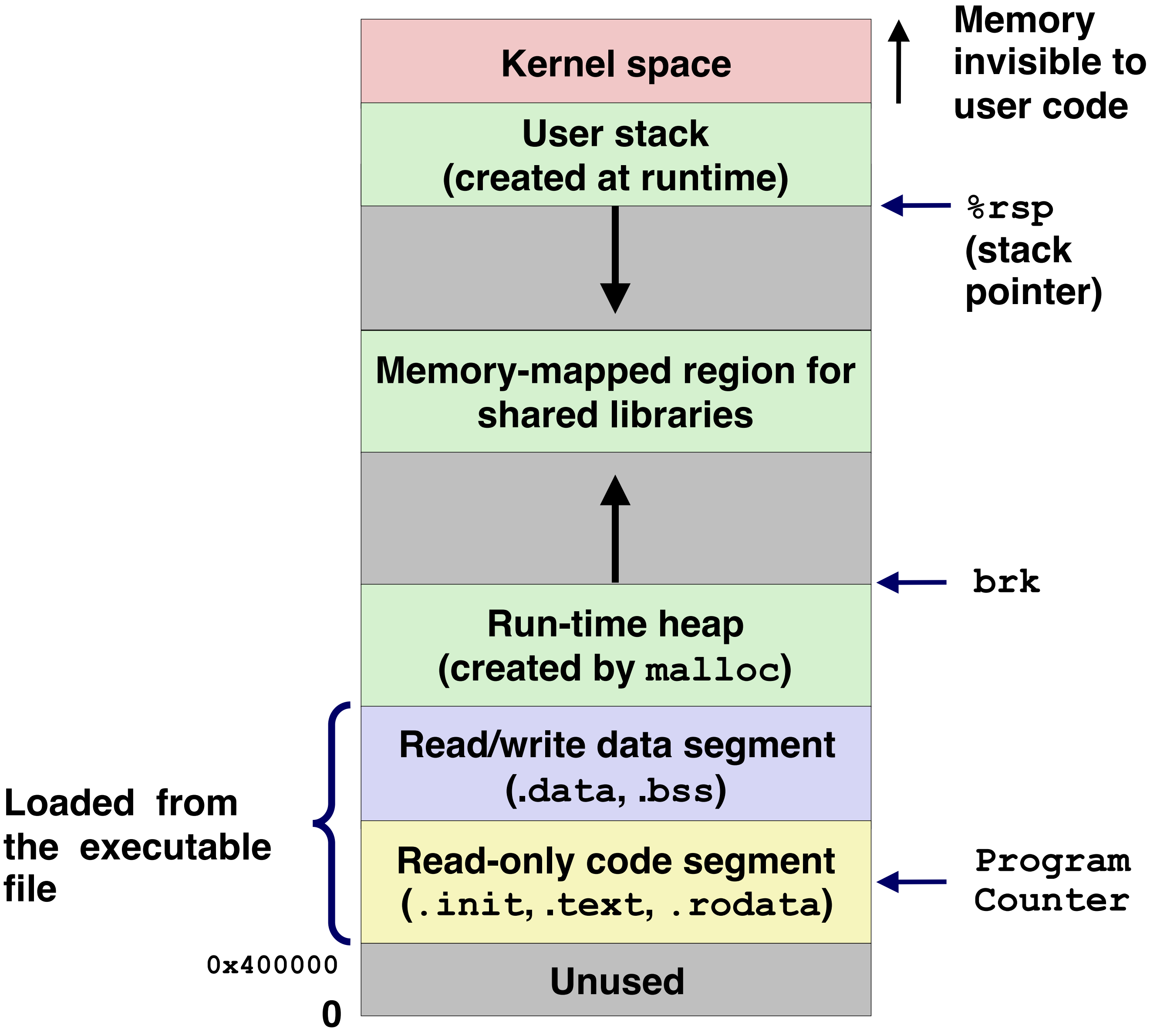
Second Attempt

```
void handler(int sig)
{
    sigset_t mask_all, prev_all;
    pid_t pid;

    sigfillset(&mask_all);
    while ((pid = wait(NULL)) > 0) {
        sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
        deletejob(pid);
        sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
}
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;

    sigfillset(&mask_all);
    signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        if ((pid = Fork()) == 0) {
            Execve("/bin/date", argv, NULL);
        }
        sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
        addjob(pid);
        sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
    exit(0);
}
```

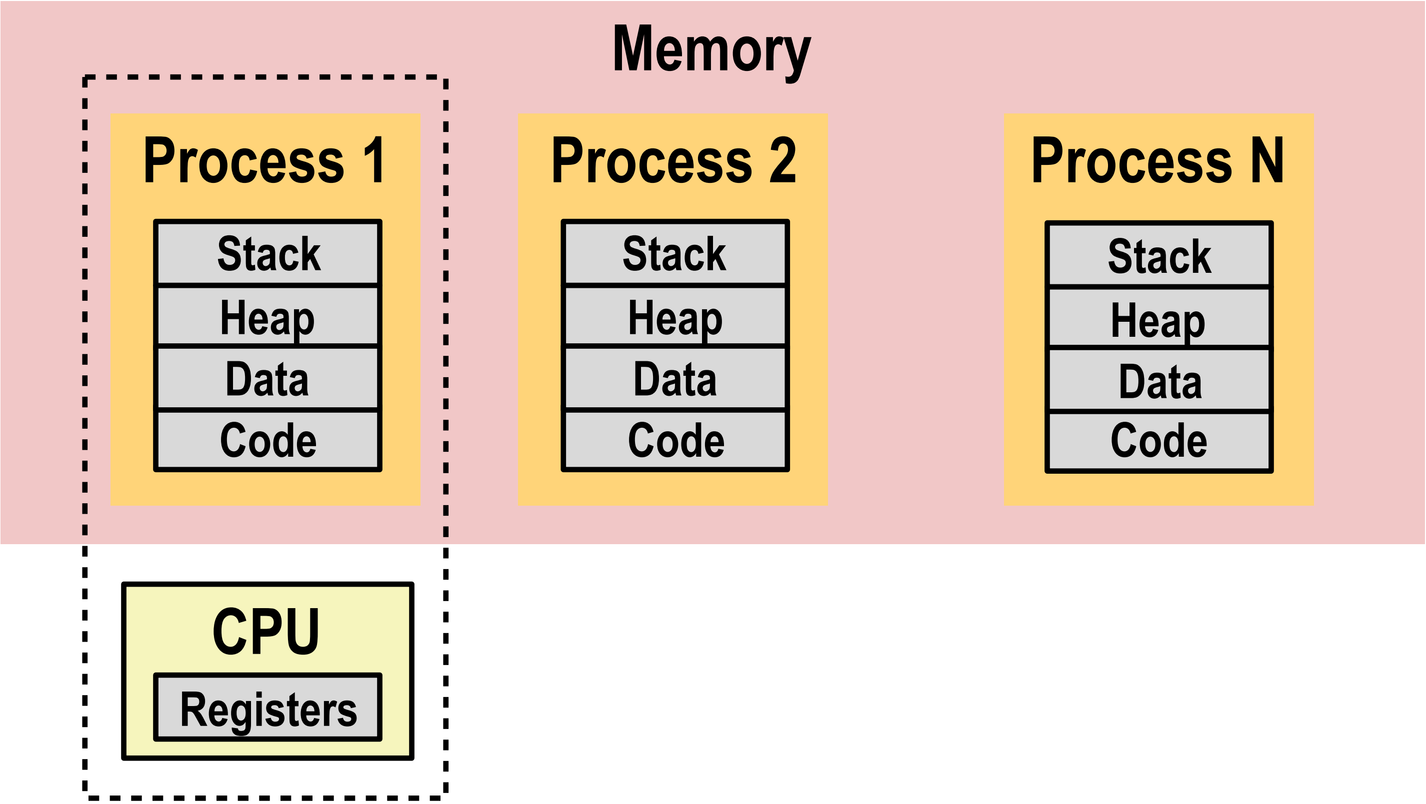
Third Attempt (The Correct One)

```
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, mask_one, prev_one;

    Sigfillset(&mask_all);
    Sigemptyset(&mask_one);
    Sigaddset(&mask_one, SIGCHLD);
    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        Sigprocmask(SIG_BLOCK, &mask_one, &prev_one); /* Block SIGCHLD */
        if ((pid = Fork()) == 0) { /* Child process */
            Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
            Execve("/bin/date", argv, NULL);
        }
        addjob(pid);  /* Add the child to the job list */
        Sigprocmask(SIG_SETMASK, &prev_one, NULL);  /* Unblock SIGCHLD */
    }
    exit(0);
}
```

Process Address Space



Multiprocessing Illustration



Memory

Process 1

Stack

Heap

Data

Code

Process 2

Stack

Heap

Data

Code

Process N

Stack

Heap

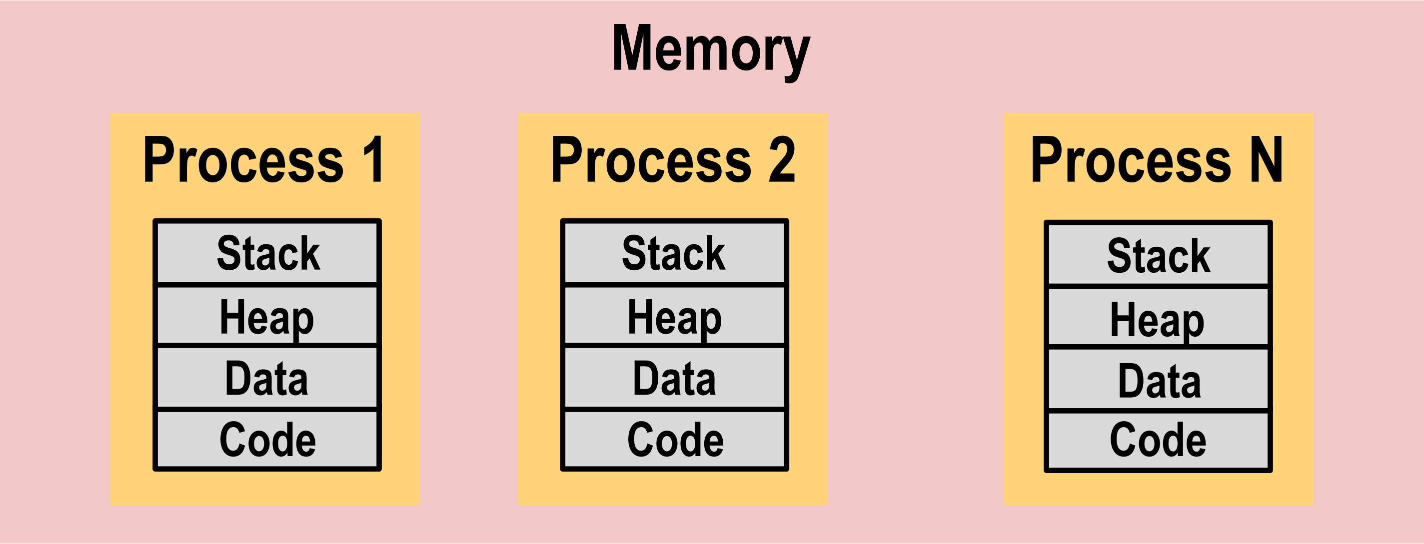
Data

Code

CPU

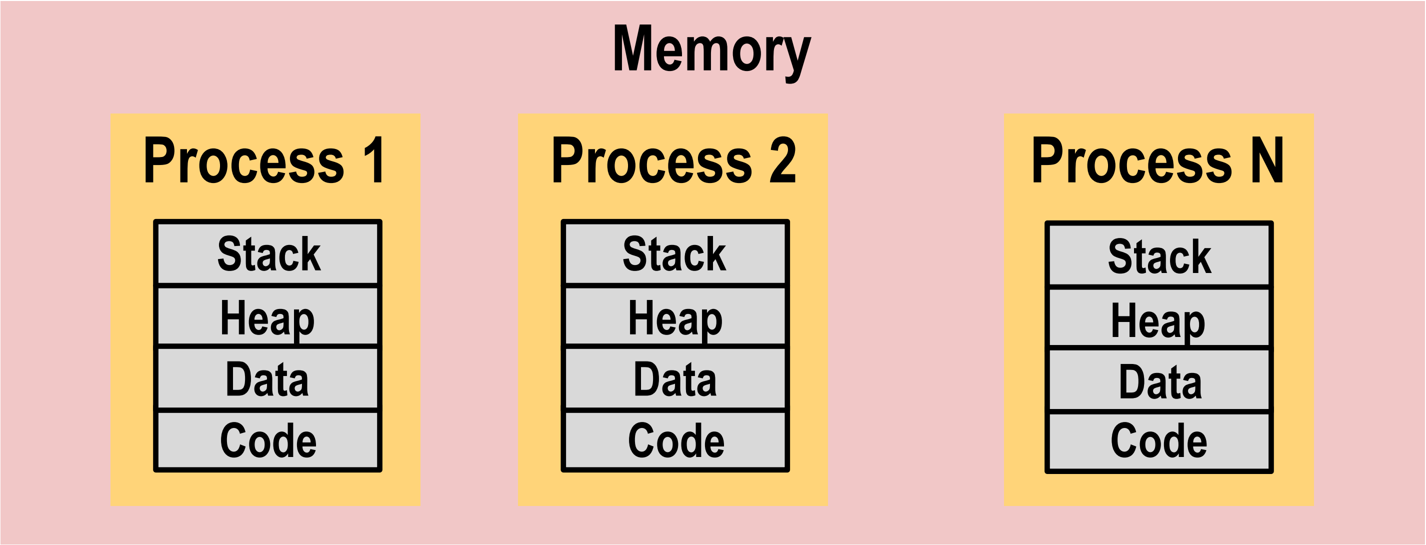
Registers

Problem



- Space:
  - Each process's address space is huge (64-bit): can memory hold it (16GB is just 34-bit)?
  - 2^48 bytes is about 282 TB
  - There are multiple processes, increasing the overhead further

Problem



- Space:
  - Each process's address space is huge (64-bit): can memory hold it (16GB is just 34-bit)?
  - 2^48 bytes is about 282 TB
  - There are multiple processes, increasing the overhead further
- Solution: store all the data in disk, and use memory only for most recently used data
  - Does this sound similar?

The Big Idea: Virtual Memory

The Big Idea: Virtual Memory

- What Does a Programmer Want?

The Big Idea: Virtual Memory

- What Does a Programmer Want?
- Infinitely large, infinitely fast memory
  - Preferably automatically moved to where it is needed

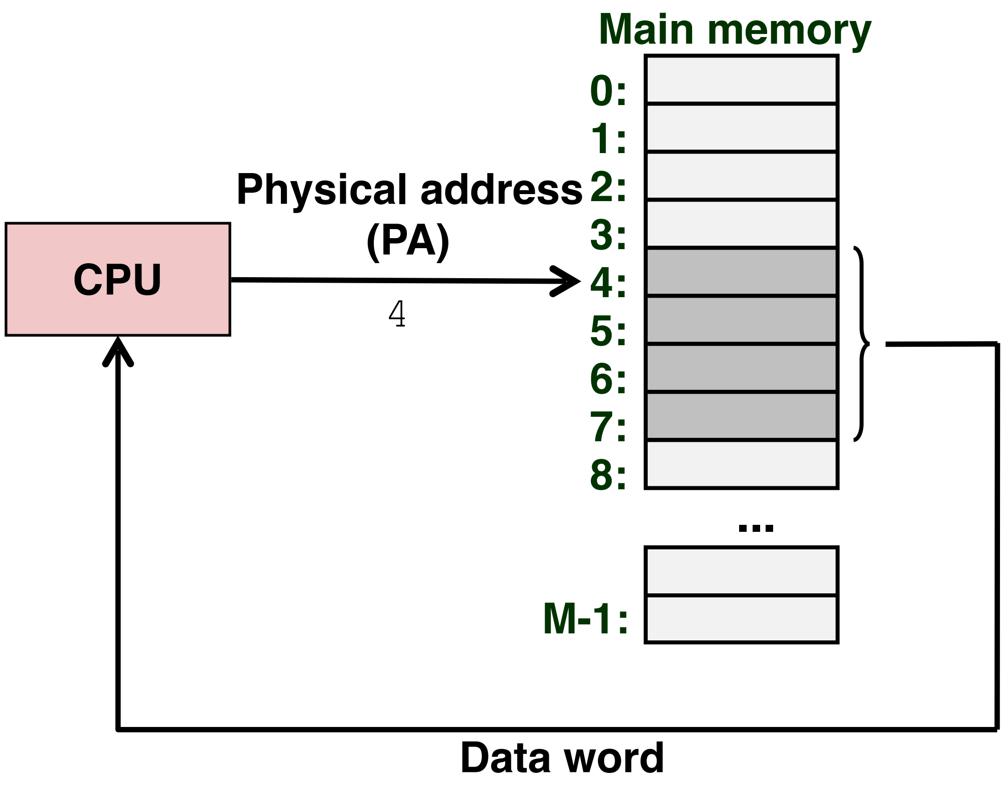
The Big Idea: Virtual Memory

- What Does a Programmer Want?
- Infinitely large, infinitely fast memory
  - Preferably automatically moved to where it is needed
- Virtual memory to the rescue
  - Present a large, uniform memory to programmers
  - Data in virtual memory by default stays in disk
  - Data moves to physical memory (DRAM) “**on demand**”
  - Disks (~TBs) are much larger than DRAM (~GBs), but 10,000x slower.
  - Effectively, virtual memory system transparently share the physical memory across different processes
  - Manage the sharing automatically: hardware-software collaborative strategy (too complex for hardware alone)

Today

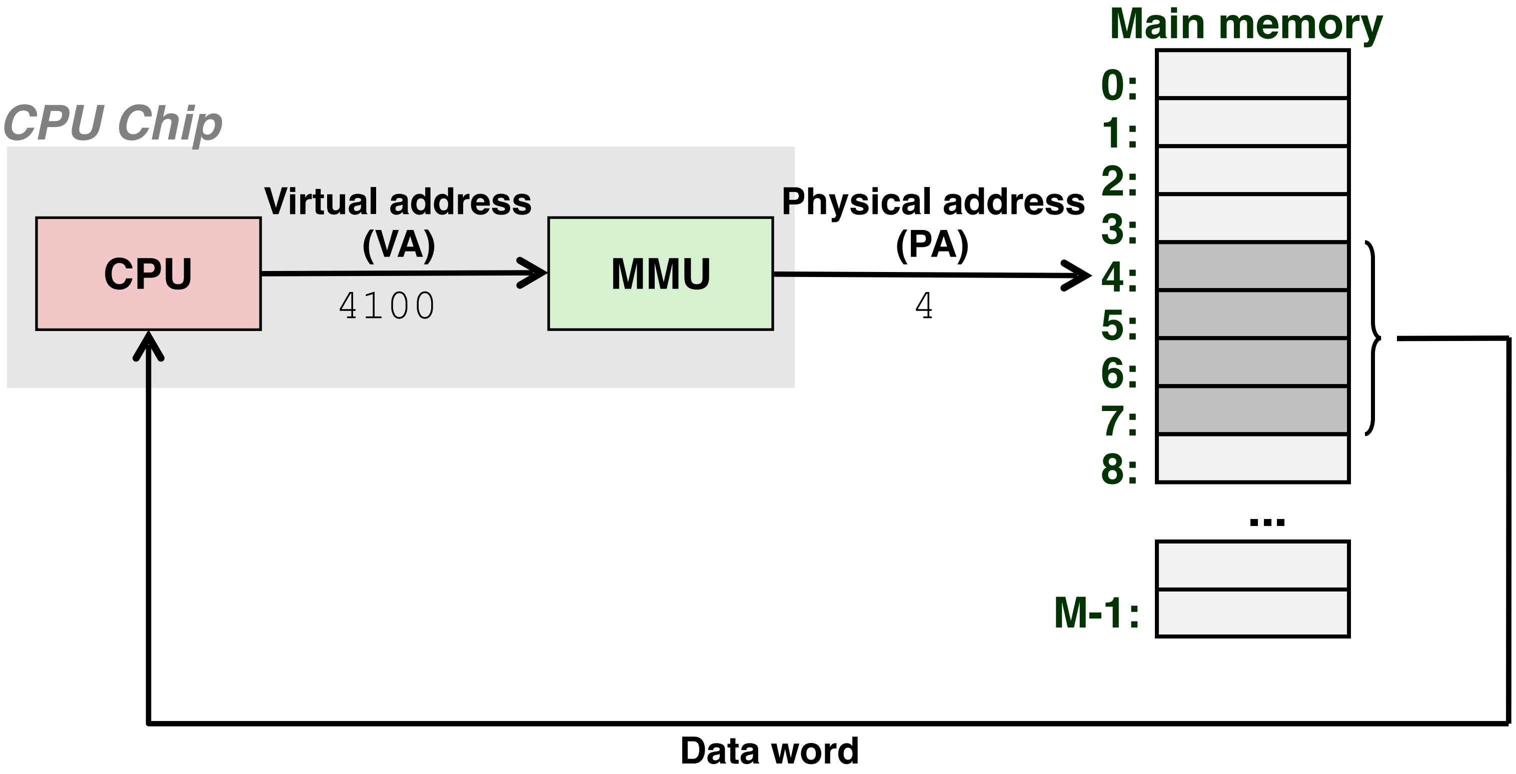
- Virtual memory (VM) illustration
- VM basic concepts and operation
- Other critical benefits of VM
- Address translation

