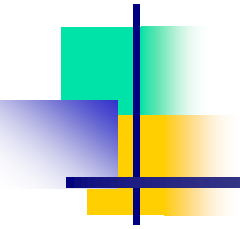
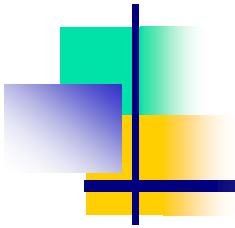


Operating Systems

Lecture 4 - Memory Management

By: Hossein Momeni
momeni@iust.ac.ir



Contents

- Basic memory management
- Swapping
- Virtual memory
- Page replacement algorithms
- Design issues for paging systems
- Segmentation

In an ideal world…

- The ideal world has memory that is
  - Very large
  - Very fast
  - Non-volatile (doesn't go away when power is turned off)

- Memory management goal: make the real world look as much like the ideal world as possible

Memory

- Memory is an important resource that should be managed carefully.
- Memory Hierarchy:
  - **Cache**: Small, expensive, fast, volatile, …
  - **RAM:** Medium-speed, medium price, volatile,
  - …
  - **Disk Storage:** Large, slow, non-volatile, …
- What's the role of the Operating System?

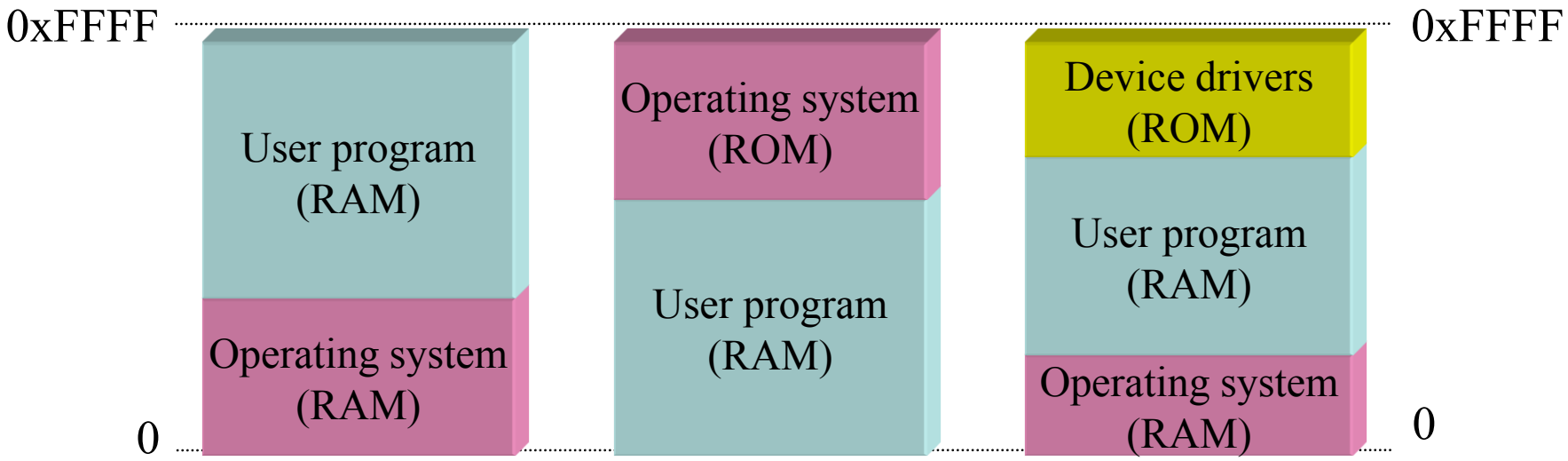
Basic Memory Management

- Memory management systems can be divided into two classes:
  - Those that move processes back and forth between main memory and disk. (Swapping and Paging)
  - Those that do not. !!!

- The second class is simpler and would be discussed first.

Monoprogramming without Swapping or Paging

- An operating system with one user process
- No swapping or paging
- Three simple ways of organizing memory



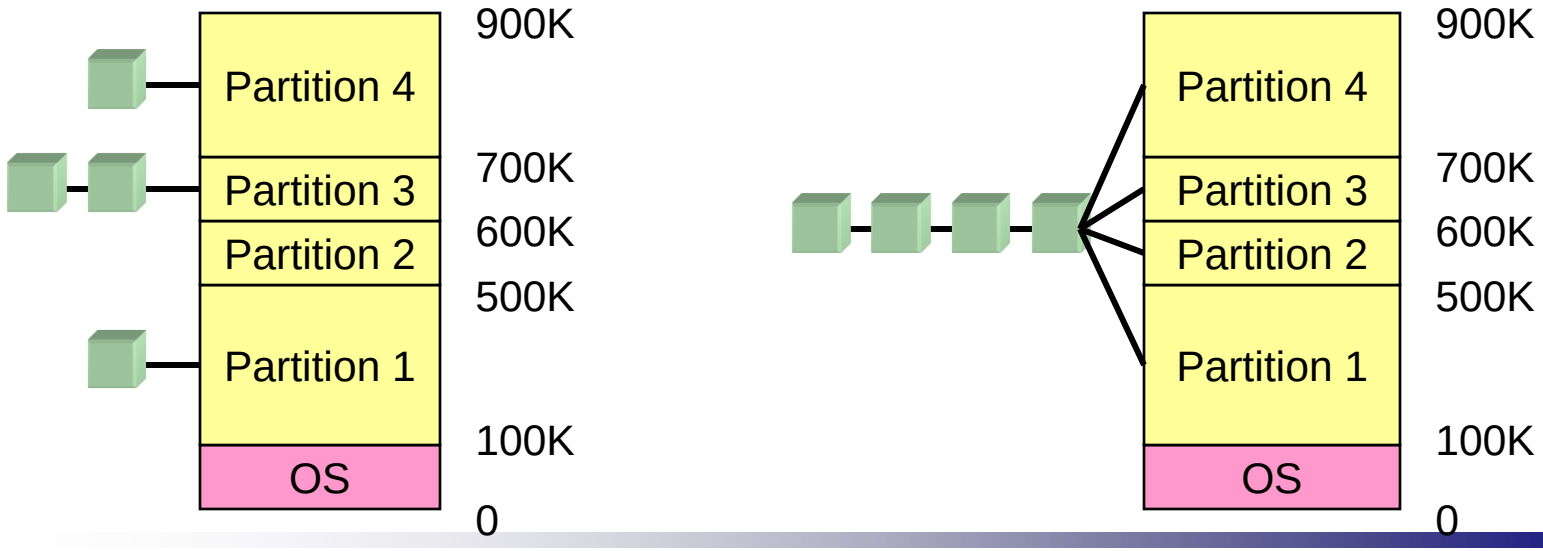
- Mainframe and Mini Computer- Embedded systems,- Personal Computer (Ms-DOS)

Mono-Programming

- When the system is organized in this way, only one process at a time can be running.
  - User types a command
  - The OS copies the requested program from disk to memory and executes it.
  - When the process finishes, the OS displays a prompt and OS waits for a new command.
  - The next commands would be overwritten on the first one.

Multiprogramming with Fixed Partitions

- Fixed memory partitions
  - Divide memory into fixed spaces
  - Assign a process to a space when it's free
- Mechanisms
  - Separate input queues for each partition
  - Single input queue: better ability to optimize CPU usage



Multiprogramming – Fixed Partitions

- When a job arrives, it can be put into the input queue for the smallest partition large enough to hold it.
- Since the partitions are fixed in this scheme, any space in a partition not used by a job is wasted while that job runs
- The disadvantage of sorting the incoming jobs into separate queues becomes apparent when the queue of a large partition is empty but other queues are full.
- So, it would be possible to put all the jobs in one queue:
  - Pick the jobs one by one.
  - Search in the queue for the best job when a partition becomes empty. (What's the problem?)

Relocation & Protection

- Multiprogramming introduces two essential problems: relocation and protection

- Different jobs would be run at different addresses.
- Is it possible for the linker to put the right address in the final binary code?
- This problem is known as the relocation.
  - One solution is to modify the instructions as the program is loaded into main memory.

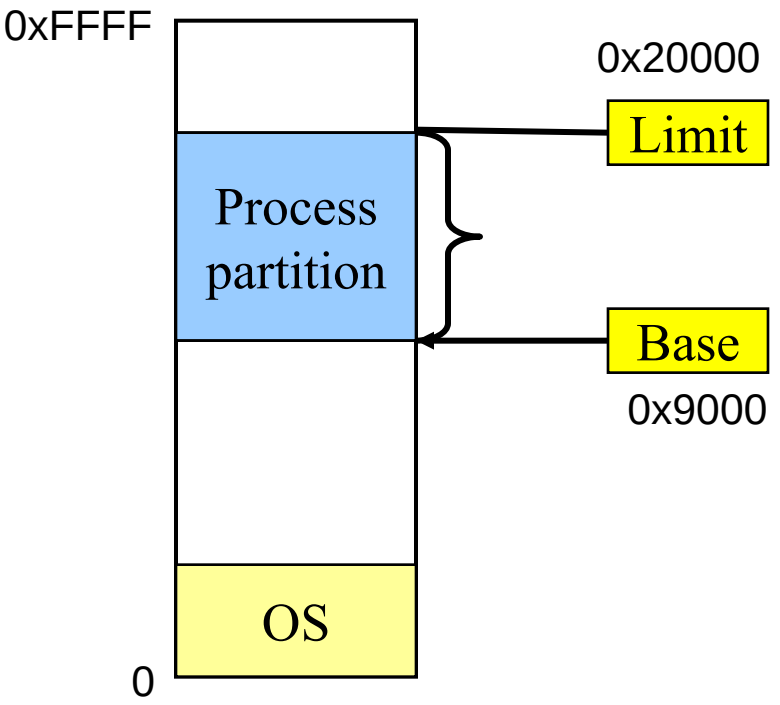
Relocation & Protection

- How to protect against invalid access of user programs to main memory?
  - In IBM 360, the memory was divided into blocks of 2K and assigned a 4-bit protection code.
  - On each memory access, the operating system checks the code of the process and the one written in hardware.

Relocation & Protection

- Another solution is to equip the machine with two registers, base and limit.

