Operating System Design and Implementation

Lecture 12: Paging

Tsung Tai Yeh

Tuesday: 3:30 – 5:20 pm
Classroom: ED-302

Acknowledgements and Disclaimer

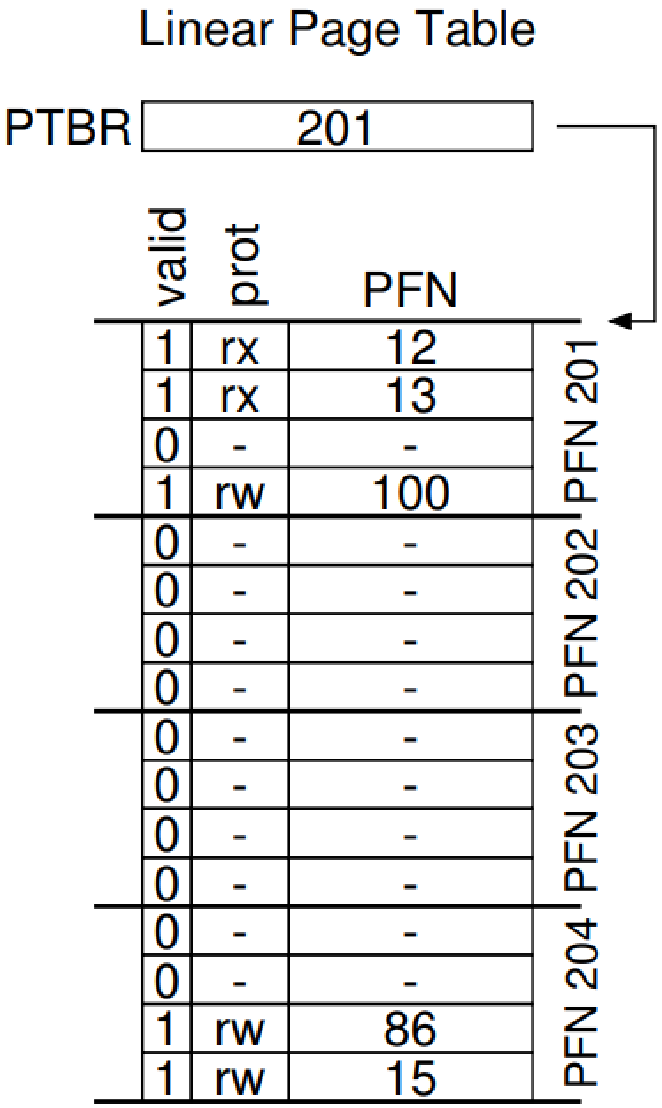
- Slides was developed in the reference with
  MIT 6.828 Operating system engineering class, 2018
  MIT 6.004 Operating system, 2018
  Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC
  Onur Mutlu, Computer architecture, ece 447, Carnegie Mellon University

Outline

- Multi-level page table
- Demand paging
- The inverted page table
- Page sharing

Page table (linear structure)

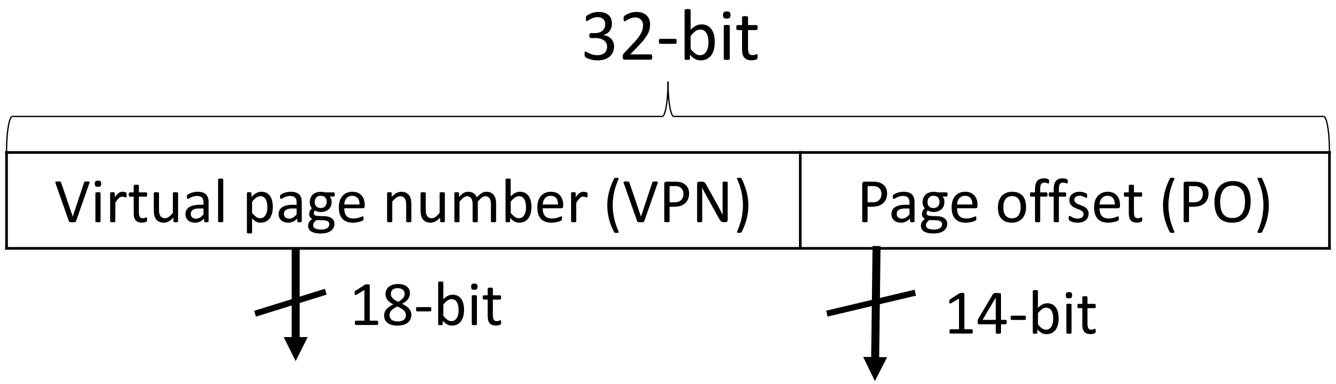
- Page table (linear structure) can be vary large !
  - 32-bit address (2^{32} bytes), 4KB (2^{12} bytes) pages, 4B PT entry
  - The number of page is ($2^{32}/2^{12} = 2^{20}$),
  - One page table size is 2^{20} x 4 bytes = 4MB per process
  - Hundreds of processes -> Hundreds of MB for PT



https://pages.cs.wisc.edu/~remzi/OSTEP/vm-smalltables.pdf

Bigger pages

- A 32-bit address, but increase page size from 4KB to 16KB
  - Each PTE is 4 bytes and now we have 2^{18} entries in our page table
  - The total size of a page table is (2^{18} x 4 bytes = 1MB)
  - The page table size (4MB in case 4 KB page size)
- What problems are shown with this approach ?
  - **Internal fragmentation** (big pages lead to waste within each page)



Page size

- **Arguments for larger page size**
  - Leads to a smaller page table
  - May be more efficient for disk access (block size of disk)
  - Larger page size – TLB entries capture more addresses per entry, so there are fewer misses, with the “right” locality
  - x86 page sizes: 4KB, 2MB, 4MB …
- **Arguments for smaller page size**
  - Conserve storage space – less fragmentation

