Universidade Estadual de Campinas
Instituto de Computação

MC504 Sistemas Operacionais

T11

Memory Virtualization
Paging

Referência principal
Ch.18 of *Operating Systems: Three Easy Pieces* by Remzi and Andrea Arpaci-Dusseau (pages.cs.wisc.edu/~remzi/OSTEP/)

Discutido em classe em 03 de setembro de 2018

Arthur João Catto, PhD

2º semestre de 2018

Partitioning Issues

- We have seen that
  - With fixed partitioning, any program, no matter how small, occupies an entire partition.
    - This sort of memory waste is called **internal fragmentation**.
  - With variable partitioning, due to the process allocation policy, holes of various sizes may appear and be scattered throughout memory.
    - This sort of memory waste is called **external fragmentation**.

Noncontiguous allocation

- One way to fight external fragmentation is to make the physical address space of a process noncontiguous.
  - The idea is to make all the available physical memory accessible to any running process.

- We have already studied segmentation, an early noncontiguous allocation concept, which reminds us of the pros and cons of variable partitioning.

- Paging is another early noncontiguous allocation concept, which takes a different approach.
  - It was designed at the University of Manchester and implemented in the Atlas Computer, which was commissioned in 1962.

The Paging Approach

- The idea is to use fixed-sized units instead of variable-sized logical segments.
- Split virtual memory into blocks of the same size (**pages**).
- Split physical memory into fixed-sized blocks (**frames**).
  - Frame size is 2^x bytes (today usually $9 \leq x \leq 13$ but it can be as high as 31).
- Keep track of all free frames.
- To run a program with n pages, load it into n free frames.
- Set up a **page table** to map virtual pages to the associated physical frames.

Points to ponder

- How can we virtualize memory with pages, so as to avoid the problems of segmentation?
- What are the basic techniques?
- How do we make those techniques work well, with minimal space and time overheads?
- Is this strategy prone to…
  - Internal fragmentation? Why?
  - External fragmentation? Why?

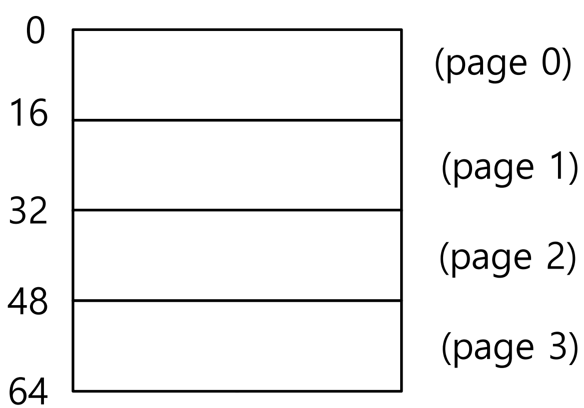
Advantages Of Paging

- Flexibility
  - It supports the abstraction of an address space effectively.
    - E.g. there will be no assumptions about how heap and stack grow and are used.
- Simplicity
  - Managing the page table requires only two hardware registers
    - Page-table base register (PTBR) points to page table.
    - Page-table length register (PTLR) indicates the size of the page table.
  - It is easy to manage the free-space.
    - The pages in the address space and the frames in the physical memory are the same size.
    - It is easy to allocate and keep a free list, e.g. by using a bitmap like 0011111100000001100, where each bit represents one physical page frame.