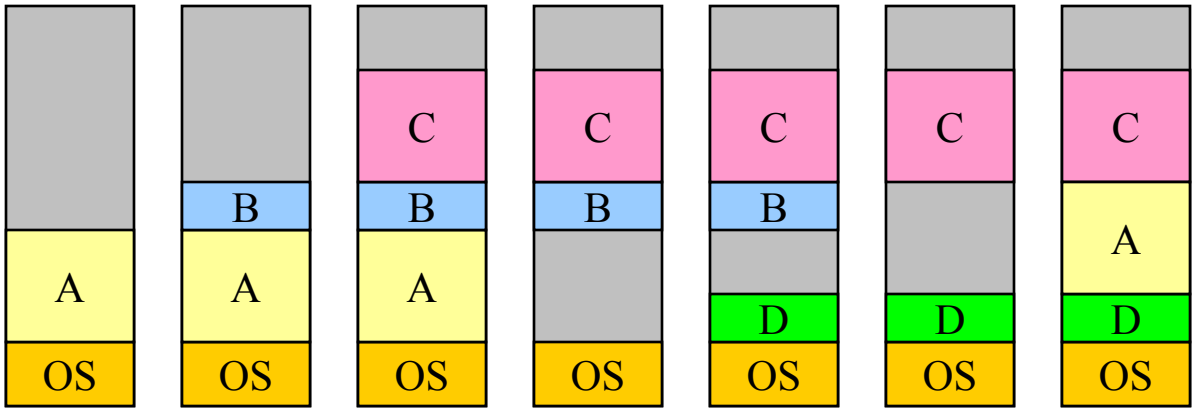
- Address generation
  - Physical address: location in actual memory
  - Logical address: location from the process's point of view
  - Physical address = base + logical address
  - Logical address larger than limit => error



Multiprogramming with Swapping

- Batch system: fixed partitioning is good
- Time sharing system: there is not enough main memory to hold all the currently active processes.
- There are two solution approach:
  - Swapping
  - Virtual Memory

Swapping



- Memory allocation changes as
  - Processes come into memory
  - Processes leave memory
    - Swapped to disk
    - Complete execution
- Gray regions are unused memory

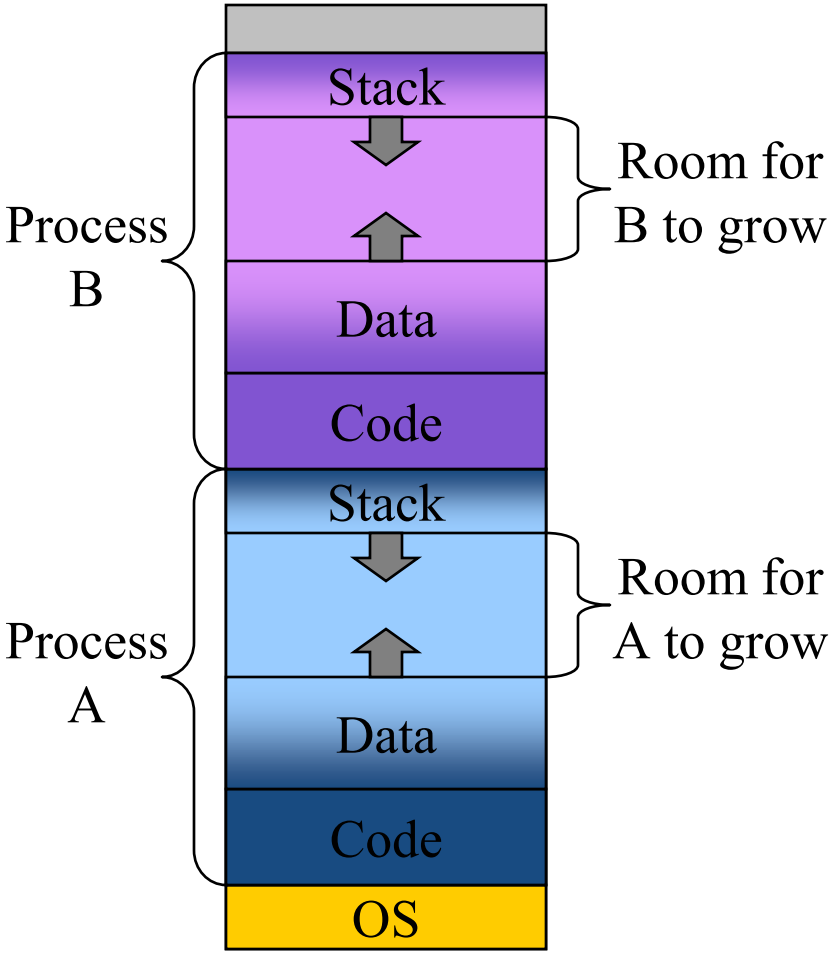
• Swap in from disk
• Swap out to disk

Memory Compaction

- Fixed Partition versus Variable partition
- Swapping creates multiple holes in main memory.
- It's possible to combine all of them into a big hole.
- This technique is called memory compaction.
- It is usually not done because it requires a lot of CPU time.

Swapping: leaving room to grow

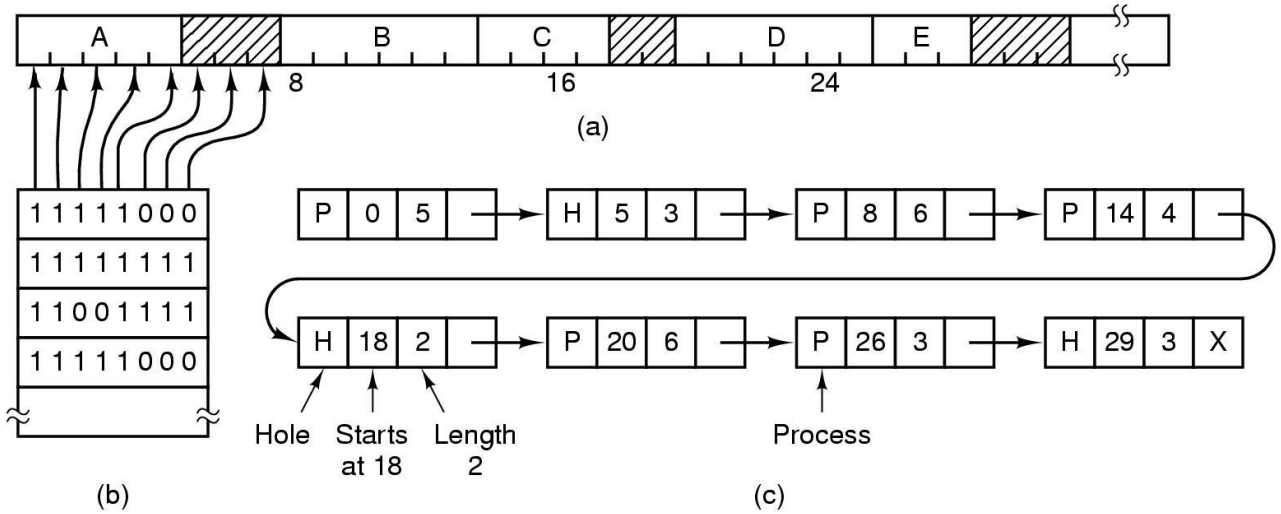
- Need to allow for programs to grow
  - Allocate more memory for data
  - Larger stack
- Handled by allocating more space than is necessary at the start
  - Inefficient: wastes memory that's not currently in use
  - What if the process requests too much memory?



Tracking Memory Usage

- When the memory is allocated dynamically, the operating system should manage it.
- There are two main ways to keep track of memory:
  - Bit maps
  - Free lists

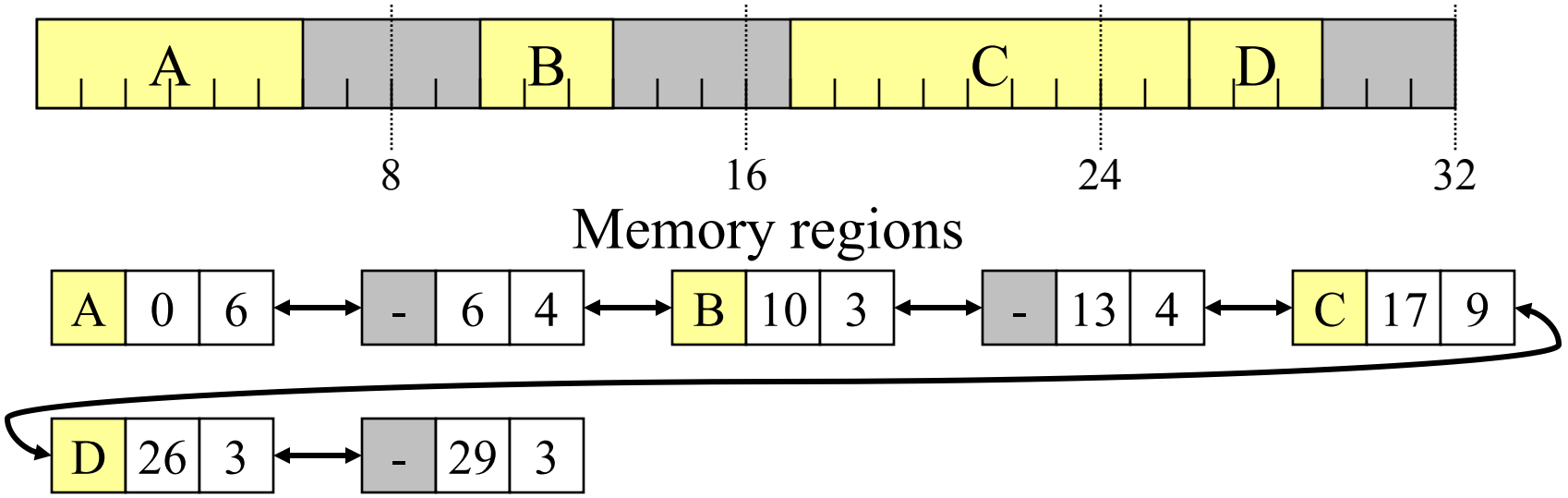
Bit Maps



- Part of memory with 5 processes, 3 holes
  - tick marks show allocation units
  - shaded regions are free
- Corresponding bit map
- Same information as a list

Linked Lists

- Keep track of free/allocated memory regions with a linked list
  - Each entry in the list corresponds to a contiguous region of memory
  - Entry can indicate either allocated or free (and, optionally, owning process)
  - May have separate lists for free and allocated areas



Allocating Memory

- There are some algorithms for memory allocation in the case of using link lists:
  - First fit
  - Next fit
  - Best fit
  - Worst fit
  - Quick fit

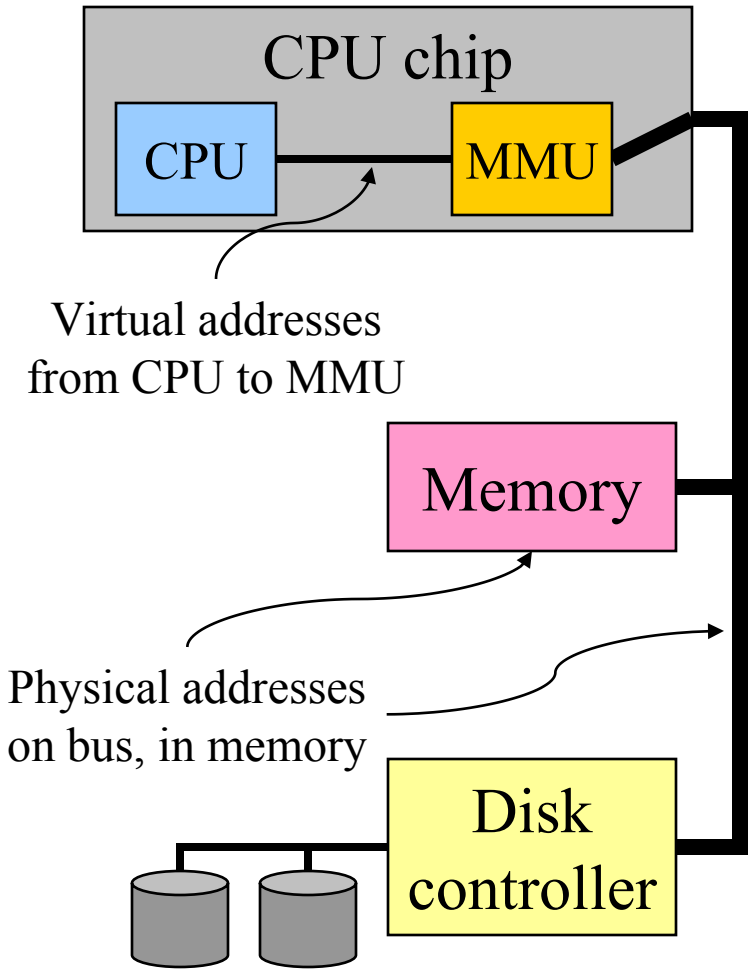
Swapping Problems

- Problems with swapping
  - Process must fit into physical memory (impossible to run larger processes)
  - Memory becomes fragmented (External)
  - Processes are either in memory or on disk: half and half doesn't do any good