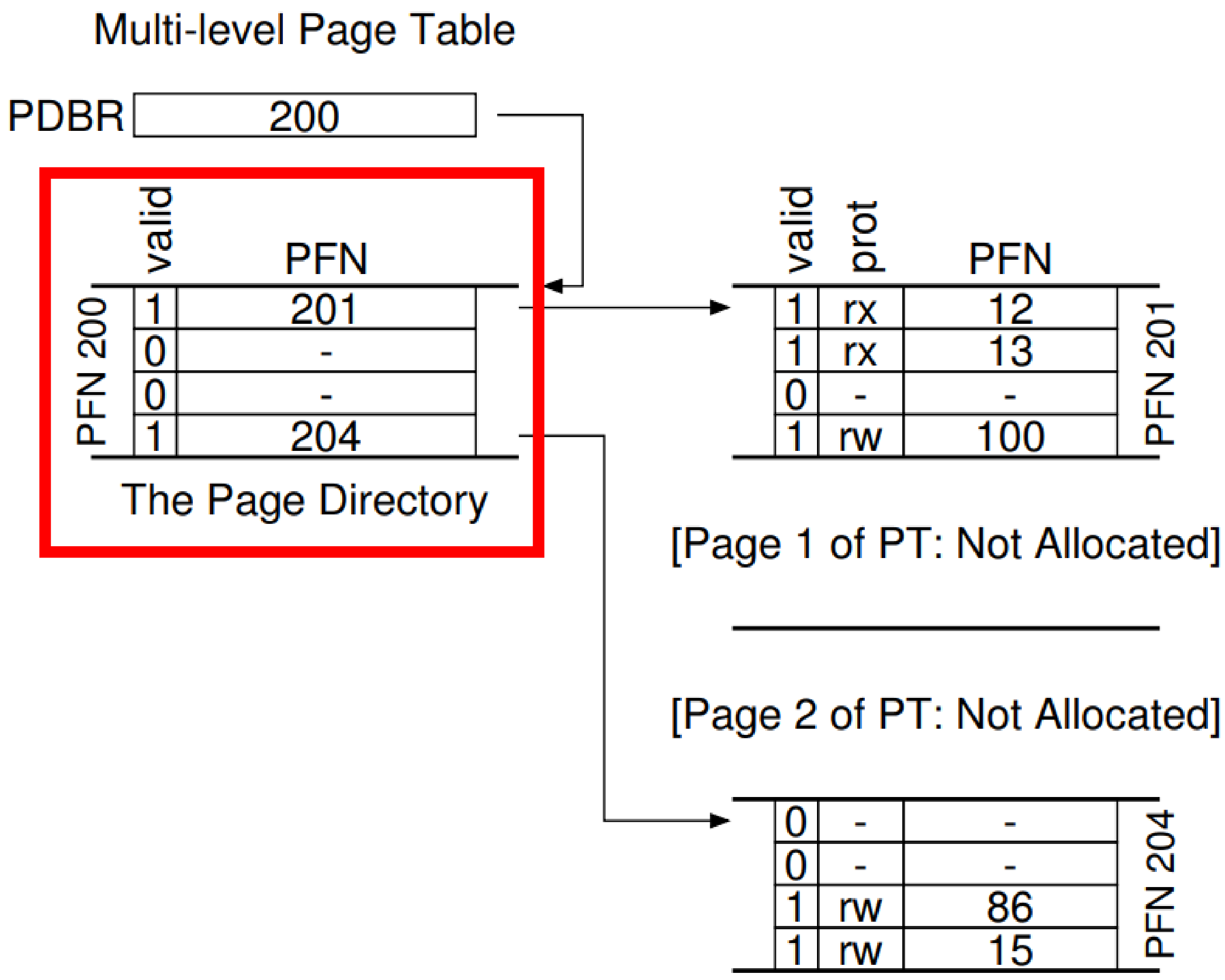
https://people.cs.pitt.edu/~childers/CS2410/slides/lect-virtual-memory.pdf

Multi-level page tables

- Turn page table into a tree (hierarchy) structure
  - Divide page table (PT) into page sized chunks
  - Hold only the part of PT where PT entries are valid
  - Directory points to portions of the PT
  - Directory says where to find PT or that chunk is invalid

Multi-level page table (cont.)

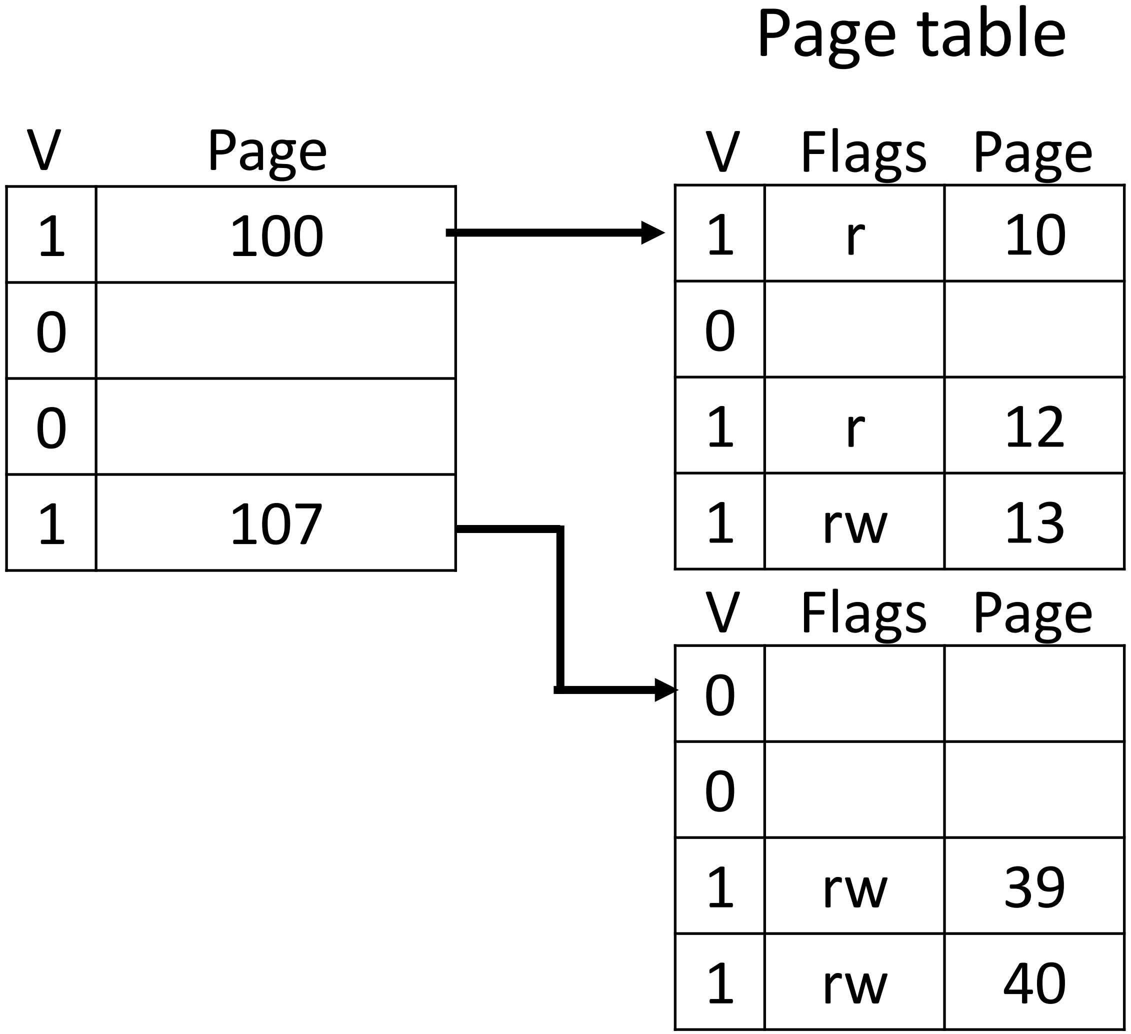
- **Multi-level page table**
  - Chop up the page table into page-sized units
  - **Page directory** tells where a page of the page table is
    - A number of **page directory entries (PDE)**
    - A **page frame number (PFN),** and a valid bit



