CSCI-UA.0201-003

Computer Systems Organization

Lecture 18: Virtual Memory: Systems

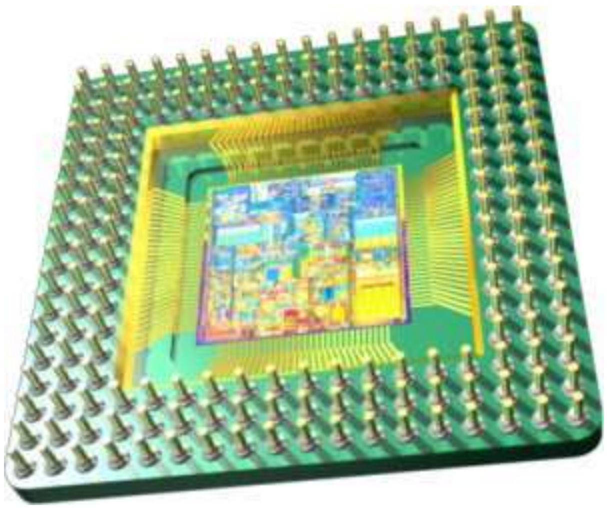
Mohamed Zahran (aka Z)

mzahran@cs.nyu.edu

http://www.mzahran.com

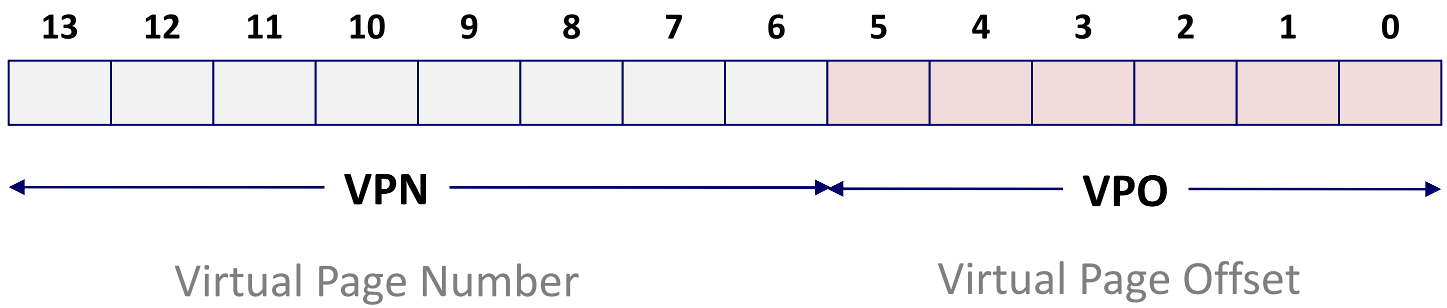
Some slides adapted (and slightly modified) from:

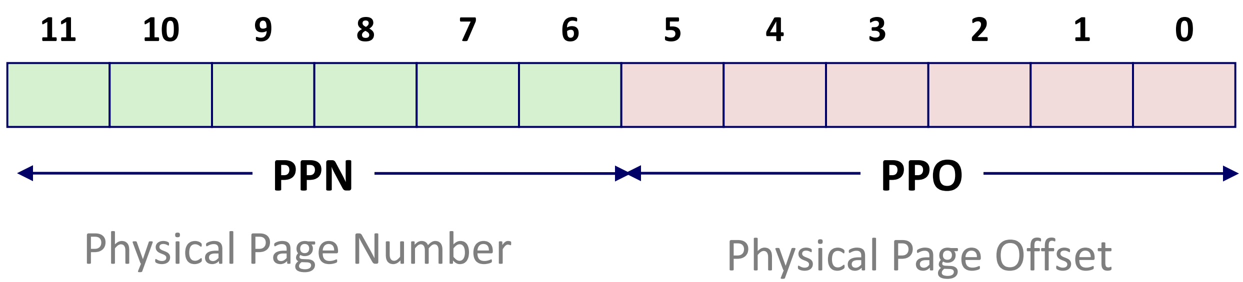
- Clark Barrett
- Jinyang Li
- Randy Bryant
- Dave O'Hallaron



Toy Memory System Example

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





Toy Memory System Page Table

VPN	PPN	Valid
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

VPN	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

1-level page table: How many PTEs?

