OPERATING SYSTEMS

Virtual Memory & Demand Paging

Operating Systems

School of Data & Computer Science
Sun Yat-sen University

Lecture Notes: os_sysu@163.com
Instructor: Guoyang Cai
email: isscgy@mail.sysu.edu.cn

中山大學
SUN YAT-SEN UNIVERSITY

Contents

Virtual Memory

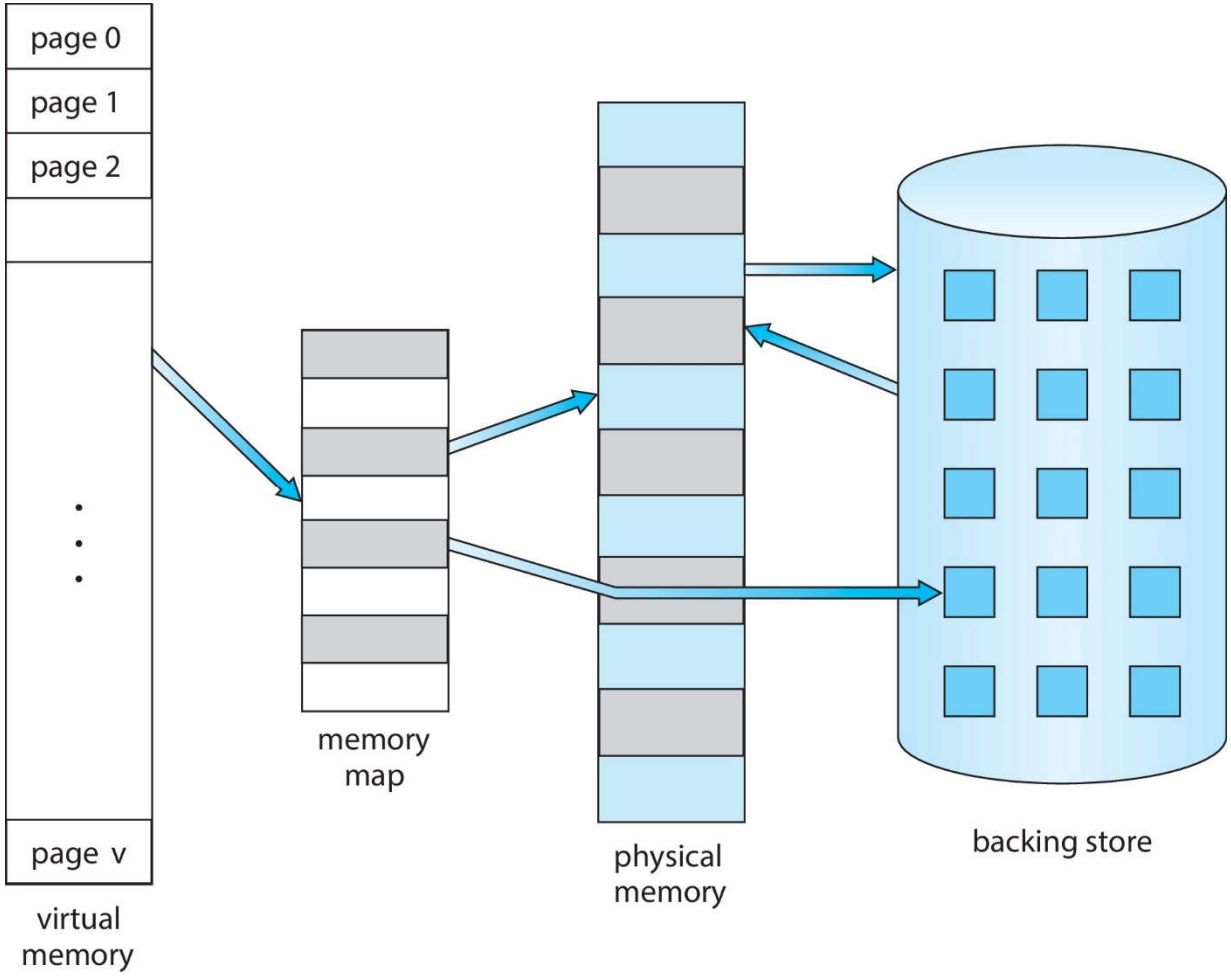
- Real Memory Management keeps many processes in memory simultaneously to allow multiprogramming.
  - An entire process should be in memory before it can execute.
- Virtual Memory is a technique that allows the execution of processes that are not completely in memory.
  - Only part of the program needs to be in memory for execution. Programs can be larger than physical memory available.
    - They are mainly maintained on secondary memory (disk).
    - More programs are running concurrently.
    - Less I/O is needed to load or swap processes.
  - The scheme abstracts main memory into an extremely large, uniform array of storage, separating user logical memory from physical memory.
    - It frees programmers from the concerns of memory-storage limitations.
- Virtual Memory also allows processes to share files and libraries, and to implement shared memory, and provides an efficient mechanism for process creation.

Virtual Memory

- Virtual Memory can be implemented via:
  - Demand paging
  - Demand segmentation.
- Based on Paging/Segmentation, a process may be broken up into pieces (pages or segments) that do not need to be located contiguously in main memory.
  - Based on the Locality Principle, all pieces of a process do not need to be loaded in main memory during execution; all addresses are virtual.

Virtual Memory

- Virtual Memory that is larger than available physical memory.



Virtual Memory

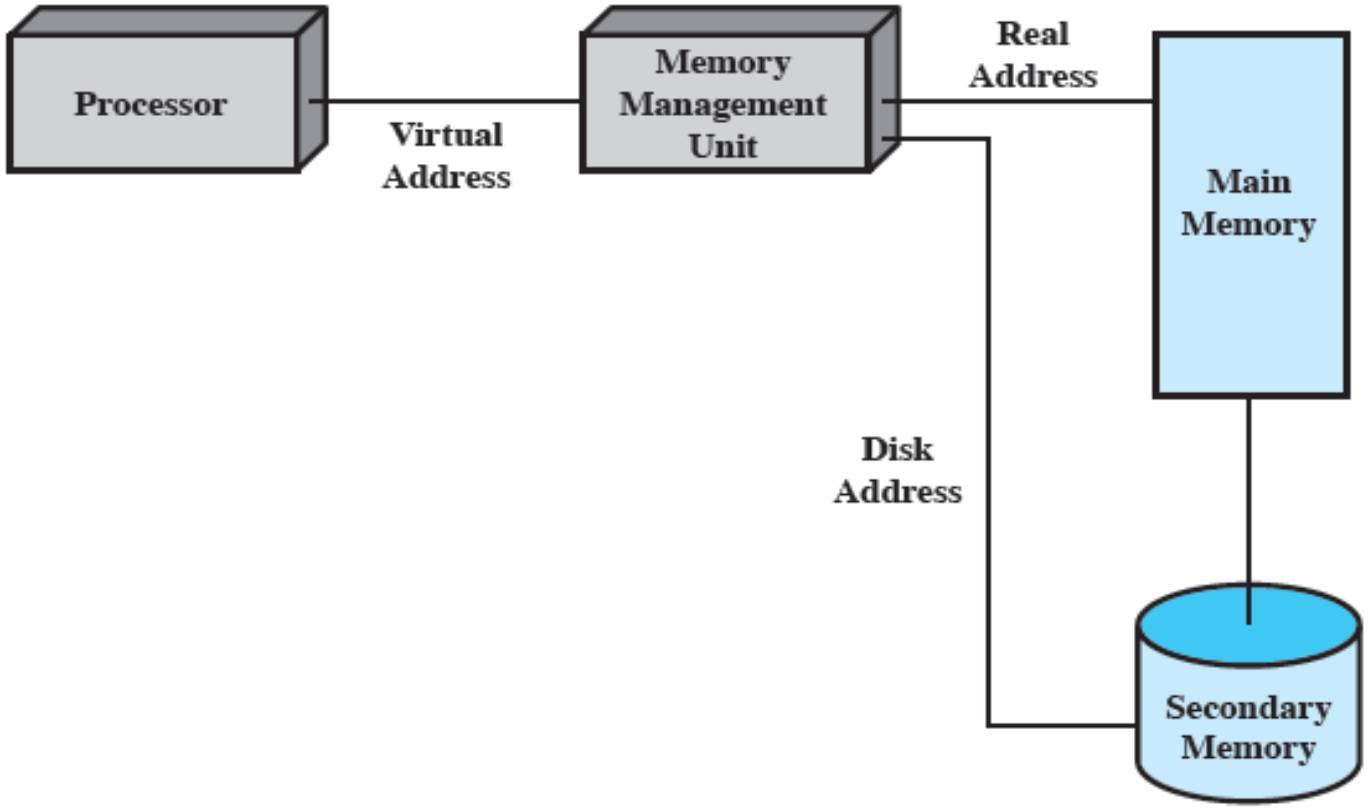
- Advantages of Partial Loading
  - More processes can be maintained in main memory.
    - only load in some of the pieces of each process
    - with more processes in main memory, it is more likely that a process will be in the Ready state at any given time
  - A process can now execute even if it is larger than the main memory size.
    - It is even possible to use more bits for logical addresses than the bits needed for addressing the physical memory.
  - Example
    - Just 16 bits are needed to address a physical memory of 64KB.
    - Let's use a page size of 1KB so that 10 bits are needed for offsets within a page.
    - We may use a number of bits larger than 6, say 22, assuming a 32-bit address.

Virtual Memory

- Support Needed for Virtual Memory
  - Memory management hardware must support paging and/or segmentation.
  - OS must be able to manage the movement of pages and/or segments between external memory and main memory, including placement and replacement of pages/segments.

Virtual Memory

- Virtual Memory Addressing.