Virtual Memory

- Basic idea: allow the OS to hand out more memory than exists on the system
- Keep recently used stuff in physical memory and move less recently used stuff to disk
- Keep all of this hidden from processes
  - Processes still see an address space from 0 – max address
  - Movement of information to and from disk handled by the OS without process help
- Overlay
  - Performed by programmers

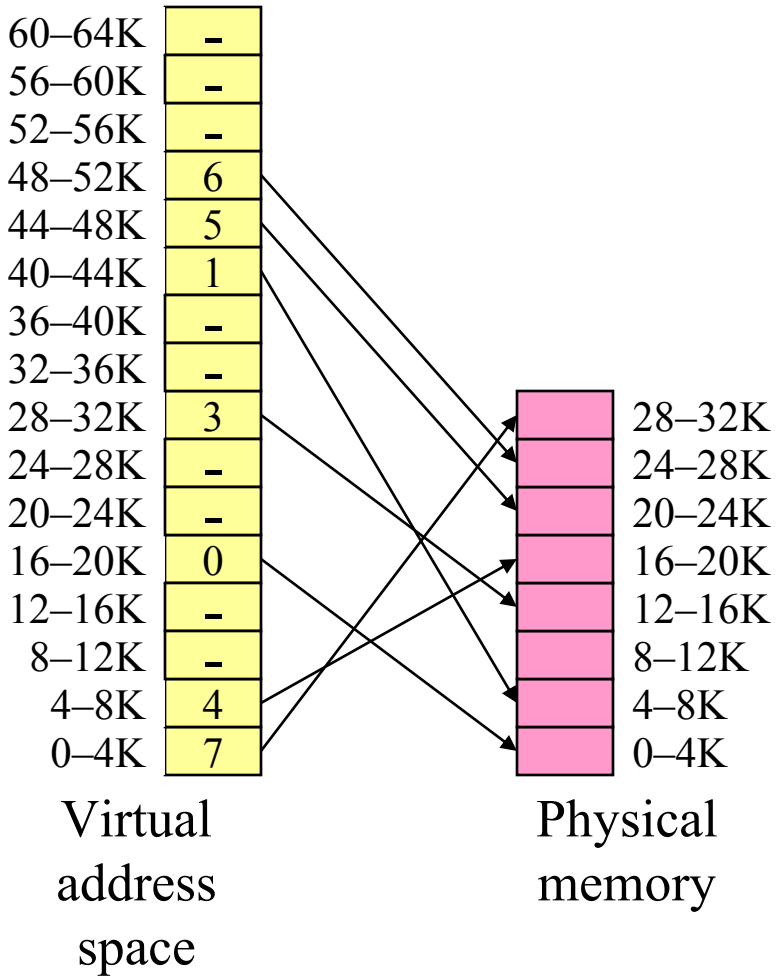
Virtual and physical addresses



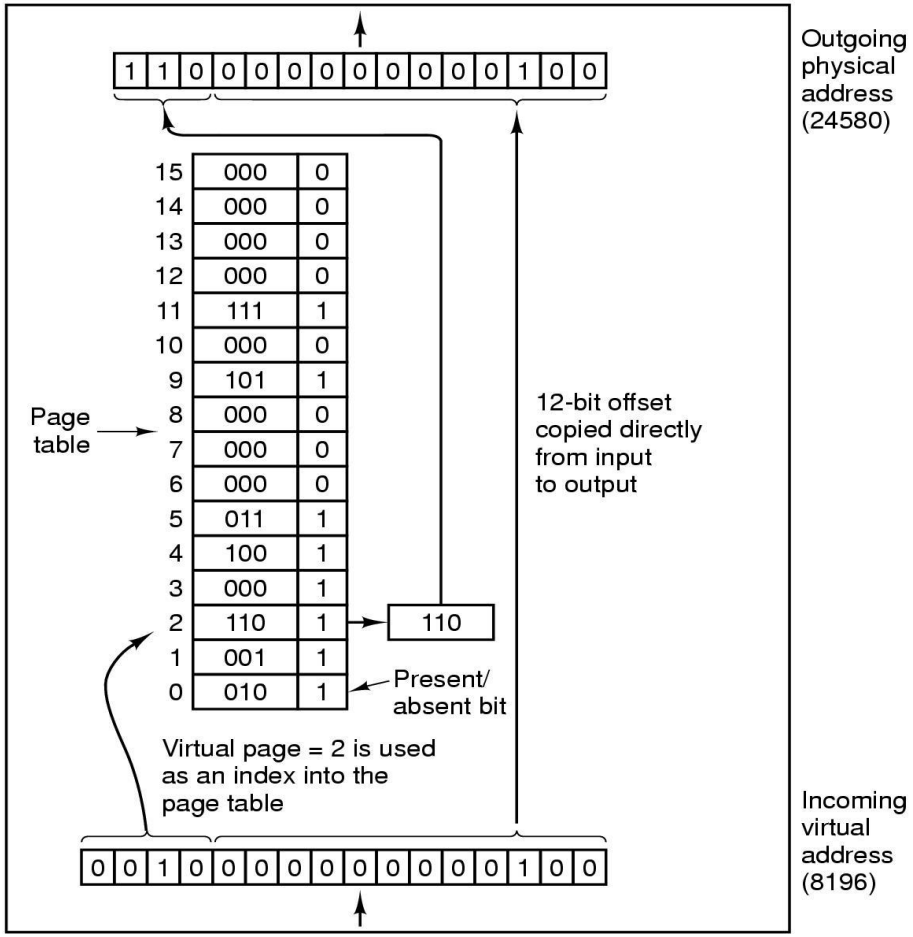
- Program uses *virtual addresses*
  - Addresses local to the process
  - Hardware translates virtual address to *physical address*
- Translation done by the ***Memory Management Unit***
  - Usually on the same chip as the CPU
  - Only physical addresses leave the CPU/MMU chip
- Physical memory indexed by physical addresses

Paging and Page Table

- Virtual addresses mapped to physical addresses
  - Unit of mapping is called a *page*
  - All addresses in the same virtual page are in the same physical page
  - *Page table entry* (PTE) contains translation for a single page
- Table translates virtual page number to physical page number
  - Not all virtual memory has a physical page
- Example:
  - 64 KB virtual memory
  - 32 KB physical memory



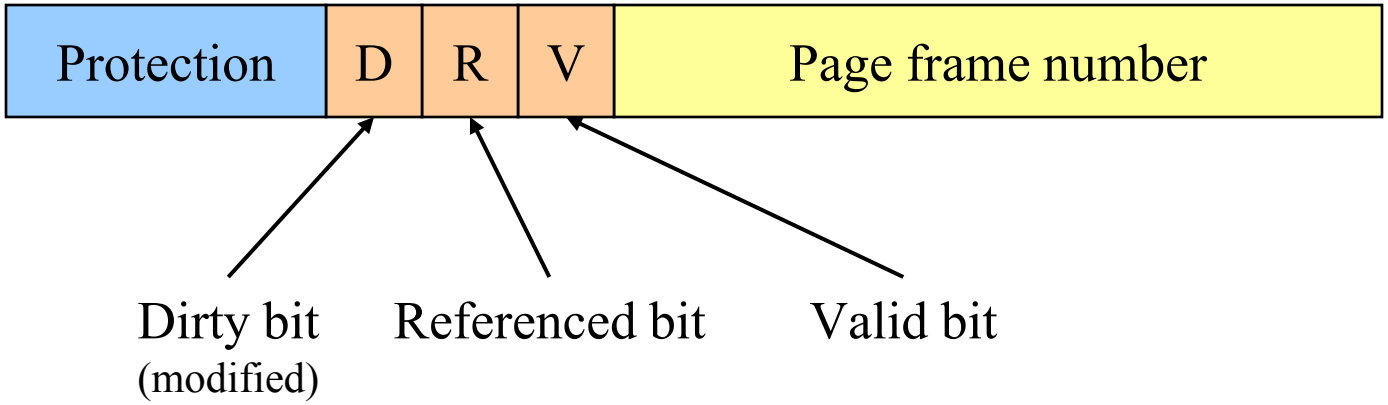
Single level Page Tables



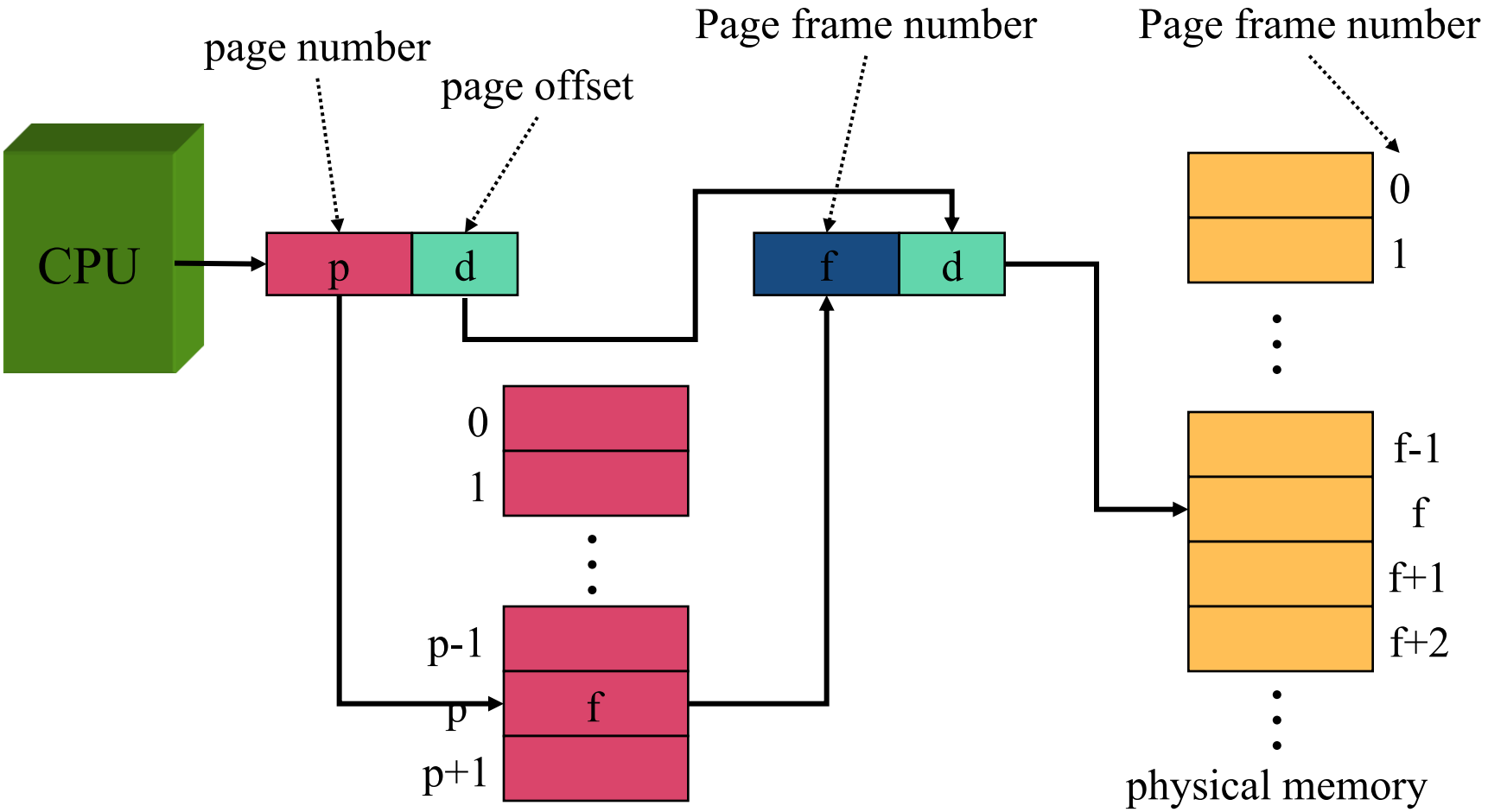
Internal operation of MMU with 16 pages (4 KB)

What's in a page table entry?

