Memory Management

- Basic Concepts
  - Memory allocation
  - Swapping
- Virtual Memory
- Paging & Segmentation
  - Demand Paging
  - Page replacement algorithms
  - Working set model
- Linux Memory Management

Memory Management

Memory is key component of computer

- e.g. every instruction cycle involves memory access

Memory management needs to provide:

- Memory allocation
- Memory protection

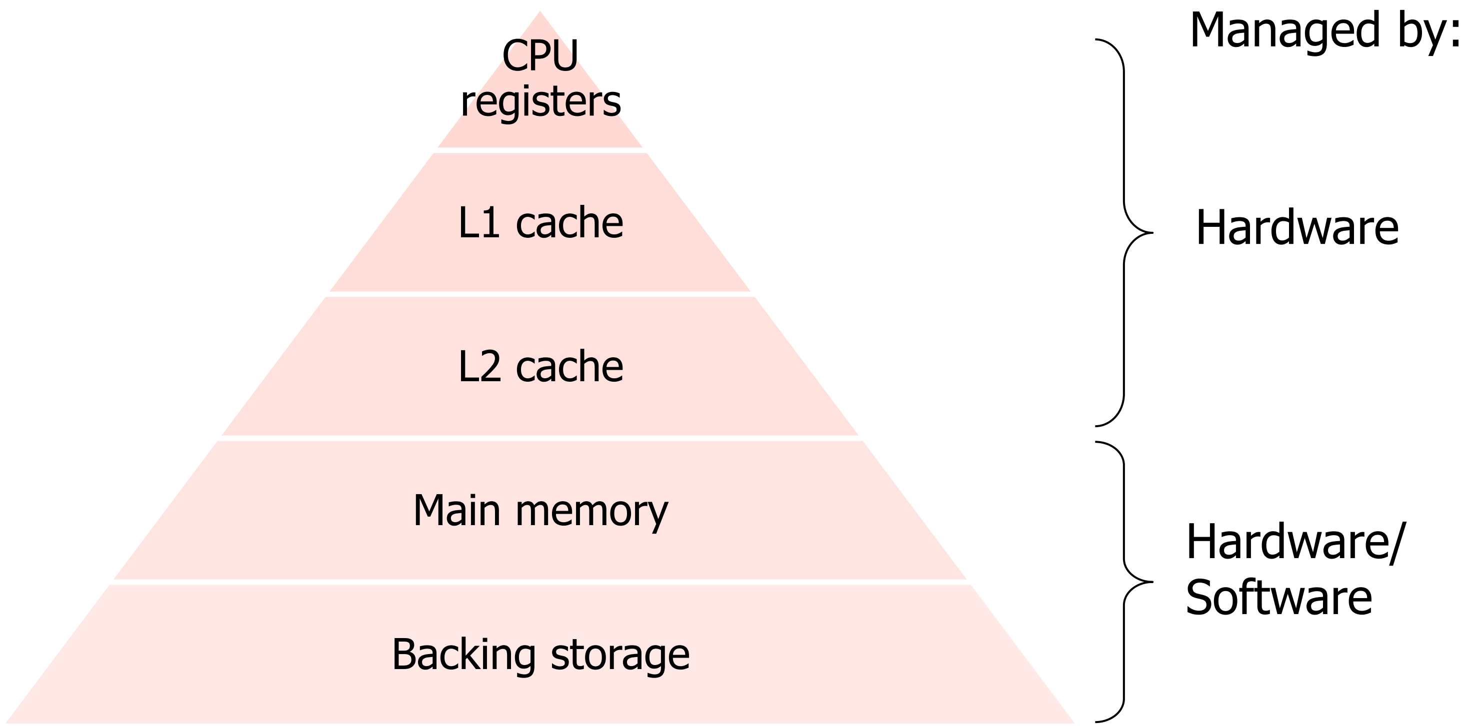
Characteristics

- No knowledge how memory addresses generated
  - e.g. instruction counter, indexing, indirection, …
- No knowledge what memory addresses used for
  - e.g. instructions or data
- True for simple case but may want protection with respect to read, write execute etc

Memory Hierarchy

Hardware: CPU registers and main memory

- Register access in one CPU clock cycle (or less)
- Main memory can take many cycles
- Caches sit between main memory and CPU registers



Logical vs. Physical Address Space

Memory management binds logical address space to physical address space

- **Logical address**
  - Generated by the CPU
  - Address space seen by process
- **Physical address**
  - Address seen by the memory unit
  - Refers to physical system memory

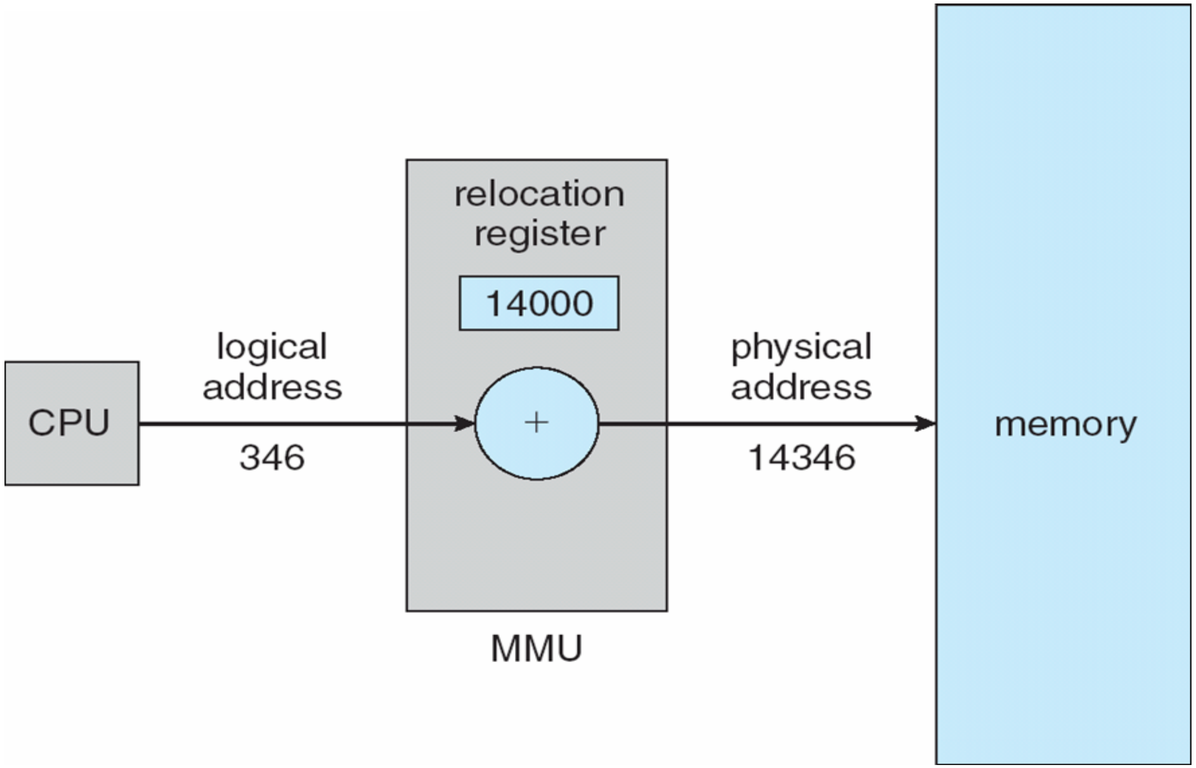
Logical and physical addresses:

- same in compile- and load-time address-binding schemes
- different in execution-time address-binding scheme

Memory-Management Unit (MMU)

Hardware device for mapping logical to physical addresses

- e.g. add value in relocation register to every address generated by process when sent to memory
- User process deals with logical addresses only
- Has to be fast ➔ implemented in hardware



Contiguous Memory Allocation I

Main memory usually split into two partitions:

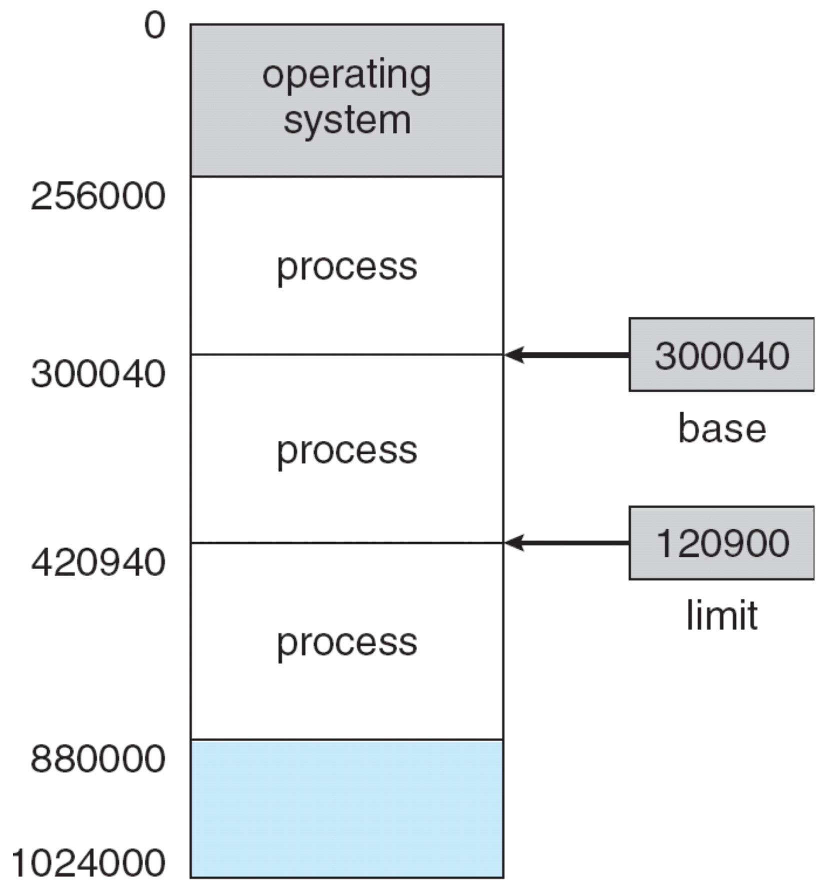
- Resident operating system (**kernel**)
  - Usually held in low memory with interrupt vector
- User processes (**user**)
  - Held in high memory

Contiguous allocation with *relocation registers*

- Base register contains physical start address for process
- Limit register contains maximum logical address for process
- MMU maps logical address dynamically
  - If logical address > limit then error
  - Physical address = logical address + base

Contiguous Memory Allocation II

Base and limit registers define logical address space



e.g. JMP 100 would go to location 300140

Multiple-Partition Allocation

Hole

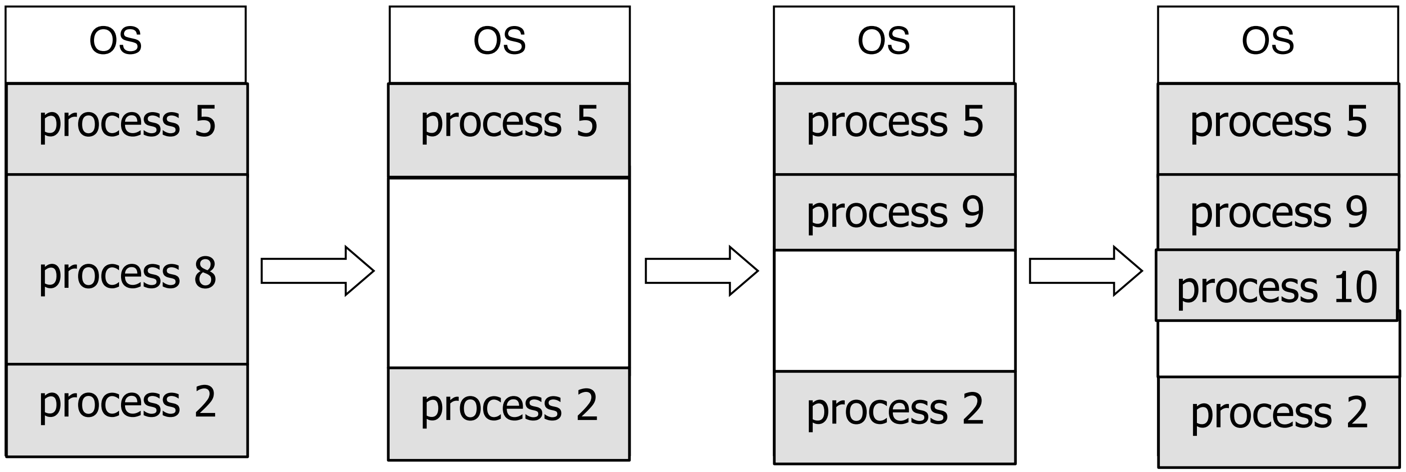
- Block of available memory
- Holes of various size scattered throughout memory

When new process arrives:

- allocate memory from hole large enough

OS maintains information about:

a) allocated partitions b) free partitions (holes)



Dynamic Storage Allocation

How to satisfy request of size n from list of free holes:

First-fit: Allocate first hole that is big enough

Best-fit: Allocate smallest hole that is big enough

- Must search entire list, unless ordered by size
- Produces smallest leftover hole

Worst-fit: Allocate largest hole

- Must also search entire list
- Produces largest leftover hole

Why best-fit or worst fit?

☛ First-fit and best-fit *better* than worst-fit in terms of speed and storage utilisation

Fragmentation

External fragmentation

- Total memory exists to satisfy request, but not contiguous

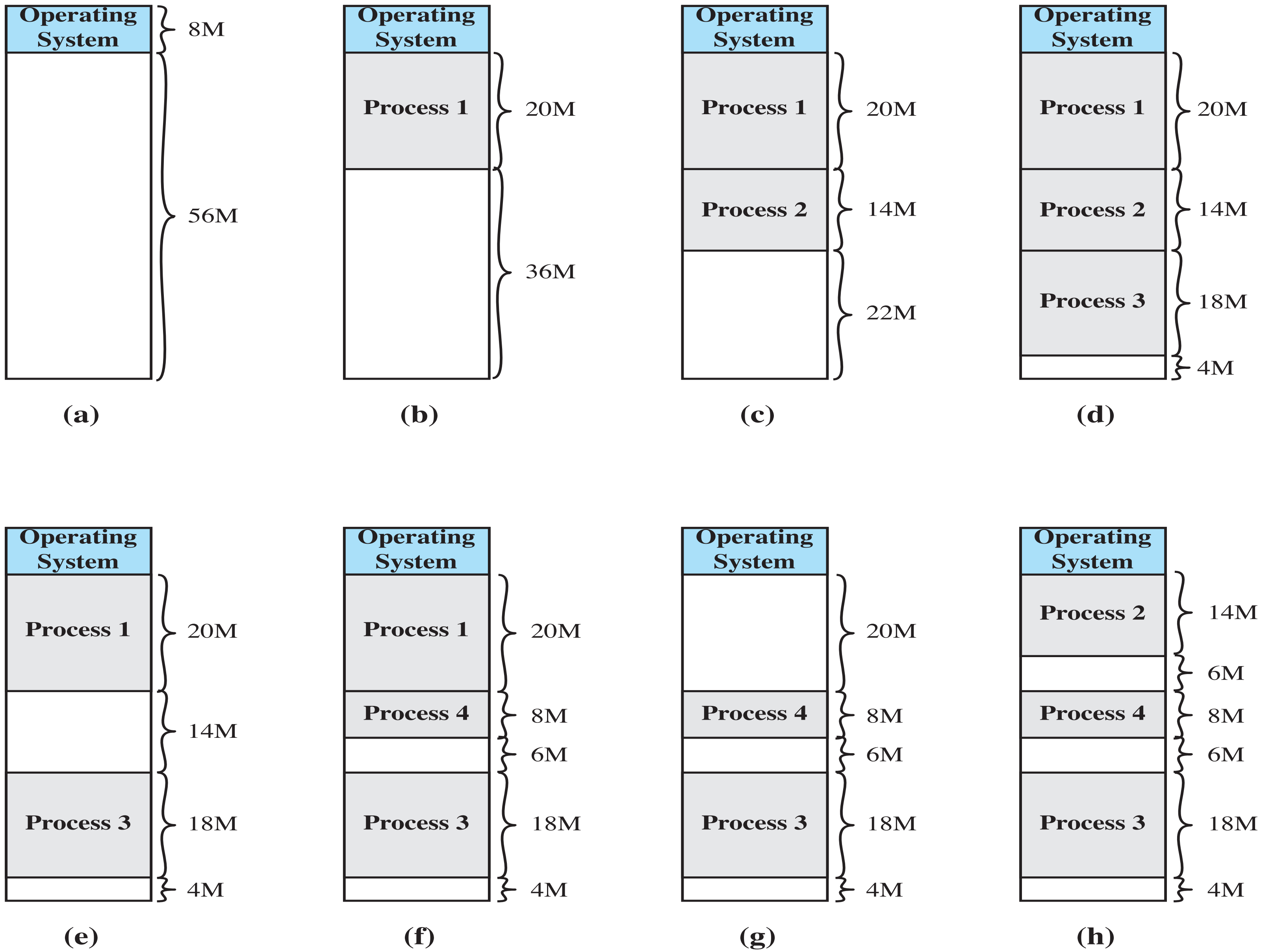
Internal fragmentation

- Allocate in multiples of block size e.g. 4KB.
- Allocated memory larger than requested memory
- Size difference internal to partition ➔ not used

Reduce external fragmentation by **compaction**

- Shuffle memory contents to place all free memory together in one large block
- Leads to I/O bottlenecks