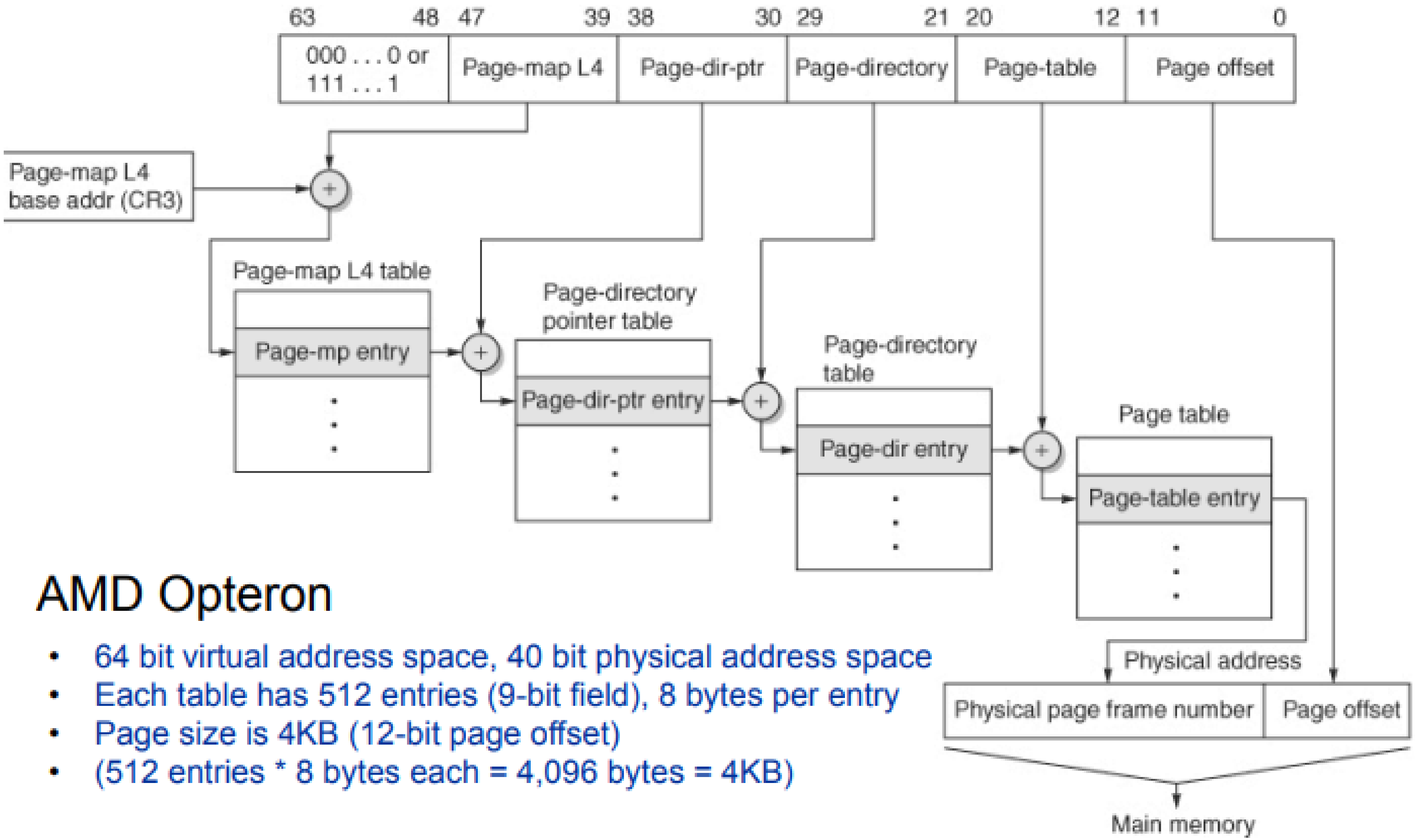
https://pages.cs.wisc.edu/~remzi/OSTEP/vm-smalltables.pdf

Multi-level page table (cont.)

- What are advantages of multi-level page table ?
  - Only allocate "using" page-table space
  - Compact and supports **sparse** address space



Multi-level paging table (cont.)



AMD Opteron

- 64 bit virtual address space, 40 bit physical address space
- Each table has 512 entries (9-bit field), 8 bytes per entry
- Page size is 4KB (12-bit page offset)
- (512 entries * 8 bytes each = 4,096 bytes = 4KB)

Important formula for paging table

- Number of entries in page table
  - (virtual address space size) / (page size) = number of pages
- Virtual address space size
  - 2^n Bytes
- Size of page table
  - (Number of entries in page table) x (size of PTE)

Case study of multi-level paging table

- How many levels of page tables would be required ?
  - A virtual memory system with physical memory of 8 GB, a page size of 8 KB, 46 bit virtual address, and PTE size is 4 B
- Initially
  - Page size = 8 KB = 2^{13} B
  - Virtual address space size = 2^{46} B
  - PTE = 4 B = 2^2 B
  - Number of pages or number of entries in page table = 2^{46} B / 2^{13} B = 2^{33}
  - Size of page table = 2^{33} x 2^2 B = 2^{35} B

Case study of multi-level paging table (cont.)

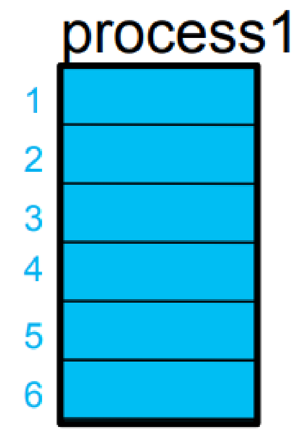
- How many levels of page tables would be required ?
  - A virtual memory system with physical memory of 8 GB, a page size of 8 KB, 46 bit virtual address, and PTE size is 4 B
- Now, size of page table > page size (2^{35} B > 2^{13} B)
  - Create one more level
  - Number of page tables in last level
    2^{35} B / 2^{13} B = 2^{22}
  - Size of page table [second last level]
  - 2^{22} x 2^{2} B = 2^{24} B

Case study of multi-level paging table (cont.)

- How many levels of page tables would be required ?
  - A virtual memory system with physical memory of 8 GB, a page size of 8 KB, 46 bit virtual address, and PTE size is 4 B
- Now, size of page table > page size (2^{24} B > 2^{13} B)
  - Create one more level [third last level]
  - Number of page tables in second last level
    = 2^{24} B / 2^{13} B = 2^{11}
  - Size of page table [third last level]=
    = 2^{11} x 2^{2} B = 2^{13} B = page size

Virtual memory

- Do we need to load all blocks into memory before the process starts executing ?
  - **No !!**
  - Some code may not even be executed
- How to reduce the loading of unnecessary pages ?



