CS3281 / CS5281

Advanced Virtual Memory

CS3281 / CS5281

Spring 2024

**Some lecture slides borrowed and adapted from CMU's "Computer Systems: A Programmer's Perspective" and MIT's 6.S081 Course*

Today

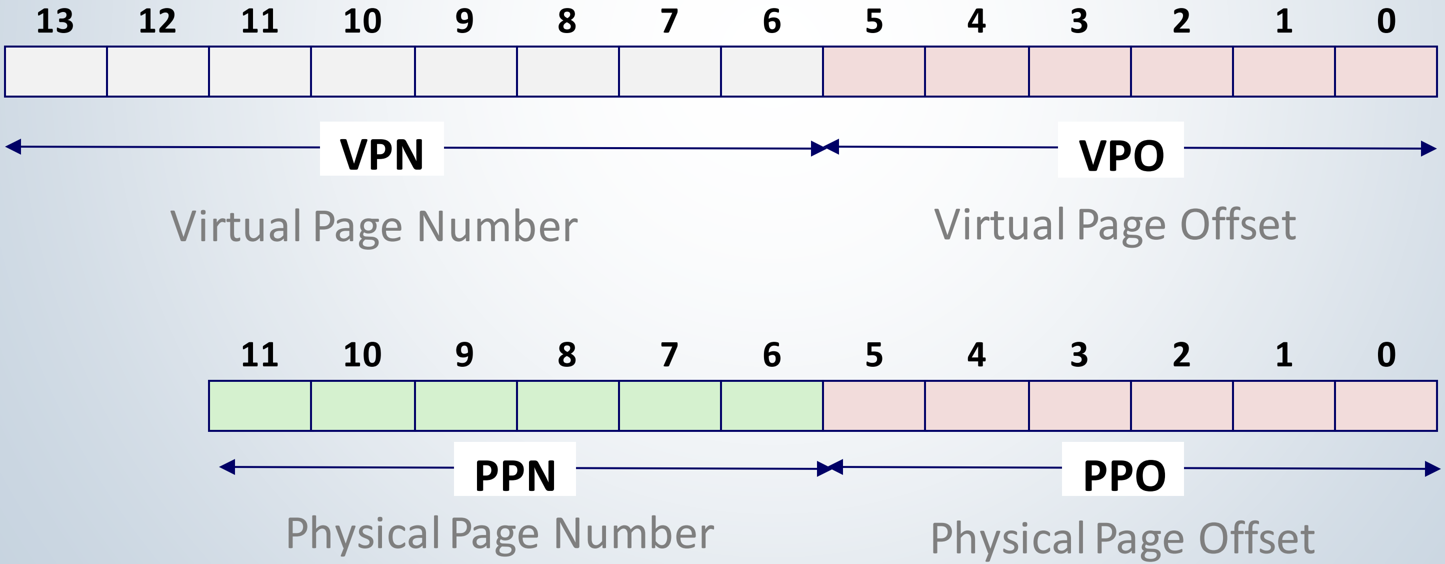
- Simple memory system example
- Case Study: RISC-V
- Memory mapping

Review of Symbols

- Basic Parameters
  - **$N = 2^n$**: Number of addresses in virtual address space
  - **$M = 2^m$**: Number of addresses in physical address space
  - **$P = 2^p$** : Page size (bytes)
- Components of the virtual address (VA)
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number
- Components of the physical address (PA)
  - **PPO**: Physical page offset (same as VPO)
  - **PPN:** Physical page number

Simple Memory System Example

- Addressing
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bytes



Virtual Page Number

Virtual Page Offset

Physical Page Number

Physical Page Offset

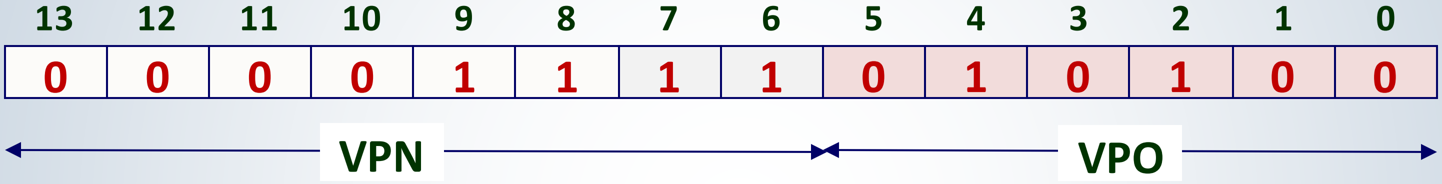
Simply Memory System Page Table

- Only show first 16 entries (out of 256)

