Memory Management

3 – 3.7

Address binding

- Binding
  - Association between a name and a location containing values
  - Example:

```
main:
  push rbp
  mov  rbp, rsp
  sub  rsp, 32
  lea rcx, [msg]
  call printf
  xor rax, rax
  call ExitProcess
```

```
main:
  push rbp
  mov  rbp, rsp
  sub  rsp, 32
  lea rcx, [msg]
  call 0x8FC00
  xor rax, rax
  call 0x9DDA0
```

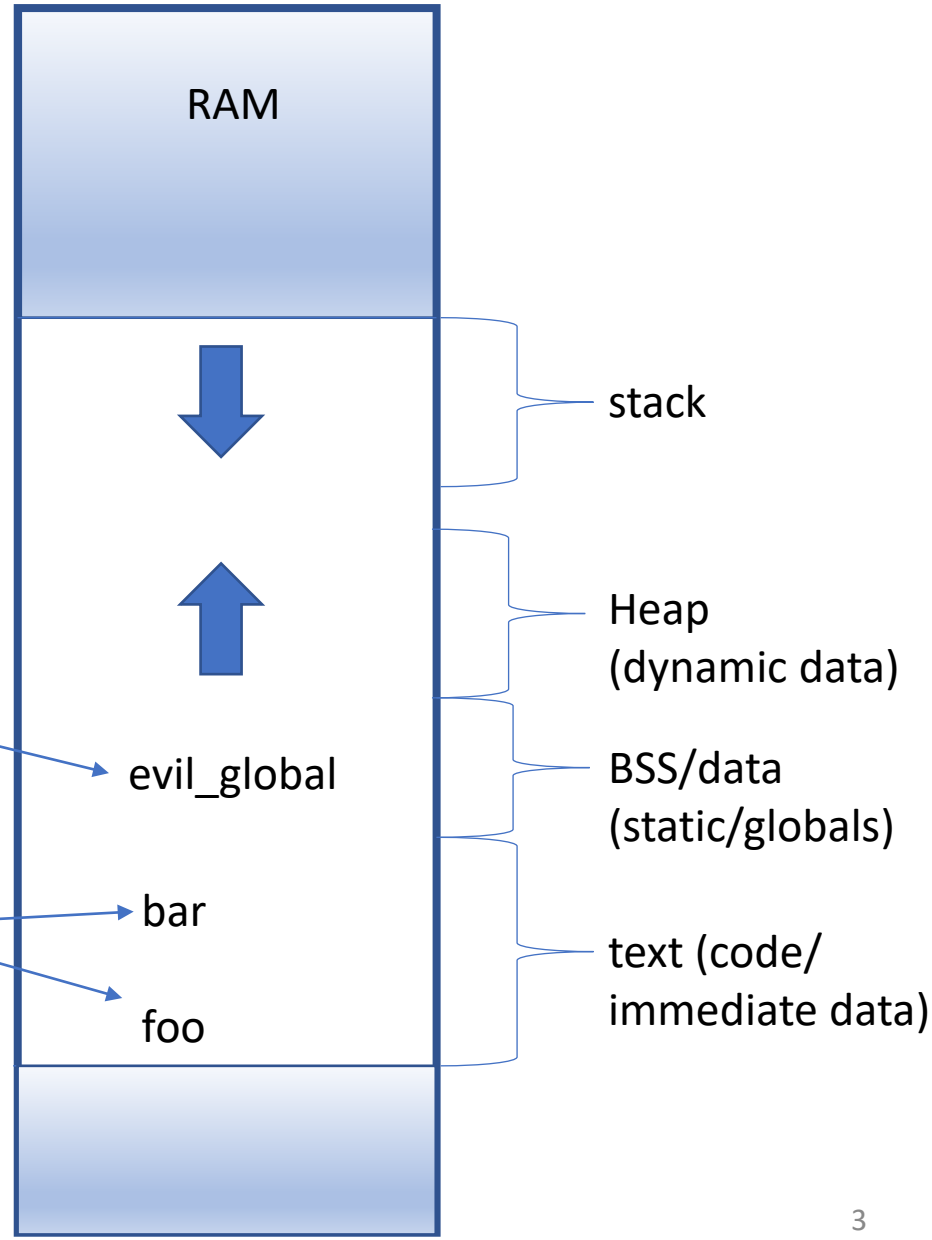
The subroutines printf and ExitProcess are stored in specific locations and the process must have actual values for these.

- How is this determined?

Address binding

- Compile time – Addresses are determined when the machine code is written.

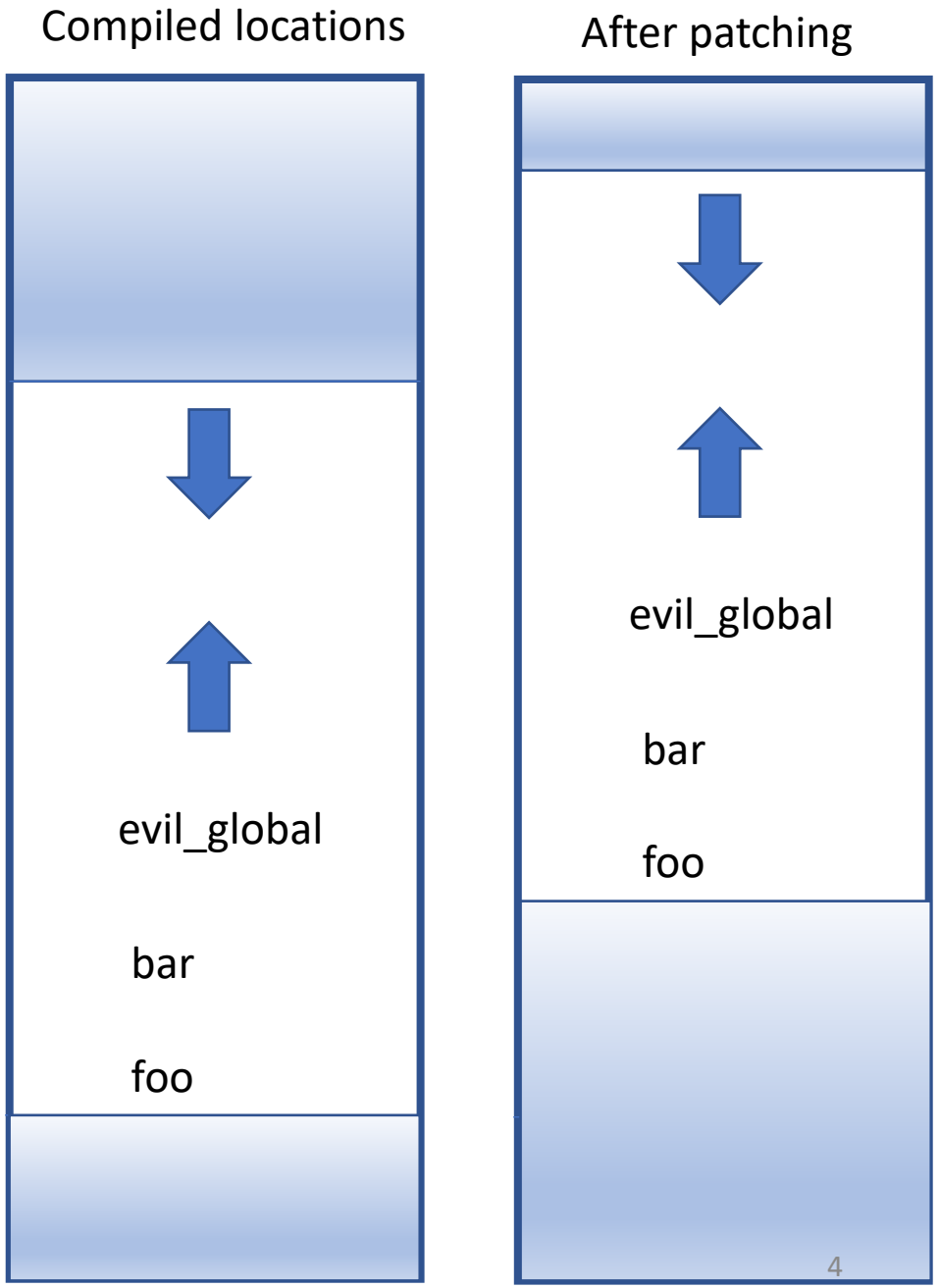
```
int evil_global = 10;

foo(…) {
   …
}

bar(…) {
   …
}
```



block started by symbol (BSS) – Region for uninitialized static data

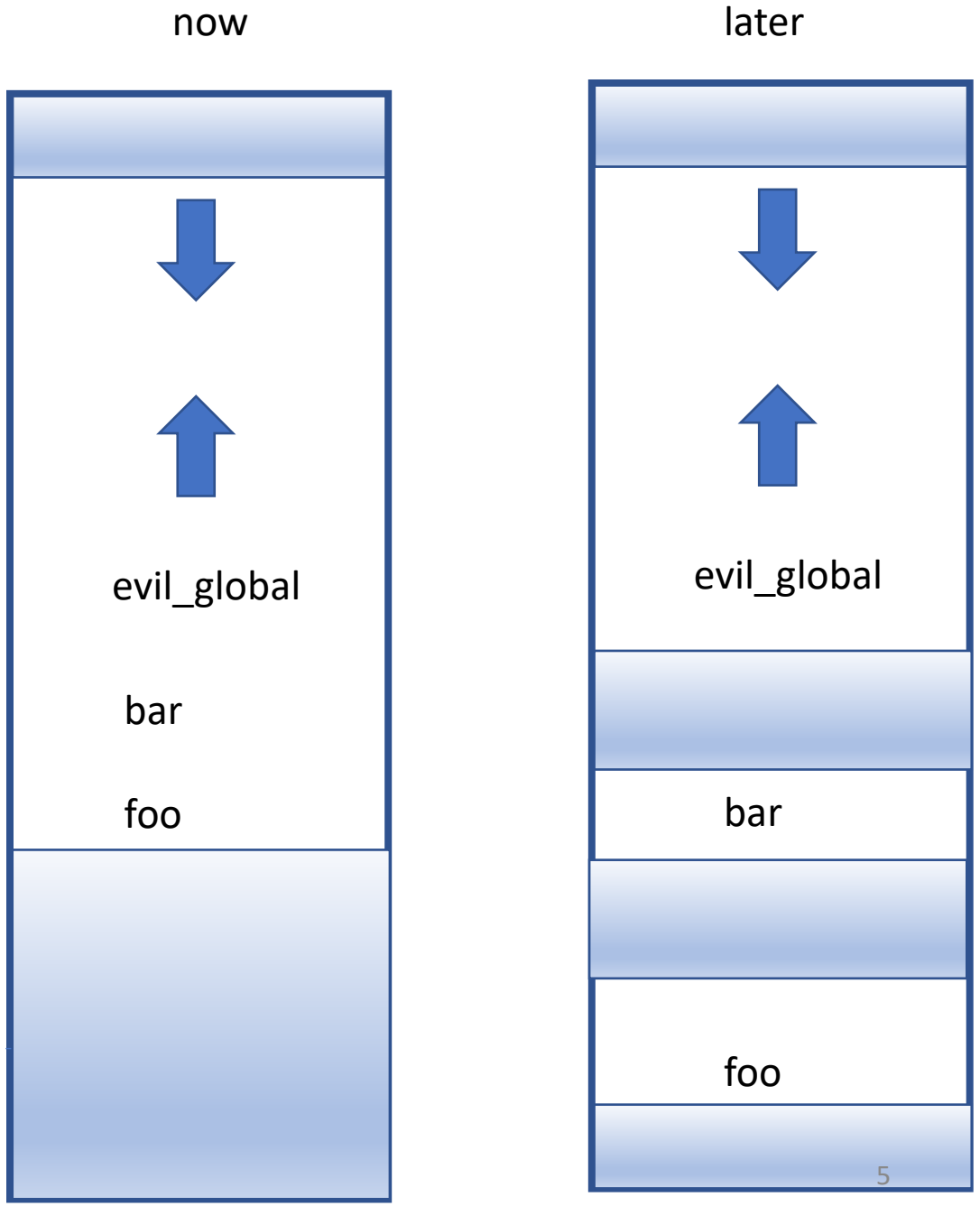
Address binding

- Load time
  - Compiler suggests locations, but includes information in object file about locations that are referencing specific addresses.
  - Loader program requests memory from OS.
  - Loader uses object file data to *patch* the object file loaded from disk to run in the locations allocated by the OS.

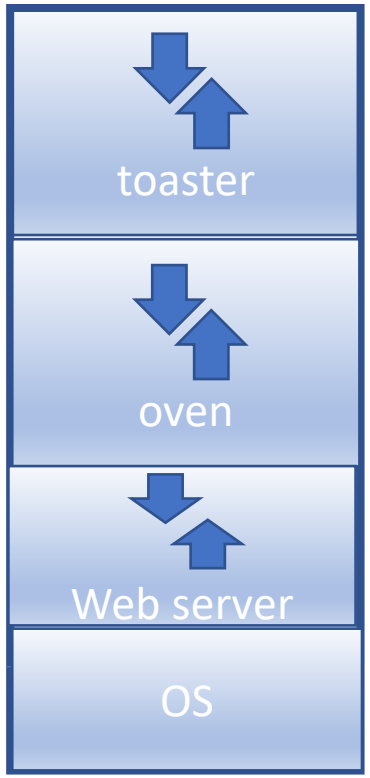


Address binding

- Execution time binding
  - Code can be relocated *while it is running* without impact.
  - Except in special cases, requires a virtual (logical) address space
  - In many virtual spaces, memory no longer needs to be contiguous!



Multiprogramming and memory management

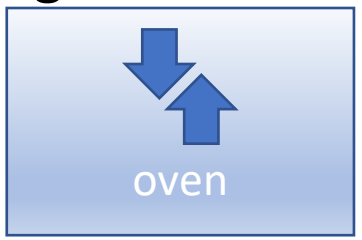


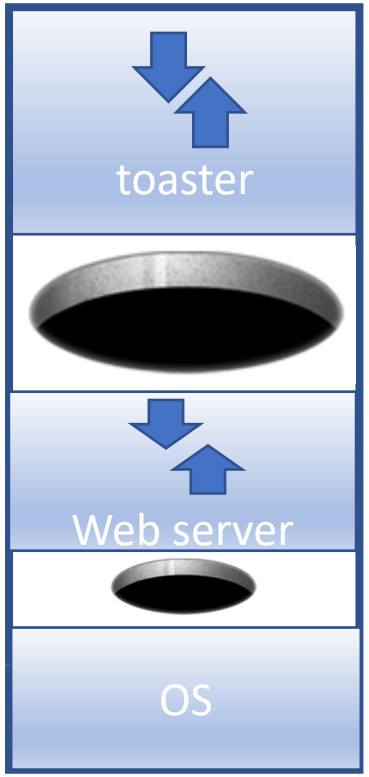
- Compile-time
  - Programs must be compiled to a static memory location.
  - Does not work for general purpose computing

TurboChef Sota Countertop Oven, yes this can be yours for only 10K...

Multiprogramming and memory management

- Load-time
  - For now, we assume all programs fit in memory.
  - Situation becomes more complicated.
  - Need to find a “hole” to place the new program.



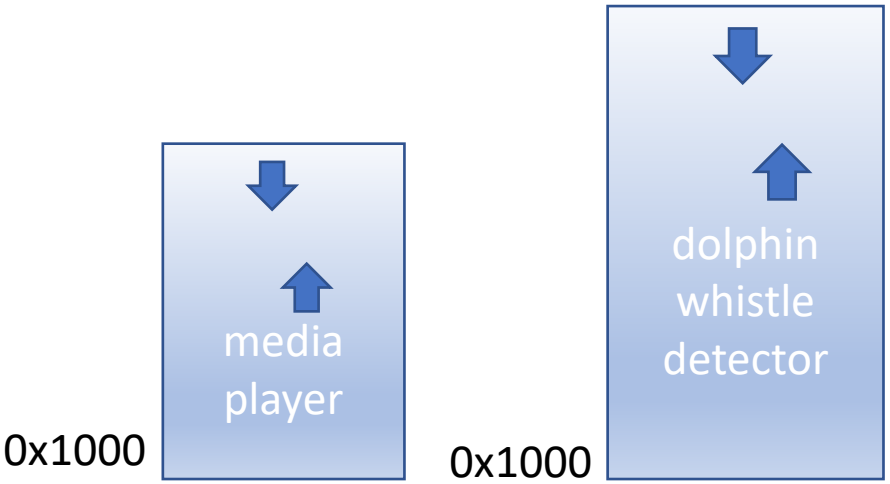


  - Two issues:
    - External fragmentation:
      - when holes are too small, they are no longer useful.
    - Internal fragmentation:
      - Must determine how much memory to allocate to a process
      - Too much → wasted space

Note: Many of the issues we will see in memory management will be revisited in file systems… oy vay!