process page table

block	page frame
1	14
2	2
3	13
4	4
5	1
6	8

http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/4_Memory.pdf

Demand paging

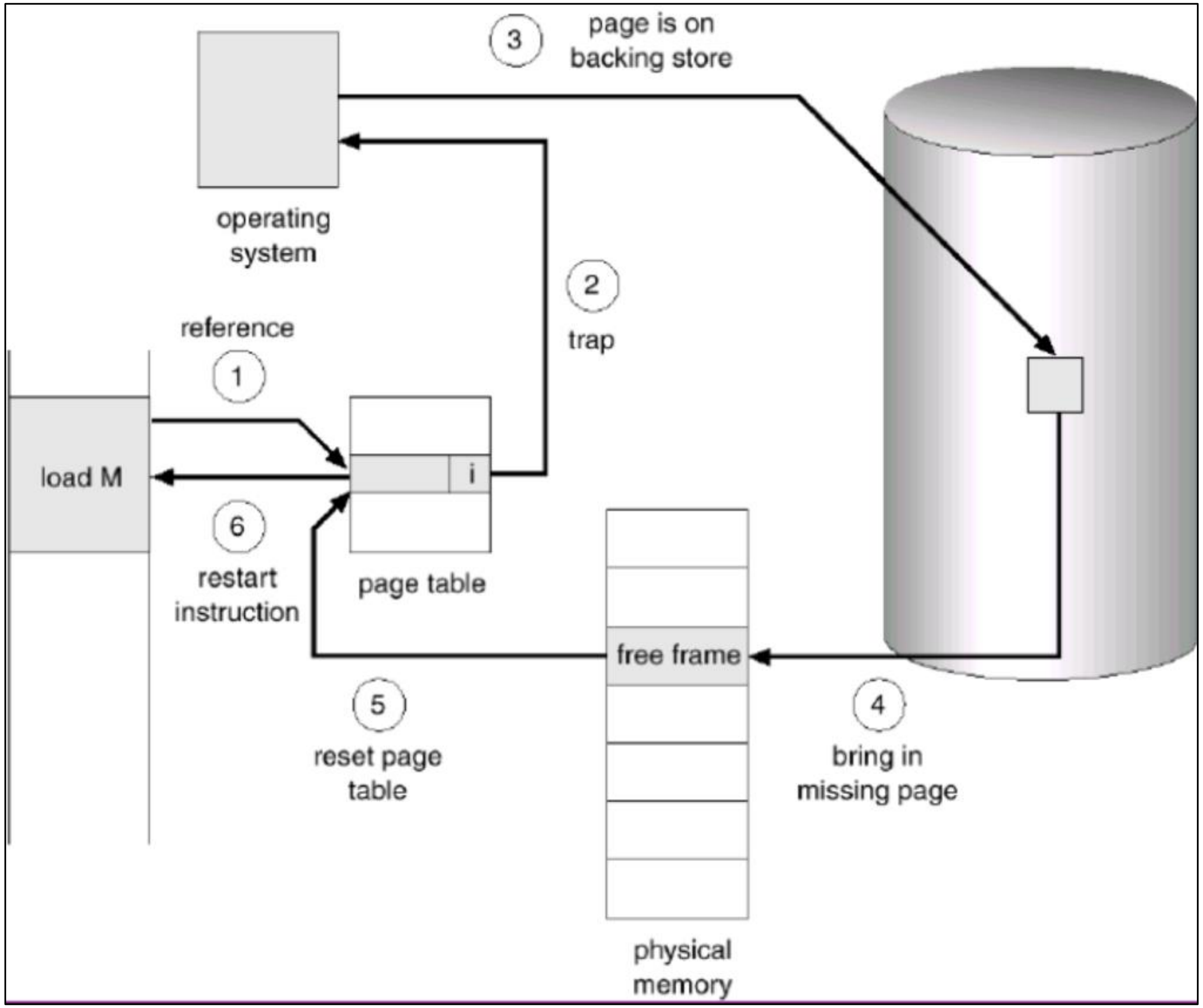
- **Demand paging**
  - Pages are loaded from disk to RAM, **only when needed**
- **Why demand paging ?**
  - Reducing I/O
  - More users and decrease the number of memory requests
- **Pure demand paging**
  - When no pages are loaded into memory initially, pages generates page faults. No prediction !!
- **Pre-paging**
  - Predict which pages will be used and swap them into the RAM

Demand paging

- **How does demand paging work ?**
  - Using **present bit** in process page table to indicate if the block is in RAM or not
  - The process issues a page fault interrupt if an accessed page is not present in RAM
  - The OS loads the page into RAM and mark the present bit to 1
  - The OS removes another block from RAM if no pages on RAM are free

Page fault

- **Steps of page fault**
  - OS check page table referenced by process control block (PCB)
    - Abort -> invalid reference
    - Continue -> not in memory
  - Find an empty frame
  - Swap a page from disk
  - Reset page table valid bit
  - Restart instruction



https://hackmd.io/@Pl-eQT9CQaS0jhExKqL8_w/ryp5XonRZ?type=view

Timeline of a page fault

Initializing

1. Trap to operating system
2. Save state in PCB

Page fault handling

3. Vector to page fault handler
4. Check in page table if fault address is truly invalid
5. Find/create free page frame possible involves disk write

Disk processing

6. Issue disk read for page
   a. Wait until request queued at disk controller
   b. Wait for seek/latency
   c. Wait for data transfer (DMA)

OS scheduling

7. Schedule other processes / threads while waiting
8. Take disk interrupt
9. Update page table
10. Add process to run queue
11. Wait for process to be scheduled next
12. Restore state from PCB
13. Return from OS

Demand paging implementation

- Keep copy of process's pages on disk, some are in RAM
  - Extend page table to include **"present" and "valid"**
  - Access to "page out" data by using **trap**
- **Inside the trap handler**
  - Access pages that are temporarily paged out to disk
  - Allocate a physical page frame to hold contents (might require another page to be paged out)
  - Copy data from disk to allocated page frame **(slow!!)**
  - Update page table entry
  - OS schedules processes

Effective access times (EAT)

- **EAT** is used to **measure the performance of demand paging**
- **Parameter**
  - p: page fault rate; ma: memory access time; pft: page fault time
  - EAT: (1 – p) x ma + p x pft
- Discussion
  - The EAT is proportional to page fault time

Effective access times (EAT)

- **What is average access latency ?**
  - L1 cache: 2 cycles
  - L2 cache: 10 cycles
  - Main memory: 150 cycles
  - Disk: 10 ms -> 30, 000, 000 cycles on 3.0 GHz processor
- **Assume access having following characteristics:**
  - 98% handled by L1 cache
  - 1% handled by L2 cache
  - 0.99% handled by DRAM
  - 0.01% cause page fault
  - What’s the average access latency ?

Effective access times (EAT)

- **Assume access having following characteristics:**
  - 98% handled by L1 cache
  - 1% handled by L2 cache
  - 0.99% handled by DRAM
  - 0.01% cause page fault
  - What's the average access latency ?
- **Average access latency:**
  - (0.98 x 2) + (0.01 x 10) + (0.99 x 150) + (0.0001 x 30,000,000) = 1.96 + 0.1 + 1.485 + 3000 = about 3000 cycles / access
- Need **LOW** page fault rates to sustain performance !!

More issues

- **Page selection policy**
  - When do we load a page ?
- **Page replacement policy**
  - What pages do we swap to disk to make room for new pages ?
  - When do we swap pages out to disk ?