VPN	PPN	Valid
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

Address Translation Example #1

Virtual Address: 0x03D4

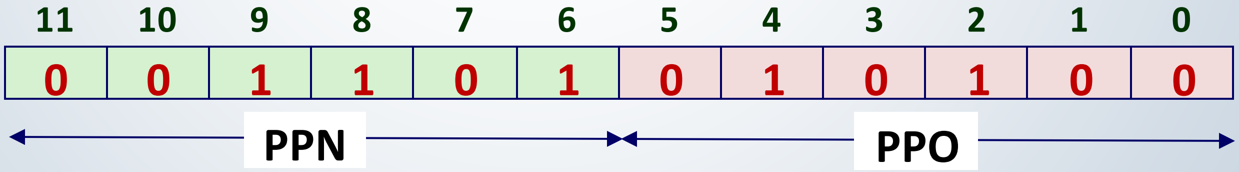


13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0

VPN: **0x0F**

Page Fault: N

PPN: **0x0D**



11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	1	0	1	0	0

Address Translation Example #2

Virtual Address: 0x0020

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0

VPN **VPO**

VPN: **0x00**

Page Fault: N

PPN: **0x28**

11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	1	0	0	0	0	0

PPN **PPO**

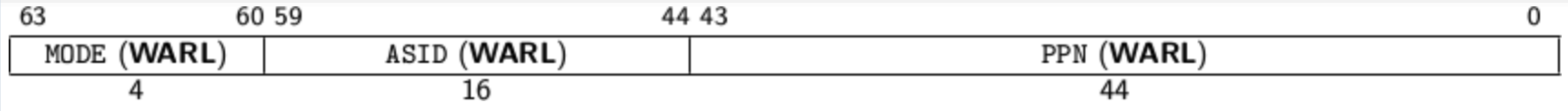
Today

- Simple memory system example
- Case Study: RISC-V
- Memory mapping

Virtual Memory in RISC-V

- Supports different addressing modes:
  - Sv32, Sv39, sV48 -> number of virtual address bits
  - We focus on Sv39, which has a 3-level page table
- Register called supervisor address translation and protection (satp) points to the page root
- satp is set using a special instruction called control status register write (csrw)
- Only allowed in kernel model. Why?

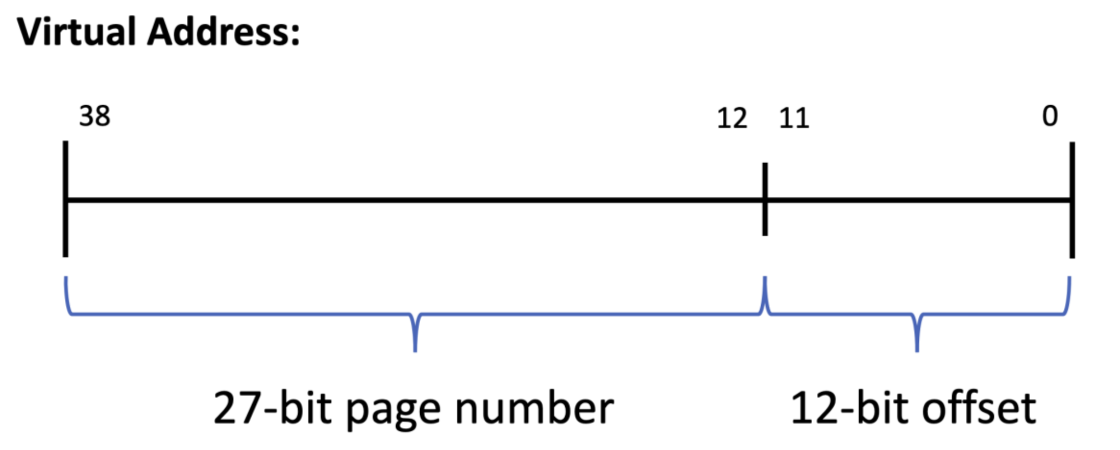
satp



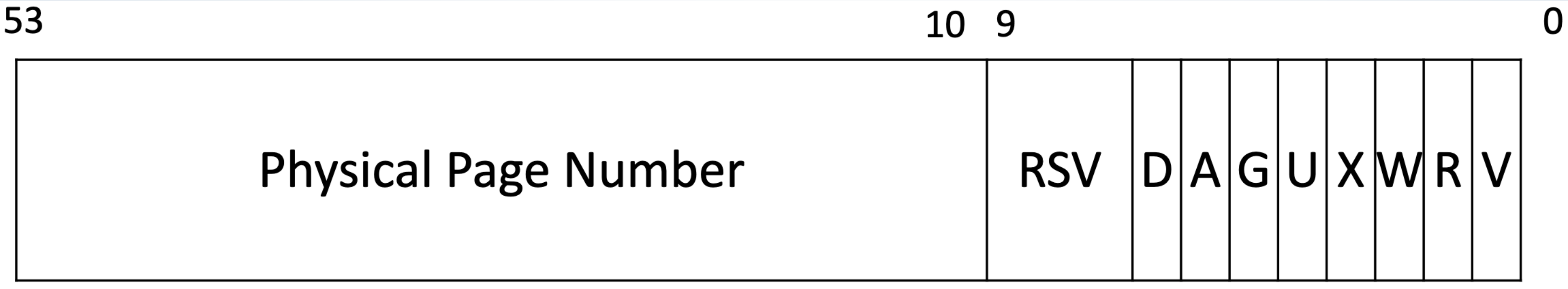
63 — 60	59 — 44	43 — 0
MODE (WARL)	ASID (WARL)	PPN (WARL)
4	16	44

Virtual Memory in RISC-V (Sv39)

- Virtual addresses are divided in 4-KB pages
- 39 bit address
- 4KB = 2^12
- 39 – 12 = 27 bits for page number



