Operating Systems

CSE 231

Instructor: Sambuddho Chakravarty

(Semester: Monsoon 2020)

Week 9: Nov. 16 – Nov. 19

Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures

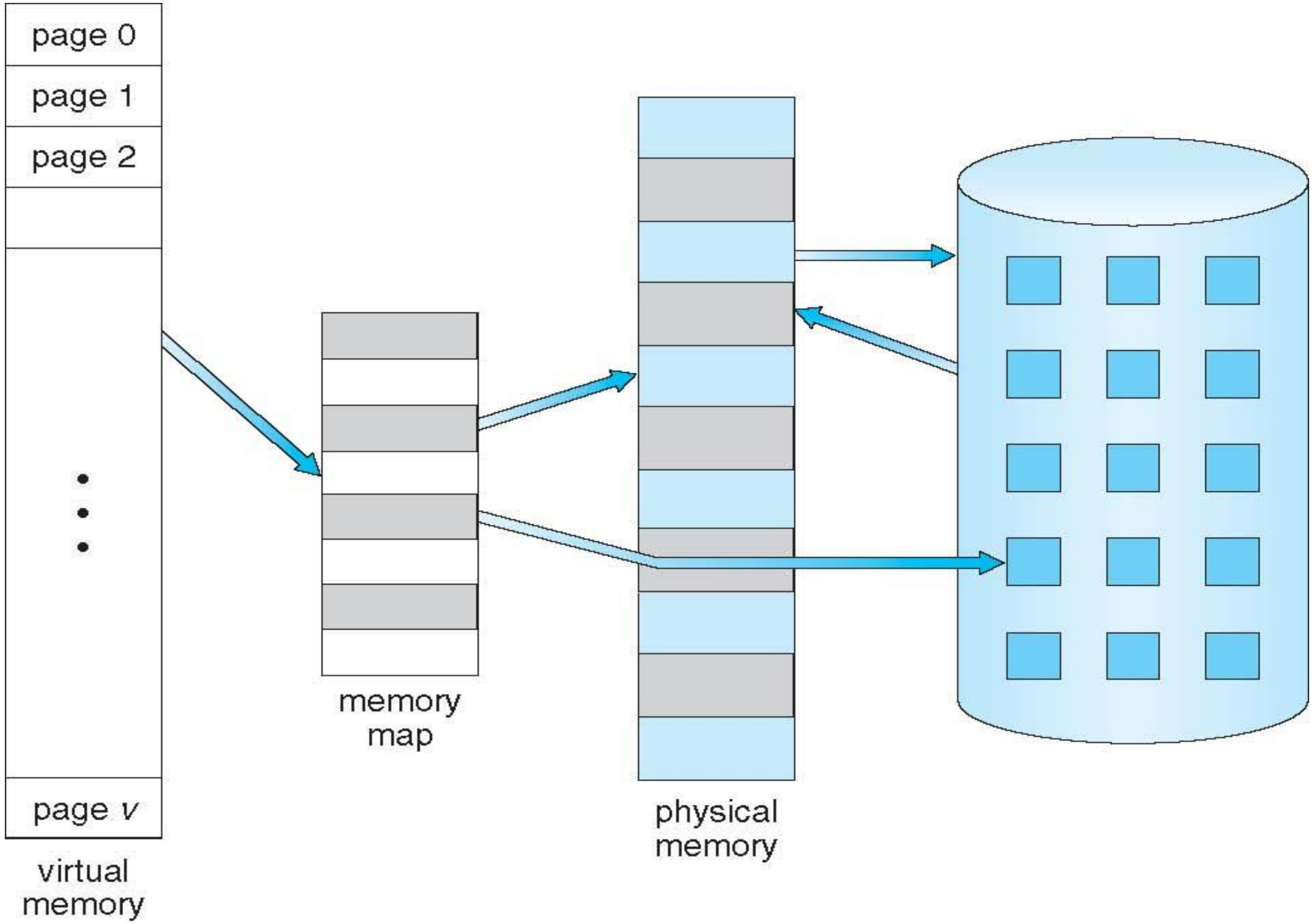
- Entire program code not needed at same time

- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running -> more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory -> each user program runs faster