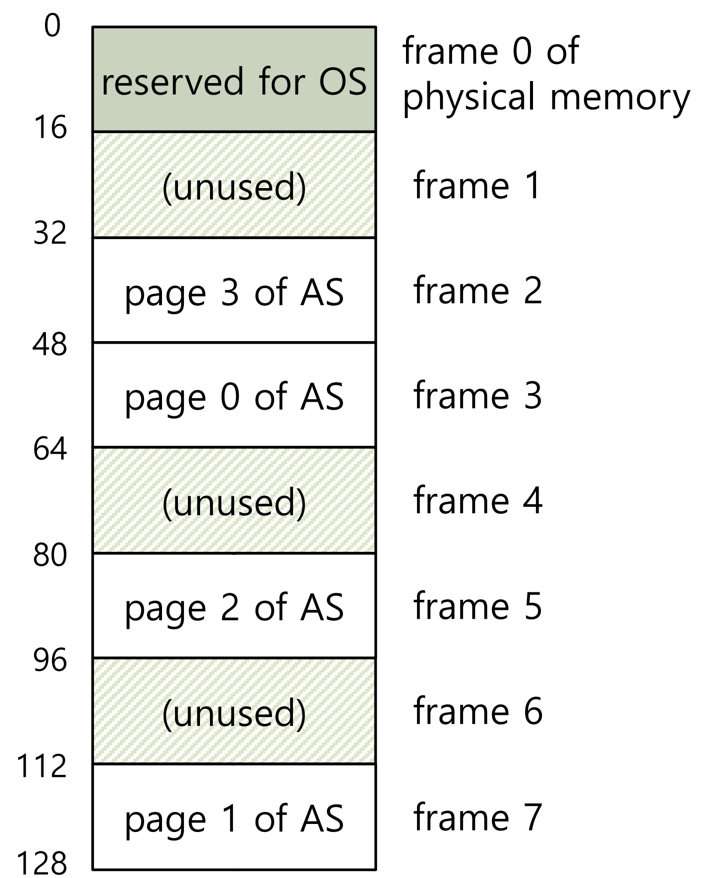
Example

A Simple Paging System

- Consider
  - A tiny 64B address space with four 16B pages
  - A small 128B physical memory with eight 16B page frames.