- Each entry in the page table contains
  - Valid bit: set if this logical page number has a corresponding physical frame in memory
    - If not valid, remainder of PTE is irrelevant
  - Page frame number: page in physical memory
  - Referenced bit: set if data on the page has been accessed
  - Dirty (modified) bit :set if data on the page has been modified
  - Protection information (read, write, Executable)



Address translation architecture

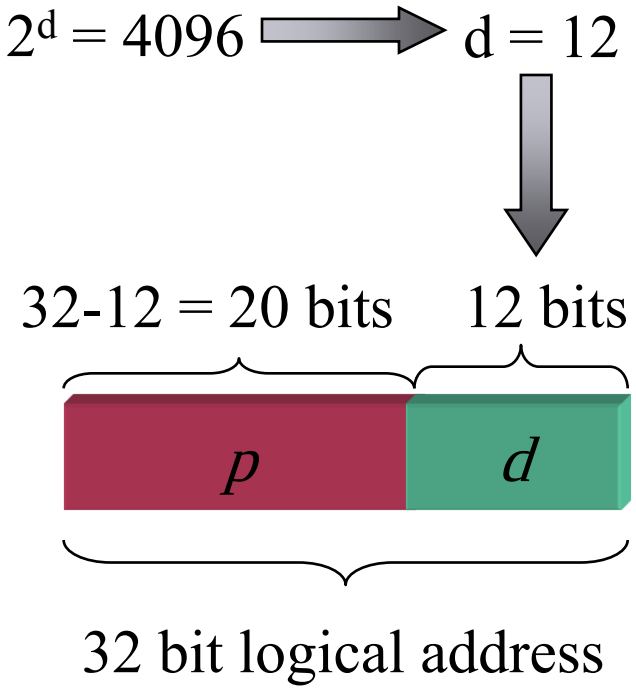


Mapping logical => physical address

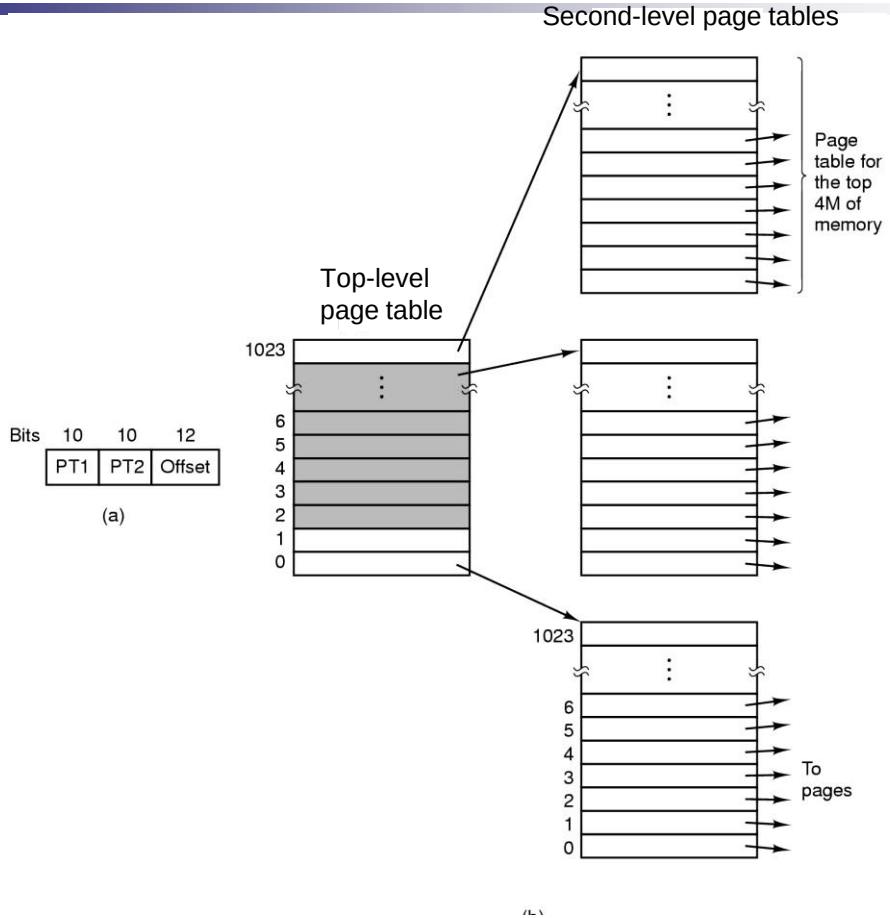
- Split address from CPU into two pieces
  - Page number (*p*)
  - Page offset (*d*)
- Page number
  - Index into page table
  - Page table contains base address of page in physical memory
- Page offset
  - Added to base address to get actual physical memory address
- Page size = 2^d bytes

Example:

- 4 KB (=4096 byte) pages
- 32 bit logical addresses



Multi level Page Tables

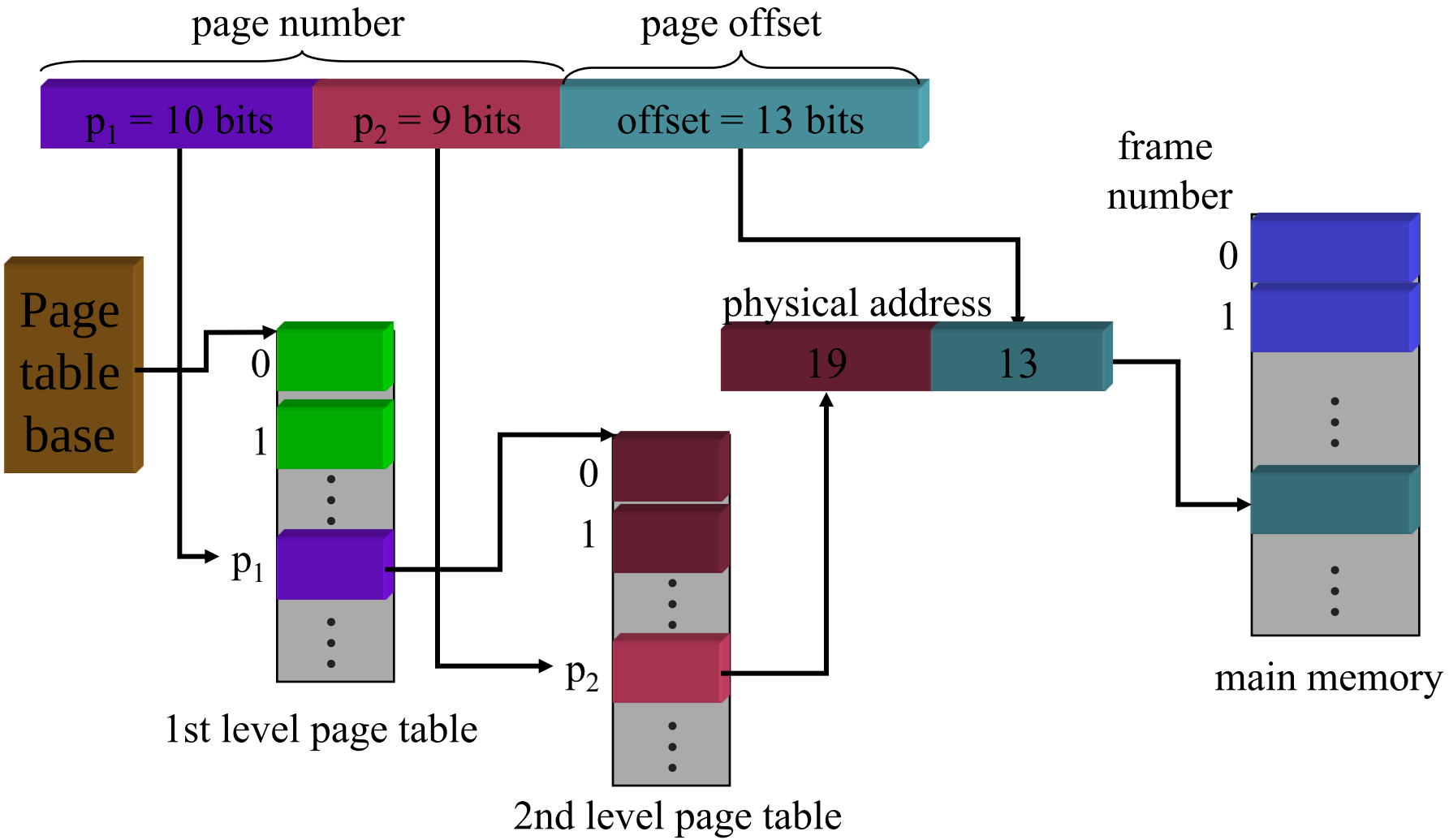


- 32 bit address with 2 page table fields
- Two-level page tables

Two-level paging: example

- System characteristics
  - 8 KB pages
  - 32-bit logical address divided into 13 bit page offset, 19 bit page number
- Page number divided into:
  - 10 bit page number
  - 9 bit page offset
- Logical address looks like this:
  - p_1 is an index into the 1st level page table
  - p_2 is an index into the 2nd level page table pointed to by p_1