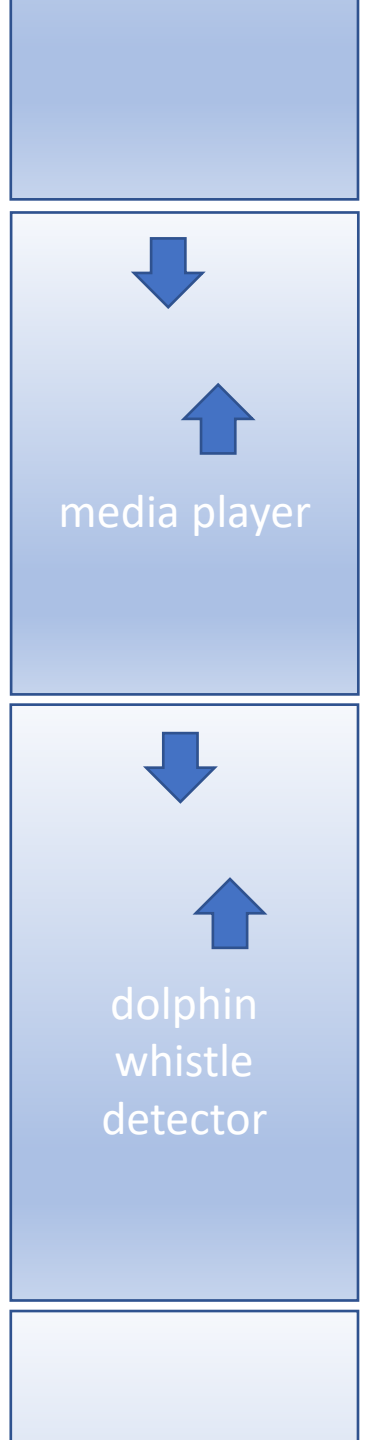
Memory abstraction

- Exposing physical address space is problematic
- Memory abstraction lets each process have its own address space
- Transparent to process



logical address space

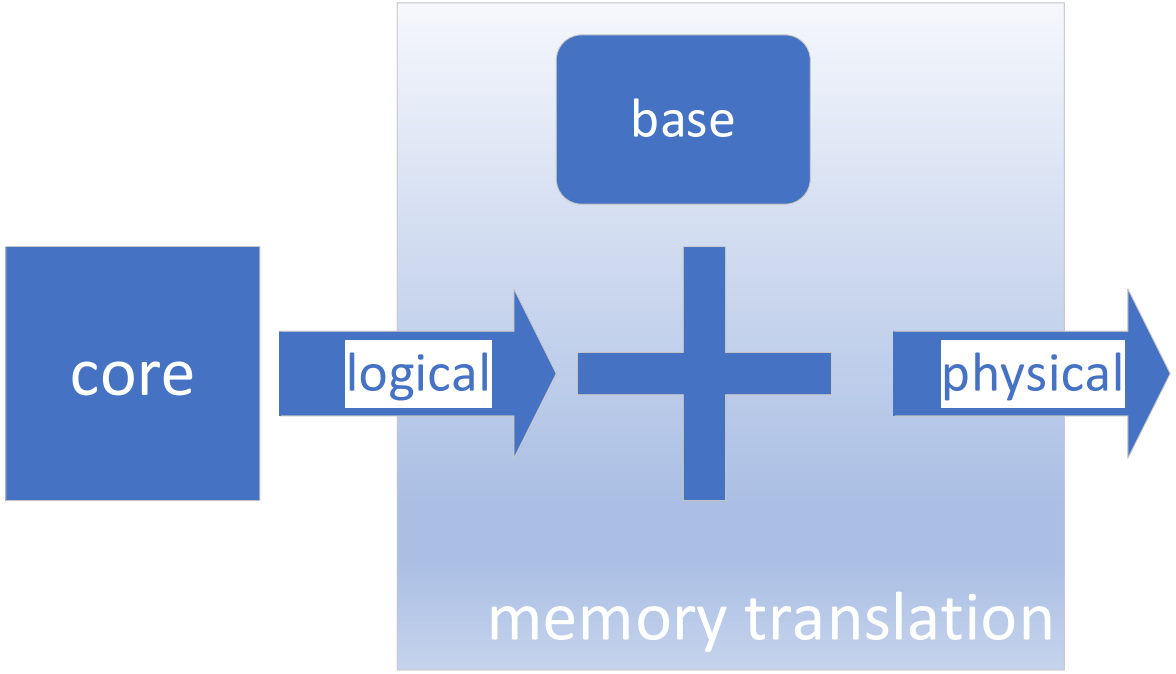
sidebar:
We did not start our logical address space at 0.
This is a common strategy, why?



physical address space

Memory abstraction

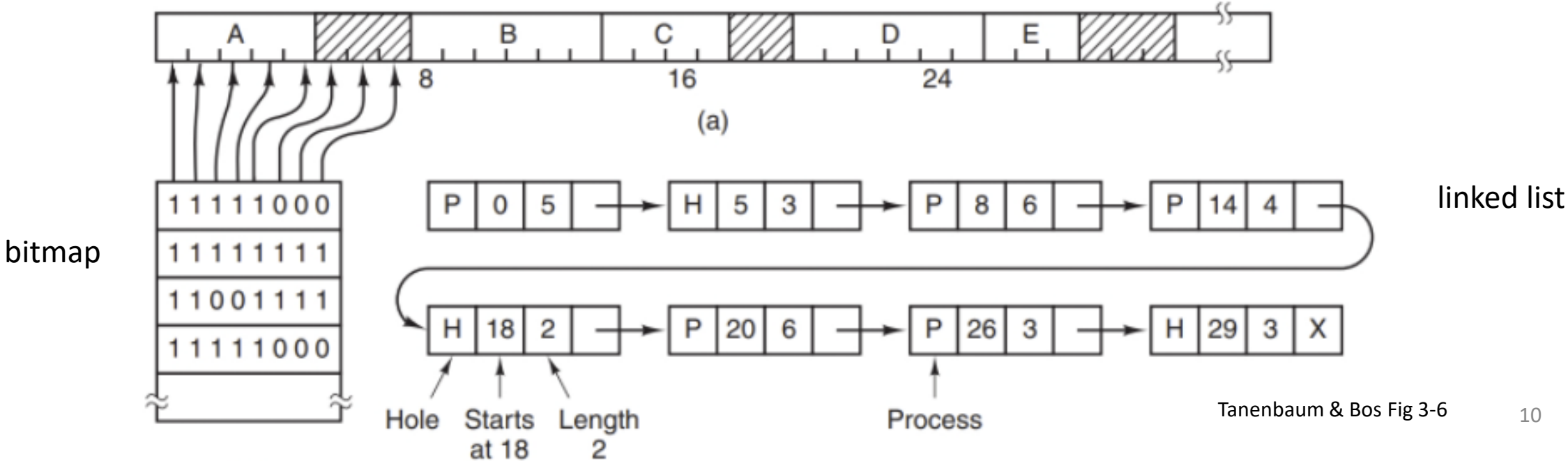
Need something to translate from logical to physical



- Base limit registers provided an early mechanism to do this
- Hardware performs translation and makes sure that logical address does not exceed the limit.
- Memory management units are a more sophisticated way to do this…

Contiguous memory management

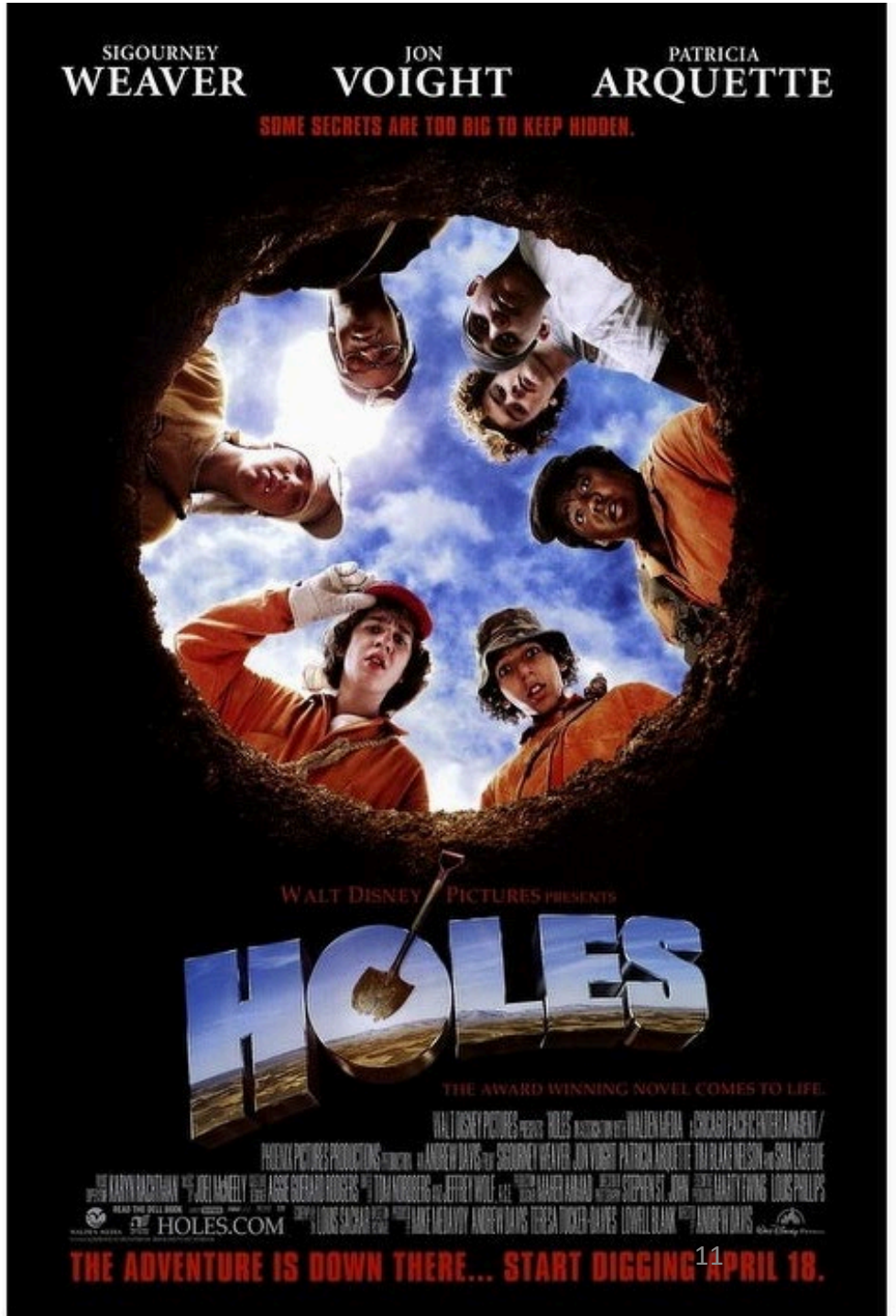
- With simple (or no) memory abstraction, we must allocate contiguous memory for processes.
- Common strategy: Chunk into memory blocks of X bytes
- Data structures manage memory, two common structures:



Tanenbaum & Bos Fig 3-6

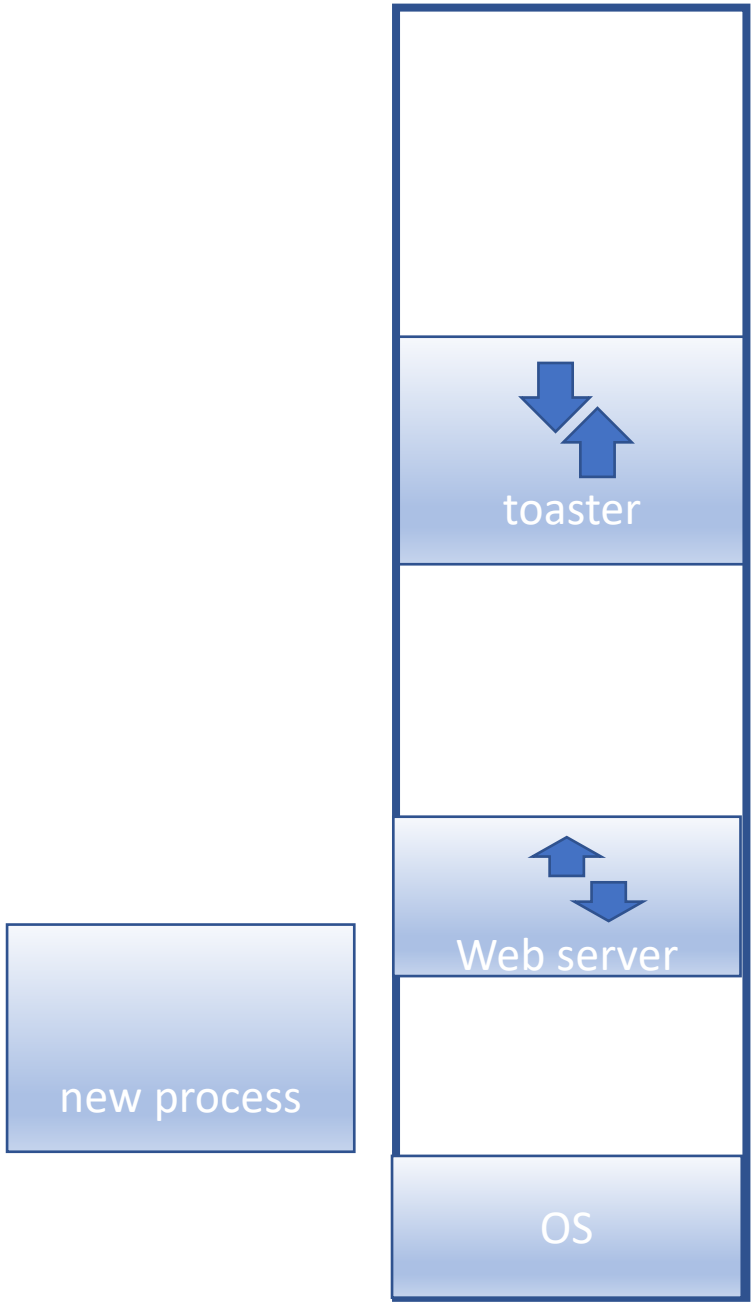
Contiguous memory management

- Searching for holes
  - Bitmap search can be slow, bits can straddle word boundaries
  - Complexity of search in either data structure?
  - There are different strategies for finding the right sized hole



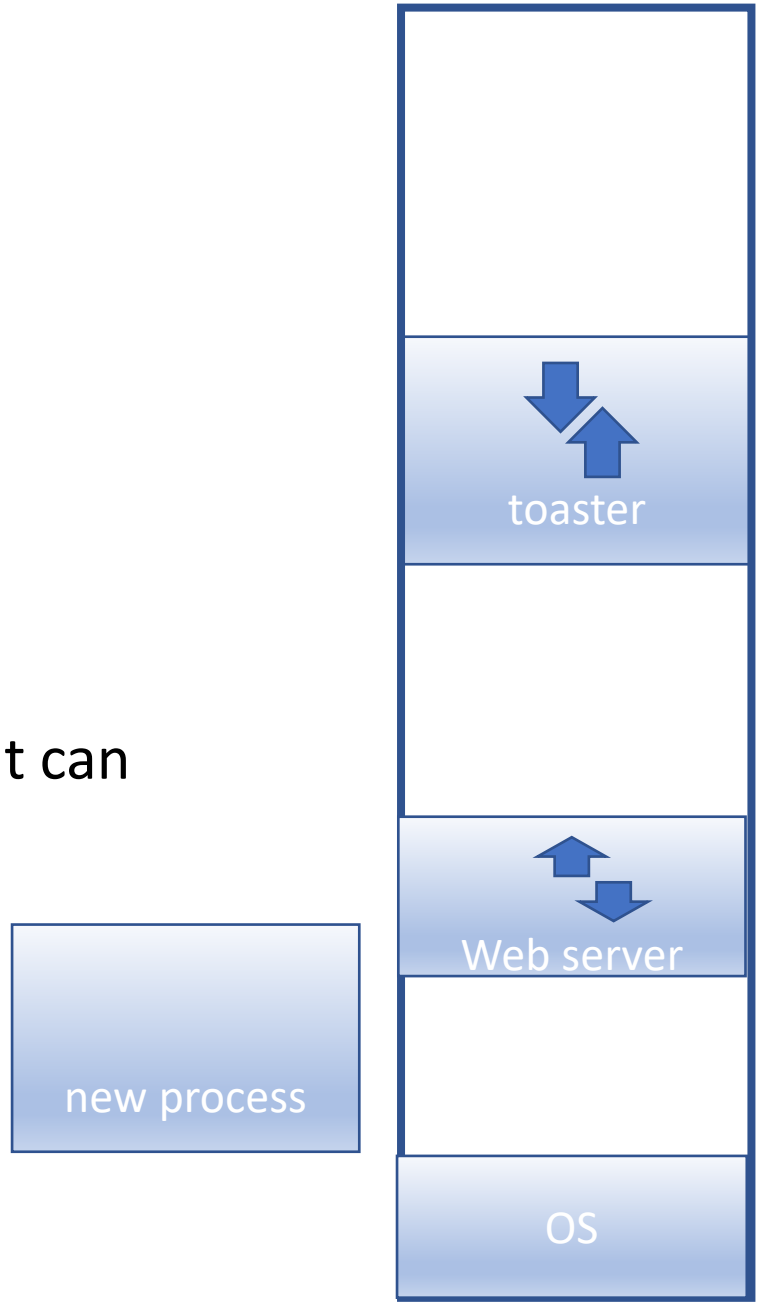
Finding holes

- First fit
- Next fit
- Best fit
- Worst fit
- Quick fit



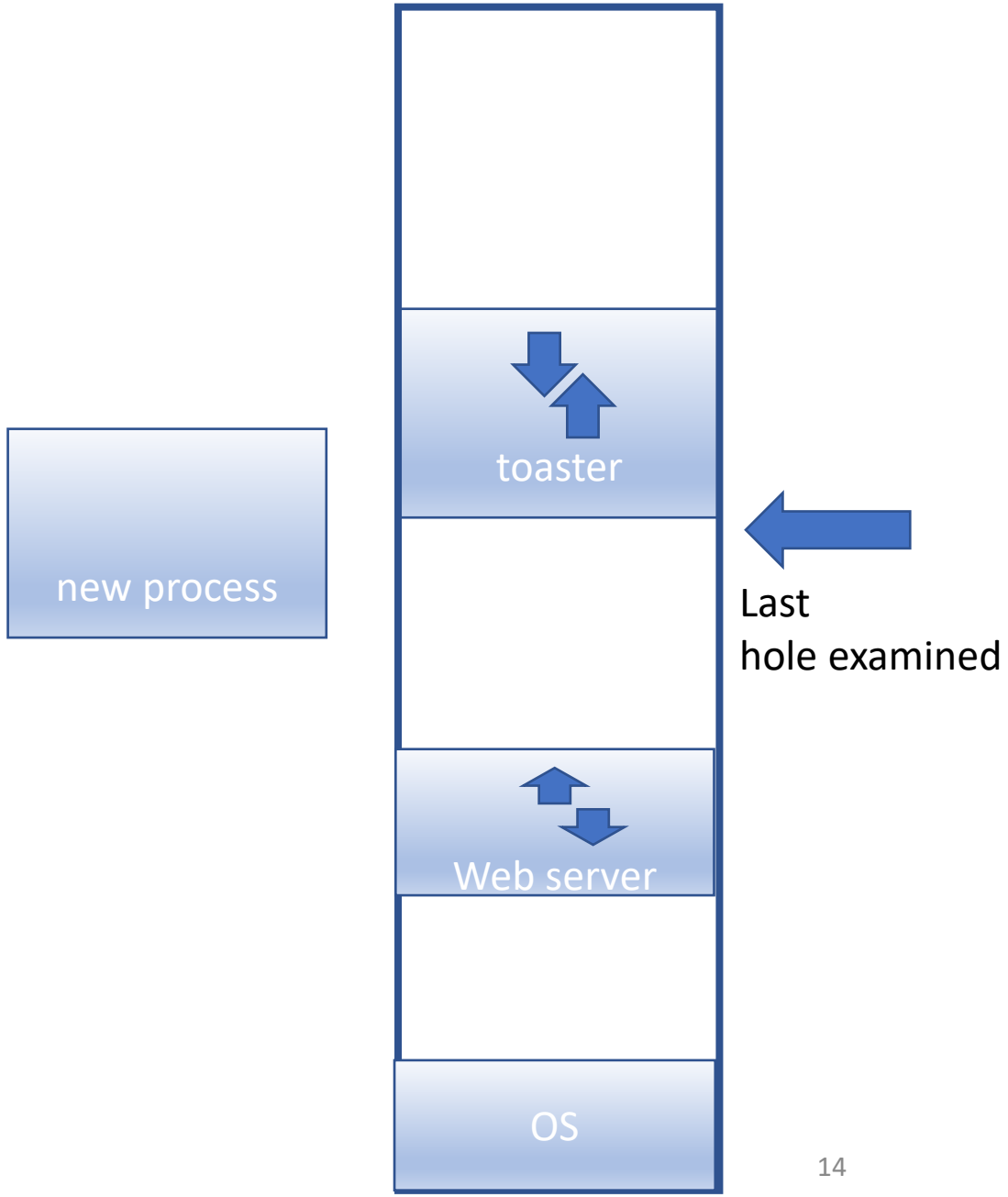
Finding holes

- First fit
  - Find the first hole that fits the process
  - Tends to leave small holes over time (External or internal fragmentation?)
- Best fit
  - Search the hole list for the smallest hole that can hold the process.
  - Slower
  - Leaves small holes quickly
  - In this example, same as first fit, but certainly not always.