A System Using Physical Addressing



- Used in “simple” systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames

A System Using Virtual Addressing



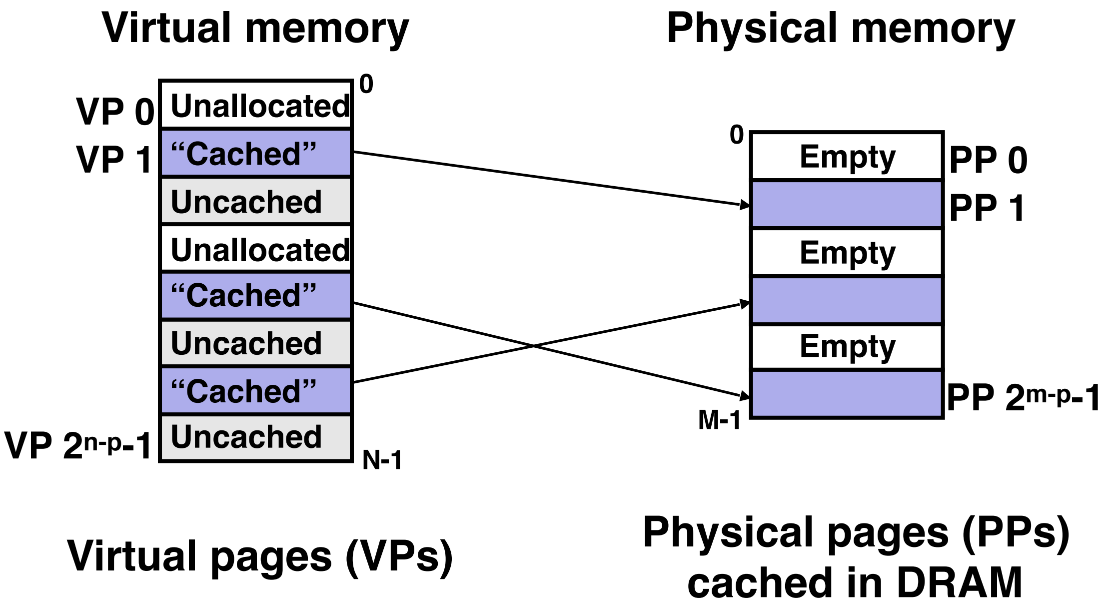
- Used in all modern servers, laptops, and smart phones
- One of the great ideas in computer science
- MMU: Memory Management Unit

Today

- Virtual memory (VM) illustration
- VM basic concepts and operation
- Other critical benefits of VM
- Address translation

VM Concepts

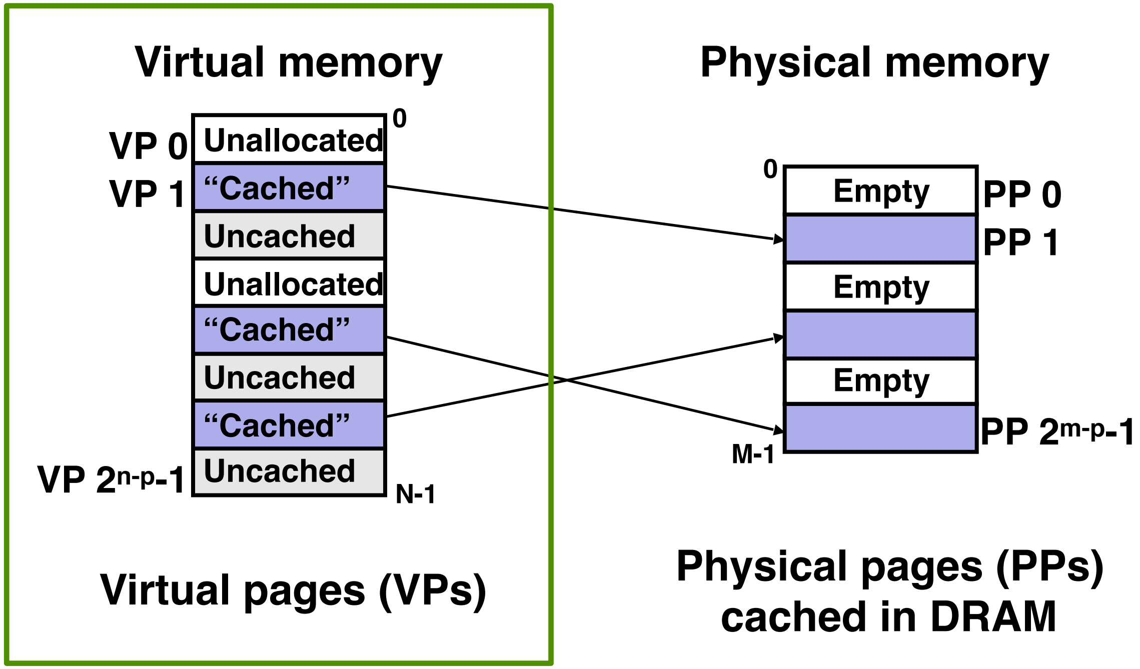
- Conceptually, *virtual memory* is an array of N contiguous *pages* (page size $P = 2^p$ bytes)
- Physical memory is also divided into pages. Each physical page (sometimes called *frames*) has the same size as a virtual page. Physical memory has way fewer pages.
- A page can either be on the (“uncached”) disk or in the physical memory (“cached”).



VM Concepts

- Conceptually, *virtual memory* is an array of N contiguous *pages* (page size P = 2^p bytes)
- Physical memory is also divided into pages. Each physical page (sometimes called *frames*) has the same size as a virtual page. Physical memory has way fewer pages.
- A page can either be on the ("uncached") disk or in the physical memory ("cached").

What programmers see

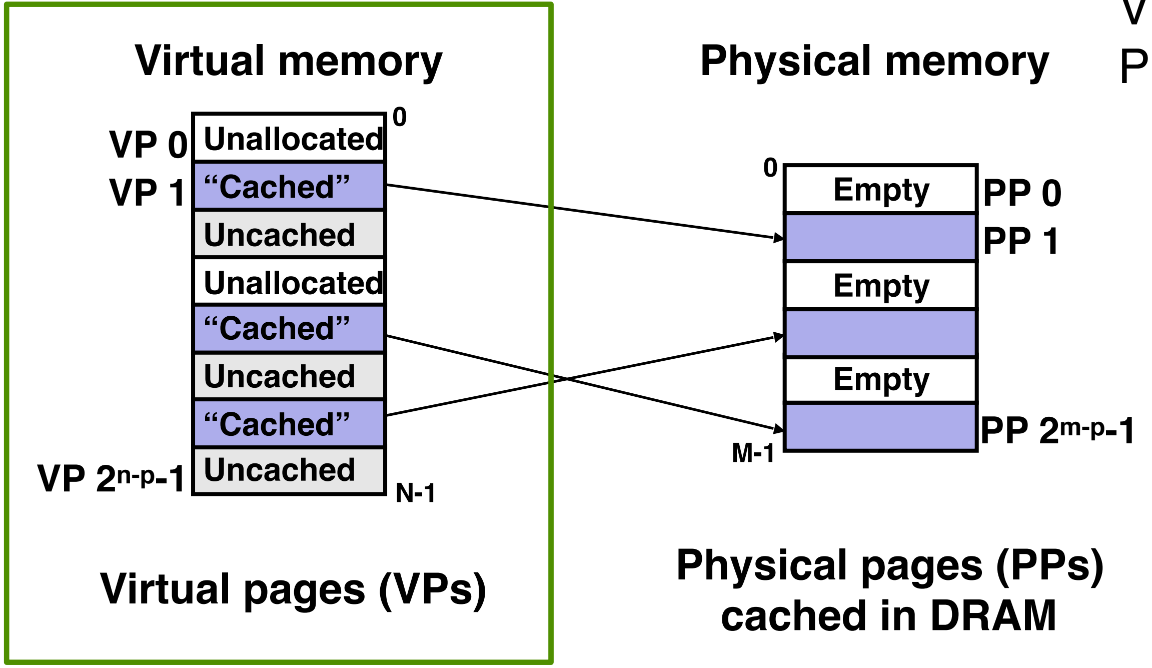


VM Concepts

- Conceptually, *virtual memory* is an array of N contiguous *pages* (page size P = 2^p bytes)
- Physical memory is also divided into pages. Each physical page (sometimes called *frames*) has the same size as a virtual page. Physical memory has way fewer pages.
- A page can either be on the ("uncached") disk or in the physical memory ("cached").

What programmers see

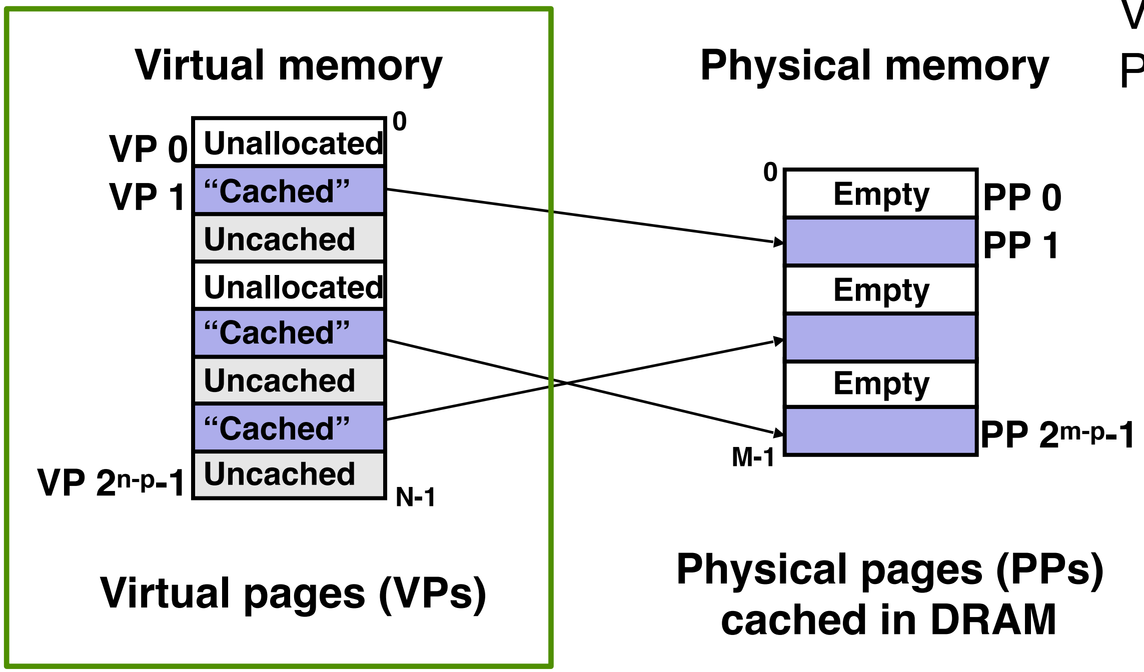
Assuming page size is 4K.
Virtual memory size is 4GB
Physical memory size is 1GB



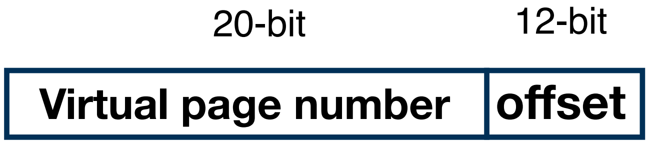
VM Concepts

- Conceptually, *virtual memory* is an array of N contiguous *pages* (page size $P = 2^p$ bytes)
- Physical memory is also divided into pages. Each physical page (sometimes called *frames*) has the same size as a virtual page. Physical memory has way fewer pages.
- A page can either be on the ("uncached") disk or in the physical memory ("cached").

What programmers see



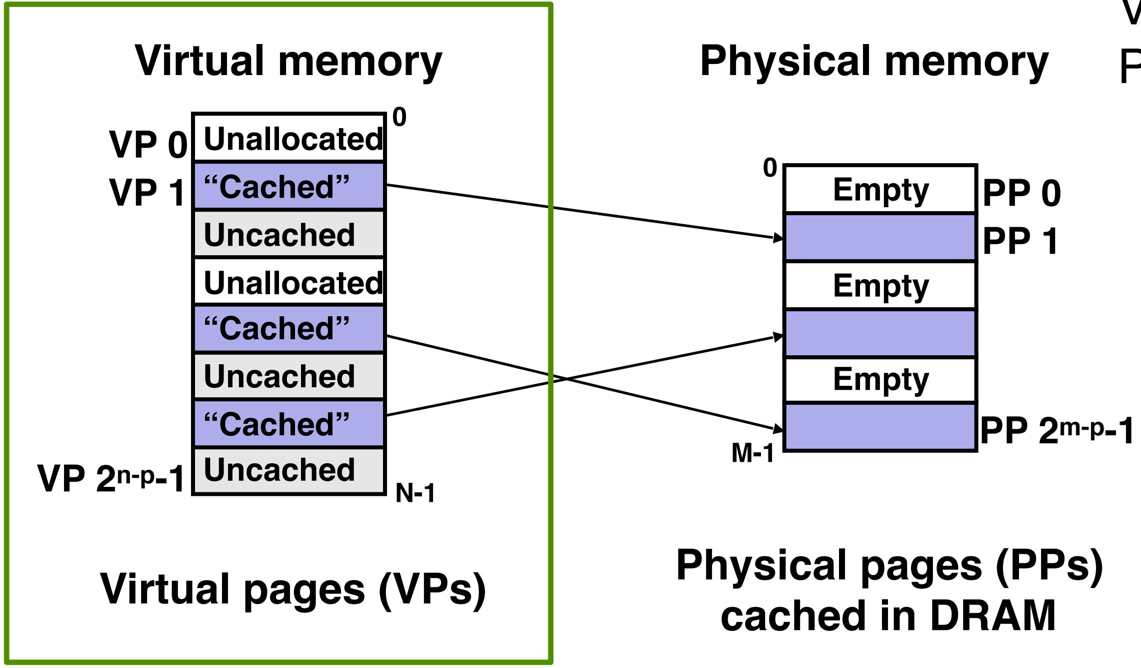
Assuming page size is 4K.
Virtual memory size is 4GB
Physical memory size is 1GB



VM Concepts

- Conceptually, *virtual memory* is an array of N contiguous *pages* (page size P = 2^p bytes)
- Physical memory is also divided into pages. Each physical page (sometimes called *frames*) has the same size as a virtual page. Physical memory has way fewer pages.
- A page can either be on the ("uncached") disk or in the physical memory ("cached").

What programmers see



Virtual memory

VP 0 **Unallocated** **0**
VP 1 **"Cached"**
Uncached
Unallocated
"Cached"
Uncached
"Cached"
VP 2^{n-p}-1 **Uncached** **N-1**

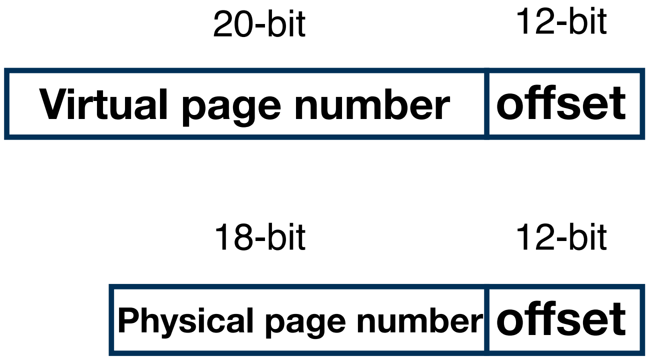
Virtual pages (VPs)

Physical memory

0 **Empty** **PP 0**
PP 1
Empty
Empty
M-1 **PP 2^{m-p}-1**

Physical pages (PPs) cached in DRAM

Assuming page size is 4K.
Virtual memory size is 4GB
Physical memory size is 1GB



20-bit 12-bit
Virtual page number | **offset**

18-bit 12-bit
Physical page number | **offset**

Analogy for Address Translation: A Secure Hotel

Analogy for Address Translation: A Secure Hotel

- Call a hotel looking for a guest; what happens?
  - Front desk routes call to room, does not give out room number
  - Guest’s name is a virtual address
  - Room number is physical address
  - Front desk is doing address translation!

Analogy for Address Translation: A Secure Hotel

- Call a hotel looking for a guest; what happens?
  - Front desk routes call to room, does not give out room number
  - Guest’s name is a virtual address
  - Room number is physical address
  - Front desk is doing address translation!