External Fragmentation



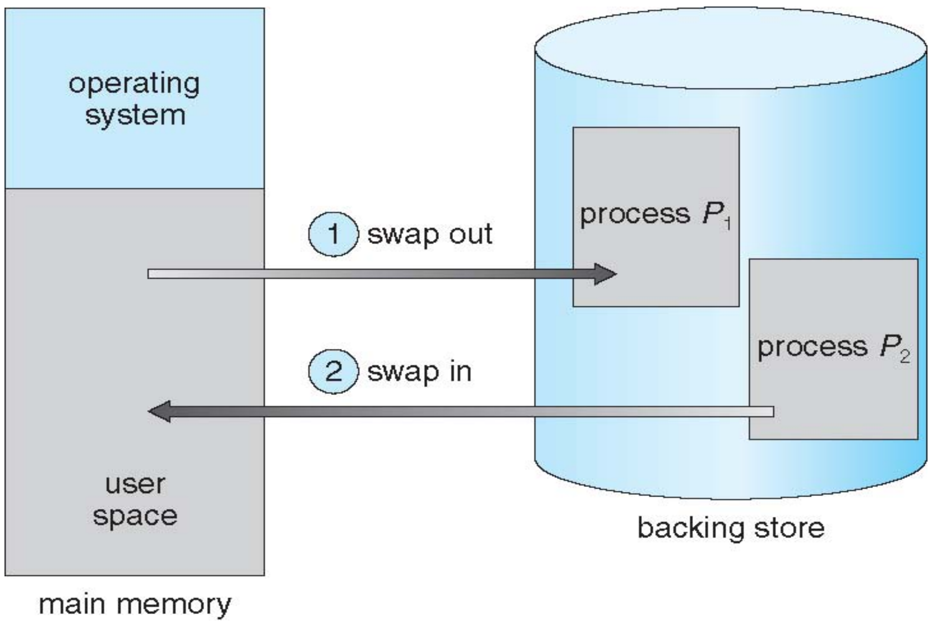
Swapping

Problem: Number of processes limited by amount of available memory

- But: only running processes need to be in memory

Solution:

- **Swap** processes temporarily out of memory to backing store
- Bring back into memory for continued execution
- Requires **swap space**
  - Can be file or dedicated partition on disk
- Transfer time major part of swap time

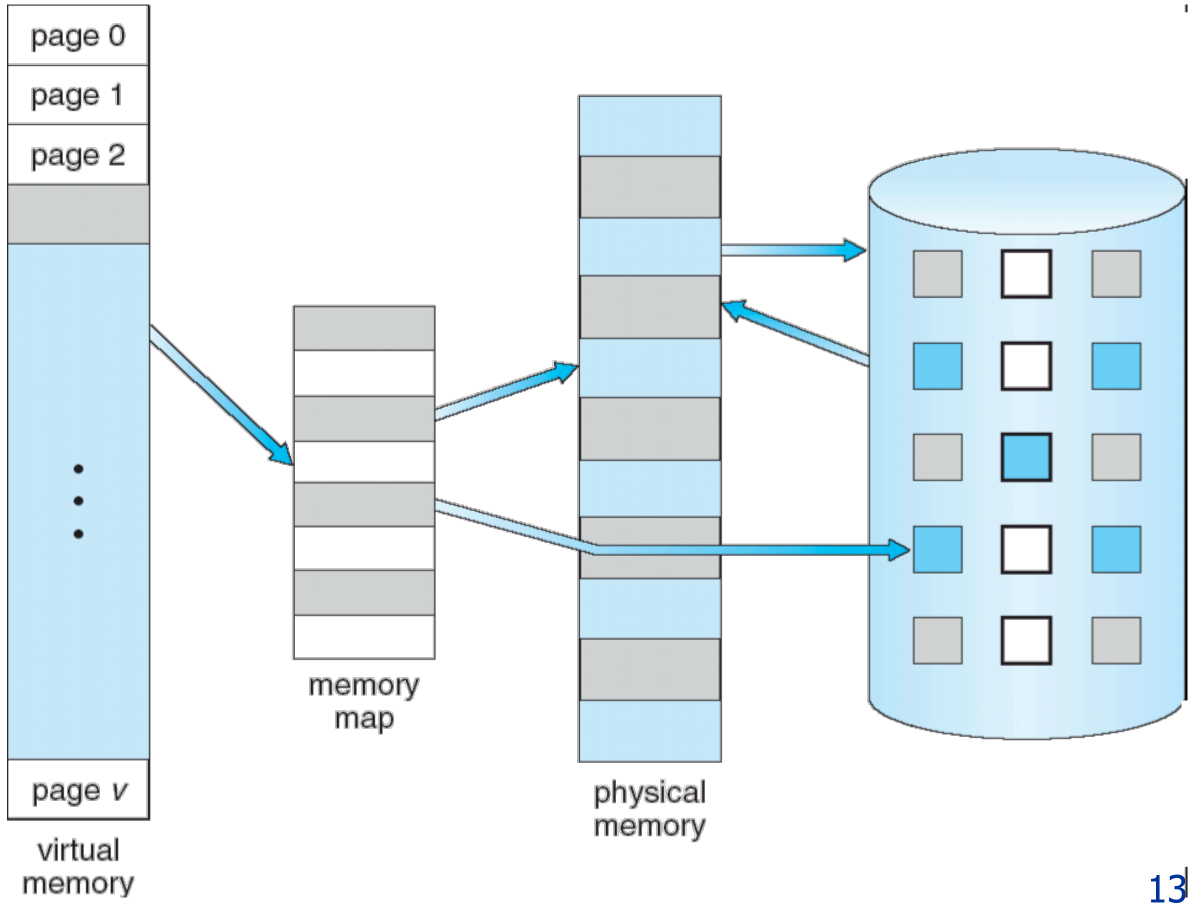


Virtual Memory with Paging

Virtual Memory

Separation of user logical memory from physical memory

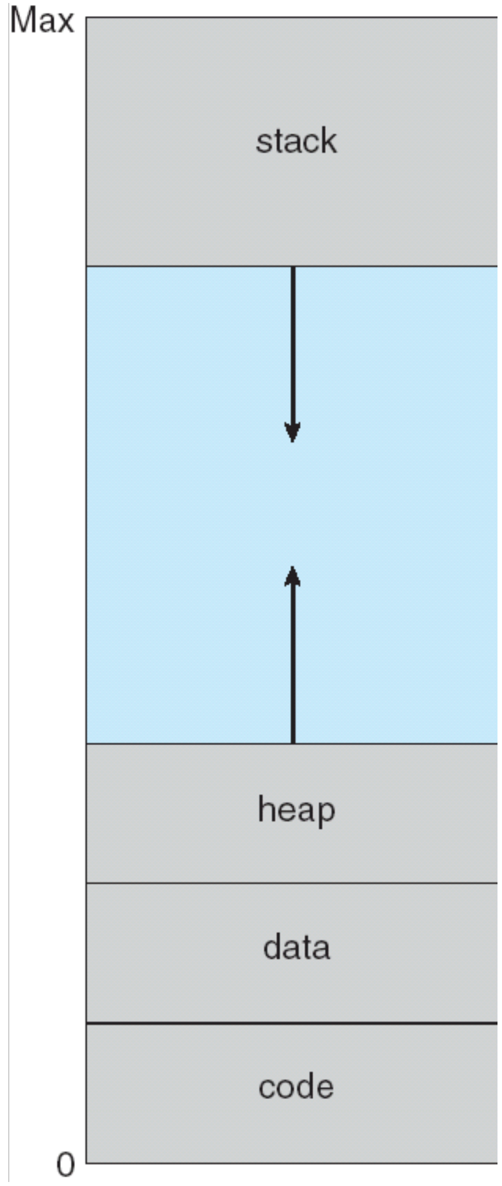
- Only part of process needs to be in memory for execution
- Logical address space can be *much* larger than physical address space
- Address spaces can be shared by several processes
- Allows for more efficient process creation



Virtual Address Space

Virtual memory can be implemented via:

- **Paging**
- **Segmentation**



Paging

Physical address space of process can be noncontiguous

- Process allocated physical memory when available
  - Avoid external fragmentation
  - Avoid problems of variable sized memory chunks

Frames

- Fixed-sized blocks of physical memory
- Keep track of all free frames

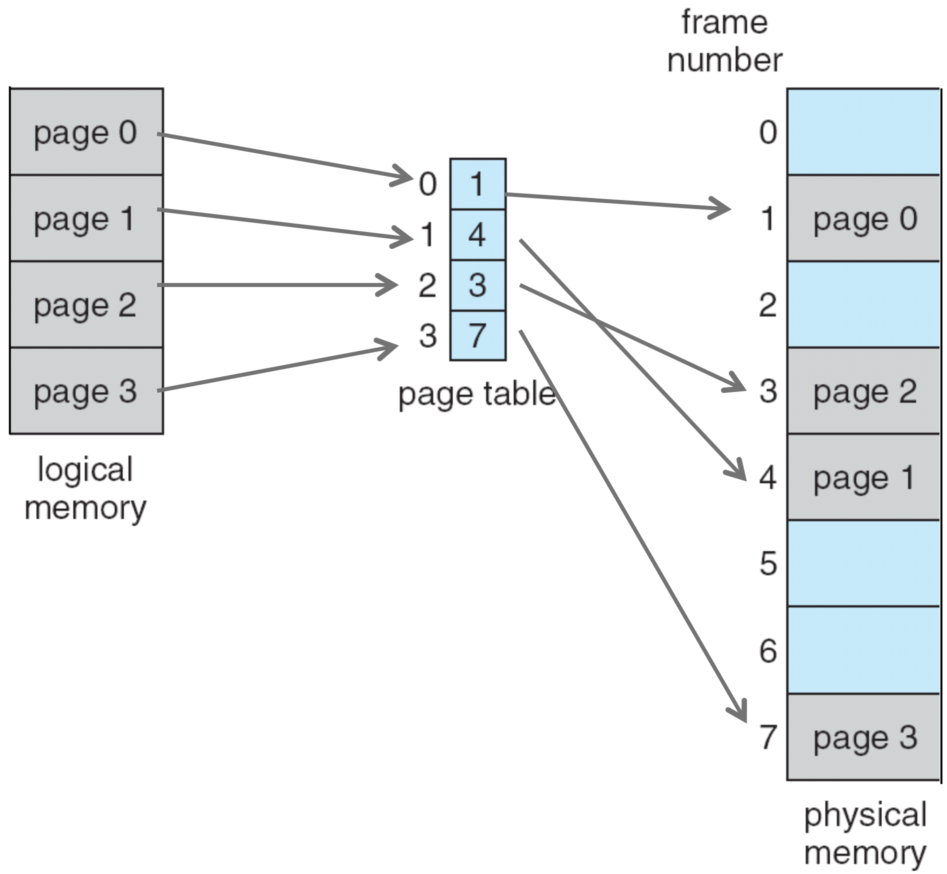
Pages

- Block of same size of logical memory

To run program of size *n* pages

- Find *n* free frames and load program
- Set up **page table** to translate logical to physical addresses

Page Table



frame number
page 0
page 1
page 2
page 3
logical memory
0
1
2
3
1
4
3
7
page table
0
1
2
3
4
5
6
7
page 0
page 2
page 1
page 3
physical memory

Address Translation I

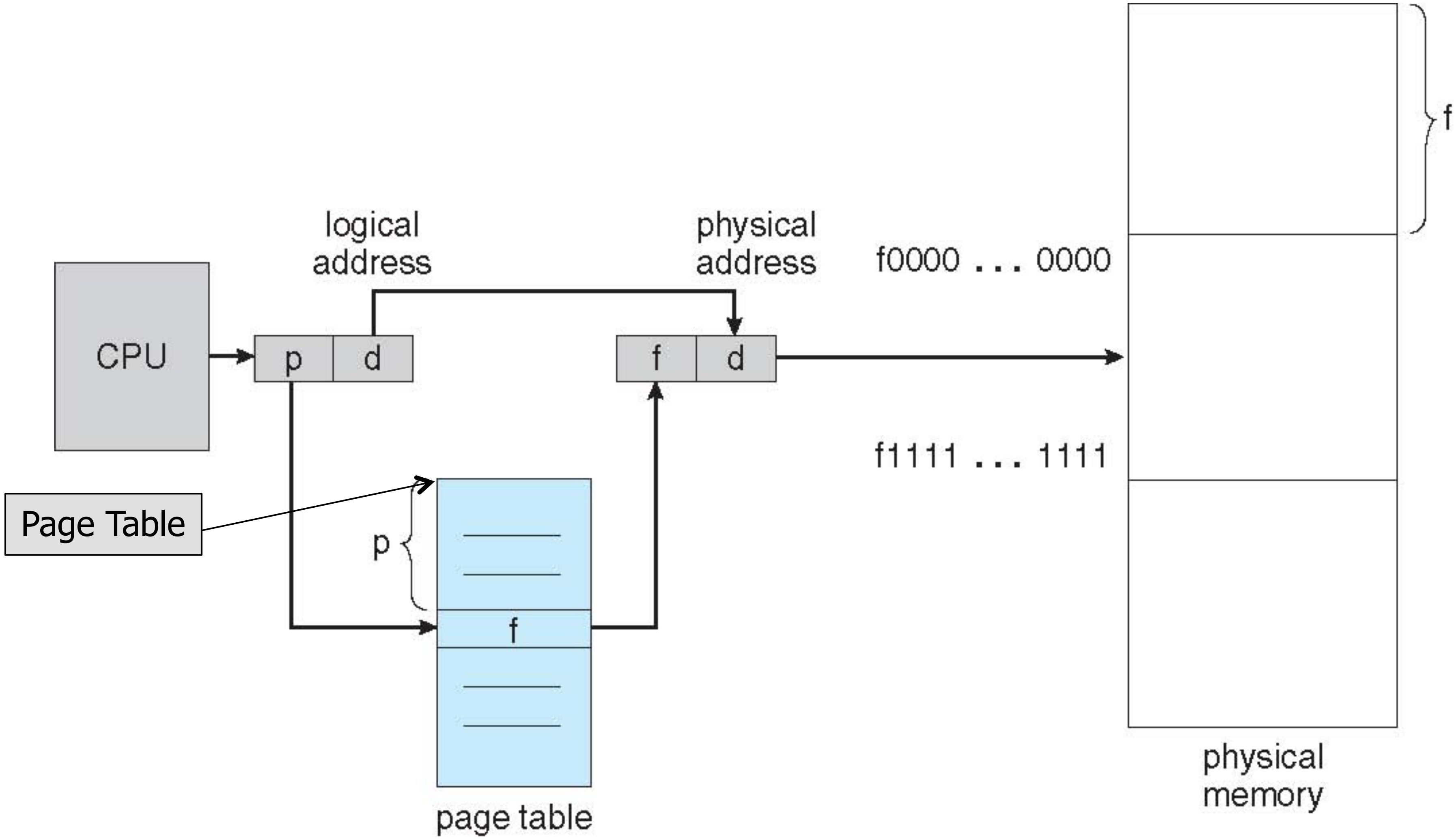
Address generated by CPU divided into:

- **Page number** (p)
  - Used as index into page table
  - Page table has base address of pages in physical memory
- **Page offset** (d)
  - Defines physical memory address sent to the memory unit
  - Combined with base address

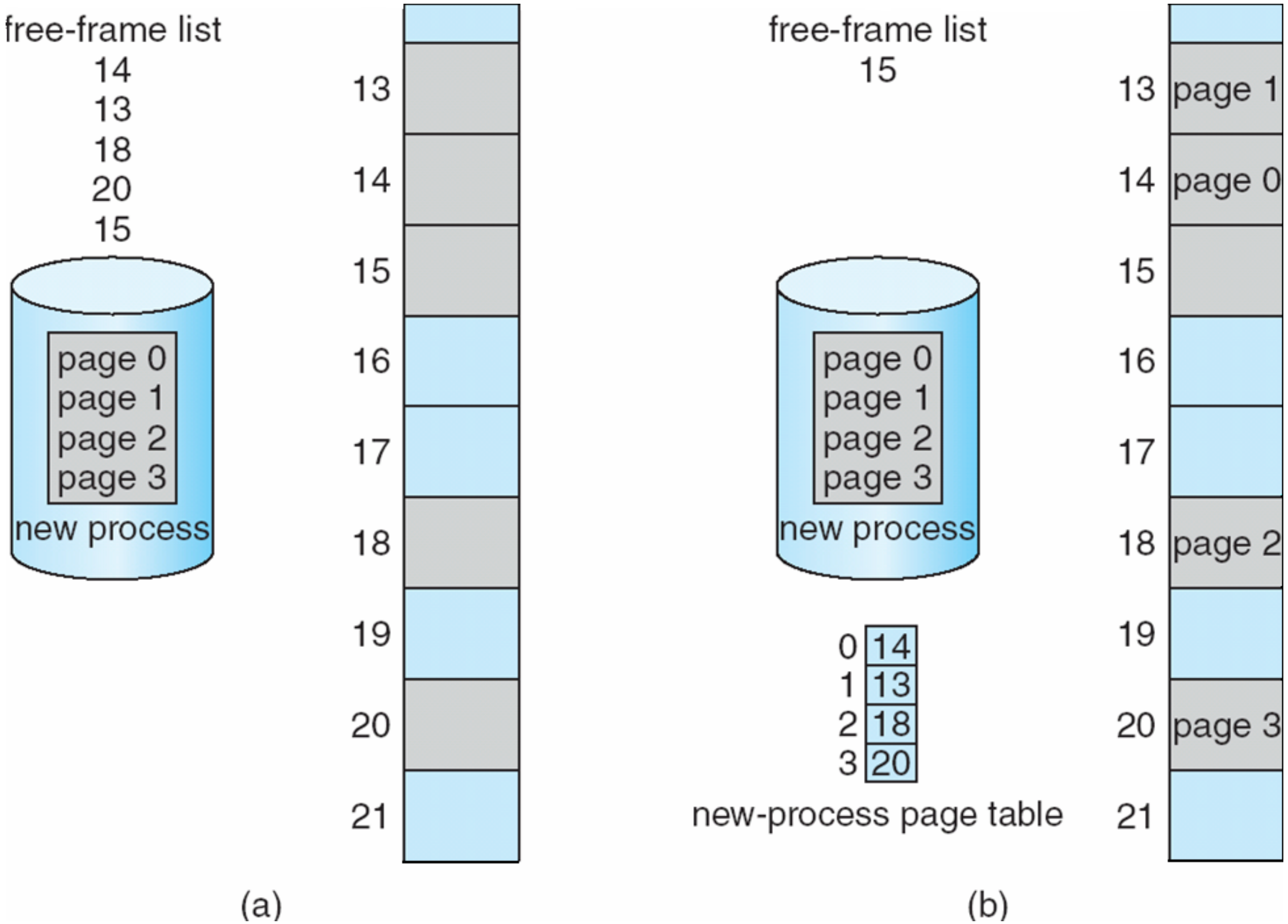
For given logical address space 2^m and page size 2^n