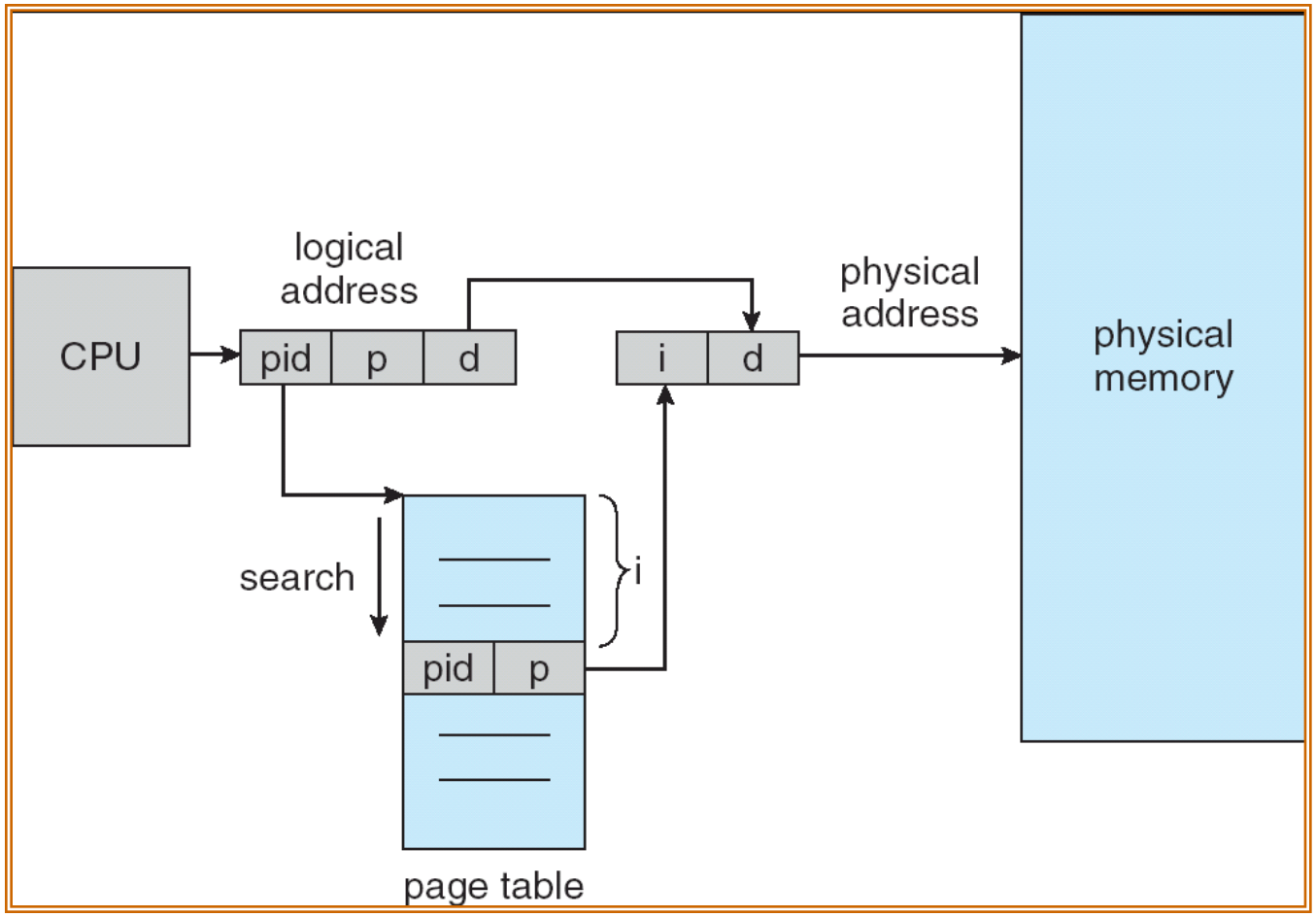
2-level address translation example



Inverted page table

- Reduce page table size further: keep one entry for each frame in memory
- PTE contains
  - Virtual address pointing to this frame
  - Information about the process that owns this page
- Search page table by
  - Hashing the virtual page number and process ID
  - Starting at the entry corresponding to the hash result
  - Search until either the entry is found or a limit is reached
- Page frame number is index of PTE
- Improve performance by using more advanced hashing algorithms

Inverted page table



TLBs – Translation Lookaside Buffers

- Based on *Locality of Reference*
- A hardware in MMU

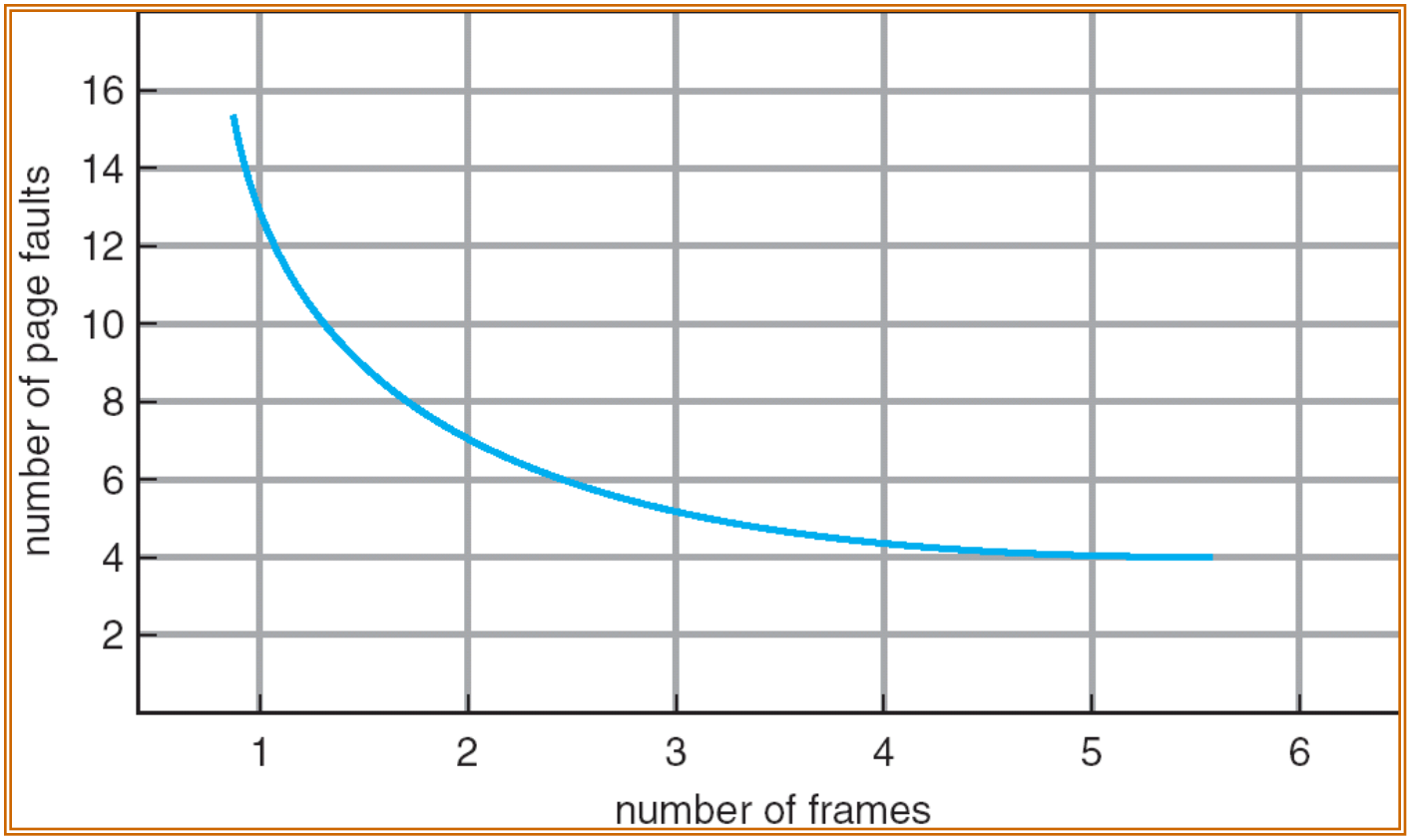
Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

- All the page table would be kept on main memory.
- Effective Access Time:
  - $T_{Access} = T_{TLB} + (1-H_{TLB})*(T_{memory}) + T_{memory}$

Page Replacement Algorithms

- Page fault forces a choice
  - No room for new page
  - Which page must be removed to make room for an incoming page?
- How a page is removed from physical memory?
  - If the page is unmodified, simply overwrite it: a copy already exists on disk
  - If the page has been modified, it must be written back to disk: prefer unmodified pages?

Page Faults


number of page faults
16
14
12
10
8
6
4
2
1
2
3
4
5
6
number of frames

Optimal Page Replacement

- What's the best we can possibly do?
  - Assume perfect knowledge of the future
  - Not realizable in practice (usually)
  - Useful for comparison: if another algorithm is within 5% of optimal, not much more can be done…
- Algorithm: replace the page that will be used furthest in the future
  - Only works if we know the whole sequence!
  - Can be approximated by running the program twice
    - Once to generate the reference trace
    - Once (or more) to apply the optimal algorithm
- Nice, but not achievable in real systems!

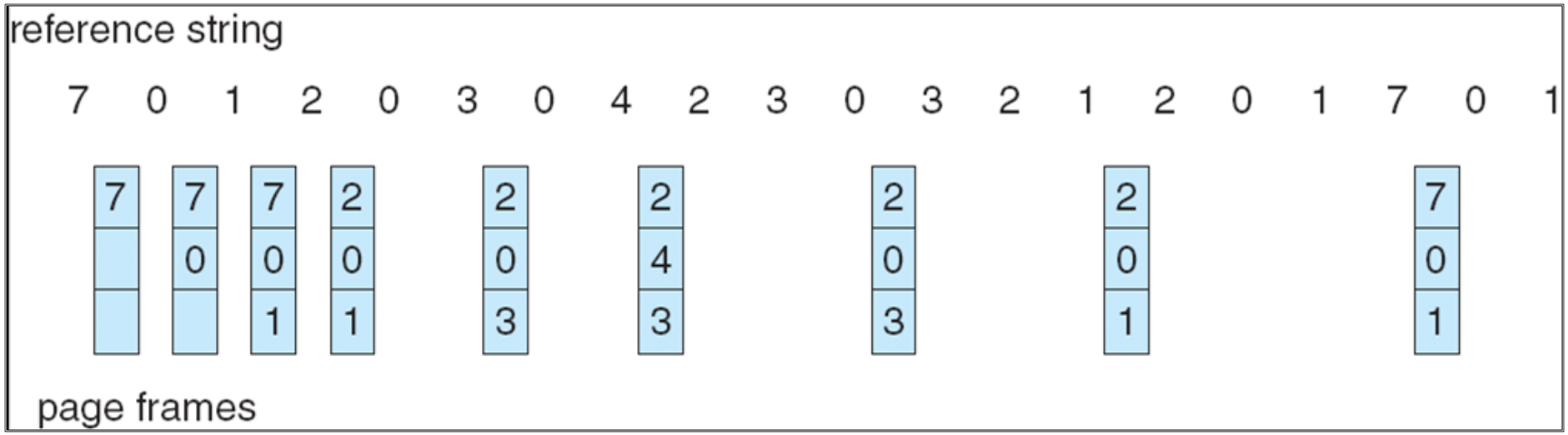
Example 1

- Replace the page that will not be used for longest period of time
- 4 frames example

  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5, 4, 3, 2,1

- Used for measuring how well your algorithm performs

Example 2



- The reference string of a process has been depicted here.