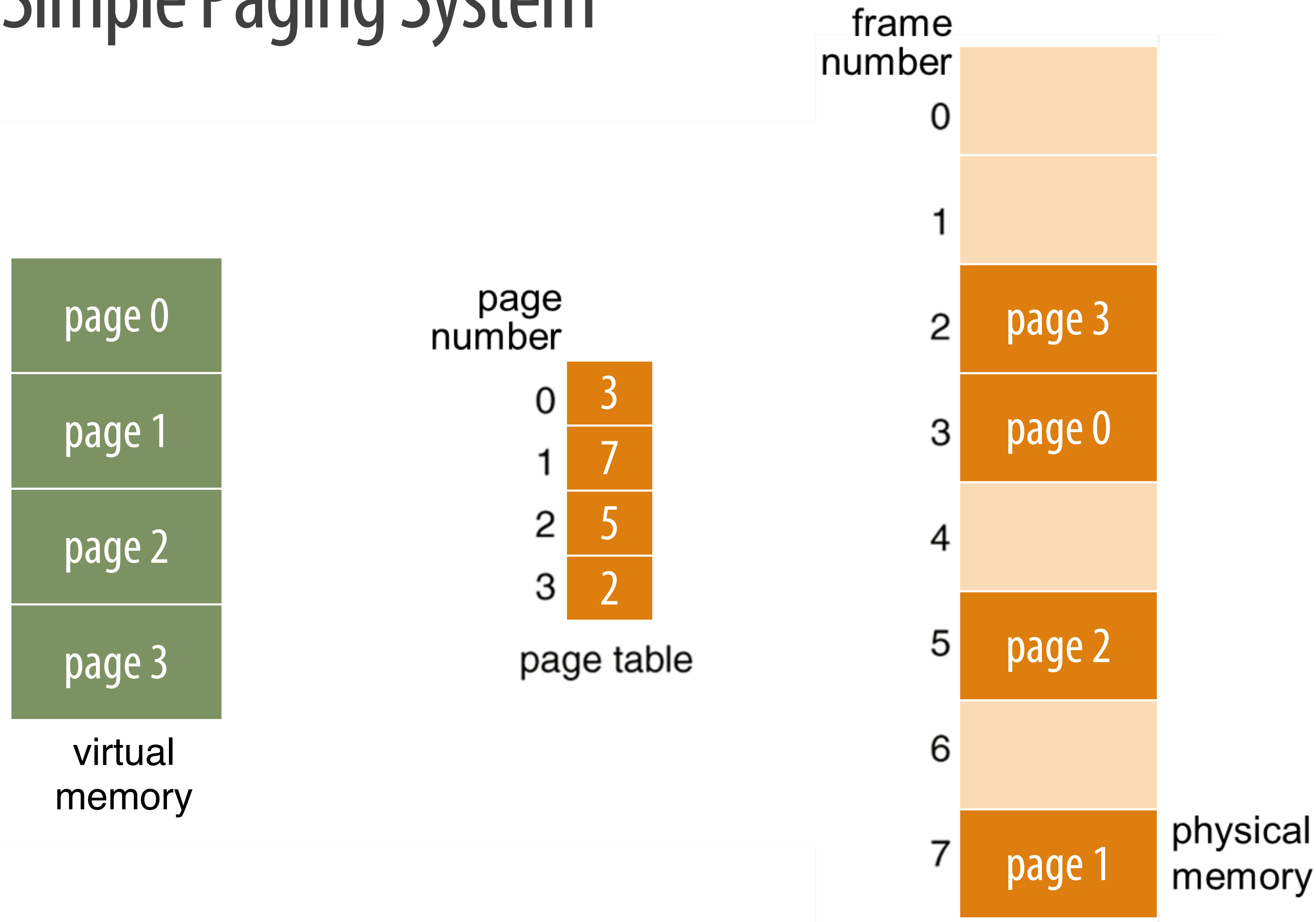
A Simple 64B Address Space