Page Table Entries

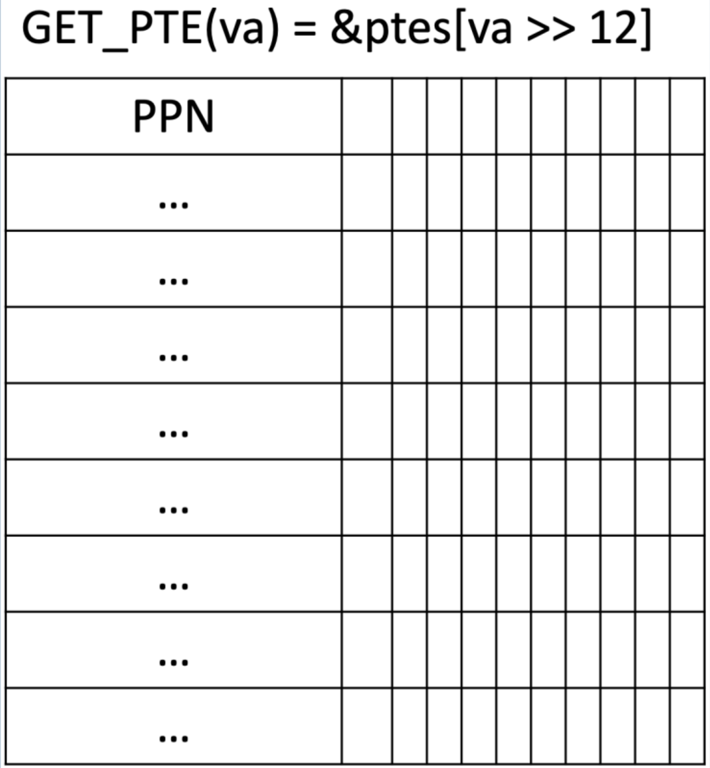


53 10 9 0

Physical Page Number RSV D A G U X W R V

- Some important information
- Physical page number: 44-bit physical page location
- U: If set, userspace can access this virtual address
- W: if set, the CPU can write to this virtual address
- V: if set, an entry for this virtual address exists (is valid)
- RSV: Ignored by MMU

What if we store PTEs in a single array?



GET_PTE(va) = &ptes[va >> 12]

PPN										
...										
...										
...										
...										
...										
...										
...										
...										

How large is this array?

What if we store PTEs in a single array?

```
GET_PTE(va) = &ptes[va >> 12]
```

PPN									
…									
…									
…									
…									
…									
…									
…									
…									

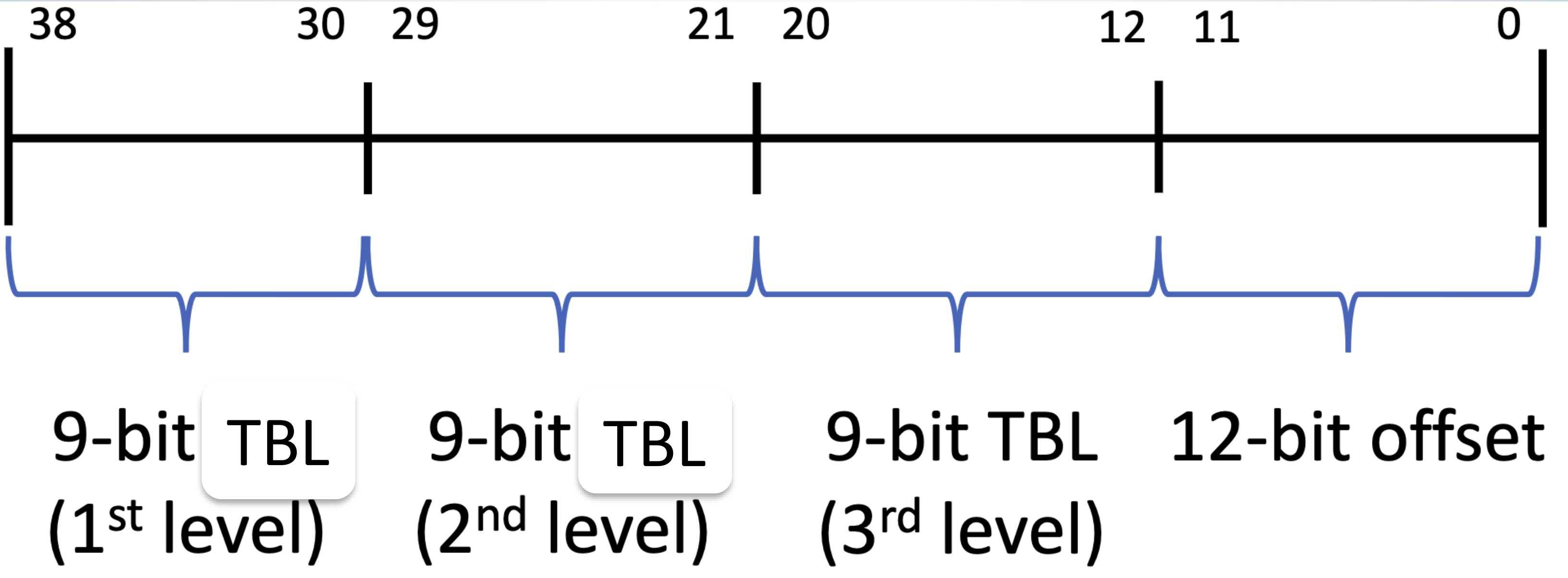
How large is this array?