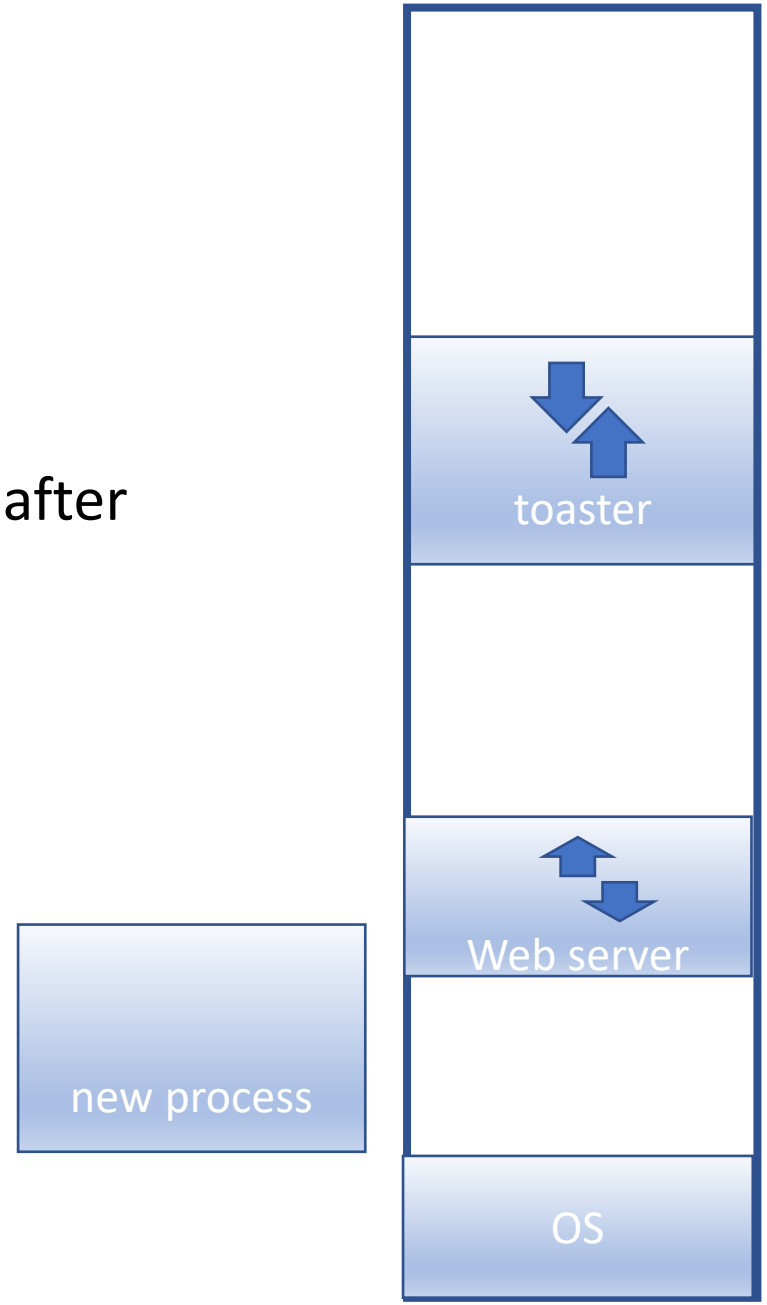
Finding holes

- Next fit – Like first fit, but remembers where we stopped searching the last time we allocated.



Finding holes

- Worst fit
  - Allocate from the largest hole.
  - Tends to make it difficult to find large holes after a while.

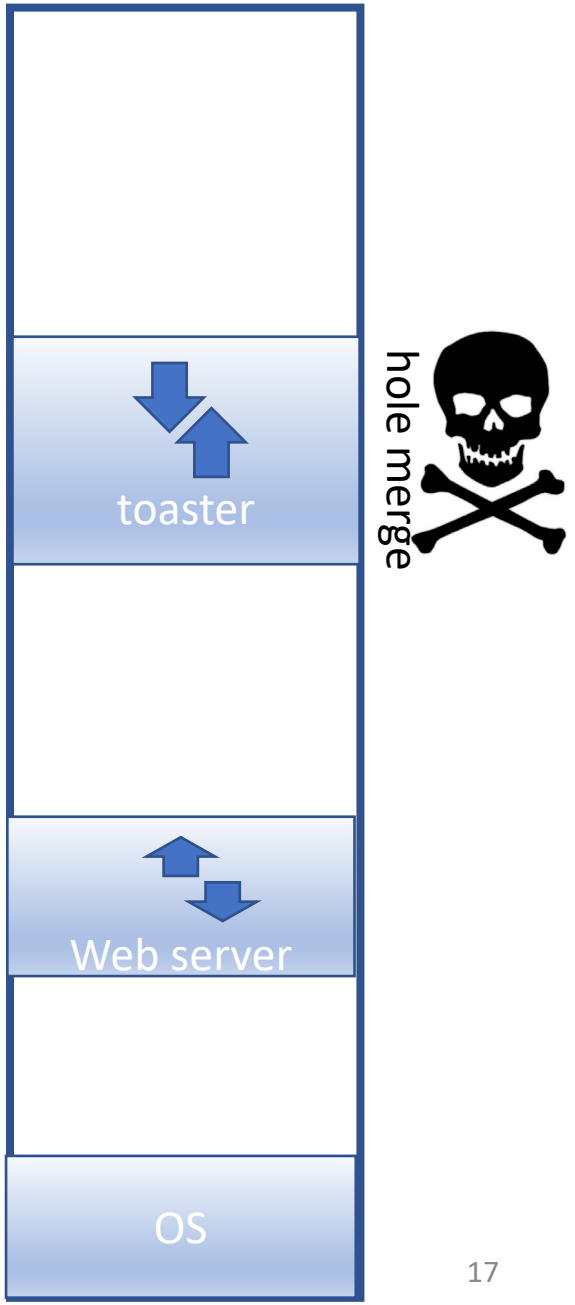


Finding holes

- Quick fit
  - Keeps separate lists for common size holes
  - Search the appropriate list
  - Odd sized holes kept in
    - separate list, or
    - next smallest sized list
      example: 1.2 MB hole
      hole lists for 1 MB and 4 MB, place in 1 MB list

Holes and terminating processes

- Upon termination, process is returned to the hole list
- Easy in a bitmap, harder in linked lists
  - Need to merge previous & subsequent holes
  - More difficult when allocated and hole lists kept separately
  - Even harder in quick fit



Holes and terminating processes

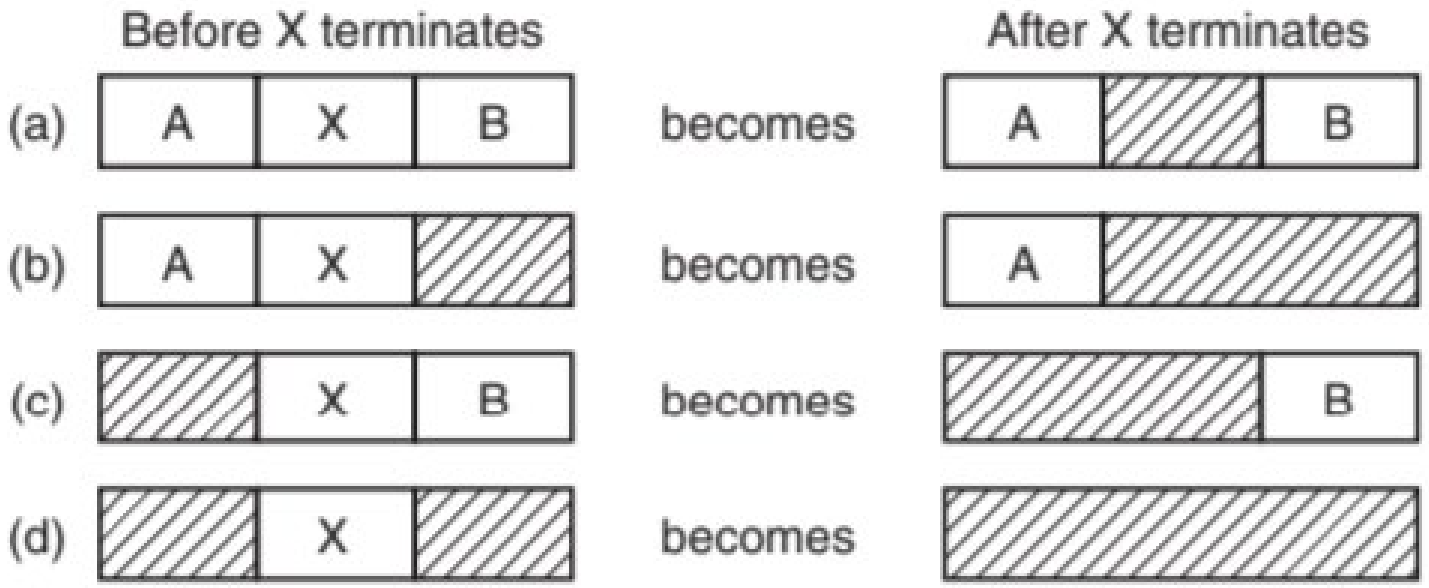
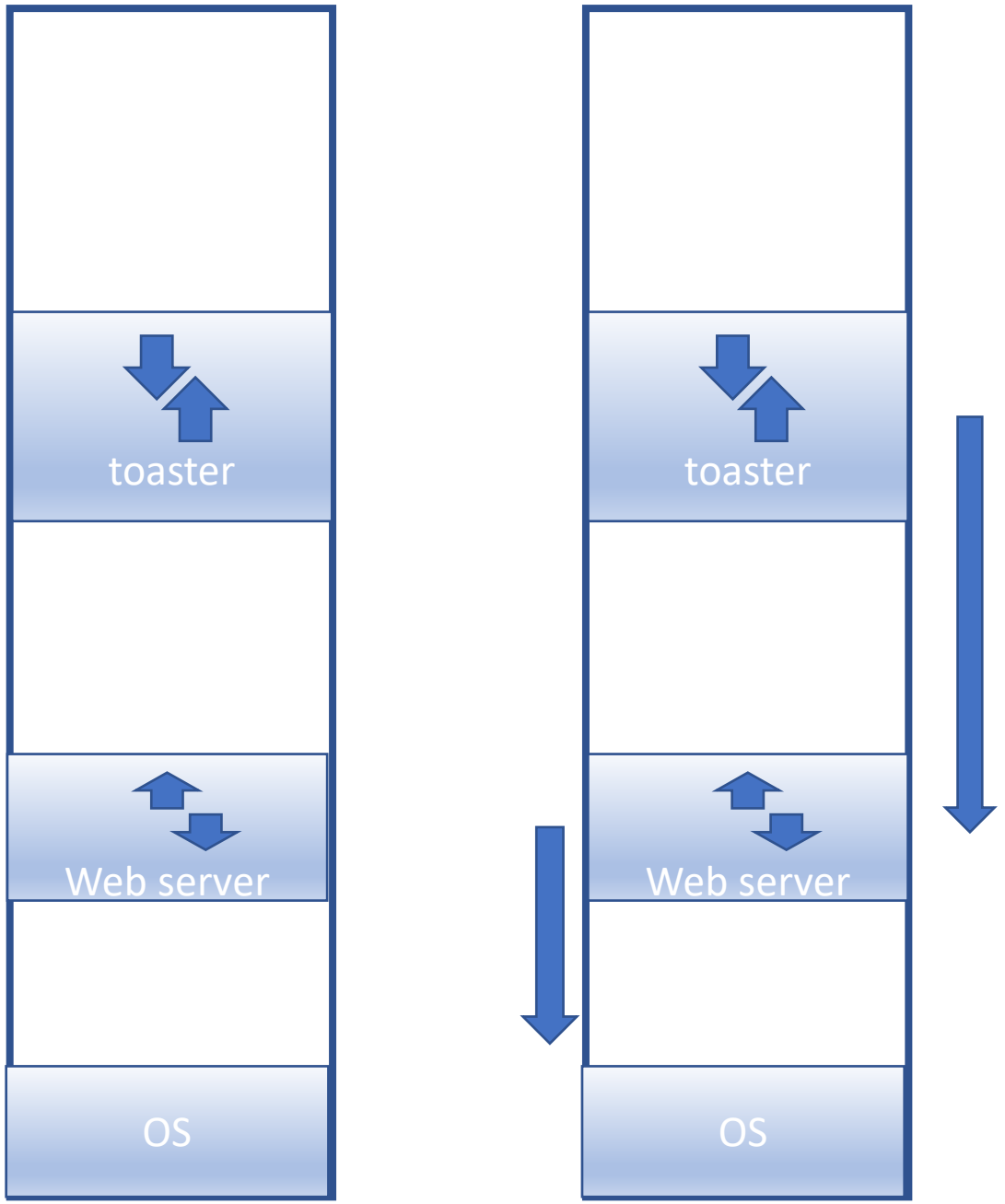


Figure 3-7. Four neighbor combinations for the terminating process, *X*.

Compaction

- Many small holes can result in failure to allocate processes.
- With runtime bindings (e.g. base/limit registers), we can move processes closer together
- Creates large hole
- Expensive in time



Virtual memory

- Logical address space paradigm (John Fotheringham 1961)
- Each process has:
  - A logical address space partitioned into fix-sized pages
  - Physical memory partitioned into frames of the same size as pages
  - Pages are mapped onto physical memory