First-In, First-Out (FIFO) Algorithm

- Maintain a list of all pages
  - Maintain the order in which they entered memory
- Page at front of list would be replaced
- Advantage: (really) easy to implement
- Disadvantage: old pages in memory may still be in use heavily.
  - This algorithm forces pages out regardless of their usage
  - Usage may be helpful in determining which pages to keep

FIFO Example

Page referenced	0	1	2	3	0	1	4	0	1	2	3	4
Youngest page	0	1	2	3	0	1	4	4	4	2	3	3
		0	1	2	3	0	1	1	1	4	2	2
Oldest page			0	1	2	3	0	0	0	1	4	4

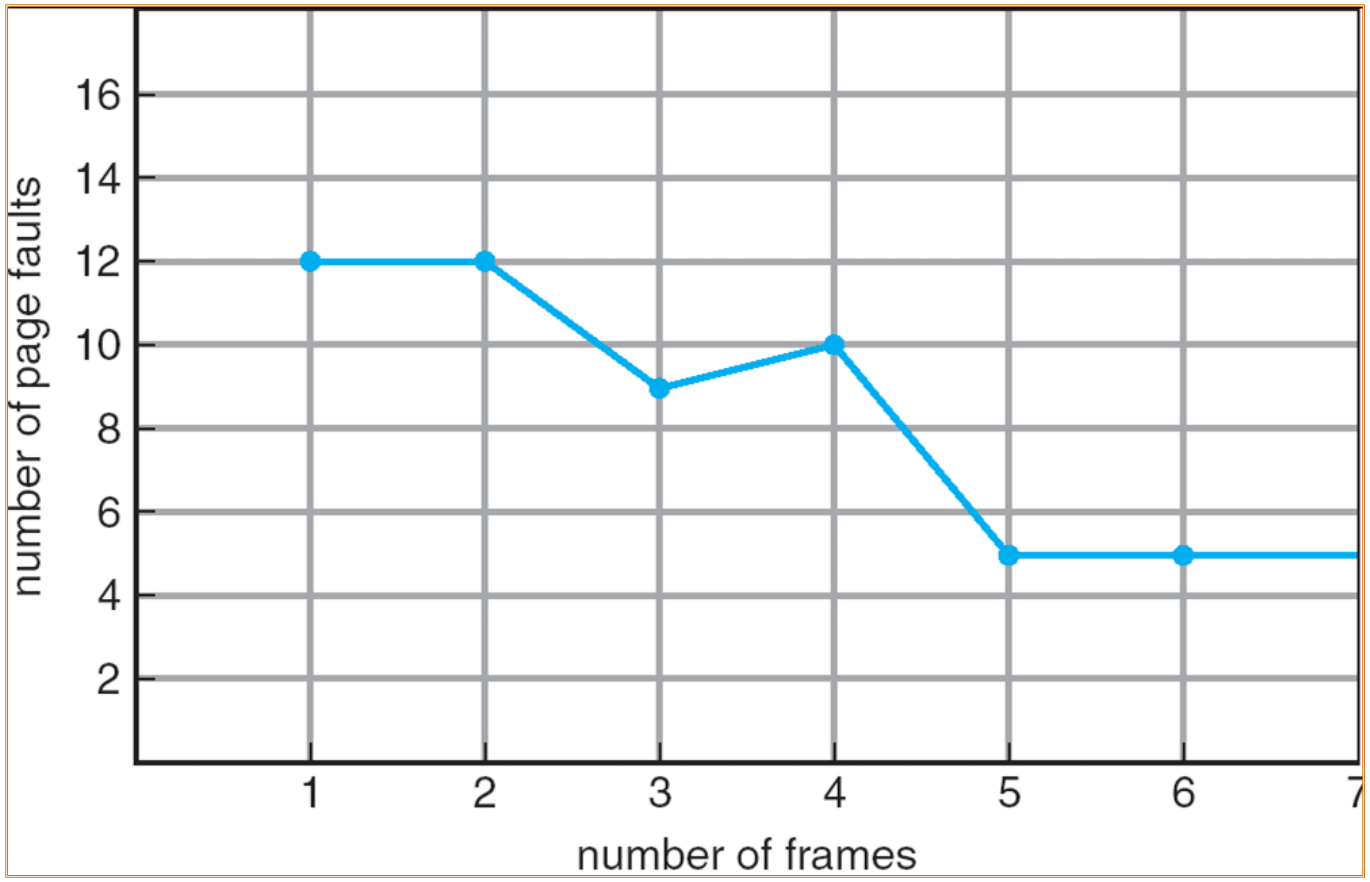
- Example: FIFO replacement on reference string
  0 1 2 3 0 1 4 0 1 2 3 4
  - Page replacements highlighted in yellow

Belady's Anomaly

- What will happen if we use 4 page frames instead of 3?
- more frames ⇒ more page faults
- This is called Belady's Anomaly
- Stack algorithms have not Belady Anomaly

Page referenced	0	1	2	3	0	1	4	0	1	2	3	4
Youngest page	0	1	2	3	3	3	4	0	1	2	3	4
		0	1	2	2	2	3	4	0	1	2	3
			0	1	1	1	2	3	4	0	1	2
Oldest page				0	0	0	1	2	3	4	0	1

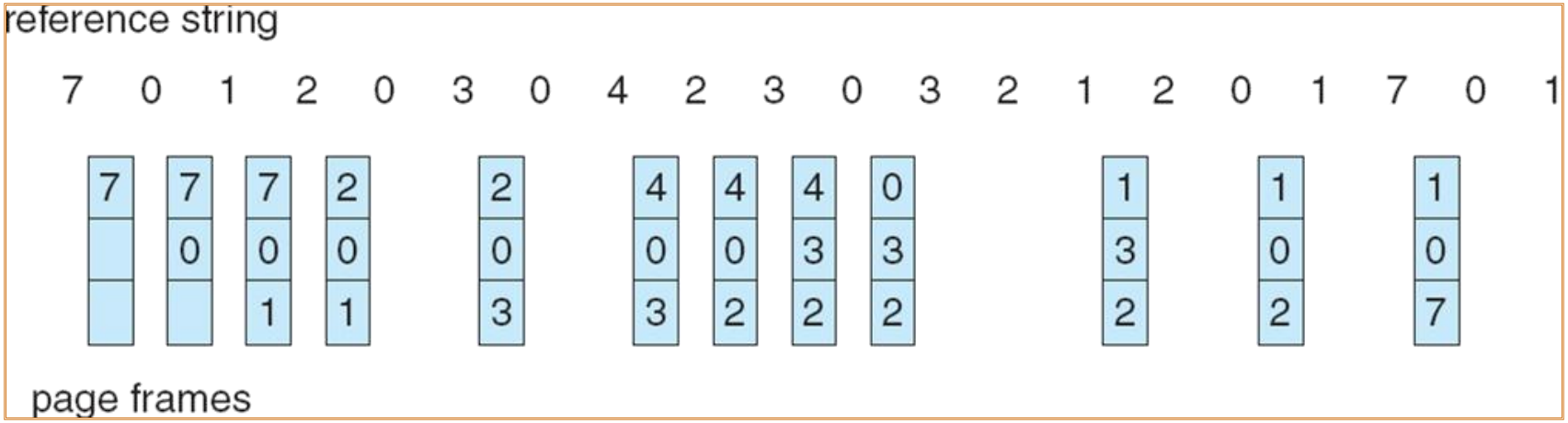
FIFO and Belady's anomaly



Least Recently Used (LRU)

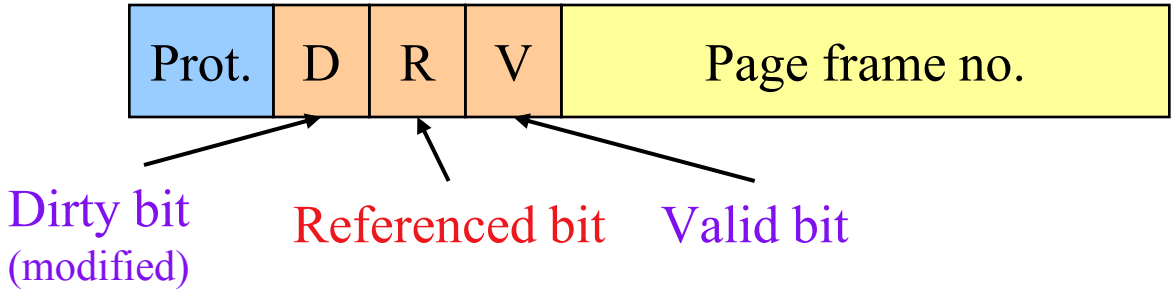
- Assume pages which are used recently will be used again soon
  - Throw out page that has been unused for longest time
- Implementation method: Counter, Matrix, Linked list
- Keep a counter in each page table entry
  - Global counter increments with each instruction
  - Copy global counter to PTE counter on a reference to the page
  - For replacement, evict page with lowest counter value
- What happens when the counter reaches Max?

LRU Example



- Some efforts related to LRU:
  - NRU
  - Second Chance
  - Clock

Not Recently Used (NRU)

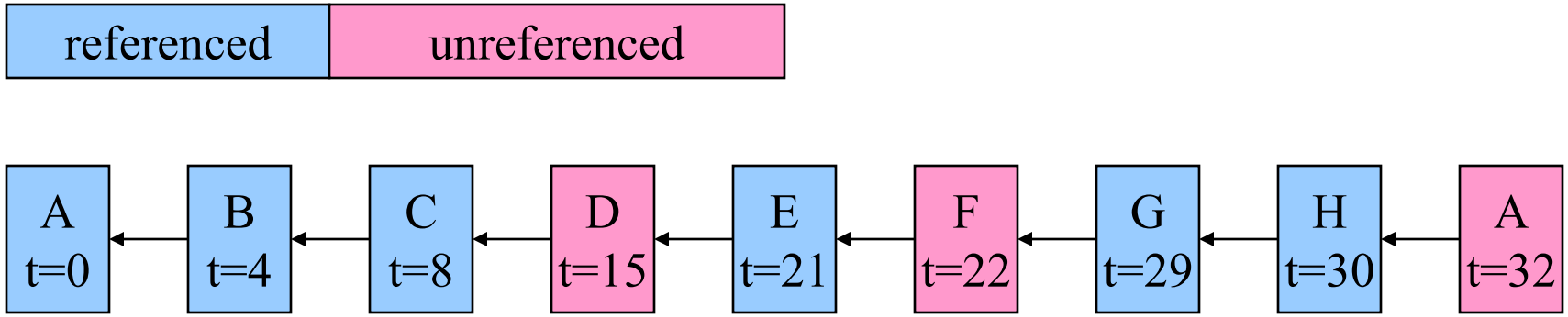


- Pages are classified into four classes
  - 0: not referenced, not dirty (0 0)
  - 1: not referenced, dirty (0 1)
  - 2: referenced, not dirty (1 0)
  - 3: referenced, dirty (1 1)
- NRU removes a page at random form lowest numbered nonempty class.
- Clear reference bit for all pages periodically

Not Recently Used (NRU)

- Easy to implement
- Efficient
- Easy to understand
- And certainly not optimal.