Each entry is (padded to) 64 bits (8 bytes)

2^27 Virtual Page Numbers (2^39/2^12)

2^27 * 8 = 2^30 = 1GB!

RISC-V Solution: Use three levels to save space



38 30 29 21 20 12 11 0

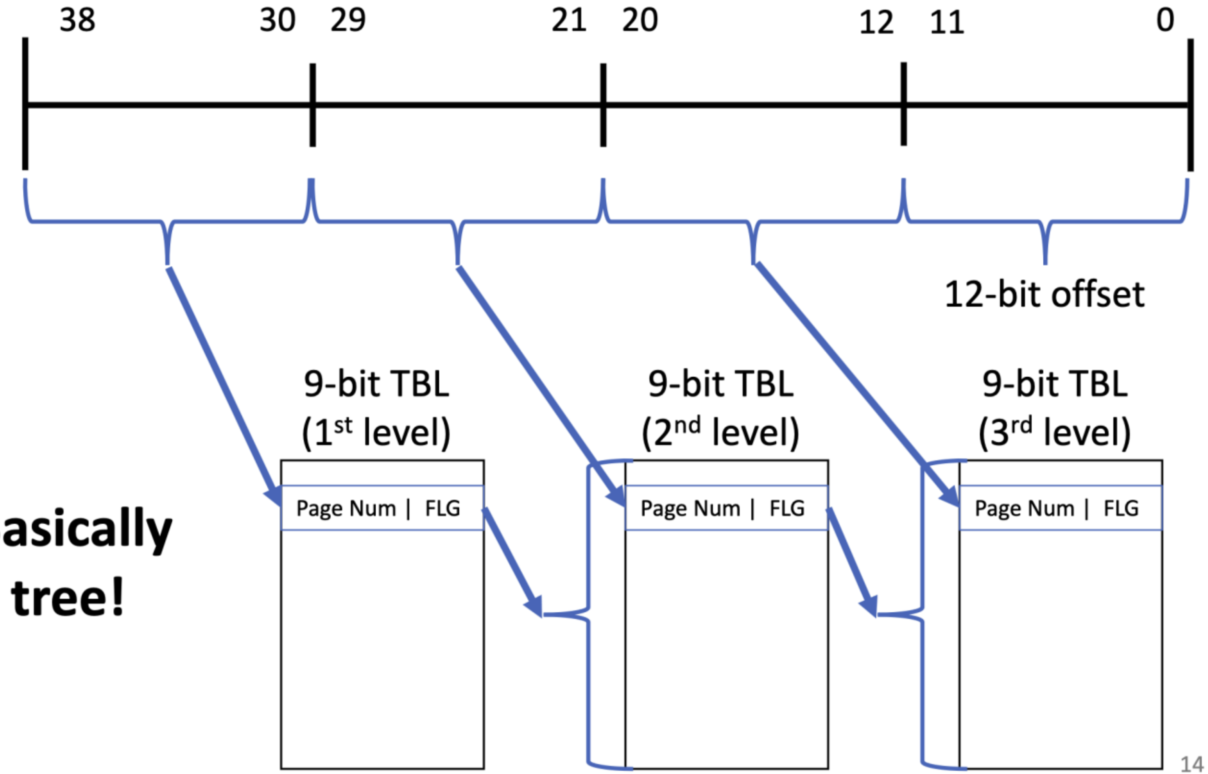
9-bit TBL (1st level)

9-bit TBL (2nd level)

9-bit TBL (3rd level)

12-bit offset

RISC-V multi-level page tables



Basically a tree!

Each table is 1 page = 4096B

Each PTE is 64 bits

How many PTEs per table?

Multi-level page table (example)

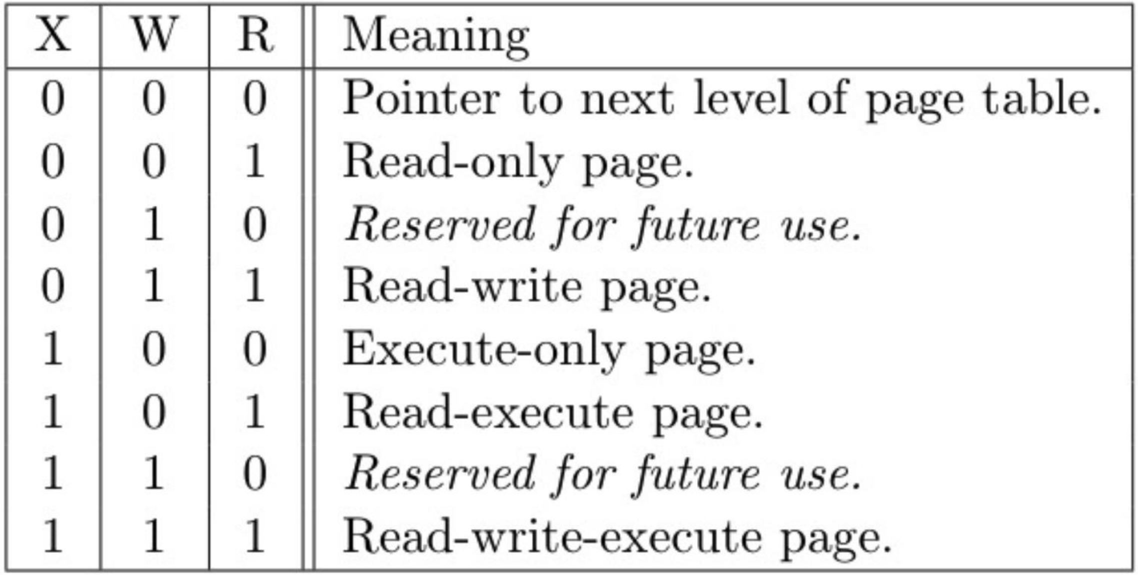
- Consider a 3-level page table. A page table has 64 entries and each entry has 64 bits. If the page size is 4K bytes, how much is the size of the virtual address space?