page number	page offset
p	*d*
$m - n$	n

Paging Hardware



Free Frames



Before allocation

After allocation

Memory Control Bits

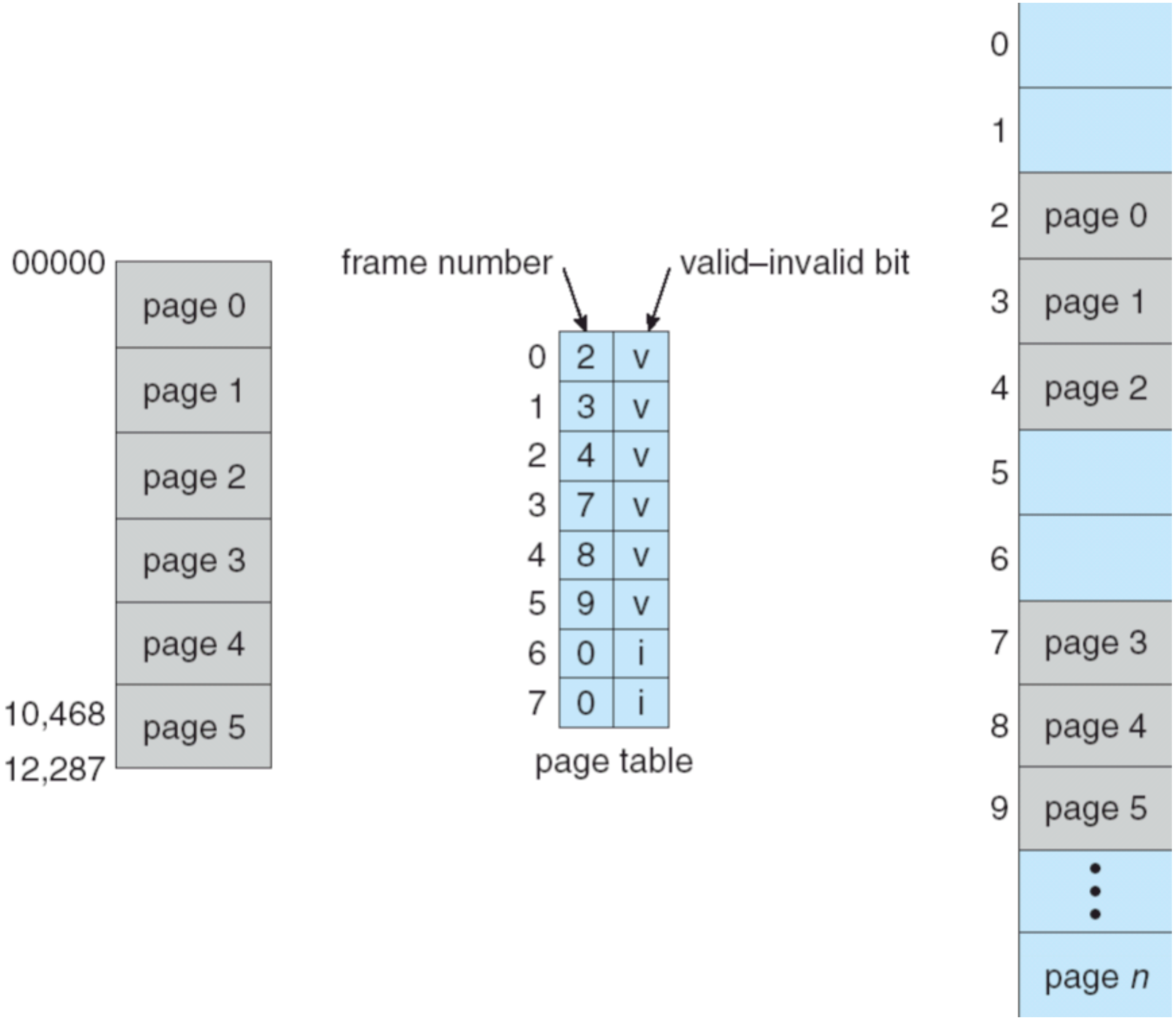
Protection: protection bits associated with a frame indicate read-only, read-write, execute only

Valid-invalid bit attached to each page table entry:

- **Valid** indicates page present
  - Associated page is in physical memory
- **Invalid** indicates page missing
  - Page not in physical memory i.e. page fault
  - Kernel trap to bring in page from backingstore

Page replacement Bits: to indicate if page has been modified or referenced (see later).
Also lock bit to prevent page being transferred out.

Memory Validity



Paging and Fragmentation

Calculating internal fragmentation

- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages + 1,086 bytes
- Internal fragmentation of 2,048 - 1,086 = 962 bytes
- Worst case fragmentation = 1 frame -1 byte
- On average fragmentation = 1/2 frame size
- So small frame sizes desirable?
- But each page table entry takes memory to track
- Page size growing over time
  - Typically 4KB but some architectures support variable page size up to 256MB

Process view and physical memory now very different

By implementation process can only access its own memory

Page Table Implementation

Page table kept in main memory

- **Page-table base register** (PTBR) points to page table
  Context switch requires update of PTBR for new process page table.
- **Page-table length register** (PRLR) indicates size

Problem: inefficient

- Every data/instruction access requires two memory accesses: one for page table and one for data/instruction

Associative Memory

Solution: Use special fast-lookup hardware cache as associative memory – also

Associative memory: Supports parallel search

Page #	Frame #

Called *Translation Look-aside Buffer (TLB)*

Address translation (p, d)

- If p in associative register, get frame # out
- Otherwise get frame # from page table in memory

Translation Look-aside Buffers (TLBs) 1

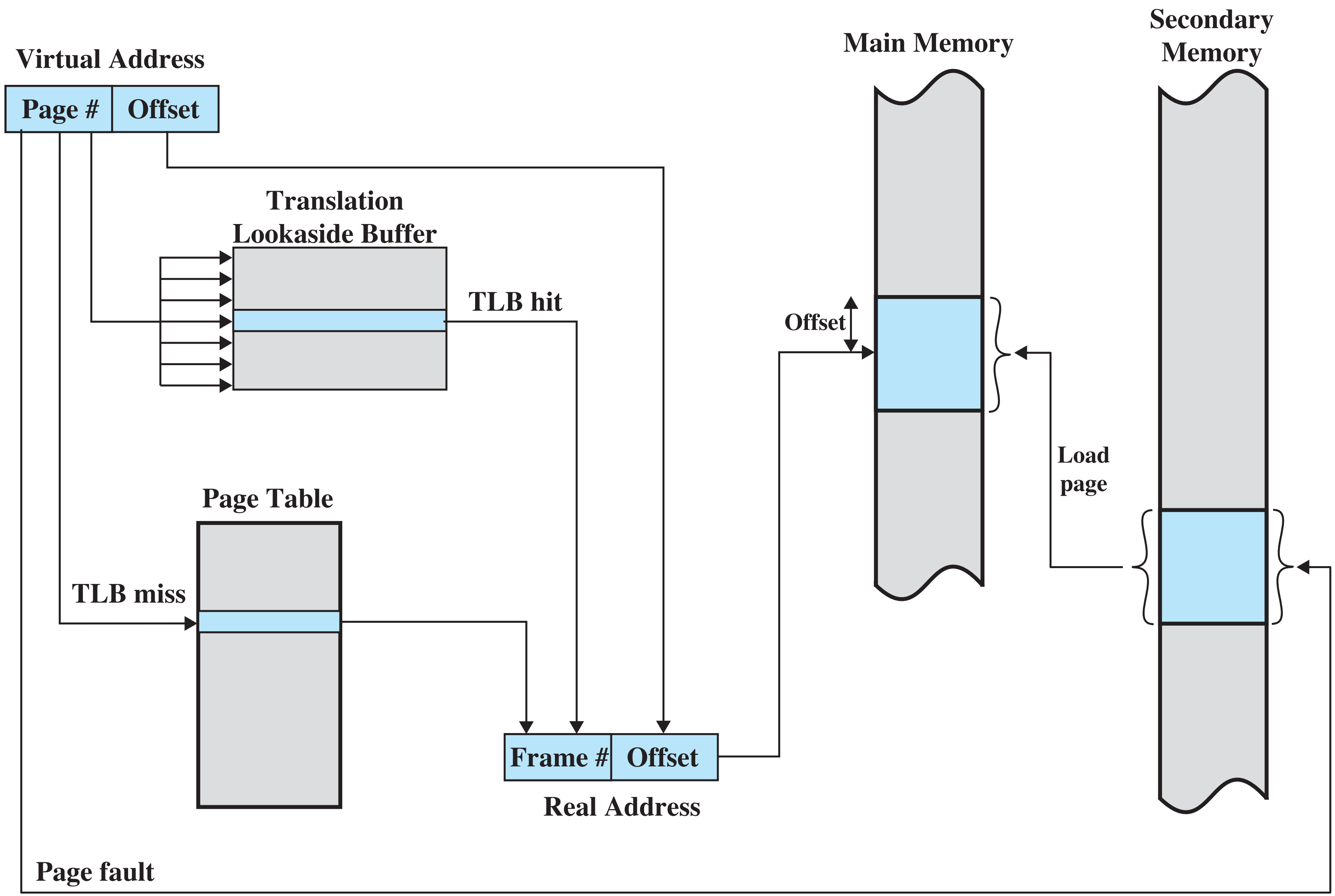
Some TLBs store address-space ids (ASIDs) in entries

- Uniquely identifies each process to provide address-space protection for that process

TLBs usually needs to be flushed after context switch

- Can lead to substantial overhead
- What about kernel pages for system calls?

Translation Look-aside Buffers (TLBs) 2



Performance: Effective Access Time

Associative Lookup = ε

- Can be < 10% of memory access time m

Hit ratio α

- Fraction of times that page found in associative registers
- Ratio related to number of associative registers

Effective Access Time (EAT)

$$EAT = (\varepsilon + m)\,\alpha + (\varepsilon + 2m)(1 - \alpha) = 2m + \varepsilon - m\alpha$$

Consider α =80%, ε = 10ns for TLB search, 100 ns for memory access

$$EAT = 110 \times 0.80 + 210 \times 0.20 = 130ns$$

A more realistic hit ration might be 99%

$$EAT = 110 \times 0.99 + 210 \times 0.01 = 111ns$$

Page Table Types

1. **Hierarchical** page table

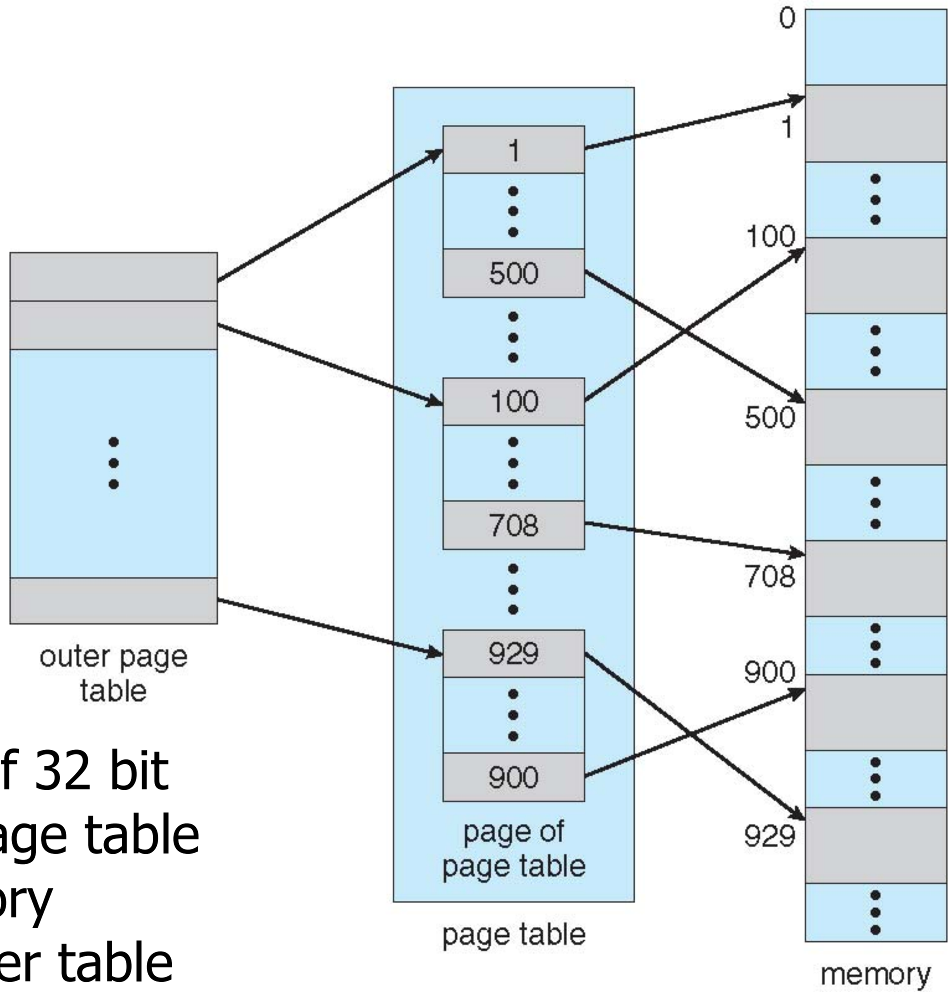
2. **Hashed** page table

3. **Inverted** page table

1. Hierarchical Page Table

Break up logical address space into multiple page tables

Simple technique: **two-level page table** for 32 bit address



- 2^{32} =4 Gb user address space

- 1024 entries of 32 bit = 4 Kb root page table Fixed in memory
- 4Mb paged user table

Two-Level Paging I

Logical address divided: (assuming 32-bit machine with 4K page size)

- Page number consisting of 20 bits
- Page offset consisting of 12 bits

Since page table paged, page number further divided:

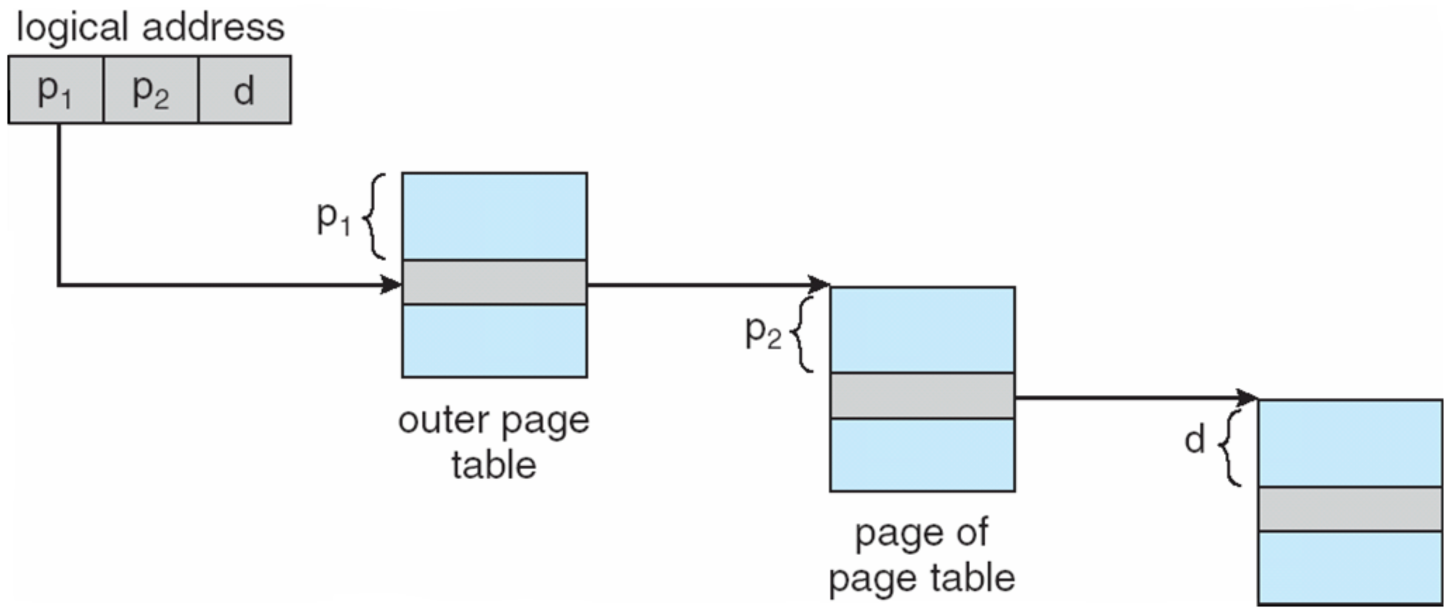
- 10-bit page number
- 12-bit page offset within 2nd level page table

Thus, logical addresses as follows:

page number		page offset
p_1	p_2	d
10	10	12

where p1 = index into the outer page table,
and p2 = displacement within page pointed to by outer page table