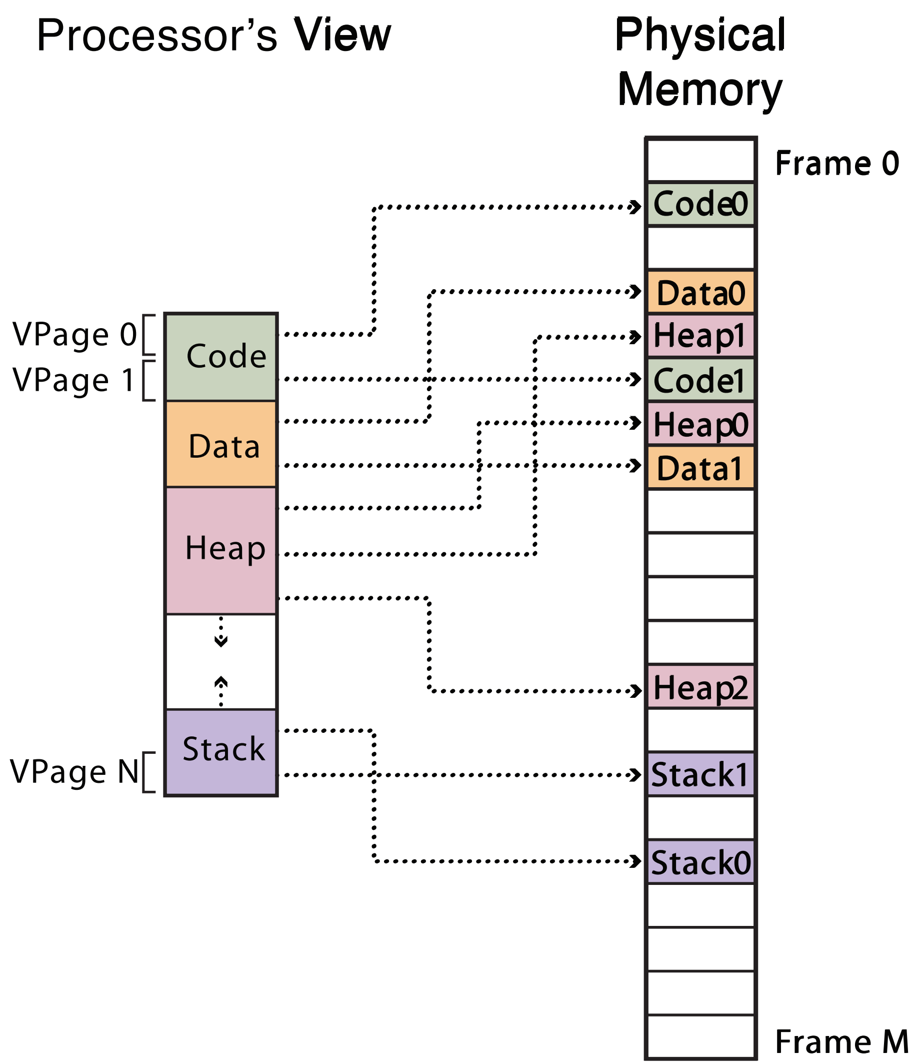
The Address Space Placed In Physical Memory