How do we use this in practice?

- CPU sets satp register to point to the first-level page directory
- There is only 1 first-level page directory per process
- By swapping the satp register, you completely change the functional address space
- Operating system modifies page tables and directories to layout memory as desired
- Hardware “walks” this page-table tree data structure to translate from virtual address to physical address and actually fetch memory

More about flags in RISC-V

- If U is cleared, only kernel can access
- If flag permission is violated, we get a page fault

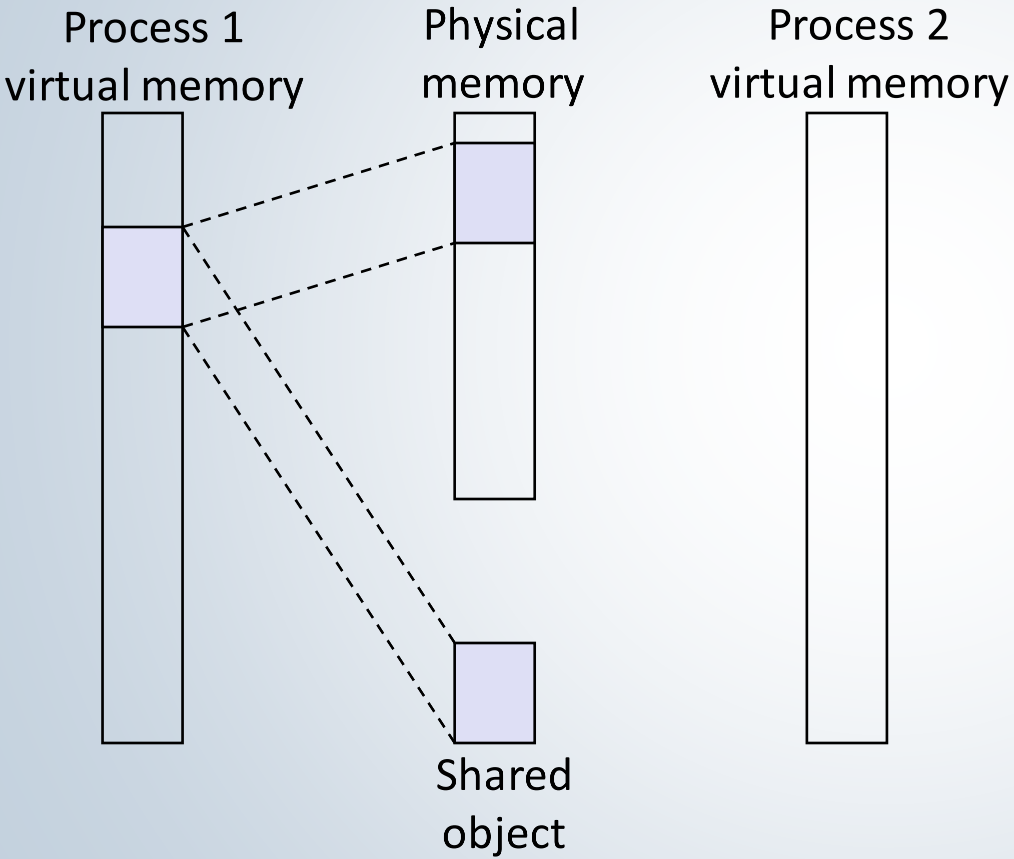


X	W	R	Meaning
0	0	0	Pointer to next level of page table.
0	0	1	Read-only page.
0	1	0	*Reserved for future use.*
0	1	1	Read-write page.
1	0	0	Execute-only page.
1	0	1	Read-execute page.
1	1	0	*Reserved for future use.*
1	1	1	Read-write-execute page.