Two-Level Paging II



Page Table Size

On 32-bit machine with 4KB pages
- Page table will be at least 4MB

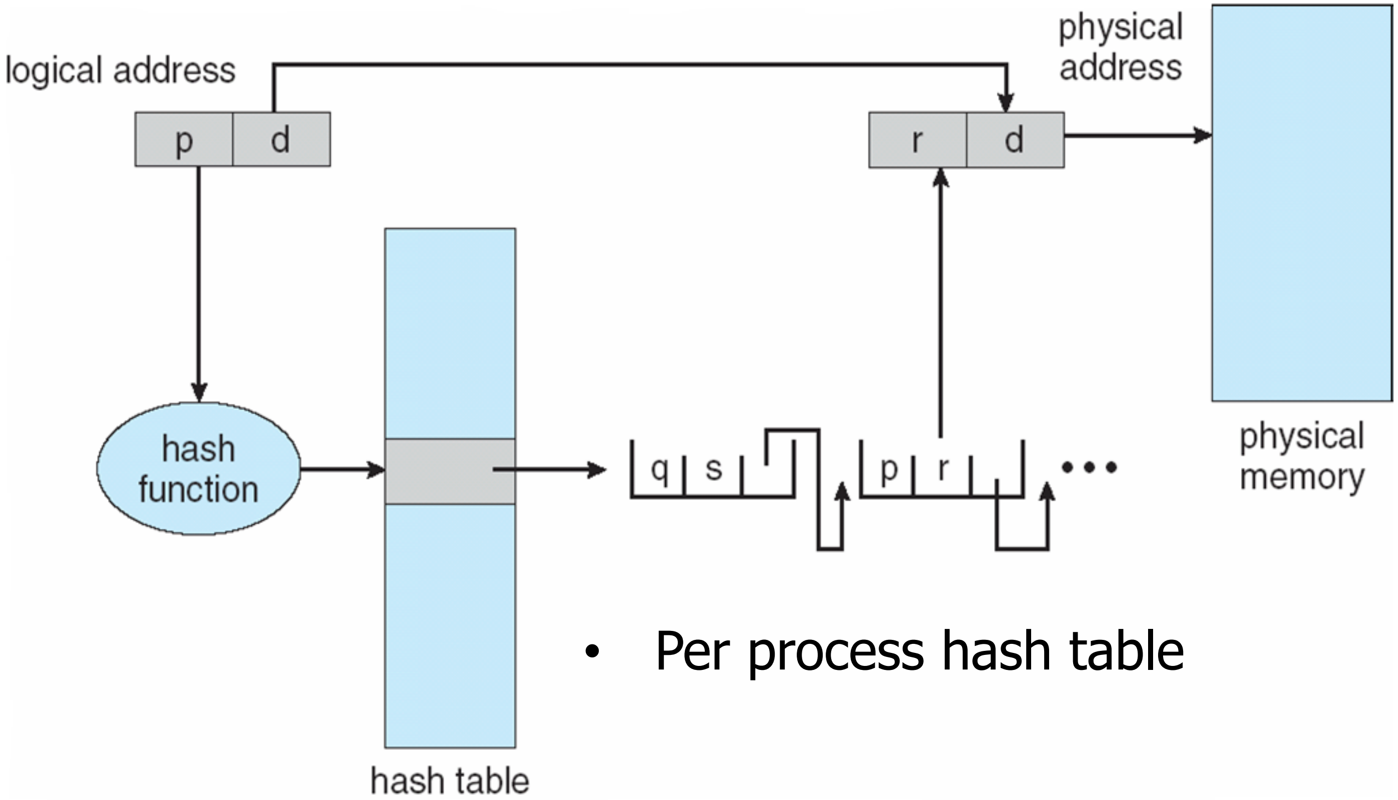
On 64-bit machine with 4KB pages
- Page table needs 2^{52} entries
- With 8 bytes per entry, that's 30 million GB…

Idea: don't store entry per page but per frame
- Use **hashed page table**
- Use **inverted page table**

2. Hashed Page Table

Hash virtual page number into page table

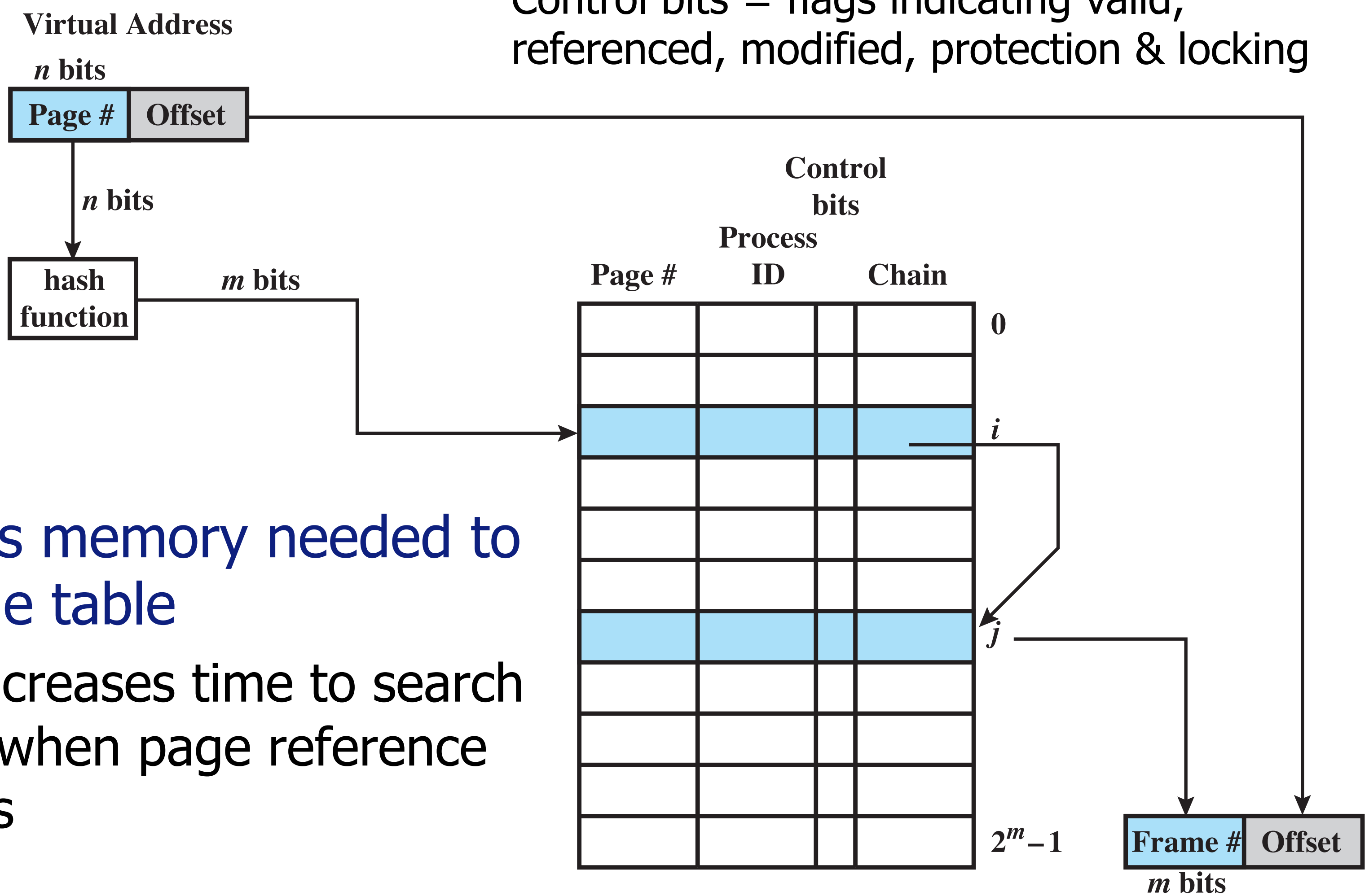
- Page table contains chain of elements hashing to same location
- Search for match of virtual page number in chain
- Extract corresponding physical frame if match found



- Per process hash table

3. Inverted Page Table

Control bits = flags indicating valid, referenced, modified, protection & locking



Inverted Page Table (one entry for each physical memory frame)

Decreases memory needed to store page table

- But increases time to search table when page reference occurs

Segmentation

Paging gives one-dimensional virtual address space

- What about separate address spaces for code, data, stack?

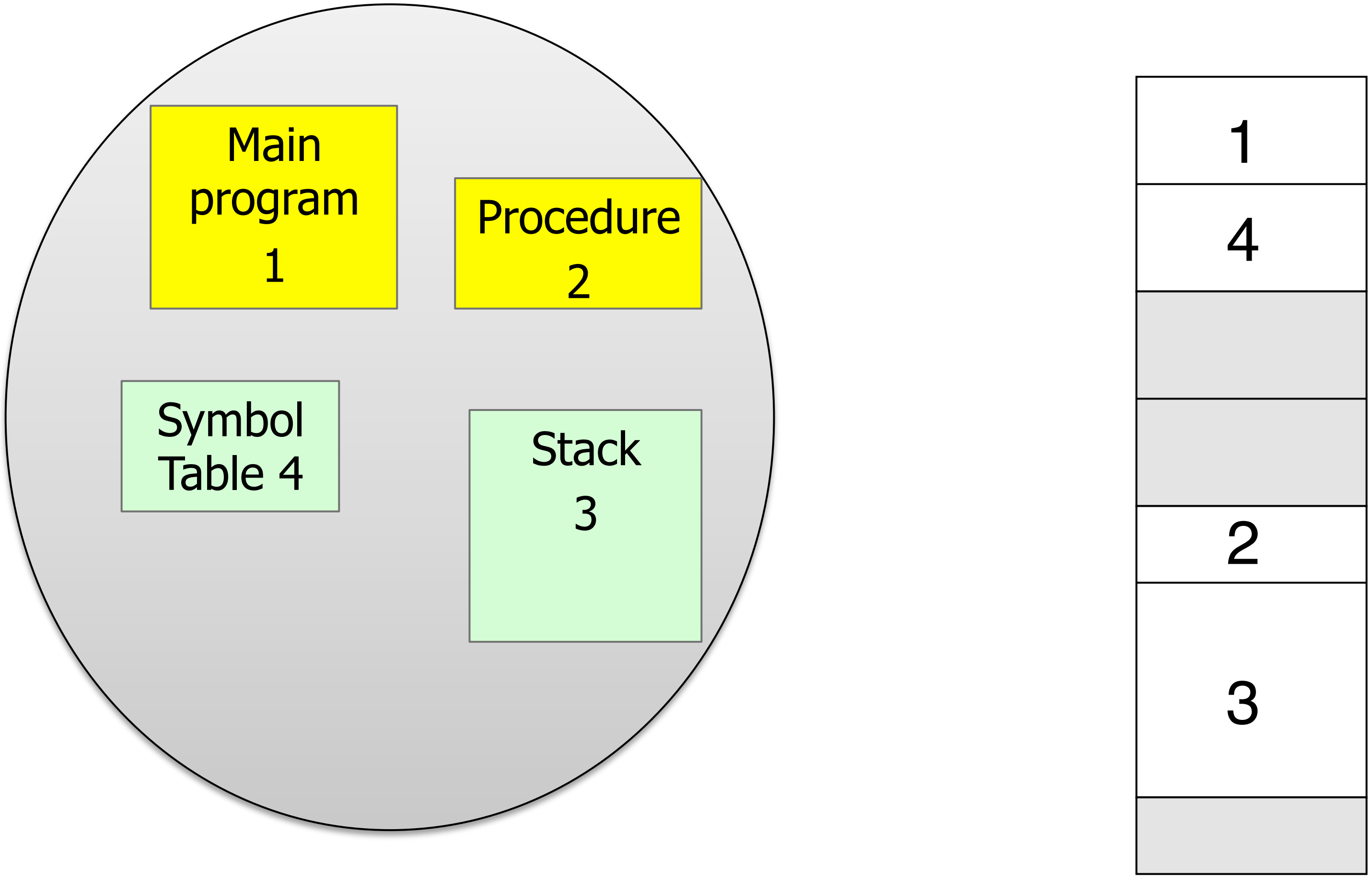
Segment

- Independent address space from 0 to some maximum
- Can grow/shrink independently
- Support different kinds of protection (read/write/execute)
- Unlike pages, programmers are aware of segments
- Segment corresponds to program, procedure, stack, object, array etc.

Memory allocation harder due to variable size

- May need to move segment which grows
- May suffer from external fragmentation
- But good for shared libraries

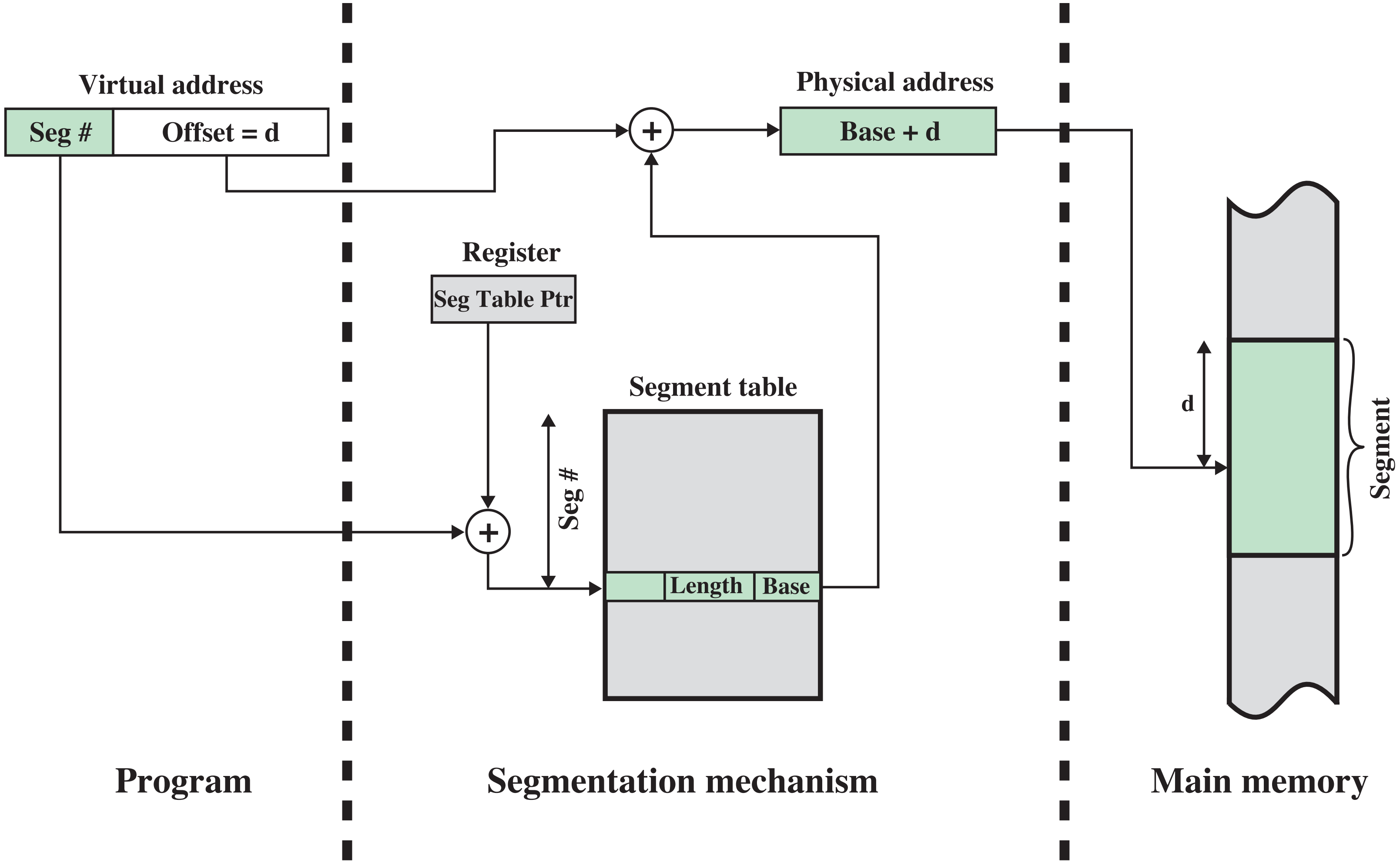
Logical view of Segmentation



User logical space

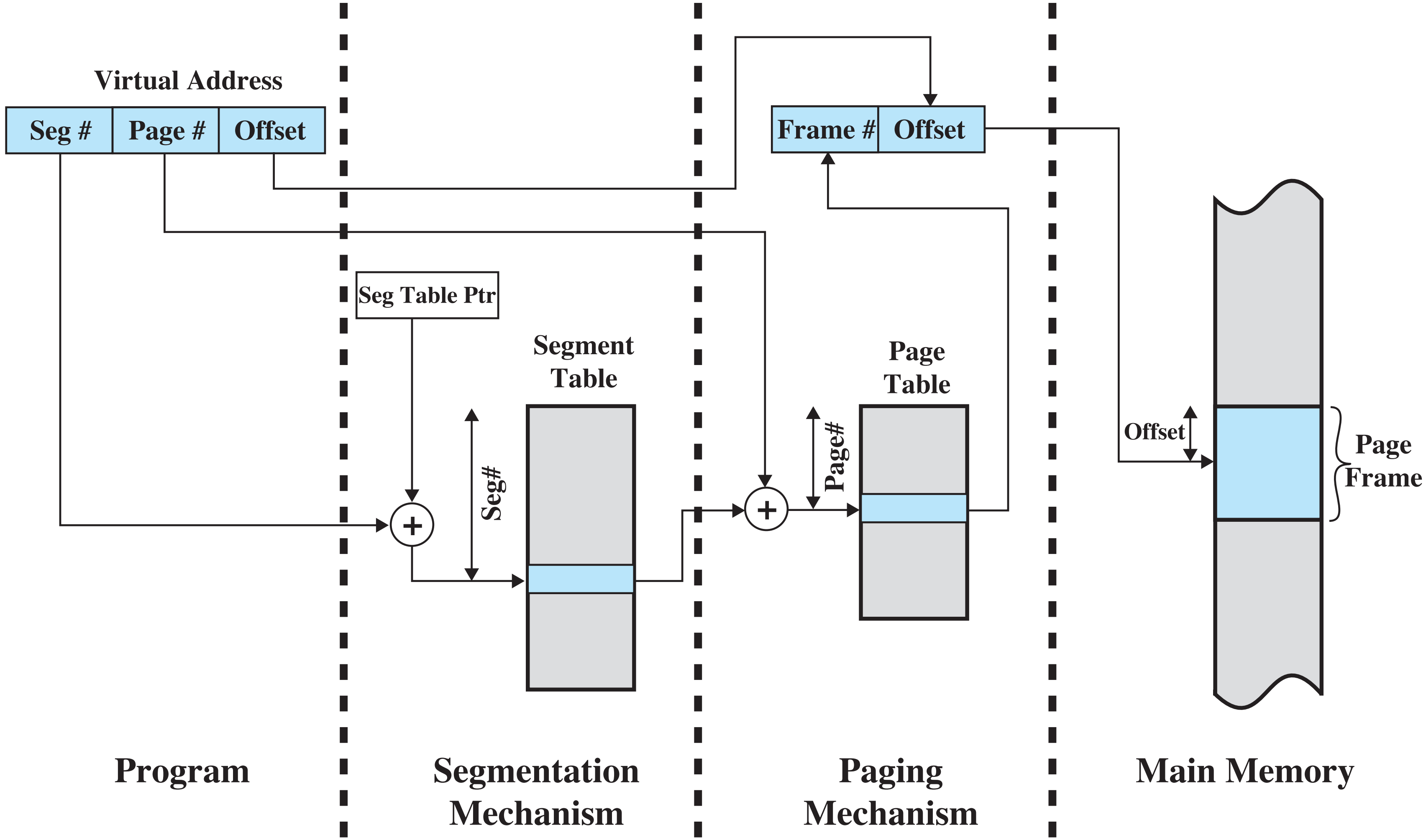
Physical memory space

Segmentation Address Translation



- One bit in table indicates whether segment is in memory
- Another bit indicates whether modified