Page selection policy

- **Demand paging**
  - Load page in response to access (page fault)
- **Pre-paging (prefetching)**
  - Predict what pages will be accesses in near future
  - Prefetch pages in advance of access
  - Problems
    - Hard to predict accurately
    - Mispredictions can cause useful pages to be replaced

Page replacement policies

- **Random**
- **FIFO** (first in, first out)
  - Throw out oldest pages
- **Optimal**
  - Throw out page used farthest in the future
- **LRU** (least recently used)
  - Throw out page not used in the longest time
- **NFU** (not frequently used)
  - Do not throw out recently used pages

Demand paging issues

- **Performance**
  - Need lots of locality -> otherwise run at disk speeds
    - If most accesses are to data already in DRAM -> great !
    - **Spatial locality**: often access "nearby" addresses
    - **Temporal locality:** Often re-access same addresses again and again
  - How to resume a process ?
    - Re-execute instruction? Only if no side effects !
  - Run other processes / threads while serving the page fault

Belay’s Anomaly

- Belay’s anomaly
  - For some replacement algorithms
  - **MORE** pages in main memory can lead to **MORE** page faults !!
- **Example:**
  - FIFO replacement policy
  - Reference string: A B C D A B E A B C D E
  - Three pages -> 9 faults
  - Four pages -> 10 faults
  - Adding more memory might not help for page faults in some replacement algorithms

Thrashing

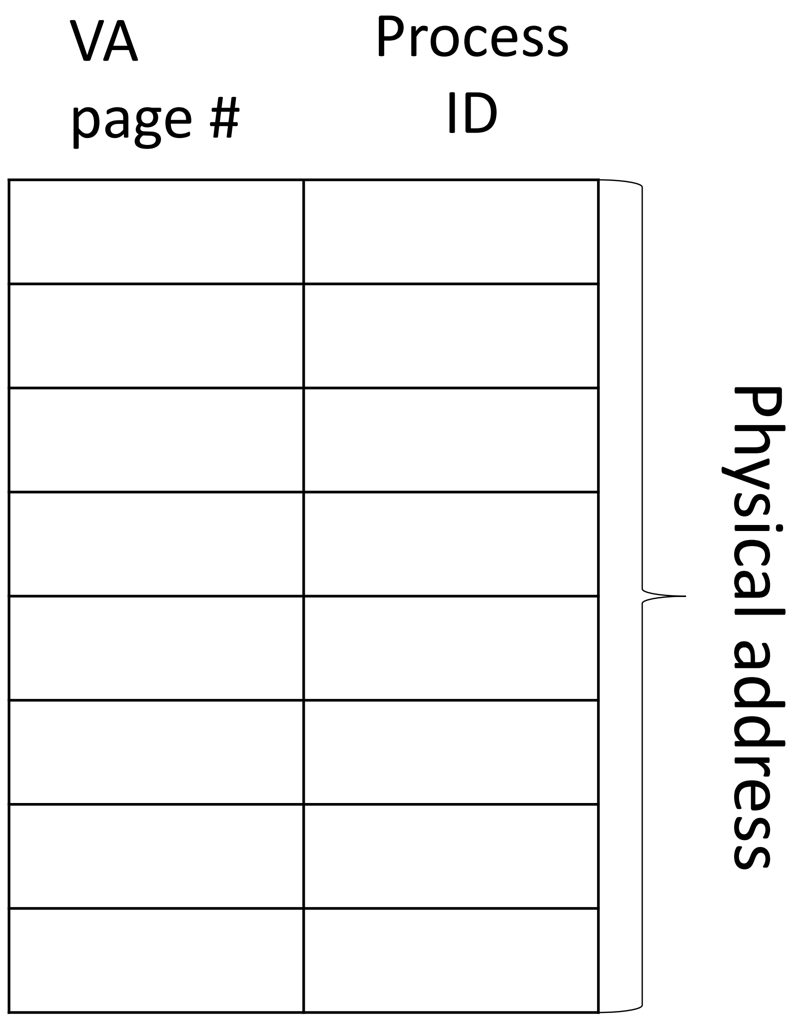
- **Working set**
  - Collection of memory currently being used by a process
- **Thrashing**
  - If all working sets do not fit in memory
  - One “hot” page replaces another
  - Percentage of accesses that generate page faults skyrockets
- **Typical solutions**
  - “swap out” entire processes
  - Invoked when page fault rate exceeds some bound
  - Linux invokes the out-of-memory (OOM) killer

Inverted page tables

- **Multi-level page table**
  - The number of levels is increased as the size of virtual memory address space grows
  - Given 64-bits address space, 4-KB page size, a PTE of 4 bytes, each page table can store 1024 entries
  - 6 (ceil(52/10)) levels are required
  - 6 memory accesses for each address translation
- Observation of the inverted page table
  - Size of physical memory is much smaller