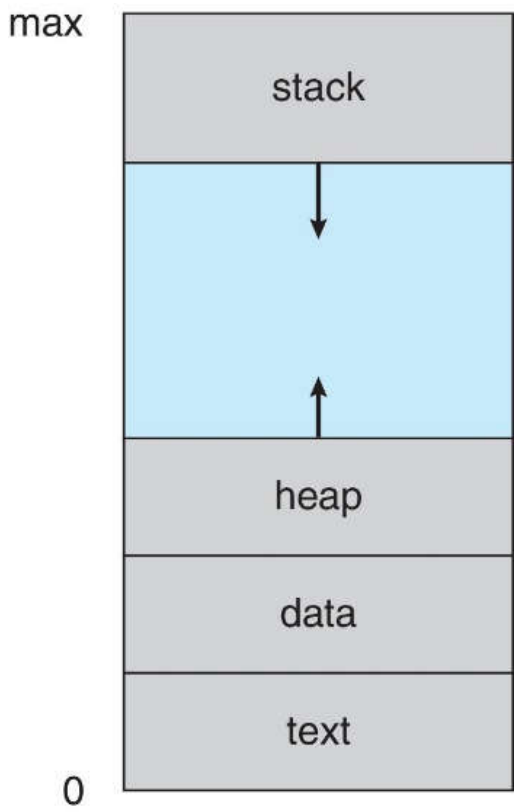


Virtual Memory

- Virtual-address Space
  - Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”:
    - Maximizes address space use
    - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
  - Enables sparse address spaces with holes left for growth, dynamically linked libraries, etc.
  - System libraries shared via mapping into virtual address space
  - Shared memory by mapping pages read-write into virtual address space
  - Pages can be shared during fork(), speeding process creation.

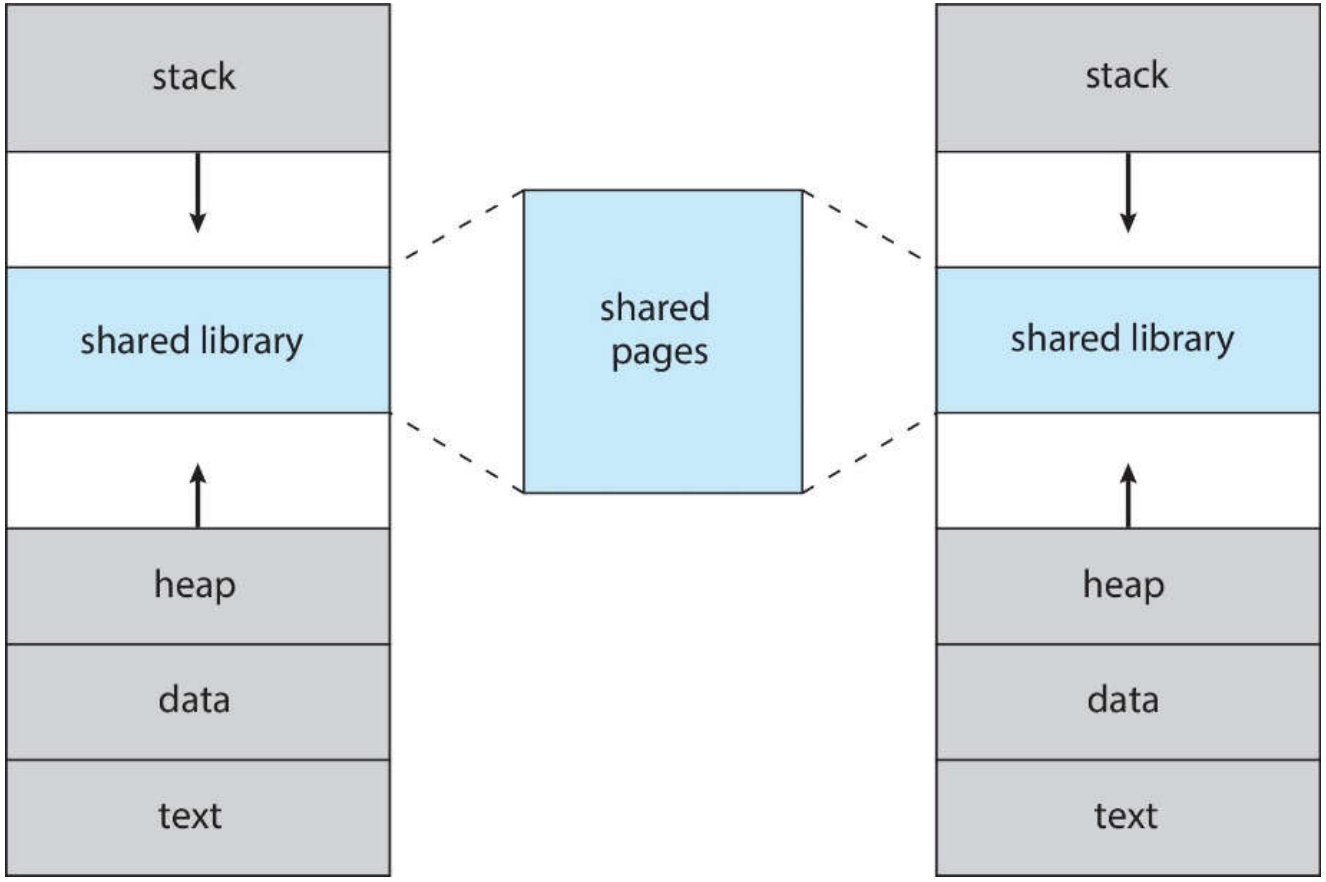
Virtual Memory

- Virtual-address Space.



Virtual Memory

- Shared Library using Virtual Memory.



Virtual Memory

- Process Execution
  - OS brings into main memory only a few pieces of the program (including its starting point).
    - Each page/segment table entry has a valid-invalid bit that is set only if the corresponding piece is in main memory.
    - The resident set is the portion of the process that is in main memory at some stage.
  - An interrupt (memory fault) is generated when the memory reference is on a piece that is not present in main memory.
    - OS places the process in a *Blocking* state.
    - OS issues a disk I/O Read request to bring into main memory the piece referenced to.
    - Another process is dispatched to run while the disk I/O takes place.
    - An interrupt is issued when disk I/O completes; this causes the OS to place the affected process back in the *Ready* state.

Basic Concepts

- 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.
- Page is needed ⇒ reference to it.
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- Similar to paging system with swapping.
  - Pager is the swapper that deals with pages.
  - With swapping, pager guesses which pages will be used before swapping out again.
- *Lazy swapper* – never swaps a page into memory unless page will be needed.
  - Pager brings in only those pages needed into memory.

Basic Concepts

- New MMU functionality is needed 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

Basic Concepts

- Valid-Invalid Bit
  - With each page table entry, a valid-invalid bit (present-absent bit) is associated.
    - *v* : page in-memory; *i* : page not-in-memory
  - Initially valid–invalid bit is set to *i* on all entries

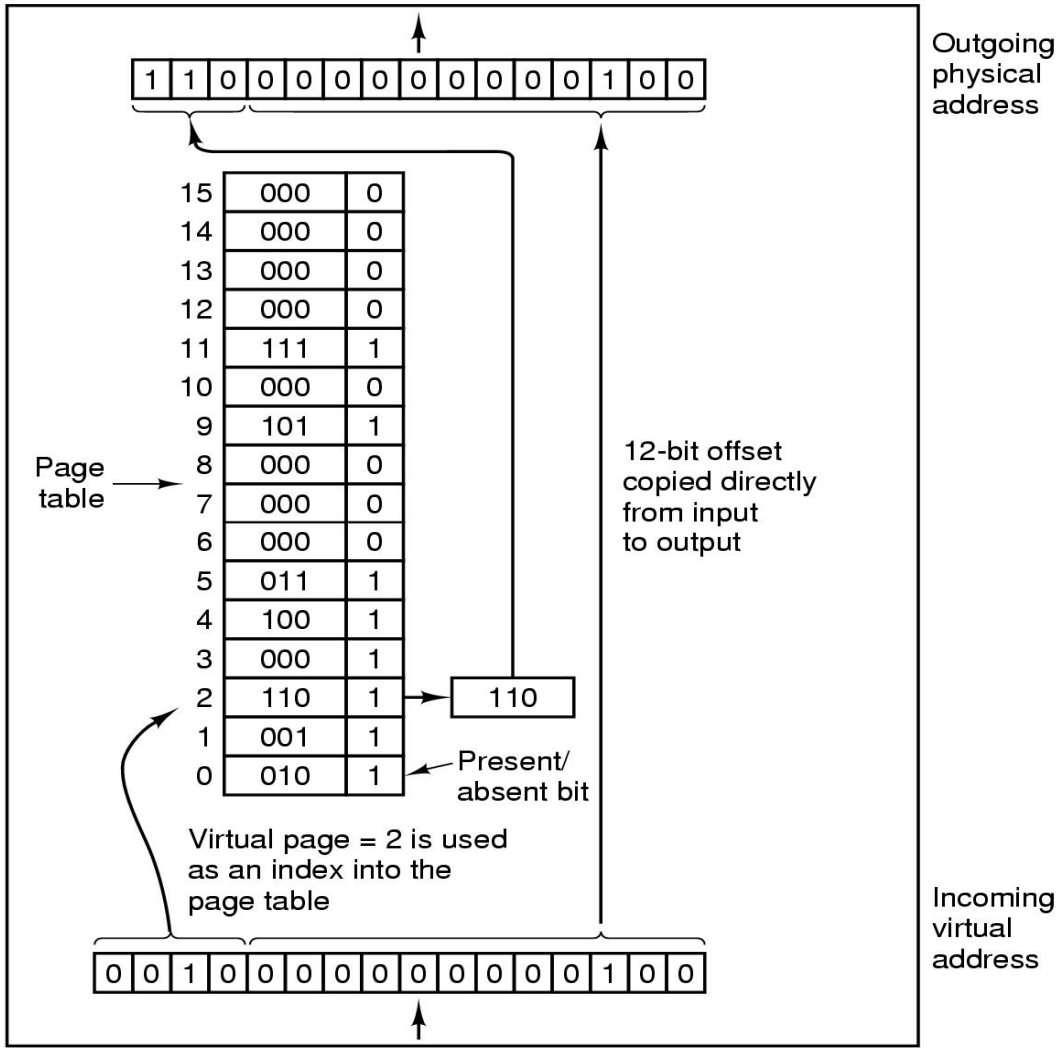
Frame #	v-i bit
	v
	v
	v
	v
	i
… …	
	i
	i

Page Table

  - During address translation, if valid–invalid bit in page table entry is *i* then page fault.