Hybrid Segmentation/Paging



IA-32 supports both (but most OSs only use paging)

Demand Paging

Demand Paging I

Bring page into memory only *when needed*

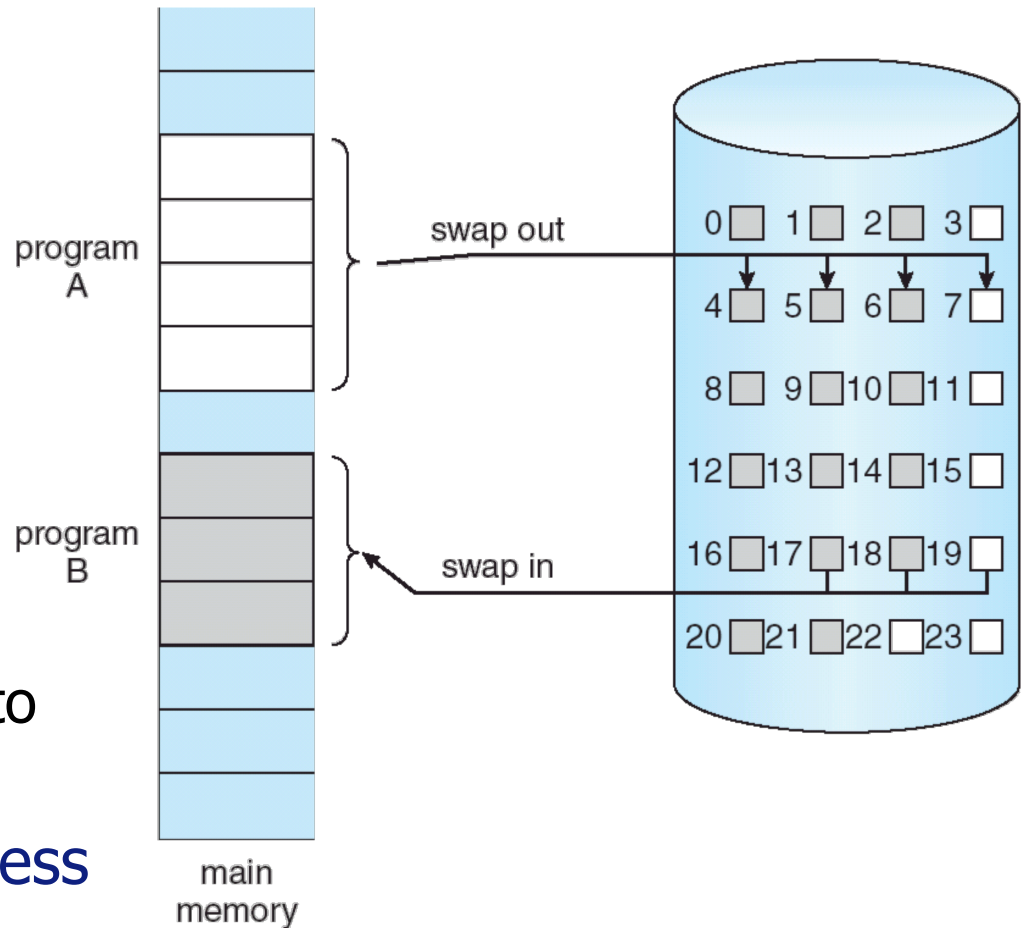
- Lower I/O load
- Less memory needed
- Faster response time
- Support for more users

Page needed ⇒ reference it

- invalid reference ⇒ abort
- not-in-memory ⇒ bring into memory

Many Page faults when process first starts

Eventually required pages are in memory so page fault rate drops



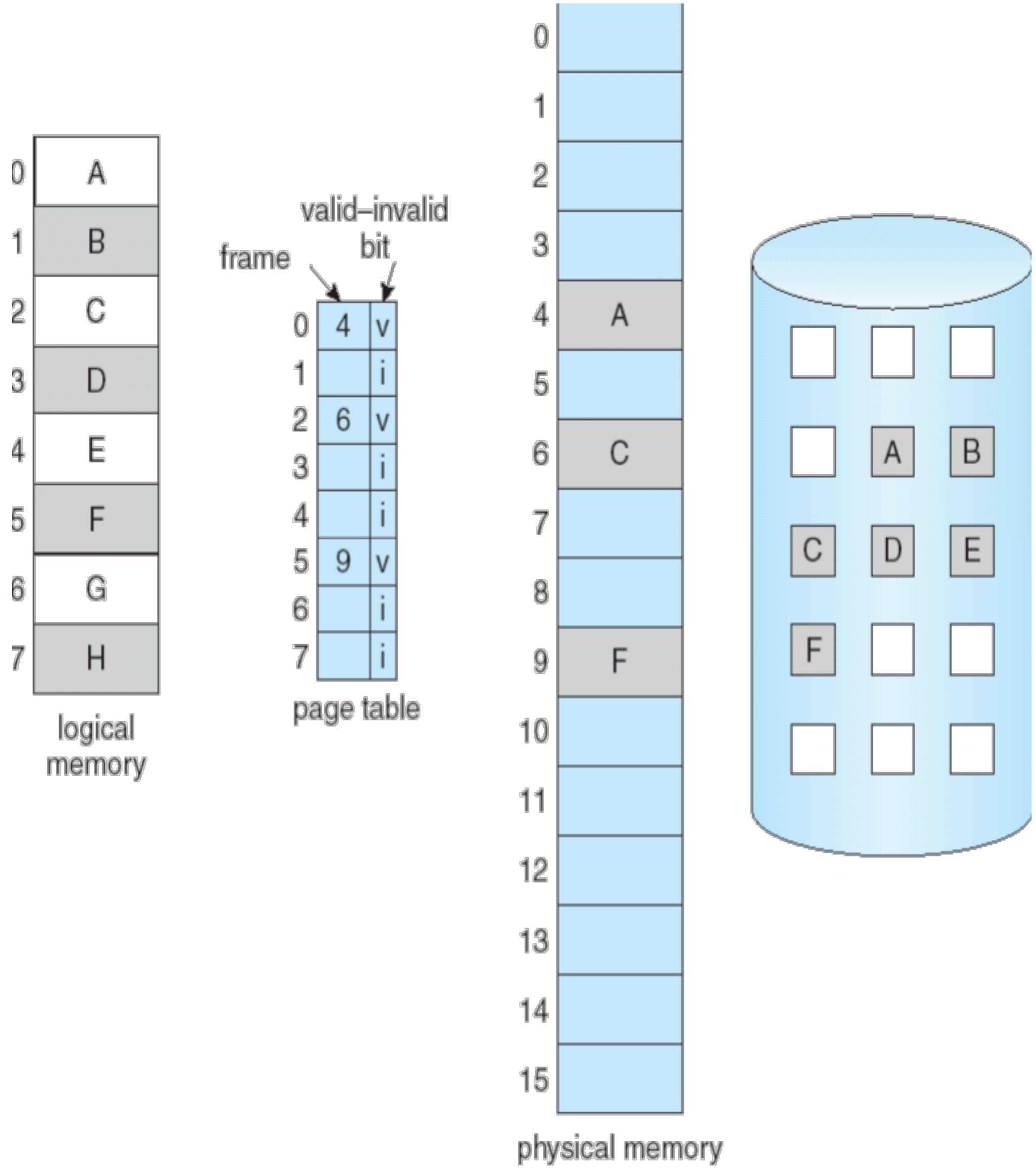
Demand Paging II

Use valid–invalid bit

1 ⇒ in-memory

0 ⇒ not-in-memory

- Initially set to 0 on all entries
- If 0 during address translation ⇒ page fault



Page Faults I

First reference, trap to OS ⇒ **page fault**

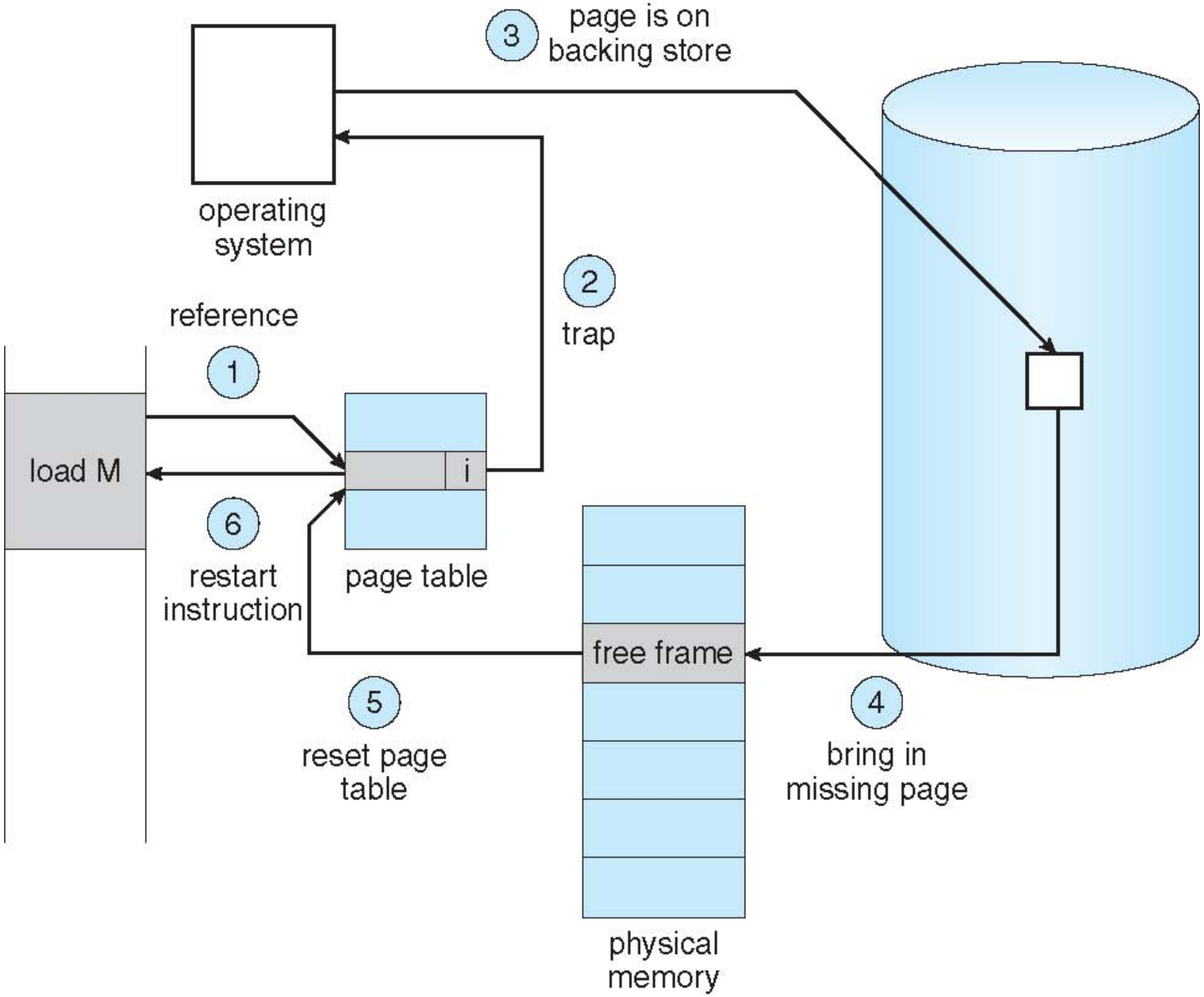
OS looks at another table to decide:

- Invalid reference ⇒ abort
- Valid reference but just not in memory ⇒ handle request

To handle valid request

- Get empty frame
- Swap page into frame
- Reset tables, validation bit = 1
- Restart last instruction

Page Faults II



Performance: Demand Paging

Page Fault Rate $0 \le p \le 1.0$

- if $p = 0$, no page faults
- if $p = 1$, every reference causes page fault

Effective Access Time (EAT)

$$
\begin{aligned}
\text{EAT} = (1 - p) \times \text{memory access} \\
+ p\,(\text{page fault overhead} \\
+ [\text{swap page out}] \\
+ \text{swap page in} \\
+ \text{restart overhead})
\end{aligned}
$$

Note: no need to swap page out if not modified

Example: Demand Paging

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT $= (1 - p) \times 200 + p$ (8 milliseconds)

  $= (1 - p) \times 200 + p \times 8{,}000{,}000$

  $= 200 + p \times 7{,}999{,}800$
- 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

  $220 > 200 + 7{,}999{,}800 \times p$

  $20 > 7{,}999{,}800 \times p$

  $p < 0.0000025$

  < one page fault in every 400,000 memory accesses

Virtual Memory Tricks

Copy-on-Write (COW)

- Allows parent and child processes to initially share same pages in memory
  - If either process <u>modifies</u> shared page, then <u>copy</u> page
- Efficient process creation: copy only modified pages
- Free pages allocated from pool of zeroed-out pages

Memory-mapped Files

- Map file into virtual address space using paging
- Simplifies programming model for I/O

I/O Interlock

- Pages must sometimes be locked into memory
  eg. Pages used for DMA from disk

Page Replacement

No free frame? Replace page

- Find some *unused* page in memory to swap out

Need strategy for **page replacement**

- Minimise number of page faults
  - Avoid bringing same page into memory several times
- Prevent over-allocation of memory
  - Page-fault service routine should include page replacement
- Use modify (dirty) bit to reduce overhead of page transfers
  - Only modified pages written to disk

Basic Page Replacement I

Find location of desired page on disk

Find free frame. Frame found?

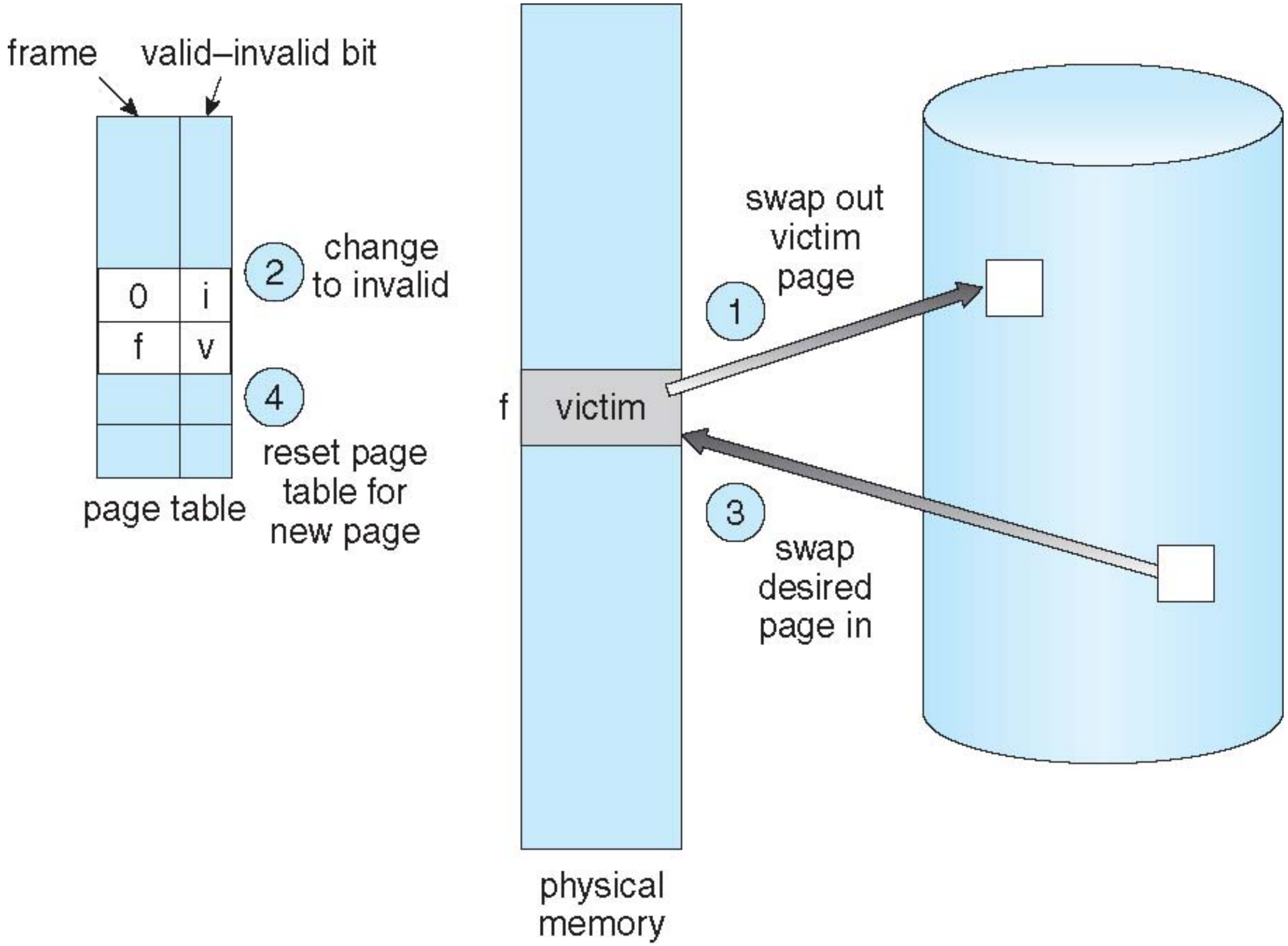
- Yes? ➔ use it
- No? ➔ use replacement algorithm to select **victim** frame