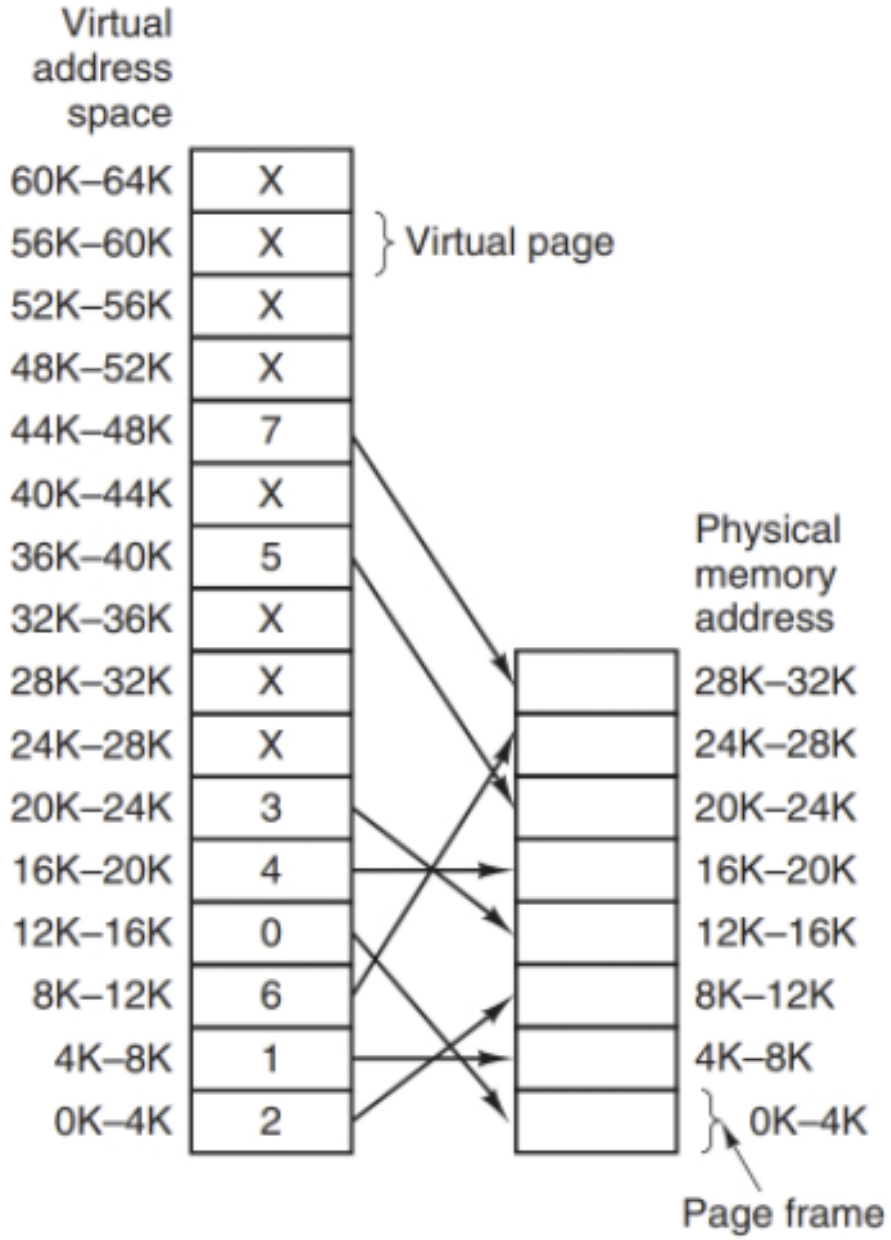
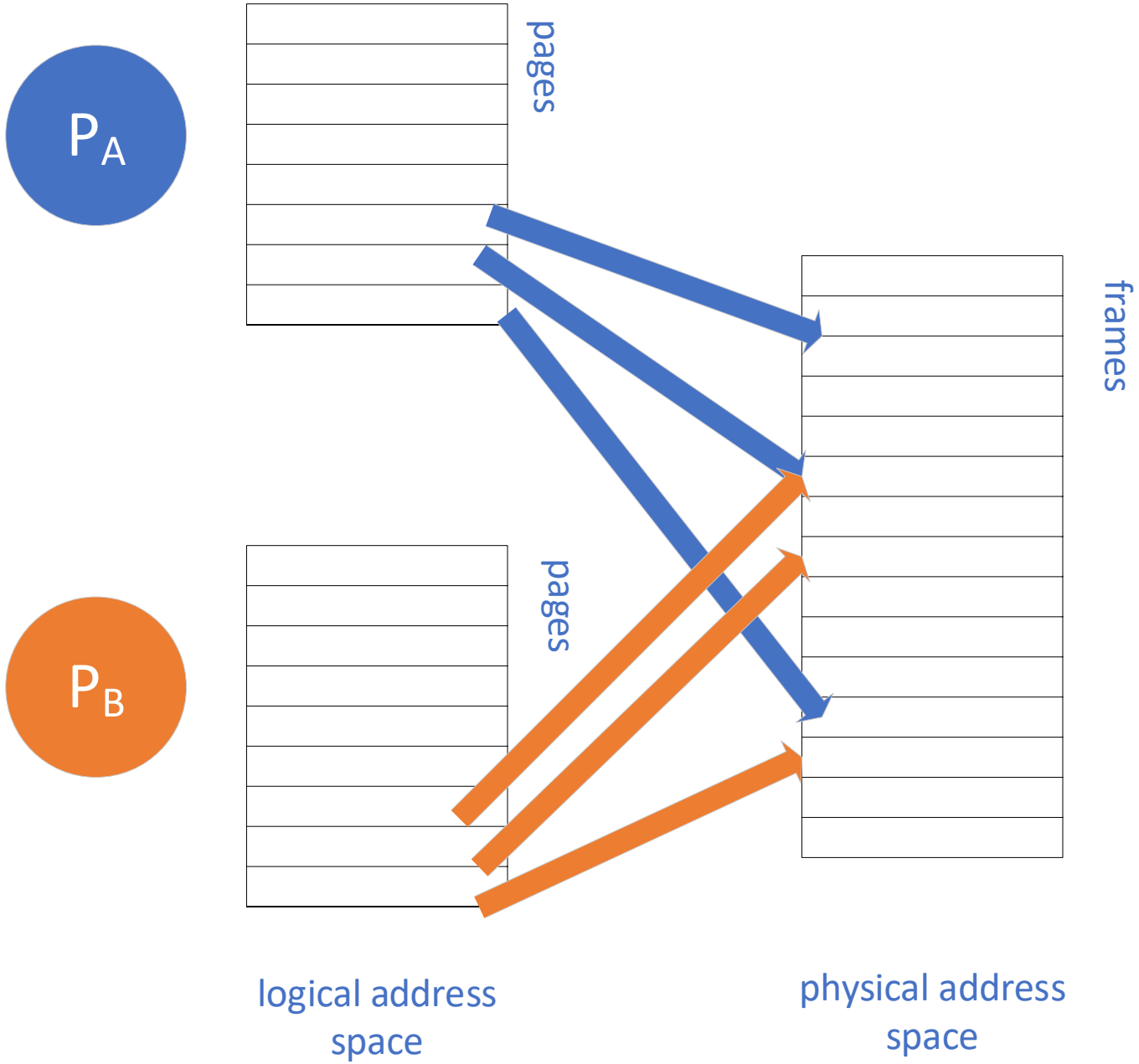


Fig. 3-9 Tanenbaum & Bos

Virtual memory

- Memory no longer needs to be contiguous.
- We can even have pages point to the same frame. (Why would we want to do this?)
- The map from logical to physical is called a page table



Logical address >> Physical memory

- x64 architecture uses 48 bits of address space → 256 tebibytes (TiB)
- Much larger than physical RAM capacity
  (tops out around 256 MB in early 2020s consumer motherboards)

- Most processes only use a small amount of logical address space

- How do we know which pages are unused?
  We introduce a presence/absence bit to the page table

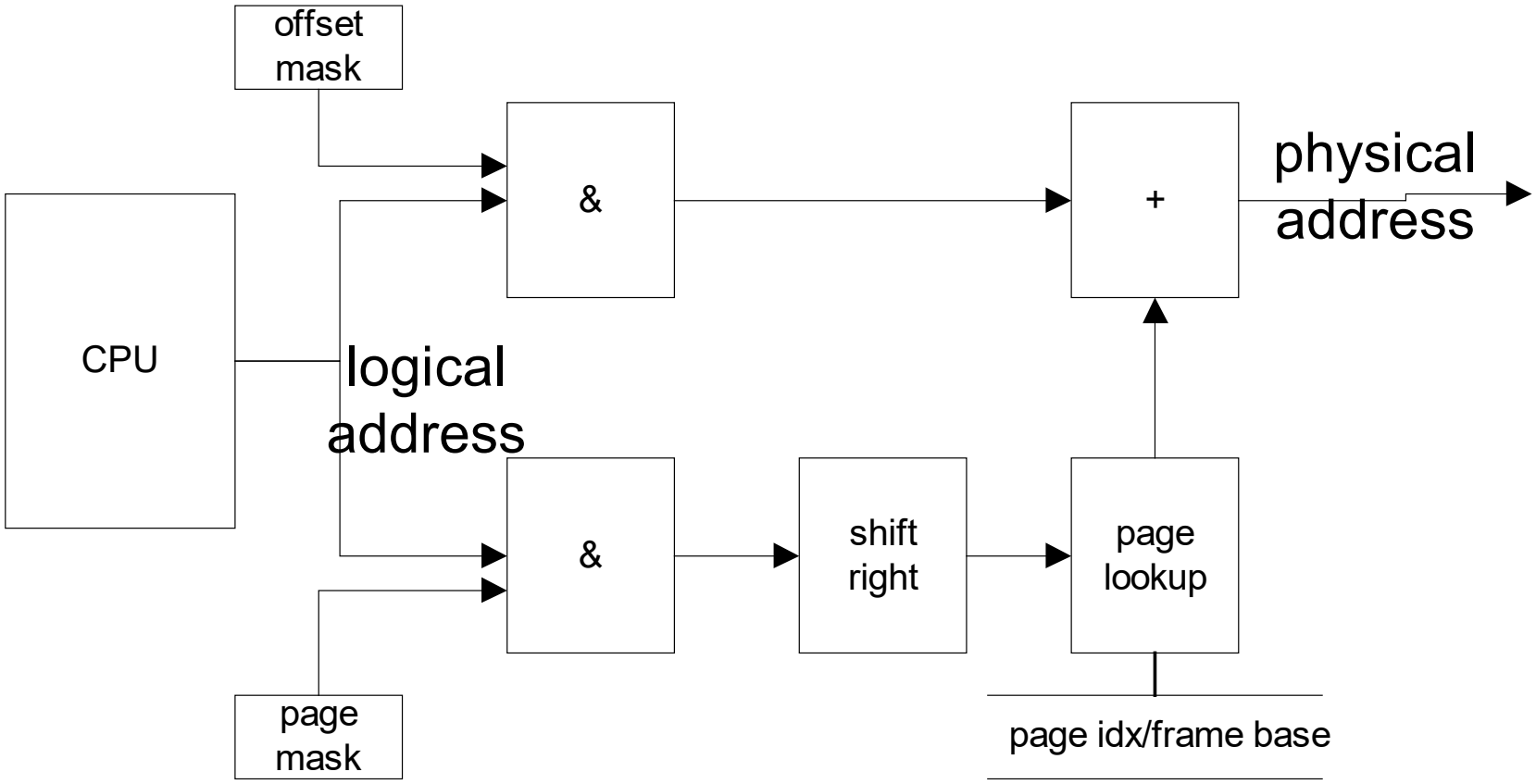
Page table

- Has one entry per page
- Indexed by the high order bits of the logical address.
  - Example: page size is 4096 bytes = 2^12
  - Consider 0x5823C4 in a 32 bit logical address space:

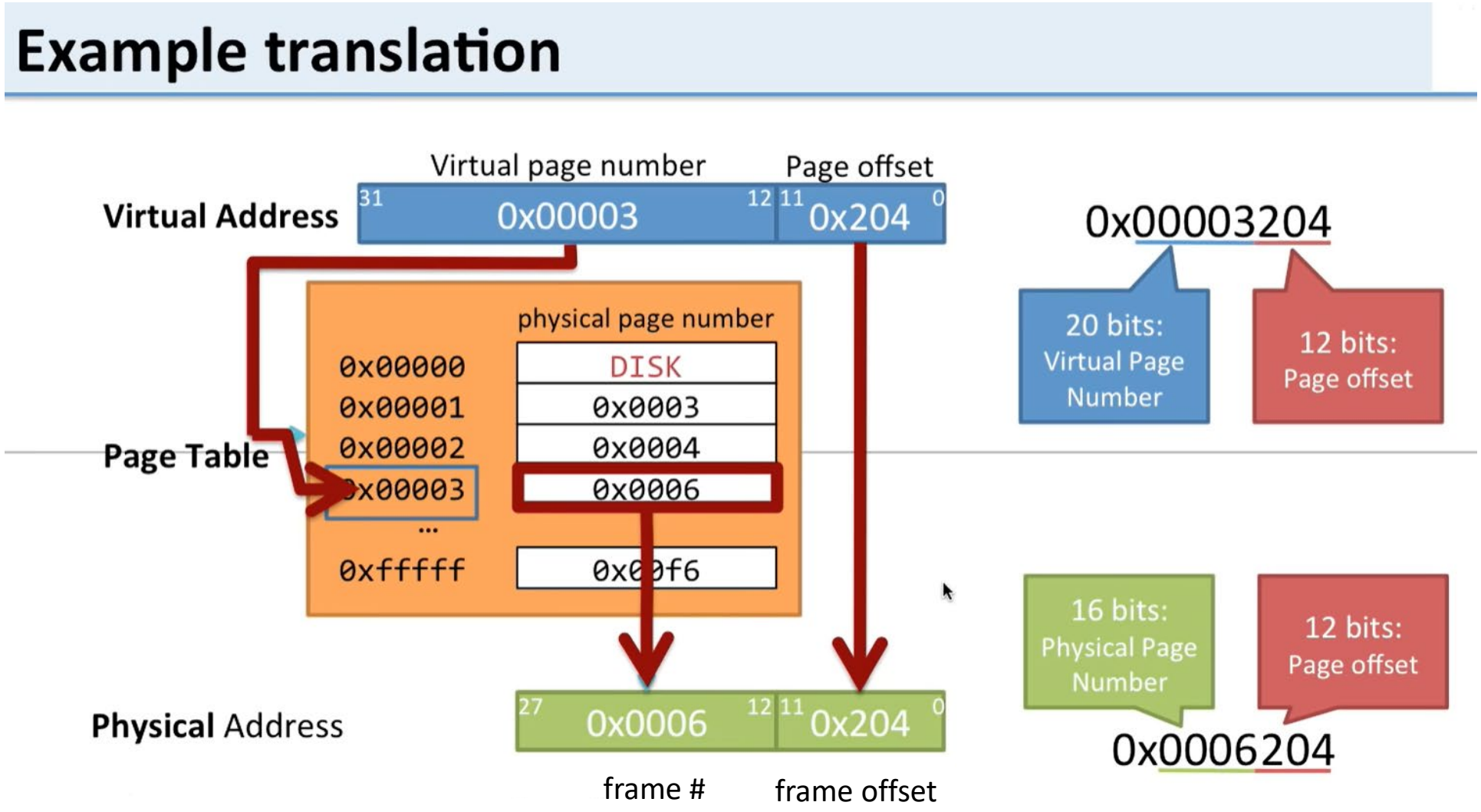
31-28	27-24	23-20	19-15	15-12	11-8	7-4	3-0
0	0	5	8	2	3	C	4
0000	0000	0101	1000	0010	0011	1100	0100
page					offset		

  - We are on page 0x00582 with an offset of 0x3C4

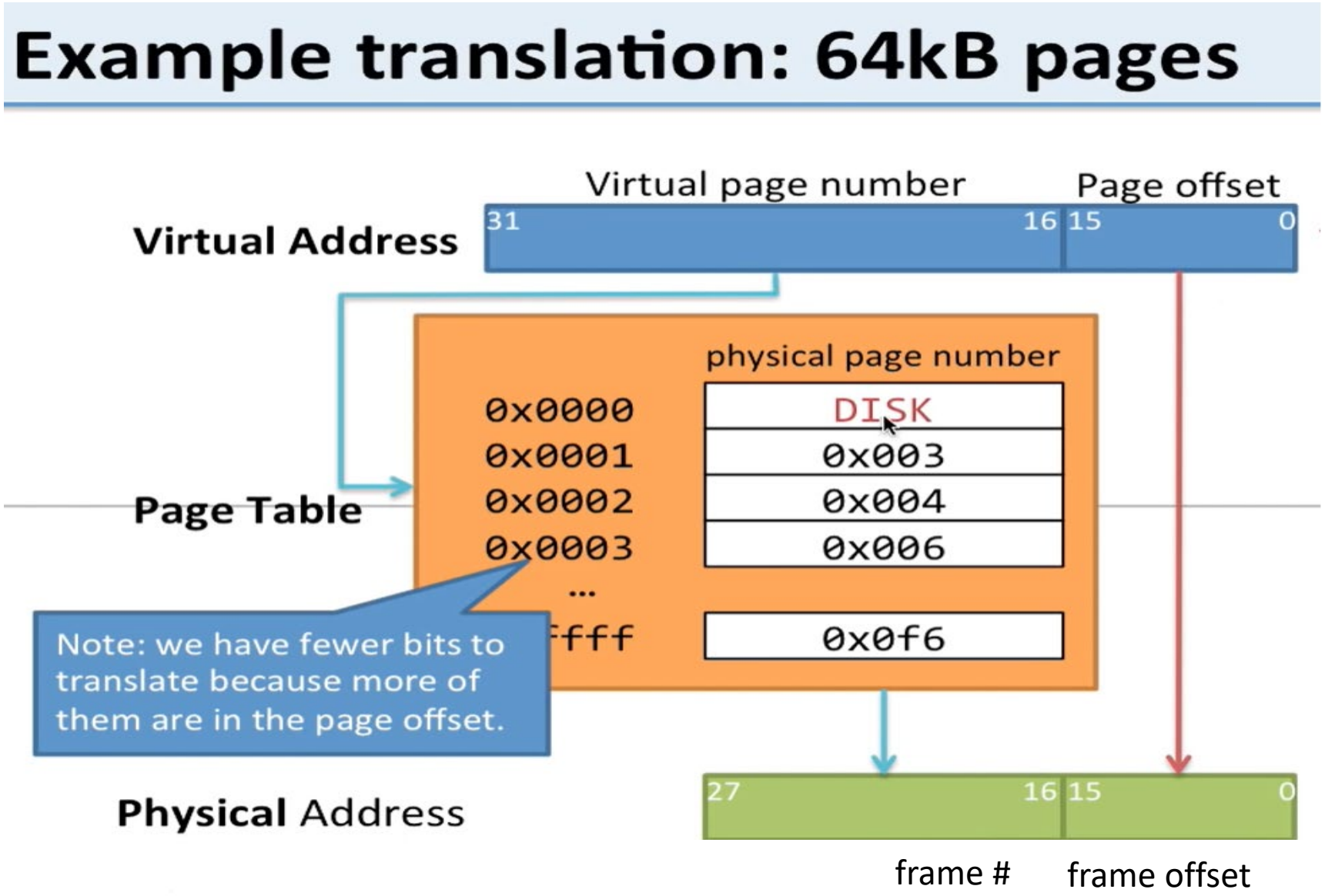
Memory management unit



Translation Example



Translation Example



Bit Masking and shifting

How is the MMU extracting the page number?