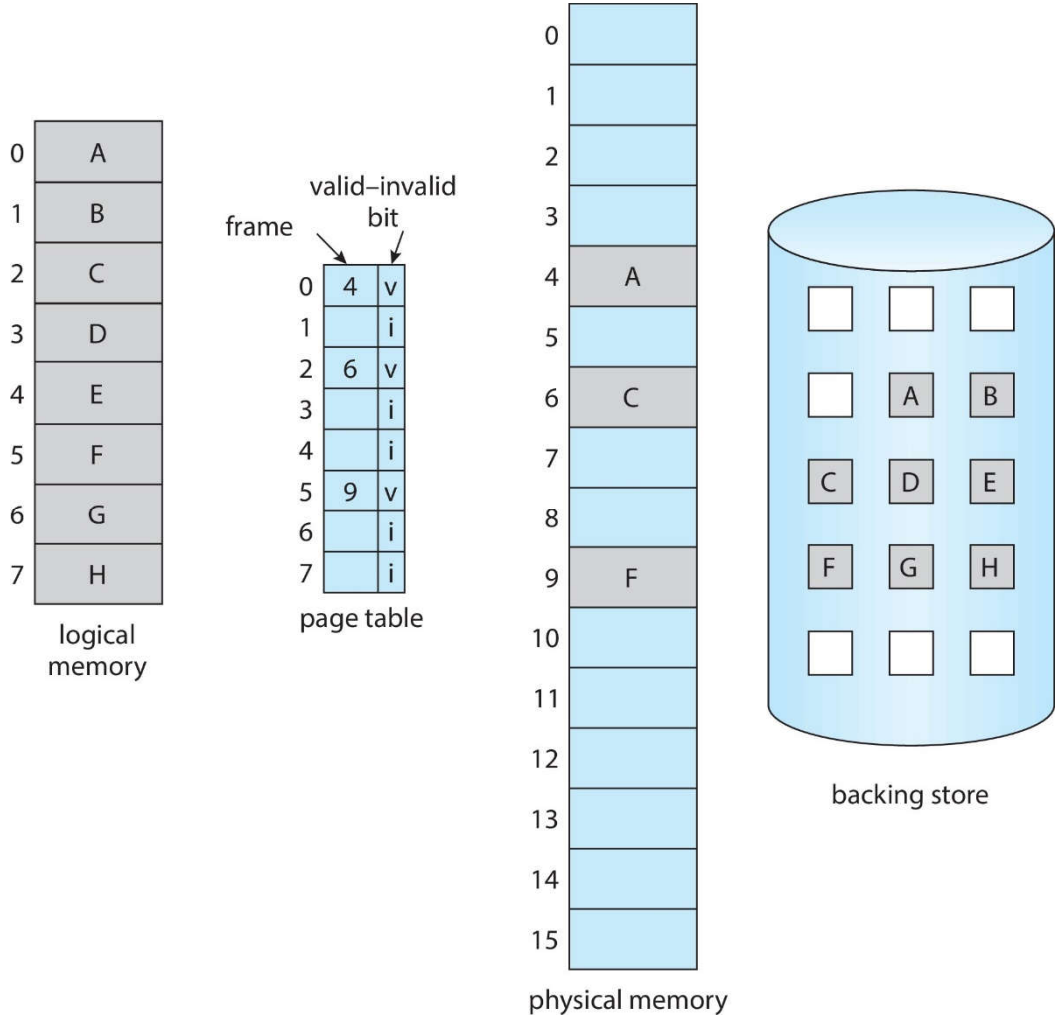
Basic Concepts

- Virtual Memory Mapping Example.



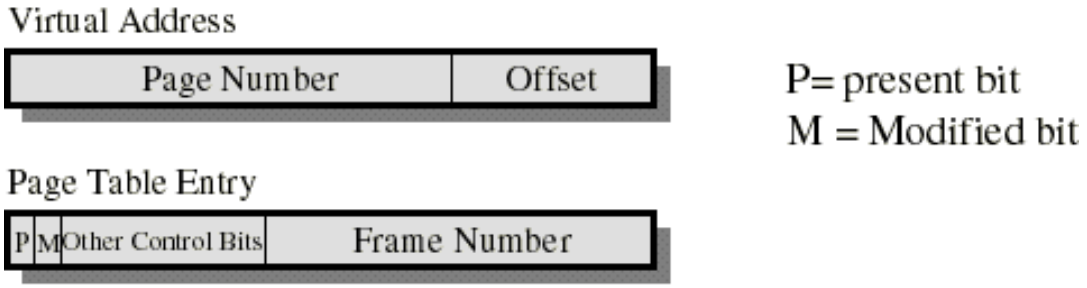
Basic Concepts

- Page Table when some Pages are not in Main Memory.



Dynamics of Demand Paging

- Typically, each process has its own page table.



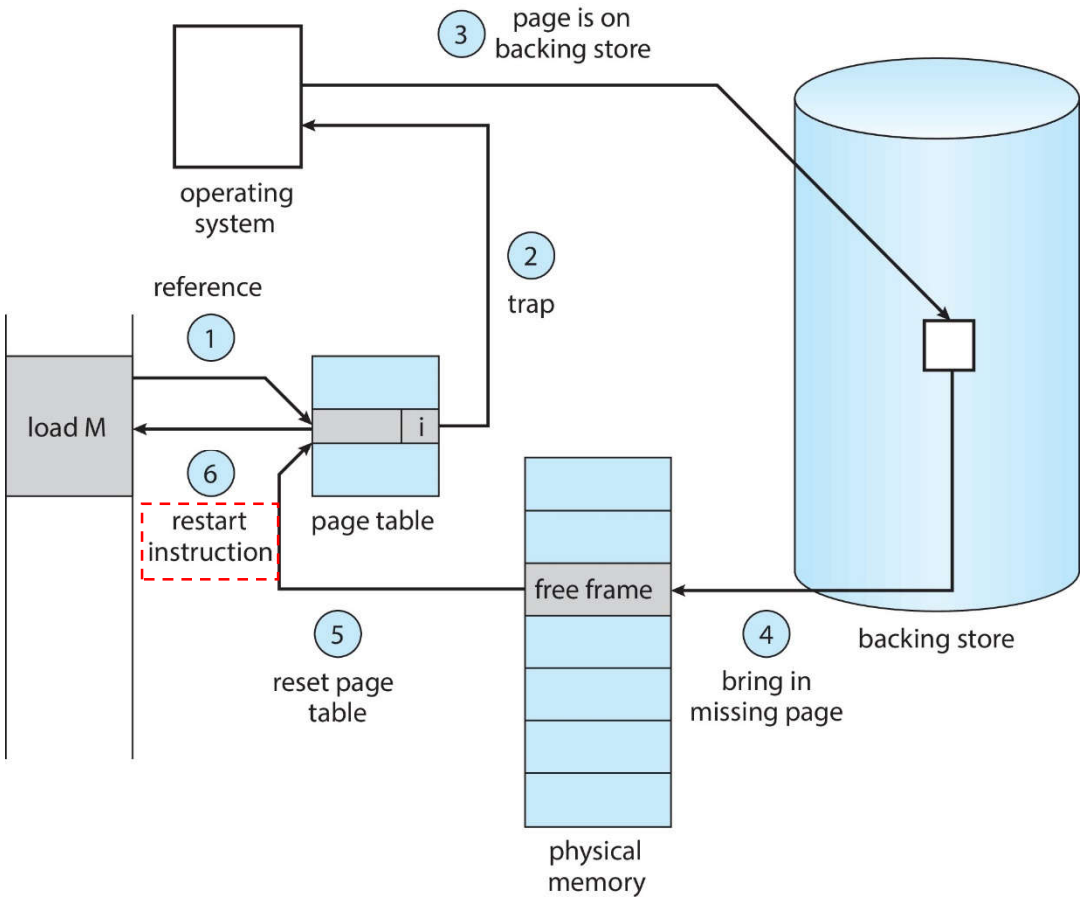
- Each page table entry contains a *present bit* (valid-invalid bit) to indicate whether the page is in main memory or not.
  - If it is in main memory, the entry contains the frame number of the corresponding page in main memory.
  - If it is not in main memory, the entry may contain the address of that page on disk or the page number may be used to index another table (often in the PCB) to obtain the address of that page on disk.

Dynamics of Demand Paging

- A *modified bit* indicates if the page has been altered since it was last loaded into main memory.
  - If no change has been made, page does not have to be written back to the disk when it needs to be swapped out.
- Other control bits may be present if protection is managed at the page level.
  - a read-only/read-write bit
  - protection level bit: kernel page or user page (more bits are used when the processor supports more than 2 protection levels)

Page Fault

- If there is a reference to a page not in memory, first reference to that page will trap to operating system: this is a *page fault* (缺页中断).



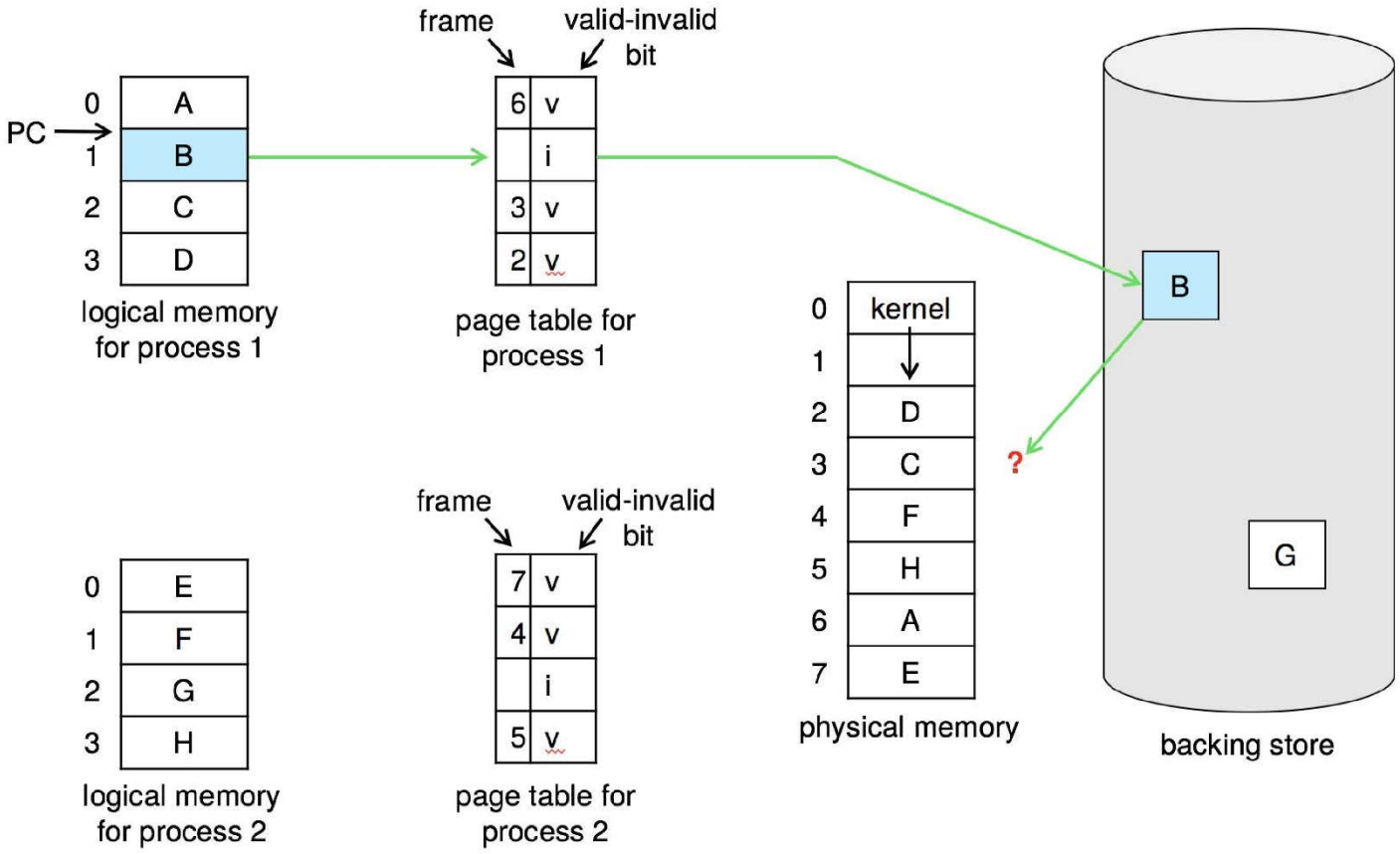
Steps in handling a page fault.