Read desired page into (newly) freed frame

Update page and frame tables

Restart process

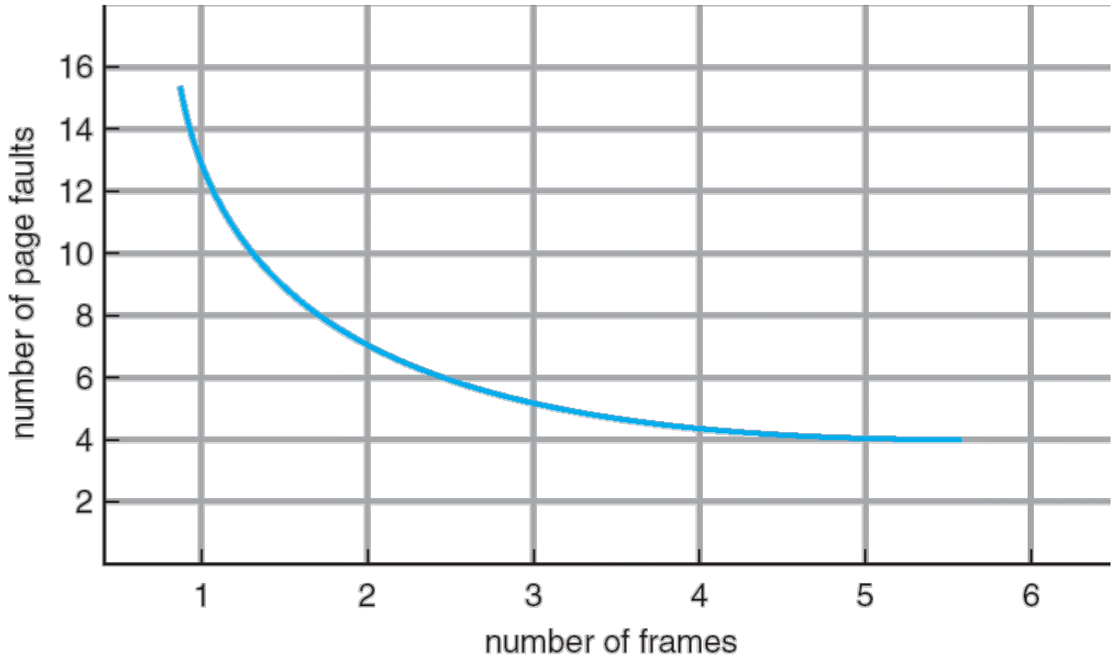
Basic Page Replacement II



Page Replacement Algorithms

Want lowest page-fault rate

- Expect page faults to decrease with more frames



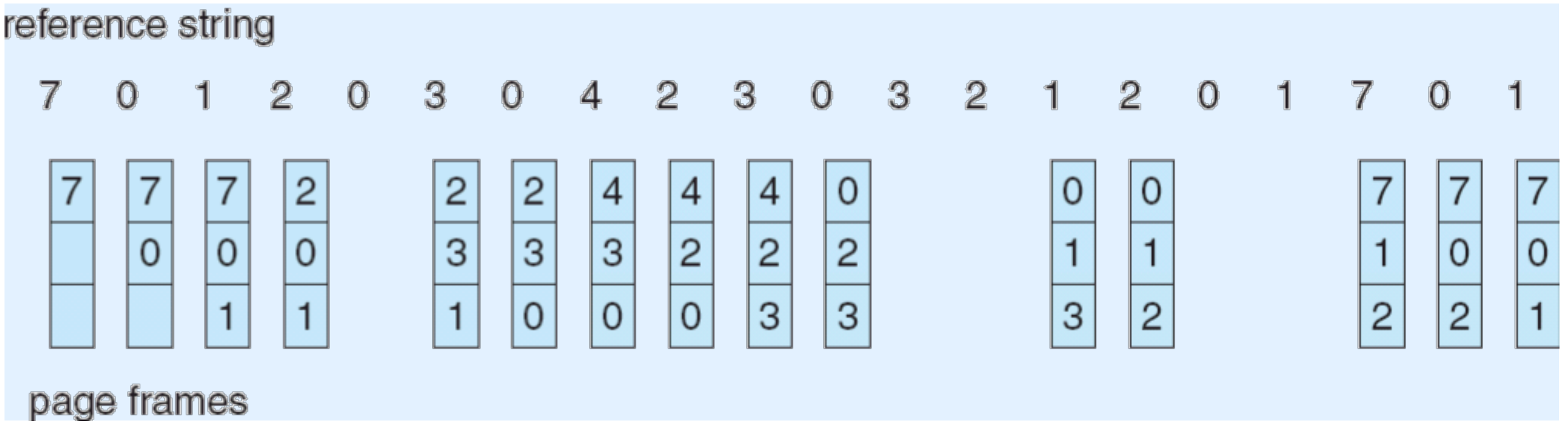
Reference String: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- Particular string of memory references
- Evaluate algorithms by computing number of page faults

First-In-First-Out (FIFO) Algorithm

Replace oldest page

– May replace heavily used page



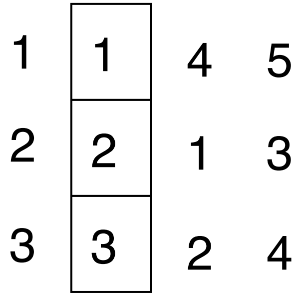
15 page replacements

Heavily used pages: 0, 2, 3 are being swapped in & out,

Belady’s Anomaly I

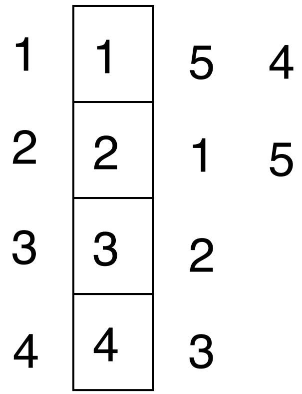
Assume 3 frames with FIFO replacement:

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



9 page faults

Assume 4 frames:



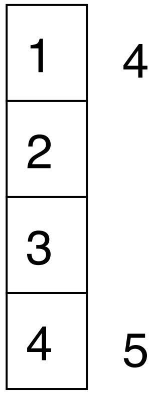
10 page faults

Belady’s Anomaly: More frames ⇒ more page faults

Optimal Algorithm

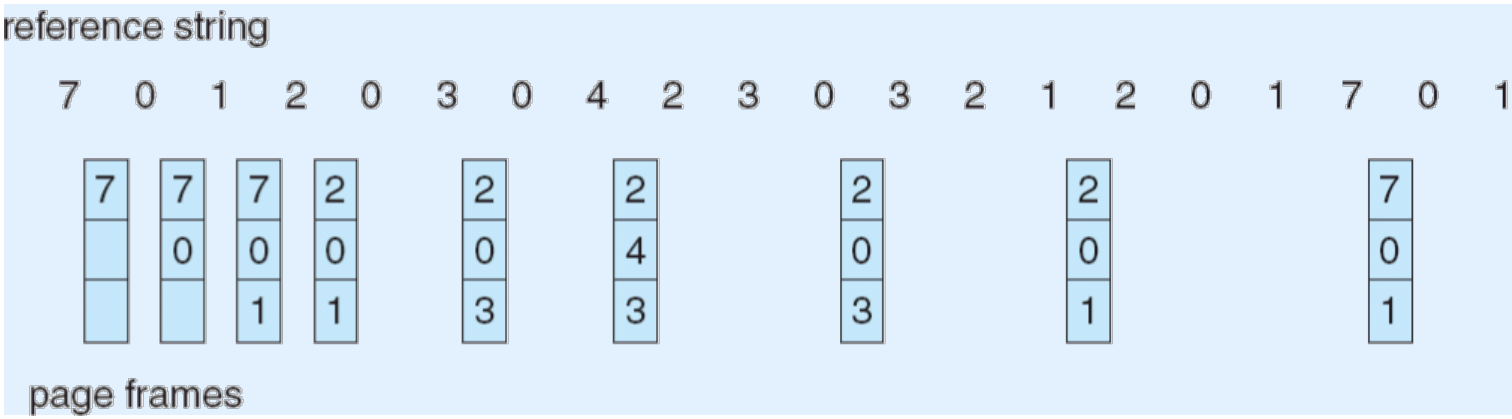
Replace page that will not be used for longest period of time

- Unimplementable, as need knowledge of future references
- Used for measuring how well algorithms perform



Assume 4 frames:
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

6 page faults

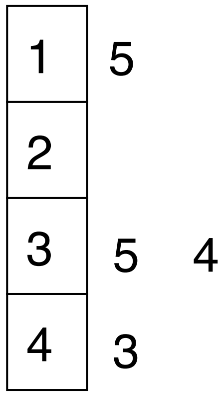


9 page replacements

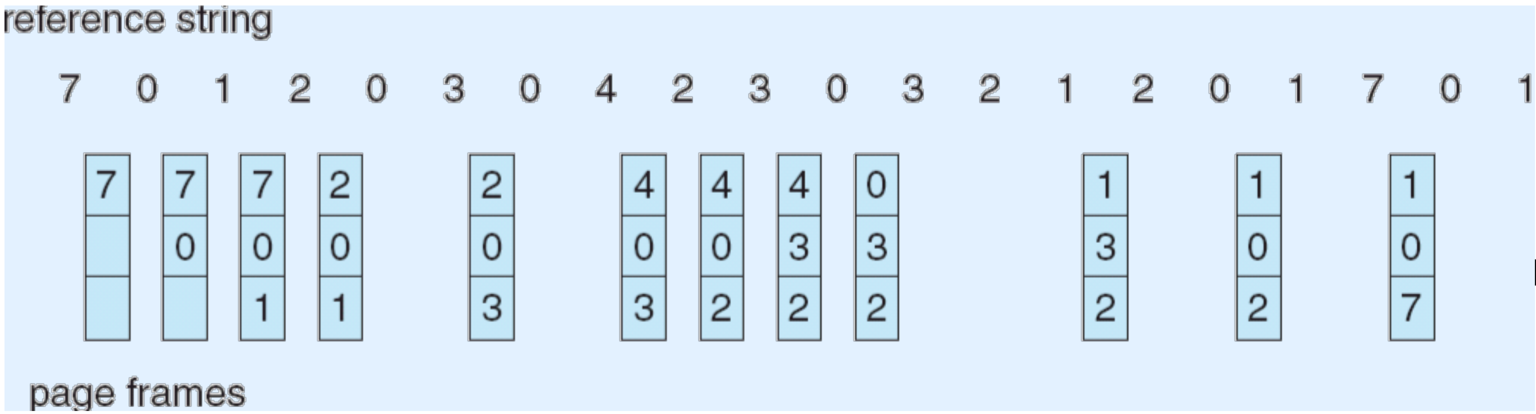
Least Recently Used (LRU) Algorithm

Each page entry has counter

- When page referenced, copy clock into counter
- When page needs to be replaced, choose lowest counter



Reference string:
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



12 page replacements

LRU Approximation Algorithms

Proper LRU expensive → use approximations instead

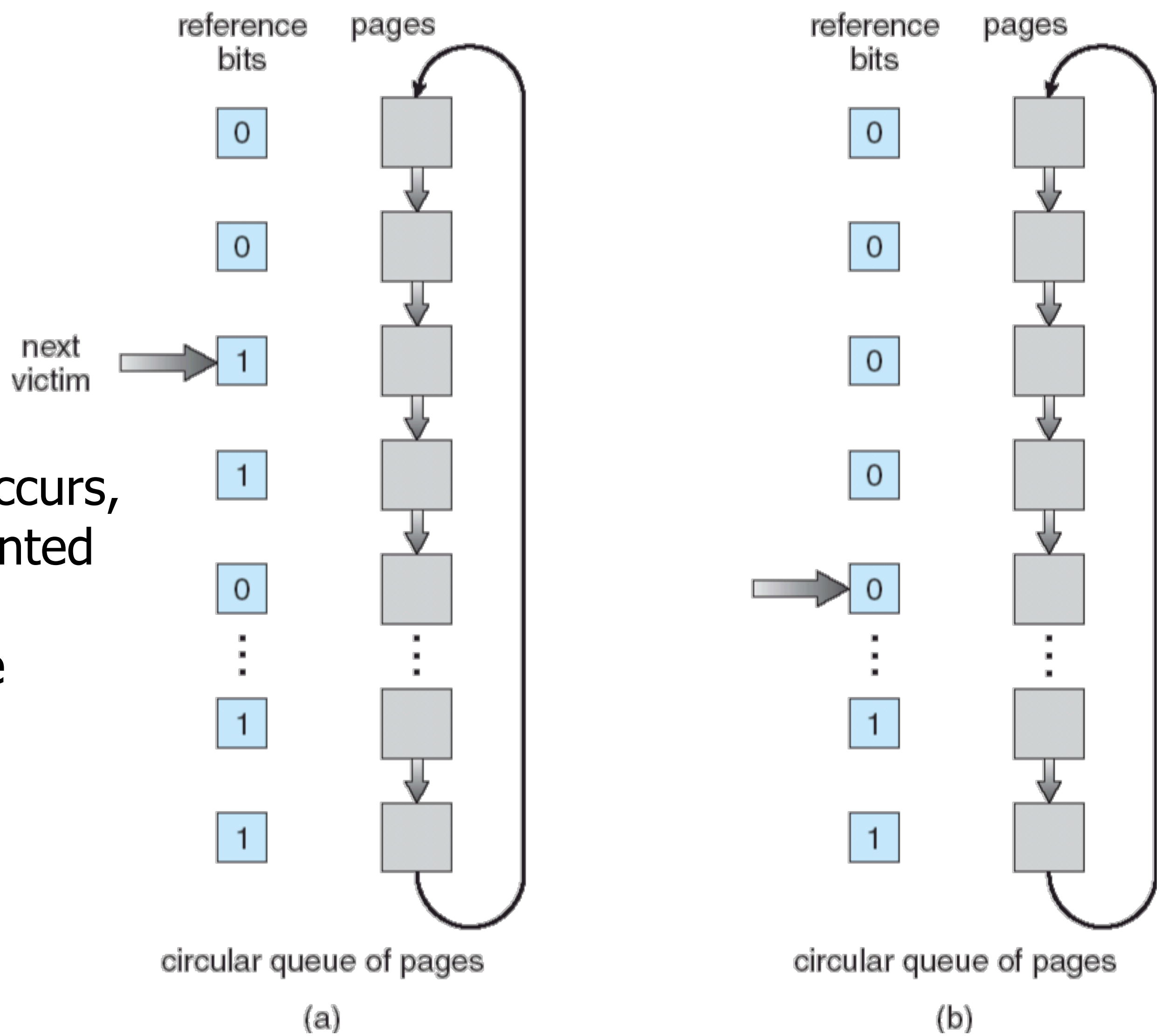
Reference bit

- With each page associate reference bit r, initially r=0
  - When page referenced, set r=1
  - Replace page with r=0 (if one exists)
- Periodically reset reference bits
- Does not provide proper order for LRU

Clock Replacement Policy

- Need reference bit r and uses clock replacement
- If page to be replaced (in clock order) has r=1 then:
  - Set r=0 and leave page in memory
  - Continue till find r=0, and replace that page
  - If all r =1, replace starting page.

Clock Page Replacement



- When page fault occurs, the page being pointed to is inspected.
- If r = 0, evict page
- If r = 1 clear r, & advance pointer.