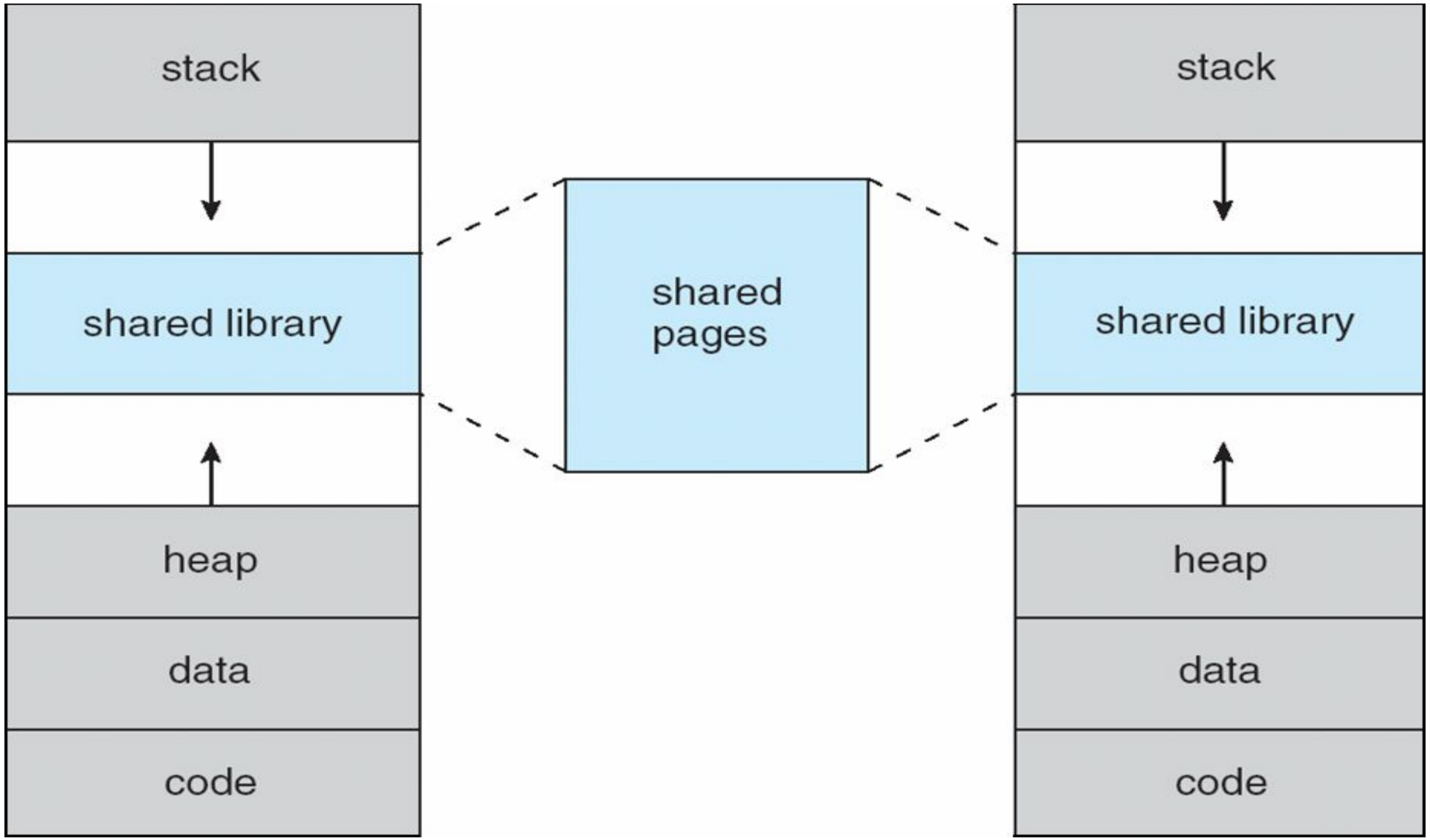
Background

- **Virtual memory** – separation of user logical memory from physical memory
    - Only part of the program needs to be in memory for execution
    - Logical address space can therefore be much larger than physical address space
    - Allows address spaces to be shared by several processes
    - Allows for more efficient process creation
    - More programs running concurrently
    - Less I/O needed to load or swap processes
- **Virtual address space** – logical view of how process is stored in memory
    - Usually start at address 0, contiguous addresses until end of space
    - Meanwhile, physical memory organized in page frames
    - MMU must map logical to physical
- Virtual memory can be implemented via:
    - Demand paging
    - Demand segmentation

Virtual Memory That is Larger Than Physical Memory

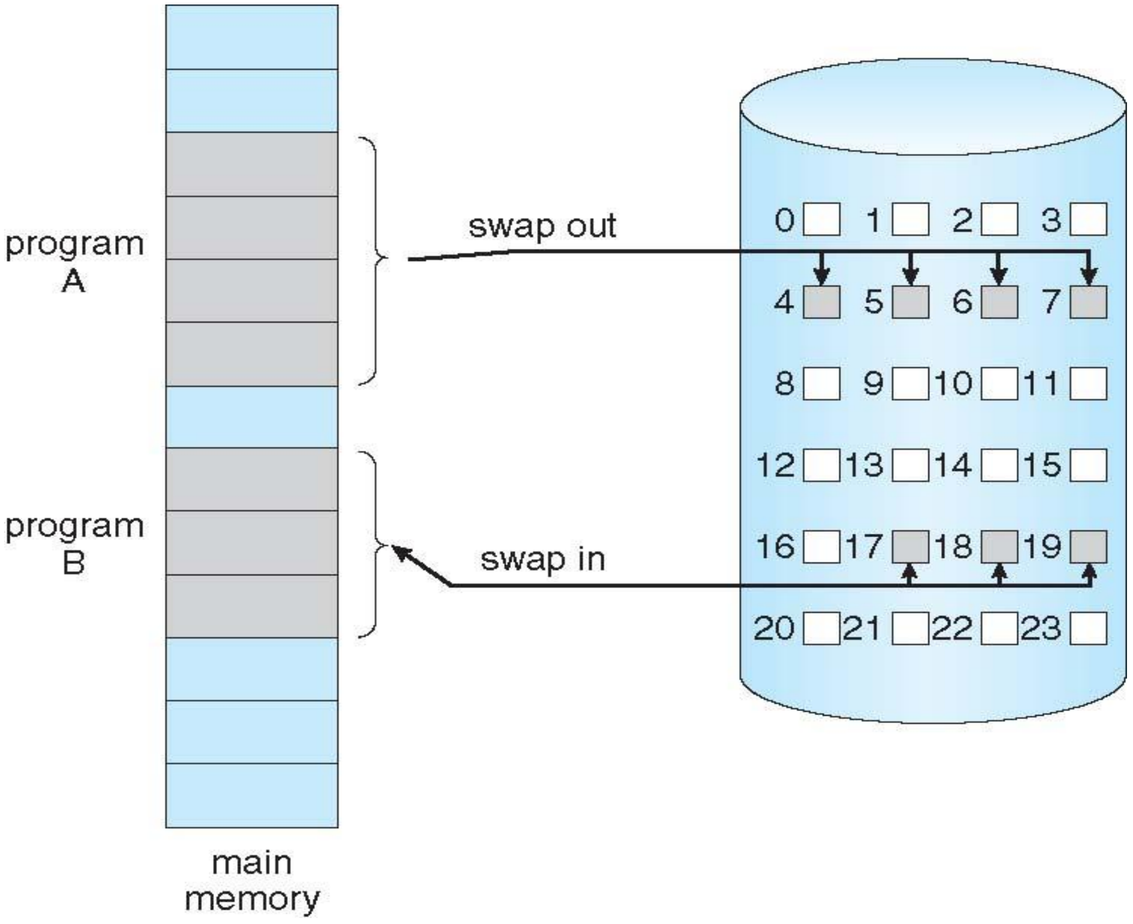


Shared Library Using Virtual Memory



Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)
- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

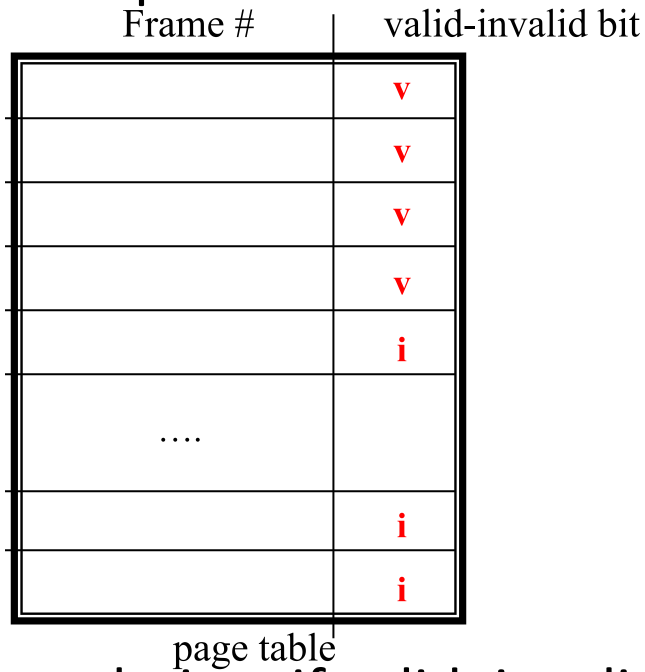


Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again
- Instead, pager brings in only those pages into memory
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
  - No difference from non demand-paging
- If page needed and not memory resident
  - Need to detect and load the page into memory from storage
    - Without changing program behavior
    - Without programmer needing to change code