Page Fault

- Steps in handling a page fault:
  - (1) If there is ever a reference to a page not in memory, the first reference to that page will trap to operating system, causing page fault.
  - (2) Page fault is handled by the appropriate operating system service routines. OS looks at another table to decide:
    - Invalid reference $\Rightarrow$ abort
    - Just not in memory
  - (3) Locate the needed page on disk (in file or in backing store)
  - (4) Find a free frame; Swap page into free frame via scheduled disk operation
  - (5) Reset page tables to indicate page now in memory:
    - Set valid-invalid bit = *v*
  - (6) Restart the instruction that caused the page fault

Page Replacement

- What happens if there is no free frame?



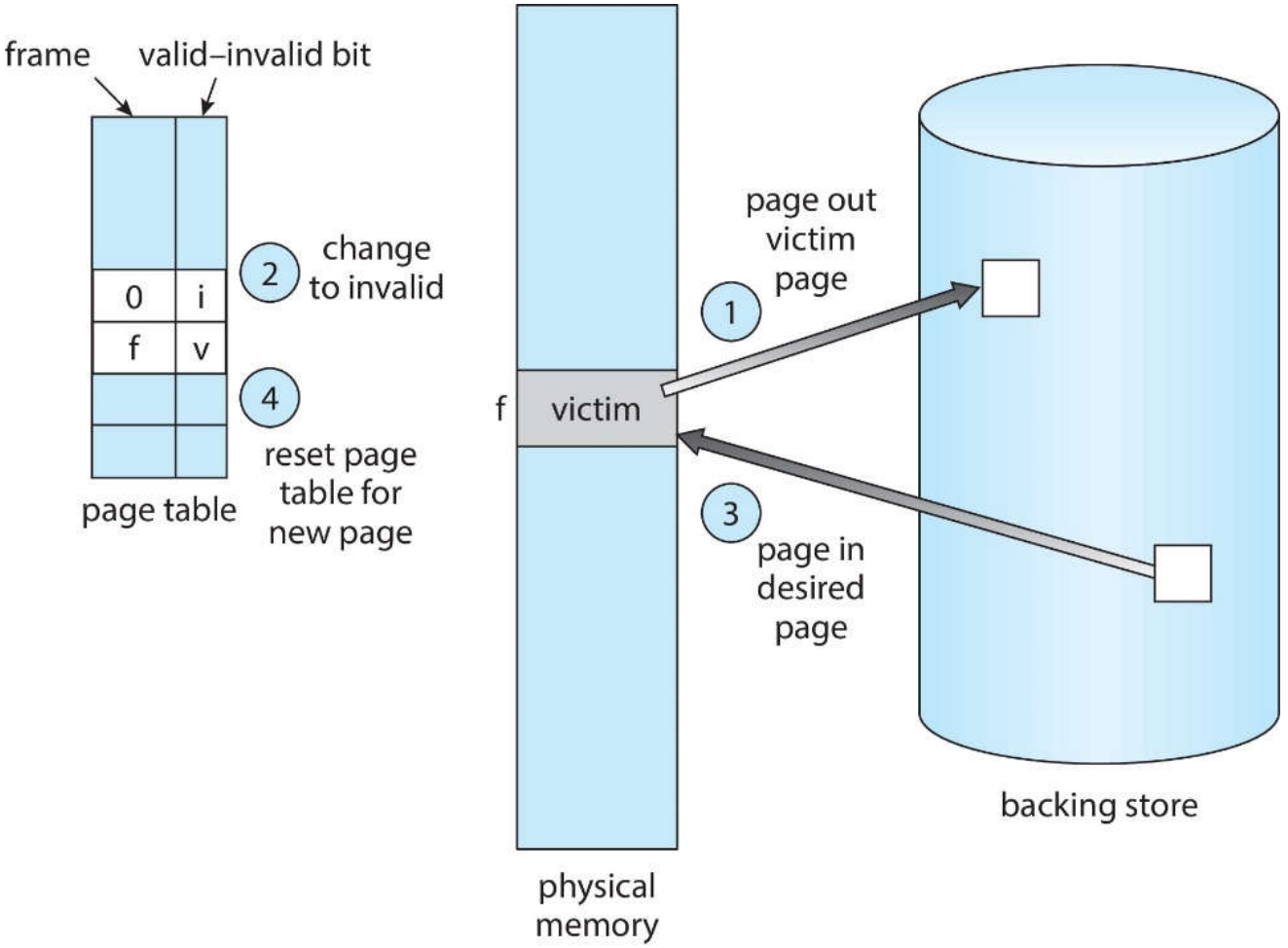
Need for Page Replacement

Page Replacement

- What happens if there is no free frame?
  - Page replacement (页面置换)
    - find some page in memory but not really in use, and then swap it out
      - page replacement algorithm (页面置换算法).
  - Performance
    - want an algorithm which will result in minimum number of page faults
  - Same page may be brought into memory several times.

Page Replacement

- Steps in handling a page replacement.



Page Replacement

- Steps in handling a 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 page*.
      - The victim page should not really in use.
  - (3) Bring the desired page into the (newly) free frame; update the page and frame tables.
  - (4) Reset the page table for the new page.
  - (5) Continue page fault handling.

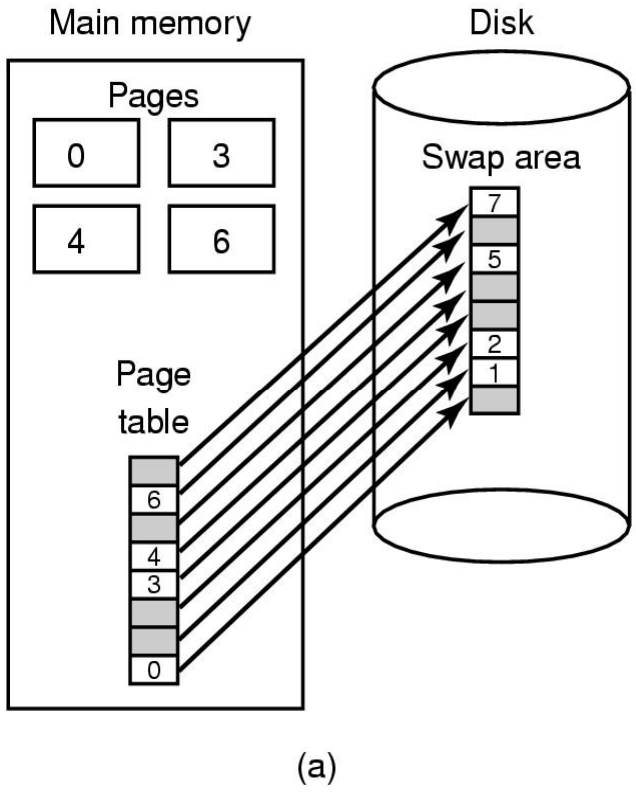
Page Replacement

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

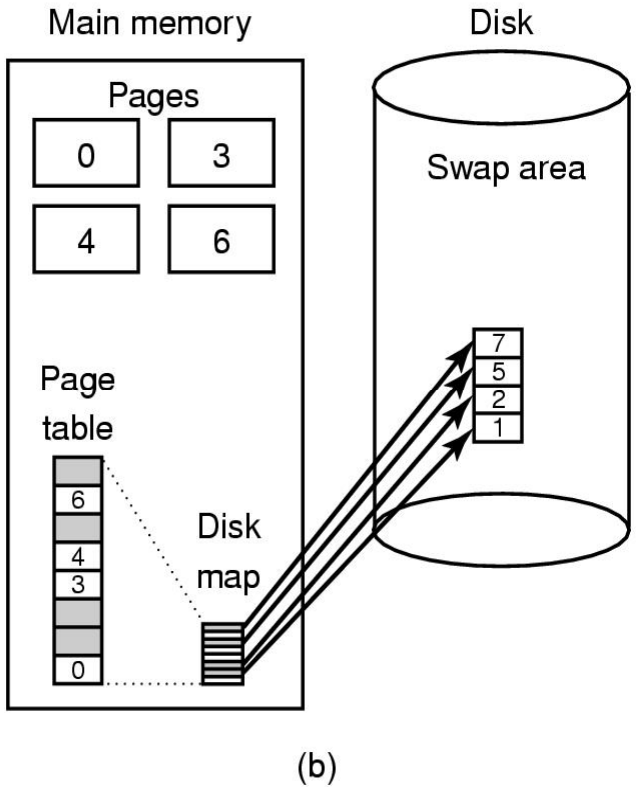
Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory.
  - This is *Pure demand paging*.
  - OS sets instruction pointer to the first instruction of process, non-memory-resident will cause page fault.
    - and for every other process pages on first access
- Actually, a given instruction could access multiple pages and cause multiple page faults.
  - consider fetch and decode of an instruction which adds 2 numbers from memory and stores result back to memory
    - pain decreased because of *locality of reference*
- Hardware support needed for demand paging.
  - Page table with valid-invalid bit
  - Secondary memory (swap device with *swap space*)
  - Instruction restarting mechanism.