- Benefits
  - **Ease of management**: Guest could change rooms (physical address). You can still find her without knowing it
  - **Protection**: Guest could have block on calls, block on calls from specific callers (permissions)
  - **Sharing**: Multiple guests (virtual addresses) can share the same room (physical address)

Different Names in Different Places

Symbolic
address

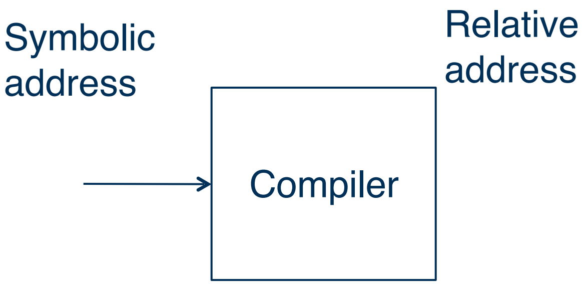
- Programmer uses text-based names (symbolic address)
  - int array[100];

Different Names in Different Places

Symbolic
address

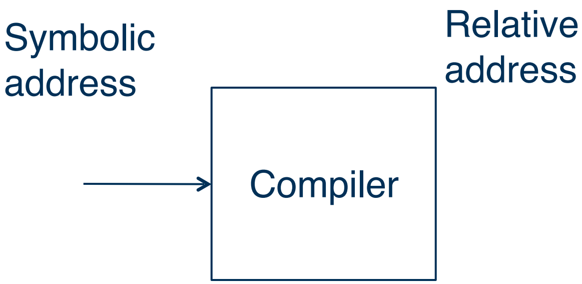
- Programmer uses text-based names (symbolic address)
    - int array[100];
- Compiler maps names to flat, uniform space
    - Starting point is relative, size specified

Different Names in Different Places



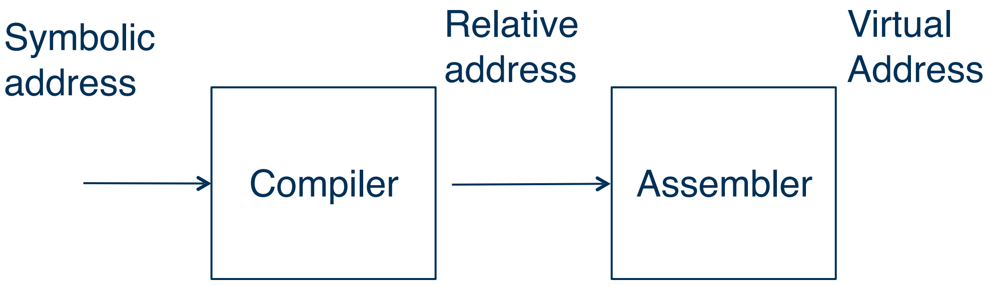
- Programmer uses text-based names (symbolic address)
    - int array[100];
- Compiler maps names to flat, uniform space
    - Starting point is relative, size specified

Different Names in Different Places



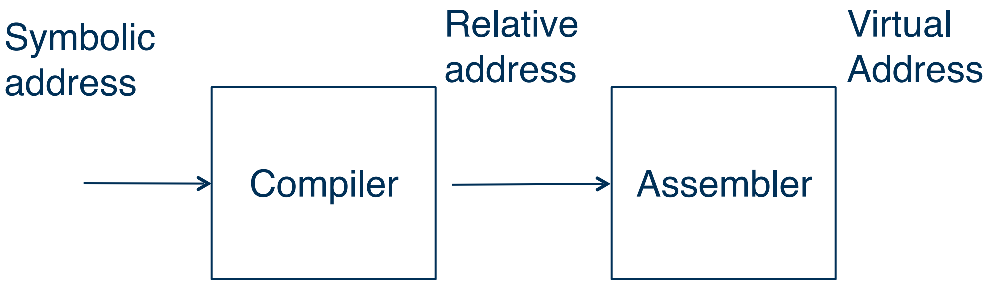
- Programmer uses text-based names (symbolic address)
  - int array[100];
- Compiler maps names to flat, uniform space
  - Starting point is relative, size specified
- Assembler maps uniform space to virtual addresses
  - Mechanical transformation (assume a start address)

Different Names in Different Places



- Programmer uses text-based names (symbolic address)
    - int array[100];
- Compiler maps names to flat, uniform space
    - Starting point is relative, size specified
- Assembler maps uniform space to virtual addresses
    - Mechanical transformation (assume a start address)

Different Names in Different Places



Symbolic address

Compiler

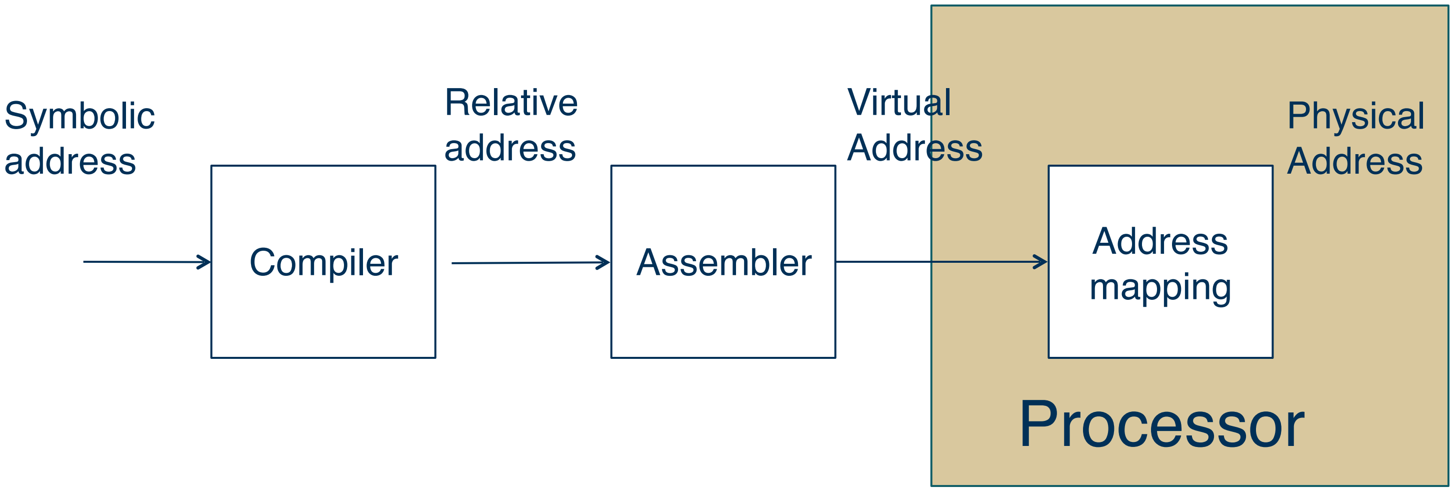
Relative address

Assembler

Virtual Address

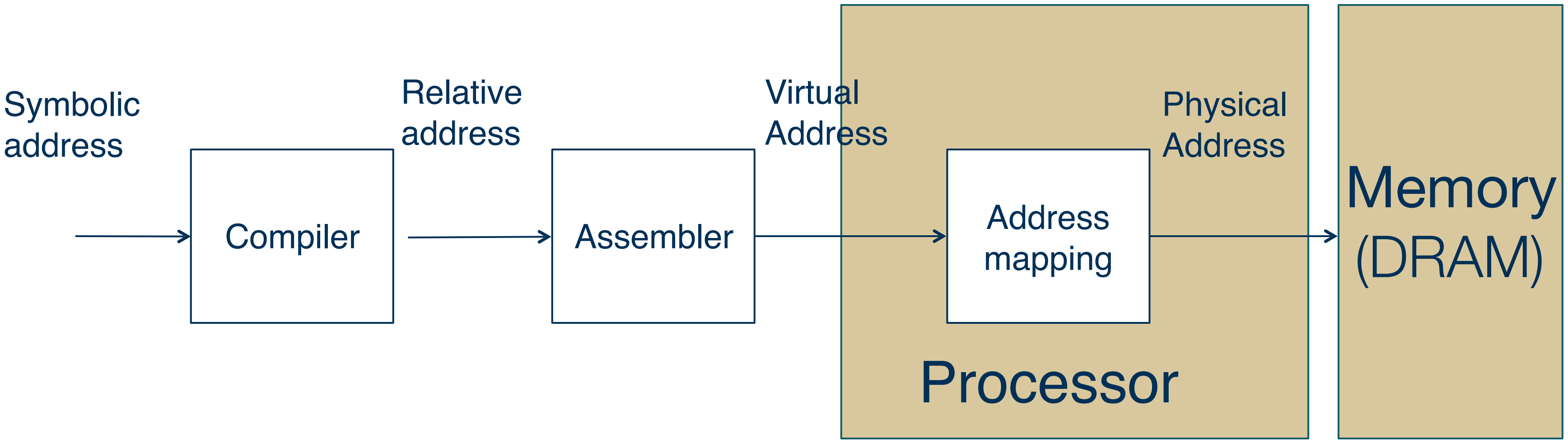
- Programmer uses text-based names (symbolic address)
  - int array[100];
- Compiler maps names to flat, uniform space
  - Starting point is relative, size specified
- Assembler maps uniform space to virtual addresses
  - Mechanical transformation (assume a start address)
- Processor instructions use virtual addresses, translates to physical addresses

Different Names in Different Places



- Programmer uses text-based names (symbolic address)
    - int array[100];
- Compiler maps names to flat, uniform space
    - Starting point is relative, size specified
- Assembler maps uniform space to virtual addresses
    - Mechanical transformation (assume a start address)
- Processor instructions use virtual addresses, translates to physical addresses

Different Names in Different Places



- Programmer uses text-based names (symbolic address)
  - int array[100];
- Compiler maps names to flat, uniform space
  - Starting point is relative, size specified
- Assembler maps uniform space to virtual addresses
  - Mechanical transformation (assume a start address)
- Processor instructions use virtual addresses, translates to physical addresses

Enabling Data Structure: Page Table

- How do we track which virtual pages are mapped to physical pages, and where they are mapped?

Enabling Data Structure: Page Table

- How do we track which virtual pages are mapped to physical pages, and where they are mapped?
- Use a table to track this. The table is called page table, in which each virtual page has an entry

Enabling Data Structure: Page Table

- How do we track which virtual pages are mapped to physical pages, and where they are mapped?
- Use a table to track this. The table is called page table, in which each virtual page has an entry
- Each entry records whether the particular virtual page is mapped to the physical memory

Enabling Data Structure: Page Table

- How do we track which virtual pages are mapped to physical pages, and where they are mapped?
- Use a table to track this. The table is called page table, in which each virtual page has an entry
- Each entry records whether the particular virtual page is mapped to the physical memory
  - If mapped, where in the physical memory it is mapped to?

Enabling Data Structure: Page Table

- How do we track which virtual pages are mapped to physical pages, and where they are mapped?
- Use a table to track this. The table is called page table, in which each virtual page has an entry
- Each entry records whether the particular virtual page is mapped to the physical memory
  - If mapped, where in the physical memory it is mapped to?
  - If not mapped, where on the disk is the virtual page?

Enabling Data Structure: Page Table

- How do we track which virtual pages are mapped to physical pages, and where they are mapped?
- Use a table to track this. The table is called page table, in which each virtual page has an entry
- Each entry records whether the particular virtual page is mapped to the physical memory
  - If mapped, where in the physical memory it is mapped to?
  - If not mapped, where on the disk is the virtual page?
- Do you need a page table for each process?

Enabling Data Structure: Page Table

- How do we track which virtual pages are mapped to physical pages, and where they are mapped?
- Use a table to track this. The table is called page table, in which each virtual page has an entry
- Each entry records whether the particular virtual page is mapped to the physical memory
  - If mapped, where in the physical memory it is mapped to?
  - If not mapped, where on the disk is the virtual page?
- Do you need a page table for each process?
  - Per-process data structure; managed by the OS kernel

Enabling Data Structure: Page Table

- A page table is an array of page table entries (PTEs) that maps every virtual page to its physical page.

Enabling Data Structure: Page Table

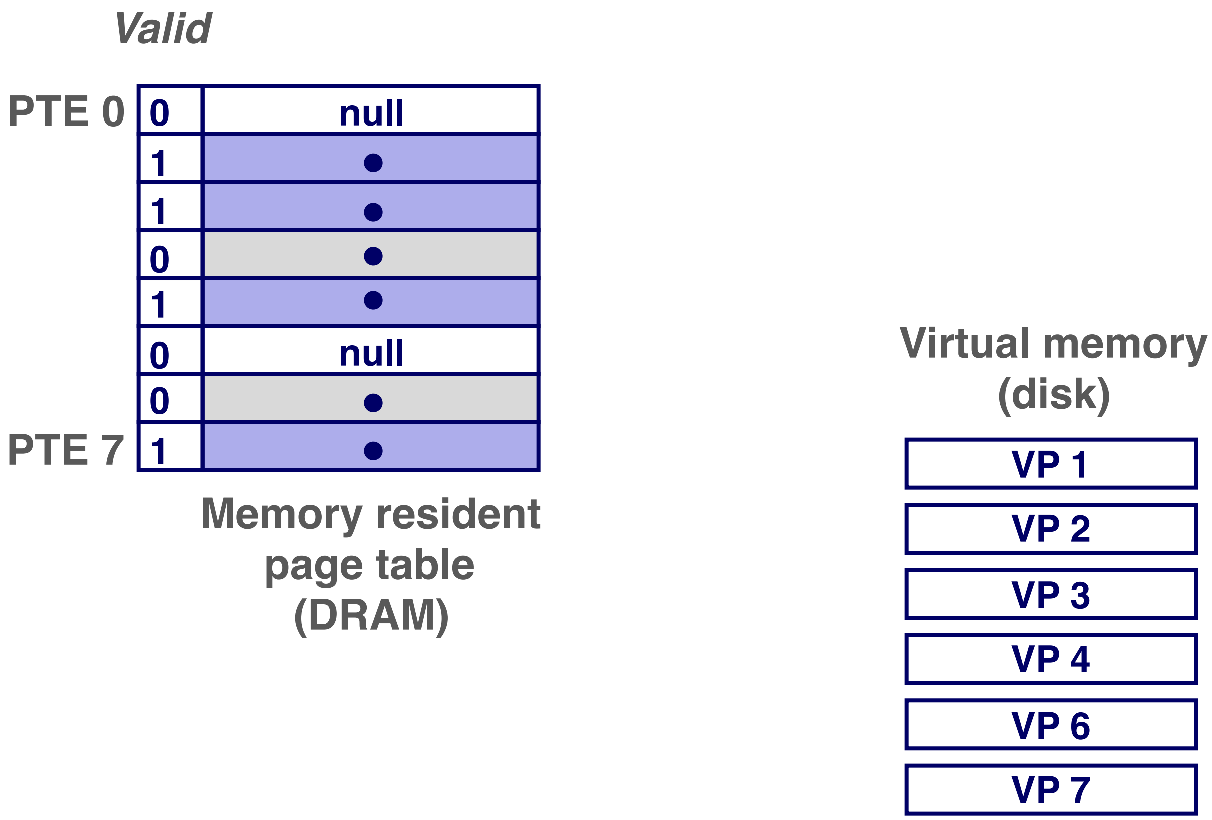
- A page table is an array of page table entries (PTEs) that maps every virtual page to its physical page.

Virtual memory (disk)

VP 1
VP 2
VP 3
VP 4
VP 6
VP 7

Enabling Data Structure: Page Table

- A page table is an array of page table entries (PTEs) that maps every virtual page to its physical page.

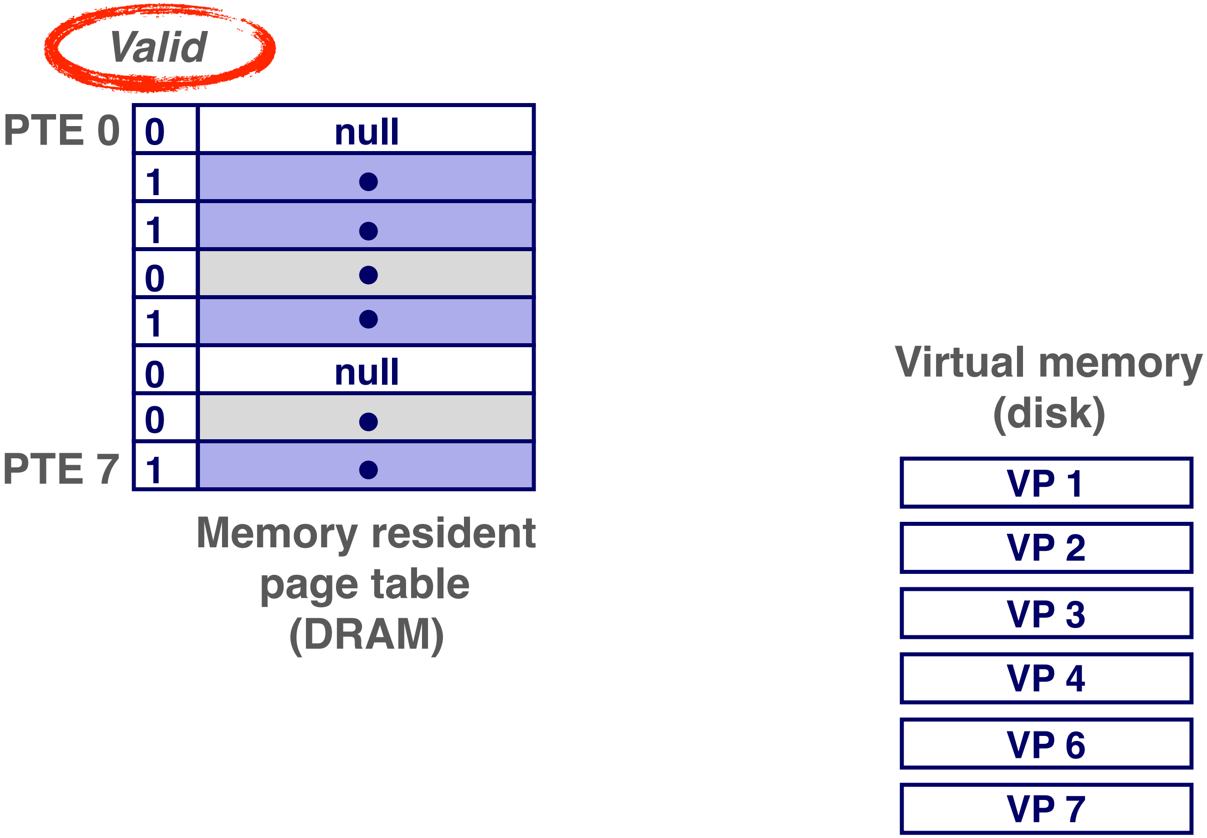


	Valid	
PTE 0	0	null
	1	●
	1	●
	0	●
	1	●
	0	null
	0	●
PTE 7	1	●

Memory resident page table (DRAM)

Enabling Data Structure: Page Table

- A page table is an array of page table entries (PTEs) that maps every virtual page to its physical page.



	Valid	
PTE 0	0	null
	1	●
	1	●
	0	●
	1	●
	0	null
	0	●
PTE 7	1	●

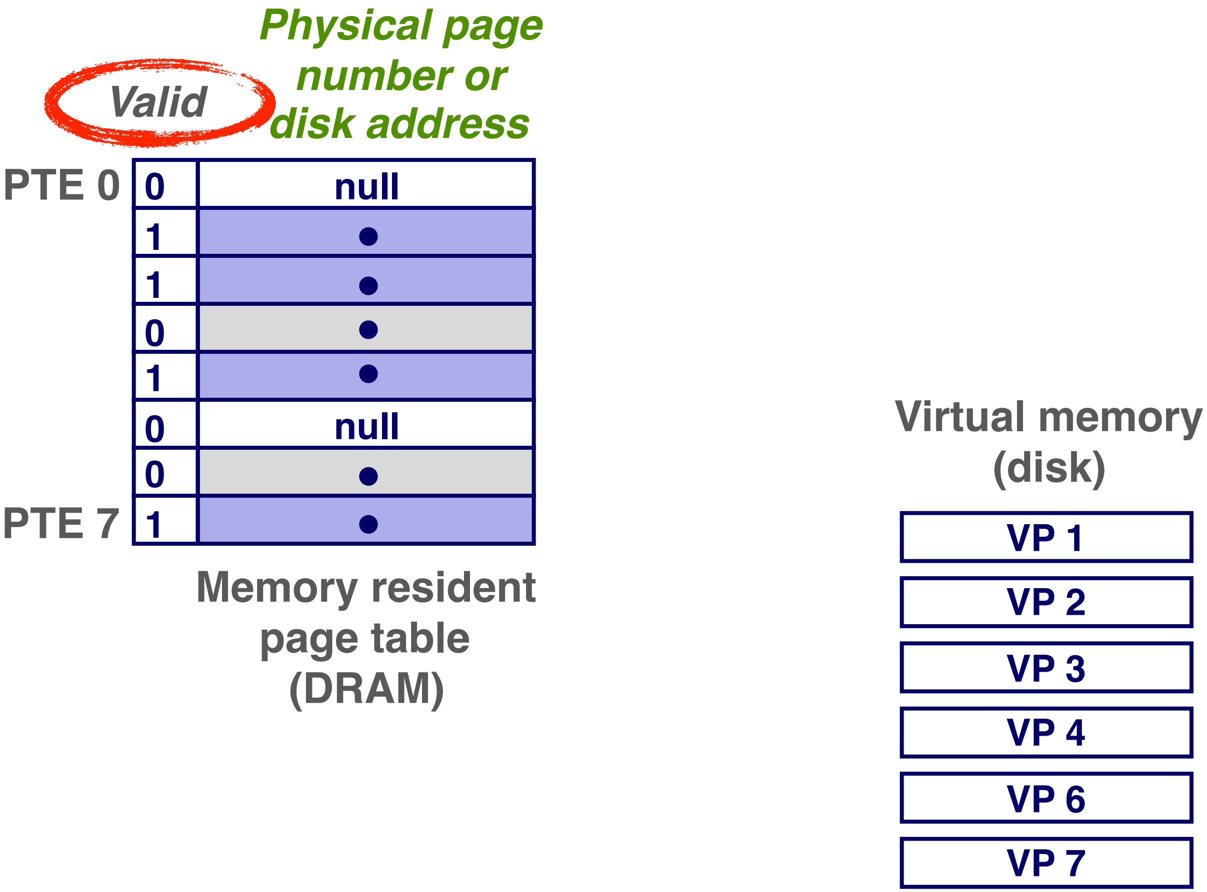
Memory resident
page table
(DRAM)

Virtual memory
(disk)

VP 1
VP 2
VP 3
VP 4
VP 6
VP 7

Enabling Data Structure: Page Table

- A page table is an array of page table entries (PTEs) that maps every virtual page to its physical page.