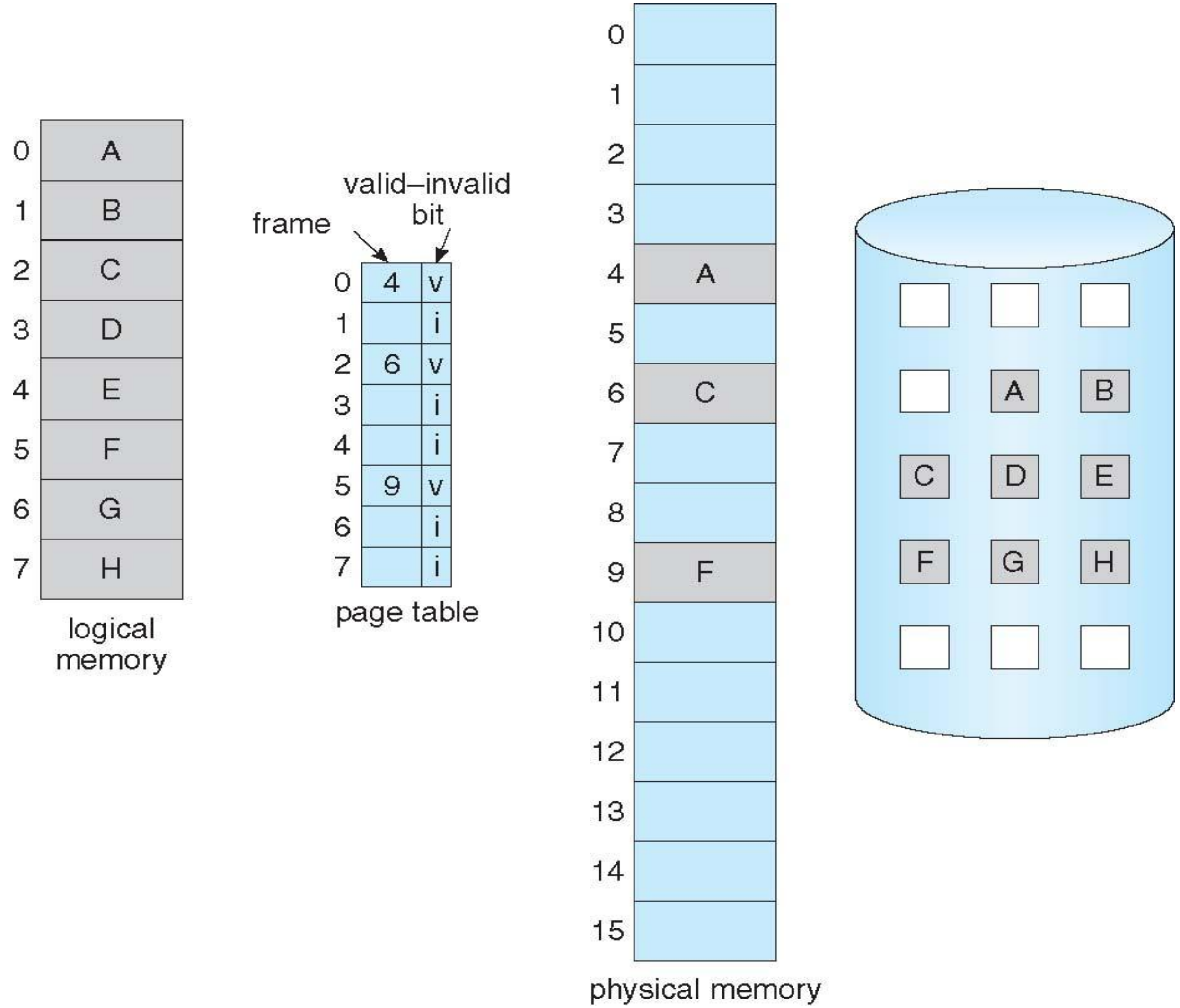
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated ($\mathbf{v} \Rightarrow$ in-memory – **memory resident**, $\mathbf{i} \Rightarrow$ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:



- During MMU address translation, if valid–invalid bit in page table entry is **i** $\Rightarrow$ page fault

Page Table When Some Pages Are Not in Main Memory



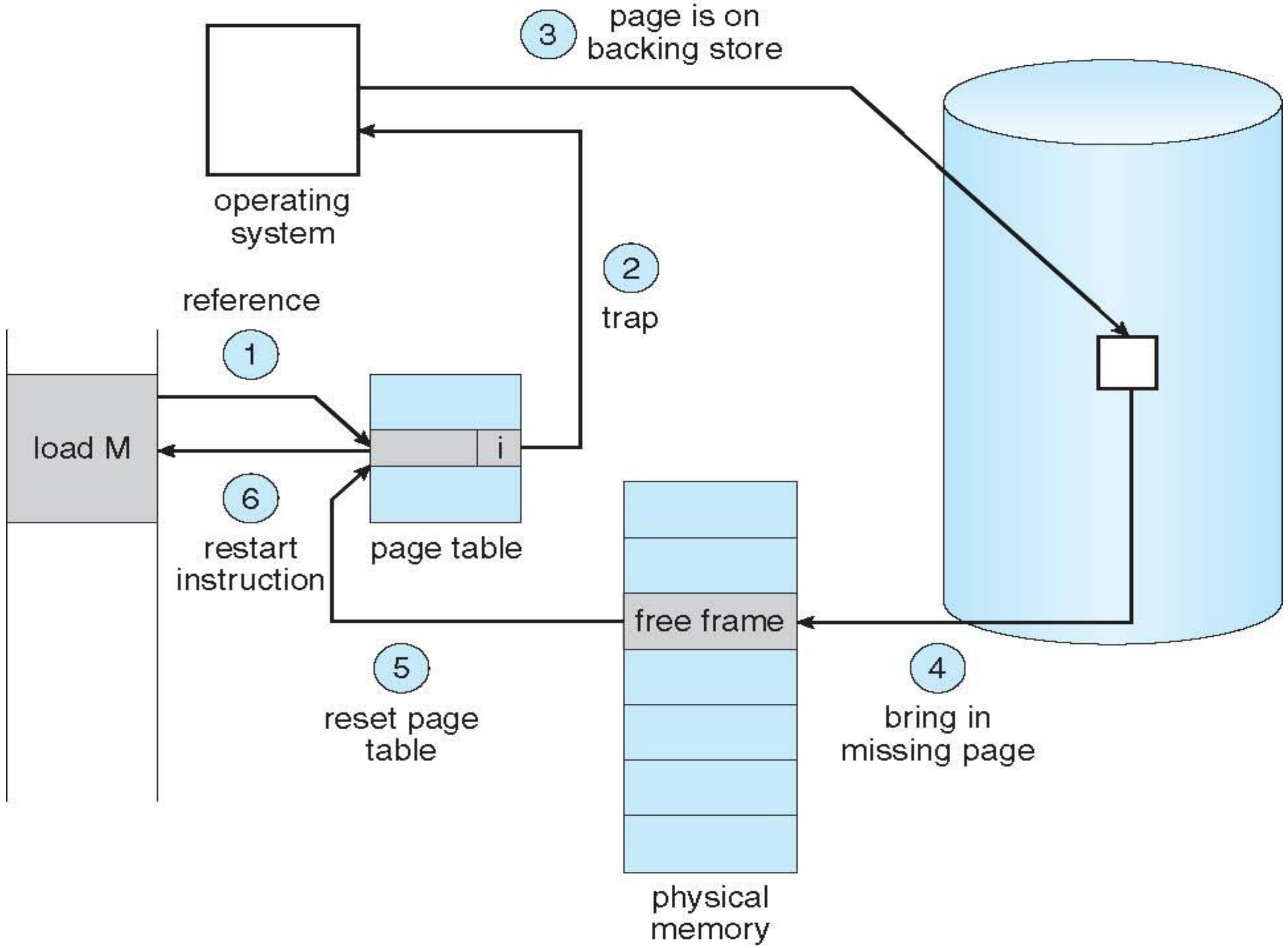
Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system:

  page fault

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
   Set validation bit = **v**
5. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Performance of Demand Paging (Cont.)

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- Page Fault Rate $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

EAT = $(1 – p)$ x memory access

+ p (page fault overhead

+ swap page out

+ swap page in

)

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds

- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p x 200 + p x 8,000,000

  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system

- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix

- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system

- Mobile systems
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

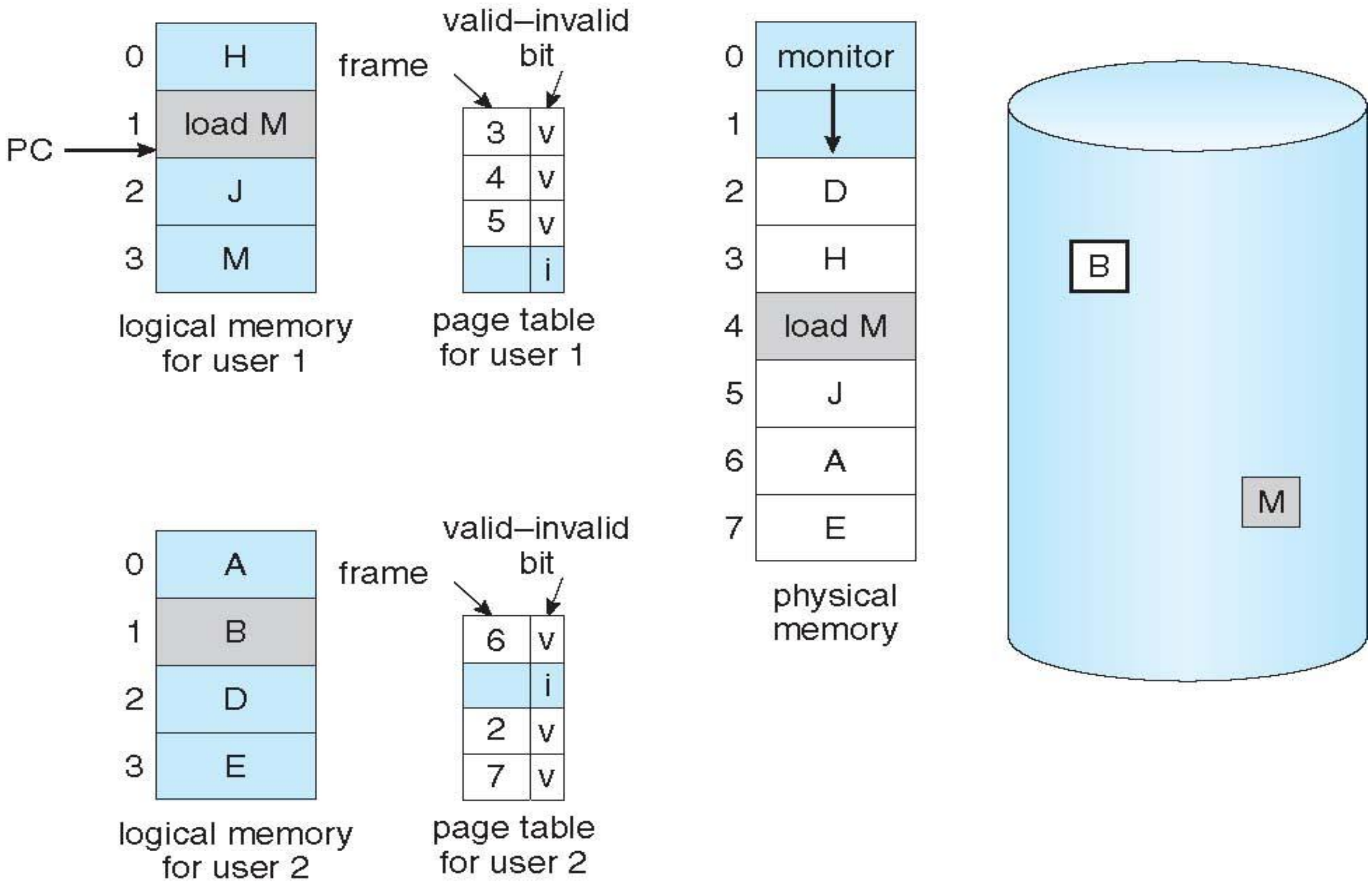
What Happens if There is no Free Frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