Backing/Swap Store



(a) Paging to static swap area.



(b) Backing up pages dynamically.

Translation Look-aside Buffer (TLB)

- Because the page table is in main memory, each virtual memory reference causes at least two physical memory accesses:
  - one to fetch the page table entry
  - one to fetch the data
- To overcome this problem a special cache is set up for page table entries, called the Translation Look-aside Buffer (TLB, 快表):
  - TLB contains page table entries that have been most recently used.
  - It works similar to main memory cache.
- Example.

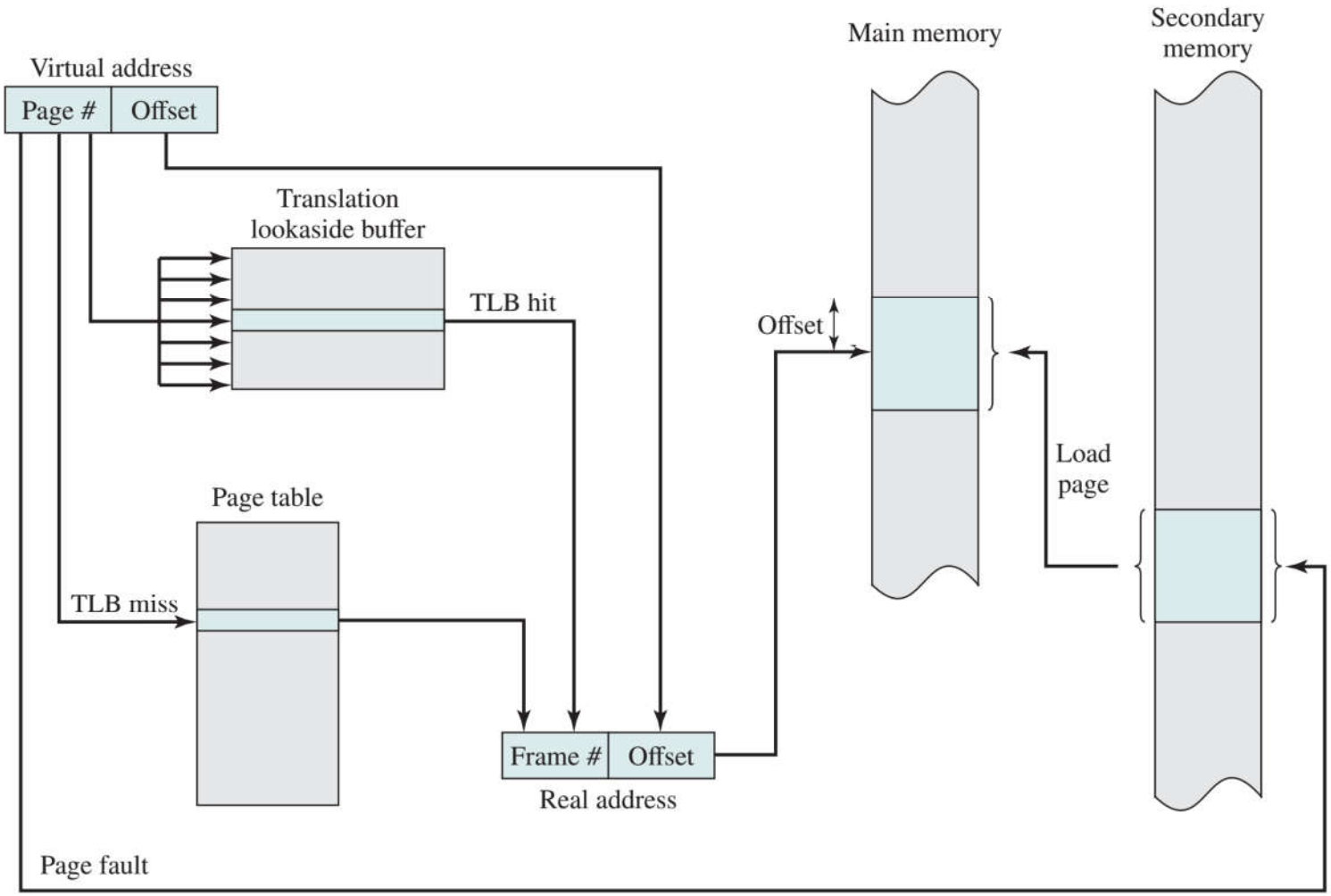
Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

TLB Dynamics

- Given a logical address, the processor examines the TLB.
  - If page table entry is present (a hit), the frame number is retrieved and the real (physical) address is formed.
  - If page table entry is not found in the TLB (a miss), the page number is used to index the process page table:
    - if valid bit is set, then the corresponding frame is accessed.
    - if not, a page fault is issued to bring in the referenced page in main memory.
- The TLB is updated to include the new page entry.

TLB Dynamics

- Use of a TLB.



Stages in Demand Paging (worse case)

(1) Trap to the operating system
(2) Save the user registers and process state
(3) Determine that the interrupt was a page fault
(4) Check that the page reference was legal and determine the location of the page on the disk
(5) Issue a read from the disk to a free frame:
    (a) Wait in a queue for this device until the read request is serviced
    (b) Wait for the device seek and/or latency time
    (c) Begin the transfer of the page to a free frame
(6) While waiting, allocate the CPU to some other user
(7) Receive an interrupt from the disk I/O subsystem (I/O completed)
(8) Save the registers and process state for the other user
(9) Determine that the interrupt was from the disk
(10) Correct the page table and other tables to show page is now in memory
(11) Wait for the CPU to be allocated to this process again
(12) Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of Demand Paging

- 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
- Let p be the probability of a page fault ($0 \le p \le 1$).
  - if p = 0, no page faults.
  - if p = 1, every reference causes a page fault.
  - page_faul_service_time = page_fault_overhead
    + swap_page_out
    + swap_page_in
    + restart_overhead.
- Effective Access Time (EAT)

  EAT = (1 – p) x memory_access_time
  + p x page_fault_service_time
  - For simplicity, we ignore the TLB lookup time ε.

Performance of Demand Paging

- Example.
  - Memory access time = 200 nanoseconds.
  - Average page_fault_service_time = 8 milliseconds.

$$\begin{aligned} EAT &= (1 - p) \times 200(ns) + p \times 8(ms) \\ &= (1 - p) \times 200 + p \times 8{,}000{,}000(ns) \\ &= 200 + 7{,}999{,}800p \ (ns). \end{aligned}$$

  - If one access out of 1,000 causes a page fault, then $p = 0.001$.

$$EAT = 8199.8 \ (ns) = 8.2(\mu s).$$

    - This is a slowdown by a factor of

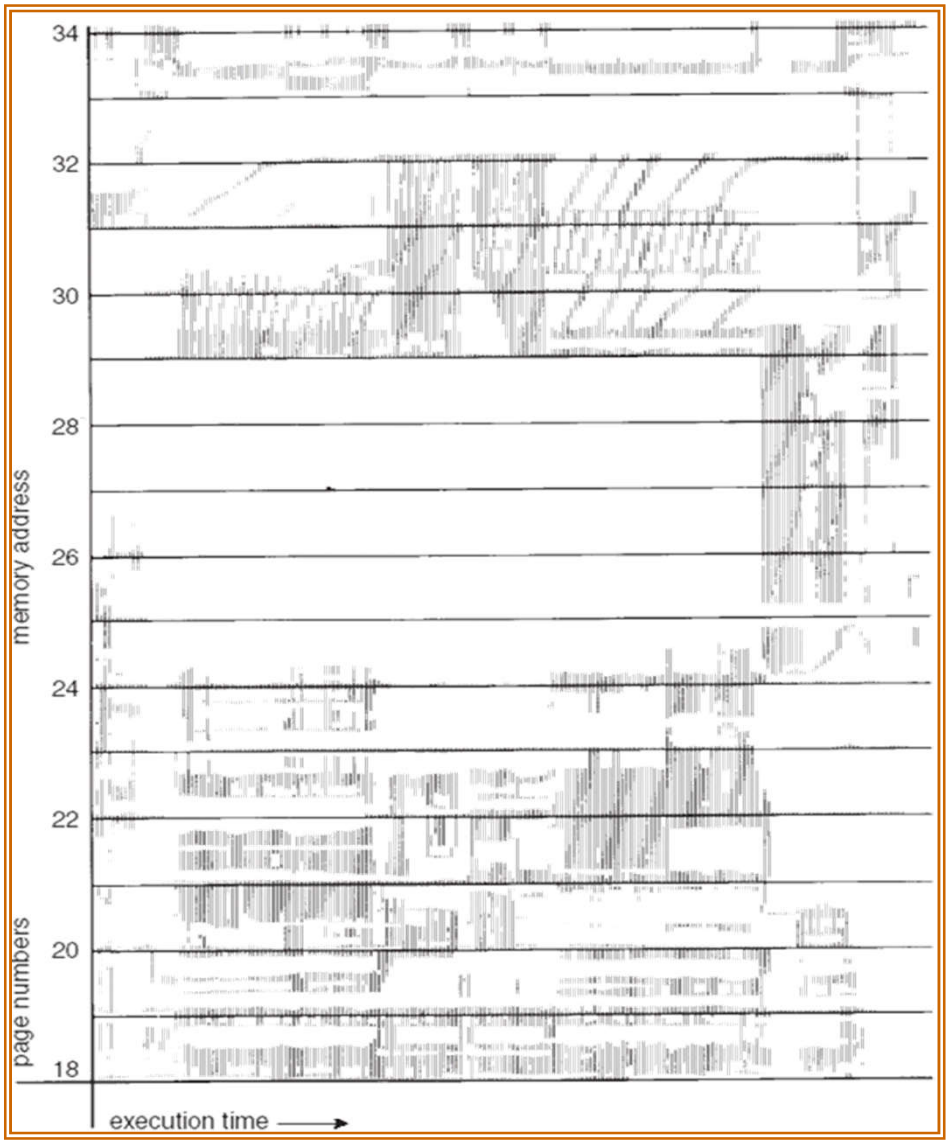
$$8.2(\mu s)/200(ns) = 41.$$

  - If we want performance degradation < 10 percent

$$\begin{aligned} &EAT / 200 < (1 + 0.1). \\ &(200 + 7{,}999{,}800p) / 200 < (1 + 0.1). \\ &p < 2.5 \times 10^{-6}. \end{aligned}$$

Locality in a Memory-Reference Pattern

- Program Memory Access Patterns have:
  - Temporal Locality (时间局部性)
  - Spatial Locality (空间局部性)
- Group of Pages accessed along a given time slice is called the “Working Set”. Working Set defines minimum number of pages needed for process to behave well.