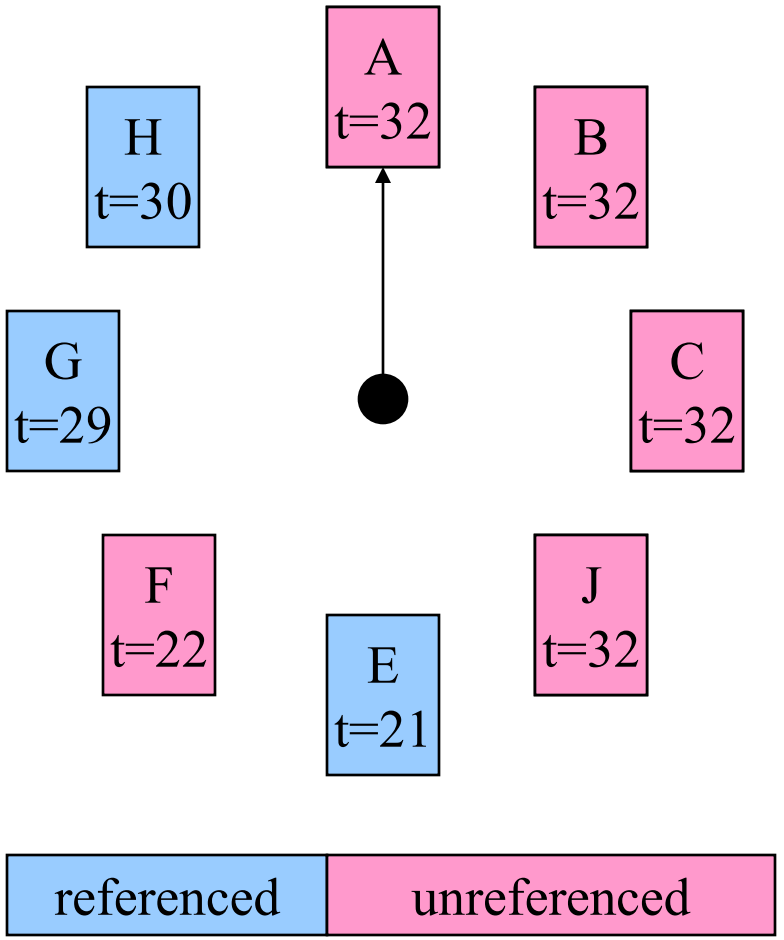
Second Chance Algorithm



- Modify FIFO to avoid throwing out heavily used pages
  - If reference bit is 0, throw the page out
  - If reference bit is 1
    - Reset the reference bit to 0
    - Move page to the tail of the list
    - Continue search for a free page
- Still easy to implement, and better than plain FIFO

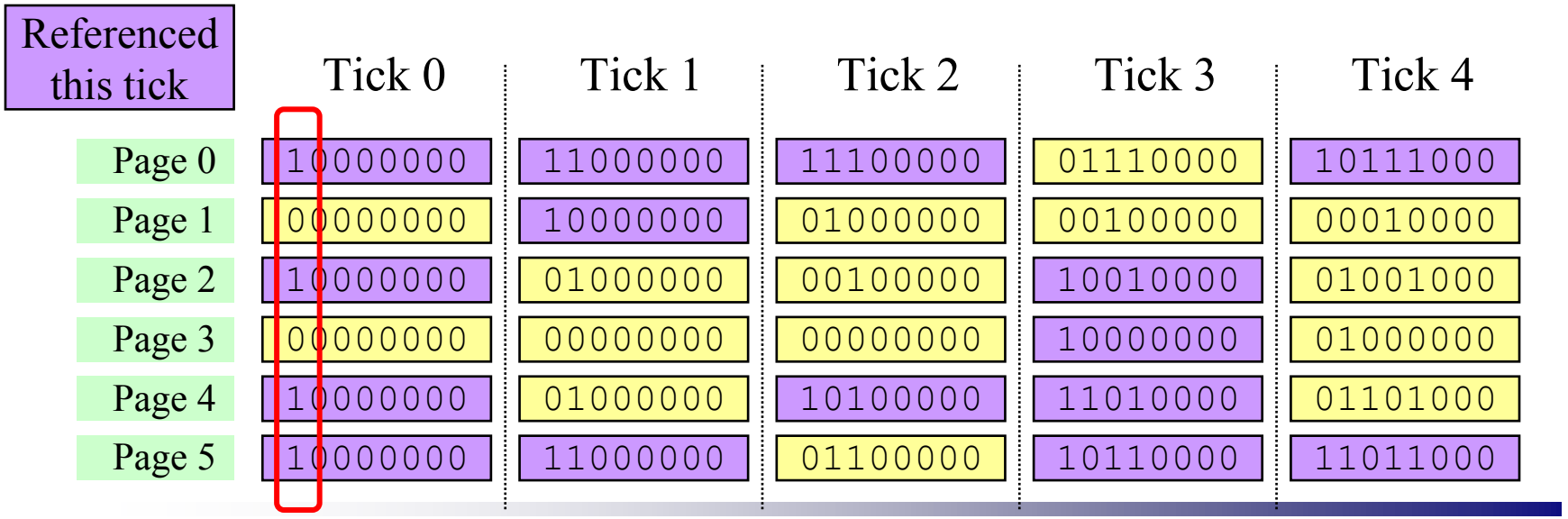
Clock Algorithm

- Same functionality as second chance
- In Second Chance Add and Remove have const!
- Simpler implementation
  - "Clock" hand points to next page to replace
  - If R=0, replace page and advance the clock hand
  - If R=1, set R=0 and advance the clock hand
- No overhead for moving pages.



Aging Algorithm

- Simulation of LRU in software
- Algorithm is:
  - Every clock tick, shift all counters right by 1 bit
  - On reference, Copy the reference bit to the counter at the clock tick
  - Clear reference bit all page



Counting Algorithms

- Keep a counter of the number of references that have been made to each page

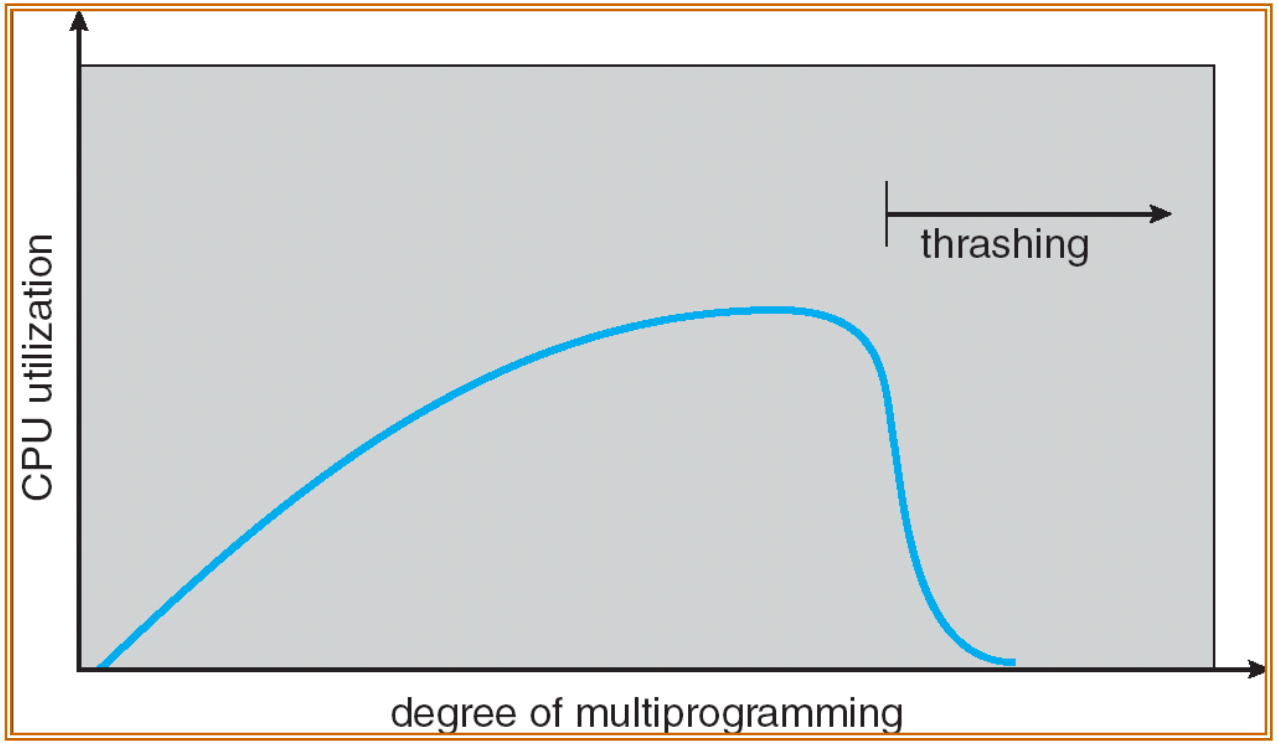
- **LFU Algorithm**: (Least Frequently Used)
  - replaces page with smallest count

- **MFU Algorithm**: (Most Frequently Used)
  - based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks 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

- Bring a page into memory when it's requested by the process
- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap

Working Set Model

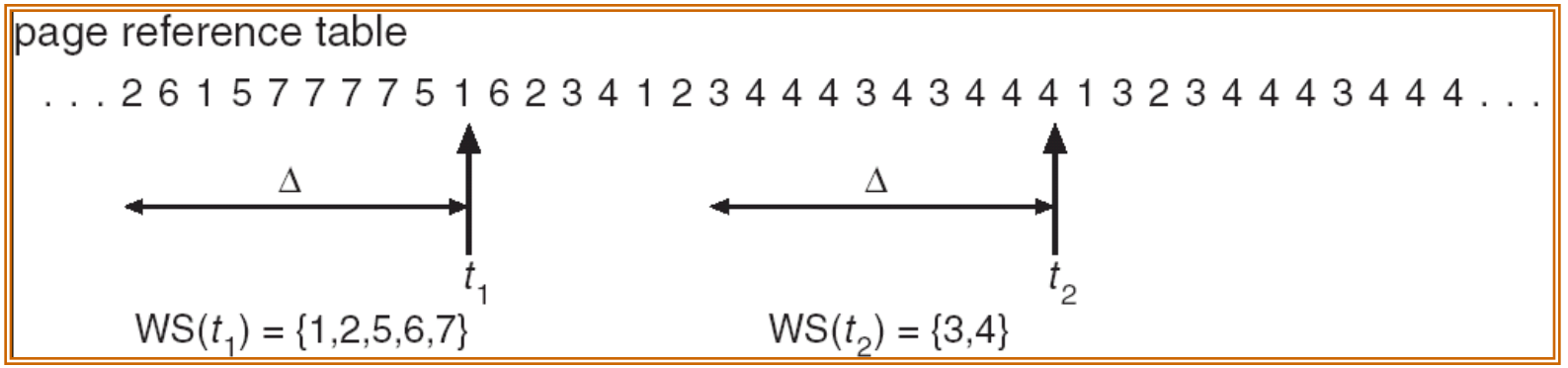
- Locality of Reference:
  - during any phase of execution, the process references only a relatively small fraction of its pages.
- Working Set:
  - The set of pages that a process is currently using.
  - Prepaging
- w(k,t)