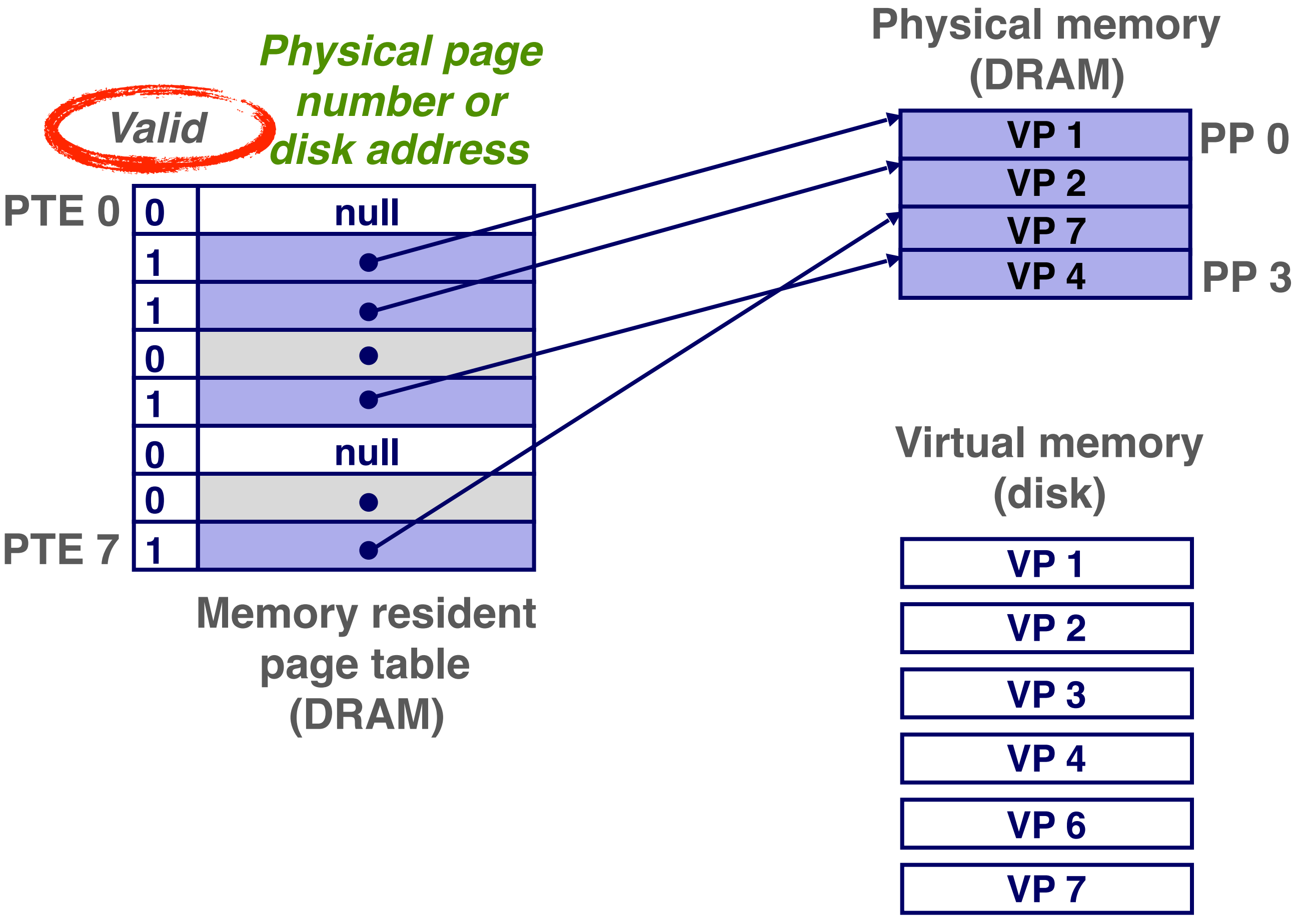
	Valid	*Physical page number or disk address*
PTE 0	0	null
	1	●
	1	●
	0	●
	1	●
	0	null
	0	●
PTE 7	1	●

Memory resident page table (DRAM)

Virtual memory (disk)

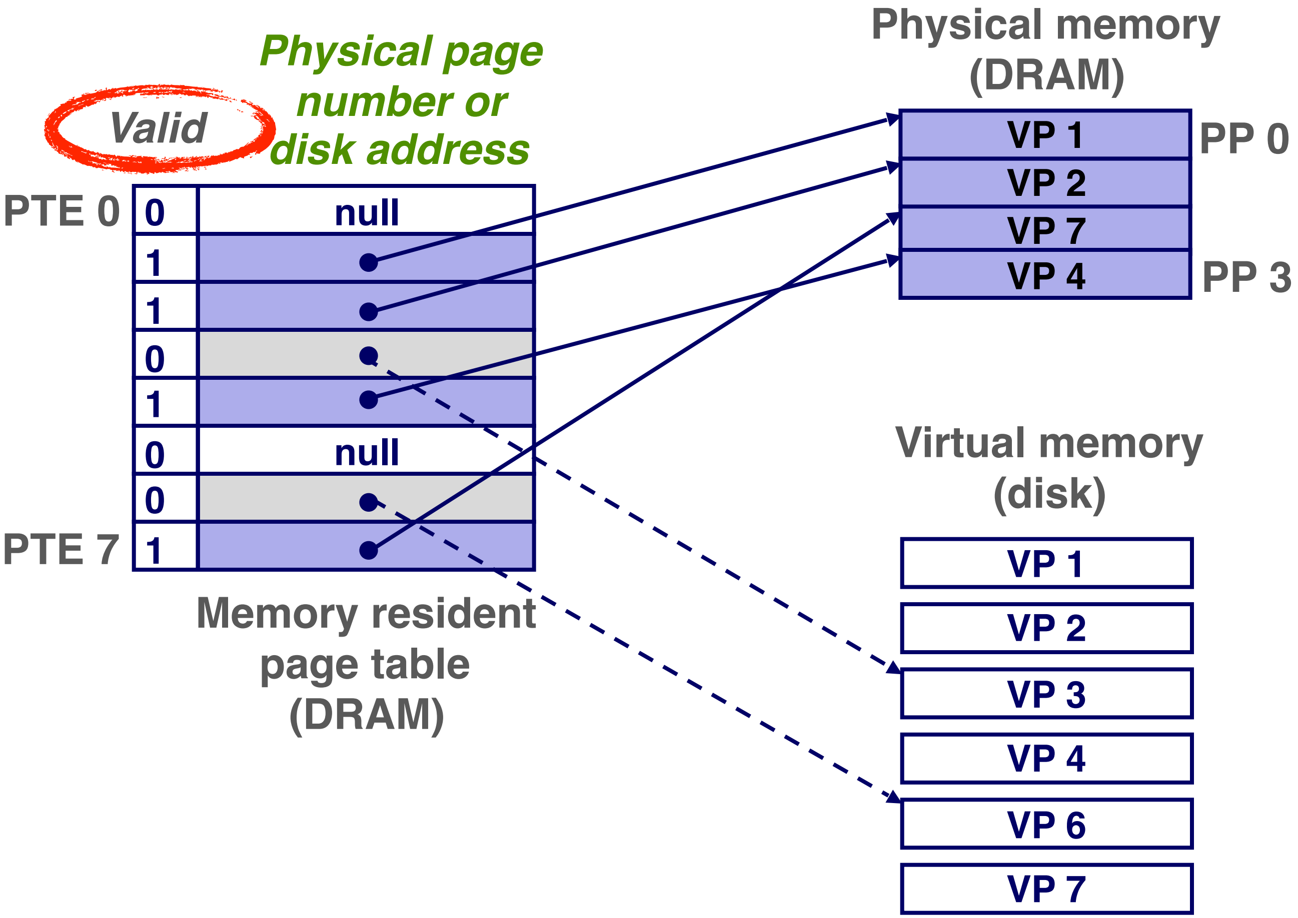
Enabling Data Structure: Page Table

- A page table is an array of page table entries (PTEs) that maps every virtual page to its physical page.



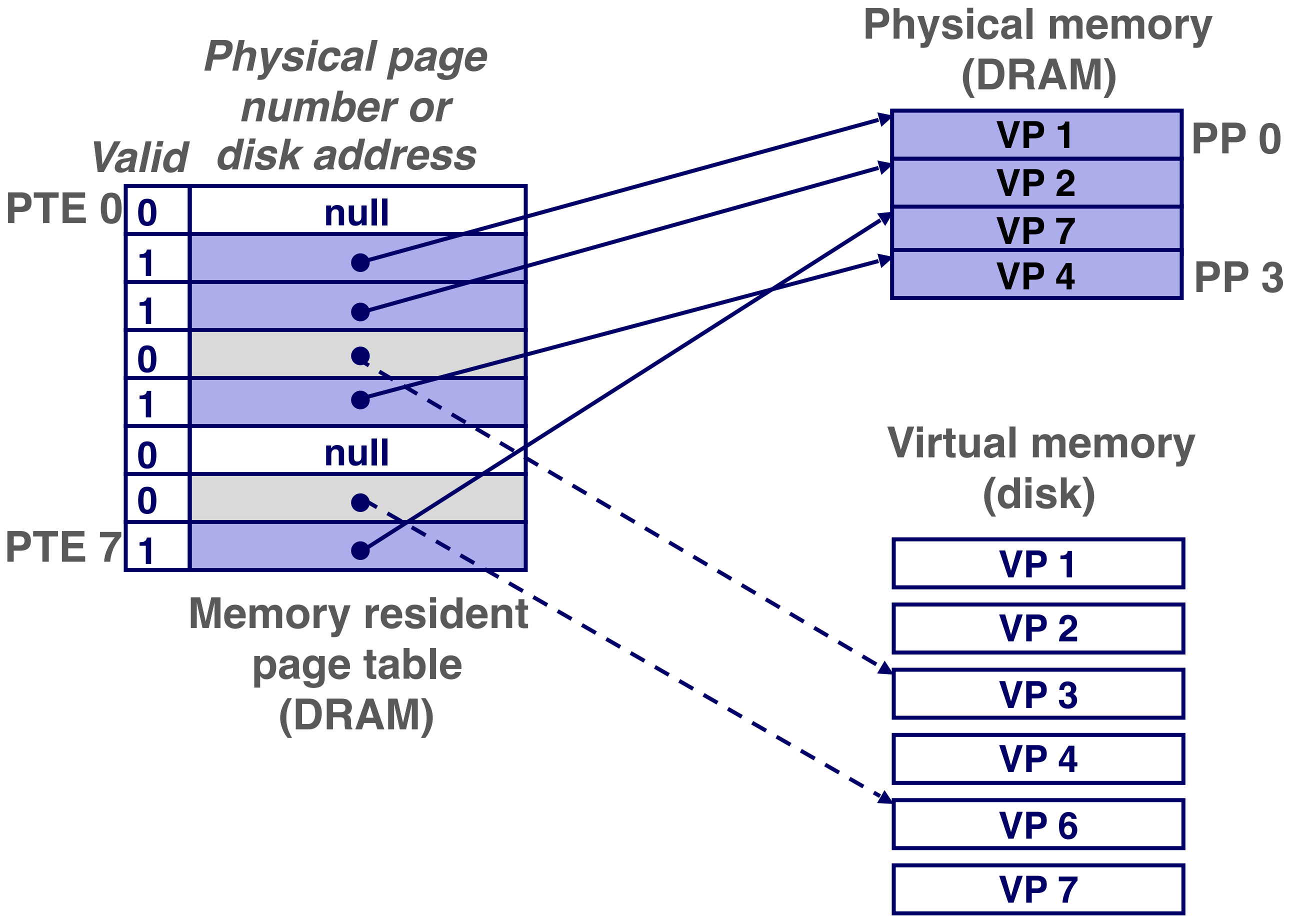
Enabling Data Structure: Page Table

- A page table is an array of page table entries (PTEs) that maps every virtual page to its physical page.