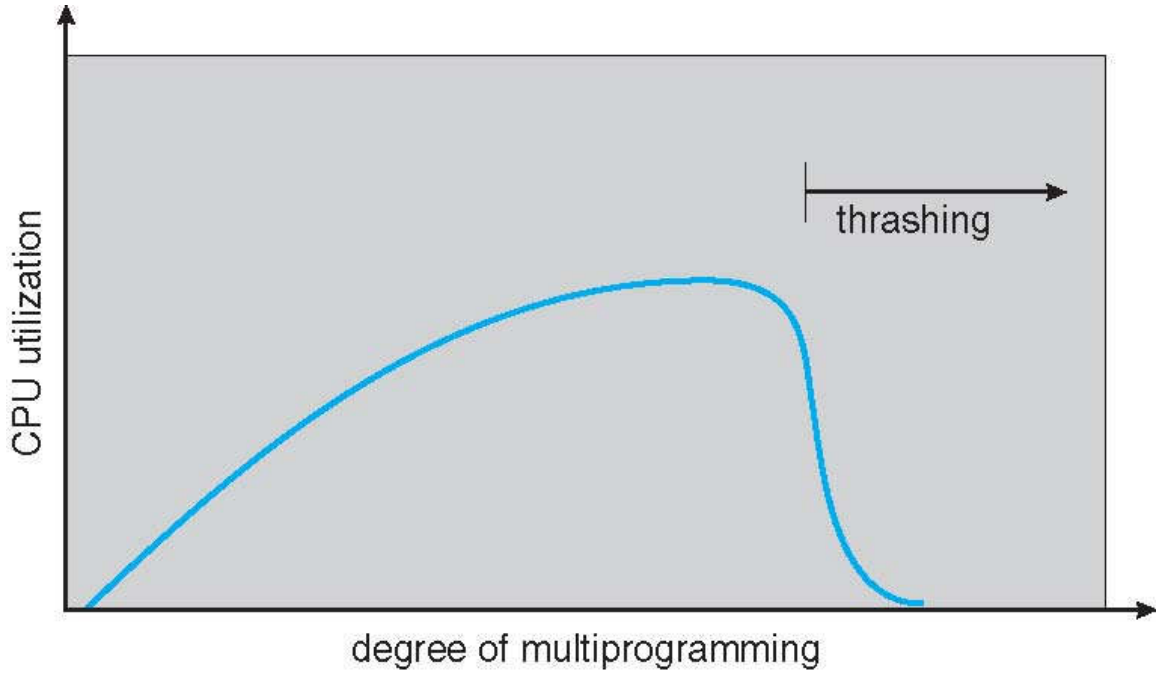
Locality and Virtual Memory

- Principle of locality of references
  - Memory references within a process tend to cluster.
- Hence, only a few pieces of a process will be needed over a short period of time.
  - It is possible to make intelligent guesses about which pieces will be needed in the future.
  - This suggests that virtual memory may work efficiently (i.e., thrashing should not occur too often).

Possibility of Thrashing

- To accommodate as many processes as possible, only a few pieces of each process are maintained in main memory.
- If a process does not have ″enough″ pages in main memory, it has to
  - cause Page fault to get page
  - replace some existing frame if main memory is over-allocated
  - but quickly need the replaced page back
- *Thrashing* (振荡/抖动)
  - The processor spends most of its time swapping pieces rather than executing user instructions.
- This leads to:
  - Low CPU utilization
  - Operating system thinking that it needs to increase the degree of multiprogramming
  - Another process being added to the system

Possibility of Thrashing



Locality and Thrashing

- Why does demand paging work?
  - Locality model:
    - Process migrates from one locality to another.
    - Localities may overlap.
- Why does thrashing occur?
  - Total size of locality > total memory size

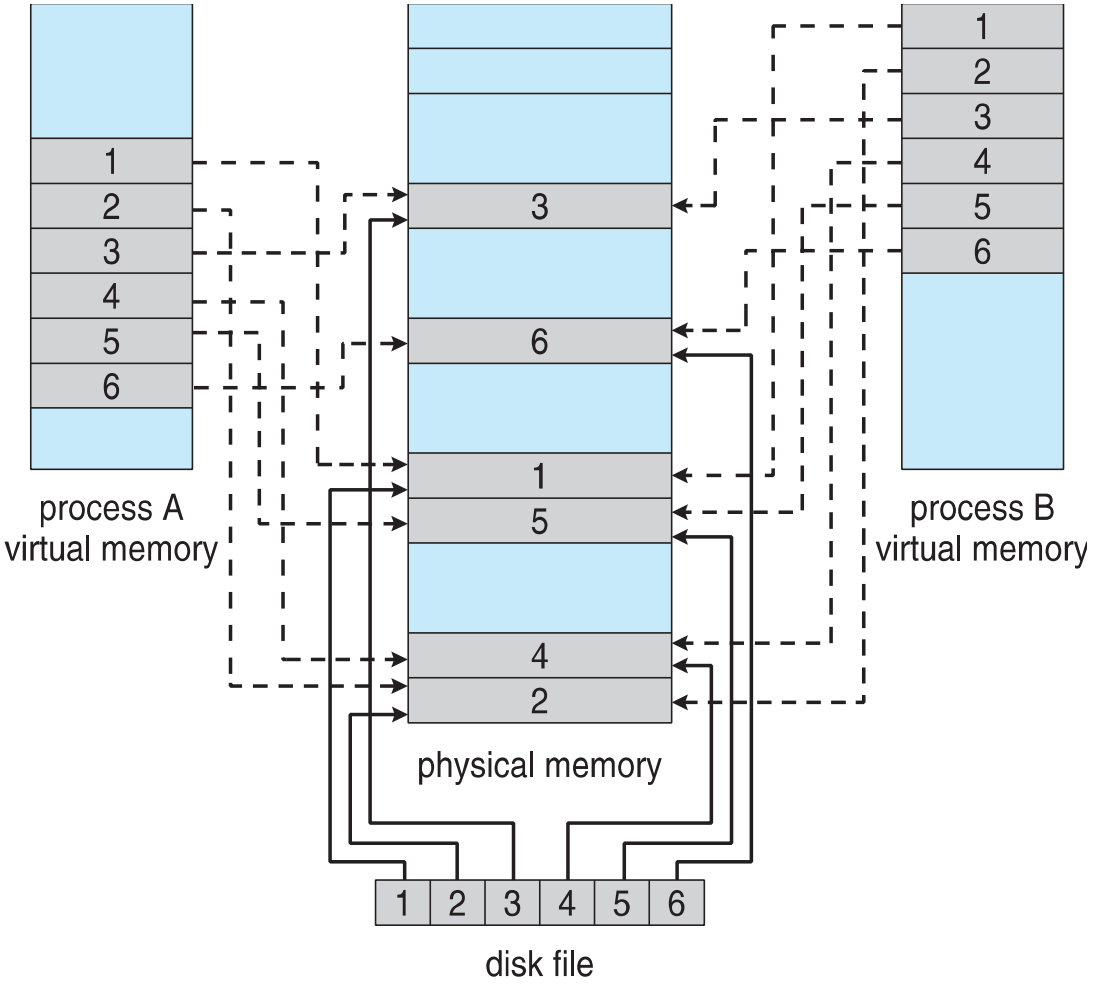
Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by *mapping* a disk block to a page in memory.
- A file is initially read using demand paging.
  - A page-sized portion of the file is read from the file system into a physical page.
  - Subsequent reads from (or writes to) the file are treated as ordinary memory accesses.
- This simplifies and speeds file access by driving file I/O through memory rather than read() and write() system calls.
  - It also allows several processes to map the same file allowing the pages in memory to be shared.
- But when does written data make it to disk?
  - Periodically and/or at file close() time
  - For example, when the pager scans for dirty pages.

Memory-Mapped Files

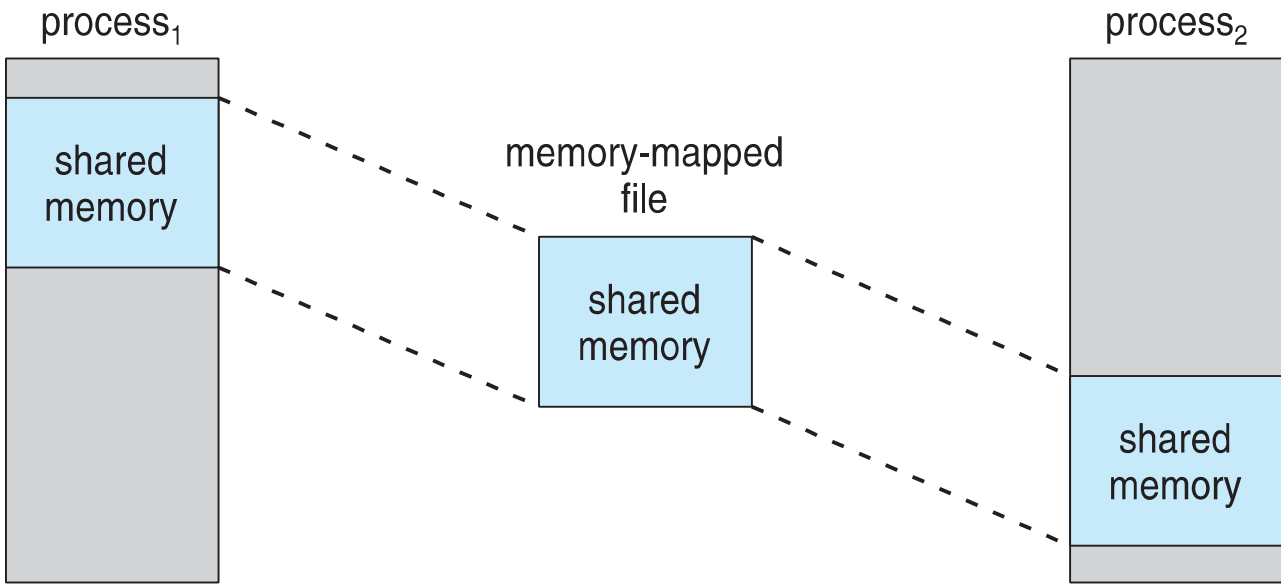
- Some operating systems use memory mapped files for standard I/O.
  - Process can explicitly request memory mapping a file via mmap() system call.
    - Now the file is mapped into process address space.
  - For standard I/O (open(), read(), write(), close()), mmap anyway
    - But map file into kernel address space.
    - Process still does read() and write()
      - copies data to and from kernel space and user space
    - Uses efficient memory management subsystem
      - avoids needing separate subsystem
  - Copy-on-Write (COW) can be used for read/write non-shared pages.
  - Memory mapped files can be used for shared memory (although again via separate system calls).