Address Translation Example

Virtual Address: `0x0354`

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

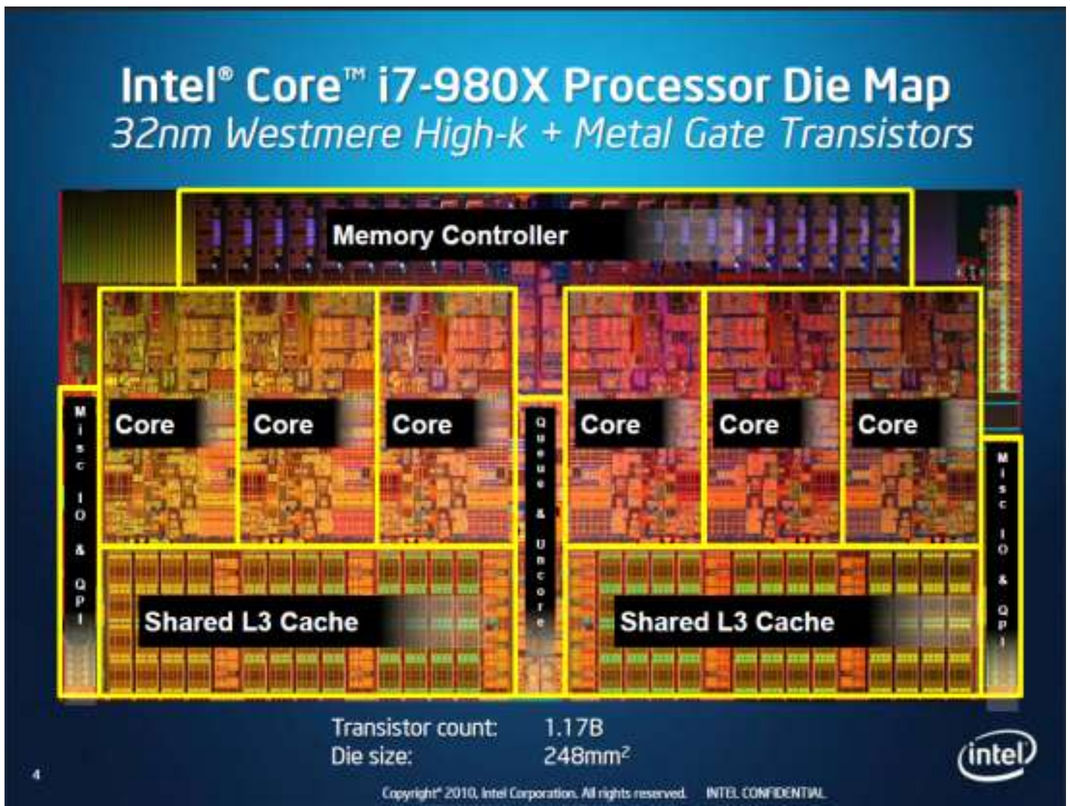
← VPN →← VPO →

VPN	*PPN*	*Valid*
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

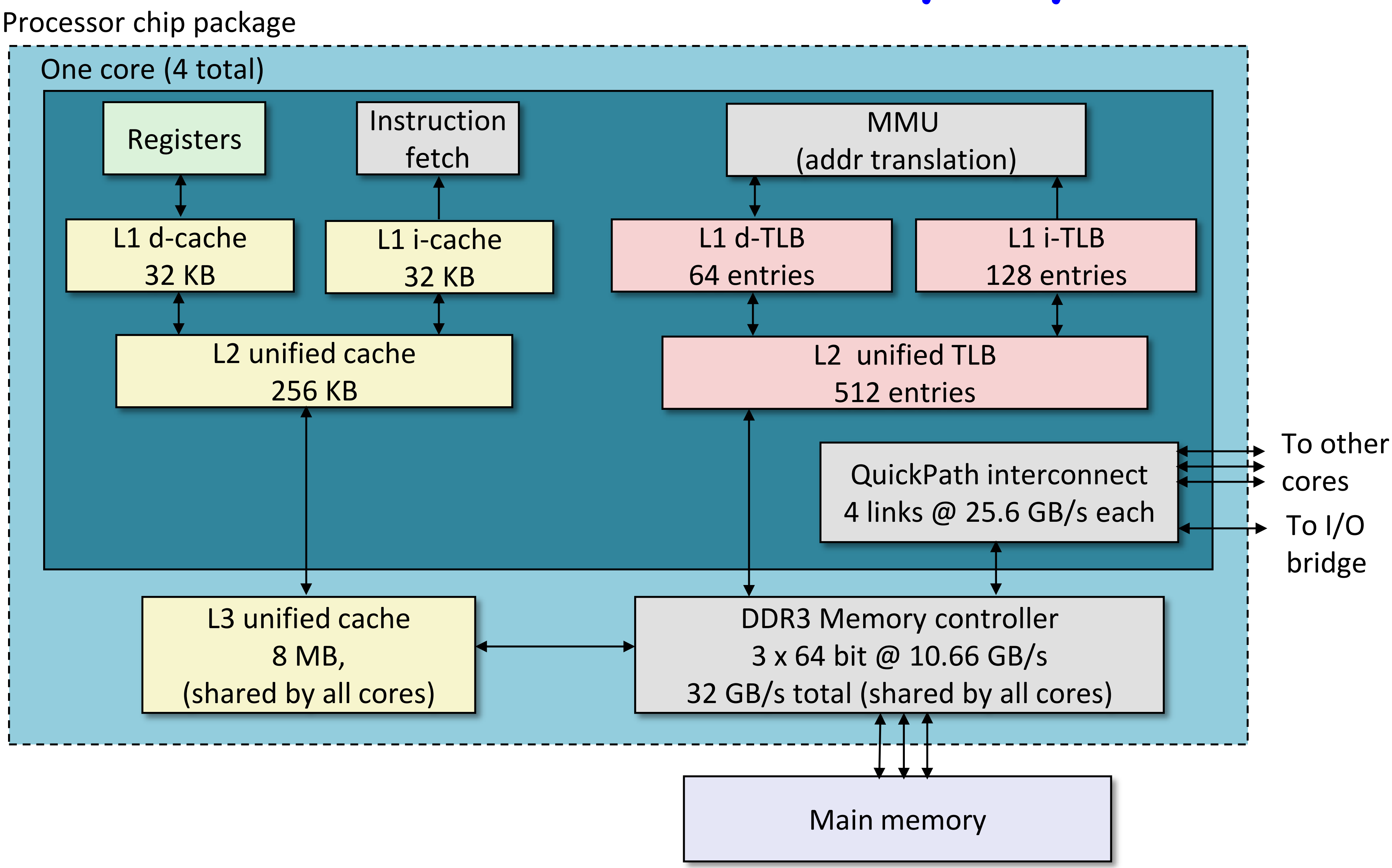
VPN	*PPN*	*Valid*
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

What’s the corresponding PPN? Physical address?

Case study: Core i7/Linux memory system (Nehalem microarchitecture)