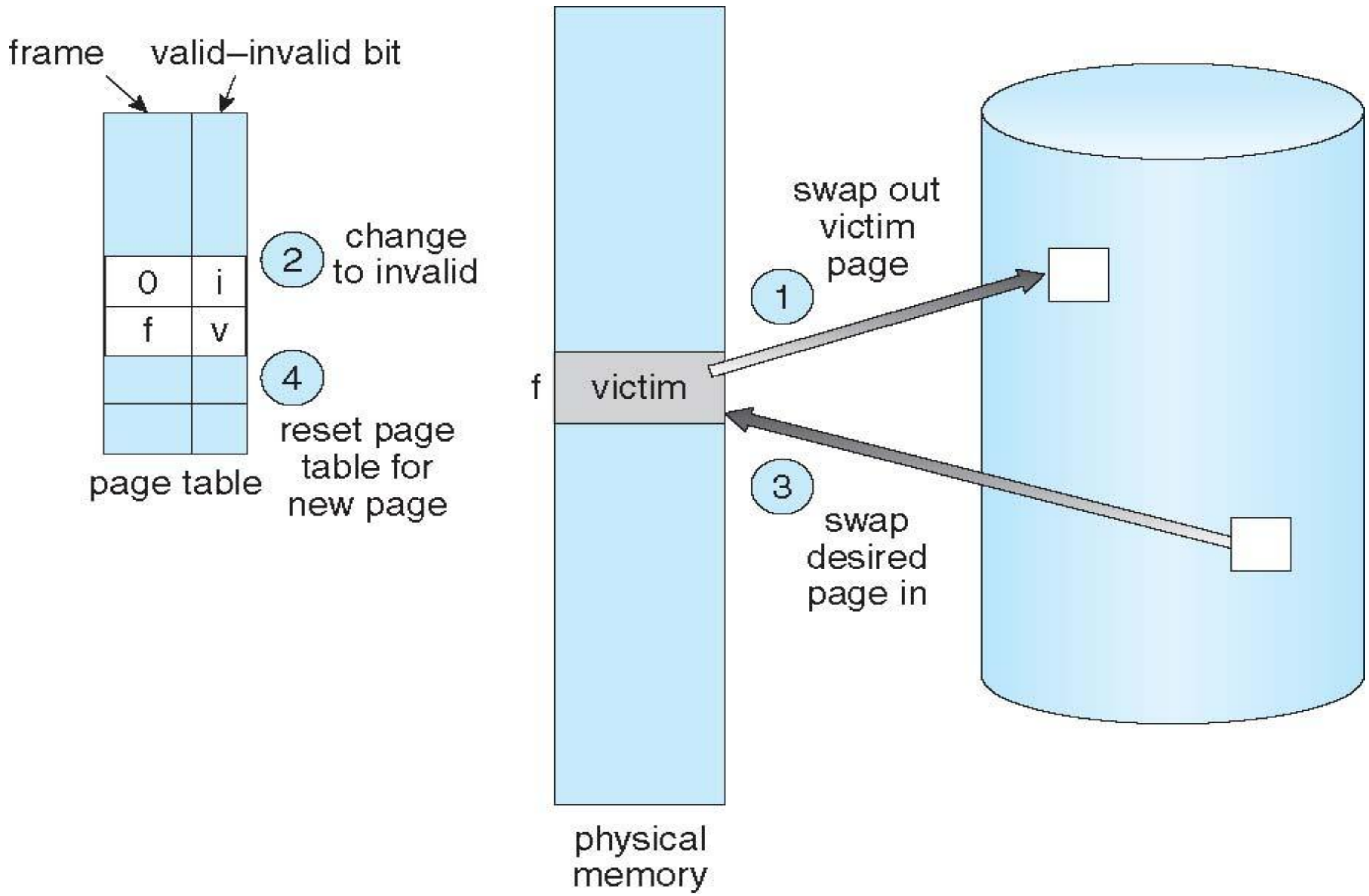
Page Replacement

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

Need For Page Replacement



Page Replacement



Basic Page Replacement

1. Find the location of the desired page on disk

2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
   - Write victim frame to disk if dirty

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT

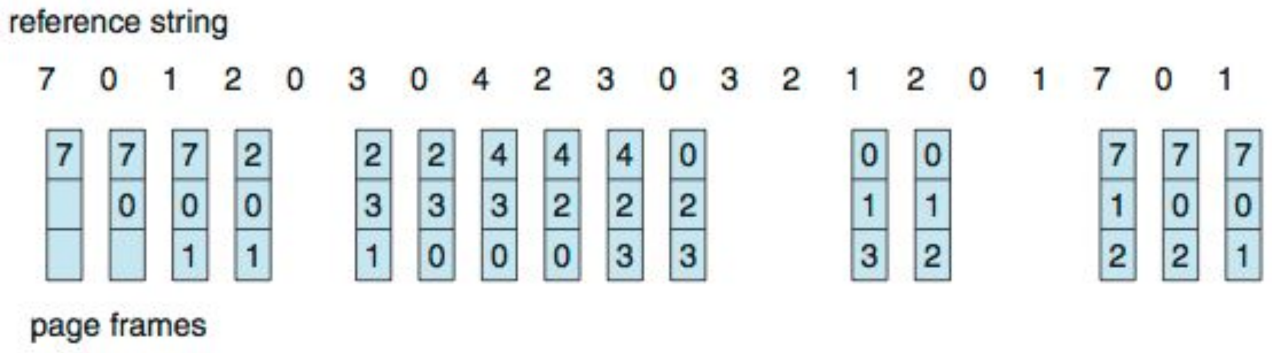
Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

  7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

First-In-First-Out (FIFO) Algorithm

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)