Example

A Simple Paging System



Paged Translation (Abstract)

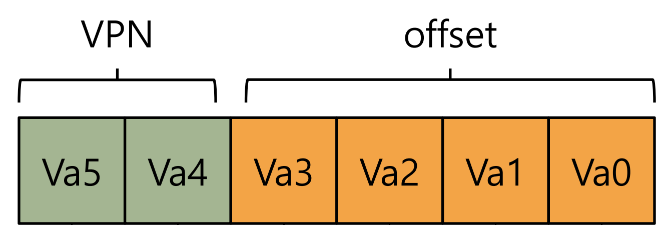


Address Translation Scheme

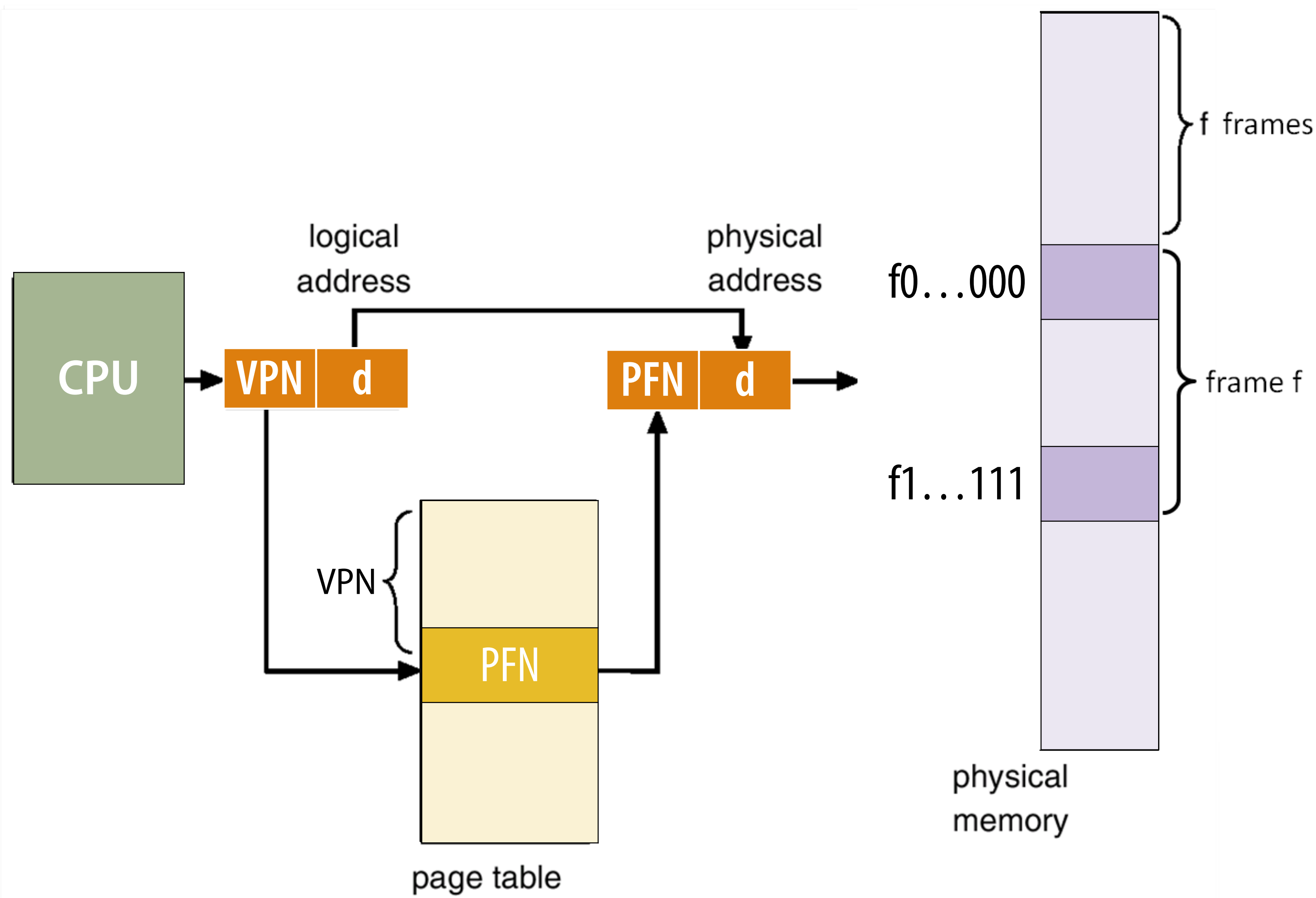
- Virtual address generated by CPU is divided into two parts