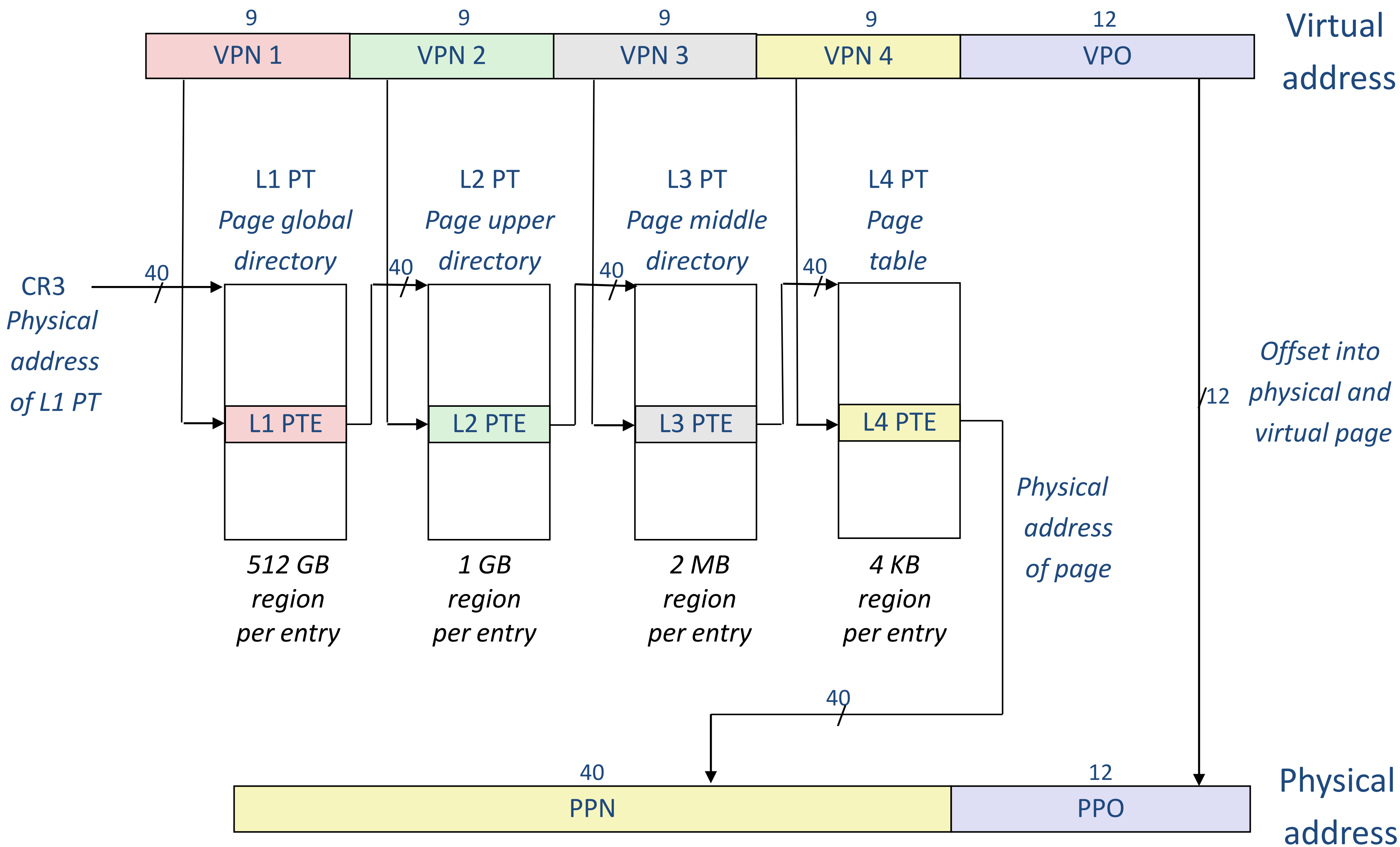
Intel Core i7 Memory System



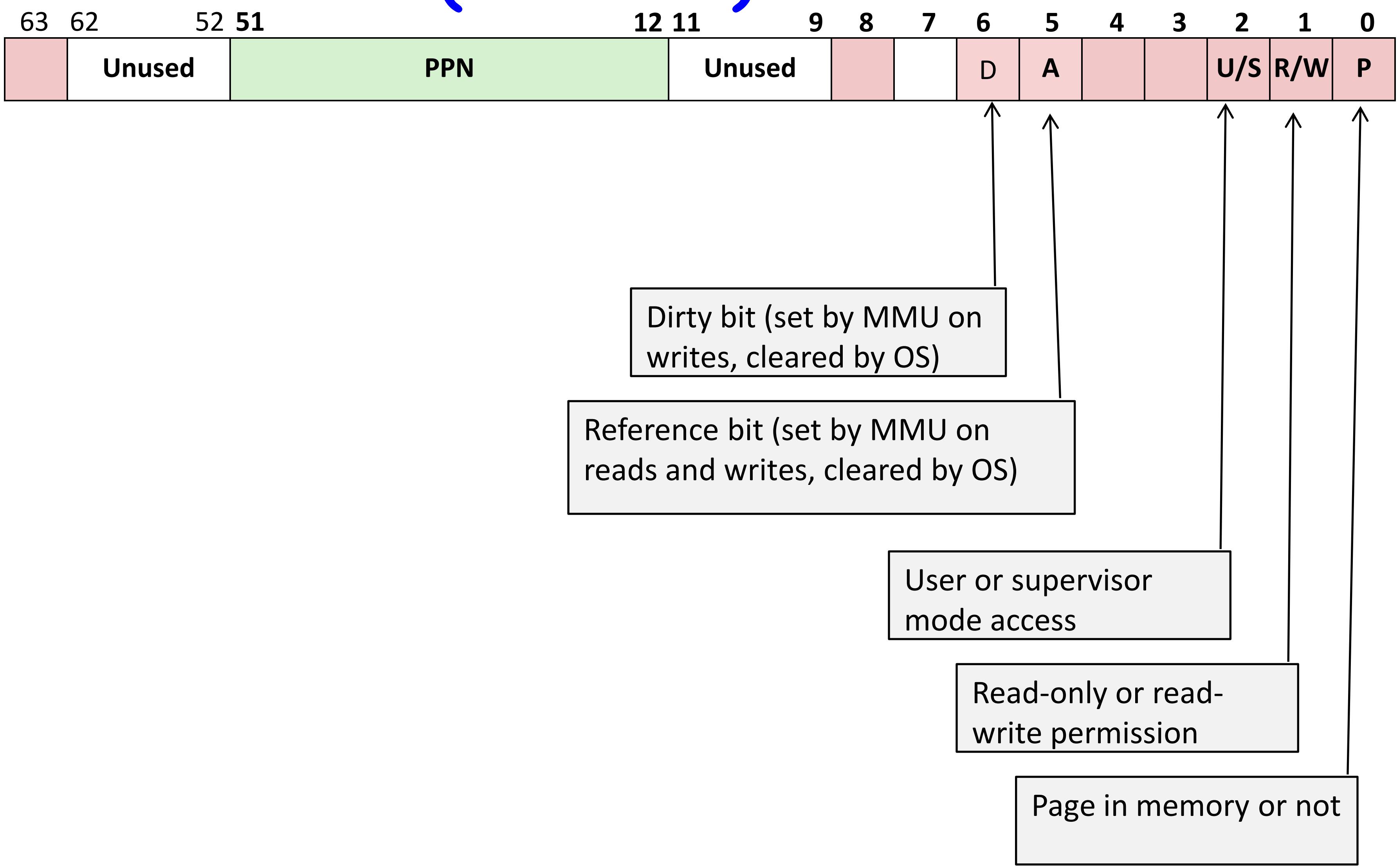
i7 Memory Hierarchy

- 48-bit virtual address
- 52-bit physical address
- TLBs are virtually addressed
- Caches are physically addressed
- Page size can be configured at start-up time as either 4KB or 4MB
  - Linux uses 4KB
- i7 uses 4-level page table hierarchy
- Each process has its own private page table hierarchy