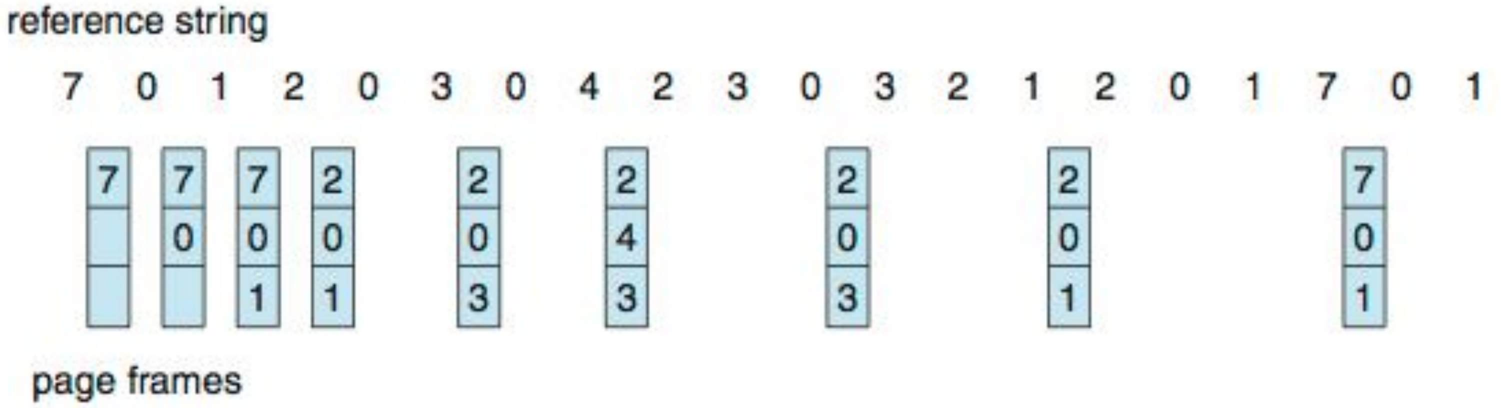


15 page faults

- How to track ages of pages?
  - Just use a FIFO queue

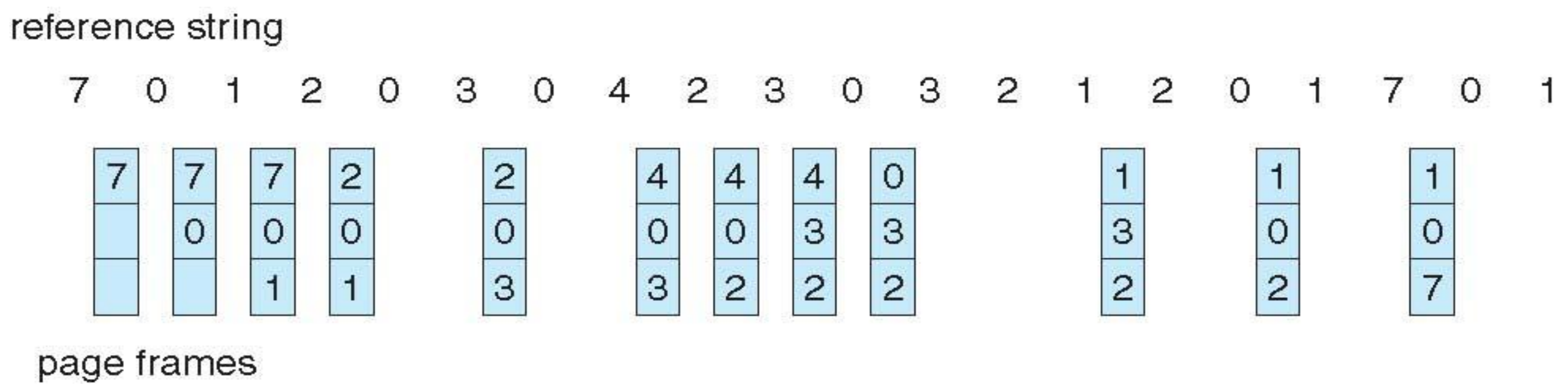
Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- How do you know this?
  - Can’t read the future
- Used for measuring how well your algorithm performs



Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page

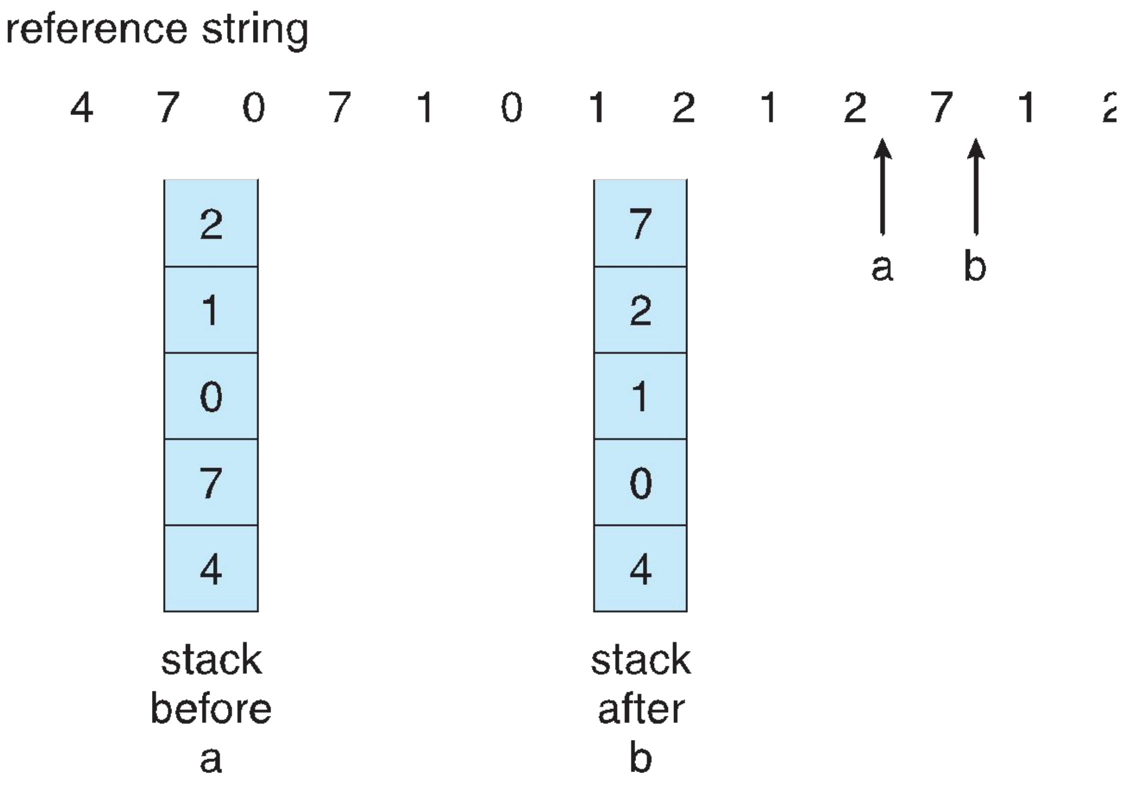


- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

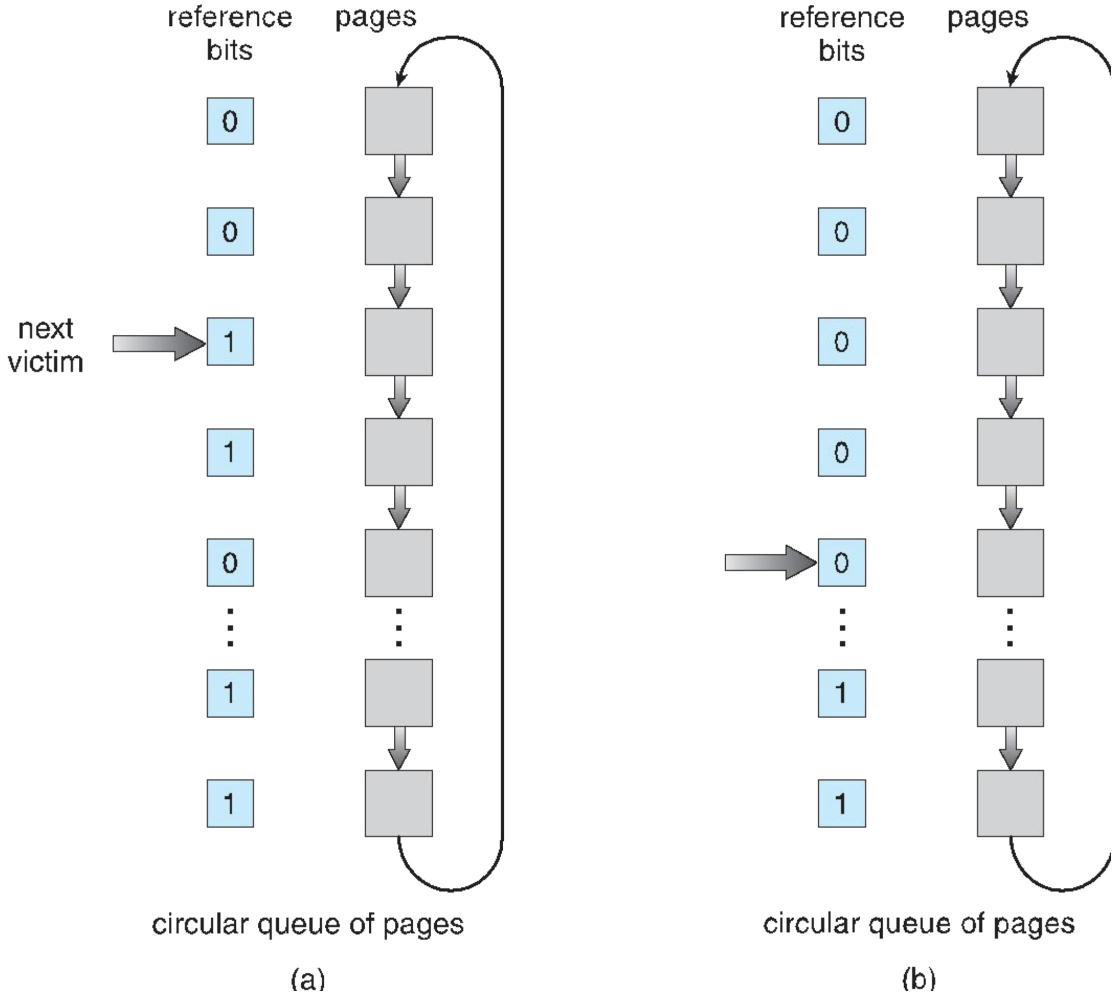
LRU Algorithm (Cont.)