Today

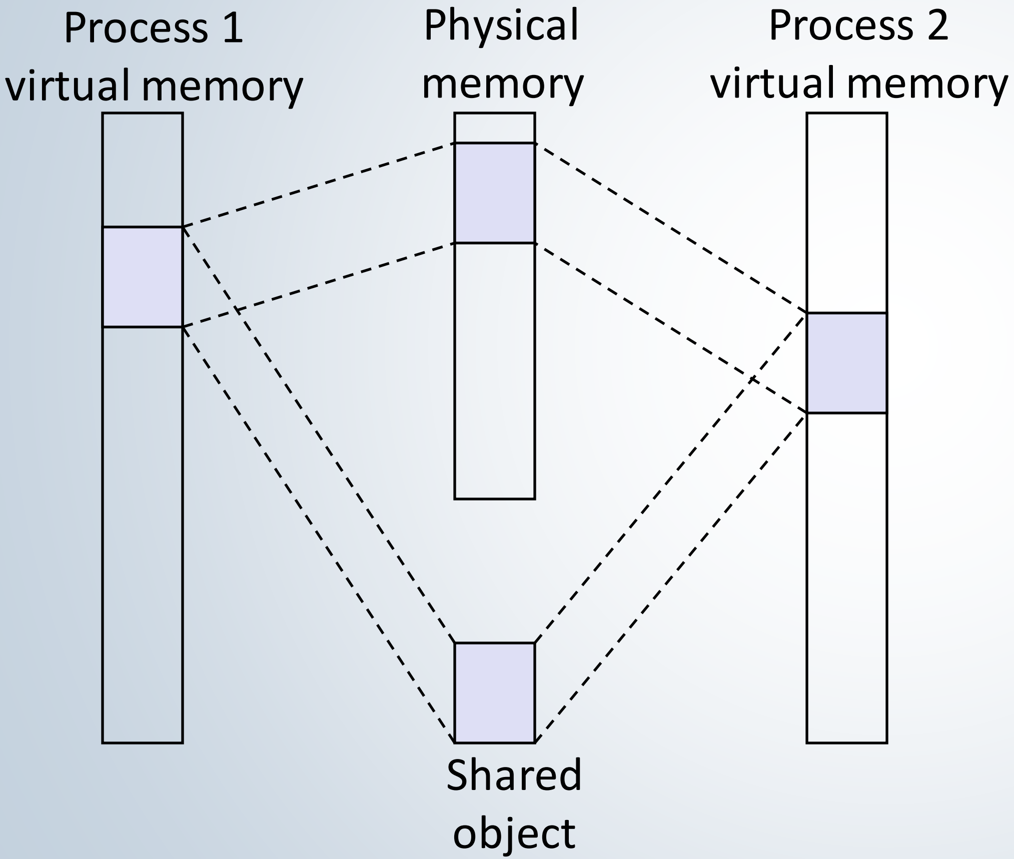
- Simple memory system example
- Case Study: RISC-V
- Shared Memory and Copy-on-Write
- Memory mapping