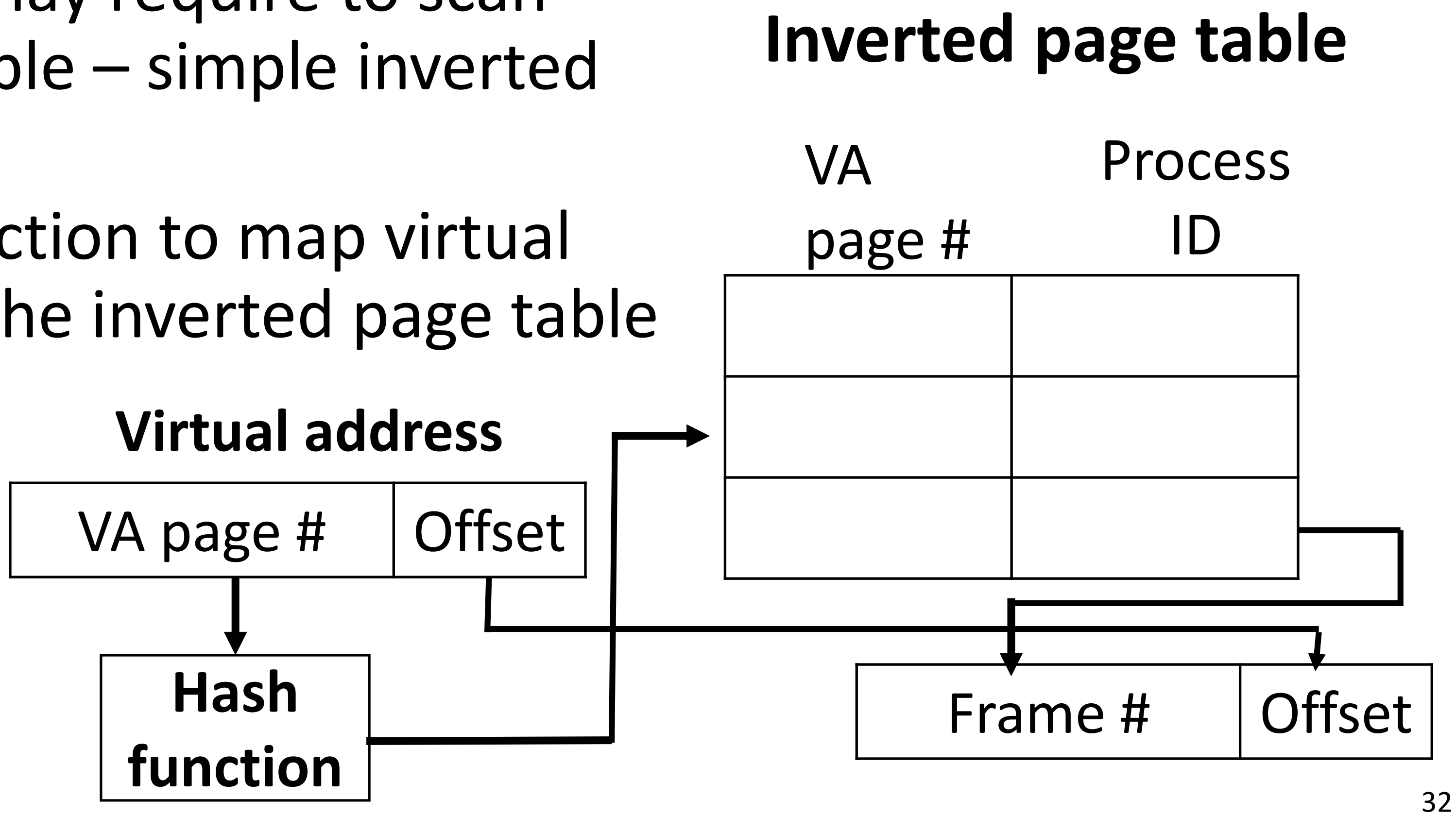
Inverted page table

- **Simple Inverted page table**
  - **The number of page table entries (PTE) = the number of physical frames**
  - Each PTE contain the pair <process ID, virtual page #>
  - Translate a virtual address, compare each <process ID, virtual page #> against each entry
  - If a match is found, the inverted page table index is used to obtain physical address



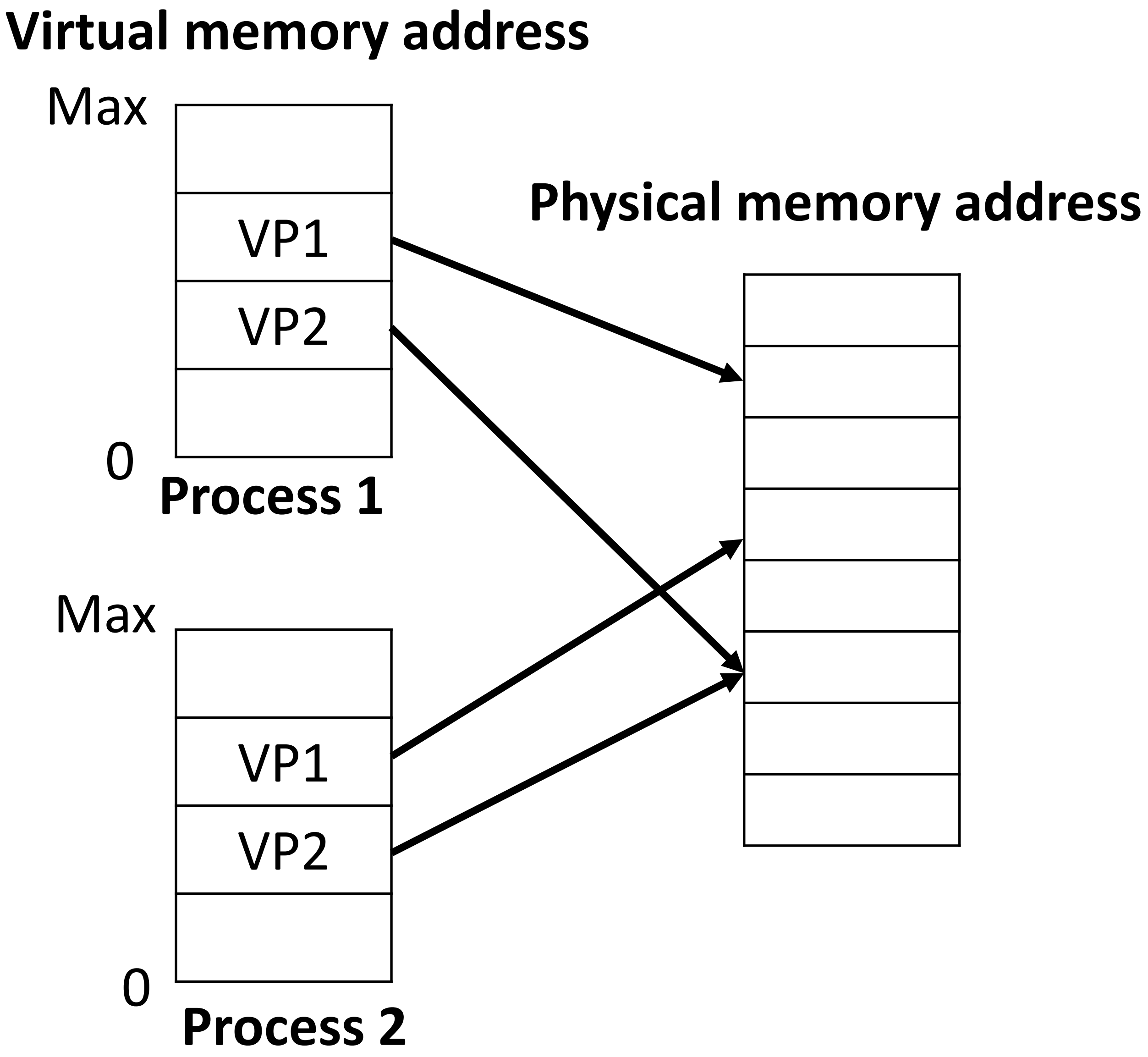
Inverted page table

- **Hashed inverted page table**
  - Finding a match may require to scan through entire table – simple inverted page table
  - Using hashed function to map virtual page # to PTE of the inverted page table