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement

Use Of A Stack to Record Most Recent Page References



Second-Chance (clock) Page-Replacement Algorithm



Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify)

1. (0, 0) neither recently used not modified – best page to replace
2. (0, 1) not recently used but modified – not quite as good, must write out before replacement
3. (1, 0) recently used but clean – probably will be used again soon
4. (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement

- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
  - Might need to search circular queue several times

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common
- **Lease Frequently Used** (**LFU**) **Algorithm**: replaces page with smallest count
- **Most Frequently Used** (**MFU**) **Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Page-Buffering Algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can given direct access to the disk, getting out of the way of the applications
  - **Raw disk** mode
- Bypasses buffering, locking, etc

Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

— $s_i = \text{size of process } p_i$

— $S = \sum s_i$

— $m = \text{total number of frames}$

— $a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 62 \approx 4$$

$$a_2 = \frac{127}{137} \times 62 \approx 57$$

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size

- If process $\boldsymbol{P_i}$ generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

Global vs. Local Allocation

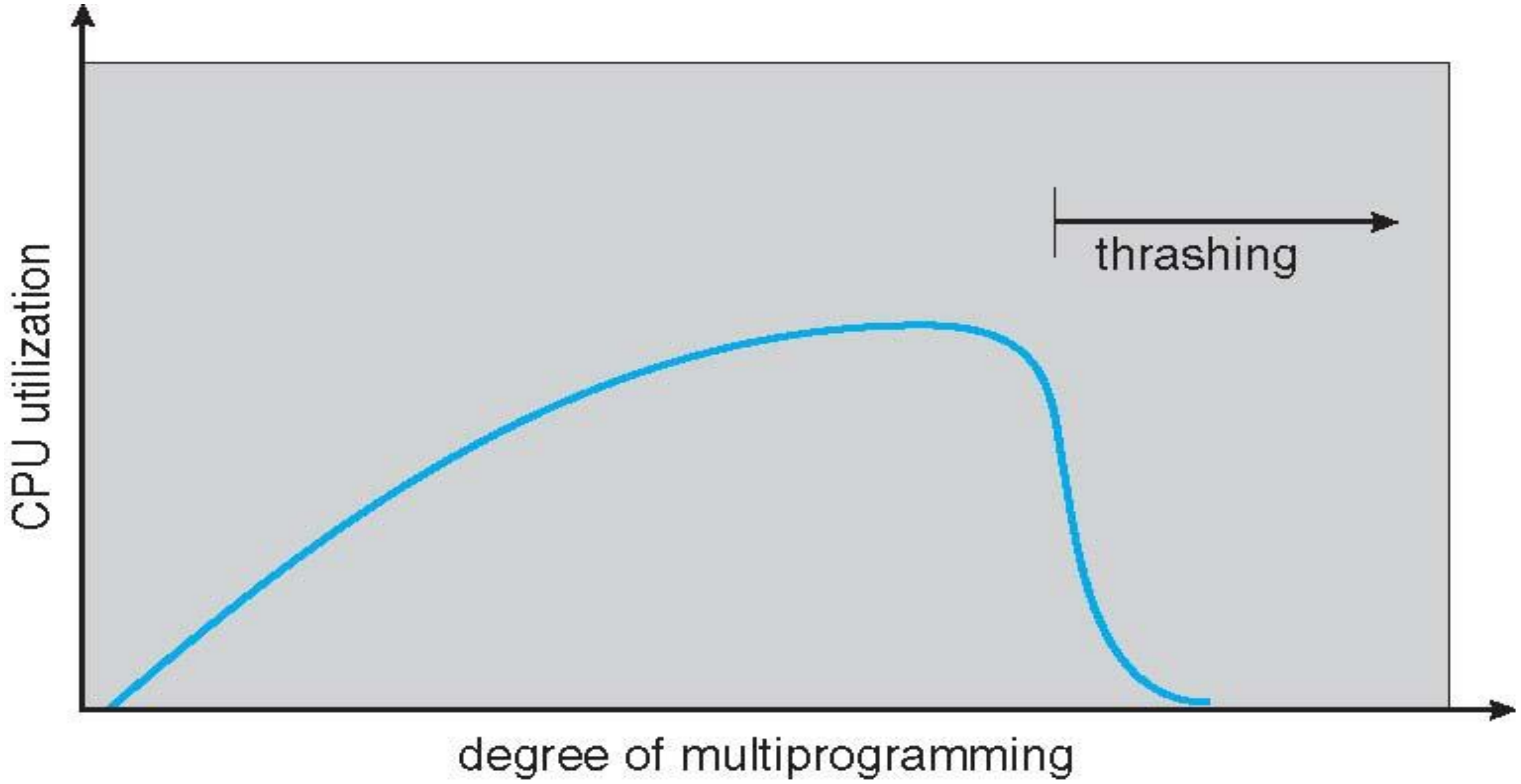
- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system

- **Thrashing** ≡ a process is busy swapping pages in and out

Thrashing (Cont.)