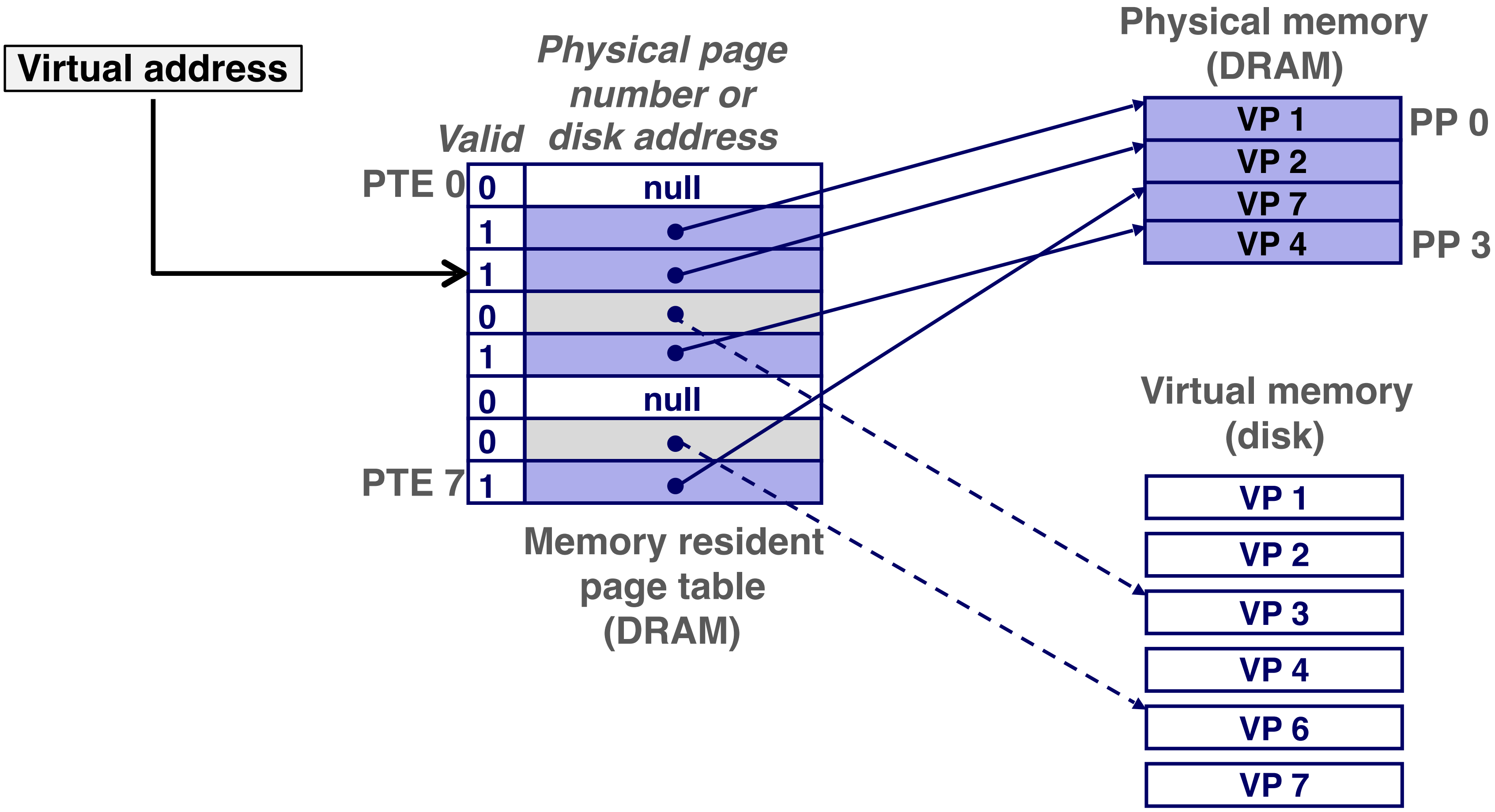
Page Hit

- *Page hit:* reference to VM word that is in physical memory



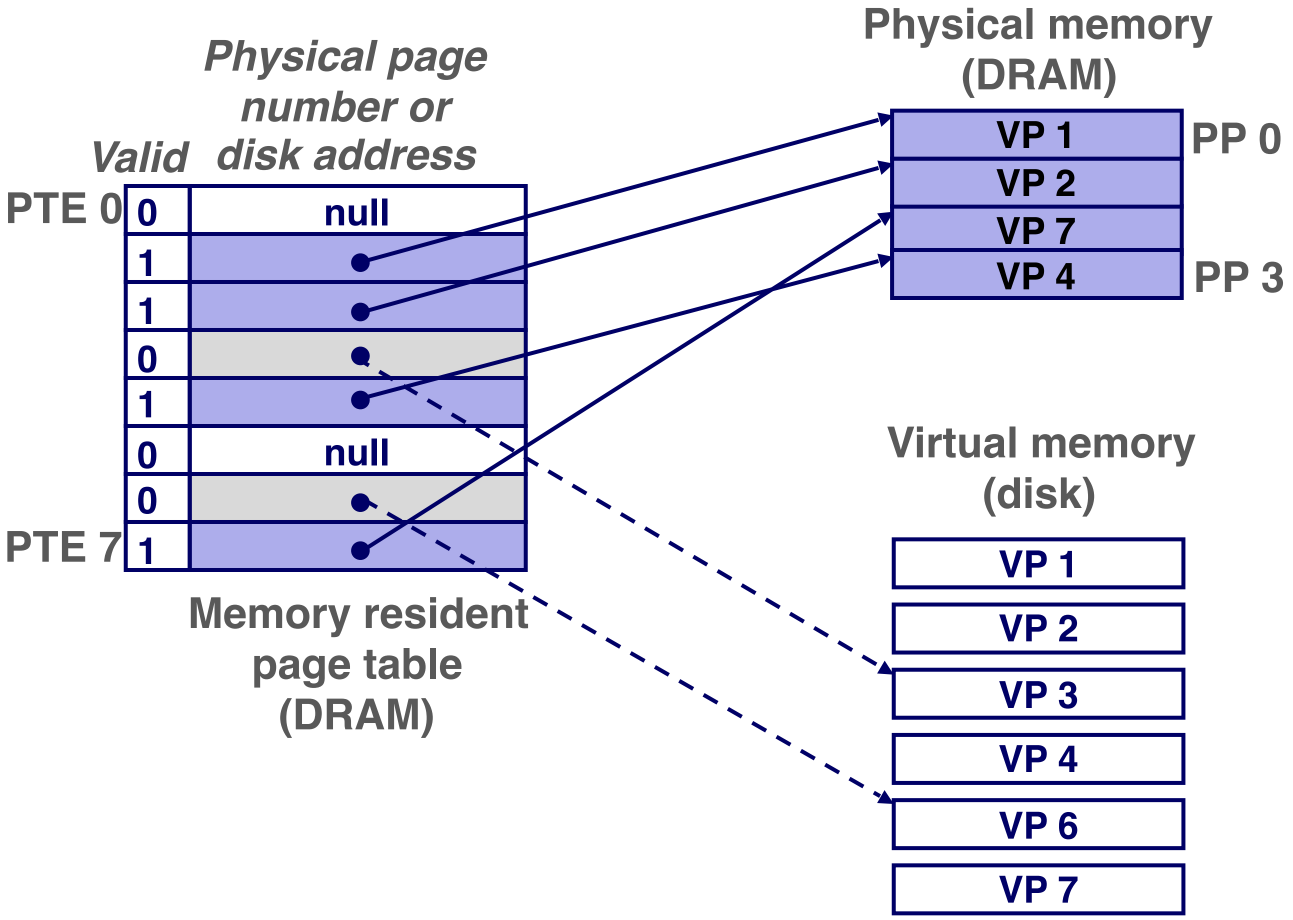
Page Hit

- *Page hit:* reference to VM word that is in physical memory



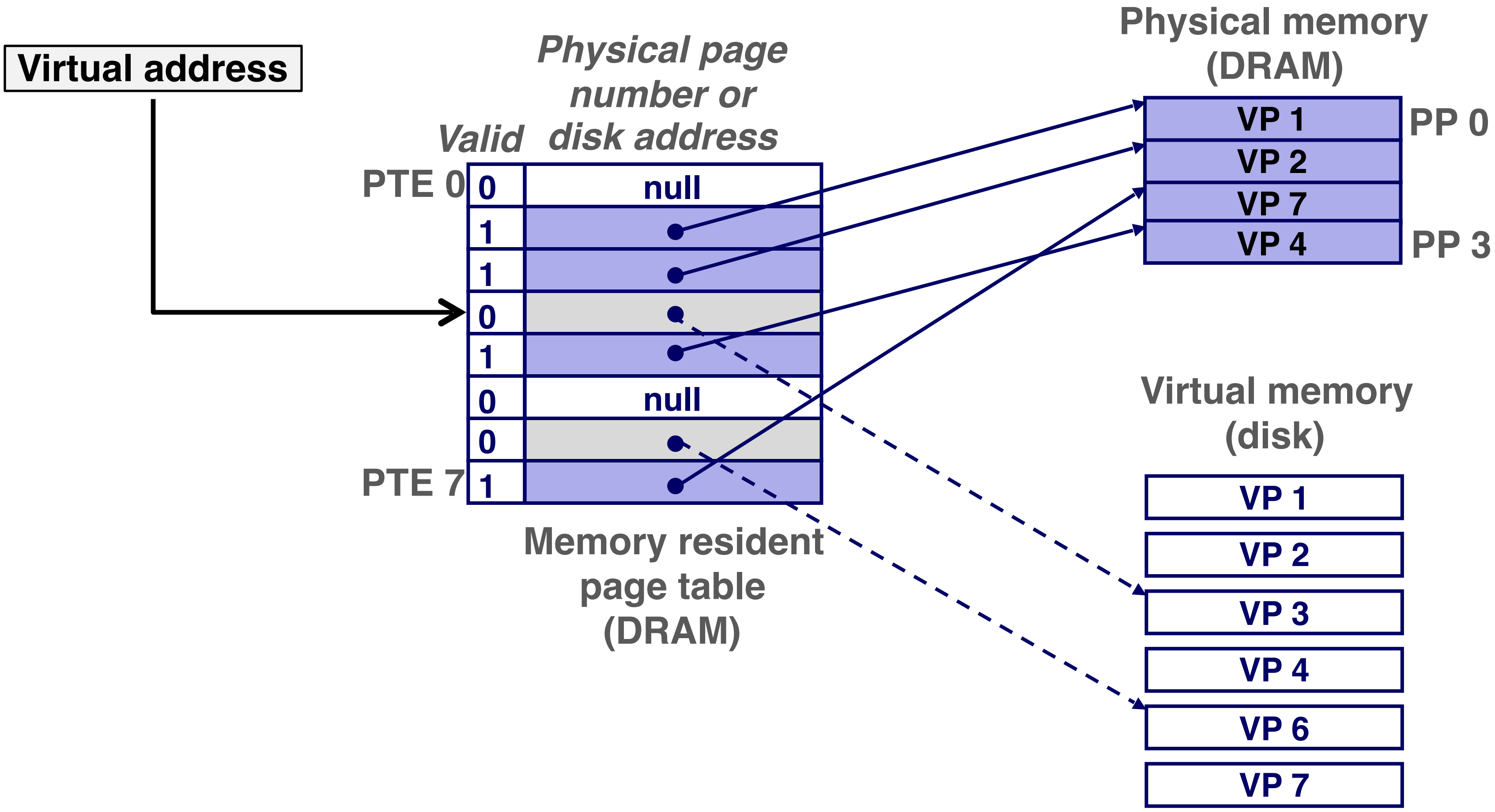
Page Fault

- *Page fault:* reference to VM word that is not in physical memory



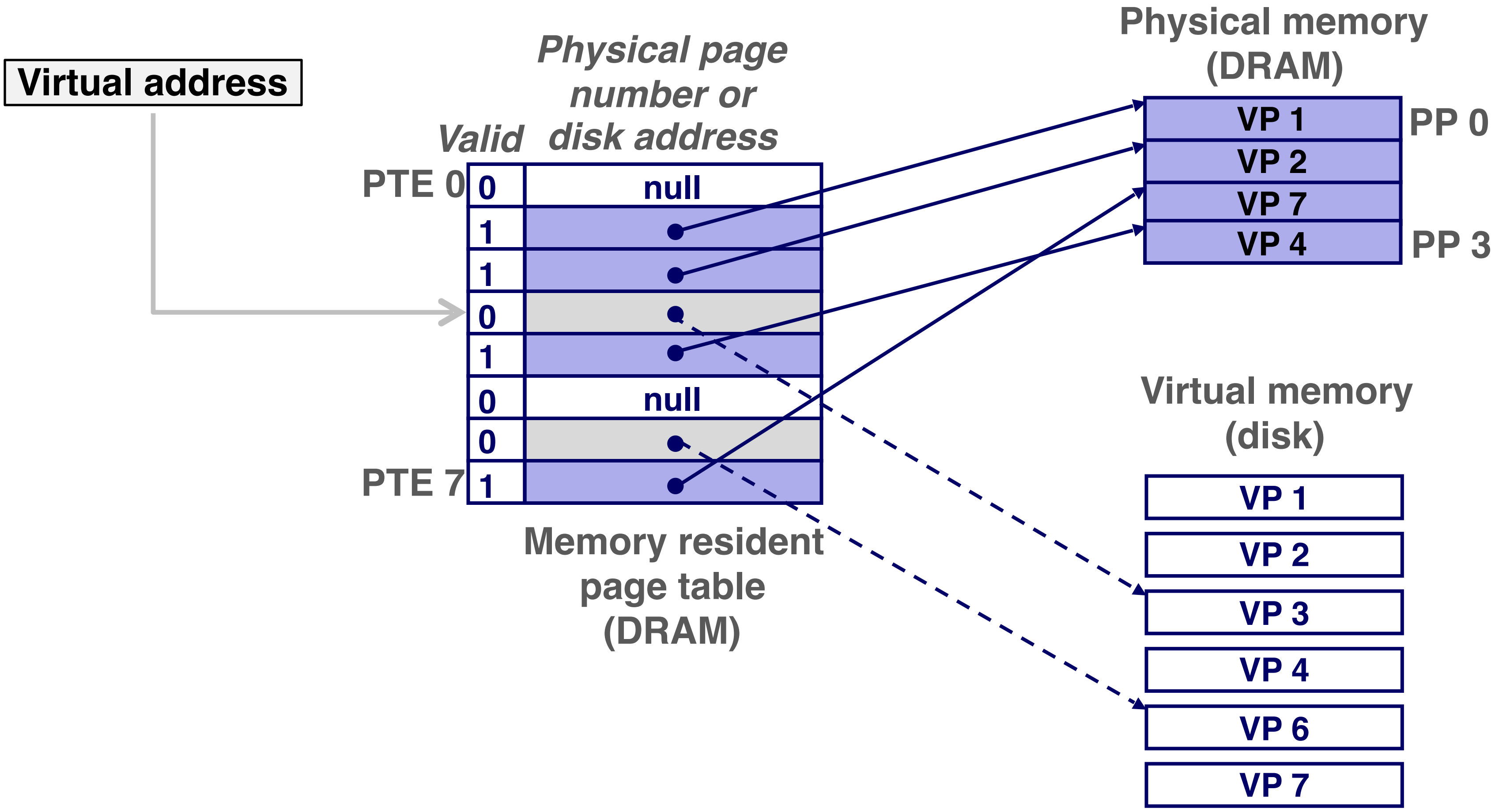
Page Fault

- *Page fault:* reference to VM word that is not in physical memory



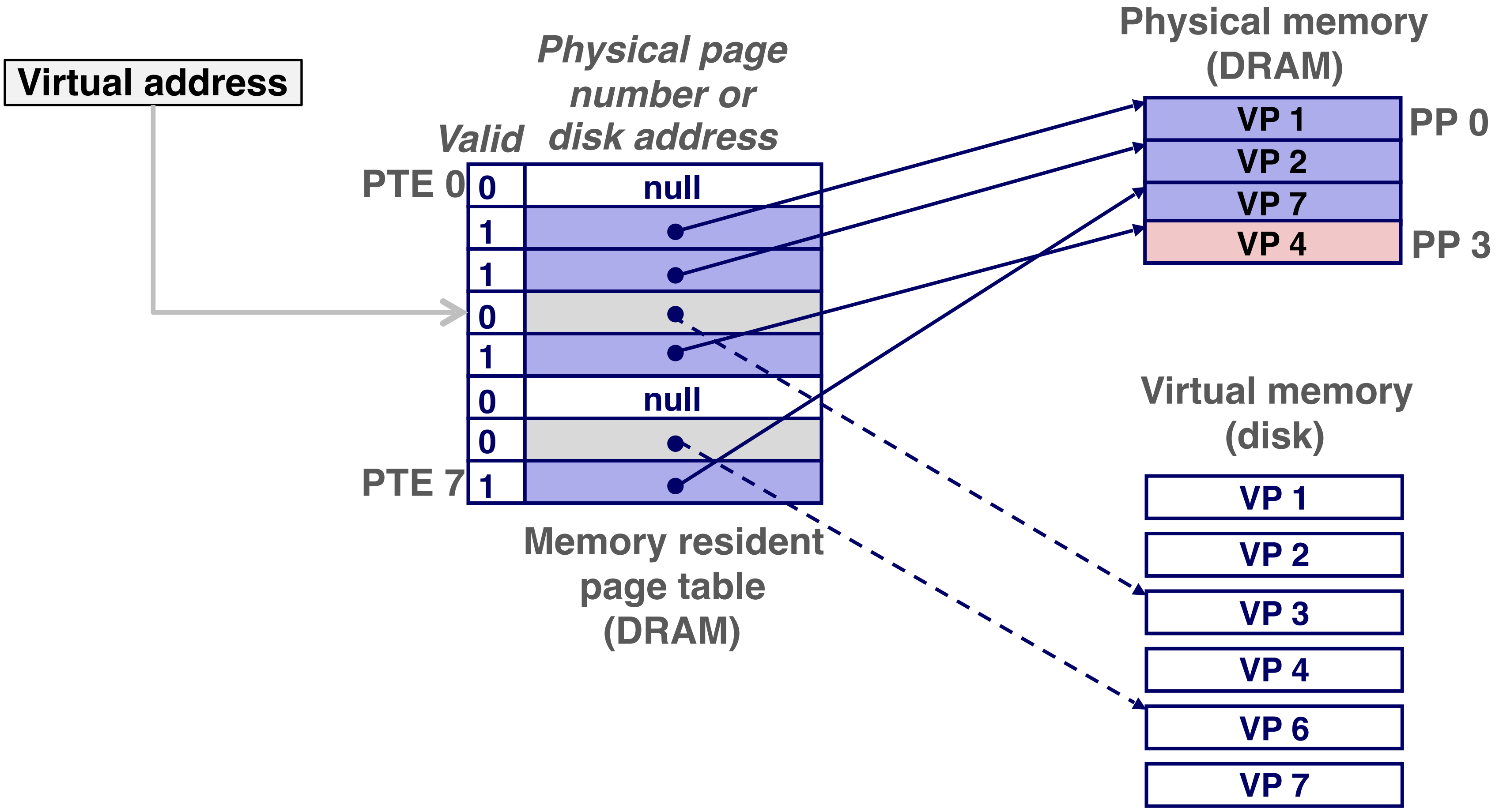
Handling Page Fault

- Page miss causes page fault (an exception)



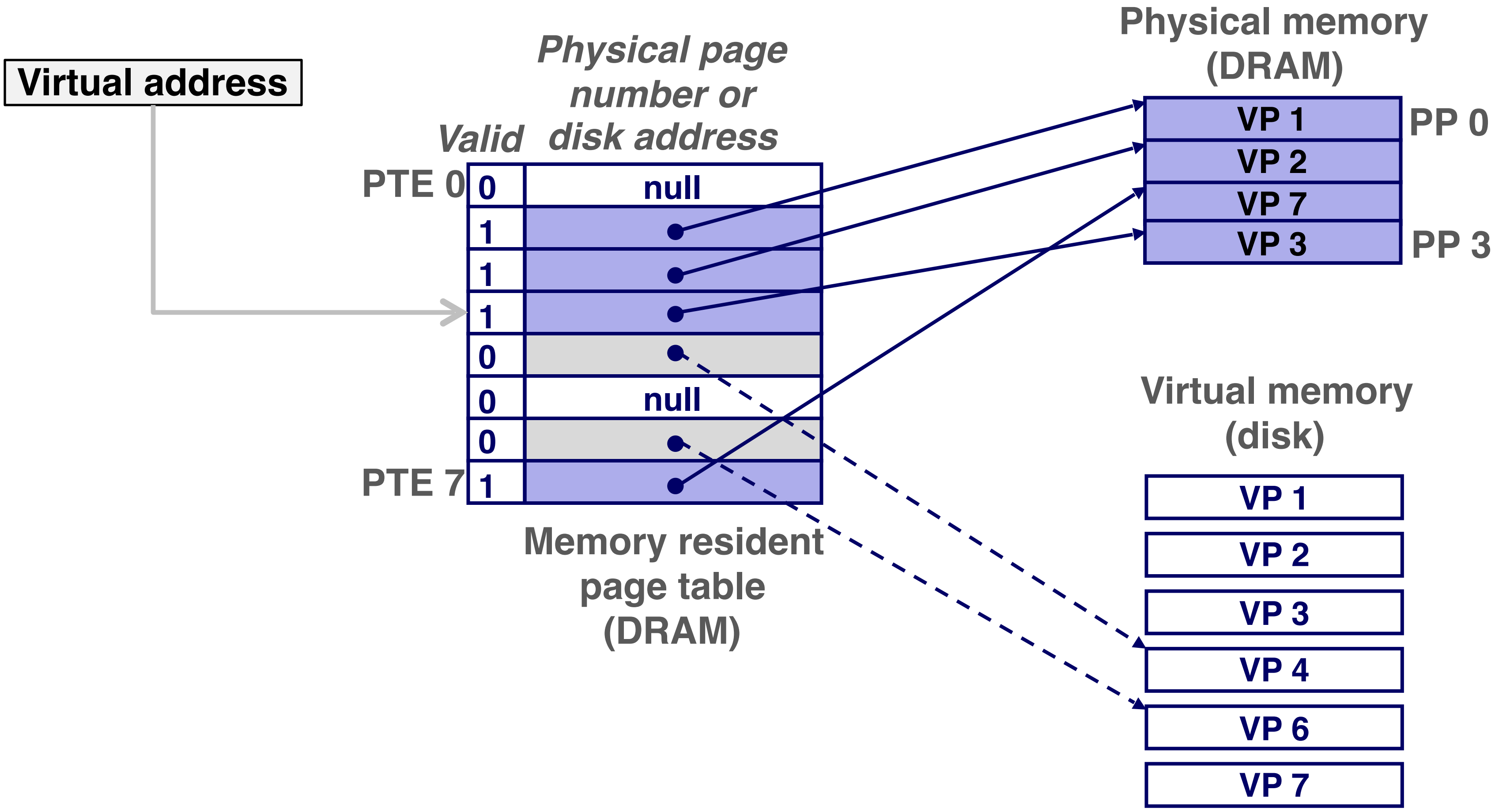
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)



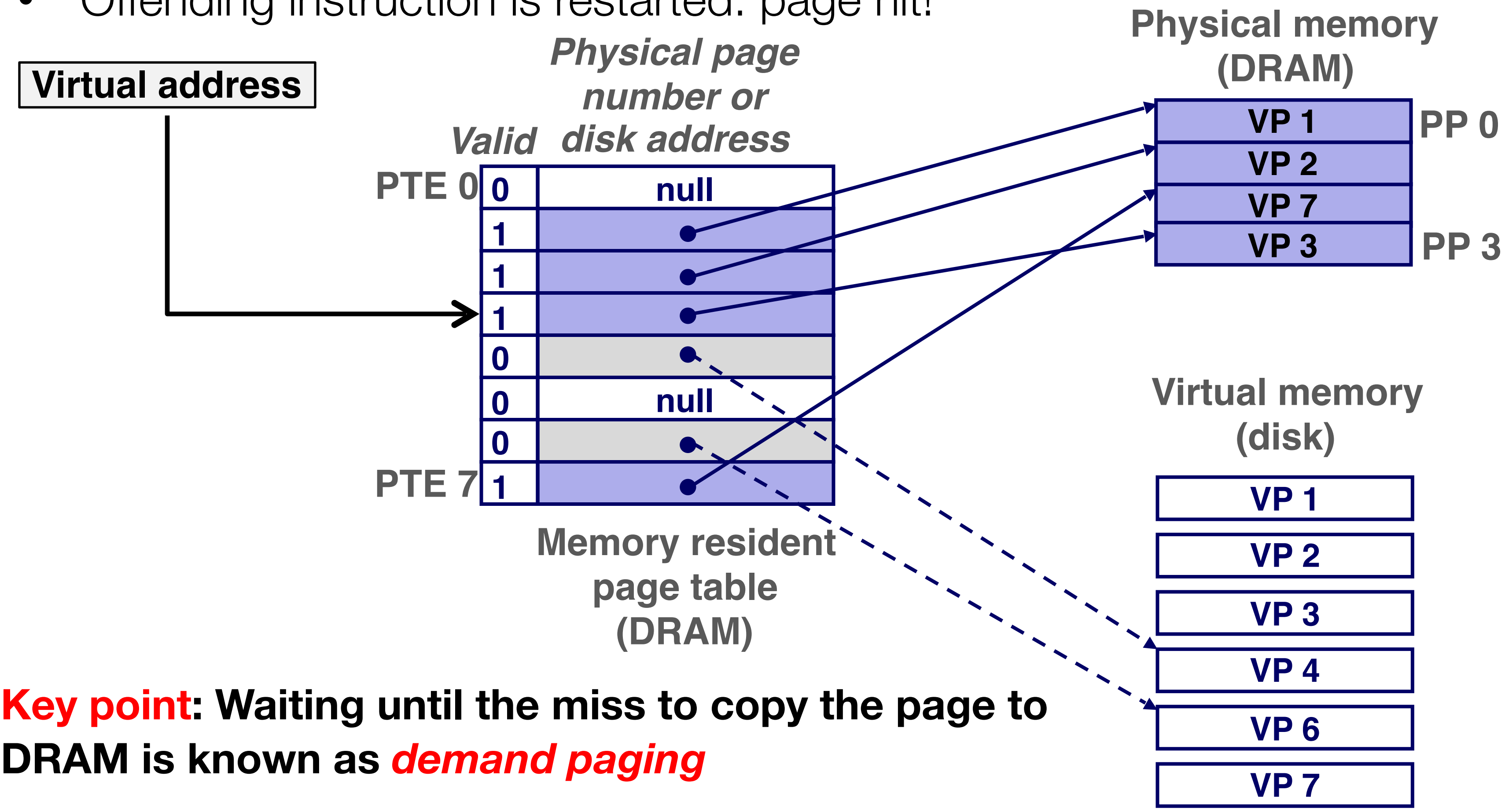
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)



Handling Page Fault

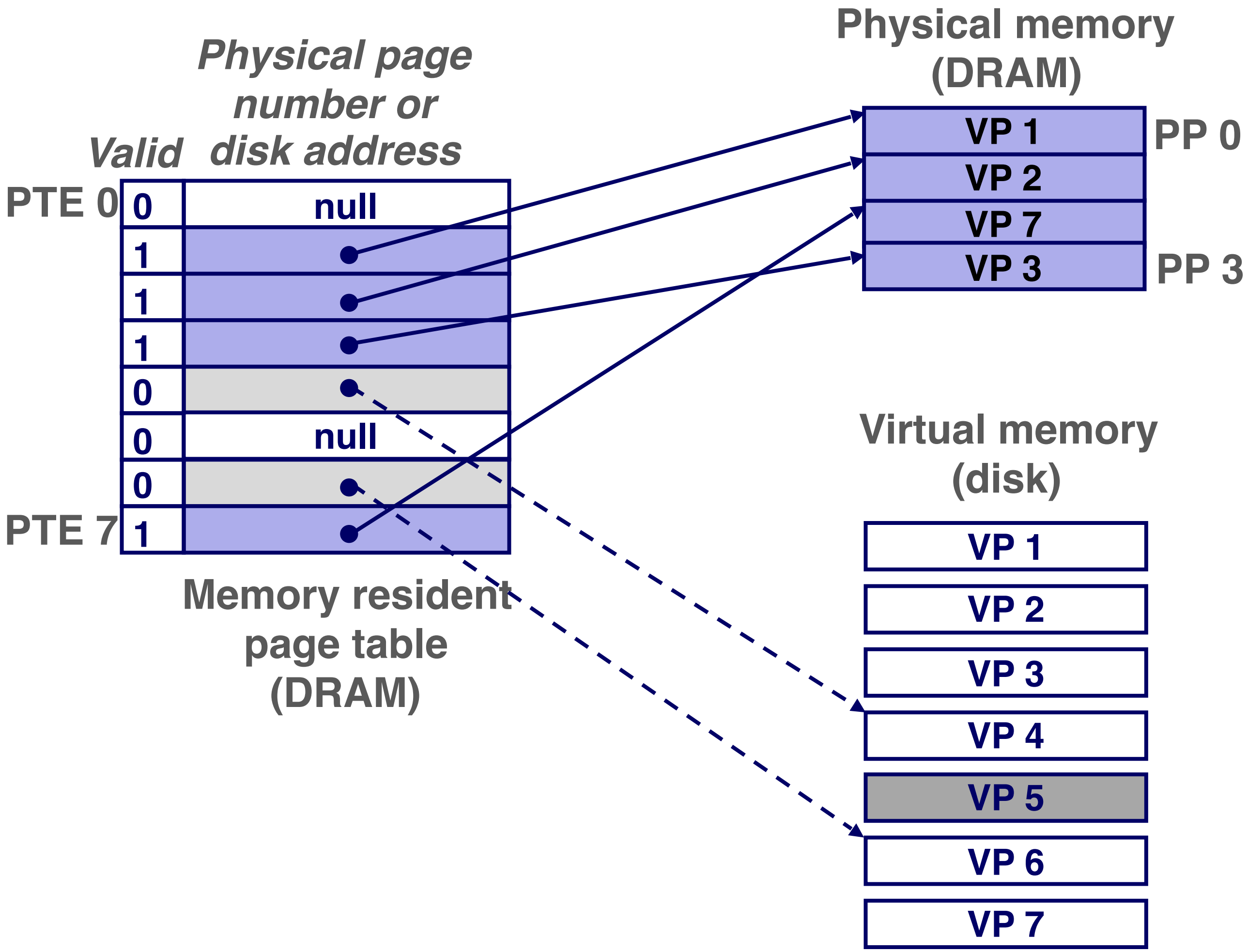
- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
- Offending instruction is restarted: page hit!