Counting Algorithms

Keep counter of number of references made to each page

LFU (least frequently used) algorithm

- Replace page with smallest count
- May replace page just brought into memory
- Page with heavy usage in past will have high count
    - Reset counters or use **aging**

MFU (most frequently used) algorithm

- Replace page with largest count
- Page with smallest count probably just brought in and yet to be used

Locality of Reference I

For program to run efficiently:

- System must maintain program's *favoured* subset of pages in main memory

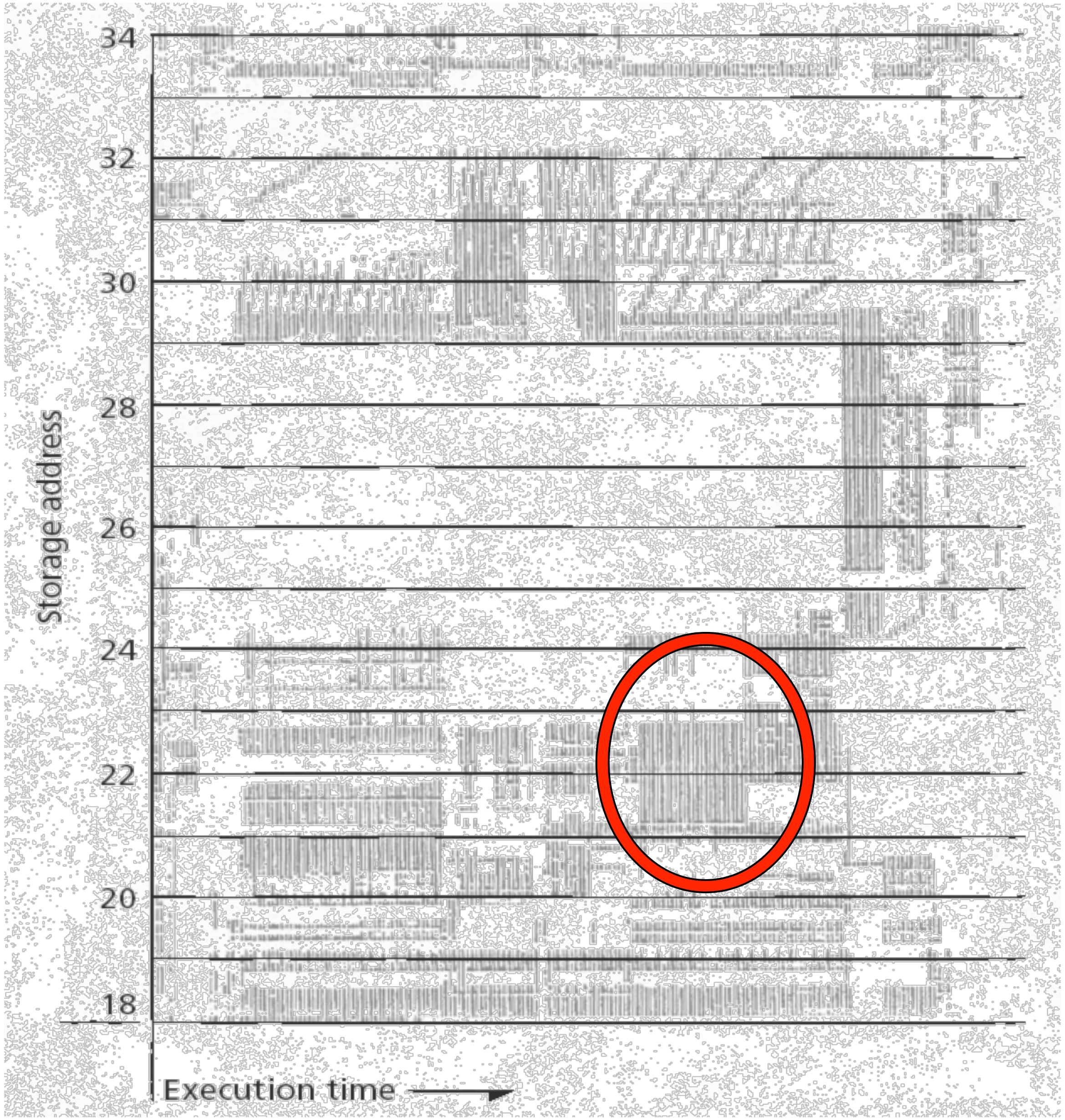
Otherwise **thrashing**

- Excessive paging activity causing low processor utilisation
- Program repeatedly requests pages from secondary storage

Locality of Reference

- Programs tend to request *same* pages in space and time

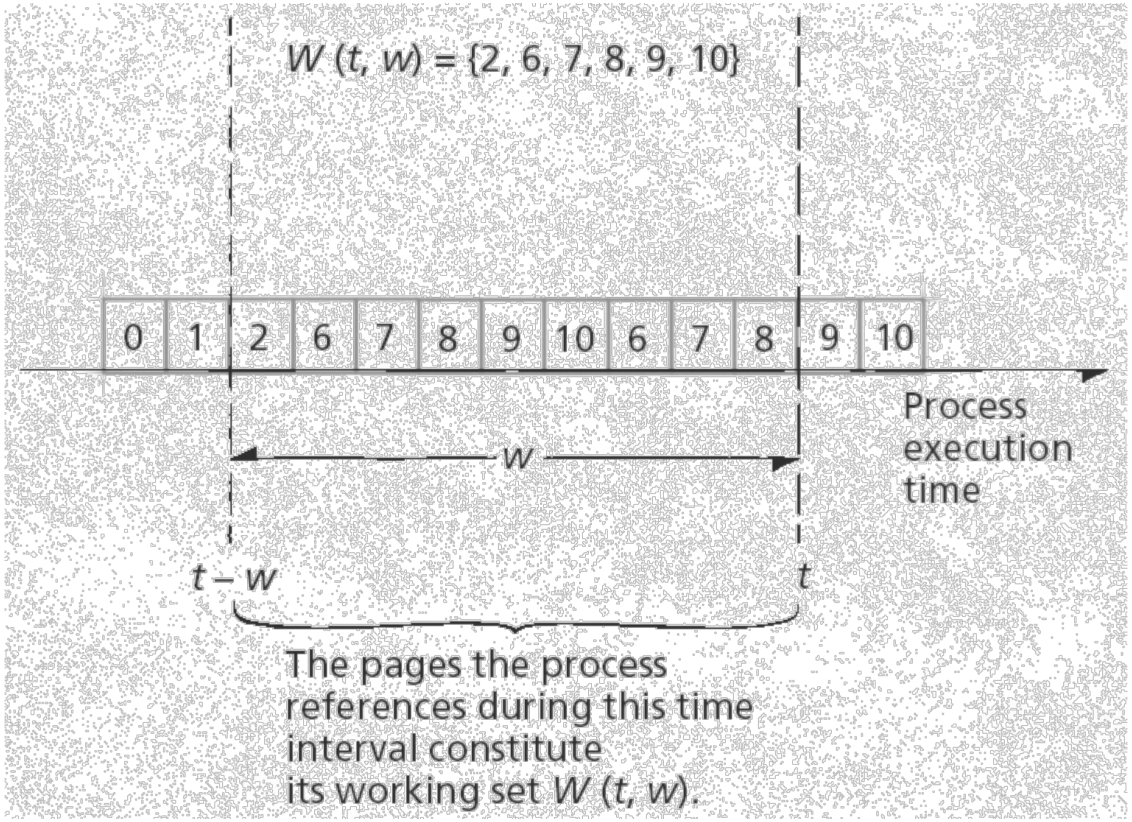
Locality of Reference II



Working Set Model

Working set of pages: W(t,w)

- Set of pages referenced by process during process-time interval *t* - *w* to *t*



WS Clock Algorithm

Idea: Add "time of last use" to Clock Replacement algorithm

- Keeps track if page in working set

At each page fault, examine page pointed to:

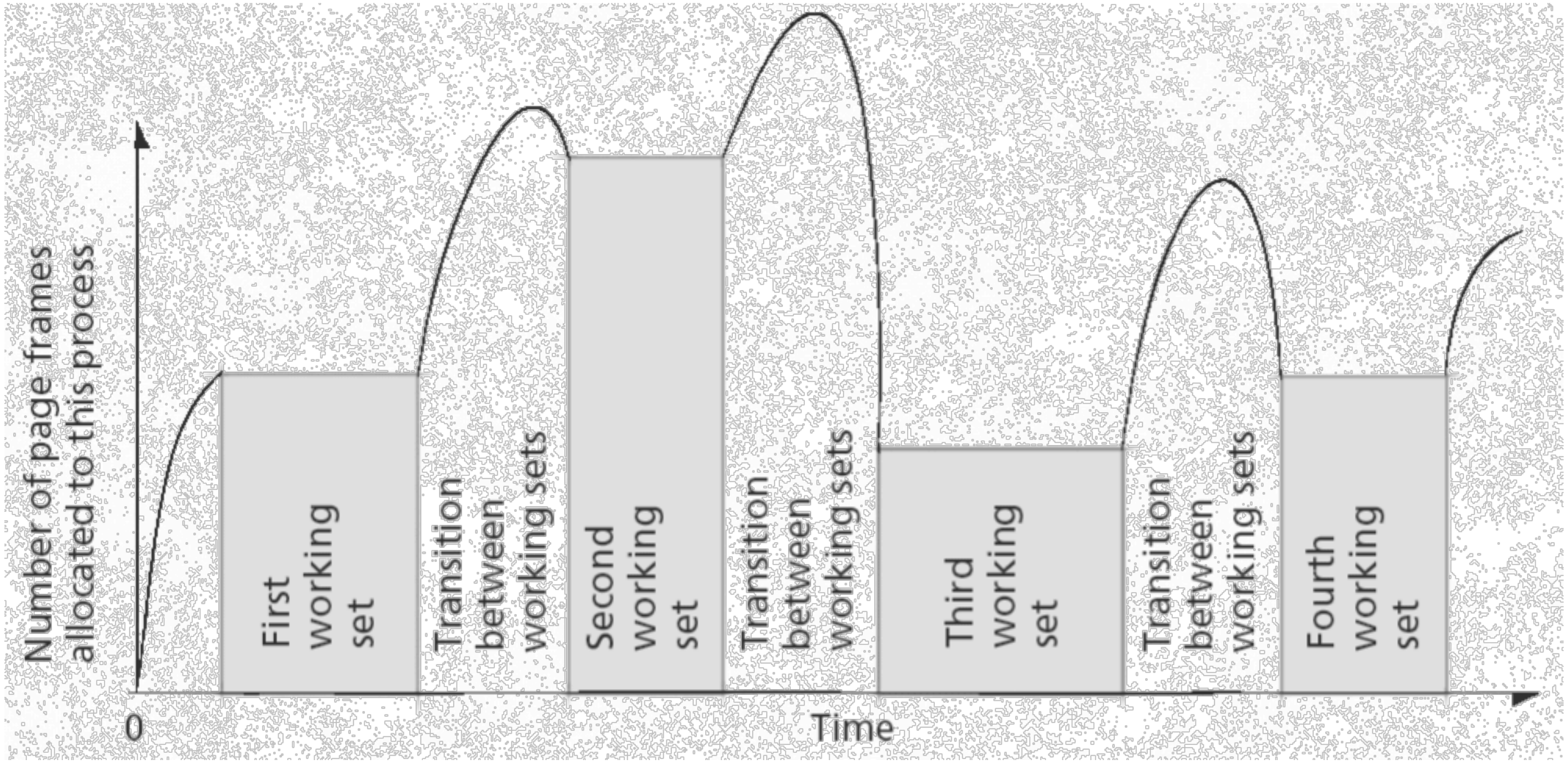
- if r=1, then set r=0 and move to next page
- if r=0, calculate age
  - if age < working set age w, continue (page in WS)
  - if age > working set age w
    - if page clean, replace
    - otherwise trigger write-back, continue

Working Set Size I

How to choose size of working set?

Processes *transition* between working sets

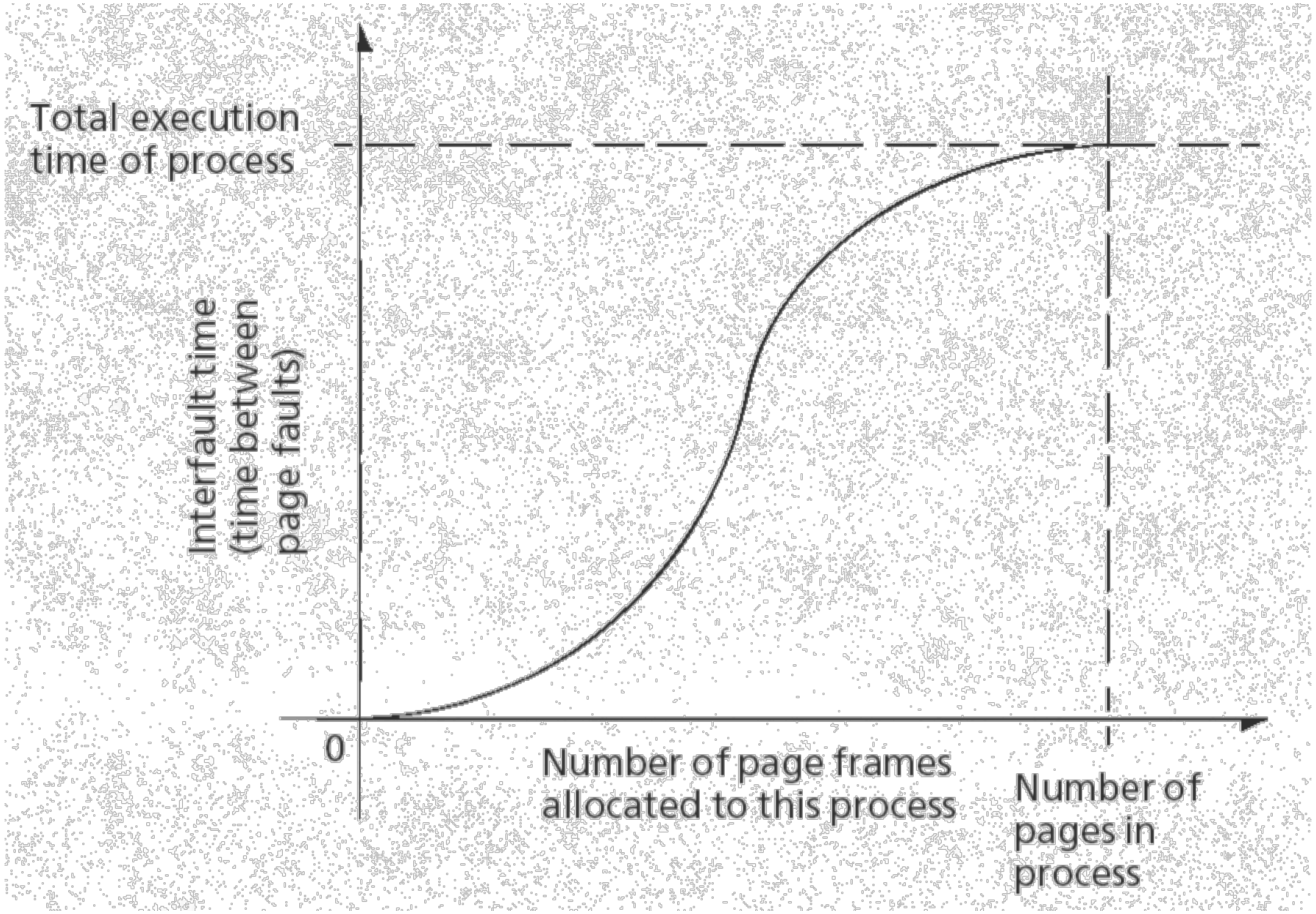
- OS temporarily maintains in memory pages outside of current working set
- Goal of memory management to reduce misallocation



Working Set Size II

Idea: observe page fault frequency

- If many faults ➔ allocate more page frames



Global vs Local Page Replacement

Local strategy

- Each process gets fixed allocation of physical memory
- Need to pick up changes in working set size

Global strategy

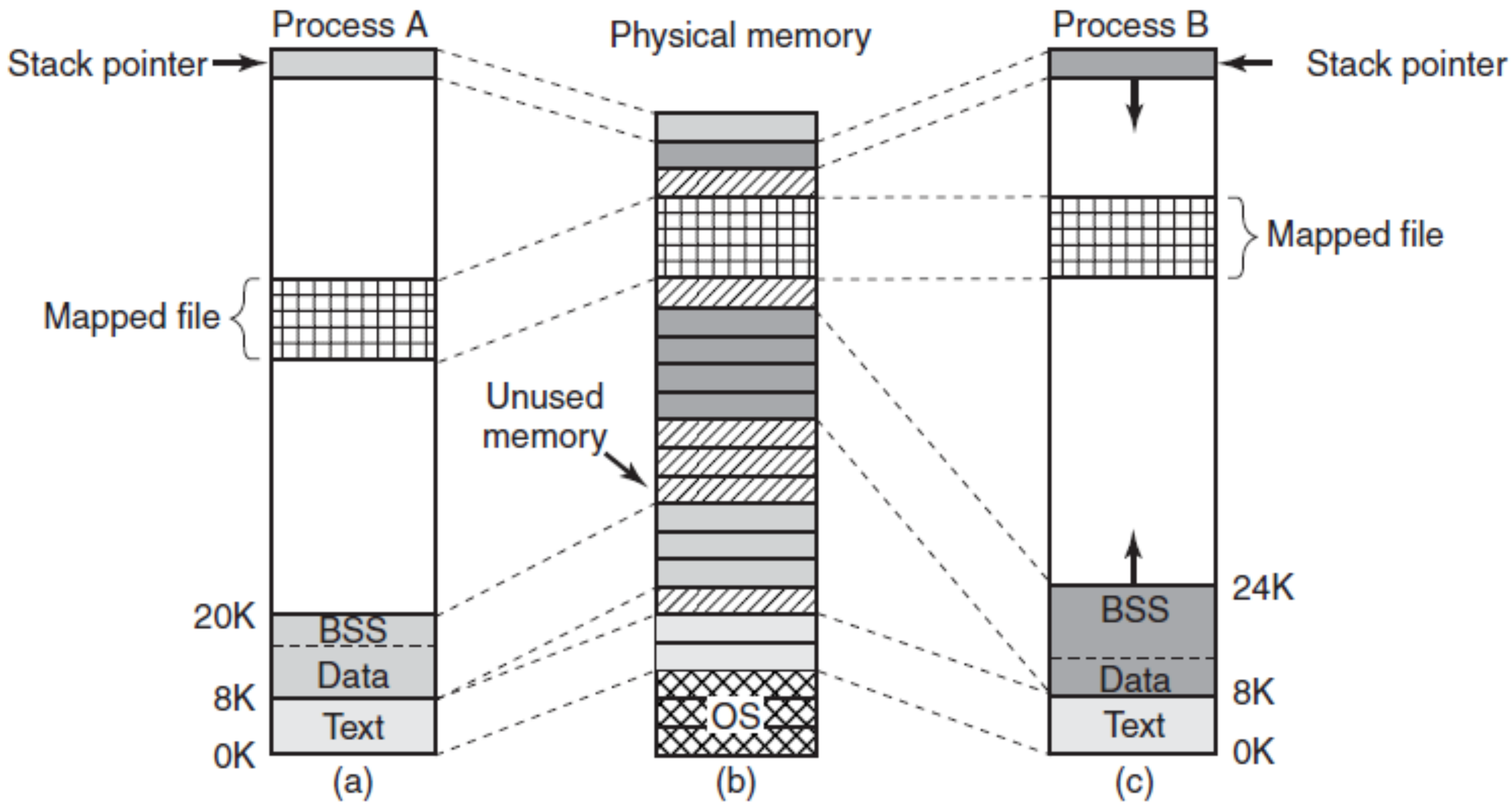
- Dynamically share memory between runnable processes
- Initially allocate memory proportional to process size
- Consider **page fault frequency** (PFF) to tune allocation
  - Measure page faults/per sec and increase/decrease allocation