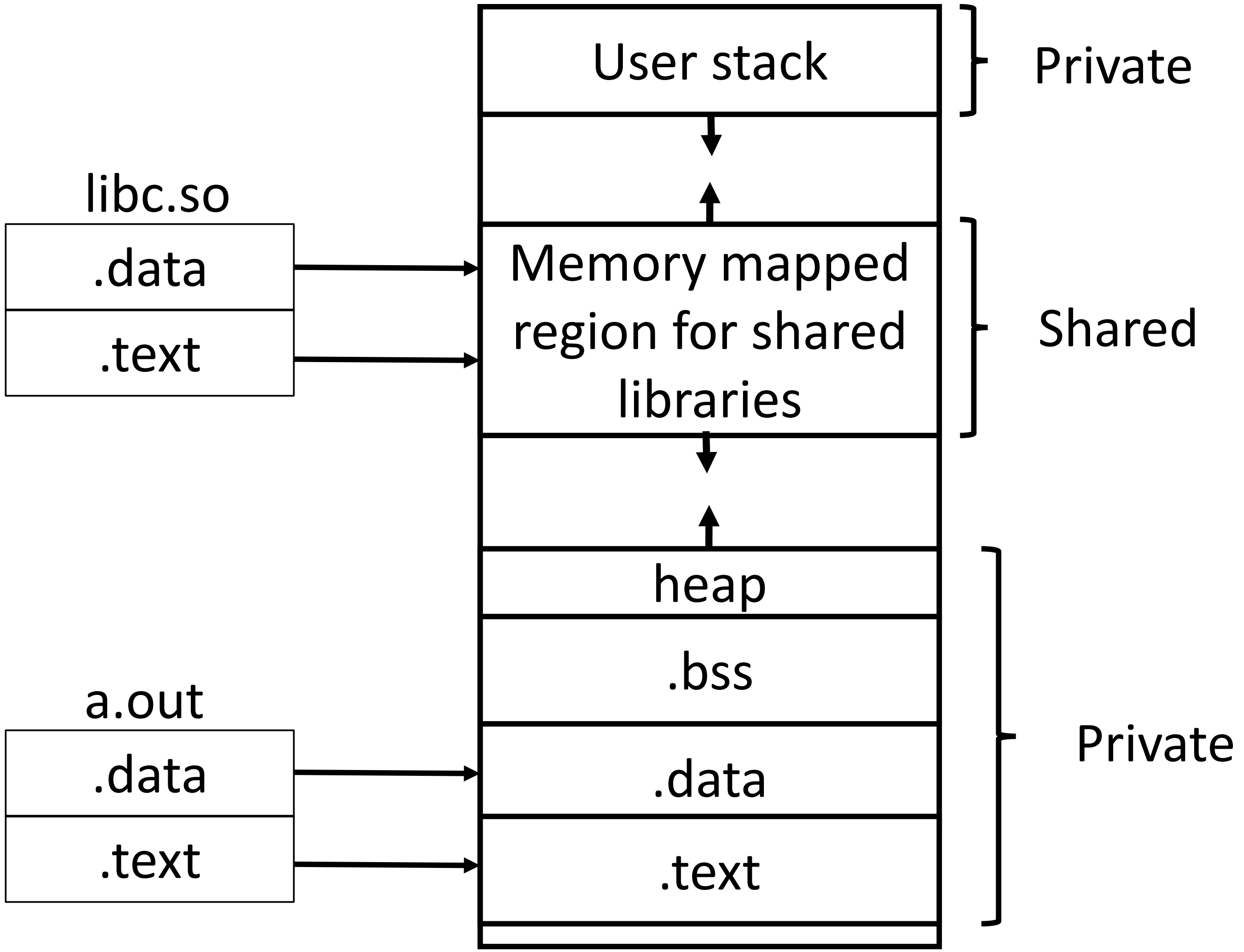


Sharing pages

- How to share memory ?
  - Entries in different process page tables map to same PPN
  - Each process can have its own registers and data
  - There can be only one copy of data kept in physical memory such as shared library
  - Each process's page table maps onto the same PPN



Process virtual memory



mmap()

- mmap()
  - Used to save memory
  - Two processes read shared read-only data from same pages in memory
  - Bypass (expensive) read, write or ioctl calls
  - Creates a more CPU-efficient mechanism for reading and writing files

mmap()

- Get a virtual address that can write to or read from

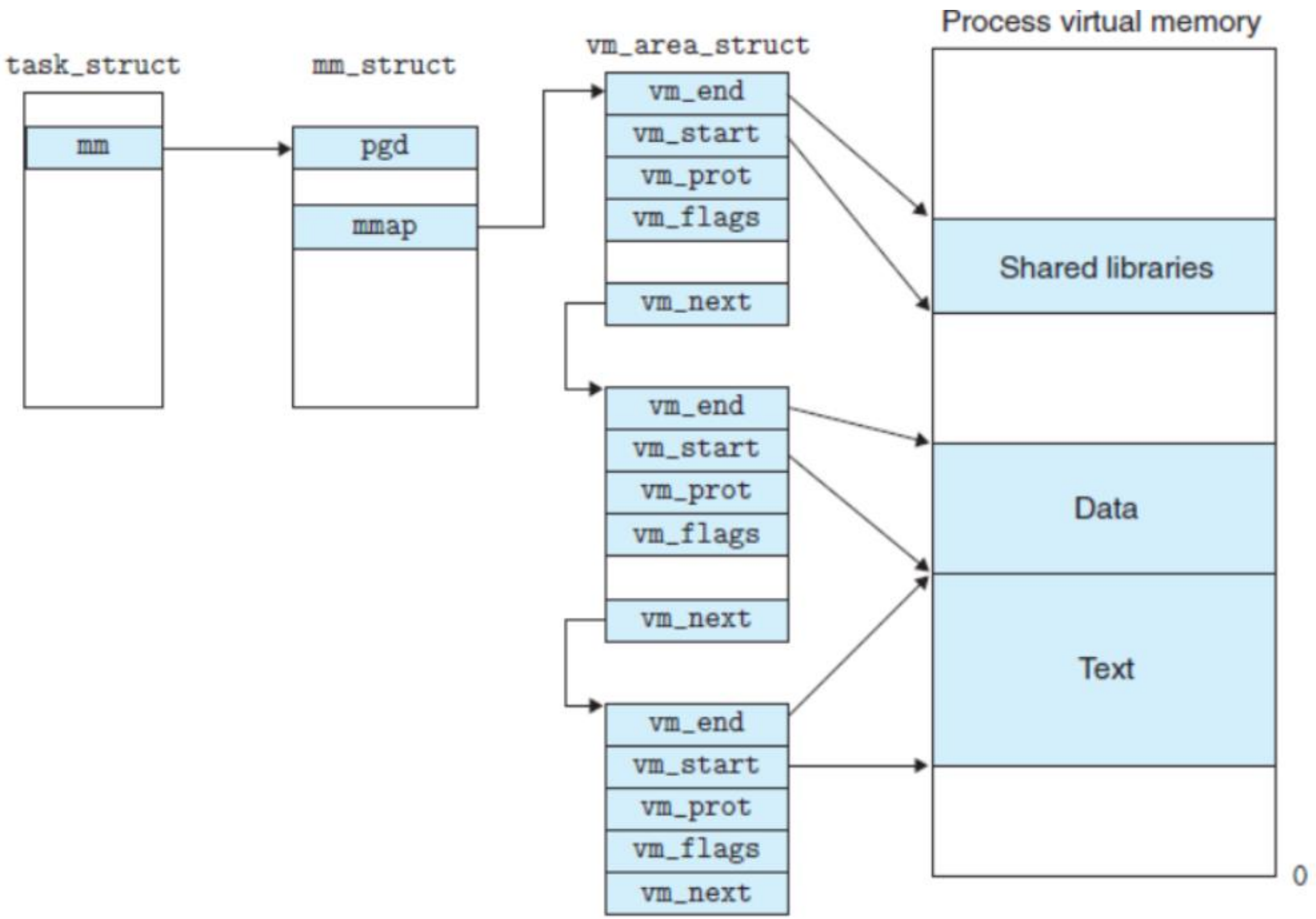
```
void *mmap (
        void    *start,     /*Often 0, preferred starting address*/
        size_t  length,     /*Length of the mapped area*/
        int     prot,       /*Permissions: read, write, execute*/
        int     flags,      /*Options: shared mapping, private copy …*/
        int     fd,         /*Open file descriptor*/
        off_t   offset      /*Offset in the file*/
);
```

flag: **MAP_SHARED**

Share this mapping. Update to the mapping are visible to other processes mapping the same region

vm_area_struct

- **struct vm_area_struct***
  - Used in Linux kernel
  - Represents an independent VM area
  - One process uses multiple vm_area_struct to indicate different VM areas



https://www.zendei.com/article/44809.html

Implement mmap in kernel space

- Initializing the mapping
  - remap_pfn_range() function
  - pfn: page frame number

```
int remap_pfn_range (
        struct  vm_area_struct *,       /*VMA struct*/
        unsigned long virt_addr,        /*Starting user virtual address*/
        unsigned long pfn,              /*pfn of the starting physical address*/
        unsigned long size,             /*Mapping size*/
        pgprot_t      prot              /*Page permissions*/
        );
```