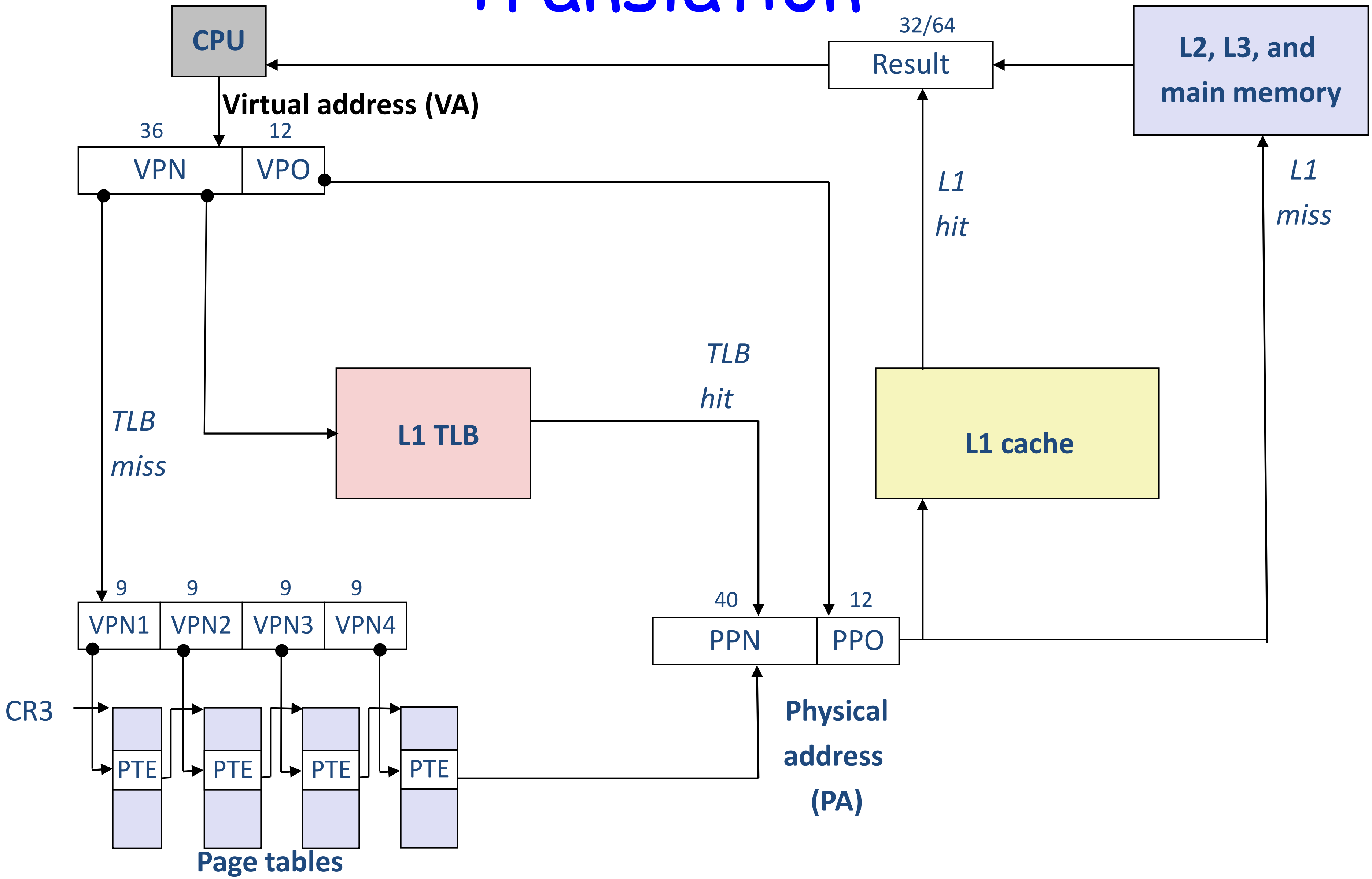
Core i7 Page Table Translation



Core i7 Page Table Entry (level-4)