Virtual Page Number (VPN)	Page offset (d)
Used as an index into a page table which contains the base address of each page in physical memory.	Combined with base address to define the physical memory address that is sent to the memory unit.

- In our simple paging example, 6 bits will be required to cover the address space.
  - There are 4 pages, so the VPN will take 2 bits
  - The size of each page is 16B, so the offset will require 4 bits.



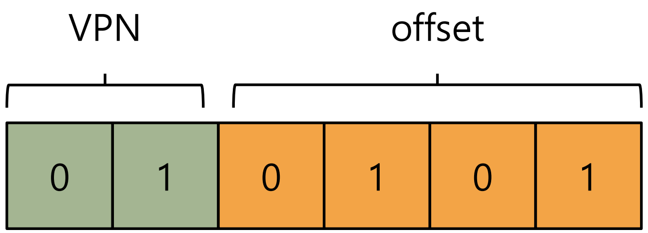
Address Translation Scheme



Example

Address Translation

- Consider the 64B address space in the simple paging scheme example.
- Take virtual address $\mathbf{21_{10}}$ and let us decompose it
  - $\mathbf{21_{10} = 010101_2}$, which means
    - VPN = 1 and
    - Offset = 5B

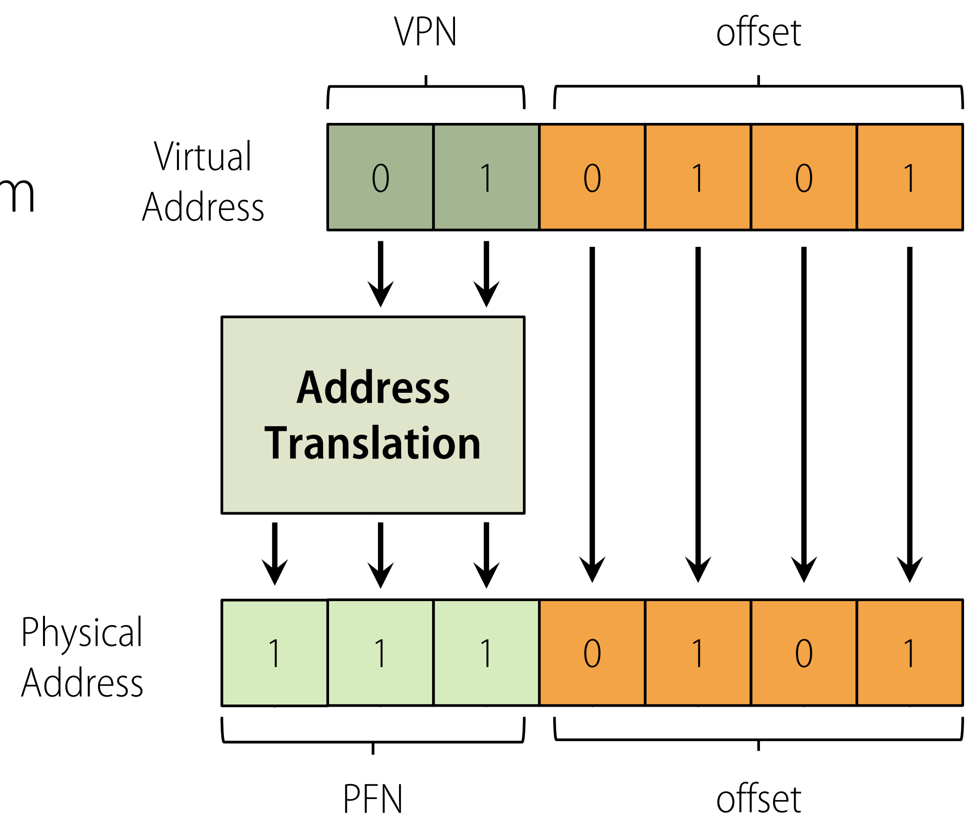


Example

Address Translation

- Assume now that, in our simple system, Virtual Pages are mapped onto Physical Frames according to the table on the left.
- Virtual address 21_{10} would be mapped onto memory address 117_{10} as shown by the diagram on the right.

VPN	PFN
0	3
1	7
2	5
3	2



Paging Example (4B pages, 32B mem)

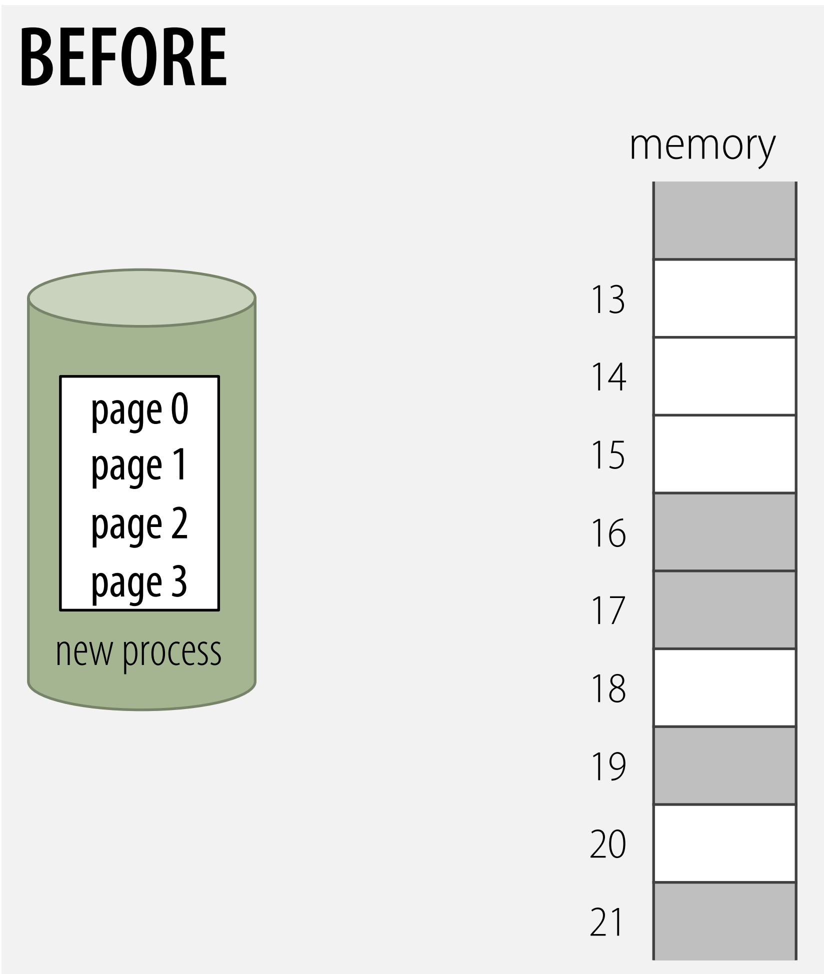
VA_{10}	VA_4	Value
0	00	A
1	01	B
2	02	C
3	03	D
4	10	E
5	11	F
6	12	G
7	13	H
8	20	I
9	21	J
10	22	K
11	23	L
12	30	M
13	31	N
14	32	O
15	33	P