	31-28	27-24	23-20	19-15	15-12	11-8	7-4	3-0
	0	0	5	8	2	3	C	4
	0000	0000	0101	1000	0010	0011	1100	0100
mask	1111	1111	1111	1111	1111	0000	0000	0000
bitwise and	0000	0000	0101	1000	0010	0000	0000	0000
	page					offset		

shift: 0000 0000 0101 1000 0010 0000 0000 0000 >> 12

= 0000 0000 0101 1000 0010 = 0x00582

Page table structure

- Fields are architecture dependent
- Typical:

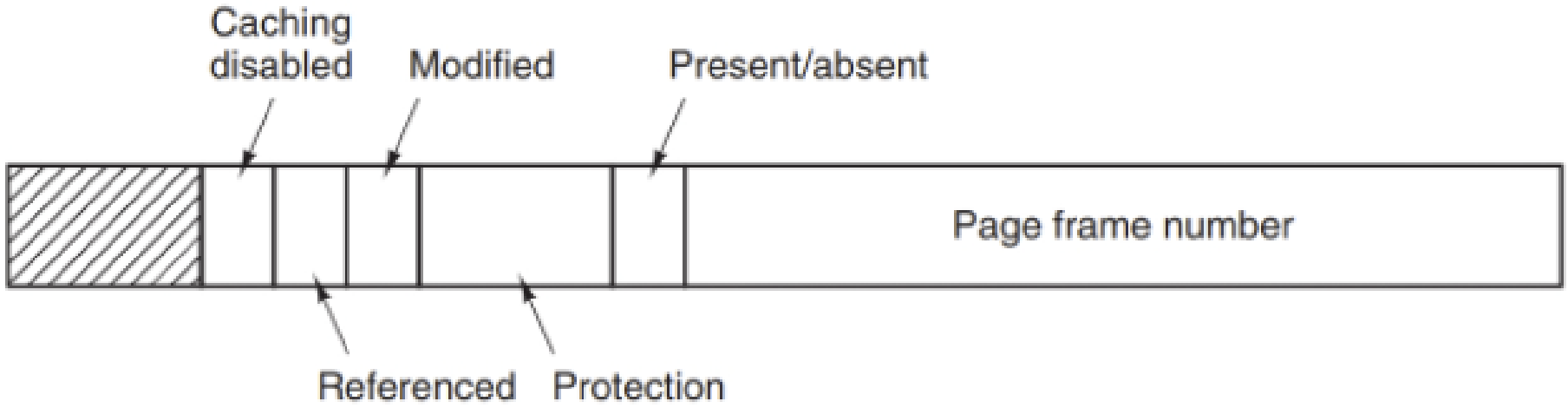


Fig. 3-11 Tanenbaum & Bos

Page table fields

- Present/absent (aka valid) – Is there a mapped frame?
- Frame number – Index of frame
  Question: 4K pages, frame 5, what is the first address of physical memory?
- Referenced – Has the page been accessed recently?
- Modified (aka dirty) – Set if page has been written to.
- Protection – Can the page be: read, written, executed?
- Cache disable – If set, page will never be stored in cache memory
  (Useful for memory mapped I/O)

Cost of paging

- Without paging: Read word

- With paging:
  - Read page table
  - Read word

- This doubles the access time for a word.
- Some page table designs we will study might triple or quadruple it.
- Page tables are too large to put in register memory.

Cost of paging example:

MOVE #27CA, (SP)- ; Push #27CA onto the stack

Memory accesses

- Fetch move instruction
- Fetch immediate data 0x27CA
- Write 0x27CA to top of stack

Involves 3 memory accesses; 6 with paging

Turbo-charging paging: translation lookaside buffer (TLB)

- Also known as address translation cache (ATC)
- Specialized cache
- Associative storage of page table entries organized by page #

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

Fig. 3-12 Tanenbaum & Bos

How TLB/ATC works

- Extract page number from logical address
- If page in cache (hit)
  then use page entry from TLB
  else (miss)
    read page table
    store entry in TLB
- Effective access time (EAT) is the average time to access memory

- EAT example:
  - access time: 100 ns
  - page table access time: 100 ns
  - hit rate: .98
  - TLB access time: 20 ns

$$hit \cdot (TLB + access) + (1 - hit) \cdot (TLB + pgaccess + access)$$

$$= .98(20 + 100) + .02(20 + 100 + 100) = 122\ ns$$

cost of paging: 22%

TLB/ATC cache management

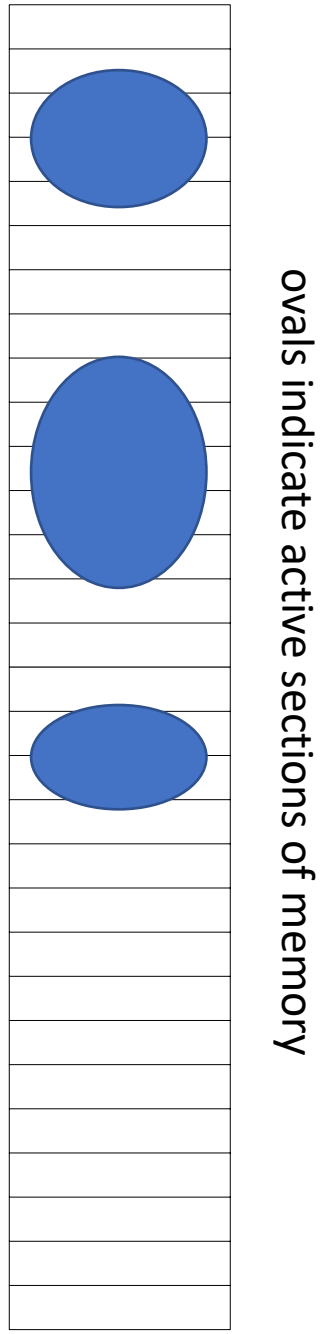
- Hardware
- Software
  - More common in reduced instruction set computers (RISC machines)
  - TLB miss generates TLB exception
  - TLB service routine
    - Reads page tables using software (implies page table design up to OS)
    - Selects TLB entry to remove and updates it with new page table entry
    - Providing a modest-sized TLB buffer (e.g. 64 entries) usually makes this efficient

TLB/ATC efficiency

- How can we get away with software cache management?
- Computer programs exhibit **locality of reference**

  Over brief periods of time, programs tend to access the same locations over and over again
  - loops, recursion, …
  - data structures such as stacks, trees, queues, …

  Can you think of a data structure that might provide poor locality of reference?



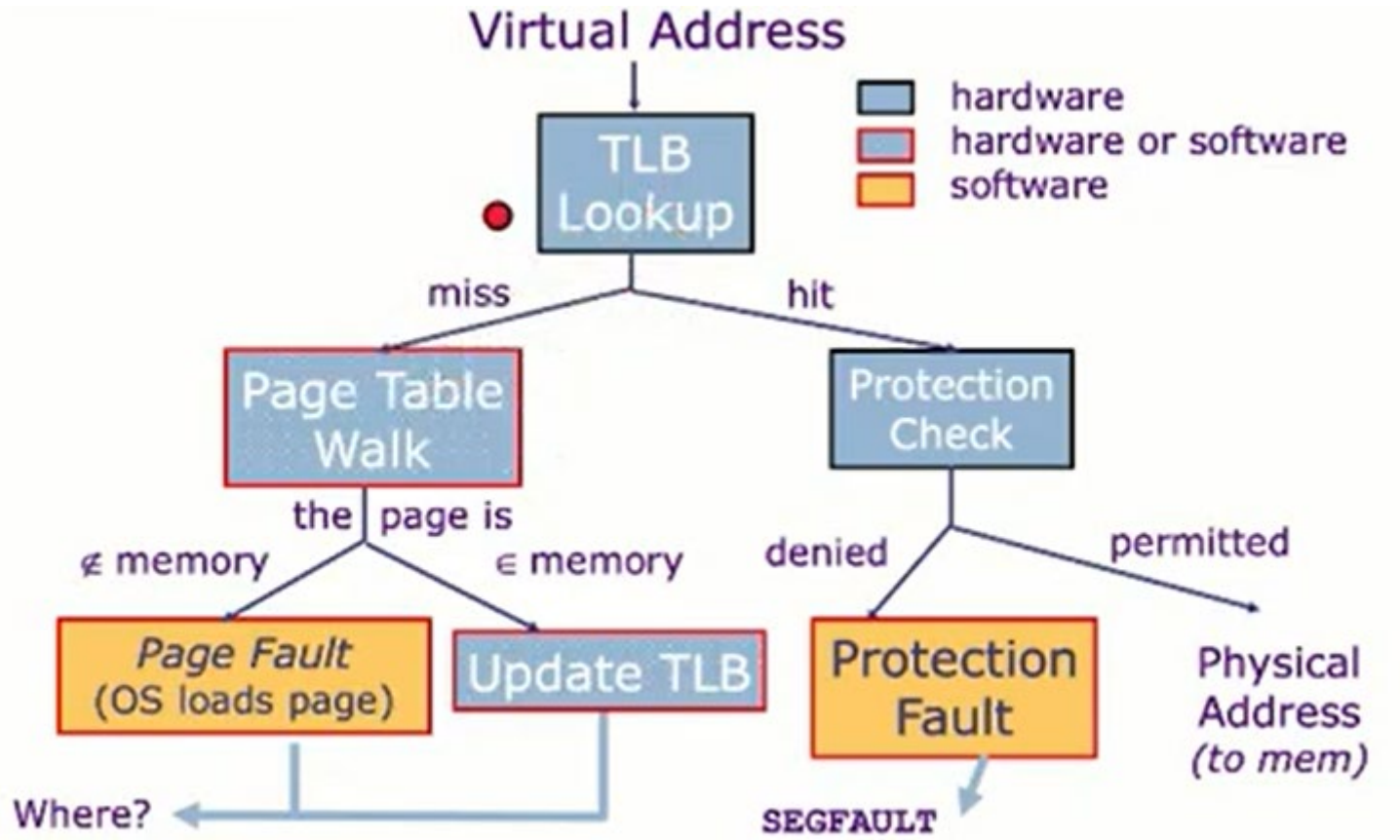
Overcommitting memory

- Up to now, we have assumed all processes fit in memory
- No longer the case…
  - We can take pages that have not been recently used and write them to swap (special data area on disk called the ***pagefile*** or **swap**)
  - The valid bit of these page table entries are set to 0
- When accessing memory, if the valid bit of the page entry is zero, a page fault exception is triggered

Page fault types

- **soft miss**
  - Page table entry not in TLB, but page is mapped to a frame
  - Requires a **page table walk** (page table lookup)
  - Pretty fast (~ couple ns), no disk needed
- **hard miss**
  - Page table entry is invalid. Requires disk I/O
  - Millions of times slower (measured in ms)

Logical to Physical Address Translation Flow



Page fault nuances

Hard page table faults can be complicated:

- Minor page fault
  - Page is mapped to a frame in another process
  - No disk I/O required
- Major page fault
  - Requires disk I/O
- Segmentation fault
  - Program accessed an illegal address
  - Send the program a signal which usually results in the process being killed

soundcloud.com

Page tables

- Simplest page table is a linear array of page table entries (e.g. array of struct)
- Problem: Modern architectures have large address spaces.
- Consider the number of 4096 byte pages in a 48 bit address space:

  $\frac{2^{48}}{2^{12}} = 2^{36}$ entries.

  If each page requires 4 bytes, we need

  $2^{36} \cdot 2^{2} = 2^{38}\text{bytes} \cdot \frac{GiB}{1024^{3} bytes} = 256\ GiB$

This number is larger than nearly all physical memories and is required for *each process*

Page tables

Two major strategies for reducing memory footprint

- Multilevel page tables
  - Tree of page tables
  - We do not need subtrees for unused areas of logical address space
- Inverted page tables
  - Physical memory centric page table

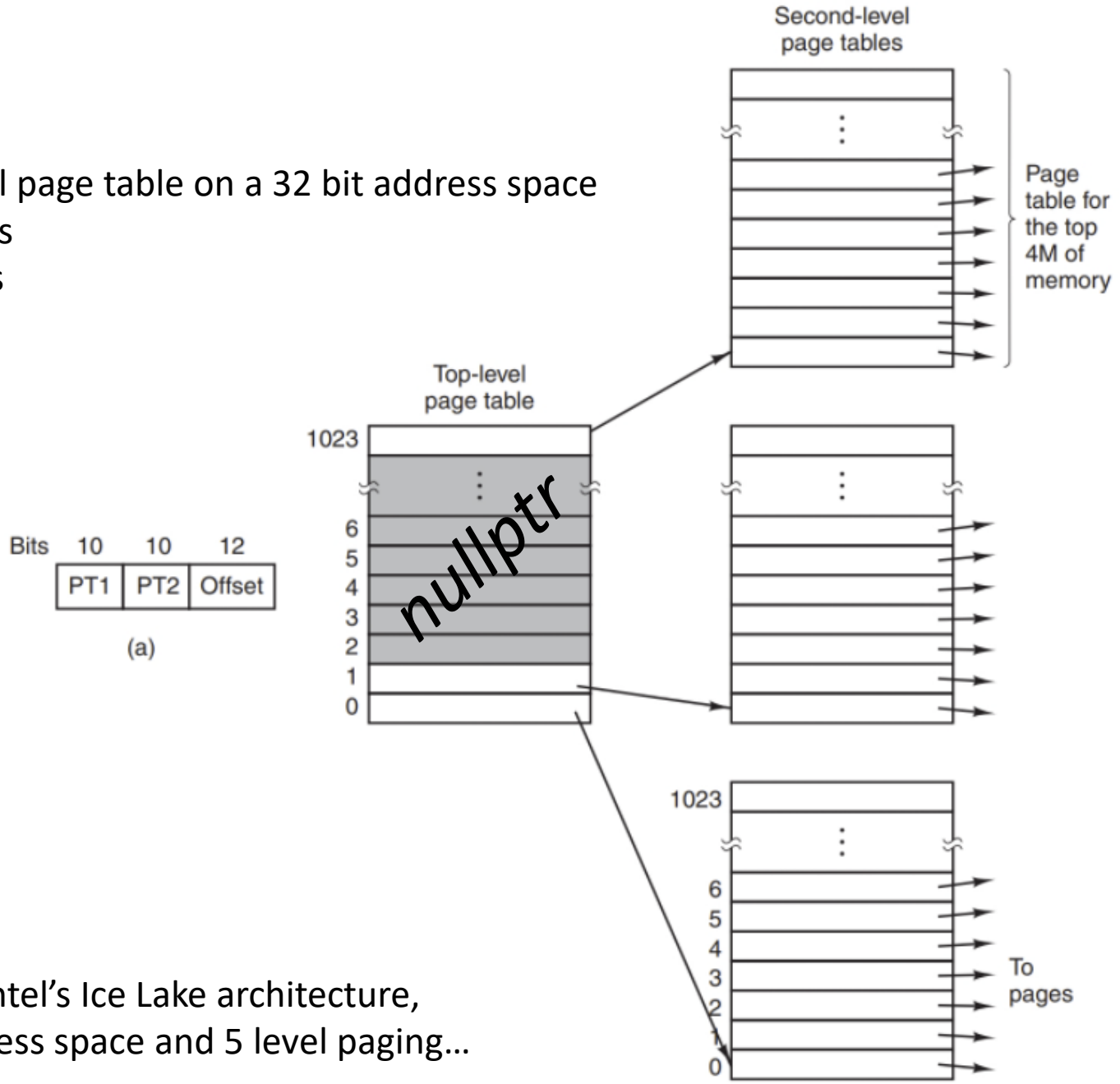
Mutlilevel page tables

- We add multiple page fields to the address
- Here we show two levels, but the number of levels is arbitrary
- Each level is defined by a number of bits that define the number of entries in the level

level 0		**level 1**			**offset**		
31-28	**27-24**	**23-20**	**19-15**	**15-12**	**11-8**	**7-4**	**3-0**
0	0	5	8	2	3	C	4
0000	0000	0101	1000	0010	0011	1100	0100

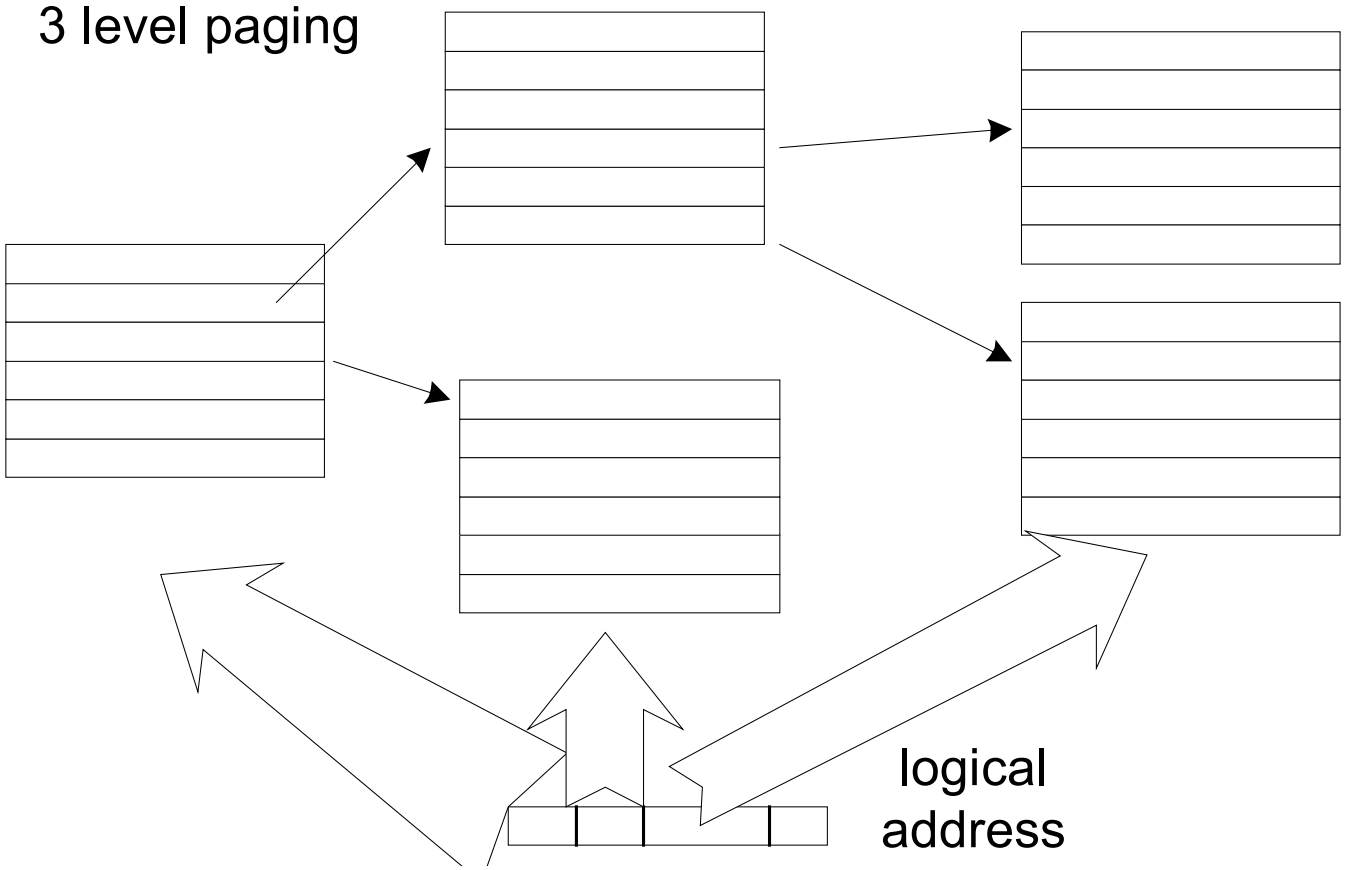
- Level 0: 8 bits → 2^8 = 256 entries, 0x00 – 0xFF
- Level 1: 12 bits → 2^{12} = 4,096 entries, 0x000 – 0xFFF