Sharing Revisited: Shared Objects



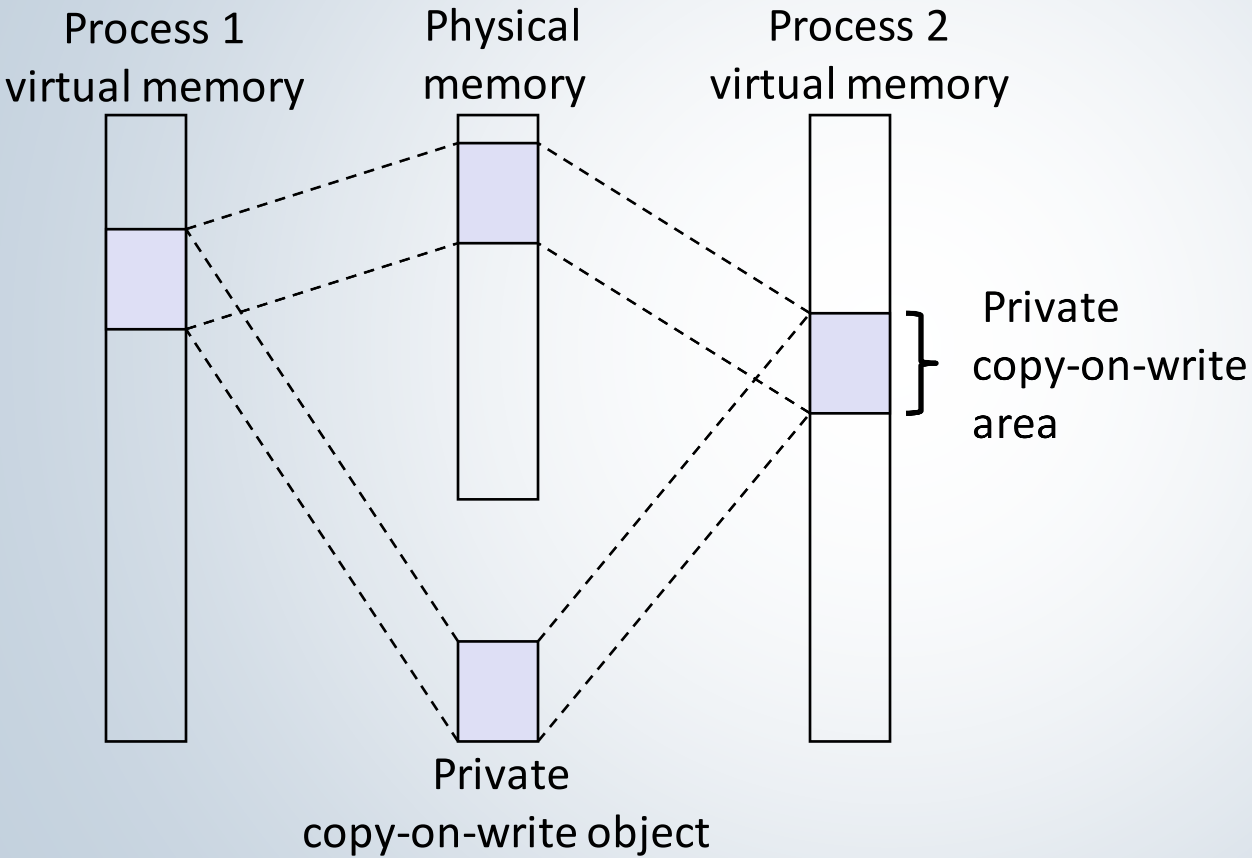
- **Process 2 maps the shared object.**

Sharing Revisited: Shared Objects



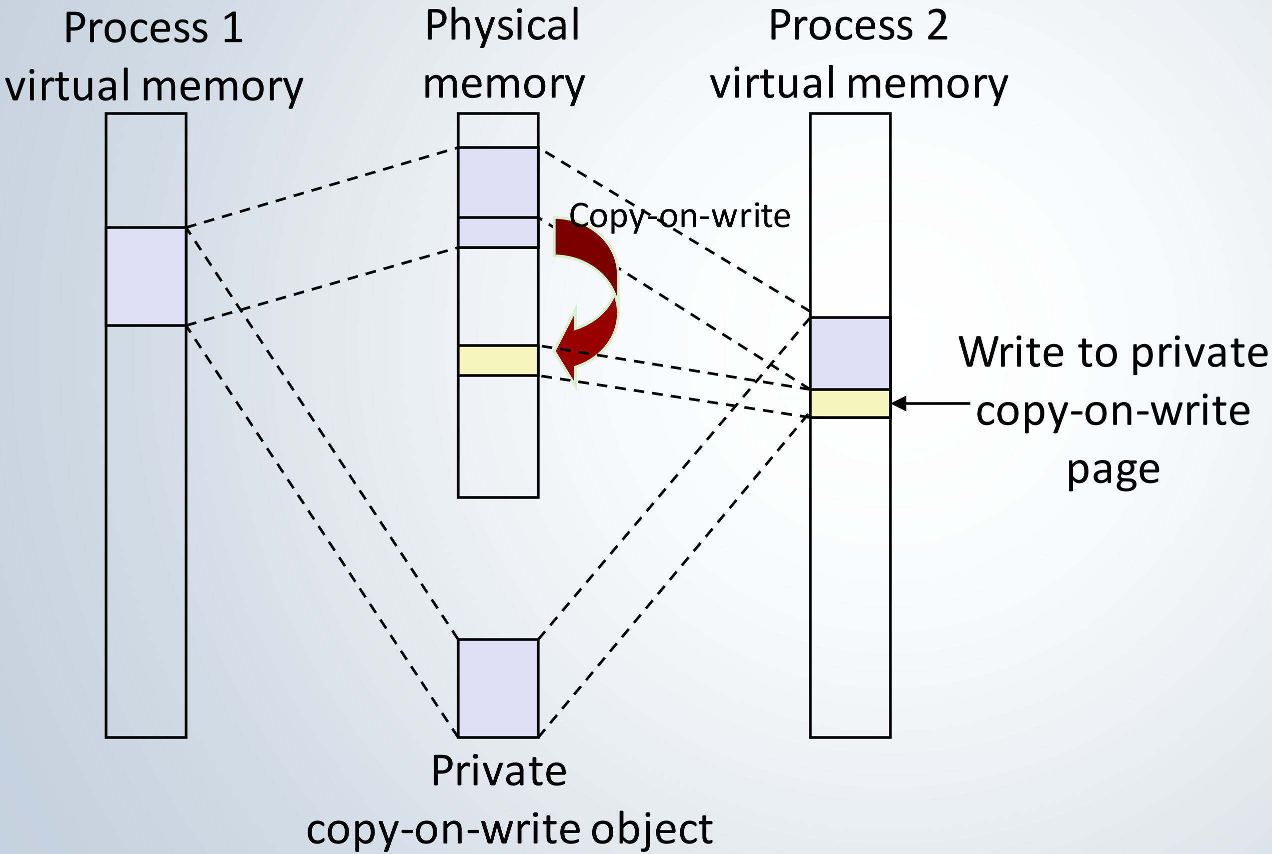
- **Process 2 maps the shared object.**
- Notice how the virtual addresses can be different.

Sharing Revisited: Copy-On-Write (COW) Objects



- Two processes mapping a *private copy-on-write (COW)* object
- Area flagged as private copy-on-write
- PTEs in private areas are flagged as read-only

Sharing Revisited: Copy-On-Write (COW) Objects



- Instruction writing to private page triggers protection fault
- Handler creates new R/W page
- Instruction restarts upon handler return
- Copying deferred as long as possible!

The fork() Function Revisited

- Can use COW memory mapping in `fork()` to provides private address space for each process without duplicating physical memory unnecessarily

- To create virtual address for new process
  - Create exact copies of current page tables
  - Flag each page in both processes as read-only (clear `write` flag)
  - Flag each `writeable page` in both processes as private COW

- On return, each process has identical view of memory but only one copy of physical memory exists

- Subsequent writes trigger COW mechanism and force pages to be duplicated when needed

Memory Mapping (not in xv6)

- VM areas initialized by associating them with disk objects.
  - Process is known as ***memory mapping***.

- Area can be *backed by* (i.e., get its initial values from) :
  - ***Regular file*** on disk (e.g., an executable object file)
    - Initial page bytes come from a section of a file
  - ***Anonymous file*** (e.g., nothing)
    - First fault will allocate a physical page full of 0's (***demand-zero page***)
    - Once the page is written to (***dirtied***), it is like any other page

- Dirty pages are copied back and forth between memory and a special *swap file*.