Memory-Mapped Files



Memory-Mapped Files

- Shared Memory via Memory-Mapped I/O.



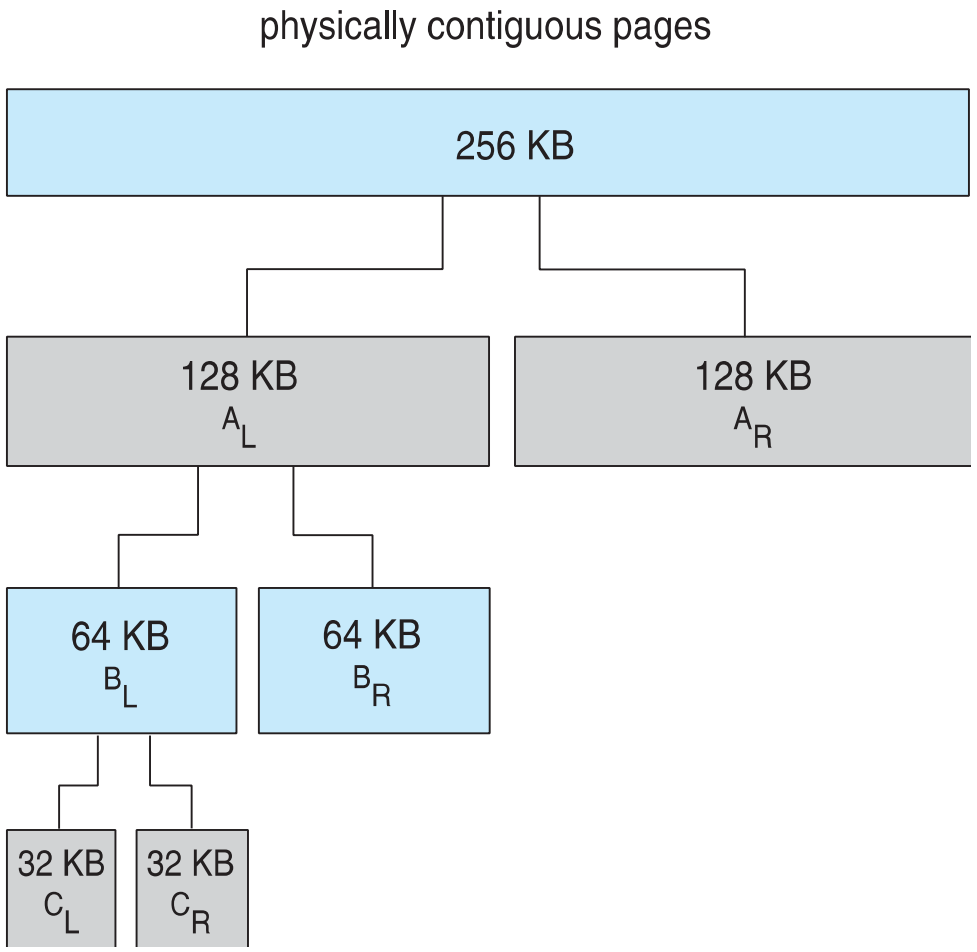
Memory-Mapped Files

- Shared Memory in Windows API
  - First create a *file mapping* for file to be mapped
    - Then establish a view of the mapped file in process's virtual address space
  - Consider producer / consumer
    - Create shared-memory object using memory mapping features by producer
    - Open file via CreateFile(), returning a HANDLE
    - Create mapping via CreateFileMapping() creating a *named shared-memory object*
    - Create view via MapViewOfFile()
  - Sample code in Textbook

Buddy System Allocator

- Buddy system allocates memory from fixed-size segment consisting of physically-contiguous pages.
- Memory allocated using *power-of-2 allocator*
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2.
    - Continue until appropriate sized chunk available
- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into A_L and A_R of 128KB each
    - One further divided into B_L and B_R of 64KB
      - One further into C_L and C_R of 32KB each – one used to satisfy request
- Advantage
  - quickly *coalesce* unused chunks into larger chunk
- Disadvantage
  - Fragmentation

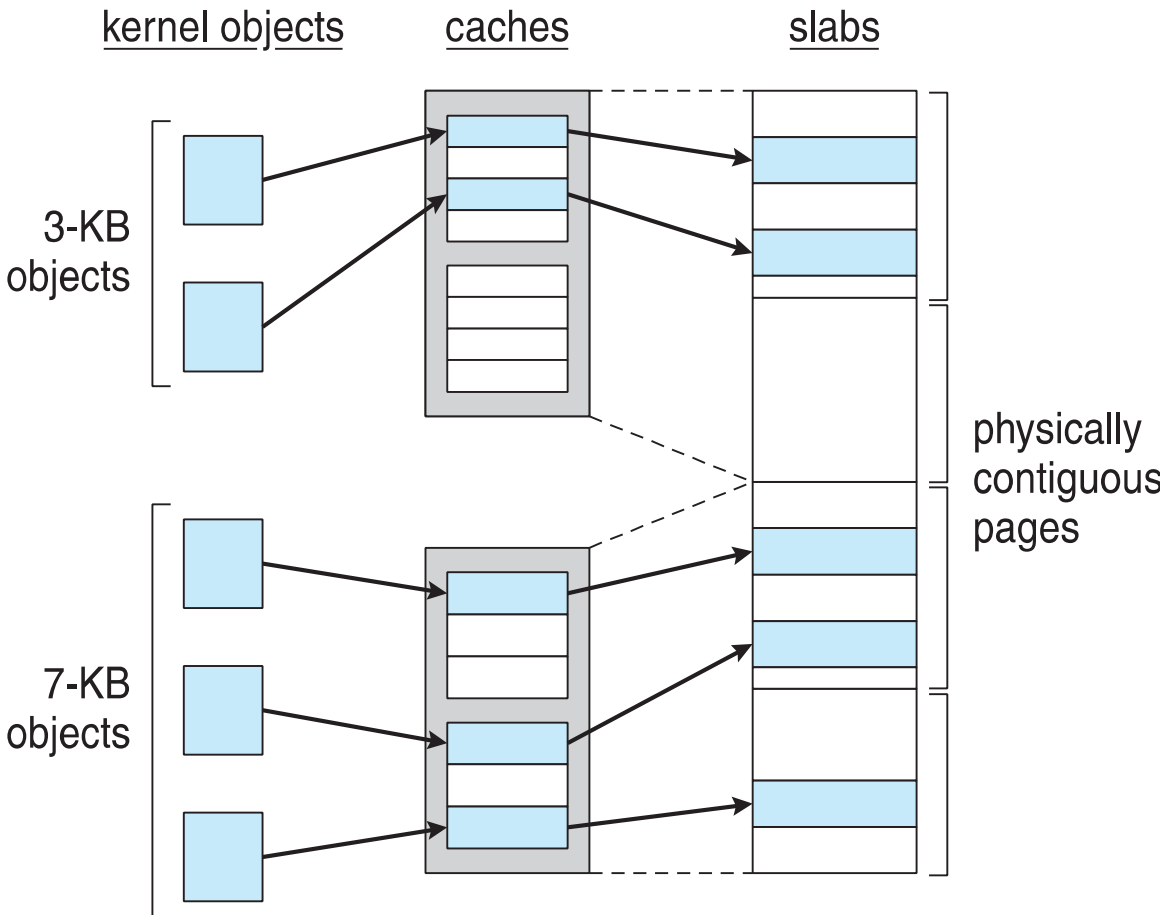
Buddy System Allocator



Slab Allocator

- Slab allocation is an alternate strategy for allocating kernel memory.
  - A slab is made up of one or more physically contiguous pages.
  - A cache consists of one or more slabs.
  - There is a single cache for each unique kernel data structure
    - E.g., a separate cache for the data structure representing process descriptors, a separate cache for file objects, a separate cache for semaphores, and so forth.
- Each cache is populated with objects that are instantiations of the kernel data structure the cache represents.
  - E.g., the cache representing semaphores stores instances of semaphore objects, the cache representing process descriptors stores instances of process descriptor objects, and so forth.
- The relationship among slabs, caches, and objects is shown in the next slide. The figure shows two kernel objects of 3KB in size and three objects of 7KB in size, each stored in a separate cache.

Slab Allocator



Slab Allocator

- When the kernel requests memory from the slab allocator for an object, the slab allocator first attempts to find a slab consisting free objects to satisfy the request.
  - If none exists, a free object is assigned from an empty slab.
  - If no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache; memory for the object is allocated from this new slab.
- The slab allocator provides two main benefits:
  - No memory is wasted due to fragmentation.
    - Each unique kernel data structure has an associated cache, and each cache is made up of one or more slabs that are divided into chunks *the size of the objects being represented*.
  - Memory requests can be satisfied quickly.
    - The slab allocation scheme is particularly effective for managing memory when objects are frequently allocated and deallocated. Objects are created in advance and thus can be quickly allocated from the cache and released to its cache by being marked as free.