Key point: **Waiting until the miss to copy the page to DRAM is known as *demand paging***

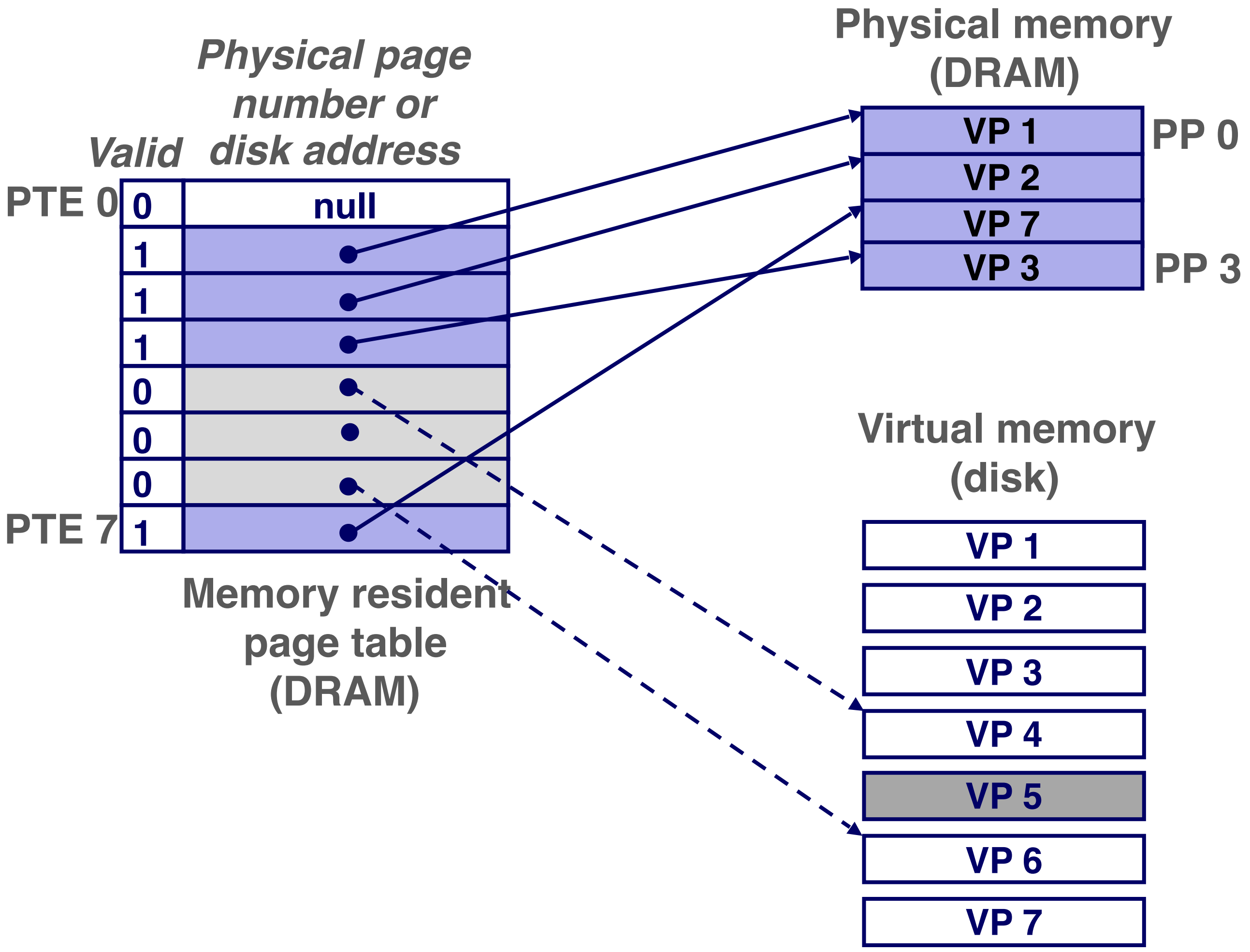
Allocating Pages

- Allocating a new page (VP 5) of virtual memory.



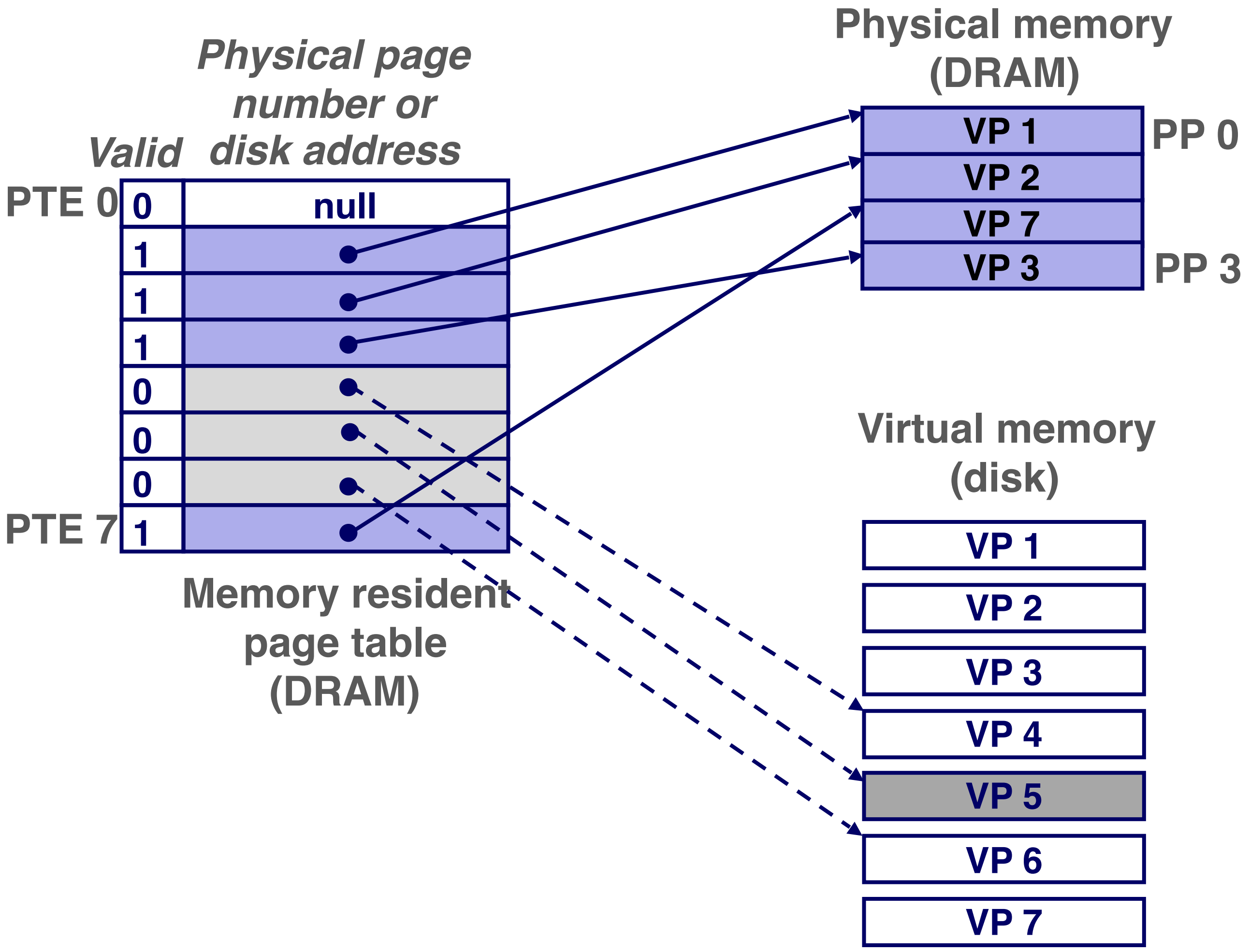
Allocating Pages

- Allocating a new page (VP 5) of virtual memory.



Allocating Pages

- Allocating a new page (VP 5) of virtual memory.



Virtual Memory Exploits Locality (Again!)

- Virtual memory seems terribly inefficient, but it works because of locality.

- At any point in time, programs tend to access a set of active virtual pages called the *working set*
  - Programs with better temporal locality will have smaller working sets

- If (working set size < main memory size)
  - Good performance for one process after initial misses

- If (SUM(working set sizes) > main memory size)
  - *Thrashing:* Performance meltdown where pages are swapped (copied) in and out continuously

Where Does Page Table Live?

Where Does Page Table Live?

- It needs to be at a specific location where we can find it
  - Some special SRAM?
  - In main memory?
  - On disk?

Where Does Page Table Live?

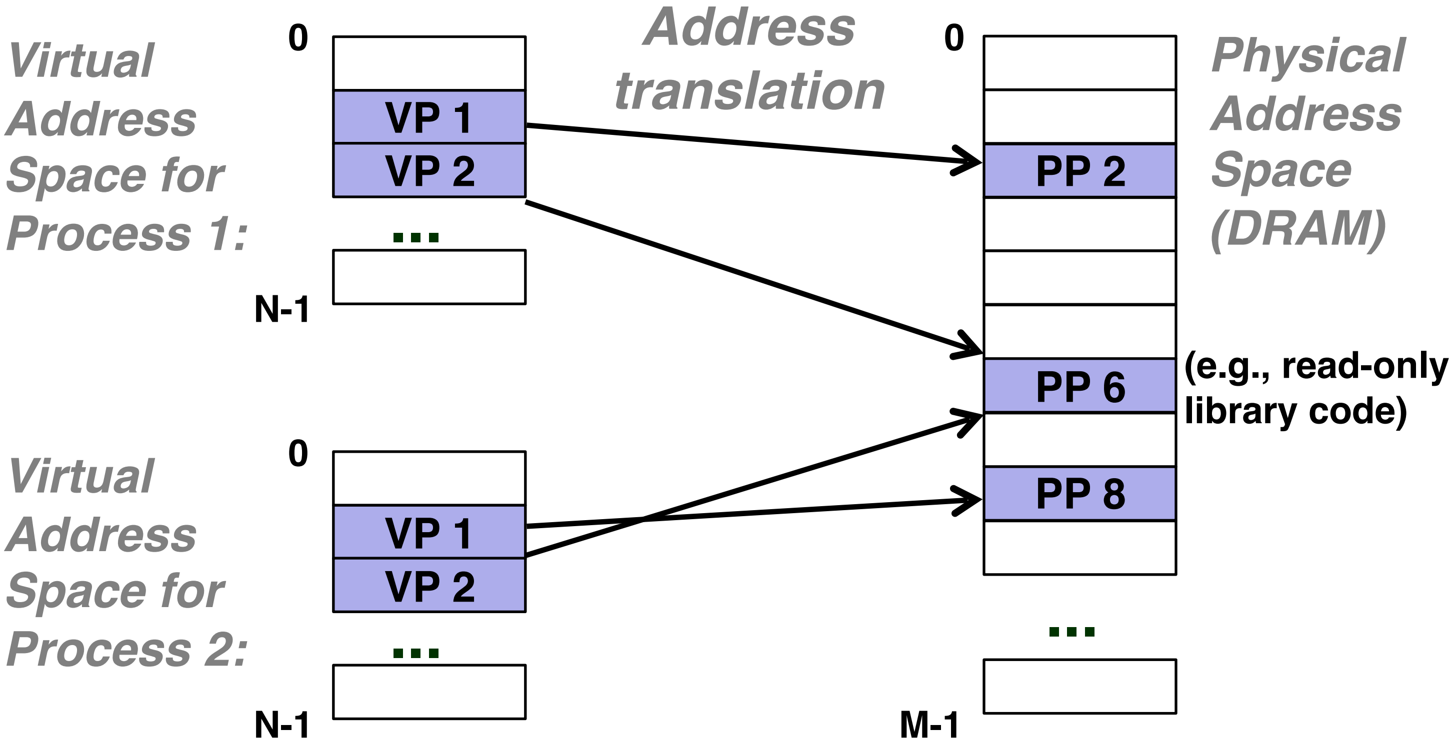
- It needs to be at a specific location where we can find it
  - Some special SRAM?
  - In main memory?
  - On disk?
- Assume 4KB page, 4GB main memory, each PTE is 8 Bytes
  - 1M PTEs in a page table
  - 8MB total size per page table
  - Too big for on-chip SRAM
  - Too slow to access in disk
  - Put the page table in DRAM, with its start address stored in a special register (Page Table Base Register)

Today

- Virtual memory (VM) illustration
- VM basic concepts and operation
- Other critical benefits of VM
- Address translation

VM as a Tool for Memory Management

- Key idea: each process has its own virtual address space
  - It can view memory as a simple linear array
  - Mapping scatters addresses through physical memory
    - Well-chosen mappings can improve locality



Virtual Memory Enables Isolations

- If all processes use physical address, it would be easy for one program to modify the data of another program. This is obviously a huge security and privacy issue.

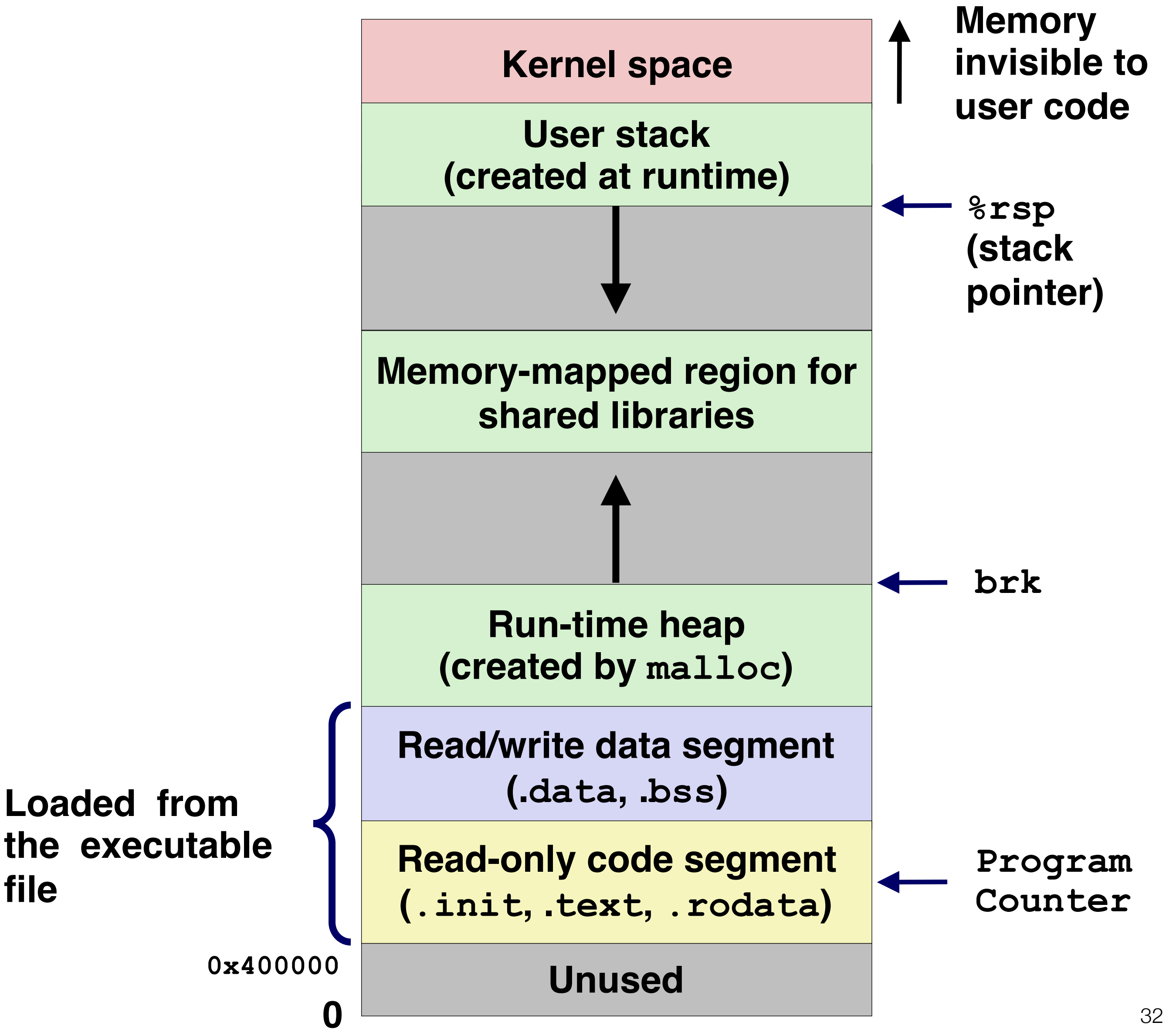
Virtual Memory Enables Isolations

- If all processes use physical address, it would be easy for one program to modify the data of another program. This is obviously a huge security and privacy issue.
- Early days (e.g., EDSAC in 50’s), ISA use physical address. To address the security issue, a program is loaded to a different address in memory every time it runs.
  - not ideal: address in programs depend on where the program is loaded in memory

Virtual Memory Enables Isolations

- If all processes use physical address, it would be easy for one program to modify the data of another program. This is obviously a huge security and privacy issue.
- Early days (e.g., EDSAC in 50’s), ISA use physical address. To address the security issue, a program is loaded to a different address in memory every time it runs.
  - not ideal: address in programs depend on where the program is loaded in memory
- With virtual memory, addresses used by program are not the same as what the processor uses to actually access memory. This naturally isolates/protect programs.