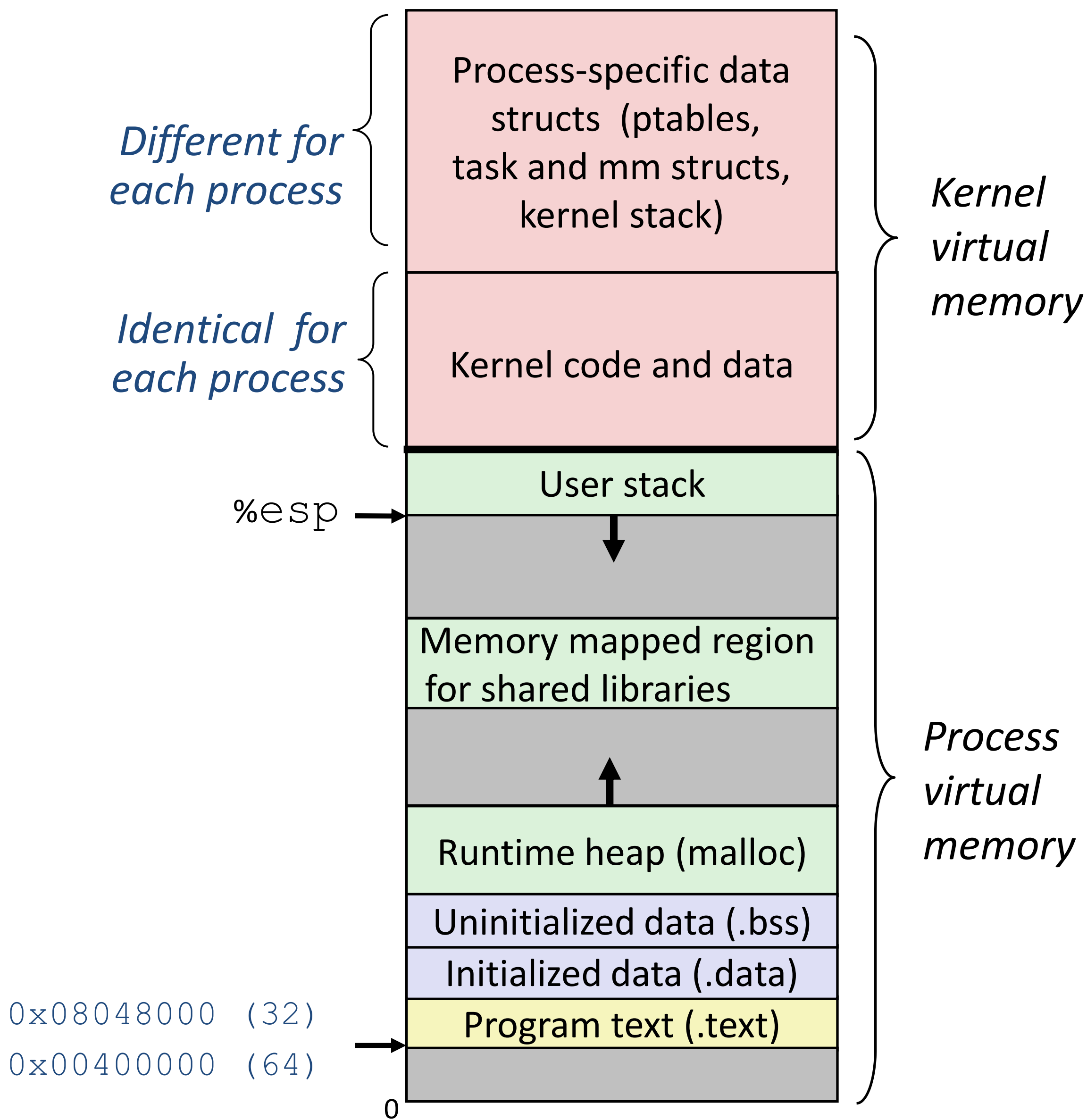


End-to-end Core i7 Address Translation

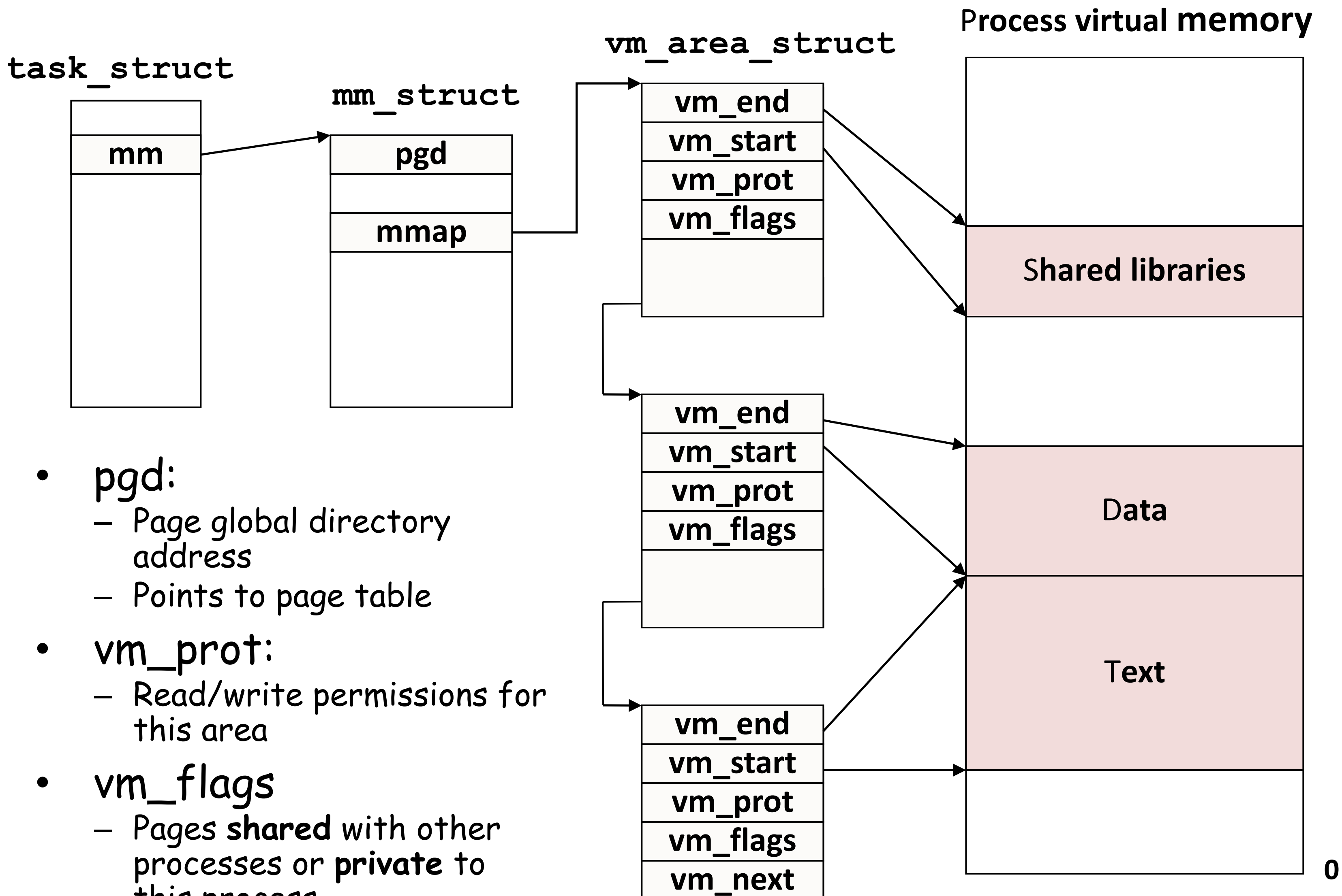


Memory mapping in Linux

Virtual Memory of a Linux Process

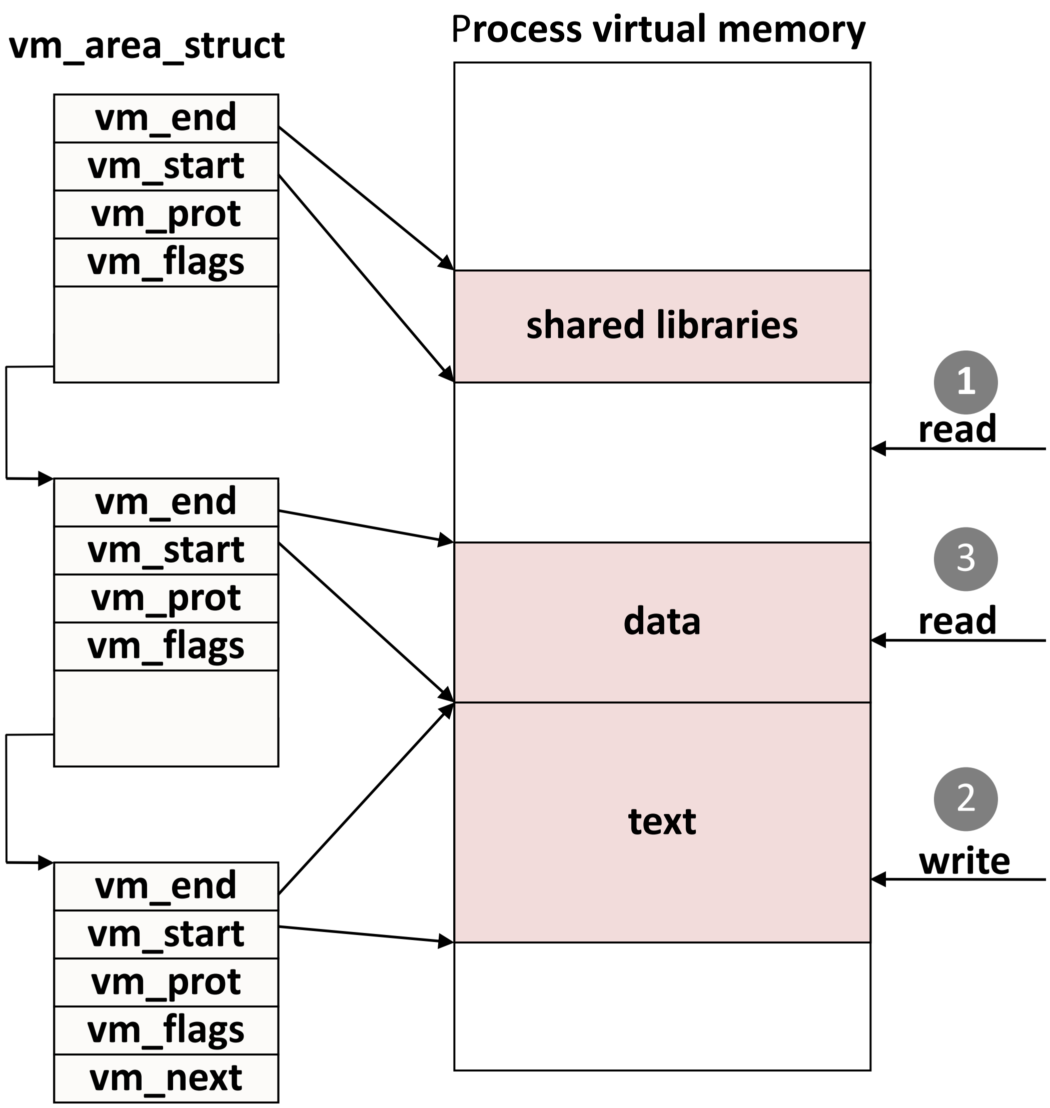


Linux Organizes VM as Collection of "Areas"



- pgd:
  - Page global directory address
  - Points to page table
- vm_prot:
  - Read/write permissions for this area
- vm_flags
  - Pages **shared** with other processes or **private** to this process

Linux Page Fault Handling



Segmentation fault:
accessing a non-existing page

Normal page fault

Protection exception:
e.g., violating permission by writing to a read-only page (Linux reports as Segmentation fault)

Memory Mapping

- VM areas initialized by associating them with disk objects.