Sample 2-level page table on a 32 bit address space
level 0: 10 bits
level 1: 10 bits



Starting with Intel's Ice Lake architecture,
57 bits of address space and 5 level paging…

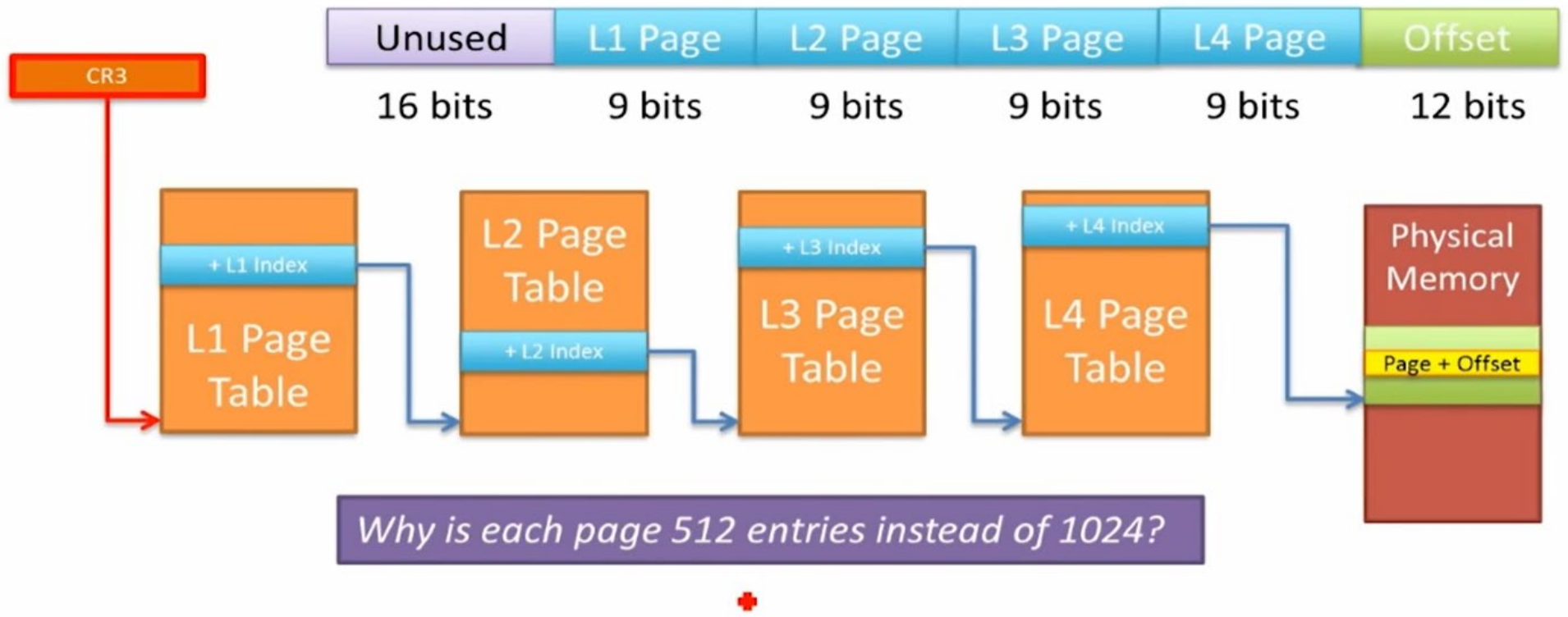
3 level paging example



Page table walks for multilevel page tables are only feasible with high TLB/ATC hit rates.

Multi-Level Paging example

- A multi-level paging setup in a 64-bit system with page size 4KB (Pentium Pro)
- How big is the virtual address space?
- 2^{48} = 256 terabytes



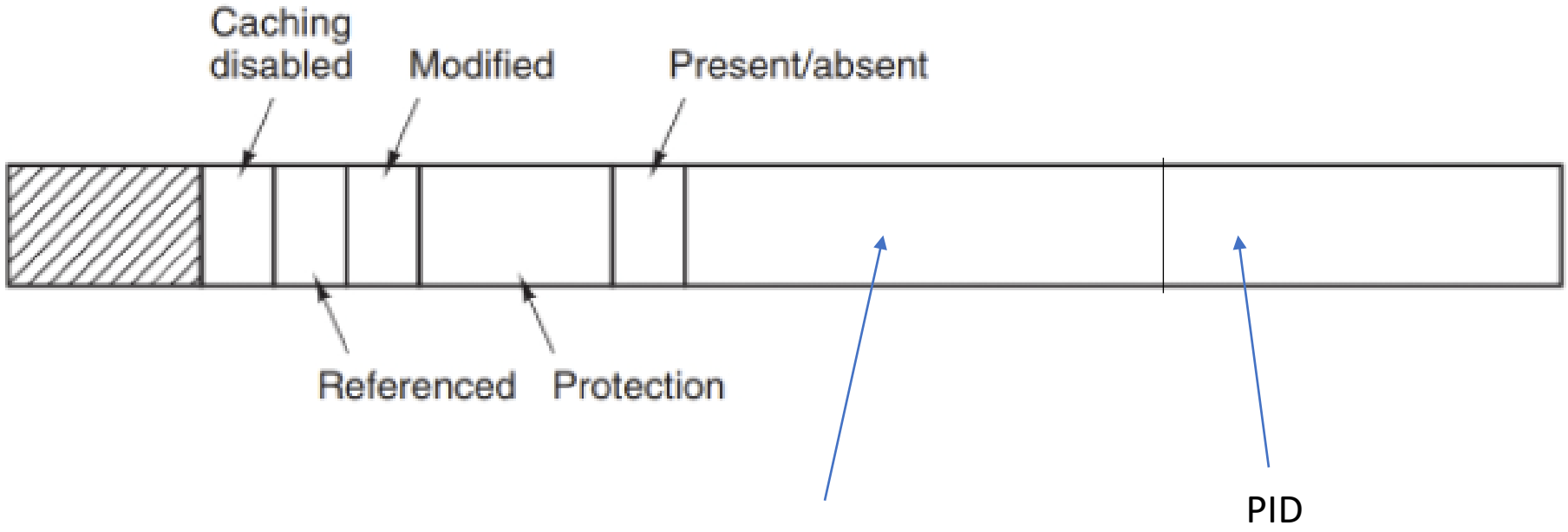
Inverted page tables

- Logical address space >> physical memory
- Why not flip the paradigm?

- Global page table instead of process specific page table
- One entry per frame
- Problem: Suppose we know that P_{99} needs to access page 0x2A? How do we find it?

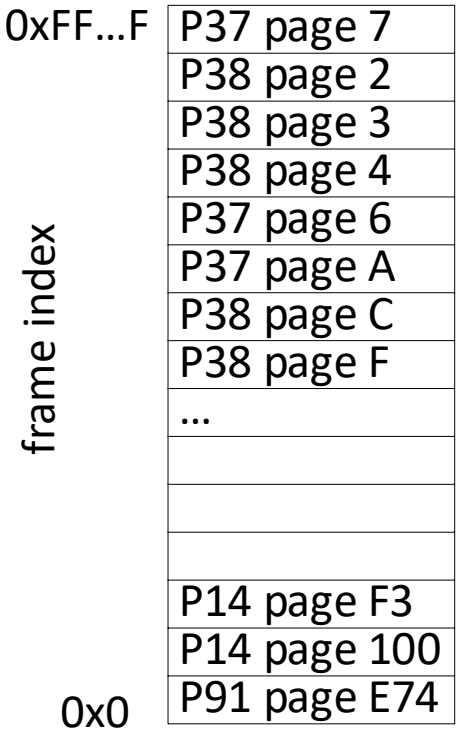
Inverted page tables

- Page table entry needs both page number and PID



Inverted page table speed

- TLB searches on PID and page number
  - Fast
  - Require associative memory on PID and page #
- For TLB miss, how can we avoid a linear search of the table?



PID/logical page
(other page table information not shown)

Inverted page table implementation

- Typically implemented as a hash table with key (PID, logical page)
- Hash collisions handled in the standard way (hash bucket lists)
- Hash values represent frame numbers
- Problematic for shared memory, but addressable

Inverted page tables

- Pros: Very small memory footprint

- Cons:
  - Logical to physical translation harder, but addressable with hash tables
  - Shared memory is trickier
  - Requires associative TLB memory with page number and PID

- A number of RISC architectures support this (e.g. PowerPC) and Intel's Itanium architecture which will stop shipping in 2021.

Paging terms

- Demand paging – Never load a page before it is needed, even when a program first starts.

- Locality of reference – Programs tend to access data in an organized way, only looking in certain regions within any given period.

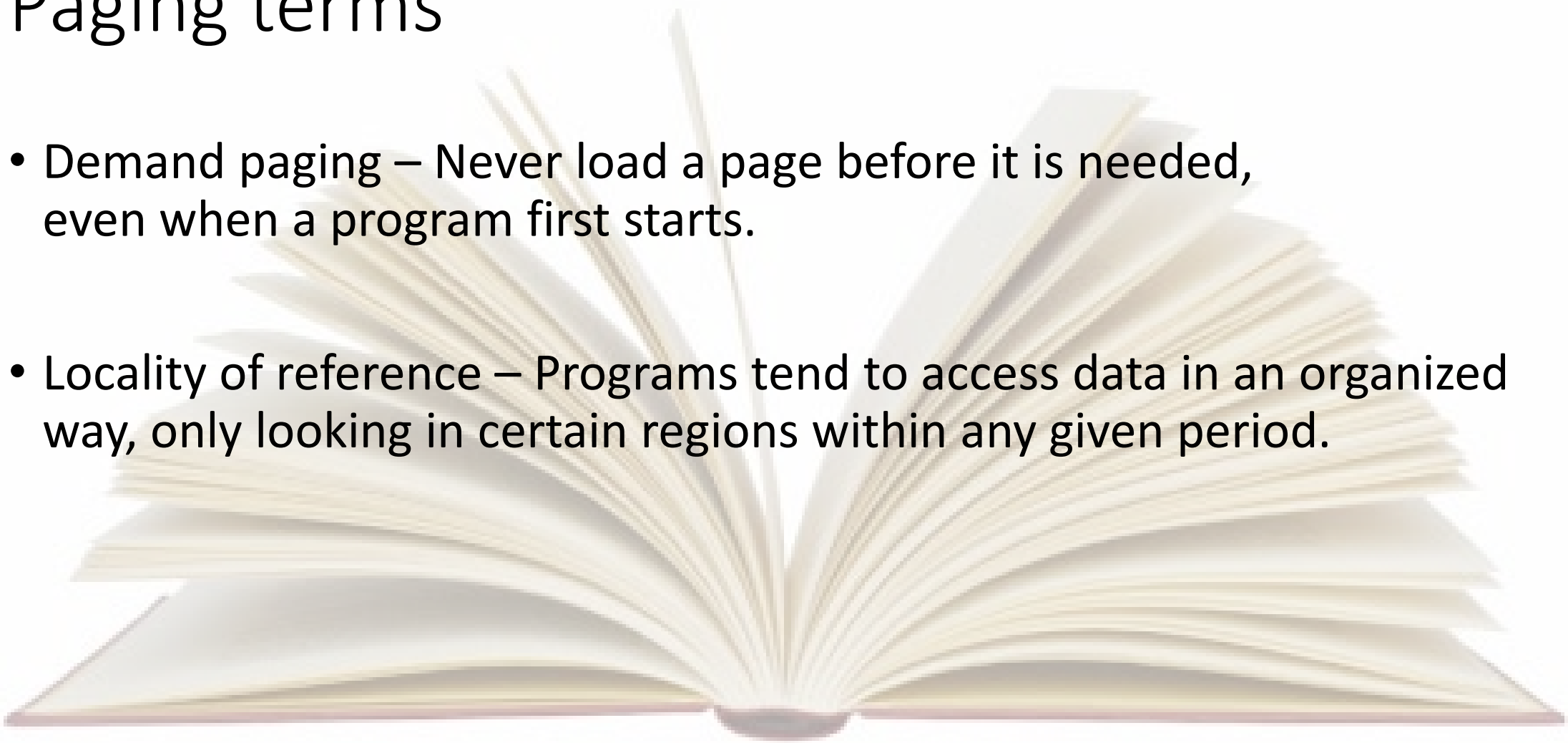


image: pngfind.com

Paging terms

- Thrashing – When a process has too few frames allocated to it, it will page fault frequently. When we spend the majority of our time servicing page faults instead of executing, this is called thrashing.

- Working set (of pages) – At any given time, the locality of reference will span a certain set of pages. If we can fit this set of pages in memory, we can execute efficiently without too many page faults and thus avoiding thrashing. The working set will change over time.

image: pngfind.com

Page replacement algorithms

- Suppose a page fault occurs and the page is on disk
- If a frame is available… we are scott free!

- There may not be a free frame to assign either because:
  - there are no more free frames OR
  - system policy does not let the process have more frames
- In this case, we need to pick a victim frame
  (rob Peter to pay Paul)

Victim frame selection algorithms

Optimal algorithm (per process)

- For each page associated with a frame, see when it will next be referenced
- Select the frame that will be referenced the farthest in the future as the victim
- Not practical, but good for simulation as a comparison

Victim frame selection strategies

- Referenced pages
  - Periodic daemon usually resets reference bits from time to time (e.g. 20 ms)
  - Reference bit present indicates this might not be a good frame to pick
- Modified pages
  - When a page has been modified, it must be saved to disk (might be needed later)
  - Expensive in time and will delay making the frame available to the page faulting process
- As managing these bits is expensive, it should be done in hardware (possible to simulate modified bit if necessary)

Not recently used

- Treats the modified (M) and reference (R) bits as a bit string:

MR	base 10	Interpretation
00	0	Not referenced, not modified
01	1	Not referenced, modified
10	2	Referenced, not modified
11	3	Referenced, modified

- On page fault, select page with lowest value, ties broken randomly.
- Adequate performance, but we can do better…

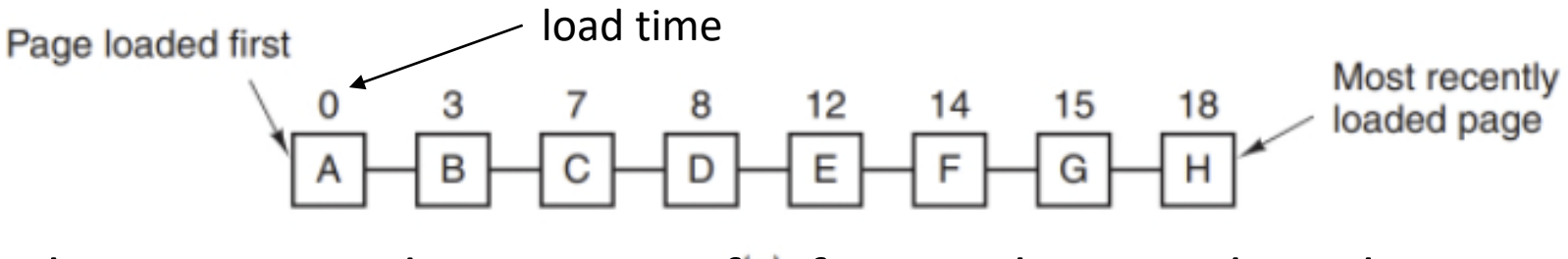
First-in, first-out (FIFO)

- Maintain a queue of pages
- Victim selected is always the oldest.

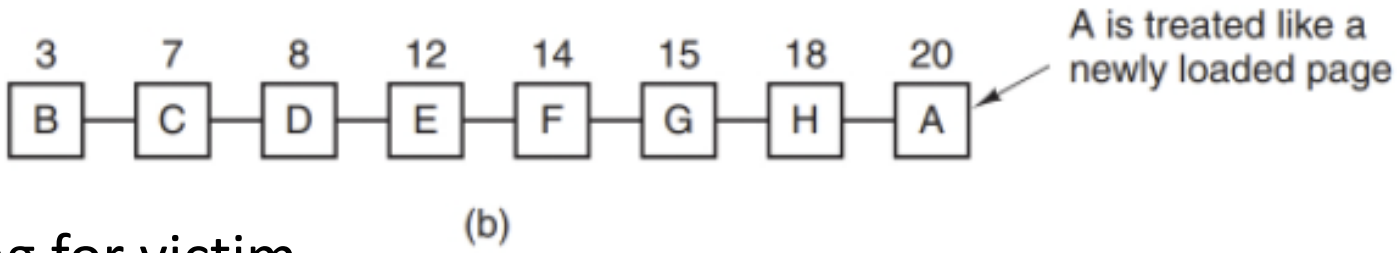
- Rarely used without some type of modification, why?