No universally agreed solution

- Linux: global page replacement
- Windows: local page replacement
- Depends on scheduling strategy (i.e. round-robin, …)

Linux Memory Management

Mapping and Sharing Memory



Memory Management System Calls

System call	Description
s = brk(addr)	Change data segment size
a = mmap(addr, len, prot, flags, fd, offset)	Map a file in
s = unmap(addr, len)	Unmap a file

Return code *s* is −1 if error

a and *addr* are memory addresses,

len is a length,

prot controls protection,

flags are miscellaneous bits,

fd is a file descriptor

offset is a file offset

Virtual Memory Layout I

On 32 bit machine process has 4 GB of space

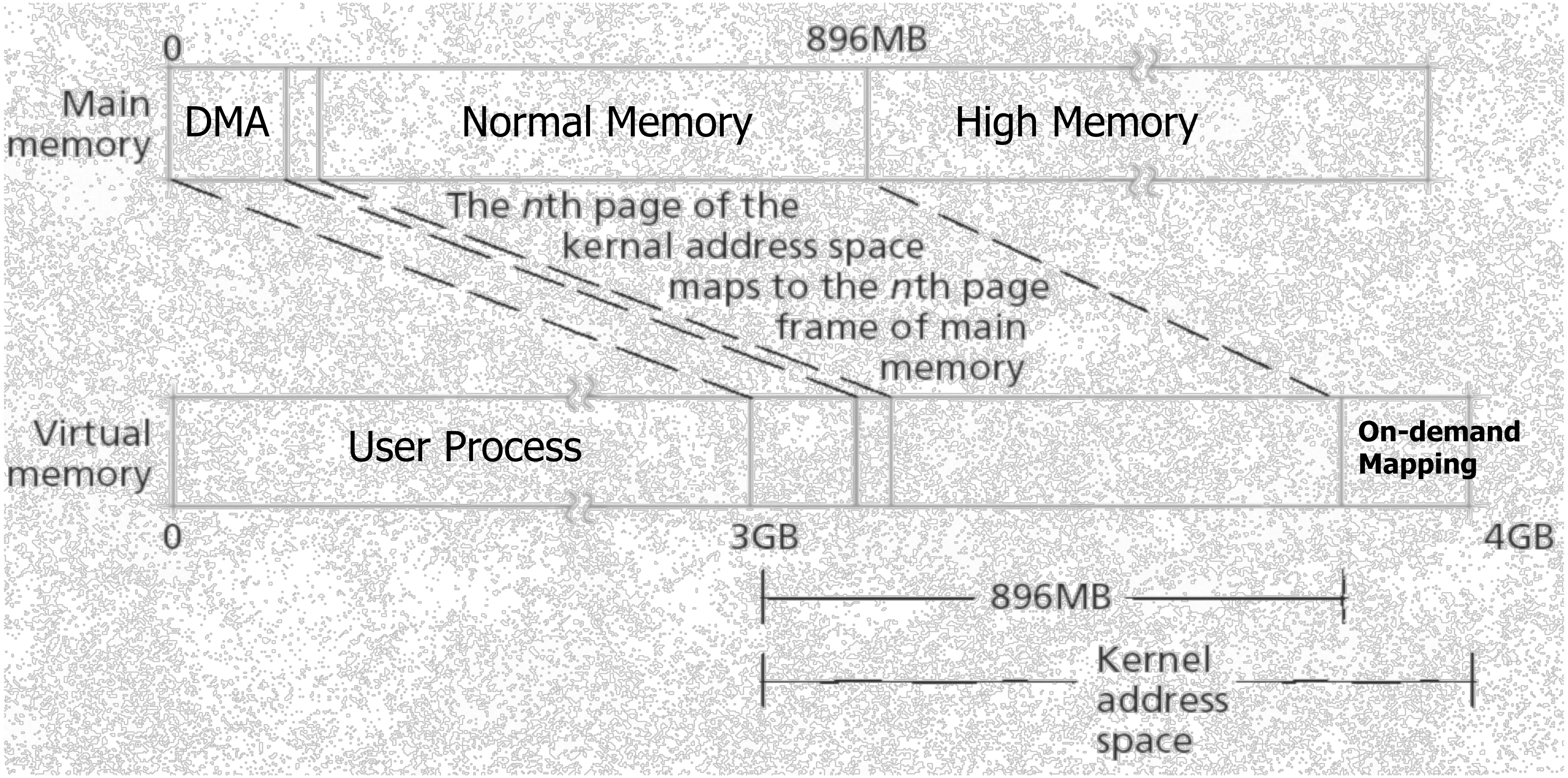
Top 1GB (3-4 GB) used for Kernel memory

- User processes can make system calls without TLB flush
- Kernel space not visible in user mode
- Kernel typically resides in 0-1GB of physical memory

Kernel maps lower 896 MB of physical memory to its virtual address space

- All memory access must be virtual but need efficient access to user memory + DMA in low memory
- Create temporary mappings for >896MB of physical memory in remaining 128MB of virtual memory

Linux: Virtual Memory Layout II



Physical Memory Management

Linux Memory zones

1. ZONE_DMA and ZONE_DMA32: pages used for DMA
2. ZONE_NORMAL: normal regularly mapped pages
3. ZONE_HIGHMEM (> 896MB): pages with high memory addresses – not permanently mapped

Kernel and memory map are pinned, i.e. never paged out

Linux: Paging

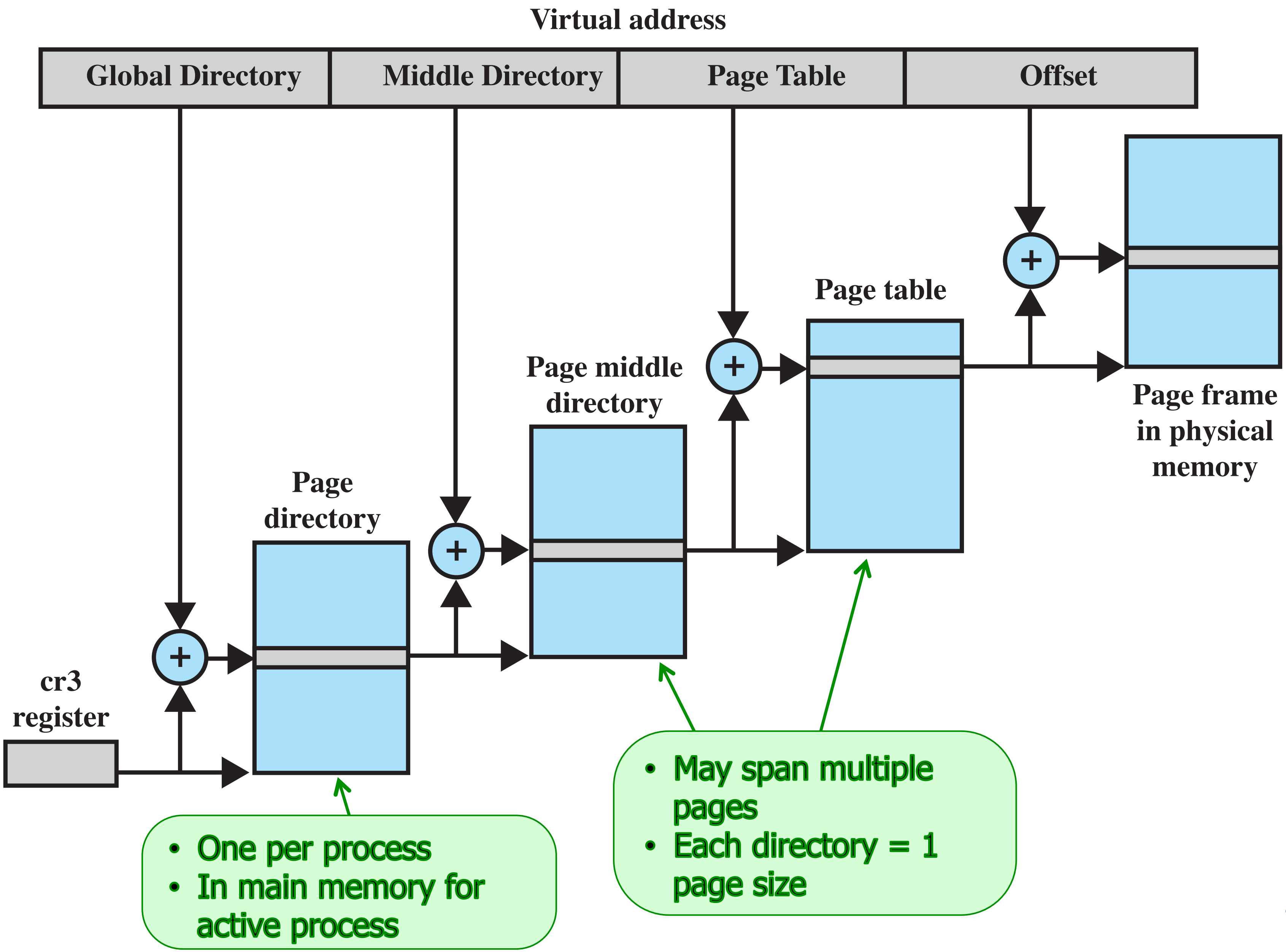
Usually on IA-32:

- 4 KB page size
- 4 GB virtual address space
- Two-level page table
  - (or three levels with Physical Address Extension (PAE))
  - Offset bits contain page status: dirty, read-only, ...

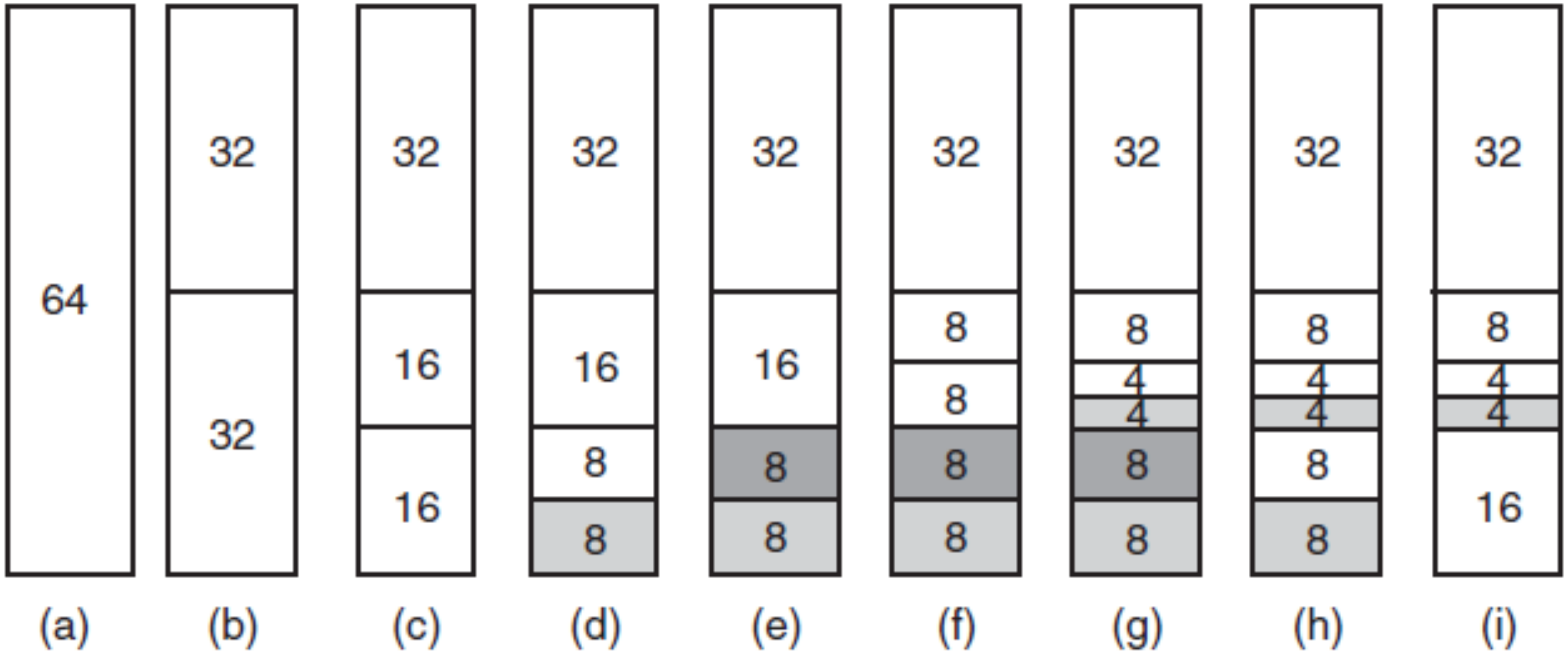
On x86-64:

- Larger page sizes (e.g. 4MB)
- Up to four-level page table

Linux 3 Level Paging



Linux Buddy Memory Allocation



- Tries to map contiguous pages to contiguous frames to optimise transfers
- Split and merge frames as required.

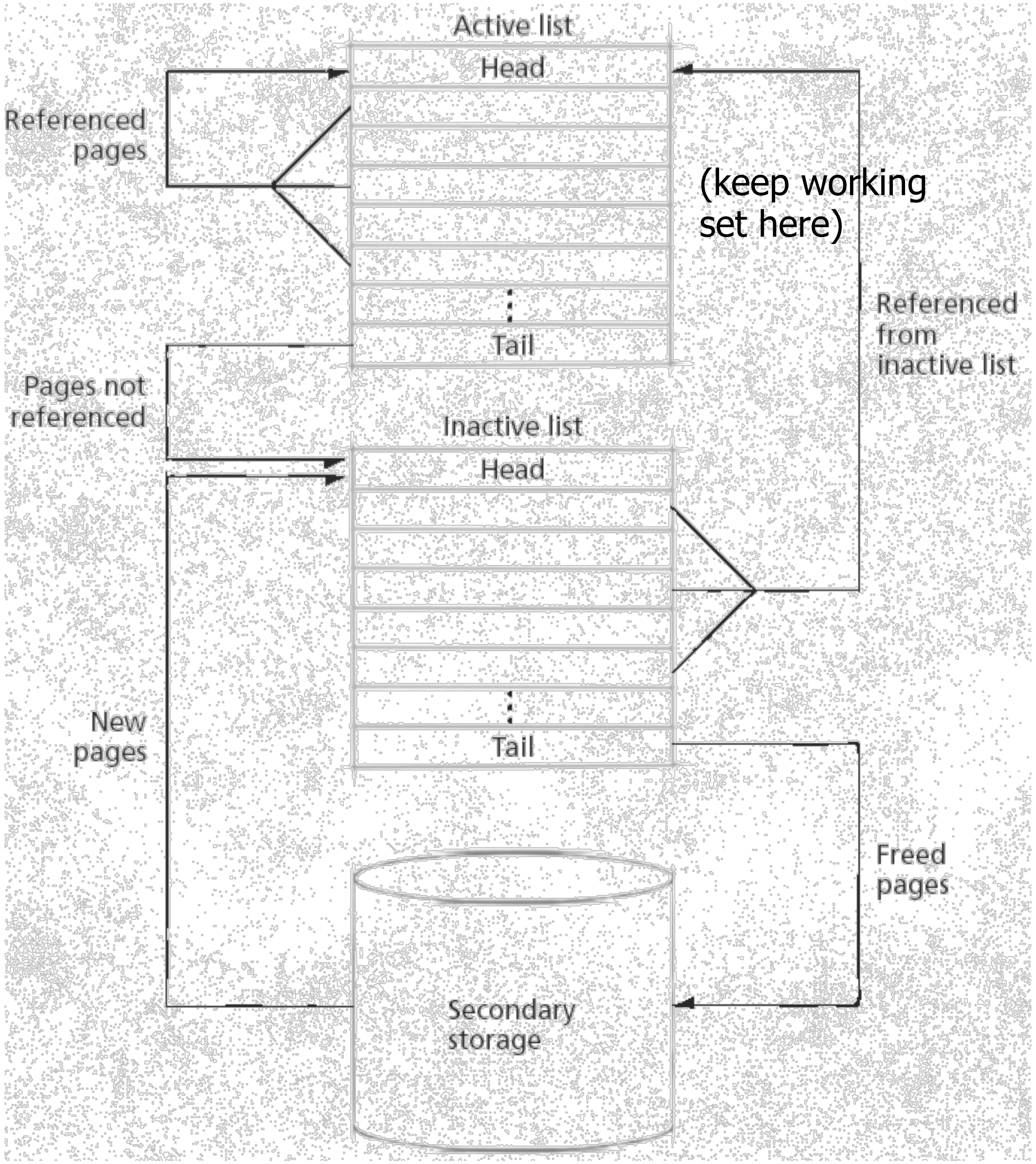
Linux: Page Replacement I

Linux uses variation of clock algorithm to approximate LRU page-replacement strategy

Memory manager uses two linked lists (and reference bits):

- Active list
  - Contains active pages
  - Most-recently used pages near head of active list
- Inactive list
  - Contains inactive pages
  - Least-recently used pages near tail of inactive list
- Only replace pages in inactive list

Linux: Page Replacement II



kswapd (swap daemon)

- Pages in inactive list reclaimed when memory low
- Uses dedicated swap partition or file
- Must handle locked and shared pages

pdflush kernel thread

- Periodically flushes dirty pages to disk

Summary

- Swap inactive whole or parts of process to backing store
- Paging: fixed size frames mapped to main memory & disk
  ➔ virtual memory > physical
- TLB: use associative memory to reduce overhead of page table access
- Segmentation: variable size chunks for code, data etc. Not much used in modern systems
- Fragmentation – wasted memory
- Page faults swap in missing page on demand
- Page replacement – policy for selecting page to swap out
  ➔ least recently used approximations
- Locality of reference ➔ working set of pages, reduce thrashing