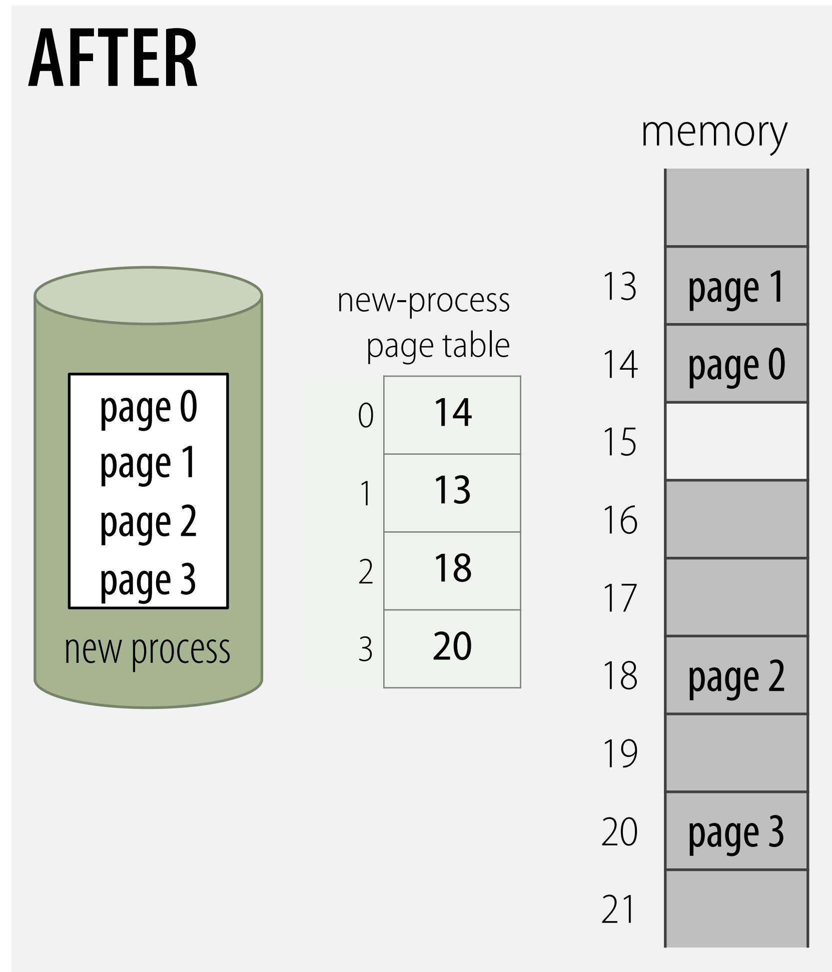
Page Table	
0	5
1	6
2	3
3	1
4	
5	
6	
7	

PA_4	PA_{10}	Value
00	0	
01	1	
02	2	
03	3	
10	4	M
11	5	N
12	6	O
13	7	P
20	8	
21	9	
22	10	
23	11	
30	12	I
31	13	J
32	14	K
33	15	L

PA_4	PA_{10}	Value
100	16	
101	17	
102	18	
103	19	
110	20	A
111	21	B
112	22	C
113	23	D
120	24	E
121	25	F
122	26	G
123	27	H
130	28	
131	29	
132	30	
133	31	

Free Frame Allocation





Points to ponder…

- How big can page tables be?
- Where are page tables stored?
- What are the typical contents of a page table?
- Does paging make the system (too) slow?