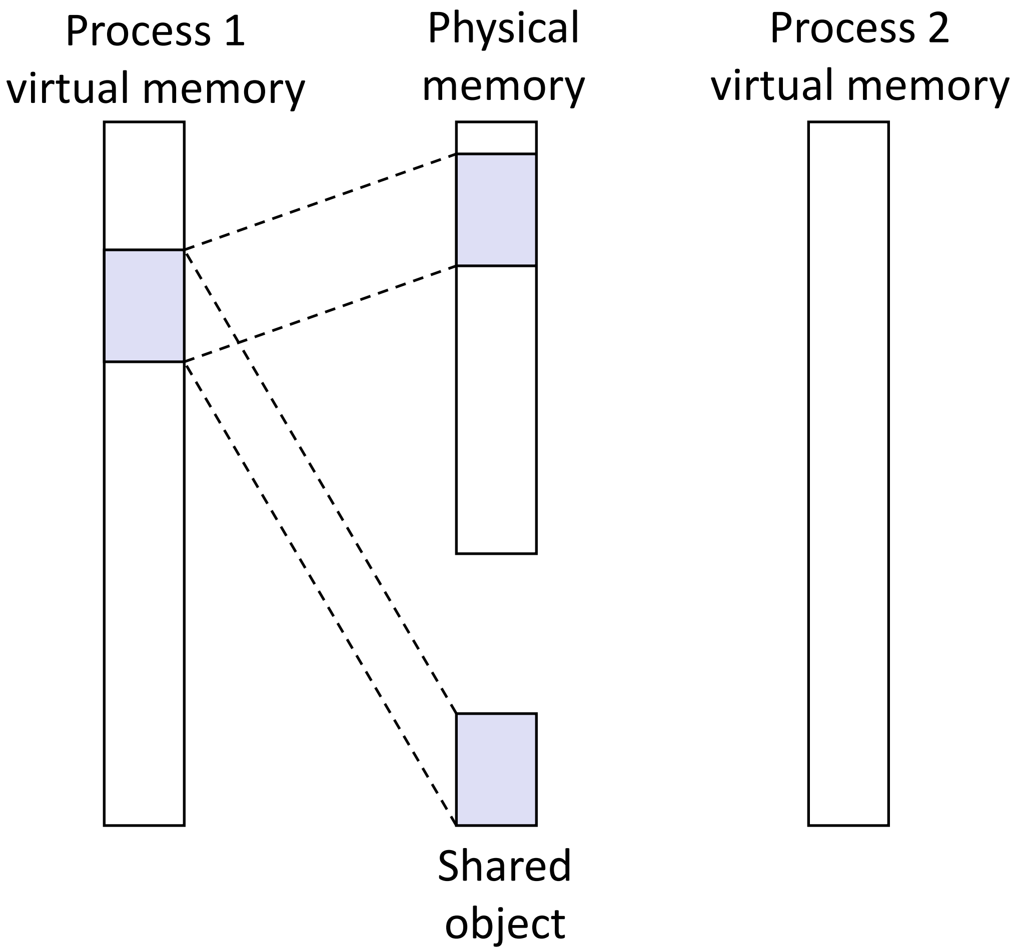
- Area can be backed by (i.e., get its initial values from) :
  - ***Regular file*** on disk (e.g., an executable object file)
    - Initial page bytes come from a section of a file
  - ***Nothing***
    - First fault will allocate a physical page full of 0's (***demand-zero page***)

- If a dirty page is kicked out from memory, OS copies it to a special *swap area* on disk

Demand paging

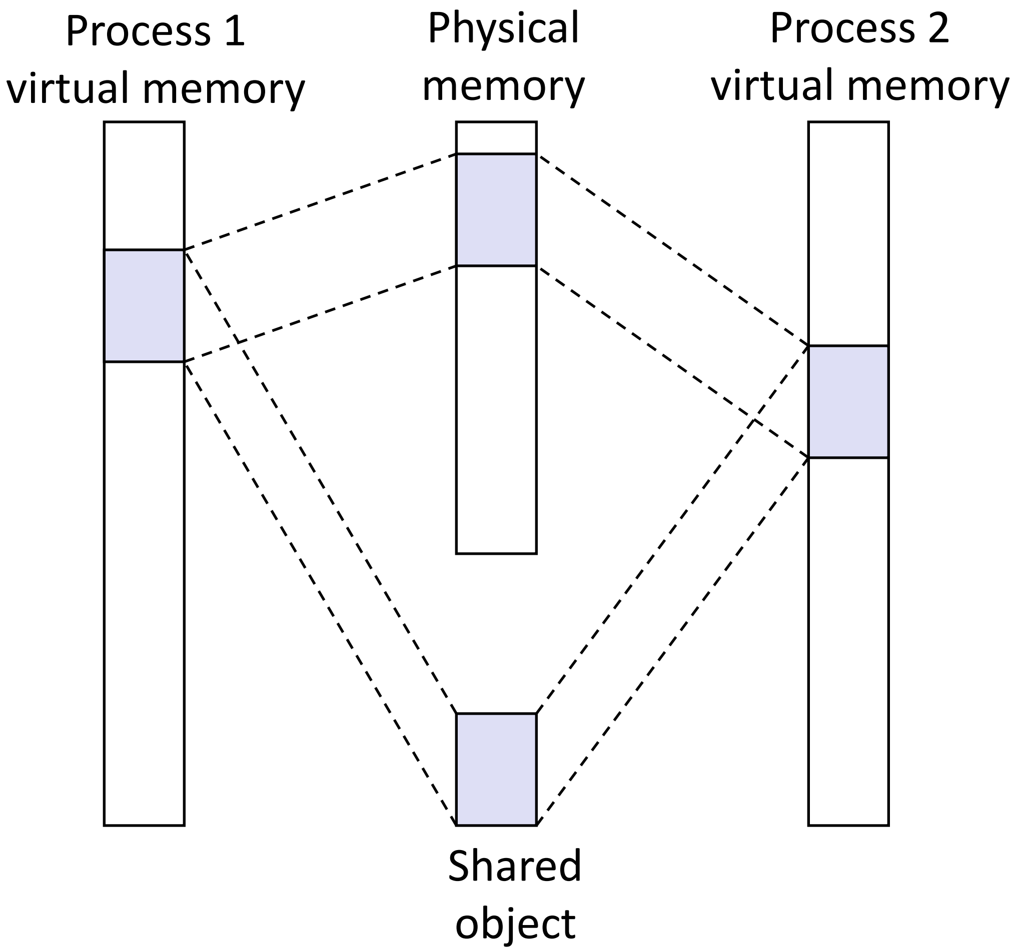
- *Key idea:* OS delays copying virtual pages into physical memory until they are referenced!
- Crucial for time and space efficiency

Sharing under demand-paging



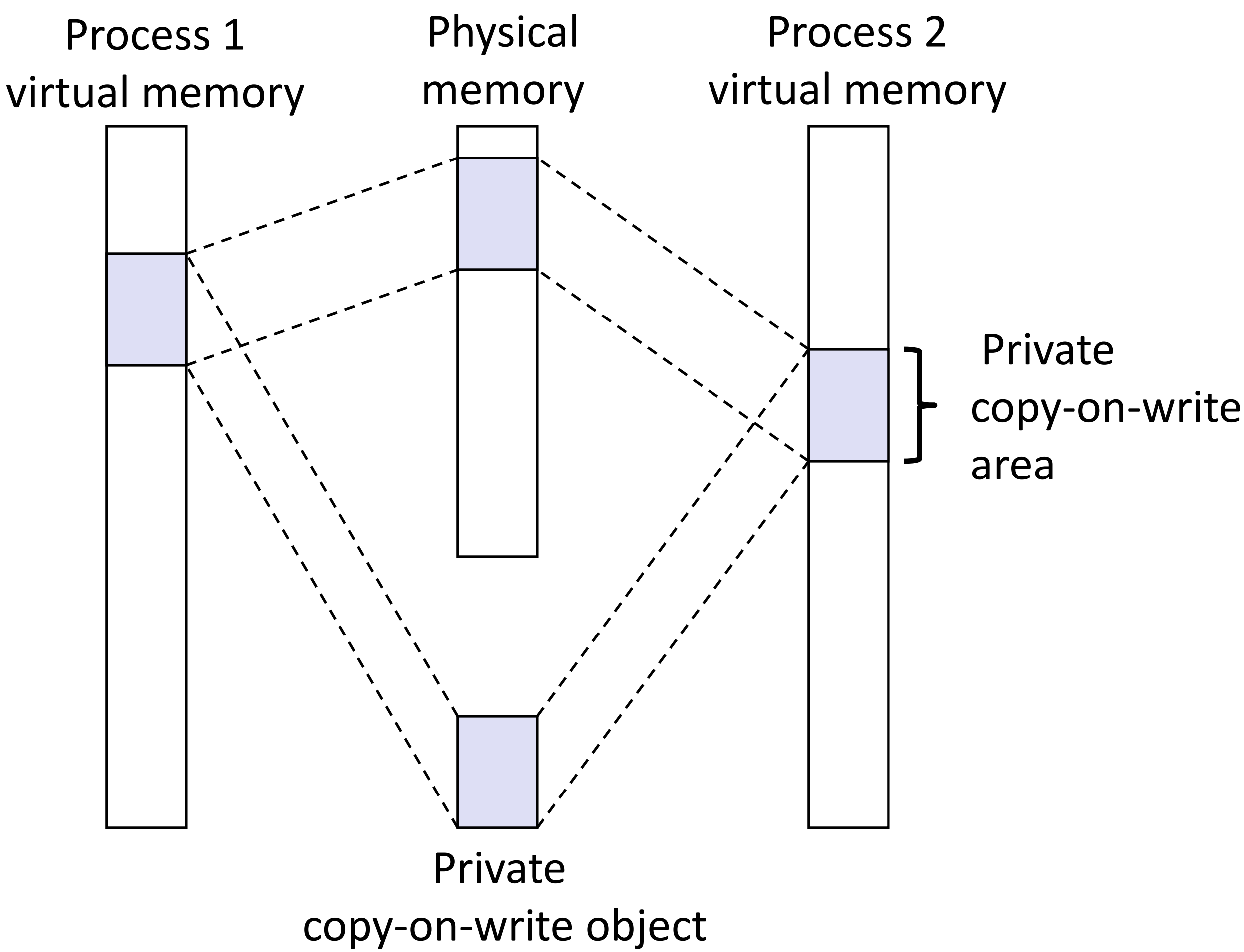
- Process 1 maps the shared object.