Second chance - FIFO modification

- Addresses problem of heavily used pages



- Page fault at time 20, dequeue A. If reference bit set, clear then enqueue



- Continue searching for victim
- Inefficient

Tanenbaum & Bos Fig. 3-15

Second Chance Clock page replacement

- More efficient second chance
- Pages organized as a ring (or clock)
- “Hand” points to next potential victim
- If not referenced
  - selected as victim and eject from clock
  - else clear reference and advance hand

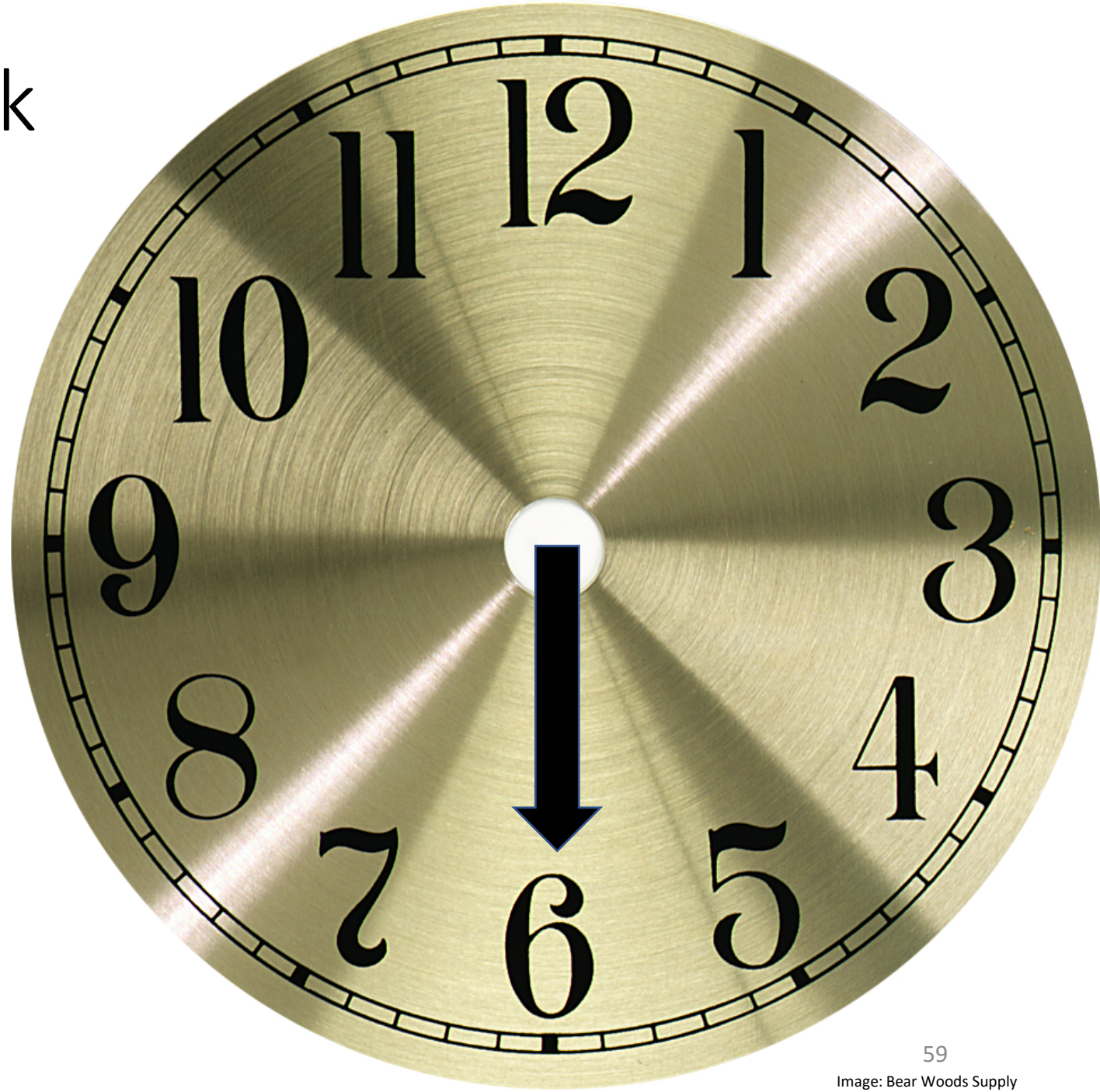


Image: Bear Woods Supply

Least recently used (LRU)

- Pages that are used a lot recently are likely to be reused
- Implies pages that have not been used in a while are the best ones to select as victims
- LRU algorithms are good approximators to the optimal algorithm

- With hardware support:
  - Page table entries have a counter field.
  - Running counter timestamps each reference
  - Find lowest numbered page entry

- Good luck finding a CPU that supports this… but we can simulate in software

cliptart.com


image: Marie Roch

Not frequently used (LRU approximation)

- Add a counter to the page table
- As pages are added, their counters are set to zero.
- When the reference bit service routine is run
  - If reference bit set, increment counter
  - Clear reference bit as usual

- On page faults, we select the page with the lowest counter
- Problem: Something might have been heavily used for a while, but not used very recently…
- Solution: Not frequently used with aging

NFU with aging

- Introduce a history bit string of N bits (e.g. N = 4) for each page
- When we service the reference bits,
  - Shift the existing count right by 1 bit:
    0101 >> 1 = 0010
  - If the reference bit is set, set the leftmost bit of the bit string:
    1 << 4 = 1000

    1000
    0010 bitwise or (|)
    1010
- When we page fault, select the lowest value (ties broken randomly)

NFU with aging example

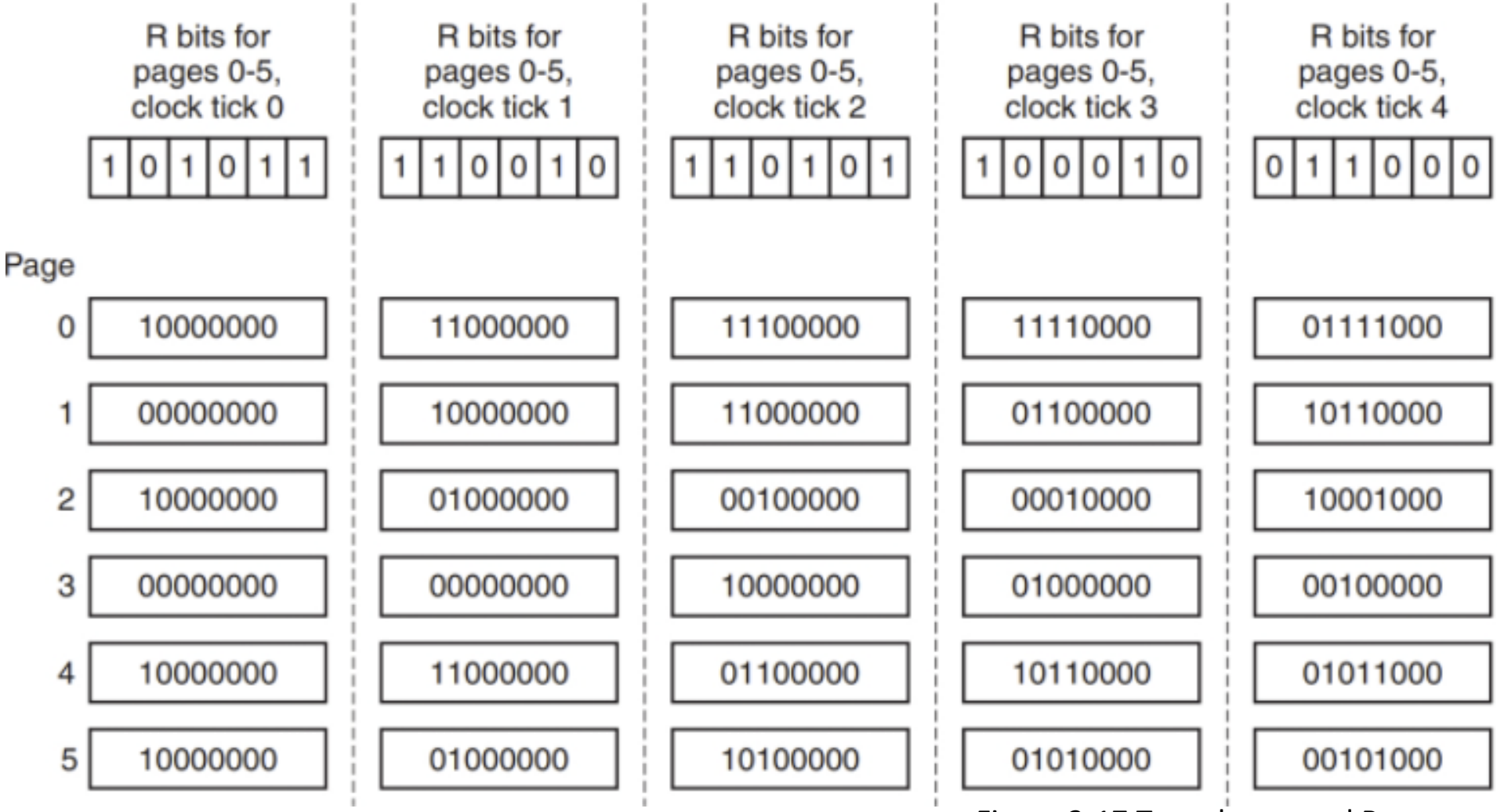


Figure 3-17 Tanenbaum and Bos

Working set algorithms

- Working set page replacement algorithms attempt to keep a processes' working set of pages in memory
- As we look over the last k references, the number of pages needed tends to plateau

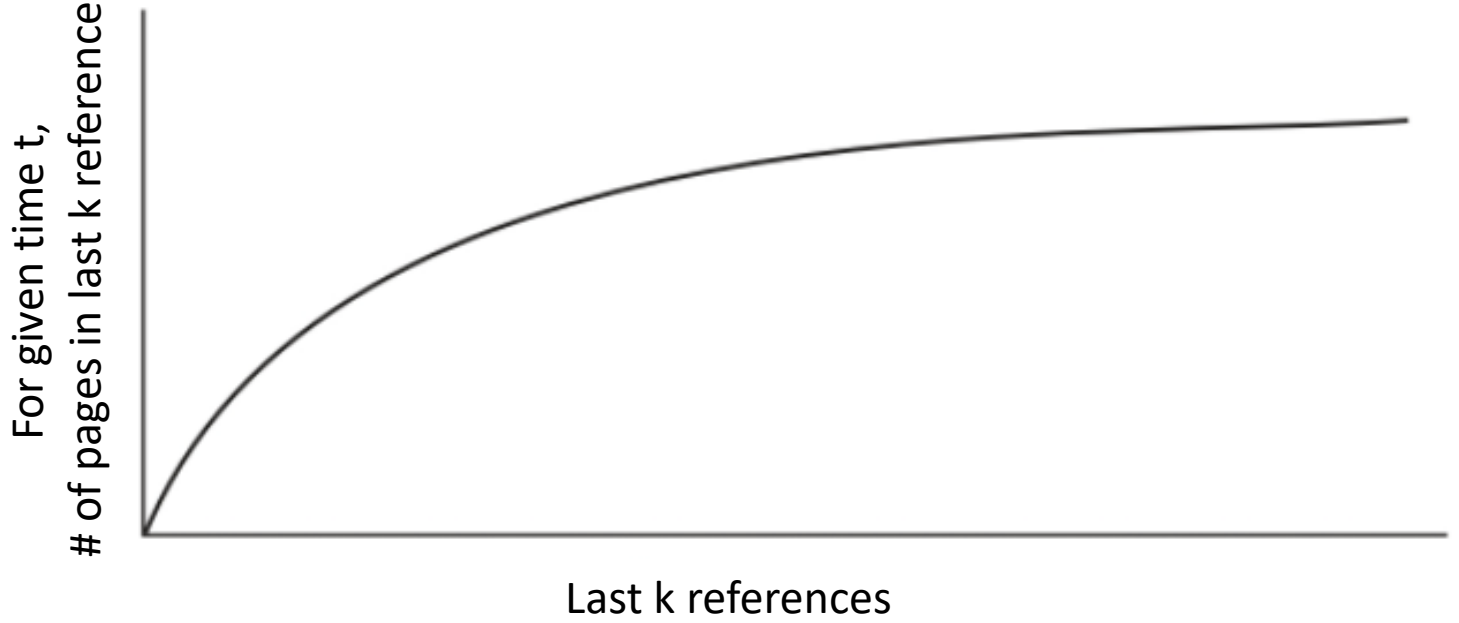


Fig. 3-18 Tanenbaum & Bos

Working set algorithms

- Difficult to measure number of previous references
- Like LRU, we turn to approximation algorithms
- We time stamp the page table entries
  - Similar idea to not frequently used with aging:
    - Explicit timestamp rather than relative
    - Updated with reference bit servicing
  - Timestamp is based on execution time, not wall clock time

Working set clock (WSClock)

- Has elements of the second chance clock combined with time stamps
- Each clock entry has:
  - time stamp indicating the amount of time the process executed when last accessed
  - reference bit
  - modified bit
- Reference bit processing
  - Copy reference bit into clock structure
  - Update page time if referenced
  - Clear reference bit

WSClock

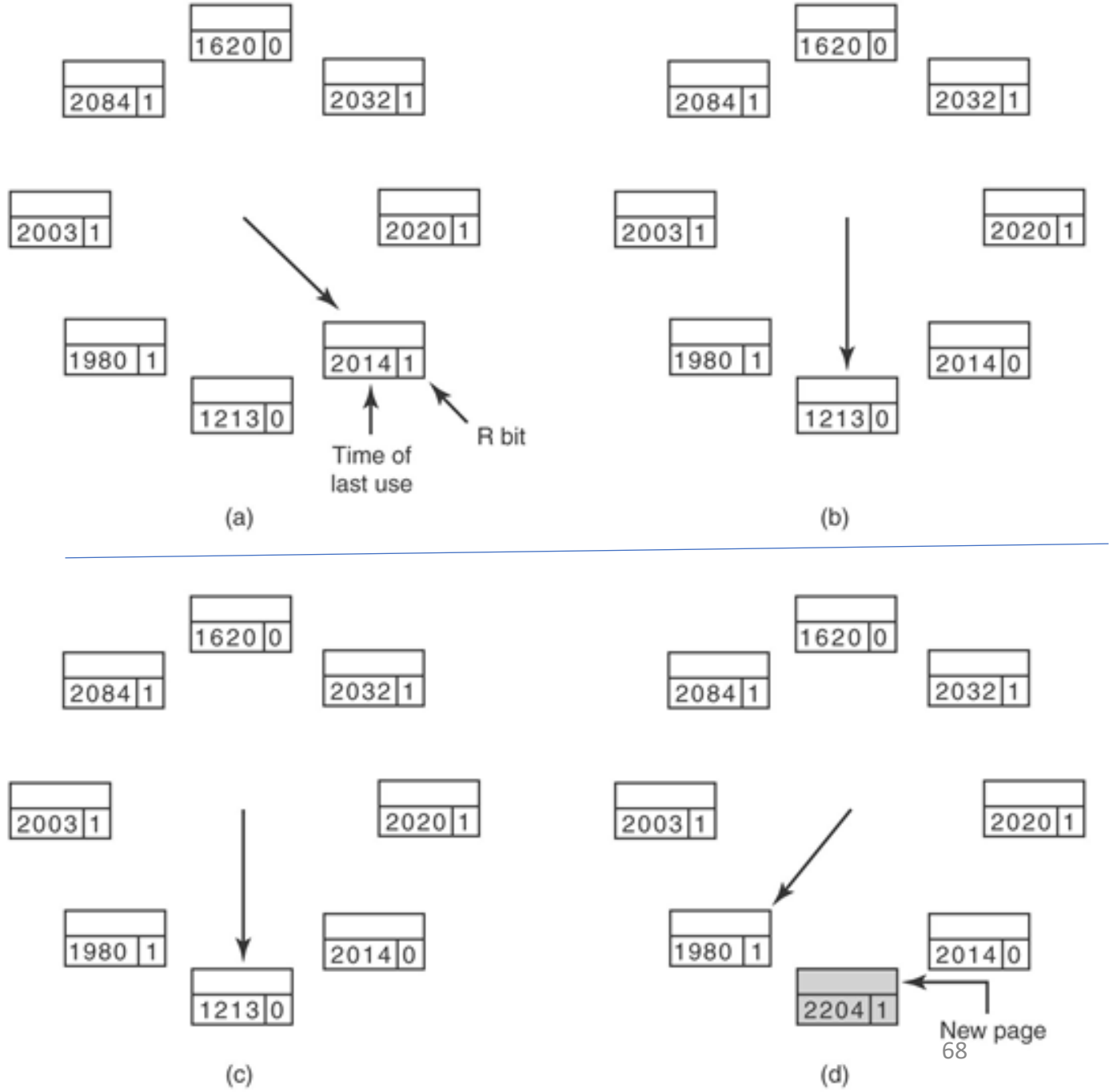
```
Starting at the clock hand

while !victim frame & !wrap around
  if reference bit set
    // no need to update ref time
    clear reference bit
  else
    if (time now – time referenced > τ)
      if (frame is modified)
        schedule write
      else
        select as victim frame

  advance clock hand
```

```
if (! victim frame)
    if writes scheduled
        wait for one of the writes
        to finish and select
     else
        all pages in working set,
        pick one at random
```

Example

Current virtual time 2204

a) Reference bit cleared
b) We move the clock hand
c) Examine time:
   2204 – 1213 > τ
d) select as victim, advance hand



Page replacement algorithms summary

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude approximation of LRU
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