Working set

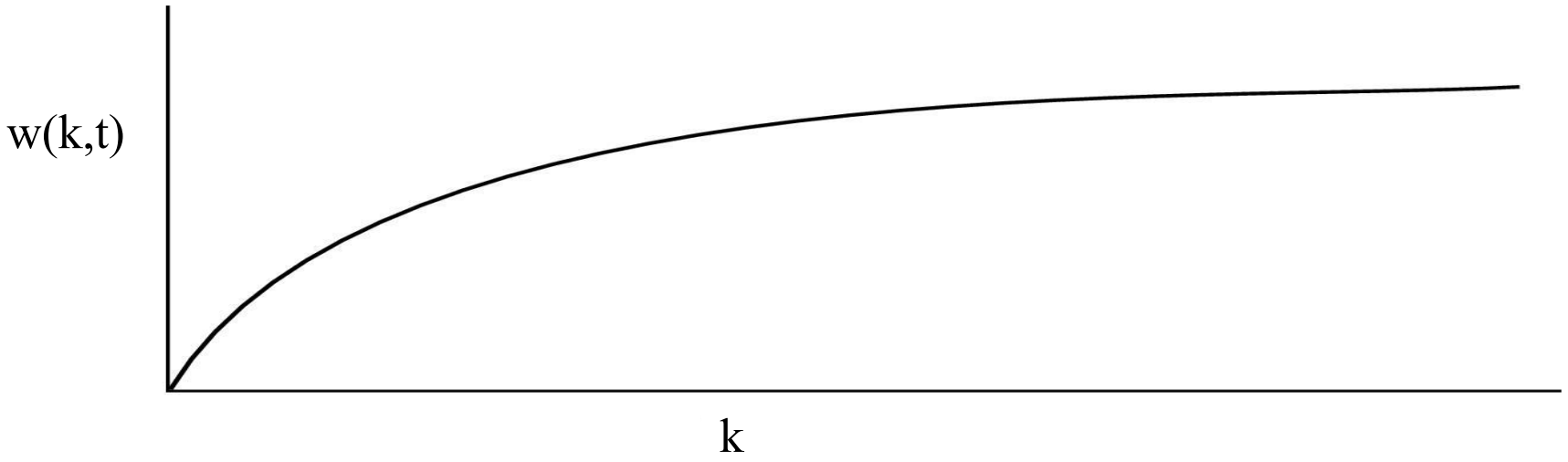
- How many pages are needed?
  - Could be all of them, but not likely
  - Instead, processes reference a small set of pages at any given time - *locality of reference*
  - Set of pages can be different for different processes or even different times in the running of a single process

- Set of pages used by a process in a given interval of time is called the *working set*
  - If entire working set is in memory, no page faults!
  - If insufficient space for working set, thrashing may occur
  - Goal: keep most of working set in memory to minimize the number of page faults suffered by a process

Working set example

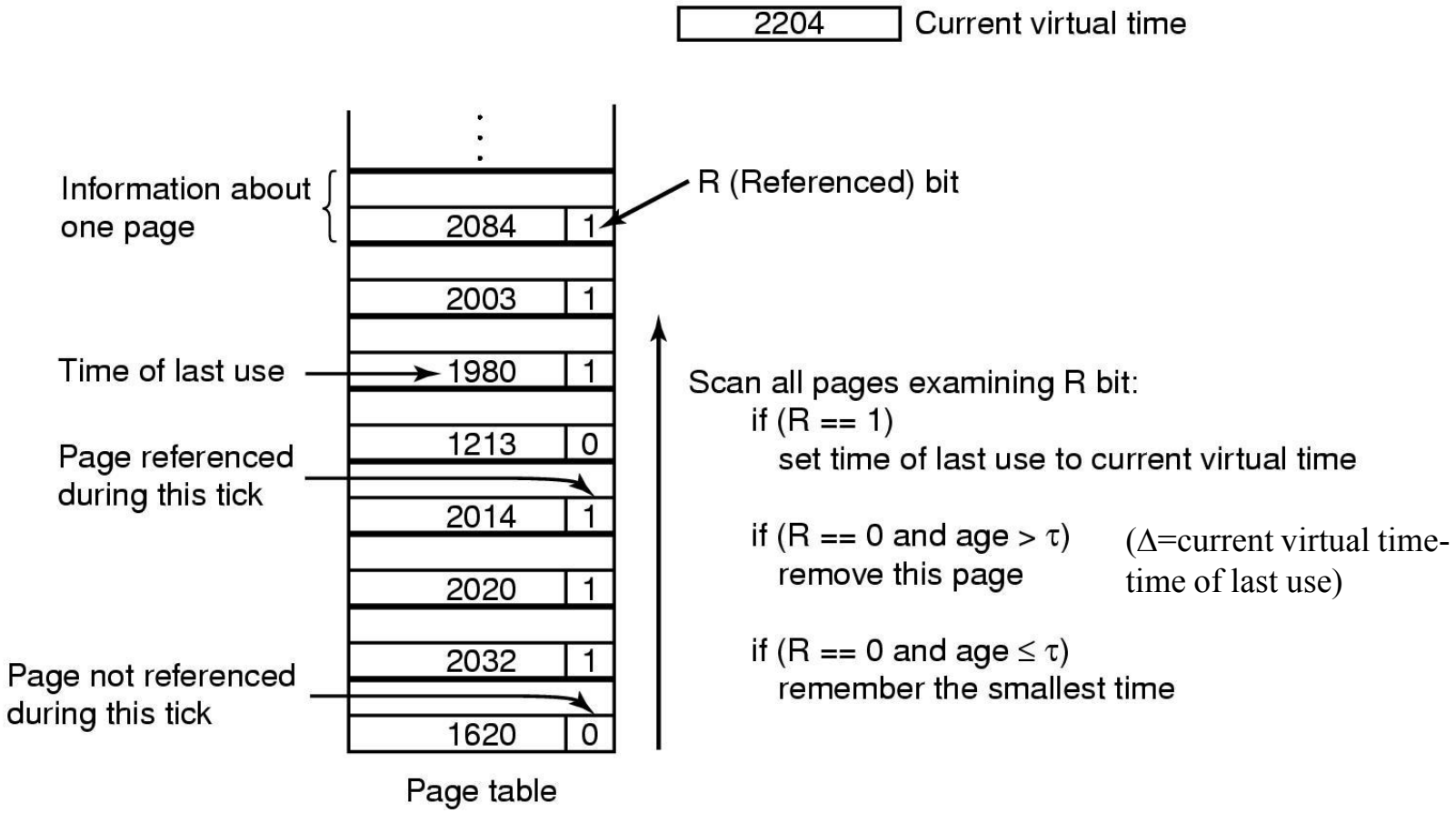


How big is the working set?



- Working set is the set of pages used by the *k* most recent memory references
- w(k,t) is the size of the working set at time *t*
- Working set may change over time
  - Size of working set can change over time as well…

Working set page replacement algorithm



Working Set example

- Example: $\Delta = 10{,}000$
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

Page replacement algorithms: summary

Algorithm	Comment
OPT (Optimal)	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Crude
FIFO (First-In, First Out)	Might throw out useful pages
Second chance	Big improvement over FIFO
Clock	Better implementation of second chance
LRU (Least Recently Used)	Excellent, but hard to implement exactly
NFU (Not Frequently Used)	Poor approximation to LRU
Aging	Good approximation to LRU, efficient to implement
Working Set	Somewhat expensive to implement
WSClock	Implementable version of Working Set

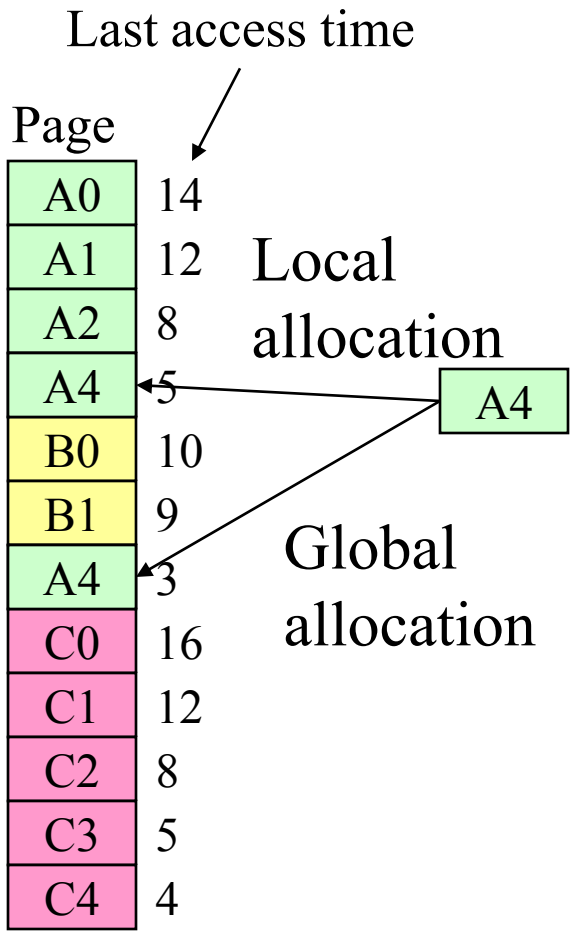
Global & Local Allocation

- **Global replacement**

  process selects a replacement frame from the set of all frames; one process can take a frame from another

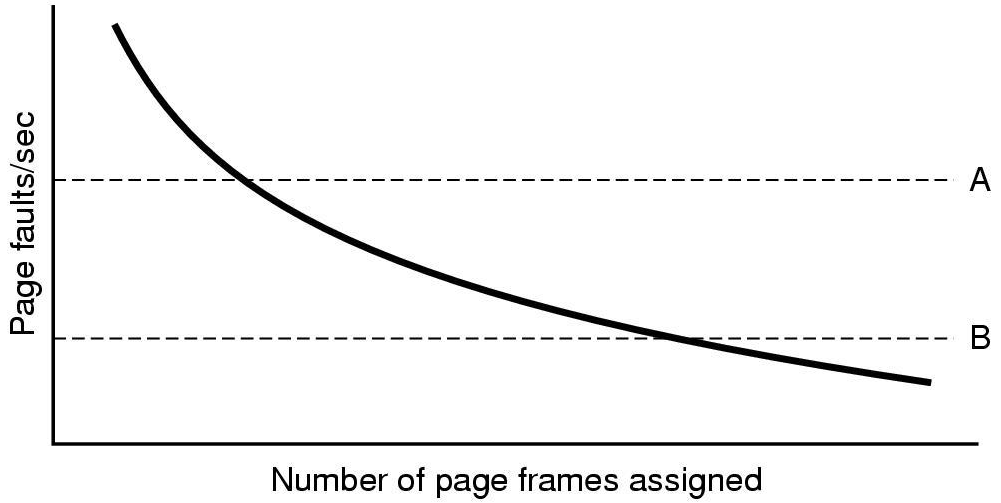
- **Local replacement**

  each process selects from only its own set of allocated frames



Page fault rate & allocated frames

- Local allocation may be more “fair”
  - Don’t penalize other processes for high page fault rate
- Global allocation is better for overall system performance
  - Take page frames from processes that don’t need them as much
  - Reduce the overall page fault rate (even though rate for a single process may go up)



Control overall page fault rate

- Despite good designs, system may still thrash
- Most (or all) processes have high page fault rate
  - Some processes need more memory, …
  - but no processes need less memory (and could give some up)
- Problem: no way to reduce page fault rate
- Solution :
  Reduce number of processes competing for memory
  - Swap one or more to disk, divide up pages they held
  - Reconsider degree of multiprogramming

How big should a page be?

- Smaller pages have advantages
  - Less internal fragmentation
  - Better fit for various data structures, code sections
  - Less unused physical memory (some pages have 20 useful bytes and the rest isn't needed currently)

- Larger pages are better because
  - Less overhead to keep track of them
    - Smaller page tables
    - TLB can point to more memory (same number of pages, but more memory per page)
    - Faster paging algorithms (fewer table entries to look through)
  - More efficient to transfer larger pages to and from disk

Page Size

- s: the average process size
- p: page size
- e: PTE size

$$\text{Overhead} = se/p + p/2$$

What's the optimum page size?