Sharing under demand-paging



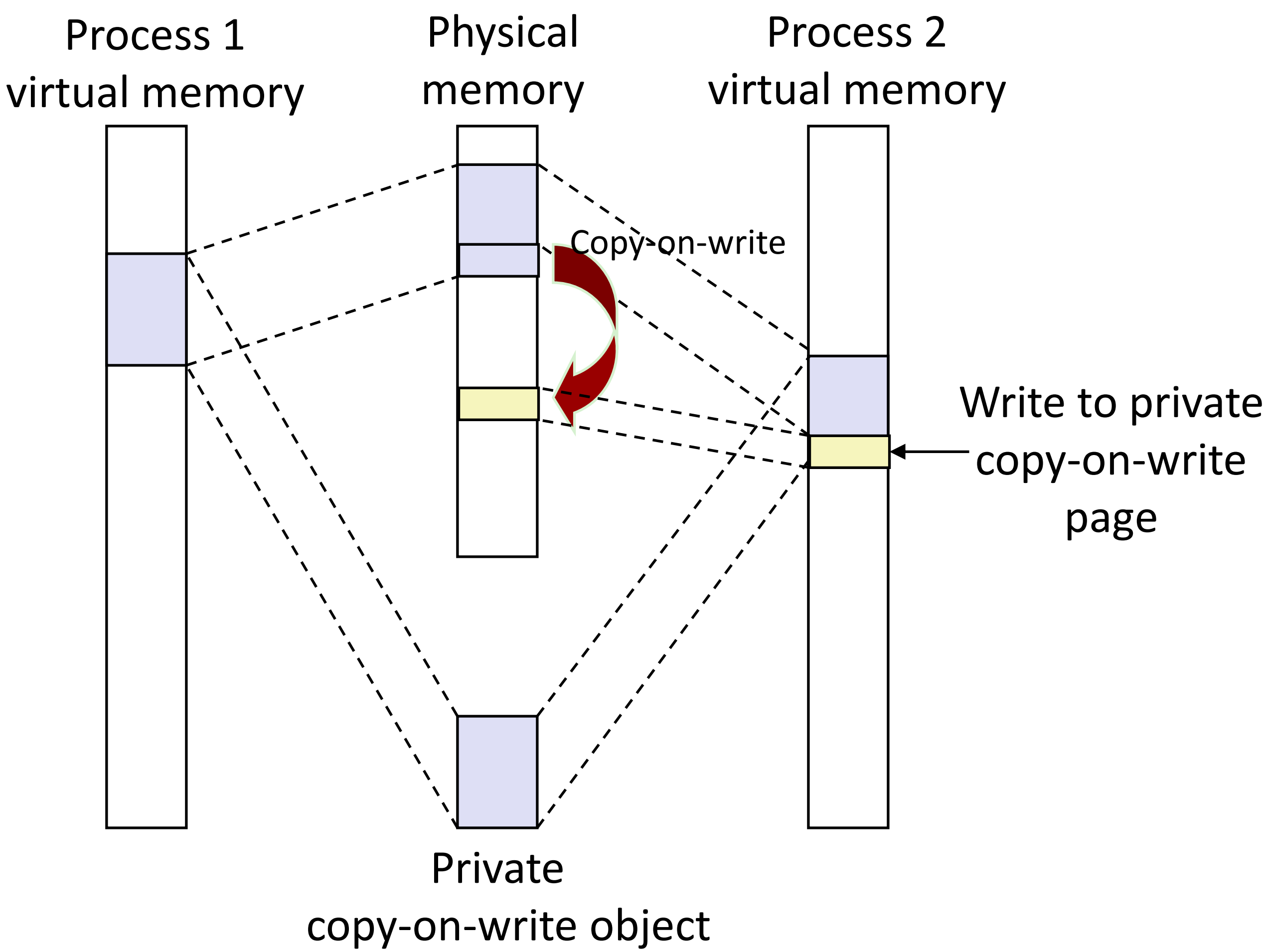
- **Process 2 maps the shared object.**
- Notice same object can be mapped to different virtual addresses

Sharing: 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: Copy-on-write (COW) Objects