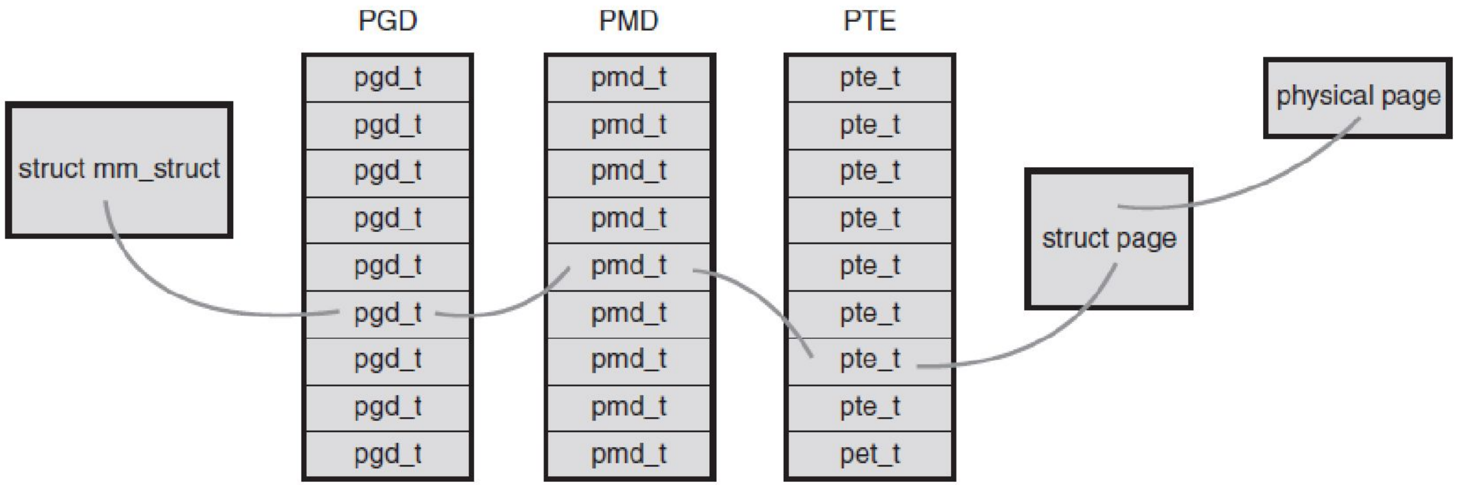
Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap

- Why does thrashing occur?
  Σ size of locality > total memory size
  - Limit effects by using local or priority page replacement

Linux Process Address Space

- Most often we use the 64-bit page tables.
- Linux page table data structures are architecture dependent and defined in <asm/page.h>



Linux Memory Descriptor

- The kernel represents a process's address space with a data structure called the *memory descriptor* – struct mm_struct (defined in <types/mm_types.h>)

mm_struct

```
struct mm_struct {
        struct vm_area_struct  *mmap;               /* list of memory areas */
        struct rb_root         mm_rb;               /* red-black tree of VMAs */
        struct vm_area_struct  *mmap_cache;         /* last used memory area */
        unsigned long          free_area_cache;     /* 1st address space hole */
        pgd_t                  *pgd;                /* page global directory */
        atomic_t               mm_users;            /* address space users */
        atomic_t               mm_count;            /* primary usage counter */
        int                    map_count;           /* number of memory areas */
        struct rw_semaphore    mmap_sem;            /* memory area semaphore */
        spinlock_t             page_table_lock;     /* page table lock */
        struct list_head       mmlist;              /* list of all mm_structs */
        unsigned long          start_code;          /* start address of code */
        unsigned long          end_code;            /* final address of code */
        unsigned long          start_data;          /* start address of data */
        unsigned long          end_data;            /* final address of data */
        unsigned long          start_brk;           /* start address of heap */
        unsigned long          brk;                 /* final address of heap */
        unsigned long          start_stack;         /* start address of stack */
        unsigned long          arg_start;           /* start of arguments */
        unsigned long          arg_end;             /* end of arguments */
        unsigned long          env_start;           /* start of environment */
        unsigned long          env_end;             /* end of environment */
        unsigned long          rss;                 /* pages allocated */
        unsigned long          total_vm;            /* total number of pages */
        unsigned long          locked_vm;           /* number of locked pages */
        unsigned long          saved_auxv[AT_VECTOR_SIZE]; /* saved auxv */
        cpumask_t              cpu_vm_mask;         /* lazy TLB switch mask */
        mm_context_t           context;             /* arch-specific data */
        unsigned long          flags;               /* status flags */
        int                    core_waiters;        /* thread core dump waiters */
        struct core_state      *core_state;         /* core dump support */
        spinlock_t             ioctx_lock;          /* AIO I/O list lock */
        struct hlist_head      ioctx_list;          /* AIO I/O list */
};
```

Virtual Memory Areas (VMAs)

- Linux virtual memory areas – struct vm_area_struct (defined in <linux/mm_types.h>).
- The vm_area_struct structure describes a single memory area over a contiguous interval in a given address space.
- The kernel treats each memory area as a unique memory object and assigns certain properties, such as permissions and a set of associated operations e.g.
- Memory areas—for e.g. , memory-mapped files or the process's user-space stack.

```
struct vm_area_struct {
        struct mm_struct            *vm_mm;         /* associated mm_struct */
        unsigned long               vm_start;       /* VMA start, inclusive */
        unsigned long               vm_end;         /* VMA end , exclusive */
        struct vm_area_struct       *vm_next;       /* list of VMA's */
        pgprot_t                    vm_page_prot;   /* access permissions */
        unsigned long               vm_flags;       /* flags */
        struct rb_node              vm_rb;          /* VMA's node in the tree */
        union {         /* links to address_space->i_mmap or i_mmap_nonlinear */
                struct {
                        struct list_head        list;
                        void                    *parent;
                        struct vm_area_struct   *head;
                } vm_set;
                struct prio_tree_node prio_tree_node;
        } shared;
        struct list_head            anon_vma_node;      /* anon_vma entry */
        struct anon_vma             *anon_vma;          /* anonymous VMA object */
        struct vm_operations_struct *vm_ops;            /* associated ops */
        unsigned long               vm_pgoff;           /* offset within file */
        struct file                 *vm_file;           /* mapped file, if any */
        void                        *vm_private_data;   /* private data */
};
```

VMA Flags

Flag	Effect on the VMA and Its Pages
VM_READ	Pages can be read from.
VM_WRITE	Pages can be written to.
VM_EXEC	Pages can be executed.
VM_SHARED	Pages are shared.
VM_MAYREAD	The VM_READ flag can be set.
VM_MAYWRITE	The VM_WRITE flag can be set.
VM_MAYEXEC	The VM_EXEC flag can be set.
VM_MAYSHARE	The VM_SHARE flag can be set.
VM_GROWSDOWN	The area can grow downward.
VM_GROWSUP	The area can grow upward.
VM_SHM	The area is used for shared memory.
VM_DENYWRITE	The area maps an unwritable file.
VM_EXECUTABLE	The area maps an executable file.
VM_LOCKED	The pages in this area are locked.
VM_IO	The area maps a device's I/O space.
VM_SEQ_READ	The pages seem to be accessed sequentially.
VM_RAND_READ	The pages seem to be accessed randomly.
VM_DONTCOPY	This area must not be copied on fork().
VM_DONTEXPAND	This area cannot grow via mremap().
VM_RESERVED	This area must not be swapped out.

vma_ops

- Set of operations associated with each VMA.
- May be defined by the users as per individual requirements.

```
struct vm_operations_struct {
        void (*open) (struct vm_area_struct *);
        void (*close) (struct vm_area_struct *);
        int (*fault) (struct vm_area_struct *, struct vm_fault *);
        int (*page_mkwrite) (struct vm_area_struct *vma, struct vm_fault *vmf);
        int (*access) (struct vm_area_struct *, unsigned long ,
                       void *, int, int);
};
```