Simplifying Linking and Loading



Kernel space

User stack
(created at runtime)

Memory-mapped region for
shared libraries

Run-time heap
(created by `malloc`)

Read/write data segment
(`.data`, `.bss`)

Read-only code segment
(`.init`, `.text`, `.rodata`)

Unused

Memory invisible to user code

`%rsp` (stack pointer)

`brk`

`Program Counter`

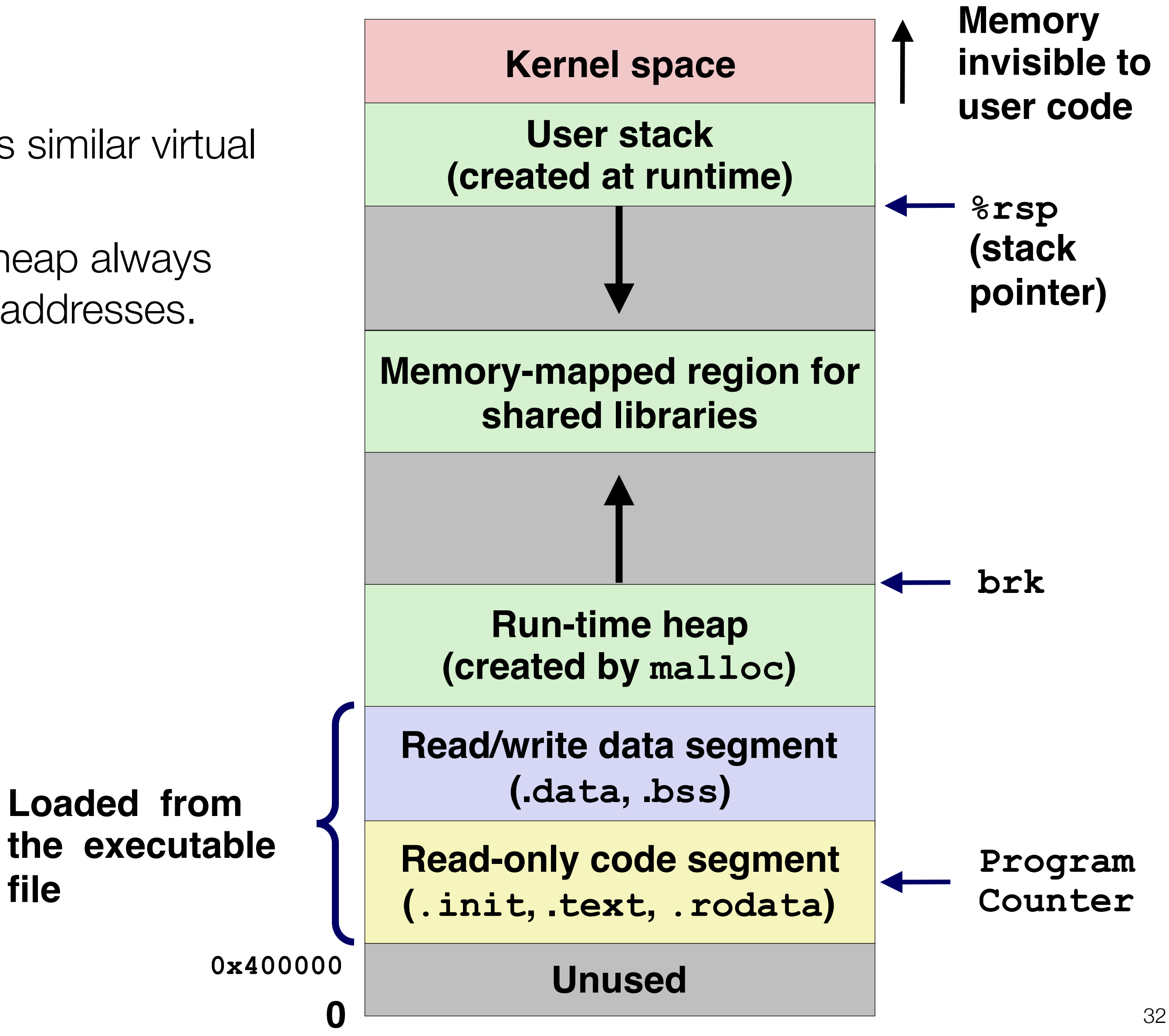
Loaded from the executable file

`0x400000`

0

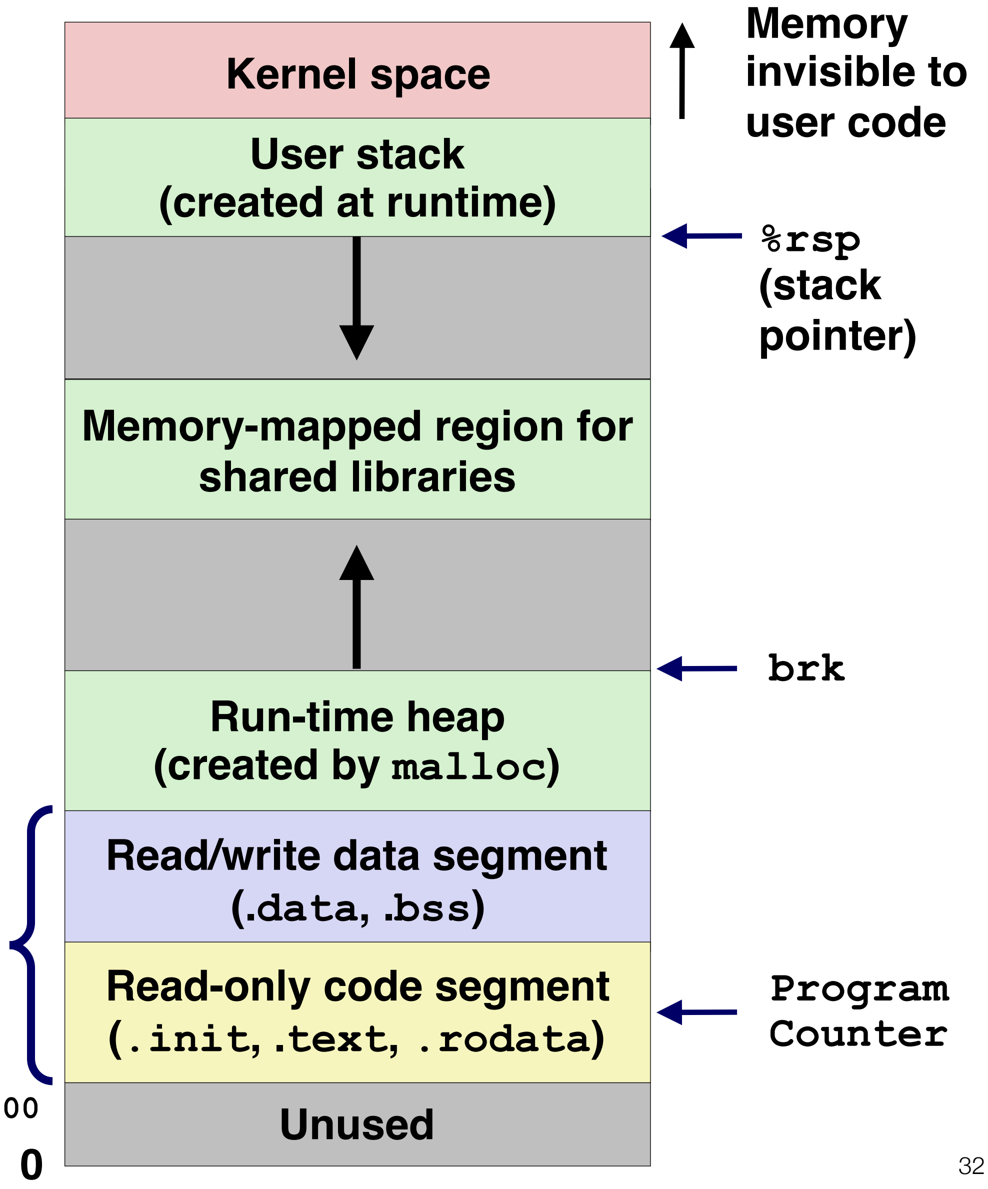
Simplifying Linking and Loading

- Linking
  - Each program has similar virtual address space
  - Code, data, and heap always start at the same addresses.



Simplifying Linking and Loading

- Linking
  - Each program has similar virtual address space
  - Code, data, and heap always start at the same addresses.
- Loading
  - `execve` allocates virtual pages for `.text` and `.data` sections & creates PTEs marked as invalid
  - The `.text` and `.data` sections are copied, page by page, on demand by the VM system



Kernel space

User stack
(created at runtime)

Memory-mapped region for
shared libraries

Run-time heap
(created by `malloc`)

Read/write data segment
(`.data`, `.bss`)

Read-only code segment
(`.init`, `.text`, `.rodata`)

Unused

Memory invisible to user code

`%rsp` (stack pointer)

`brk`

`Program Counter`

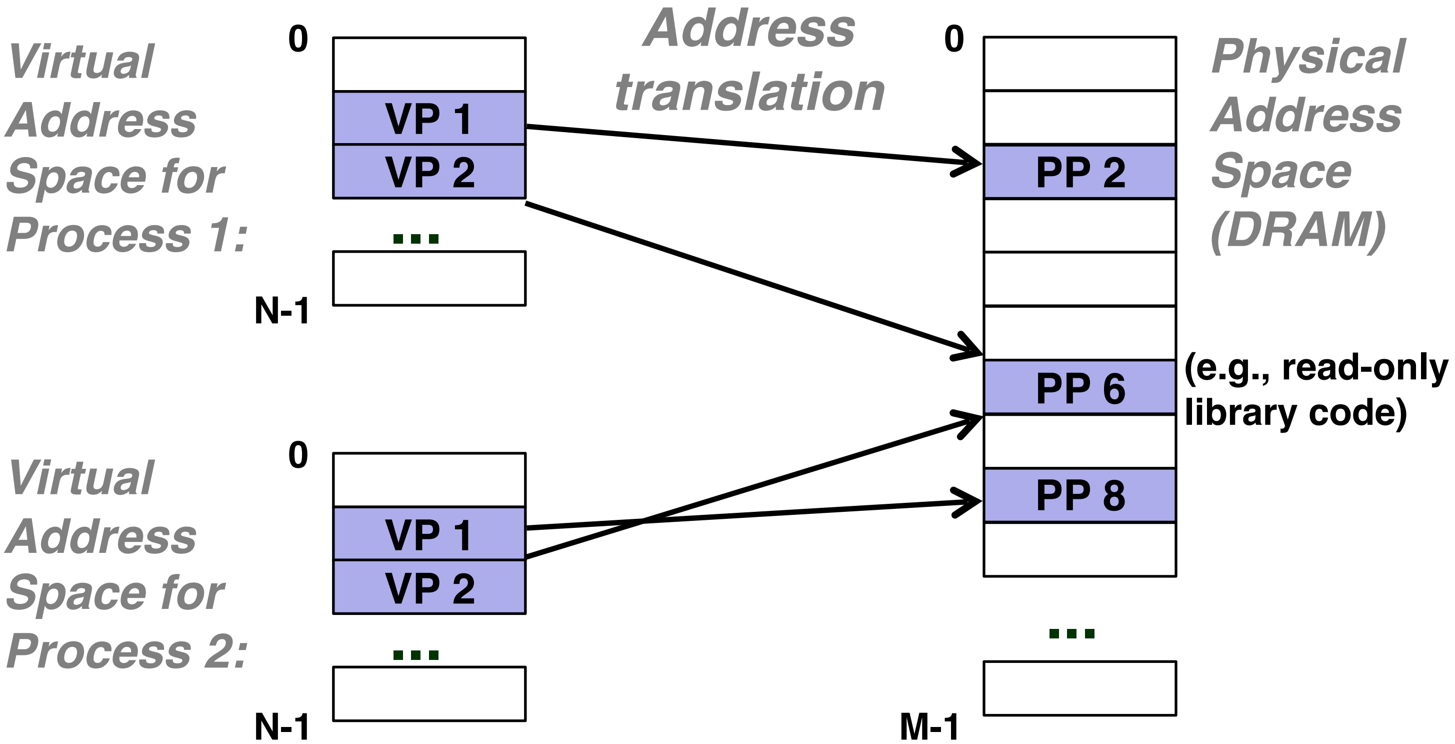
Loaded from the executable file

`0x400000`

0

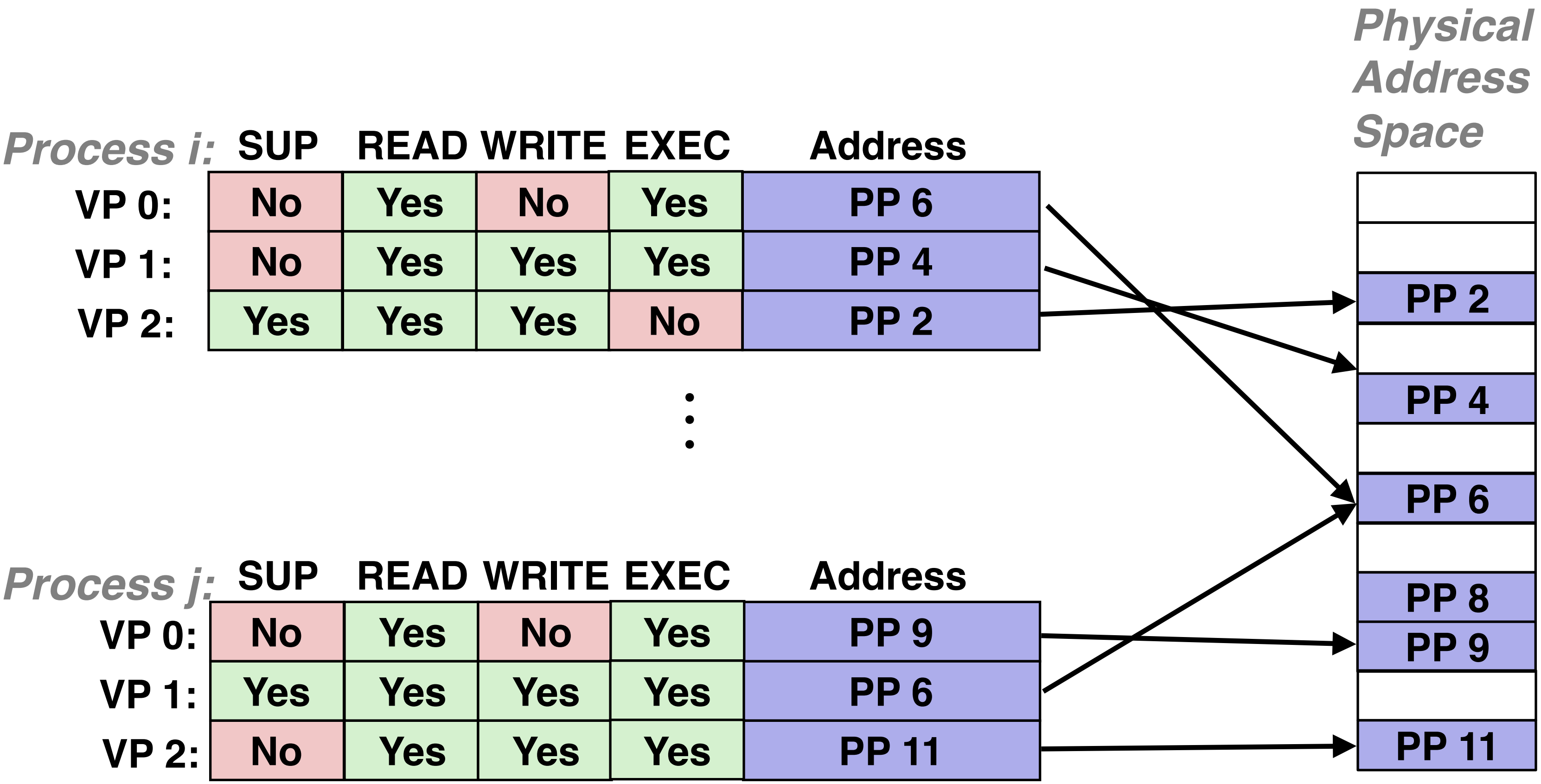
Virtual Memory Enables Sharing

- Simplifying memory allocation
  - Each virtual page can be mapped to any physical page
  - A virtual page can be stored in different physical pages at different times
- Sharing code and data among processes
  - Map virtual pages to the same physical page (here: PP 6)



VM Provides Further Protection Opportunities

- Extend PTEs with permission bits
- MMU checks these bits on each access



Process i:

	SUP	READ	WRITE	EXEC	Address
VP 0:	No	Yes	No	Yes	PP 6
VP 1:	No	Yes	Yes	Yes	PP 4
VP 2:	Yes	Yes	Yes	No	PP 2

⋮

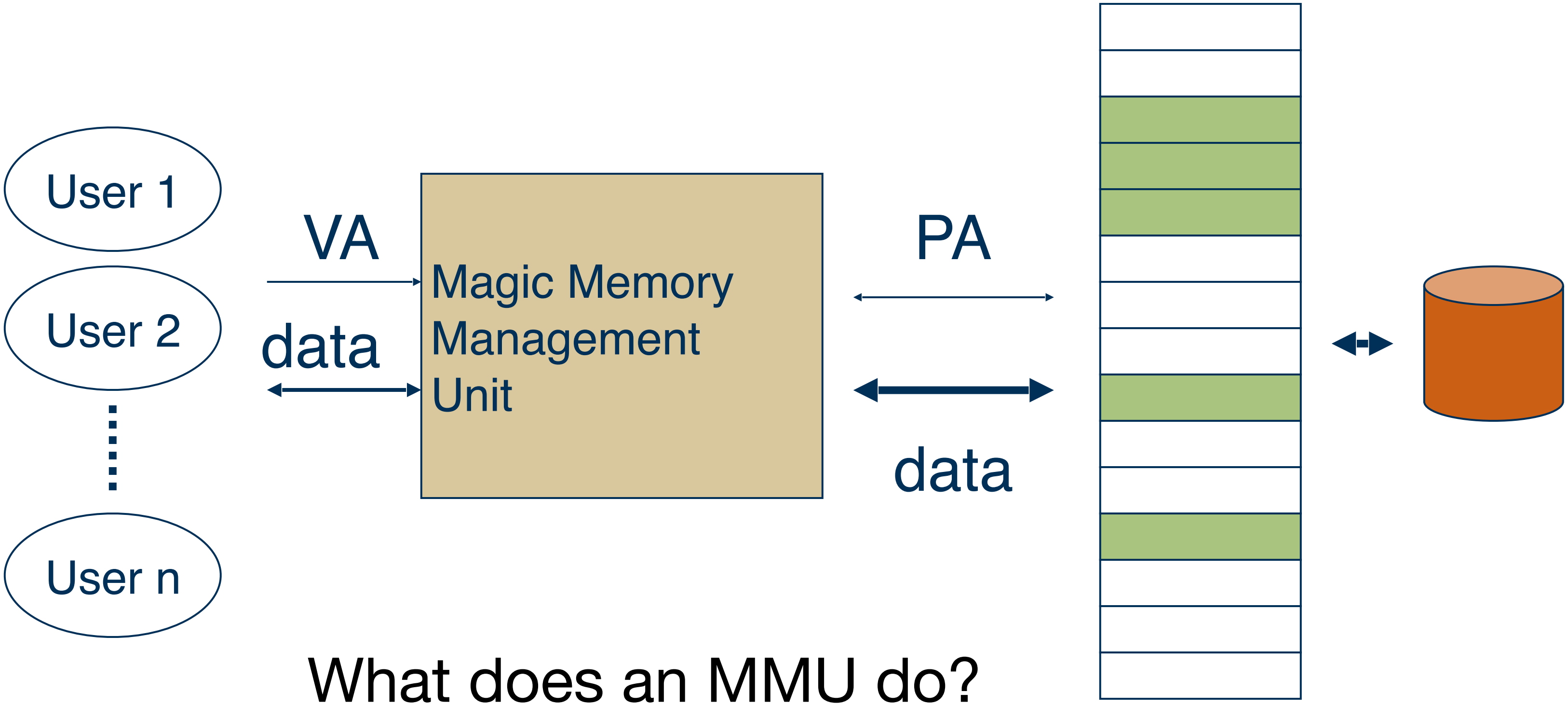
Process j:

	SUP	READ	WRITE	EXEC	Address
VP 0:	No	Yes	No	Yes	PP 9
VP 1:	Yes	Yes	Yes	Yes	PP 6
VP 2:	No	Yes	Yes	Yes	PP 11

Today

- Virtual memory (VM) illustration
- VM basic concepts and operation
- Other critical benefits of VM
- Address translation