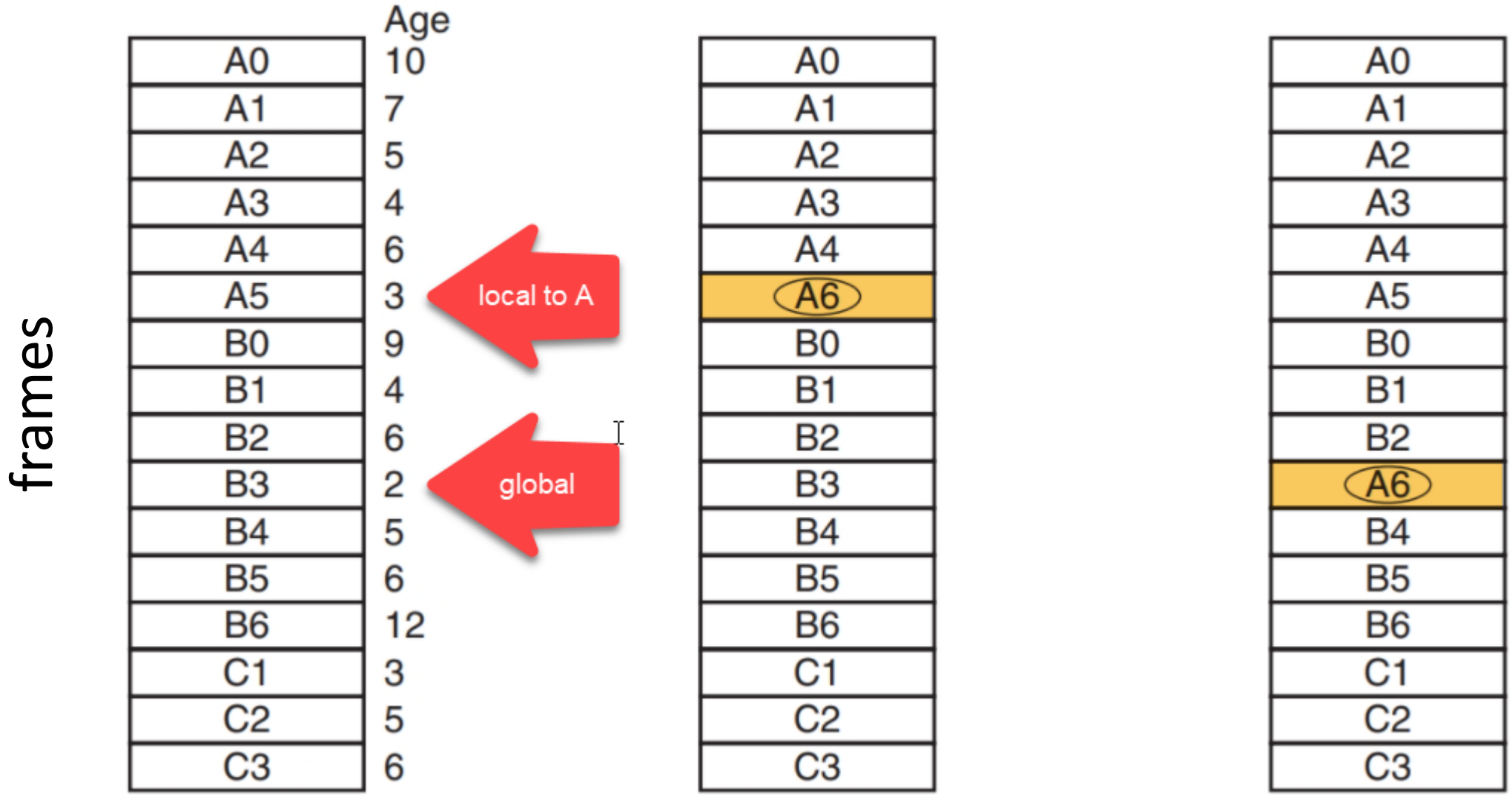
Tanenbaum & Bos Fig 3-21

Design issues: Allocation polices

- local – only select victim frames from process that needs the frame
- global – victim frame from any process

- Global usually better.
- Local polies usually result in too many or too few frames
  (Possible to mitigate by adjusting frame pool size over time)
- Too few frames can keep a process from executing
  (details later)

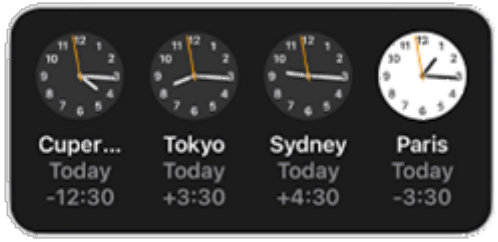
Local vs global example: P_A page 6 faults



Tanenbaum & Bos Fig. 3-22

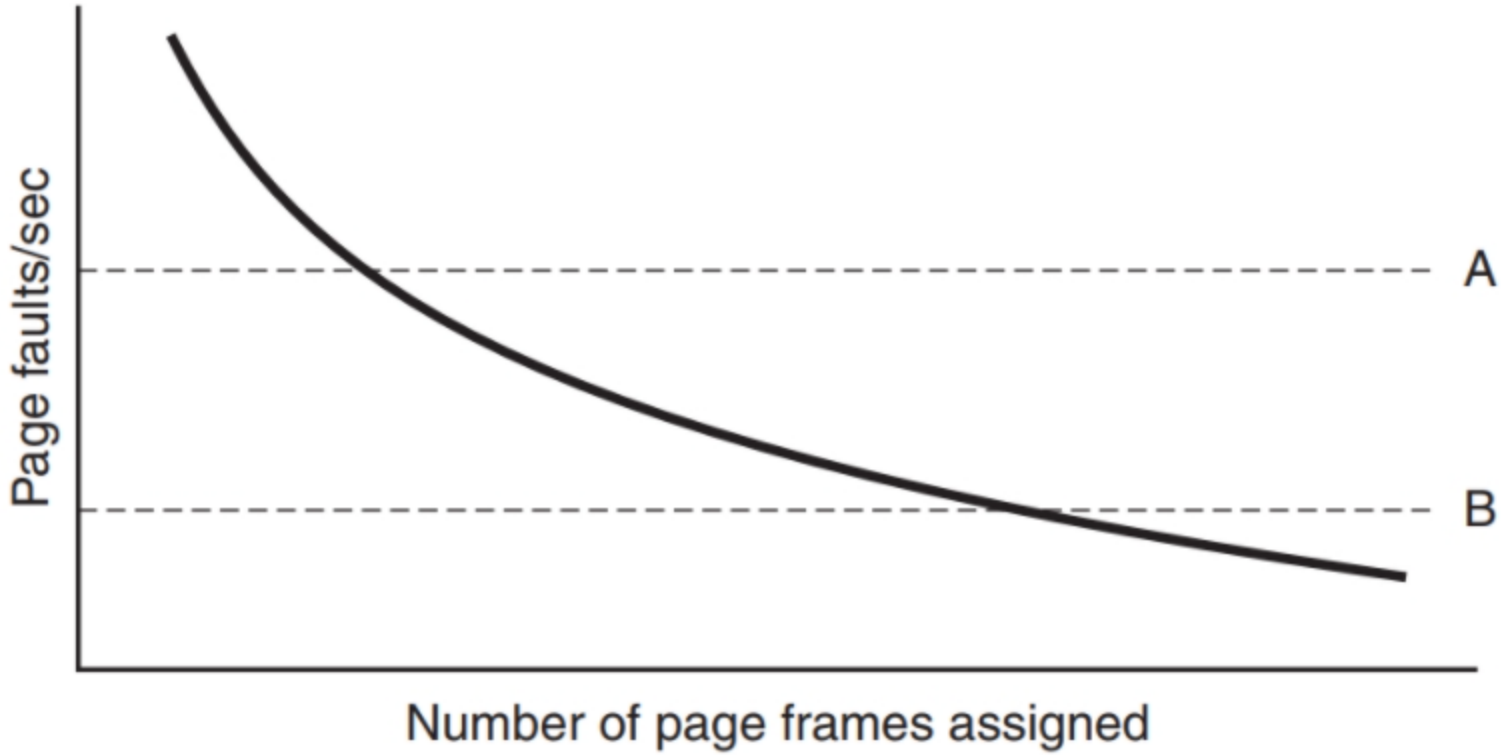
Number of frames per process

- Does not make sense to assign the same number of frames to each process
- There is a minimum number of pages
  - Architectures can access some maximum number of addresses per instruction (up to six!)
  - If a process has fewer than this, we could endlessly page fault…



Pixar Renderman

Page fault frequency models



- Is a process page faulting too often?
- Are there more pages than in the working set?

Page fault frequency models

- Used with global page replacement algorithm
- Does not select victim frames, merely lets us know whether or not frames should be allocated or deallocated from a process
- Some page replacement algorithms are implicitly local, others can be either

More design issues

- Sharing:
  - Already covered how, why do we need it?
  - Economy
    - program text
    - shared libraries (either loaded at process creation or on demand)
  - Communication

- Memory mapped files

Paging daemons

- Background process to ensure adequate supply of free frames
- Periodically looks for victims if supply inadequate
  - Retains victim information
  - Schedules writes of dirty pages

- If a preemptively chosen victim is needed again before being reallocated, can be reclaimed quickly

Page fault!

1. CPU generates interrupt, PC saved on stack
   Information about state of instruction is saved in specialized registers
2. Registers preserved (and perhaps more)
3. Determine page that faulted (TLB/page table walk)
4. Verify page is valid and access is allowable.
   If not, signal/kill process, otherwise grab free frame or select victim
5. If frame dirty, schedule write and block process. Mark frame as busy to prevent being used by another process

Page fault!

6. Once frame available and clean, schedule read from disk
7. When read completes, update page tables
8. Reset instruction to prior state (more details next)
9. Restore state
10. Process continues executing

Instruction backup

- Consider Motorola M680x0
  MOV.L #6(A1), 2(A0) ; Address indirect with displacement

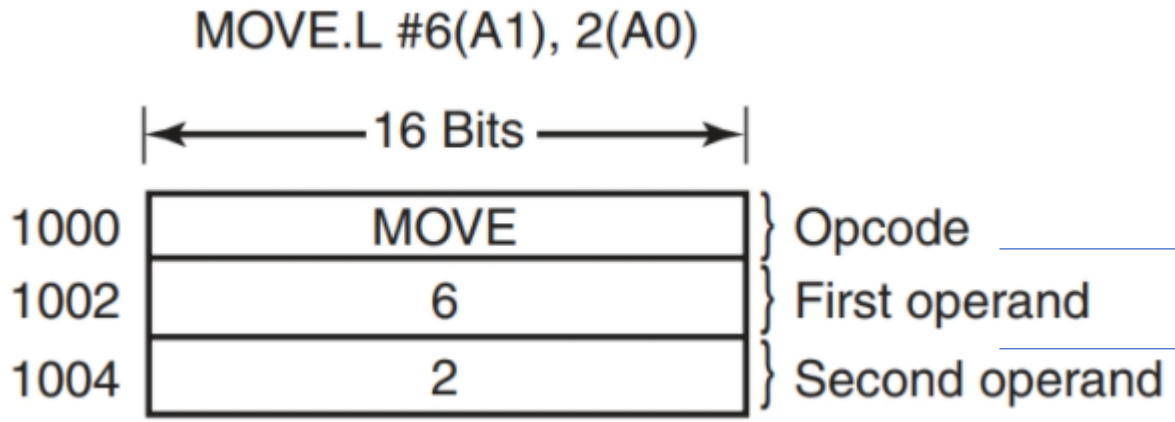


Fig. 3-27 (Tannenbaum & Bos)

- Where did the page fault occur? What was the PC?

This can get much nastier, e.g. MOVE.B (A0)+,(A1)+ registers are changed as they are accessed!

Instruction backup

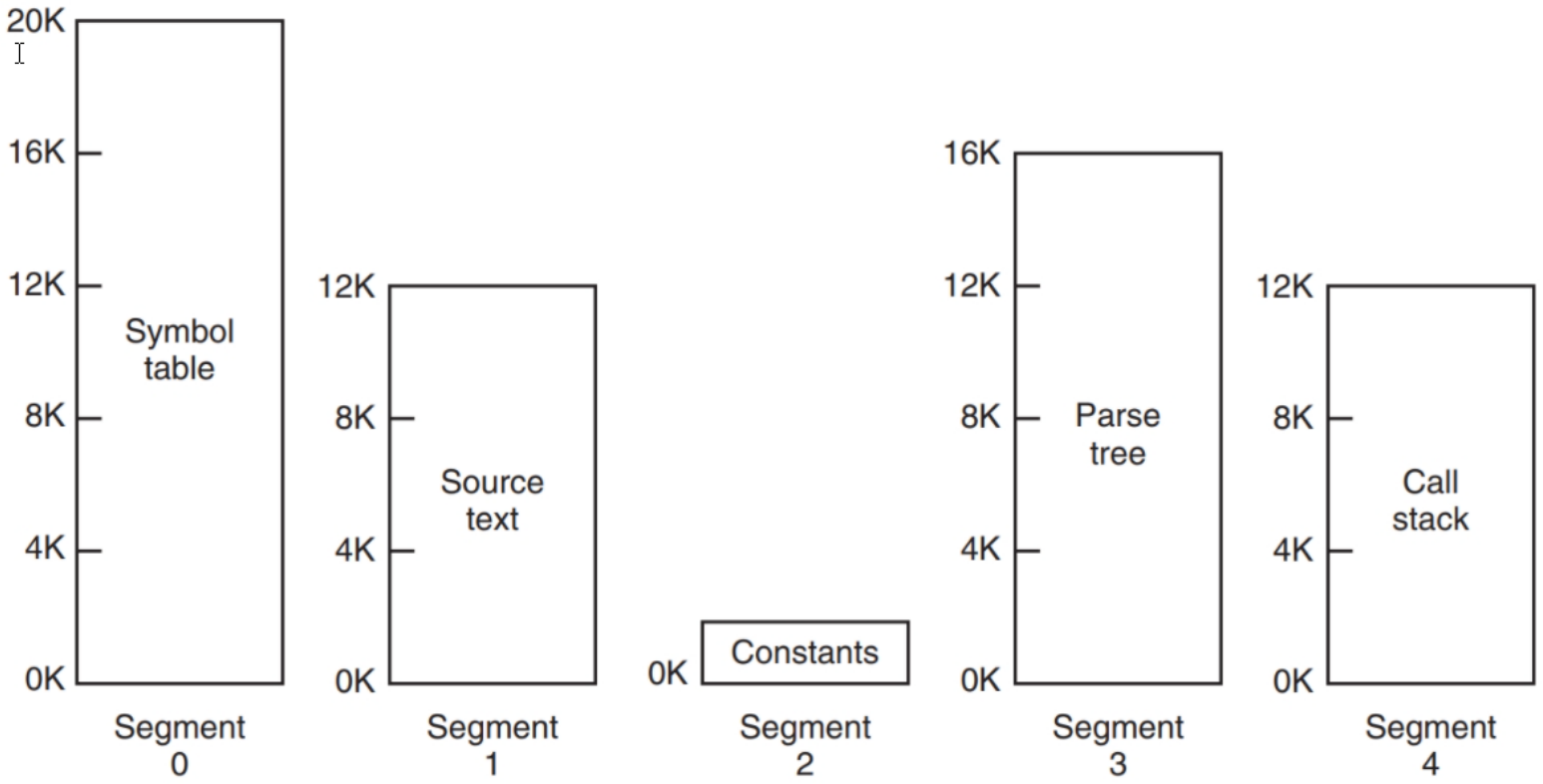
- On some machines, these changes are logged to special registers
- We can then restore state to the beginning of the instruction and restart it safely

Pinning

- Method to lock pages in memory
- Required to allow DMA
- Desirable for some high use memory locations

Segmentation

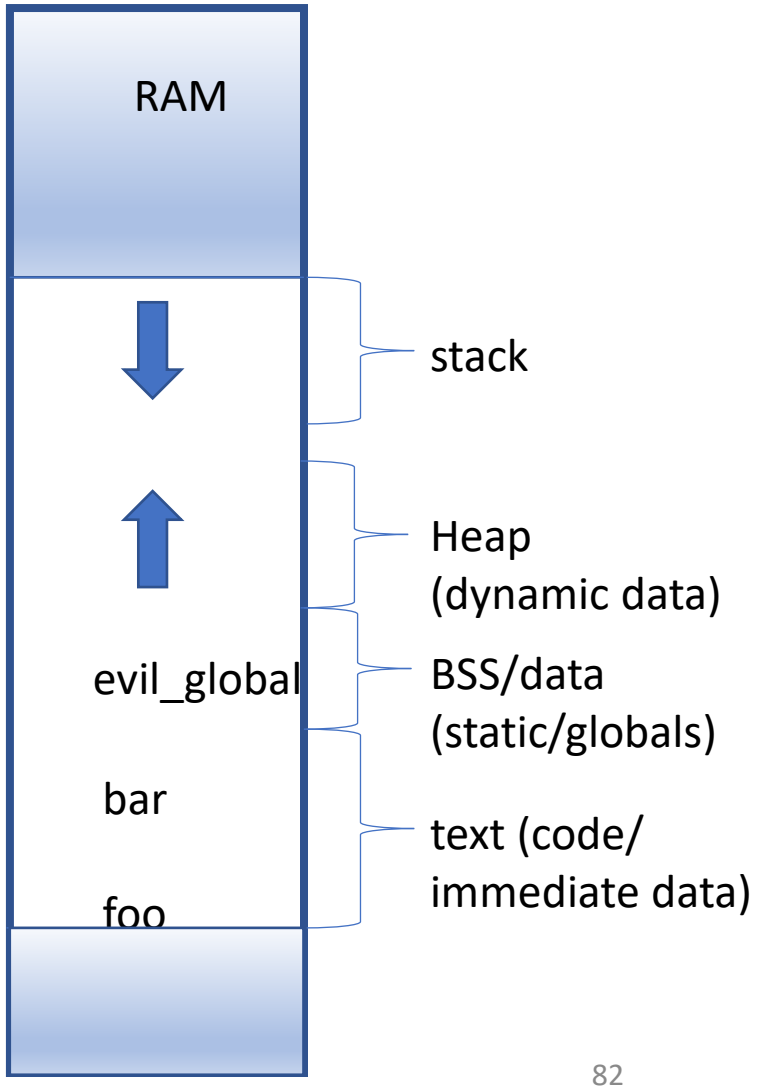
- Alternate model of logical memory
- System supports variable-sized segments



Segmentation

- At a low level (e.g. assembler), programmer must be aware of segments

- Provides flexibility
  - Segments can grow/shrink without interfering with one another
  - Contrast with our model for stack and heap:



Segmentation

- Finding physical memory for segments is tricky as they must be contiguous
- Segmentation with paging solves this by adding a paging layer on top of segments
- Not really used, pioneered by MULTICS, implemented in pre x64 Intel systems, but the major operating systems never used it