User-Level Memory Mapping (Linux)

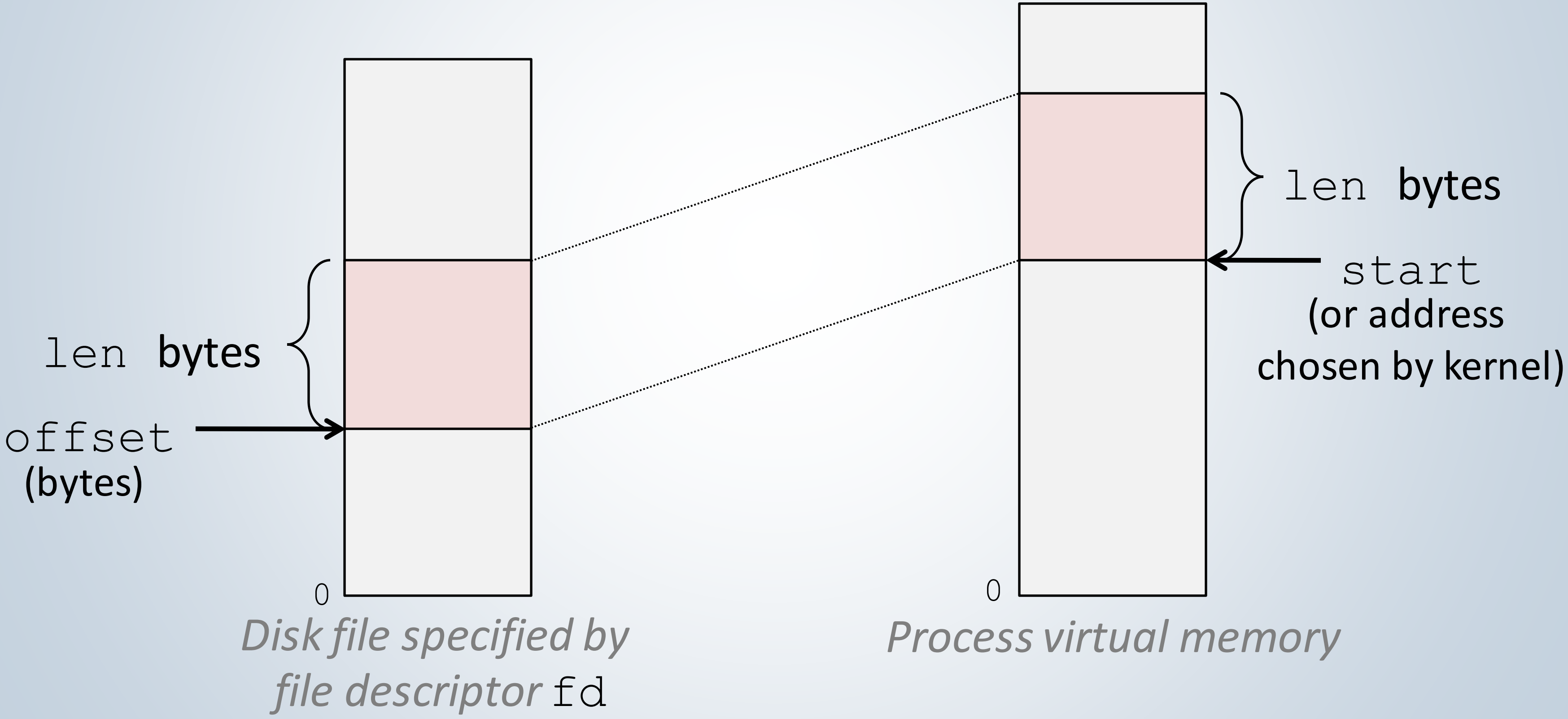
```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

- Map **`len`** bytes starting at offset **`offset`** of the file specified by file description **`fd`**, preferably at address **`start`**
  - **`start:`** may be 0 for “pick an address”
  - **`prot`**: PROT_READ, PROT_WRITE, ...
  - **`flags`**: MAP_ANON, MAP_PRIVATE, MAP_SHARED, ...

- Return a pointer to start of mapped area (may not be **`start`**)

User-Level Memory Mapping

```
void *mmap(void *start, int len, int prot, int flags, int fd, int offset)
```



Using mmap() to Copy Files (Linux)

- Copying a file to `stdout` without transferring data to user space

```
#include "csapp.h"

void mmapcopy(int fd, int size)
{

   /* Ptr to memory mapped area */
   char *bufp;

   bufp = mmap(NULL, size,
            PROT_READ,
            MAP_PRIVATE,
            fd, 0);
   Write(1, bufp, size);
   return;
}
```

```
/* mmapcopy driver */
int main(int argc, char **argv)
{
   struct stat stat;
   int fd;

   /* Check for required cmd line arg */
   if (argc != 2) {
      printf("usage: %s <filename>\n",
           argv[0]);
      exit(0);
   }

   /* Copy input file to stdout */
   fd = Open(argv[1], O_RDONLY, 0);
   Fstat(fd, &stat);
   mmapcopy(fd, stat.st_size);
   exit(0);
}
```