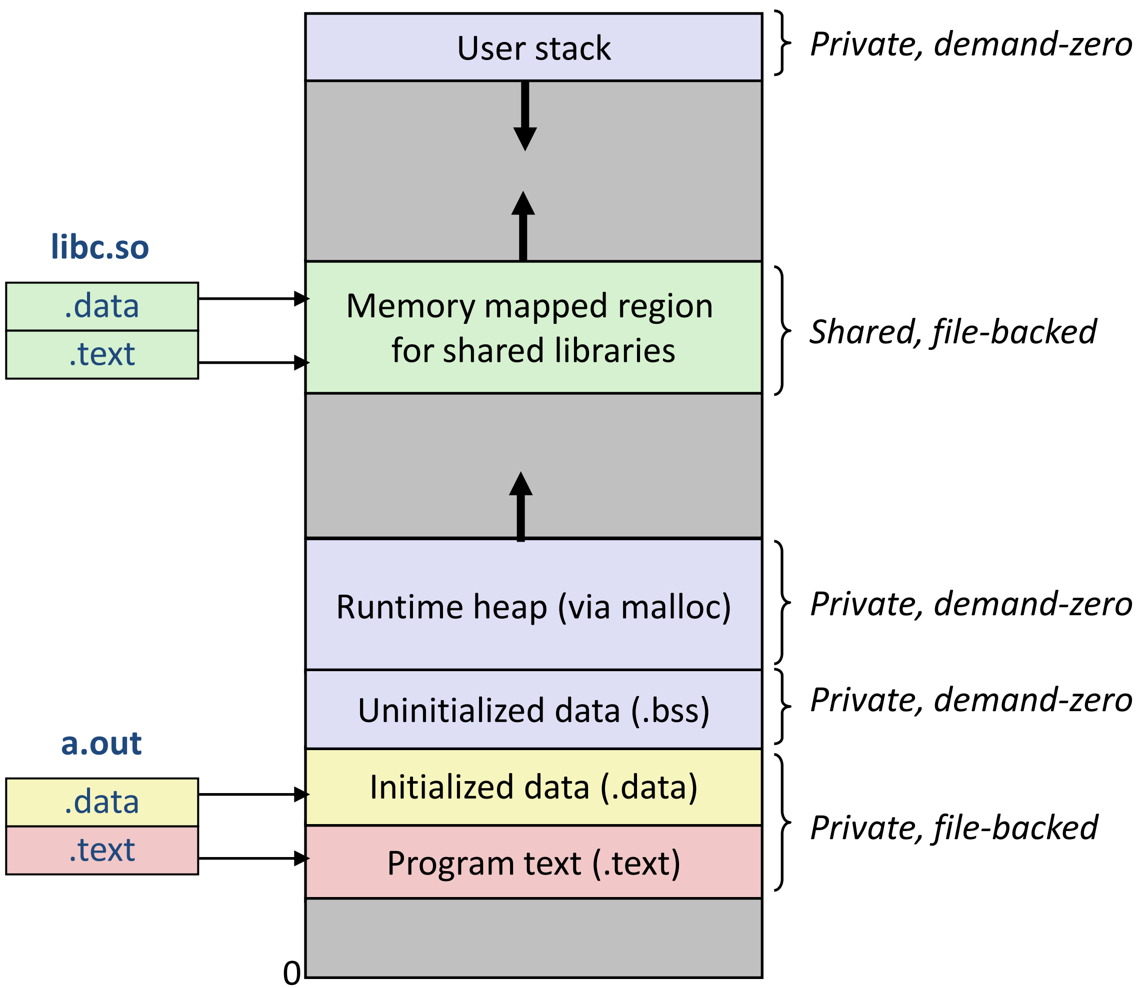
- Instruction writing to private page triggers protection fault.
- Handler creates new R/W page.
- Instruction restarts upon handler return.
- Copying deferred as long as possible!

Revisiting `fork`

- To create virtual address for new child process
  - Create an exact copy of parent's memory mapping for the child
  - Flag each memory area in both processes at COW and set each page in both processes as read-only

- Subsequent writes create new pages using COW mechanism.

Revisiting `execve`



- To load and run a new program `a.out` in the current process using `execve`:
  - Free old mapped areas and page tables
  - Create new mapped areas and corresponding page table entries
  - Set PC to entry point in `.text`
  - Subsequently, OS will fault in code and data pages as needed.

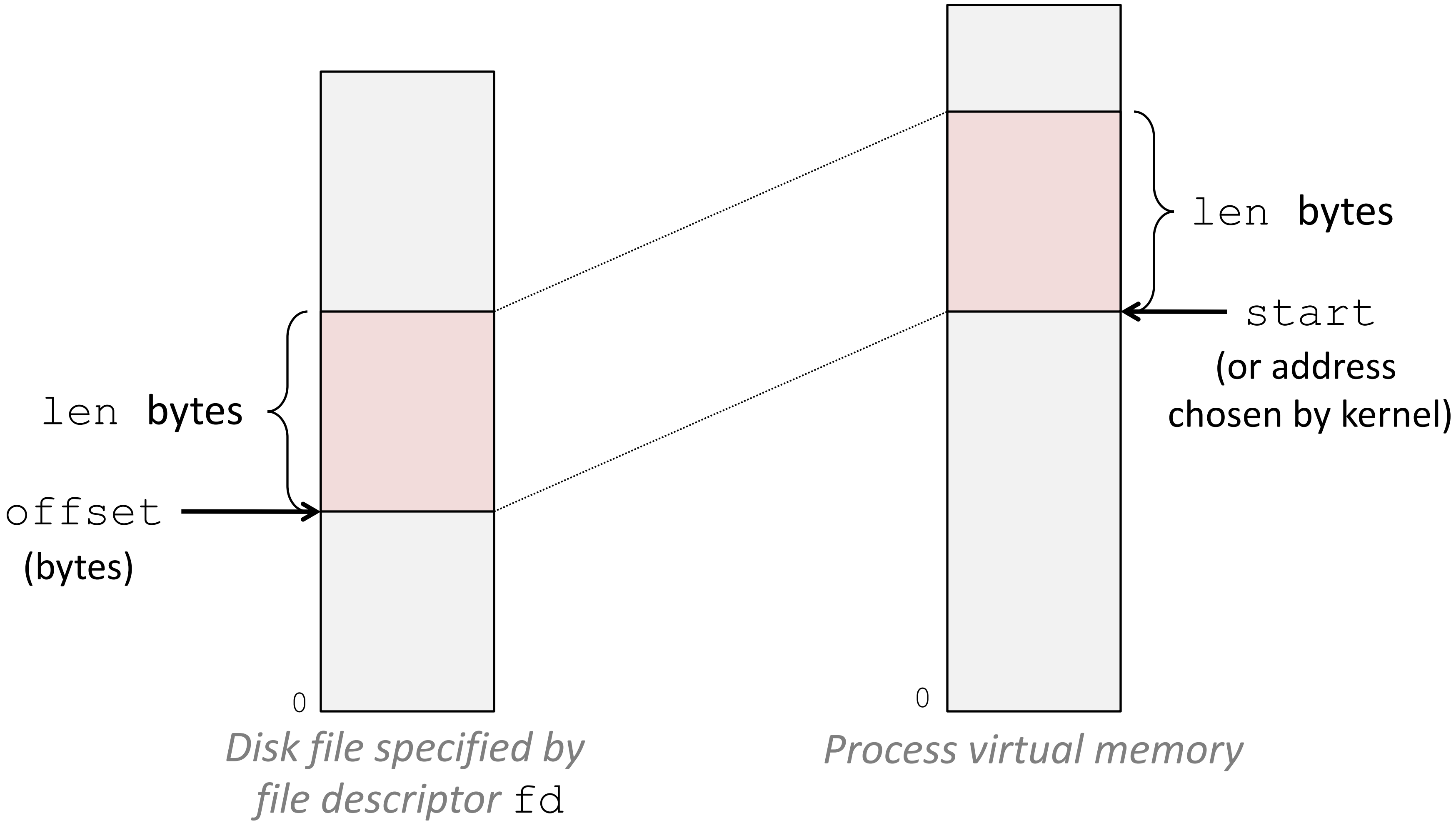
User-Level Memory Mapping

```
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

- Copying without transferring data to user space .

```
/*
 * mmapcopy - uses mmap to copy
 *            file fd to stdout
 */
void mmapcopy(int fd, int size)
{

    /* Ptr to mem-mapped VM 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 cmdline arg */
    if (argc != 2) {
        printf("usage: %s <filename>\n",
                argv[0]);
        exit(0);
    }

    /* Copy the input arg to stdout */
    fd = open(argv[1], O_RDONLY, 0);
    fstat(fd, &stat);
    mmapcopy(fd, stat.st_size);
    exit(0);
}
```

Conclusions

- In this lecture we have seen VM in action.
- It is important to know how the following pieces interact:
  - Processor
  - MMU
  - DRAM
  - Cache
  - Kernel