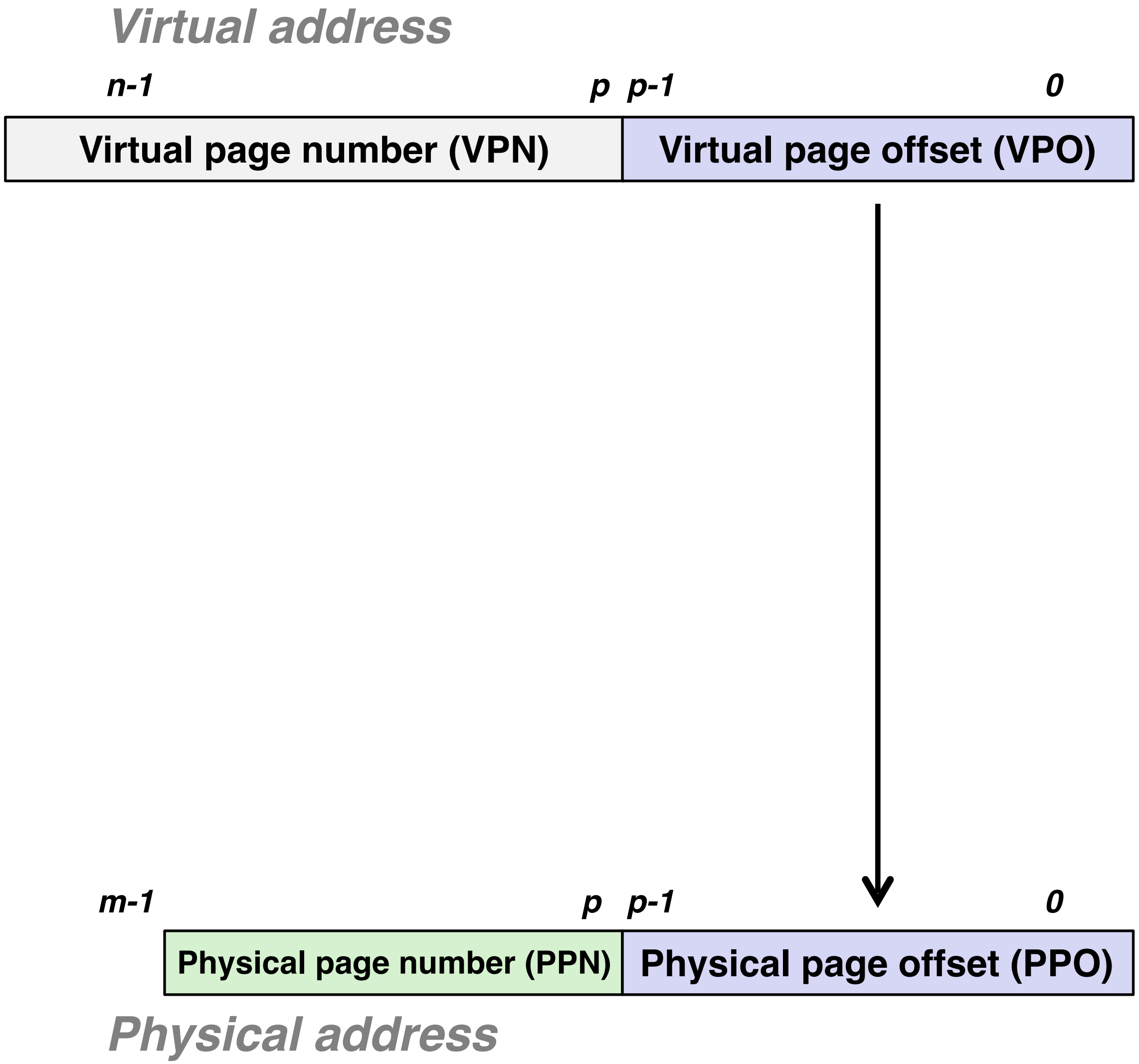
So Far…



What does an MMU do?

- Translate address
  - Enforce permissions
  - Fetch from disk

Address Translation With a Page Table



Virtual address

n-1 *p* *p-1* *0*

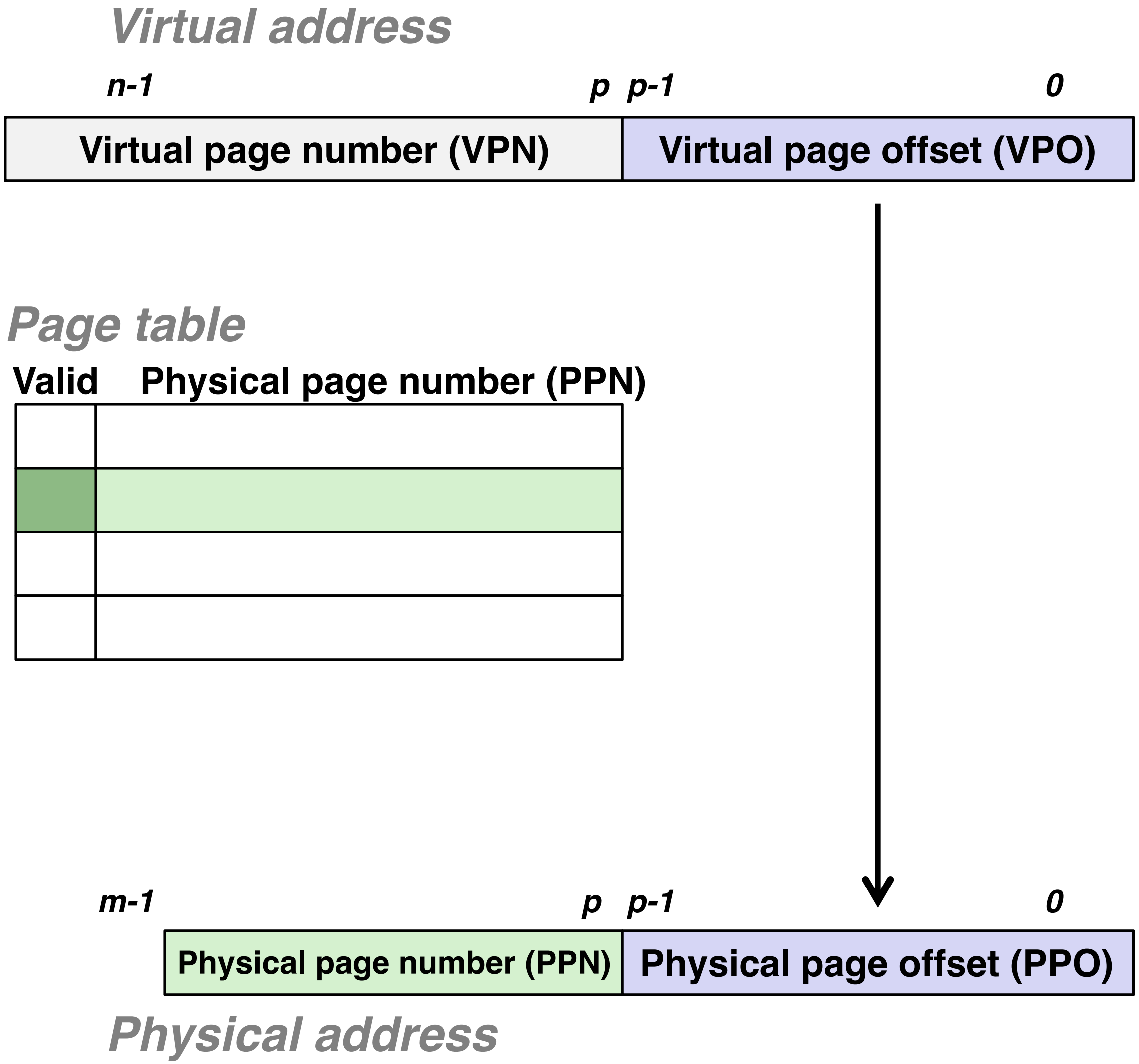
Virtual page number (VPN) | **Virtual page offset (VPO)**

m-1 *p* *p-1* *0*

Physical page number (PPN) | **Physical page offset (PPO)**

Physical address

Address Translation With a Page Table



Virtual address

n-1 *p* *p-1* *0*

Virtual page number (VPN) **Virtual page offset (VPO)**

Page table

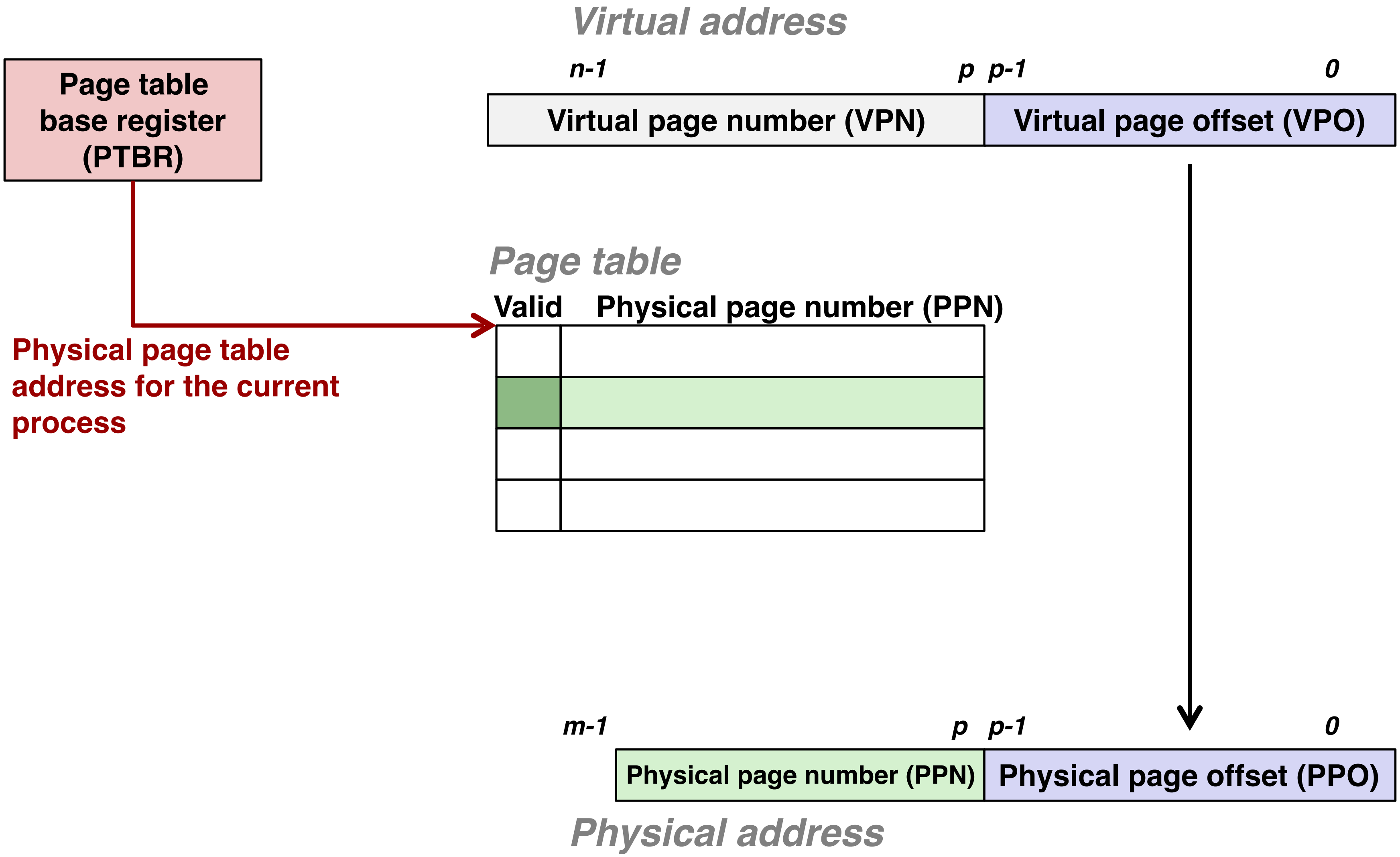
Valid **Physical page number (PPN)**

m-1 *p* *p-1* *0*

Physical page number (PPN) **Physical page offset (PPO)**

Physical address

Address Translation With a Page Table



Virtual address

n-1 *p* *p-1* *0*

Virtual page number (VPN) | Virtual page offset (VPO)

Page table base register (PTBR)

Physical page table address for the current process

Page table

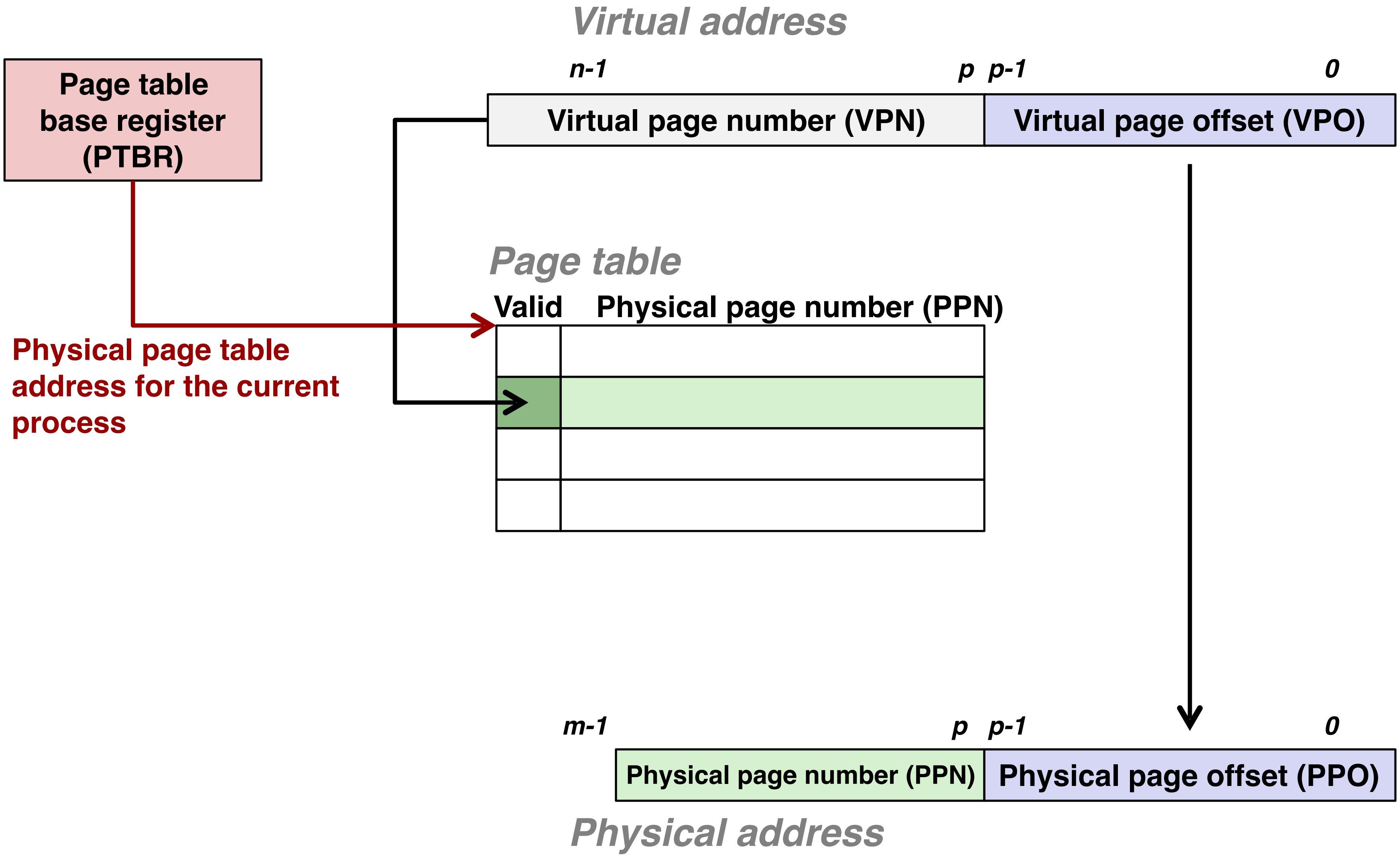
Valid Physical page number (PPN)

m-1 *p* *p-1* *0*

Physical page number (PPN) | Physical page offset (PPO)

Physical address

Address Translation With a Page Table



Virtual address

n-1 *p* *p-1* *0*

Virtual page number (VPN) | Virtual page offset (VPO)

Page table base register (PTBR)

Physical page table address for the current process

Page table

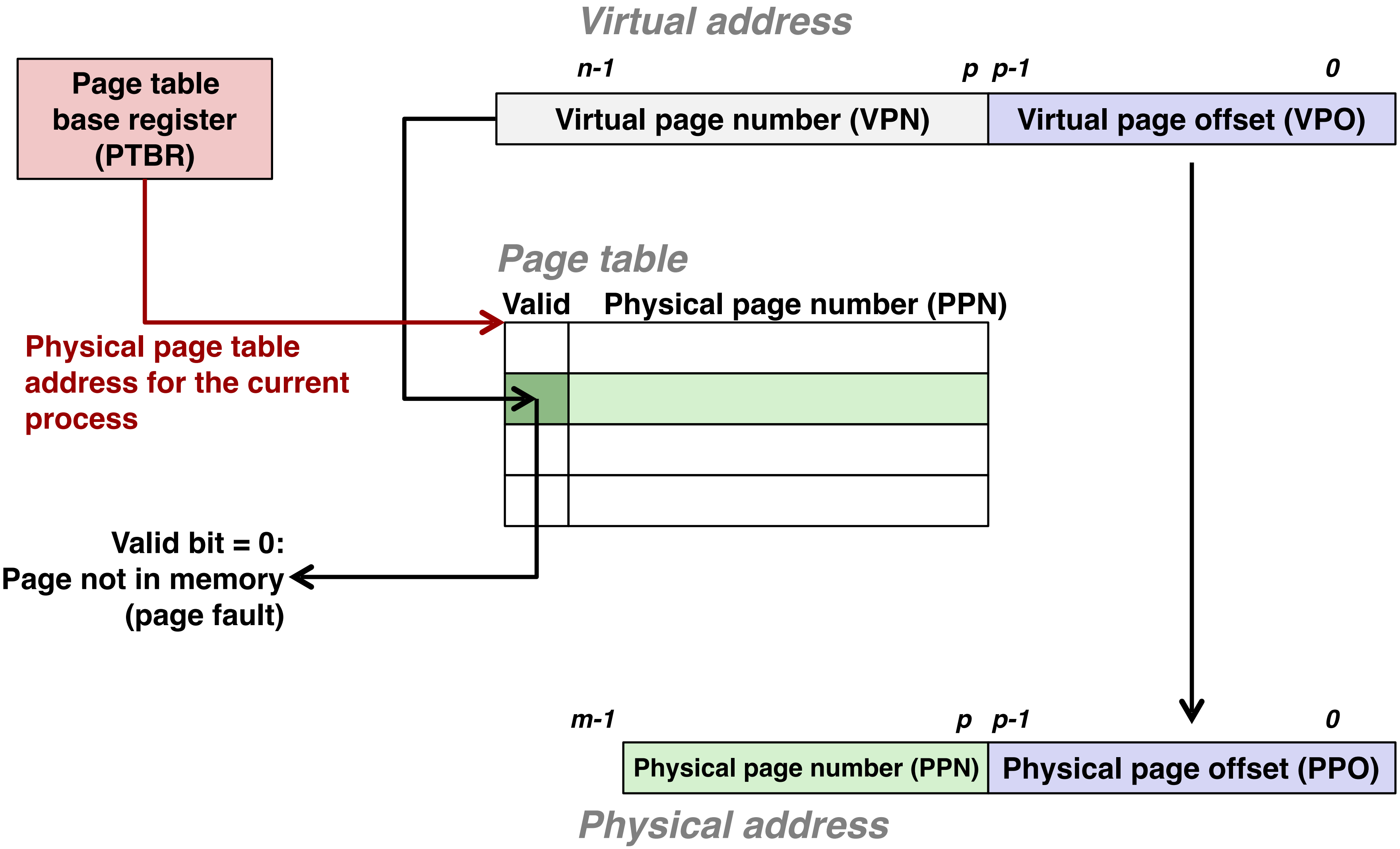
Valid Physical page number (PPN)

m-1 *p* *p-1* *0*

Physical page number (PPN) | Physical page offset (PPO)

Physical address

Address Translation With a Page Table



Virtual address

n-1 *p* *p-1* *0*

Virtual page number (VPN)

Virtual page offset (VPO)

Page table base register (PTBR)

Page table

Valid

Physical page number (PPN)

Physical page table address for the current process

Valid bit = 0:
Page not in memory
(page fault)

m-1 *p* *p-1* *0*

Physical page number (PPN)

Physical page offset (PPO)

Physical address

Address Translation With a Page Table