How Big Are Page Tables?

- Page tables can get awfully large, much larger than segment tables or base/bound registers that we have already discussed.
  - For example, take a 32-bit address space with 4KB pages.
    - Generating every possible address within a page requires $\mathbf{12}$ bits ($2^{12} = 4K$).
    - Since there are 32 bits in the address space and we have used 12, there will be 20 bits left, which means $\mathbf{2^{20}}$ pages.
    - Assuming that each entry in a page table takes 4B, this page table will occupy $2^{20} \times 4B = 4MB$.
    - And there will be one of these tables for each running process…

Where Are Page Tables Stored?

- Because page tables are so big, they cannot fit into any special on-chip hardware.
- Instead, they are kept in the kernel physical memory.

What Is In The Page Table?

- The page table is just a **data structure** that is used to map a virtual address to a physical address, and in its simplest form is just an array.
- The OS **indexes** the PT by VPN to get the Physical Frame Number and some flags, e.g.
  - **Valid (V)**: whether the frame is valid.
  - **Read/Write (R/W)**: whether the page can be read from or written to
  - **Present (P)**: whether the page is in physical memory or on disk
  - **Dirty (D)**: whether the page has been modified since it was read
  - **Accessed (A):** whether the page has been accessed

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12	11 10 9	8	7	6	5	4	3	2	1	0
Physical Frame Number (PFN)		G	PAT	D	A	PCD	PWT	U/S	R/W	P

Example: An x86 Page Table Entry (PTE)

Paging: Also Too Slow

- We have seen that in order to translate a virtual address, we need to access the Page Table to get the corresponding Physical Frame Number.
- Since the Page Table is stored in memory, this implies that, for every memory reference, paging requires the OS to perform **one extra memory reference**.
- An abstract protocol for what happens on each memory reference is shown on the next slide.

Accessing Memory With Paging

```
// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT;

// Form the address of the page-table entry (PTE)
PTEAddr = PTBR + (VPN * sizeof(PTE));

// Fetch the PTE
PTE = AccessMemory(PTEAddr);

// Check if process can access the page
if (PTE.Valid == False)
    RaiseException(SEGMENTATION_FAULT);
else if (CanAccess(PTE.ProtectBits) == False)
    RaiseException(PROTECTION_FAULT);
else {
    // Access is OK: form physical address and fetch it
    offset = VirtualAddress & OFFSET_MASK;
    PhysAddr = (PTE.PFN << PFN_SHIFT) | offset;
    Register = AccessMemory(PhysAddr);
}
```

Points to Ponder…

With paging, what is saved or restored on a process context switch?

What if the page size is very small?

What if the page size is very large?

Paging and Copy on Write

- **Can we share memory between processes?**
  - Set entries in both page tables to point to same page frames
  - Need core map of page frames to track which processes are pointing to which page frames (e.g., reference count)

- UNIX fork with copy on write
  - Copy page table of parent into child process
  - Mark all pages (in new and old page tables) as read-only
  - Trap into kernel on write (in child or parent)
  - Copy page
  - Mark both as writeable
  - Resume execution

Fill On Demand

- **Can I start running a program before its code is in physical memory?**
  - Set all page table entries to invalid
  - When a page is referenced for first time, kernel trap
  - Kernel brings page in from disk
  - Resume execution
  - Remaining pages can be transferred in the background while program is running

Sparse Address Spaces

- Might want many separate dynamic modules
  - Per-processor heaps
  - Per-thread stacks
  - Memory-mapped files
  - Dynamically linked libraries
- **What if virtual address space is large?**
  - 32-bits, 4KB pages → 500K page table entries
  - 64-bits, 4KB pages → 4 quadrillion page table entries