Simple mmap implementation

```
Static int *acme_mmap
        (struct file *file, struct vm_area_struct *vma) {
        size = vma->vm_end – vma->vm_start;
        if (size > ACME_SIZE)
                return –EINVAL;
        if(remap_pfn_range (vma,
                            vma->vm_start,
                             ACME_PHYS >> PAGE_SHIFT,
                             size,
                             vma->vm_page_prot))
                return –EAGAIN;
        return 0;
}
```

/dev/mem

- Used to provide user space applications with direct access to physical addresses
- Usage:
  - Open /dev/mem and read or write at given offset
  - What you read or write is the value at the corresponding physical address
  - Used by applications such as the X server to write directly to device memory

```
fd = open("/dev/mem", O_RDWR|O_SYNC);
map_base = mmap(NULL, 0xff, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0x20000);
```

mmap summary

- A user space process calls the **mmap** system call
- The **mmap** file operation is called
- Initializes the mapping using the device physical address
- The process gets a starting address to read from and write to
- The MMU automatically converts the process virtual addresses into physical ones
- Direct access to the hardware without expensive **read** or **write** system calls

Process view in virtual memory

- During execution, each process can only view its virtual addresses
- Process cannot
  - View another processes virtual address space
  - Determine the physical address mapping
- Process can use **inter process communication (IPC)** to perform the memory sharing
  - Message passing
  - Signals

Shared memory in Linux

- **int shmget (key, size, flags)**
  - Create a shared memory segment
  - Return ID of segment: shmid
  - Key: unique identifier of the shared memory
  - Size: size of the shared memory (rounded up to the PAGE_SIZE)
- **int shmat (shmid, addr, flags)**
  - Attach shmid shared memory to address space of the calling process
  - addr: pointer to the shared memory address space
- **int shmdt (shmid)**
  - Detach shared memory

server.c

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

#define SHMSIZE     27 /* Size of shared memory */

main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;

    key = 5678; /* some key to uniquely identifies the shared memory */

    /* Create the segment. */
    if ((shmid = shmget(key, SHMSIZE, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /* Attach the segment to our data space. */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /* Now put some things into the shared memory */
    s = shm;
    for (c = 'a'; c <= 'z'; c++)
        *s++ = c;
    *s = 0; /* end with a NULL termination */

    /* Wait until the other process changes the first character
     * to '*' the shared memory */
    while (*shm != '*')
        sleep(1);
    exit(0);
}
```

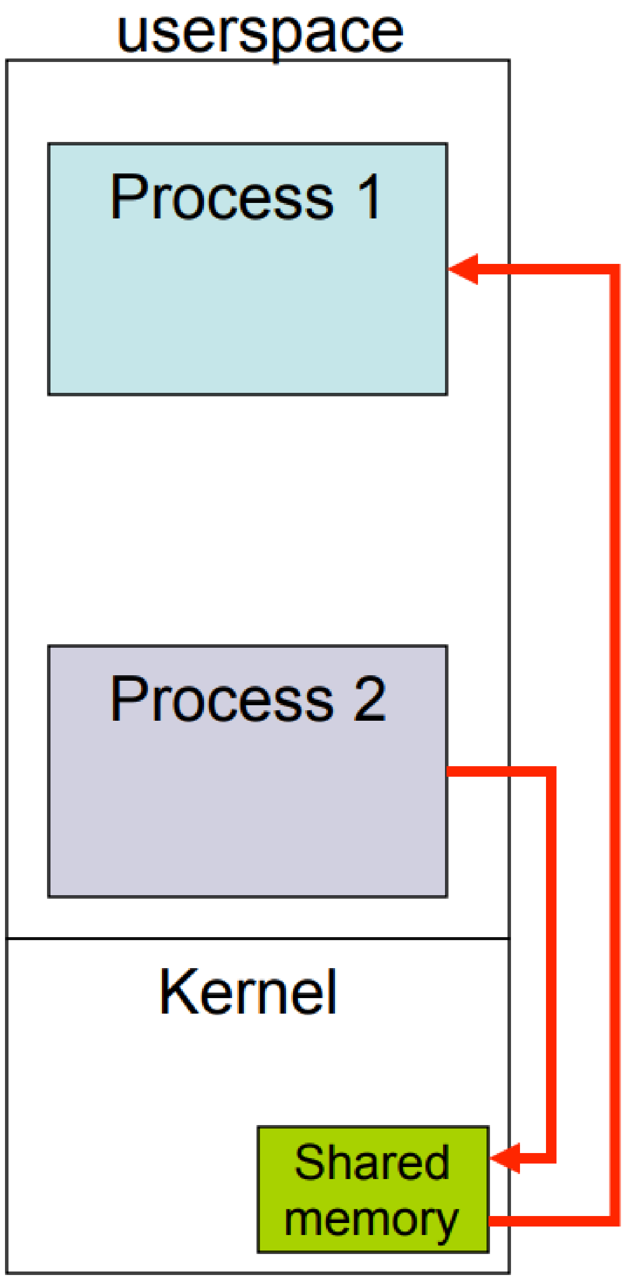
client.c

http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/8_Synchronization.pdf

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

#define SHMSIZE     27

main()
{
    int shmid;
    key_t key;
    char *shm, *s;

     /* We need to get the segment named "5678", created by the server
    key = 5678;

    /* Locate the segment. */
    if ((shmid = shmget(key, SHMSIZE, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /* Attach the segment to our data space. */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /* read what the server put in the memory. */
    for (s = shm; *s != 0; s++)
        putchar(*s);
    putchar('\n');

    /*
     * Finally, change the first character of the
     * segment to '*', indicating we have read
     * the segment.
     */
    *shm = '*';

    exit(0);
```

Message passing

- Shared memory created in the kernel
- System calls such as **send** and **receive** used for communication
  - Cooperating: each send must have a receive
- Advantages
  - Explicit sharing, less error prone
- Limitation
  - Slow
  - Each call involves marshalling/ demarshalling of information



http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/8_Synchronization.pdf

Signals

- Asynchronous unidirectional communication between processes
- Signals are a small integer
  - E.g. 9: kill, 11: segmentation fault
- Send a signal to a process
  - kill (pid, signum)
- Process handler for a signal
  - Sighandler_t signal (signum, handler);
  - Default if no handler defined

Summary

- Multi-level page table
  - Reduce the page table
- Demand paging
  - Reduce I/O, load data from disk to memory when it is needed
- Effective access time (EAT)
  - Measure the performance of demand paging
- Page replacement
- Inverted page table
- Page sharing