Slab Allocator

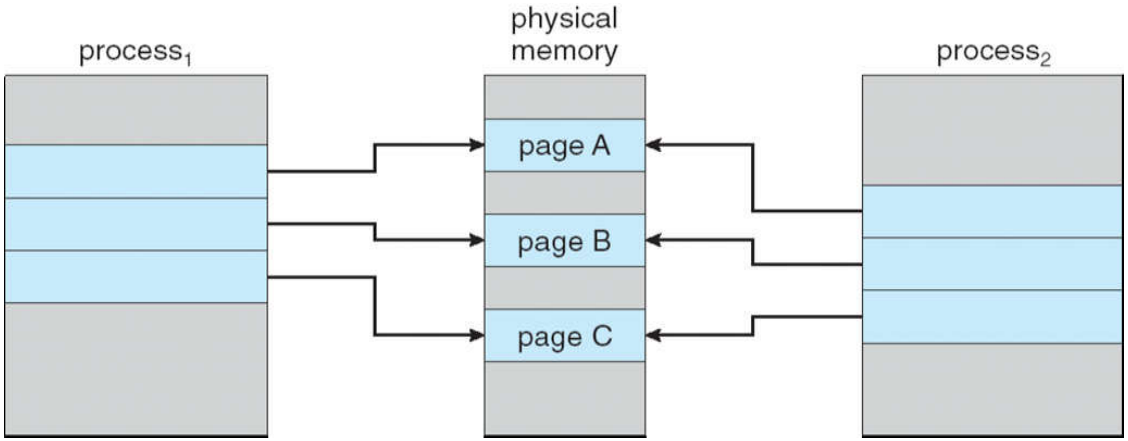
- Slab Allocator in Linux
  - Example: a process descriptor in Linux is of type struct task_struct
    - Approx 1.7KB of memory
    - New task -> allocate new task_struct from cache
    - Will use existing free task_struct
  - A slab can be in one of three possible states
    - Full – all objects are used
    - Empty – all objects are free
    - Partial – mix of free and used objects
  - Upon request, slab allocator
    (1) Uses free task_struct in a partial slab
    (2) If none, takes one from an empty slab
    (3) If no empty slab, create a new empty slab.

Copy-on-Write

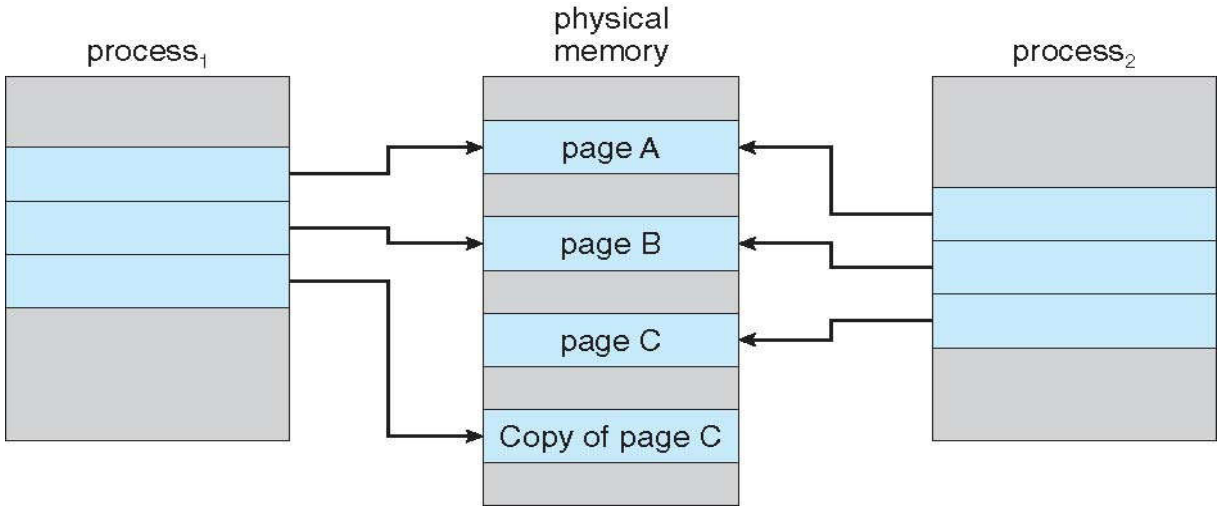
- Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory.
  - If either process modifies a shared page, only then is the page copied.
  - COW allows more efficient *process creation* as only modified pages are copied.
- In general, free pages are allocated from a pool of zero-fill-on-demand pages.

Copy-on-Write

- Before Process 1 Modifies Page C



- After Process 1 Modifies Page C



Other Issues

- Prepaging
  - Prepaging can help to reduce the large number of page faults that occurs at process startup or resumption.
    - Prepage all or some of the pages a process will need, before they are referenced.
    - If prepaged pages are unused, I/O and memory was wasted.
  - Assume s pages are prepaged and a fraction α of these pages are used:
    - Is cost of $s \times \alpha$ saved pages faults greater or less than the cost of prepaging $s \times (1- \alpha)$ unnecessary pages*?*
    - α near zero $\Rightarrow$ prepaging loses

Other Issues

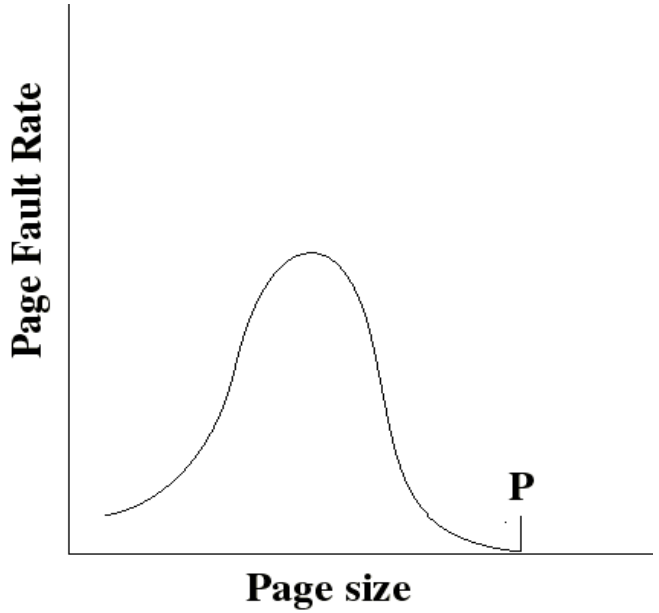
- Page Size
  - Sometimes OS designers have a choice:
    - Especially if running on custom-built CPU.
  - Page size selection must take into consideration:
    - Fragmentation
    - Page table size
    - I/O overhead
    - Number of page faults
    - Locality
    - TLB size and effectiveness
  - Always power of 2, usually in the range 2^{12} (4KB) to 2^{22} (4MB).
  - On average, growing over time.

Other Issues

- Page Size
  - Page size is defined by hardware; exact size to use is difficult.
  - Large page size is good since for a small page size, more pages are required per process; more pages per process means larger page tables.
    - a larger portion of page tables in virtual memory.
  - Large page size is good since disks are designed to efficiently transfer large blocks of data.
  - Larger page sizes means less pages in main memory; this increases the TLB hit ratio.
  - Small page size is good to minimize internal fragmentation.

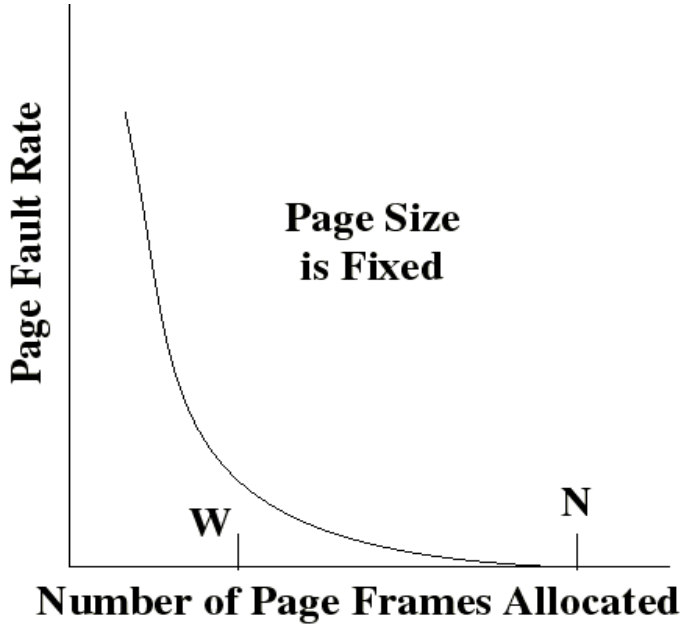
Other Issues

- Page Size
  - With a very small page size, each page matches the code that is actually used. Page fault rate is low.
  - Increased page size causes each page to contain more code that is not used. Page fault rate rises.
  - Page fault rate decreases if we approach point P where the size of a page is equal to the size of the entire process.



Other Issues

- Page Size
  - Page fault rate is also determined by the number of frames allocated per process.
  - Page faults drops to a reasonable value when W frames are allocated.
  - Drops to 0 when the number (N) of frames is such that a process is entirely in memory.



Other Issues

- Page Size
  - Page sizes from 1KB to 4KB are most commonly used. Increase in page sizes is related to trend of increasing block sizes.
  - But the issue is non trivial. Hence some processors supported multiple page sizes, for example:
    - Pentium supports 2 sizes: 4KB or 4MB
    - MIPS R4000 supports 7 sizes: 4KB to 16MB

Computer	Page Size
Atlas	512 48-bit words
Honeywell-Multics	1024 36-bit words
IBM 370/XA and 370/ESA	4 Kbytes
VAX family	512 bytes
IBM AS/400	512 bytes
DEC Alpha	8 Kbytes
MIPS	4 Kbytes to 16 Mbytes
UltraSPARC	8 Kbytes to 4 Mbytes
Pentium	4 Kbytes or 4 Mbytes
IBM POWER	4 Kbytes
Itanium	4 Kbytes to 256 Mbytes

Other Issues

- TLB Reach
  - TLB Reach (范围) is the amount of memory accessible from TLB.

$$\text{TLB\_Reach} = (\text{TLB\_Size}) \times (\text{Page\_Size})$$

    - Ideally, working set of each process is stored in the TLB Reach.
  - Increase the size of the TLB
    - might be expensive
  - Increase the Page Size
    - ref. to “Other Issues – Page Size”
  - Provide Multiple Page Sizes:
    - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation.

Other Issues

- Program Structures
  - Example.
    - In C++:

```
int A[][] = new int[1024][1024];
```

assuming each row is stored in one page.

- Program 1:

```
for (j = 0; j < A.length; j++)
    for (i = 0; i < A.length; i++)
        A[i,j] = 0;
```

  - It may be 1024 x 1024 page faults.

- Program 2:

```
for (i = 0; i < A.length; i++)
    for (j = 0; j < A.length; j++)
        A[i,j] = 0;
```

  - It may be 1024 page faults.

Other Issues

- I/O Interlock
  - I/O Interlock – Pages must sometimes be locked into memory.
    - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm.