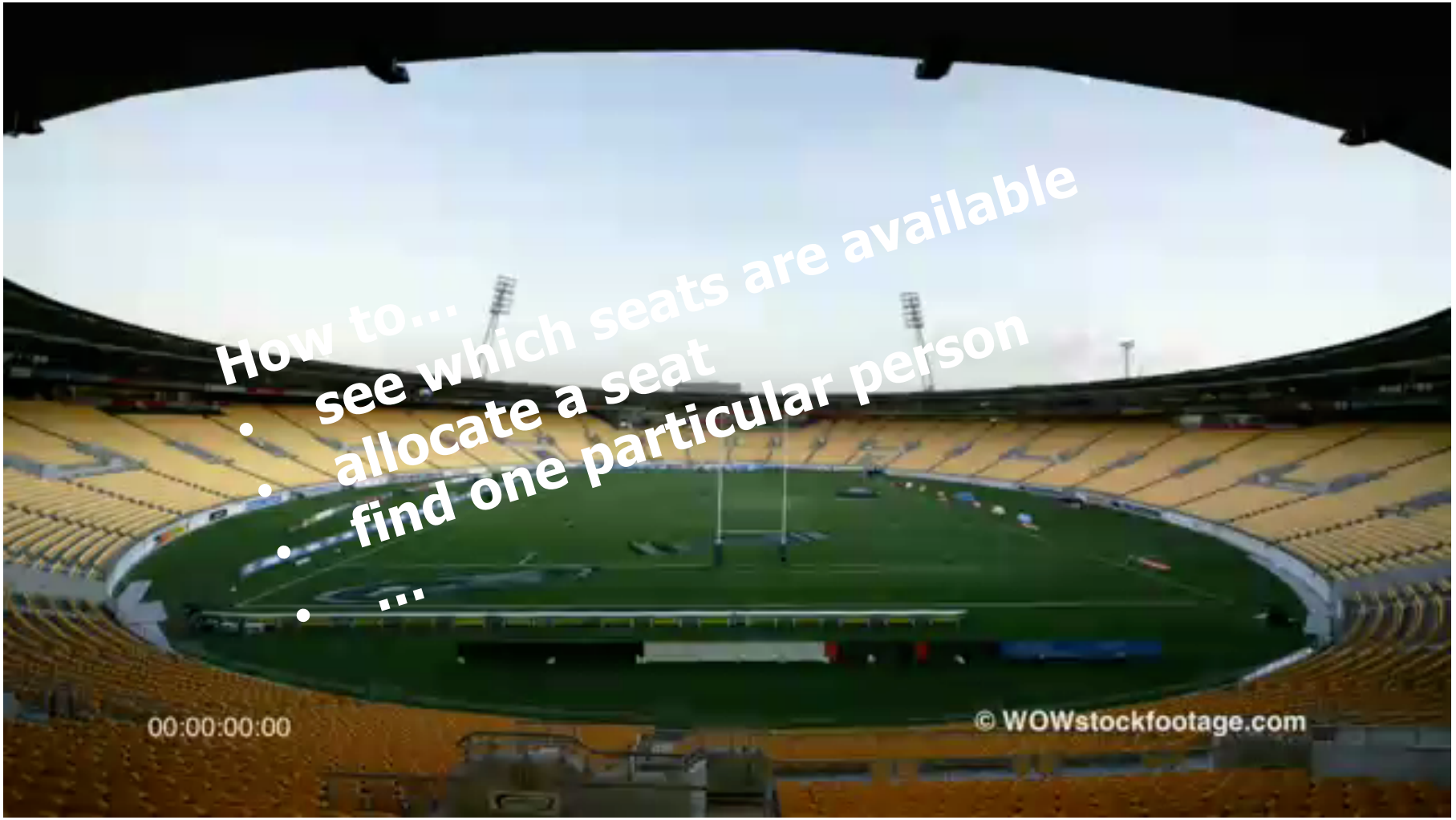
IN2140:
Introduction to Operating Systems and Data Communication

Operating Systems:

Memory

Challenge – managing memory



Overview

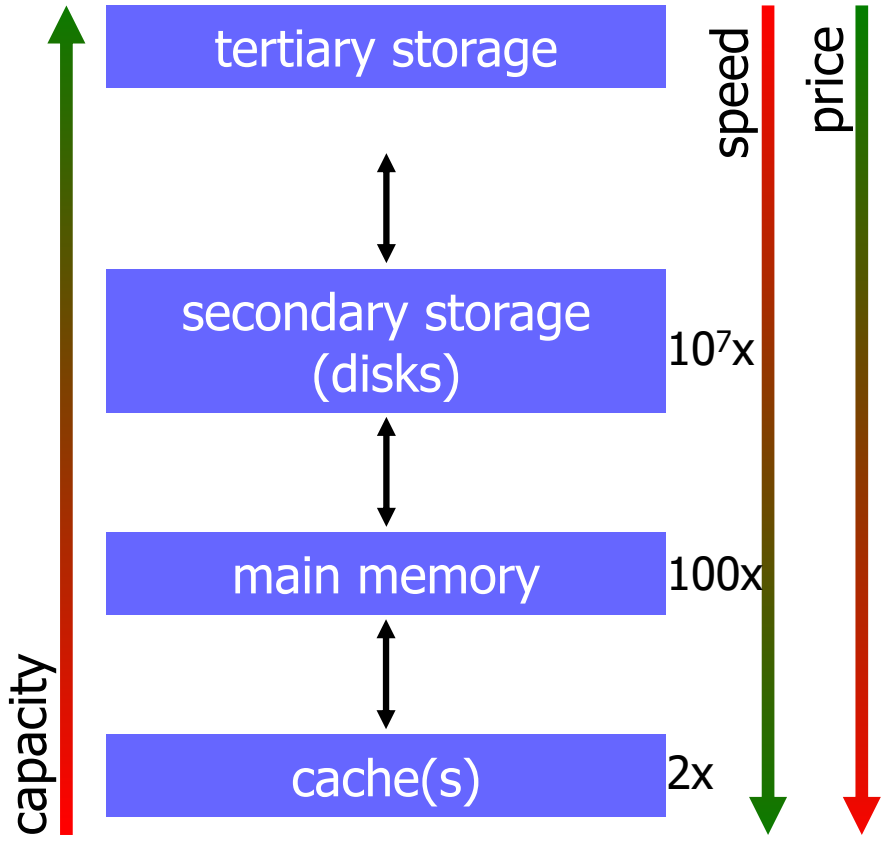
- Hierarchies
- Multiprogramming and memory management
- Addressing
- A process' memory
- Partitioning
- Paging and Segmentation
- Virtual memory
- Page replacement algorithms
- Paging example in IA32
- Data paths

Memory Management

- Memory management is concerned with managing the systems' memory resources
  - allocate space to processes
  - protect the memory regions
  - provide a virtual view of memory giving the impression of having more than the number of available bytes
  - control different levels of memory in a hierarchy

Memory Hierarchies

- A process typically needs a lot of memory:
  We can't access the disk every time we need data
- Typical computer systems therefore have several different components where data may be stored
  - different capacities
  - different speeds
  - less capacity gives faster access and higher cost per byte
- Lower levels have a copy of data in higher levels
- A typical memory hierarchy:



Memory Hierarchies

Tapes? Well, they …

- store a LOOOT of data
  (IBM TS3500 Tape Library: 2.25 exabytes)
- are SLOOOOOOOW…
  (seconds → minutes → hours)

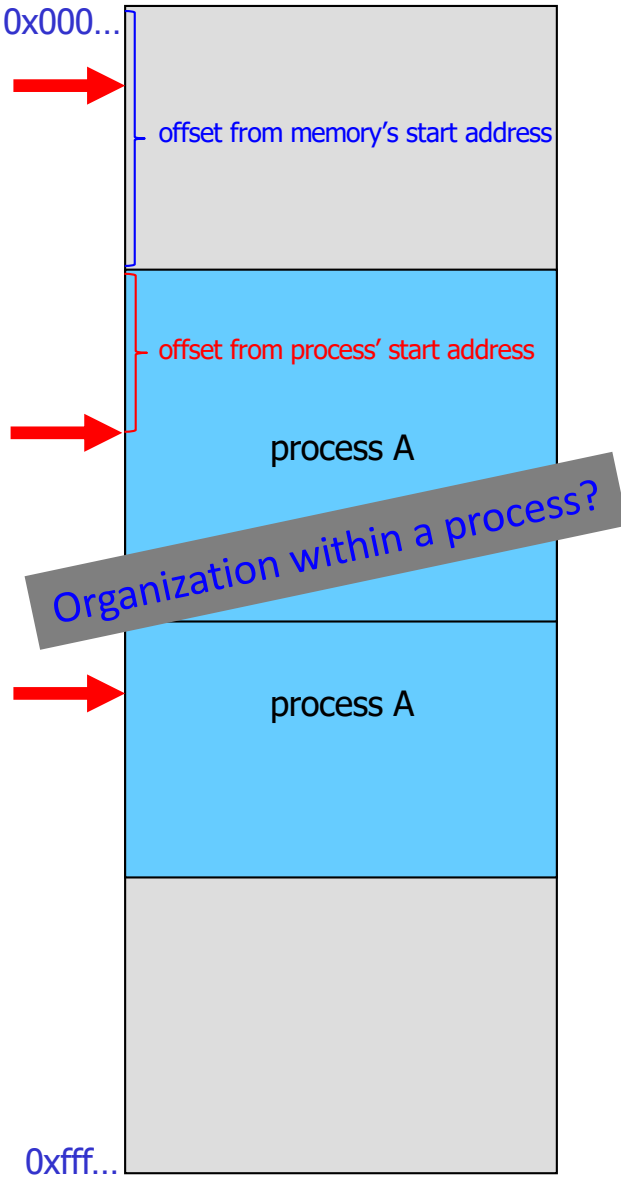
  → Going to other far away galaxies…
  → "5 sec ≈ ~290 years"

Absolute and Relative Addressing

- Hardware often uses absolute addressing
  - reserved memory regions
  - reading data by referencing the byte numbers in memory
  - read absolute byte 0x000000ff
  - *fast!!!*

- What about software?
  - read absolute byte 0x000fffff (process A)
  - ⇨ result dependent of a process' physical location
  - absolute addressing not convenient
  - but, addressing within a process is determined during programming!!??

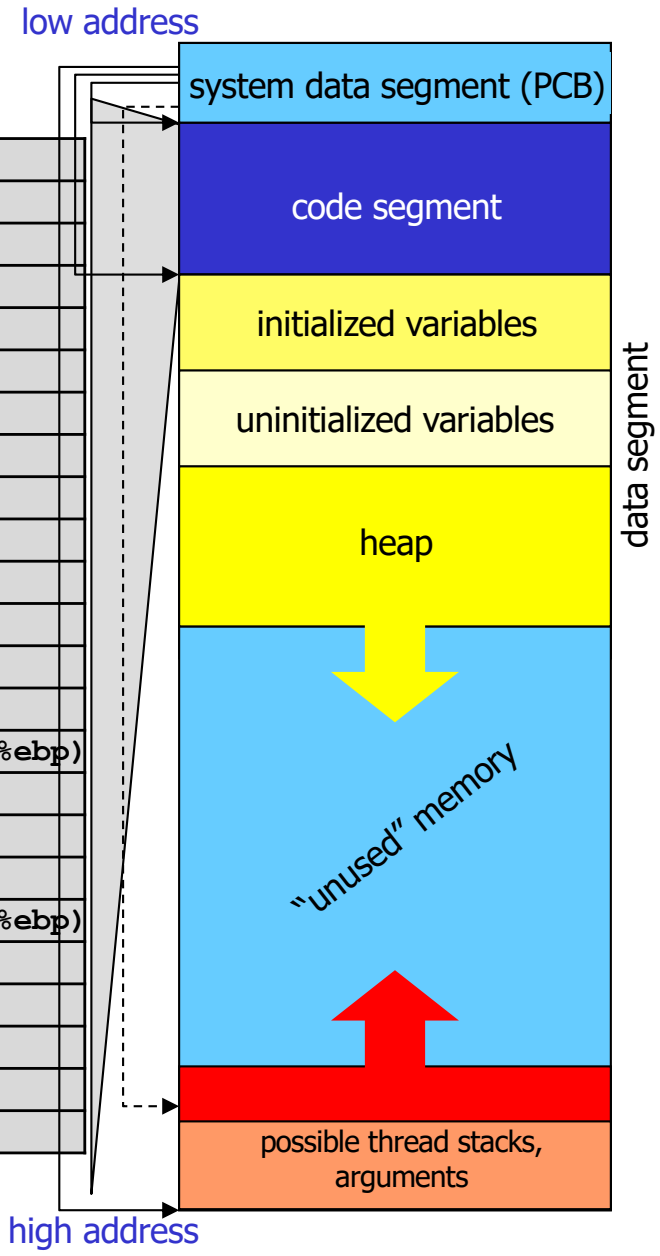
- Relative addressing
  - independent of process position in memory
  - address is expressed *relative to some base location*
  - *dynamic address translation* – find absolute address during run-time adding the **relative** and **base** addresses



Processes' Memory Layout

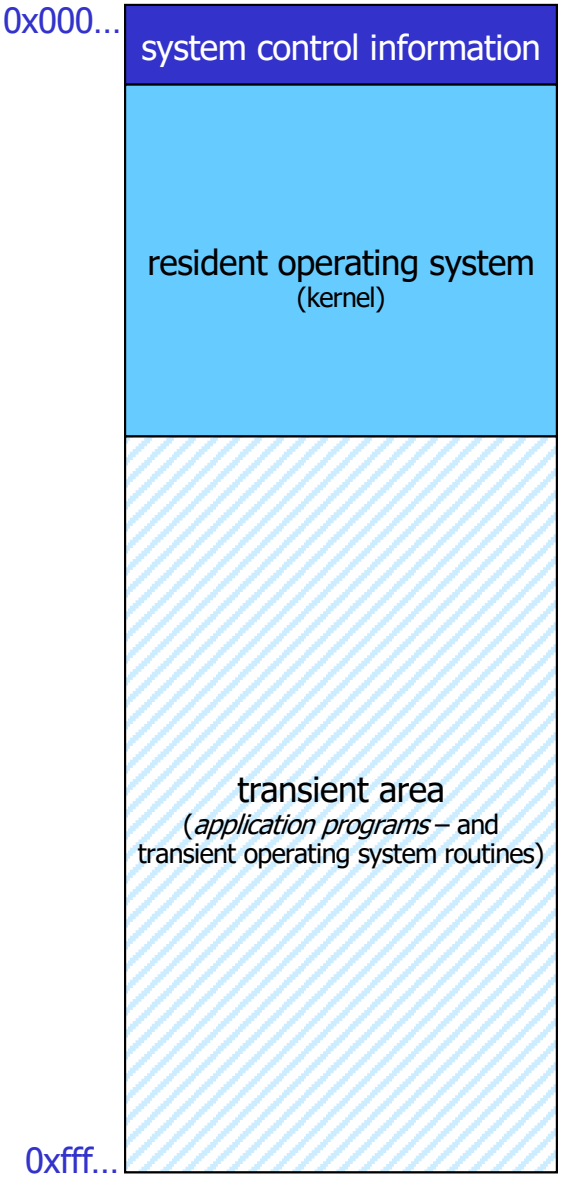
- On most architectures, a process partitions its available memory (address space), but for what?
  - a text (code) segment
    - read from program file for example by `exec`
    - usually read-only
    - can be shared
  - a data segment
    - initialized global/static variables (data)
    - uninitialized global/static variables (BSS)
    - heap
      - dynamic memory, e.g., allocated using `malloc`
      - grows against **higher** addresses
  - a stack segment
    - stores parameters/variables in a function
    - stores register states (e.g., calling function's EIP)
    - grows against **lower** addresses
  - system data segment (PCB)
    - segment pointers
    - pid
    - program and stack pointers
    - …
  - possibly more stacks for threads
  - command line arguments and environment variables at highest addresses

```
8048314 <add>:
8048314:   push   %ebp
8048315:   mov    %esp,%ebp
8048317:   mov    0xc(%ebp),%eax
804831a:   add    0x8(%ebp),%eax
804831d:   pop    %ebp
804831e:   ret
804831f <main>:
804831f:   push   %ebp
8048320:   mov    %esp,%ebp
8048322:   sub    $0x18,%esp
8048325:   and    $0xfffffff0,%esp
8048328:   mov    $0x0,%eax
804832d:   sub    %eax,%esp
804832f:   movl   $0x0,0xfffffffc(%ebp)
8048336:   movl   $0x2,0x4(%esp,1)
804833e:   movl   $0x4,(%esp,1)
8048345:   call   8048314 <add>
804834a:   mov    %eax,0xfffffffc(%ebp)
804834d:   leave
804834e:   ret
804834f:   nop
```



Global Memory Layout

- Memory is usually divided into regions
  - operating system occupies low memory
    - system control
    - resident routines
  - the remaining area is used for transient operating system routines and application programs
- How to assign memory to concurrent processes?

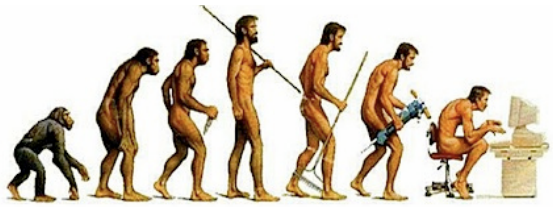


The Challenge of Multiprogramming

- Many "tasks" require memory
  - several processes concurrently loaded into memory

  - memory is needed for different tasks within a process

  - process memory demand may change over time



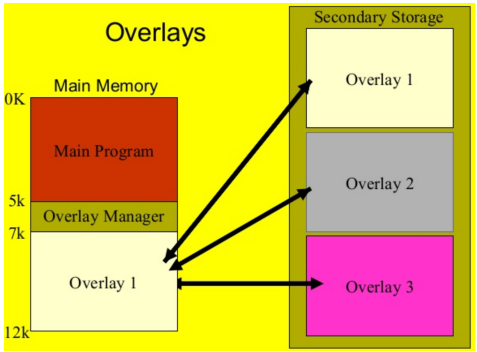
➥ **OS must arrange (dynamic) memory sharing**

Memory Management for Multiprogramming

- Use of secondary storage
  - keeping all programs and their data in memory may be impossible
  - move (parts of) a process from memory

- Swapping: remove a process from memory
  - with all of its state and data
  - store it on a secondary medium (disk, flash RAM, … , historically also tape)

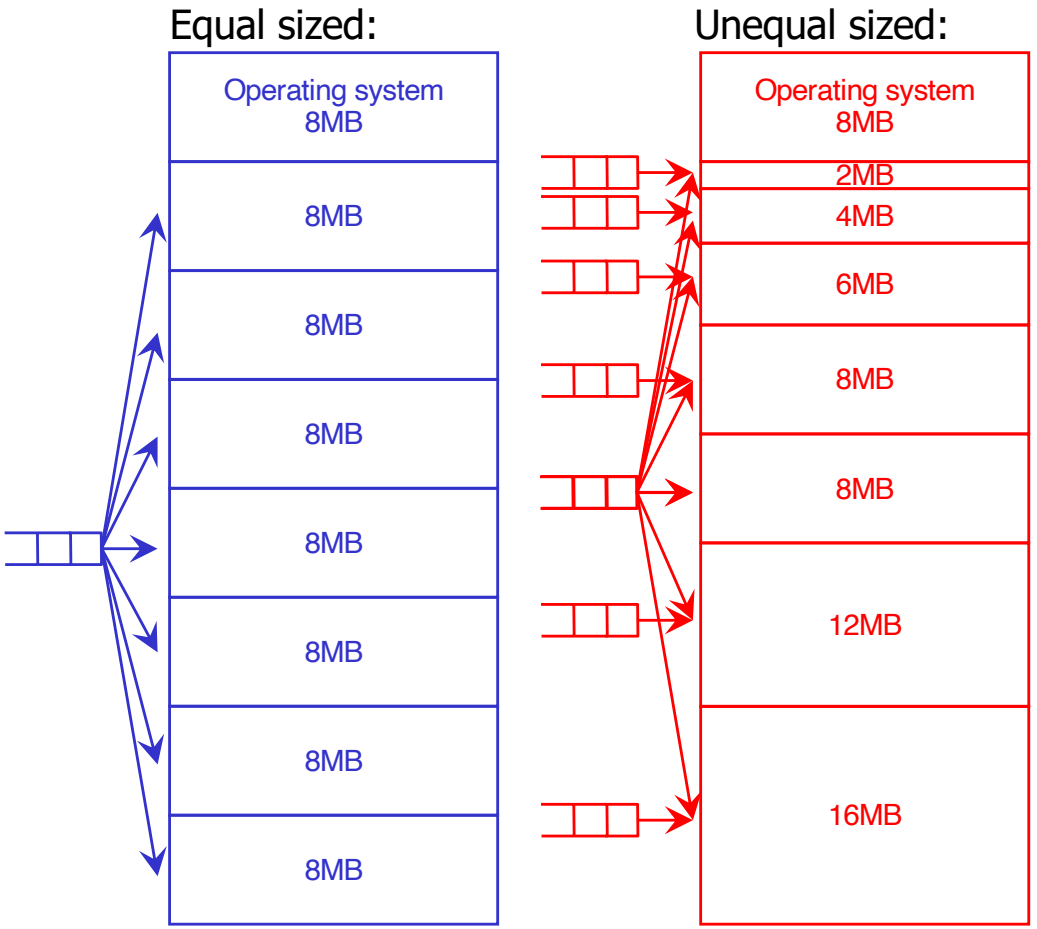
- Overlays: manually replace parts of code/data
  - programmer's rather than OS's work
  - only for very old and memory-scarce systems



- Segmentation/paging: remove parts of a process from memory
  - store it on a secondary medium
  - sizes of such parts are usually fixed

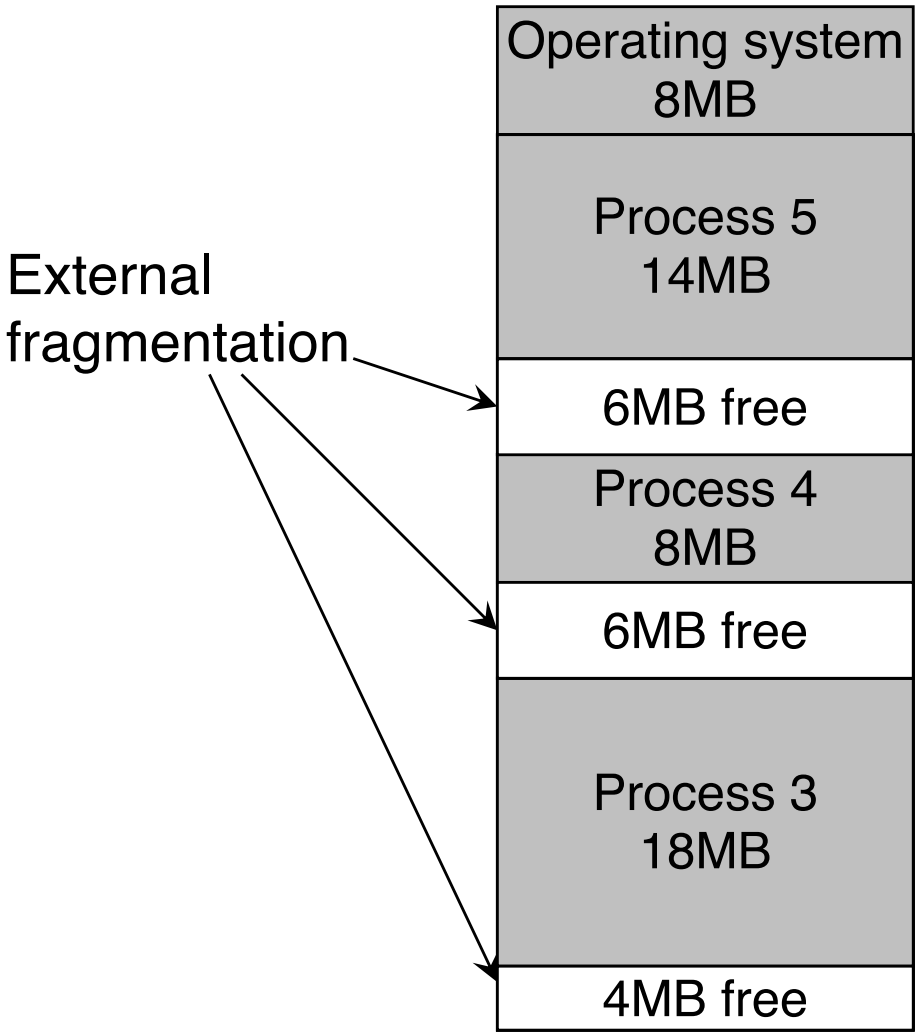
Fixed Partitioning

- Divide memory into static partitions at system initialization time (boot or earlier)
- Advantages
  - easy to implement
  - can support swapping of processes
- Equal-size partitions
  - large programs cannot be executed (unless parts of a program are loaded from disk)
  - small programs don't use the entire partition (problem called "internal fragmentation")
- Unequal-size partitions
  - large programs can be loaded at once
  - less internal fragmentation
  - require assignment of jobs to partitions
  - one queue or one queue per partition
  - ...but, what if only small or large processes?



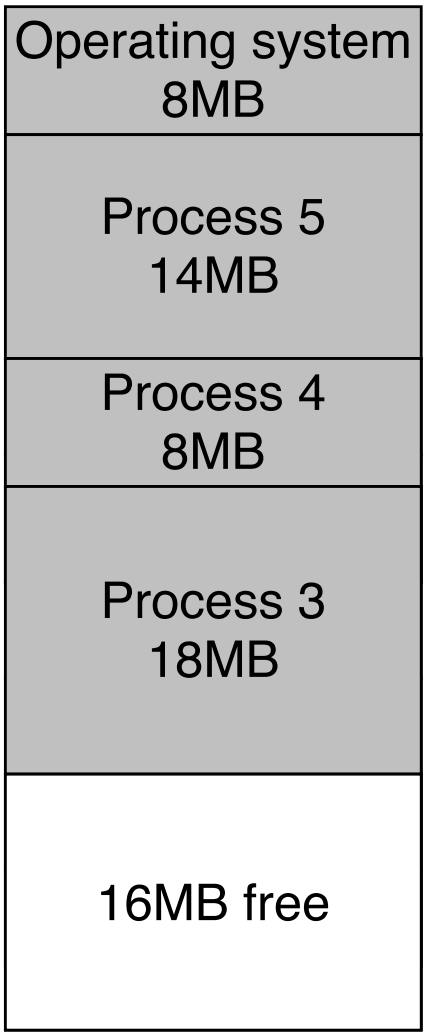
Dynamic Partitioning

- Divide memory at run-time
  - partitions are created dynamically
  - removed after jobs are finished
- External fragmentation increases with system running time



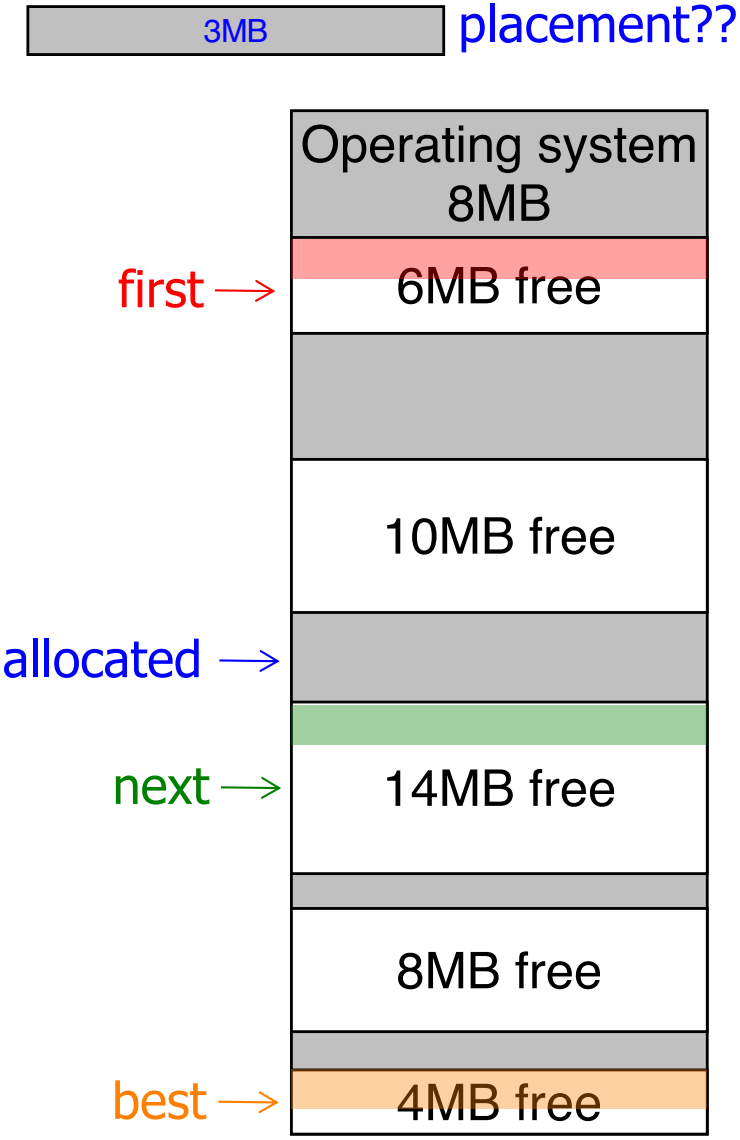
Dynamic Partitioning

- Divide memory at run-time
  - partitions are created dynamically
  - removed after jobs are finished
- External fragmentation increases with system running time
- ***Compaction*** removes fragments by moving data in memory
  - takes time
  - consumes processing resources



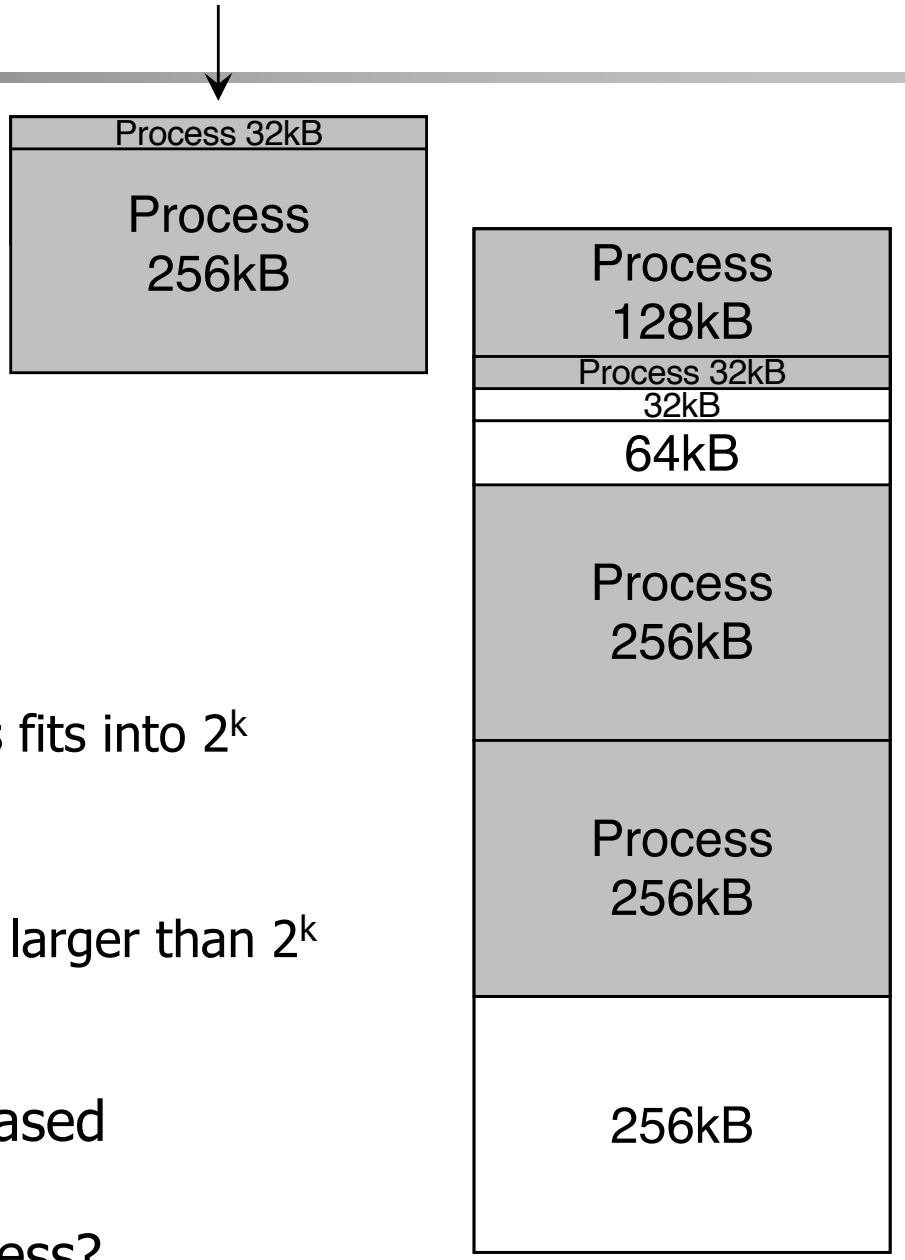
Dynamic Partitioning

- Divide memory at run-time
  - partitions are created dynamically
  - removed after jobs are finished
- External fragmentation increases with system running time
- ***Compaction*** removes fragments by moving data in memory
  - takes time
  - consumes processing resources
- Proper placement algorithm might reduce need for compaction
  - first fit – simplest, fastest, typically the best
  - next fit – almost as first fit
  - best fit – slowest, lots of small fragments, therefore often worst



The Buddy System

- Mix of fixed and dynamic partitioning
  - partitions have sizes 2^k, $L \le k \le U$
- Maintain a list of holes with sizes
- Assigning memory to a process:
  - find the smallest *k* so that the process fits into 2^k
  - find a hole of size 2^k
  - if not available, split the smallest hole larger than 2^k recursively into halves
- Merge partitions if possible when released
- … but what if I now got a 513kB process?
  … do we really need the process in continuous memory?



Segmentation

- Requiring that a process is placed in contiguous memory gives much fragmentation (and memory compaction is expensive)

- Segmentation
  - different lengths
  - determined by programmer
  - memory frames

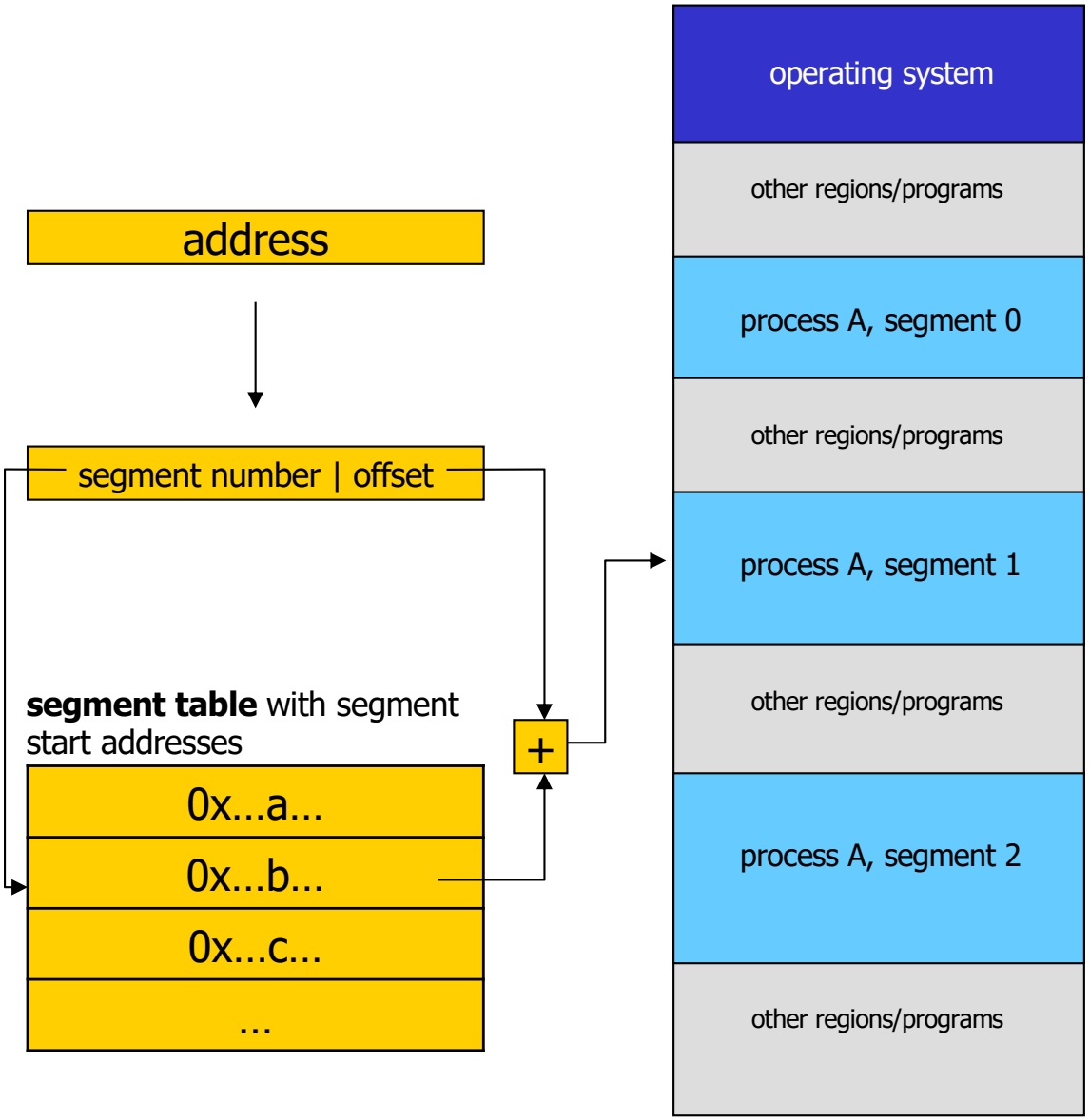
- Programmer (or compiler tool-chain) organizes program in parts
  - move control
  - needs awareness of possible segment size limits

- Pros and Cons
  - principle as in dynamic partitioning – can have different sizes
  - no internal fragmentation
  - less external fragmentation because on average smaller segments

  - adds a step to address translation

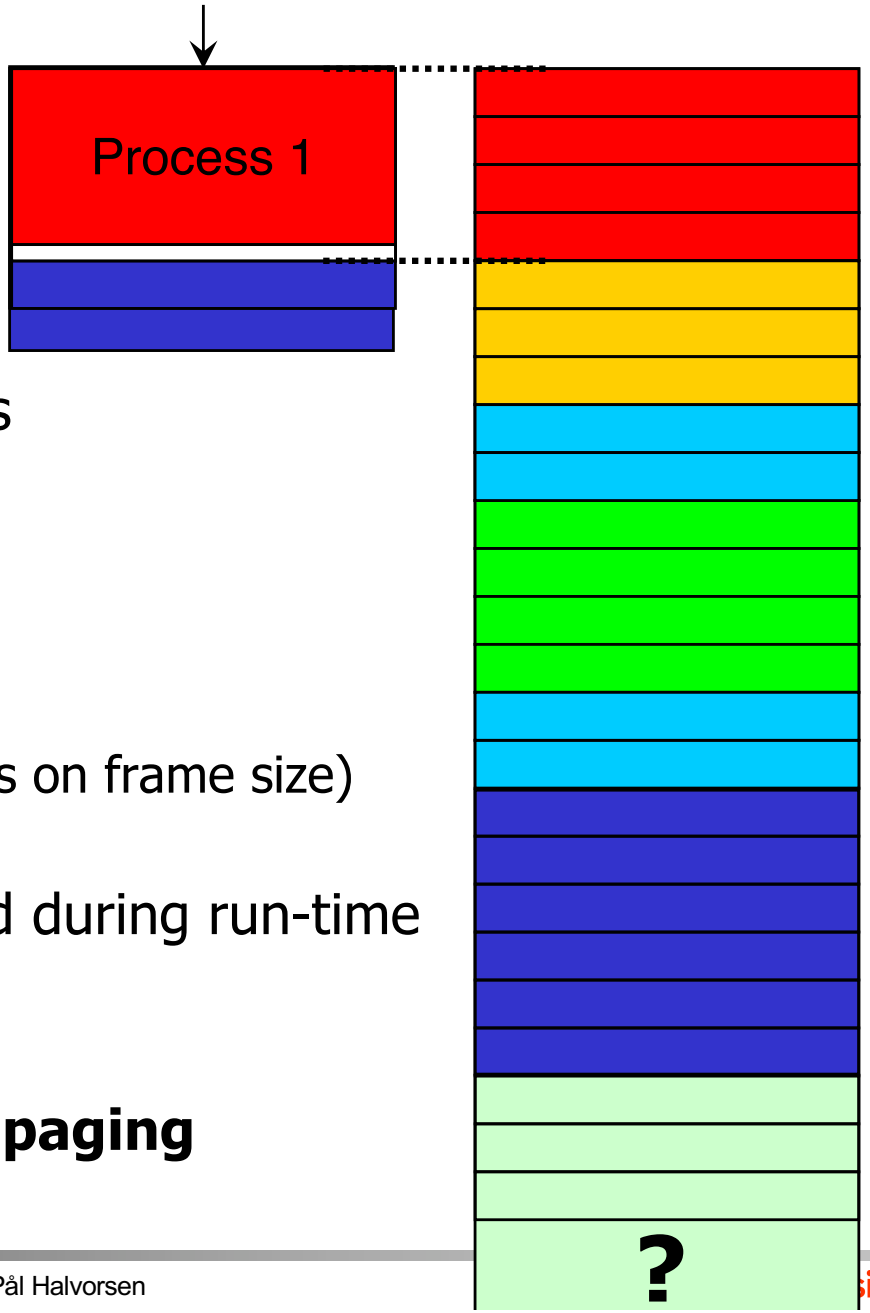
Segmentation: address lookup

1. find segment table address in register
2. extract segment number from address
3. find segment address using segment number as index to segment table
4. find absolute address within segment using relative address



Paging

- Paging
  - equal lengths determined by processor
  - one page moved into one page (memory) frame

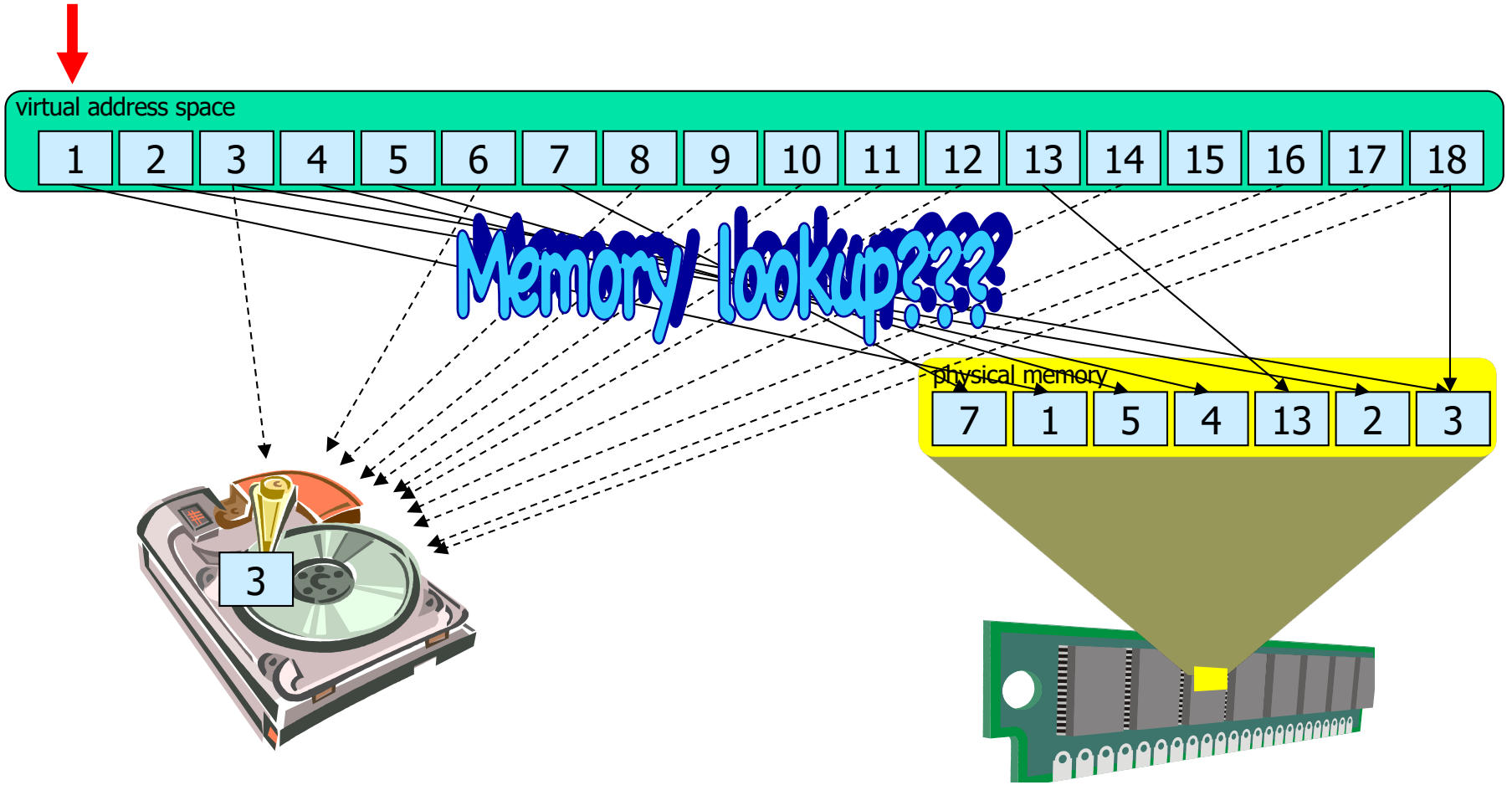


- Process is loaded into several frames (not necessarily consecutive)
- Fragmentation
  - no external fragmentation
  - little internal fragmentation (depends on frame size)
- Addresses are dynamically translated during run-time (similar to segmentation)
- **Can *combine* segmentation and paging**

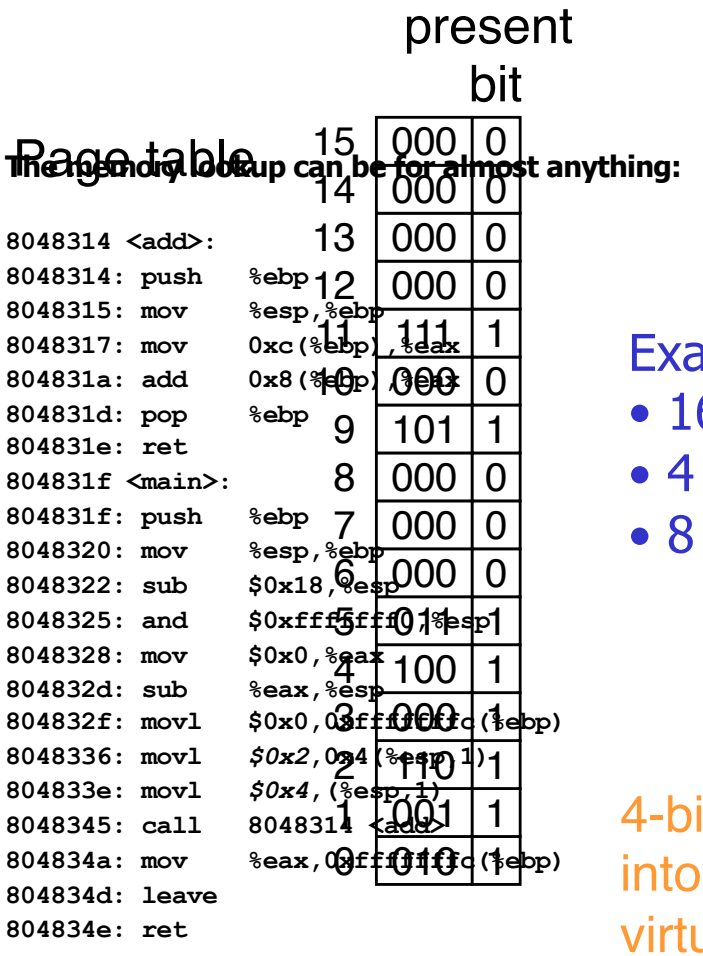
Virtual Memory

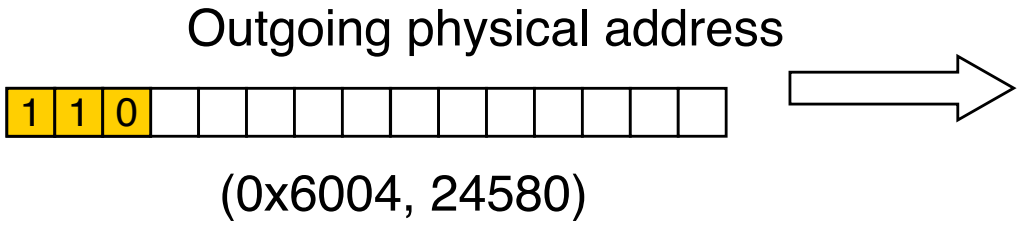
- The described partitioning schemes may be used in applications, but a modern OS also uses virtual memory:
  - early attempt to give a programmer more memory than physically available
    - older computers had relatively little main memory
    - but still today, all instructions do not have to be in memory before execution starts
      - break program into smaller independent parts
      - load currently active parts
      - when a program is finished with one part a new can be loaded
  - memory is divided into equal-sized frames often called *pages*
  - some pages reside in physical memory, others are stored on disk and retrieved if needed
  - virtual addresses are translated to physical addresses (in the MMU) using a *page table*
  - both Linux and Windows implement a flat linear 32-bit (4 GB) memory model on IA-32
    - **Windows**: 2 GB (high addresses) kernel, 2 GB (low addresses) user mode threads
    - **Linux**: 1 GB (high addresses) kernel, 3 GB (low addresses) user mode threads

Virtual Memory



Memory Lookup





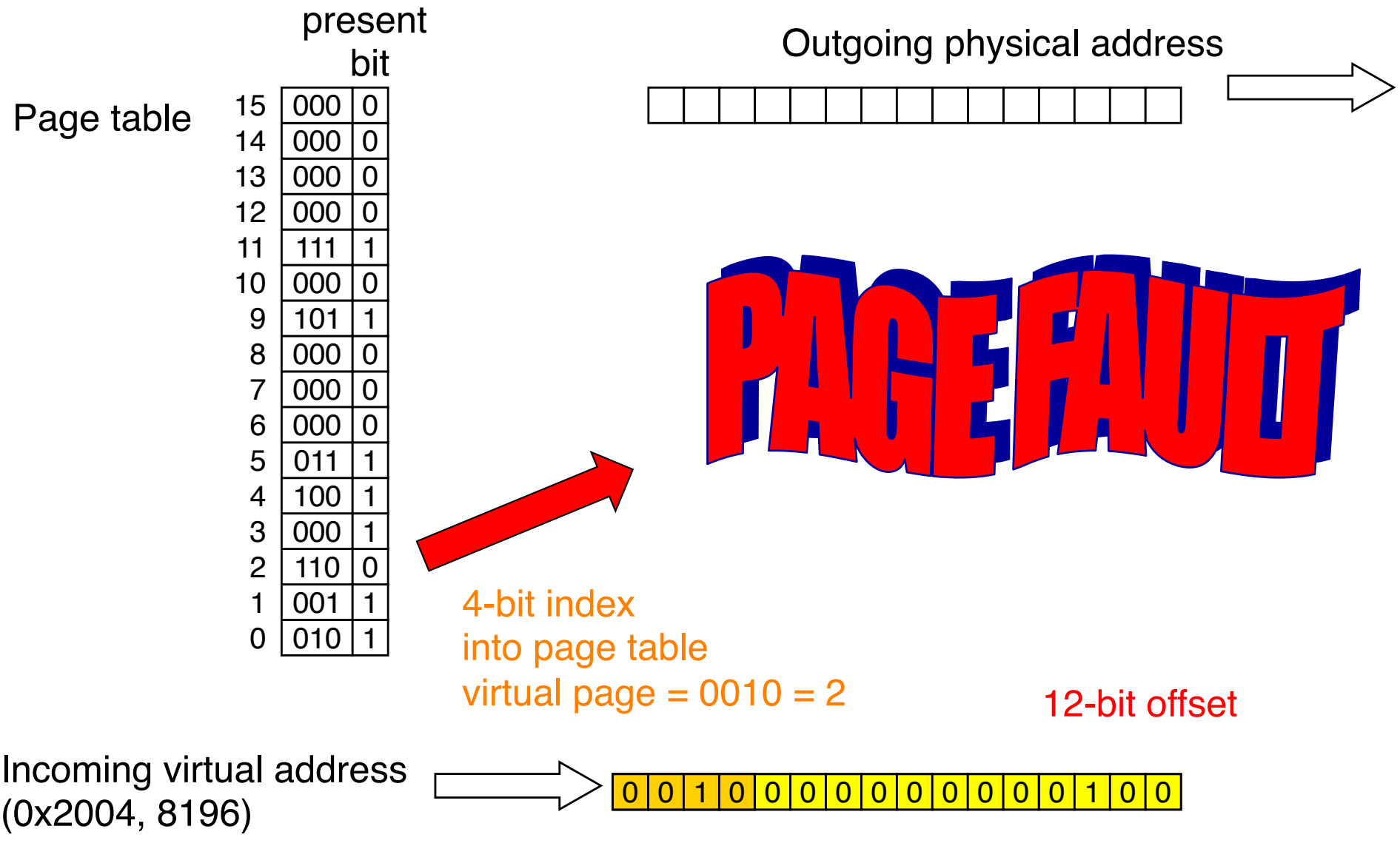
Example:
- 16 bit virtual address space → 16 pages (4-bit index)
- 4 KB pages (12-bit offsets within page)
- 8 physical pages (3-bit index)

4-bit index
into page table
virtual page = 0010 = 2

12-bit offset

Incoming virtual address
(0x2004, 8196)

Memory Lookup



Page Fault Handling

1. Hardware traps to the kernel saving program counter and process state information
2. Save general registers and other volatile information
3. OS discovers the page fault and determines which virtual page is requested
4. OS checks if the virtual page is valid and if protection is consistent with access
5. Select a page to be replaced
6. Check if selected page frame is ”dirty”, i.e., updated. If so, write back to disk, otherwise, just overwrite
7. When selected page frame is ready, the OS finds the disk address where the needed data is located and schedules a disk operation to bring in into memory
8. A disk interrupt is executed indicating that the disk I/O operation is finished, the page tables are updated, and the page frame is marked ”normal state”
9. Faulting instruction is backed up and the program counter is reset
10. Faulting process is scheduled, and OS returns to the routine that made the trap to the kernel
11. The registers and other volatile information are restored, and control is returned to user space to continue execution as no page fault had occured

Page Replacement Algorithms

- Page fault → OS has to select a page for replacement

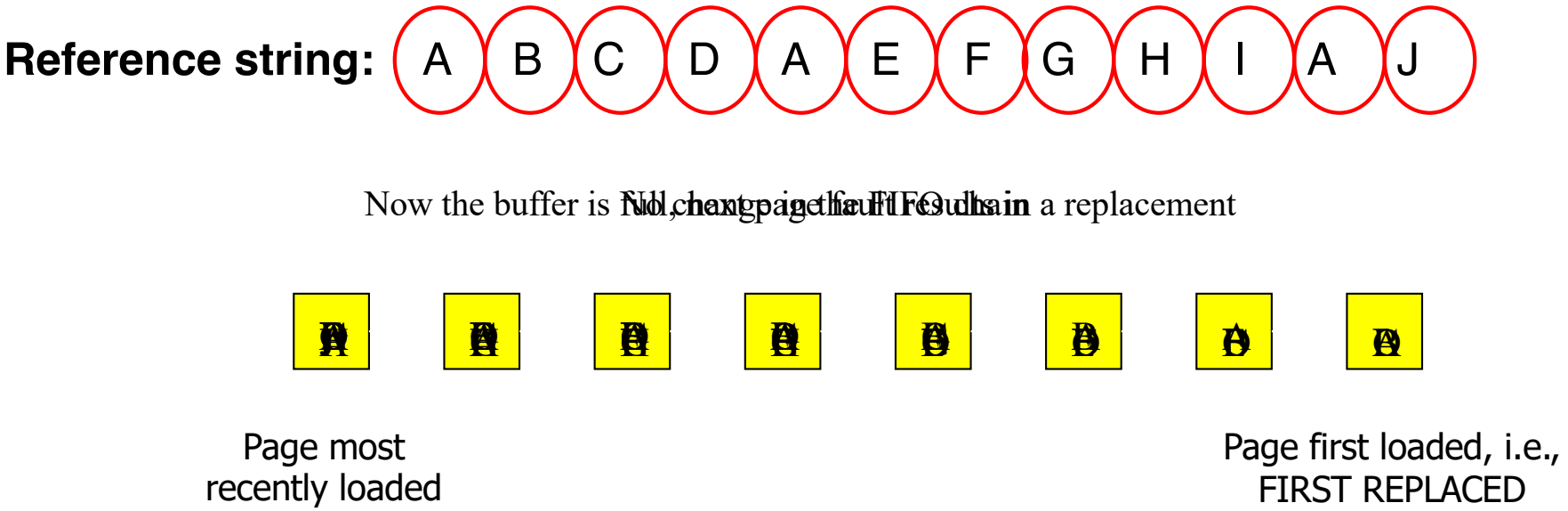
- How do we decide which page to replace?

  → determined by the ***page replacement algorithm***
  → several algorithms exist:

  - Random
  - Other algorithms take into account usage, age, etc.
    (e.g., FIFO, not recently used, least recently used, second chance, clock, ...)
  - which is best???

First In First Out (FIFO)

- All pages in memory are maintained in a list sorted by age
- FIFO replaces the oldest page, i.e., the first in the list



- Low overhead
- Non-optimal replacement (and disc accesses are EXPENSIVE)

➥ FIFO is rarely used in its pure form

Second Chance

- Modification of FIFO
- ***R*** bit: when a page is referenced again, the R bit is set,
  and the page will be treated as a newly loaded page

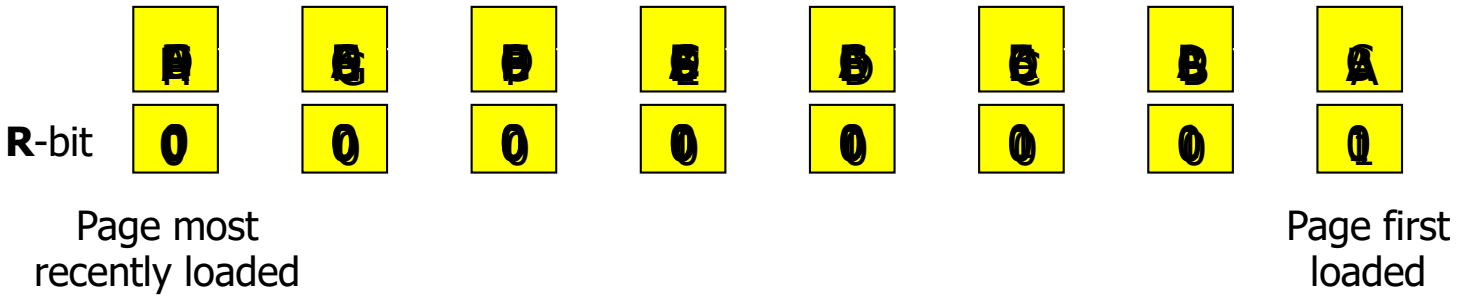
Reference string: A B C D A E F G H I

Page I will be inserted, find a page to page out by looking at the first page loaded:

-if R-bit = 0 ⇒ page out, shift chain left, and insert I last in the chain

-if R-bit = 1 ⇒ clear R-bit, move page last, and finally look at the new first page

Now the buffer is full, next page fault results in a replacement

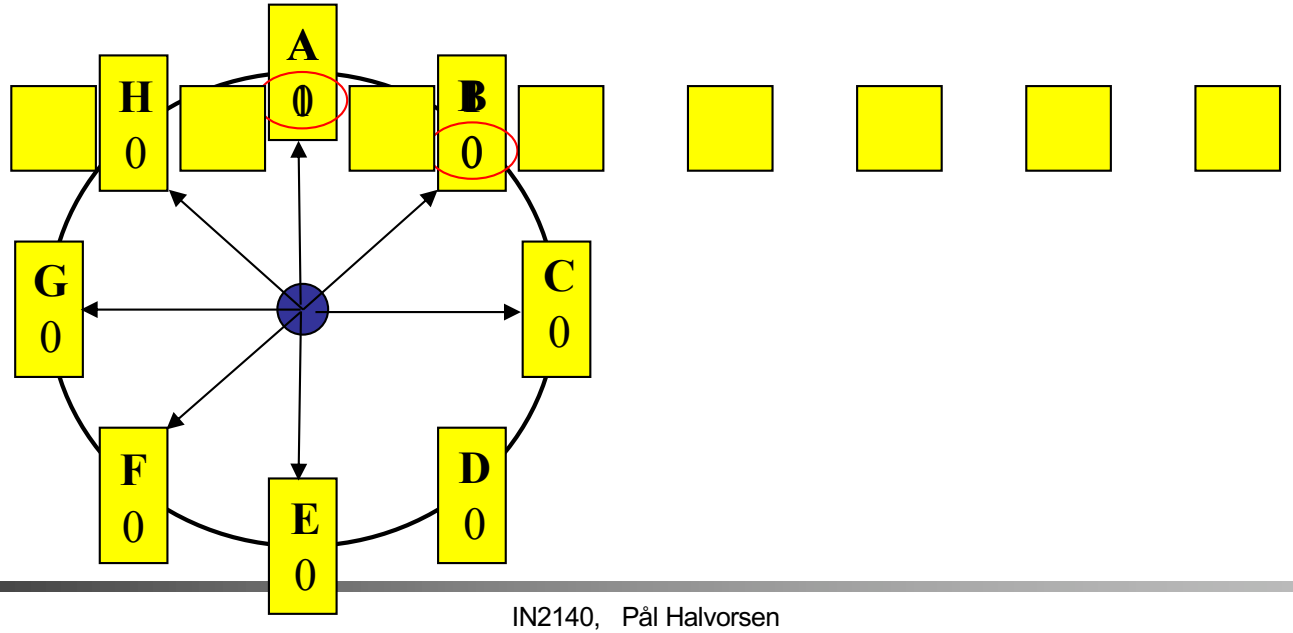


- *Second chance* is a reasonable algorithm,
  but inefficient because it is moving pages around the list

Clock

- More efficient implemention ***Second Chance***
- Circular list in form of a clock
- Pointer to the oldest page:
  - R-bit = 0 → replace and advance pointer
  - R-bit = 1 → set R-bit to 0, advance pointer until R-bit = 0, replace and advance pointer

Reference string: A B C D A E F G H I



Least Recently Used (LRU)

- Replace the page that has the longest time since last reference
- Based on the observation that

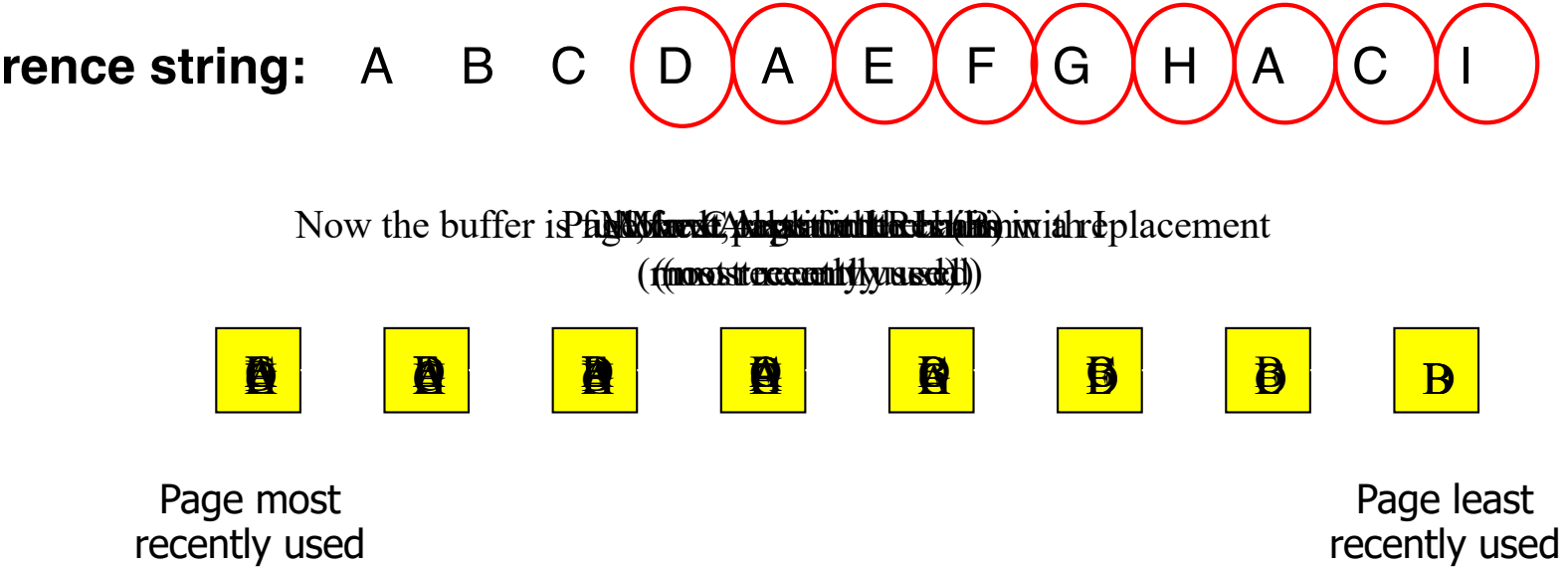
  pages that was heavily used in the last few instructions will probably be used again in the next few instructions

- Several ways to implement this algorithm

Least Recently Used (LRU)

- LRU as a linked list:

Reference string: A B C D A E F G H A C I



- Saves (usually) a lot of disk accesses
- **Expensive** - maintaining an ordered list of all pages in memory:
  - most recently used at front, least at rear
  - update this list every memory reference !!

- Many other approaches: using aging and counters (e.g., Working Set and WSClock)

Need For Application-Specific Algorithms?

- Most existing systems use an LRU-variant
  - keep a sorted list
  - replace last element in list
  - insert new data elements at the head
  - if a data element is re-accessed, move back to the end of the list

In this case, LRU replaces the next needed frame. So the answer is in many cases **YES...**

- Extreme example – video frame playout:

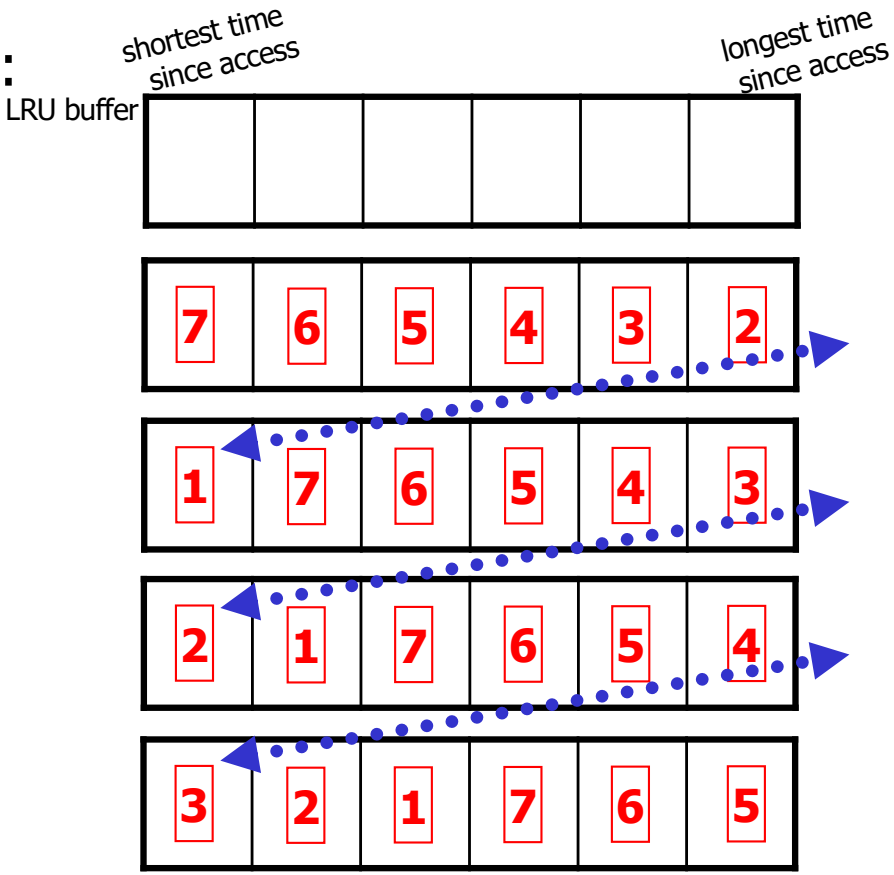
play video (7 frames): 1

rewind and restart playout at 1: 1

playout 2: 2

playout 3: 3

playout 4: 4



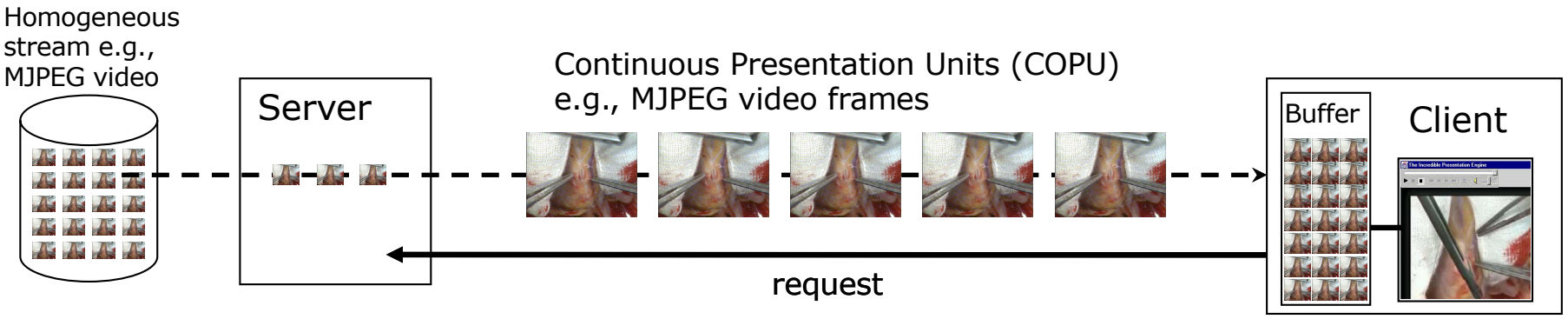
"Classification" of Mechanisms

- Block-level caching consider (possibly unrelated) set of blocks
  - each data element is viewed upon as an independent item
  - usually used in "traditional" systems
  - e.g., FIFO, LRU, LFU, CLOCK, …
  - multimedia (video) approaches:
    - *Least/Most Relevant for Presentation* (L/MRP)
    - …
- Stream-dependent caching consider (parts of) a stream object as a whole
  - related data elements are treated in the same way
  - research prototypes in multimedia systems
  - e.g.,
    - BASIC
    - DISTANCE
    - *Interval Caching* (IC)
    - *Generalized Interval Caching* (GIC)
    - Split and Merge (SAM)
    - …

Least/Most Relevant for Presentation (L/MRP)

[Moser et al. 95]

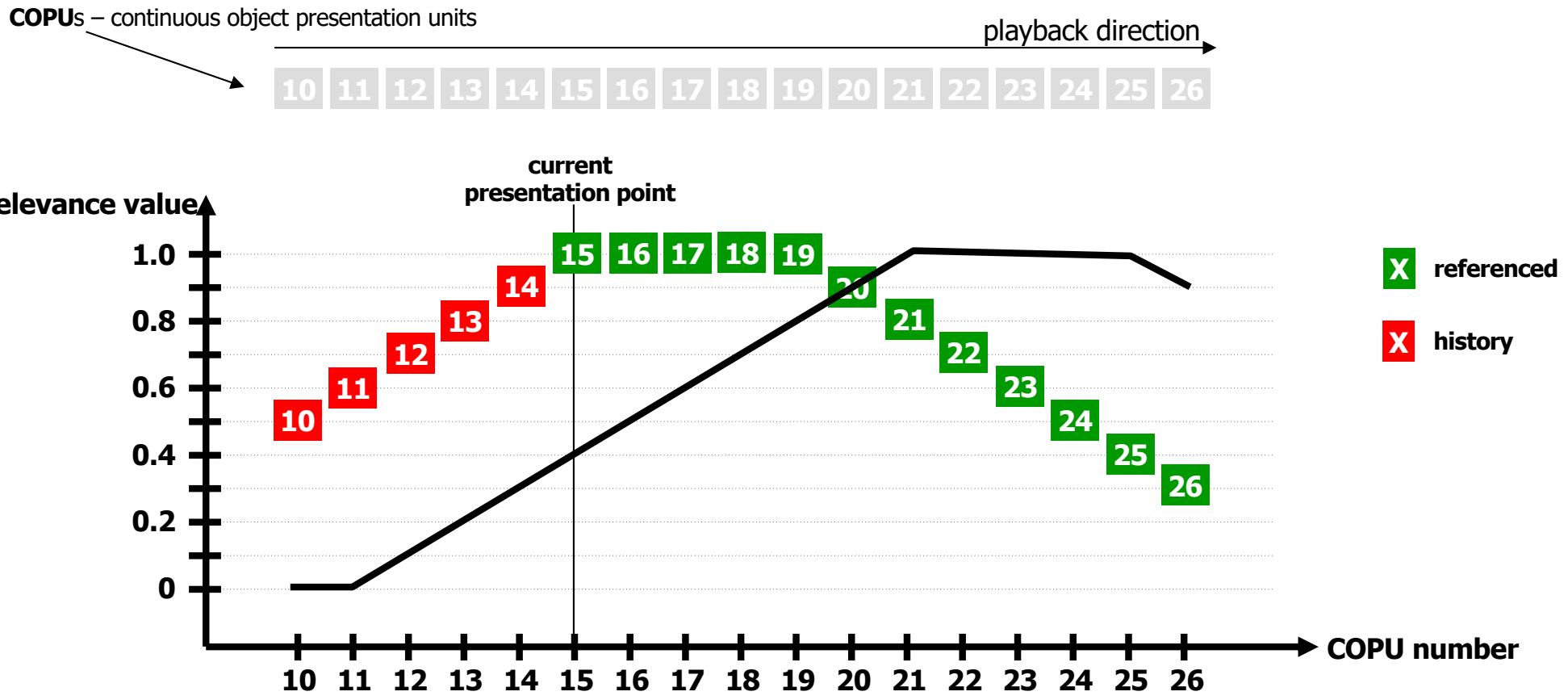
- L/MRP is a buffer management mechanism for a single interactive, continuous data stream
  - adaptable to individual multimedia applications
  - preloads units *most relevant for presentation* from disk
  - replaces units *least relevant for presentation*
  - client pull based architecture



Least/Most Relevant for Presentation (L/MRP)

[Moser et al. 95]

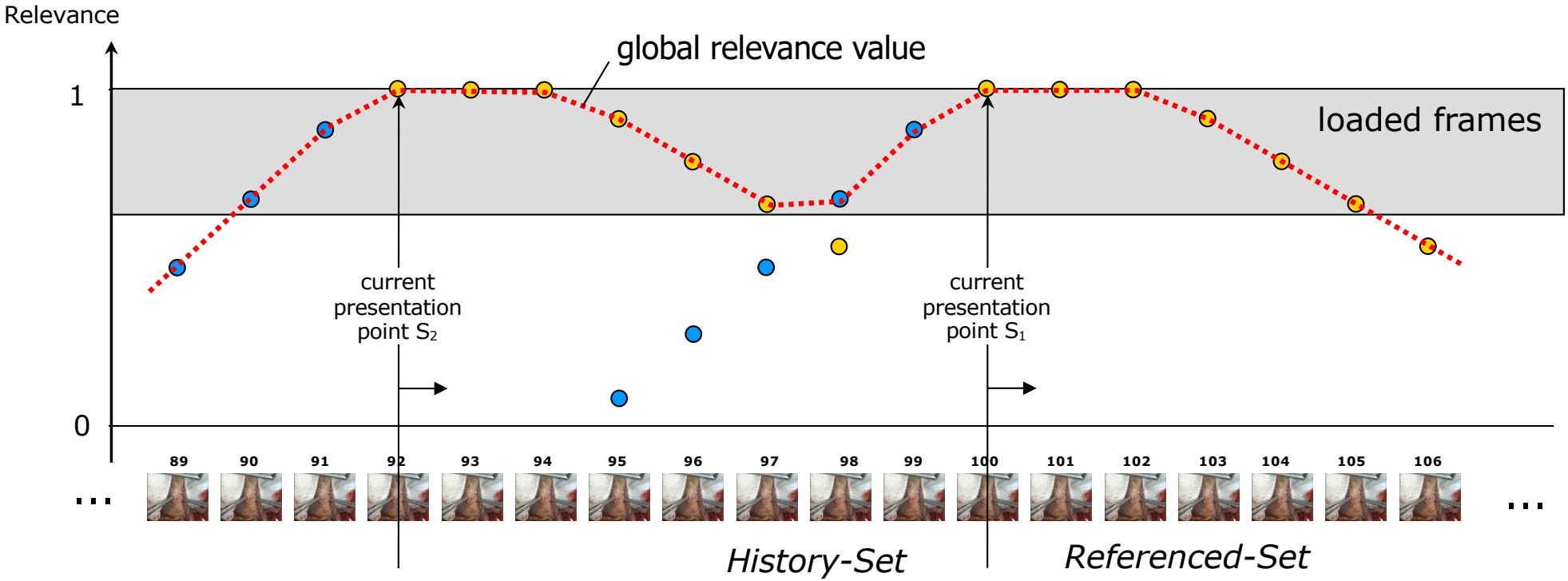
- Relevance values are calculated with respect to current playout of the multimedia stream
  - presentation point (current position in file)
  - mode / speed (forward, backward, FF, FB, jump)
  - relevance functions are configurable



Least/Most Relevant for Presentation (L/MRP)

[Moser et al. 95]

- Global relevance value
  - each COPU can have more than one relevance value
    - several viewers (clients) of the same
  - = *maximum relevance for each COPU*

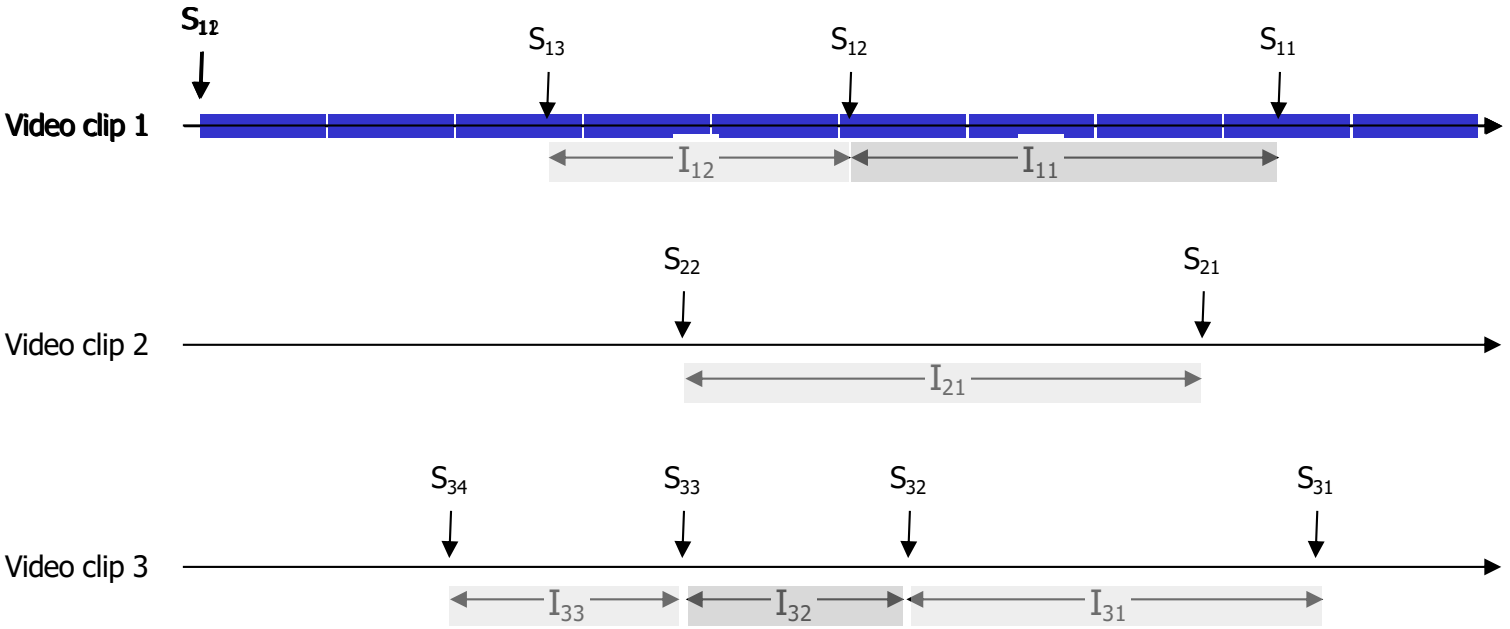


Least/Most Relevant for Presentation (L/MRP)

- L/MRP ...
  - ☺ ... gives "few" disk accesses (compared to other schemes)
  - ☺ ... supports interactivity
  - ☺ ... supports prefetching
  - ☹ ... targeted for single streams (users)
  - ☹ ... expensive (!) to execute
    (calculate relevance values for all COPUs each round)

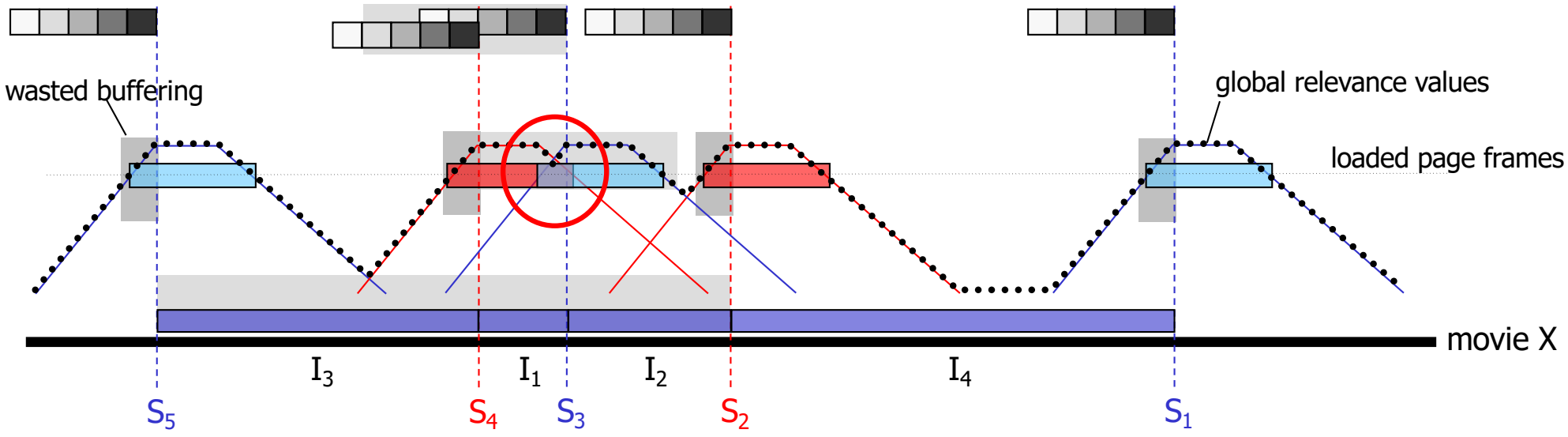
Interval Caching (IC)

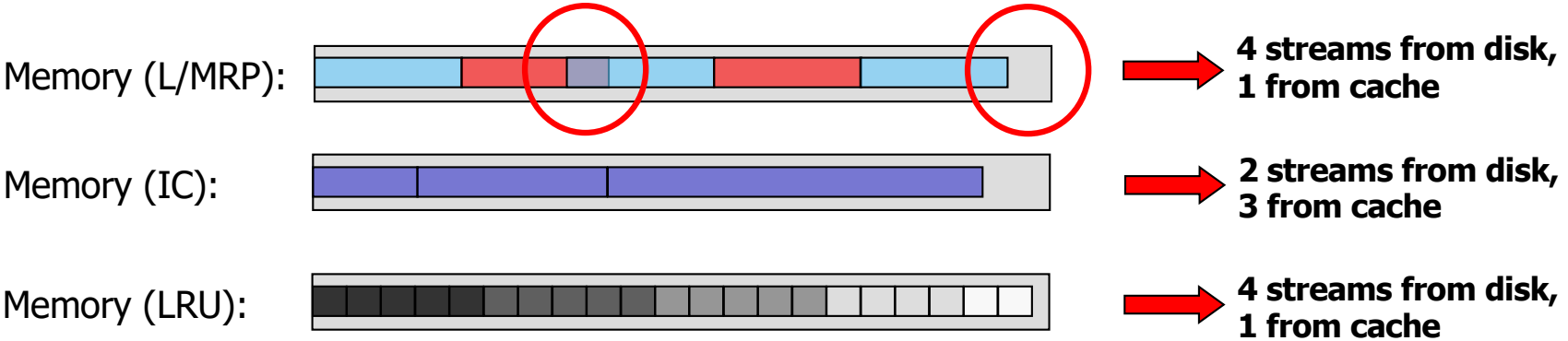
- Interval caching (IC) is a caching strategy for streaming servers
  - caches data between requests for same video stream – based on playout intervals between requests
  - *following* requests are thus served from the cache filled by *preceding* stream
  - sort intervals on length, buffer requirement is data size of interval
  - to maximize cache hit ratio (minimize disk accesses) the shortest intervals are cached first: I_{32} I_{33} I_{12} I_{31} I_{11} I_{21}



Video: LRU vs. L/MRP vs. IC Caching

- What kind of caching strategy is best (video streaming)?
  - caching effect





LRU vs. L/MRP vs. IC Caching

- What kind of caching strategy is best (video streaming)?
  - caching effect (IC best)
  - CPU requirement

LRU

```
for each I/O request
    reorder LRU chain
```

L/MRP

```
for each I/O request
    for each COPU
        RV = 0
        for each stream
            tmp = rel ( COPU, p, mode )
            RV = max ( RV, tmp )
```

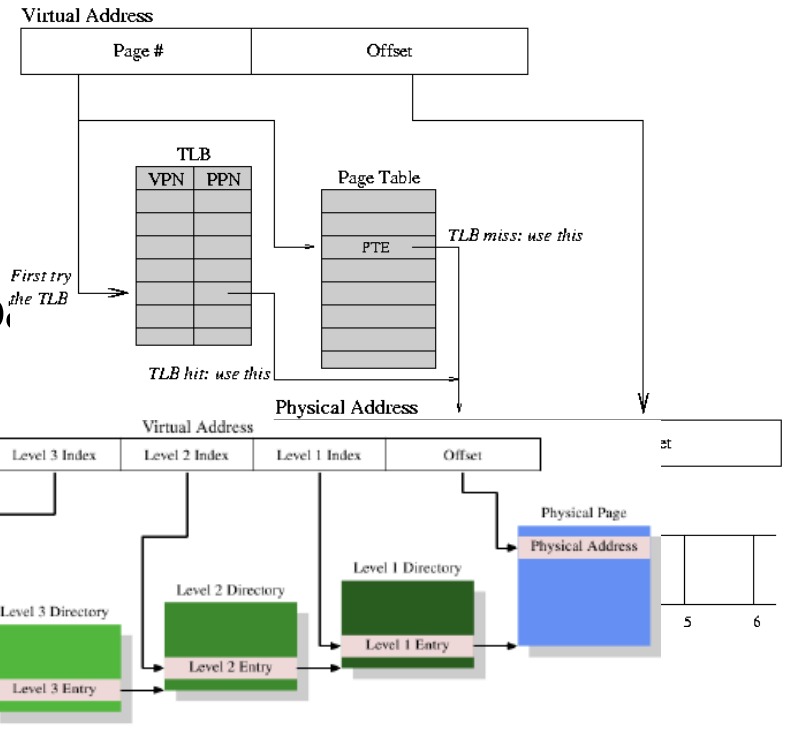
IC

```
for each block consumed
    if last part of interval
        release memory element
```

Speeding up paging...

- Every memory reference needs a virtual-to-physical mapping
- Each process has its own virtual address space (an own page table)
- Large tables:
  - 32-bit addresses, 4 KB pages → 1.048.576 entries
  - 64-bit addresses, 4 KB pages → 4.503.599.627.370.496 entries

➥ Translation lookaside buffers (aka associative memory)
  - hardware "cache" for the page table
  - a fixed number of slots containing the last page table entries



➥ Page size:
larger page sizes reduce number of p

➥ Multi-level page tables

Multi-level paging example:

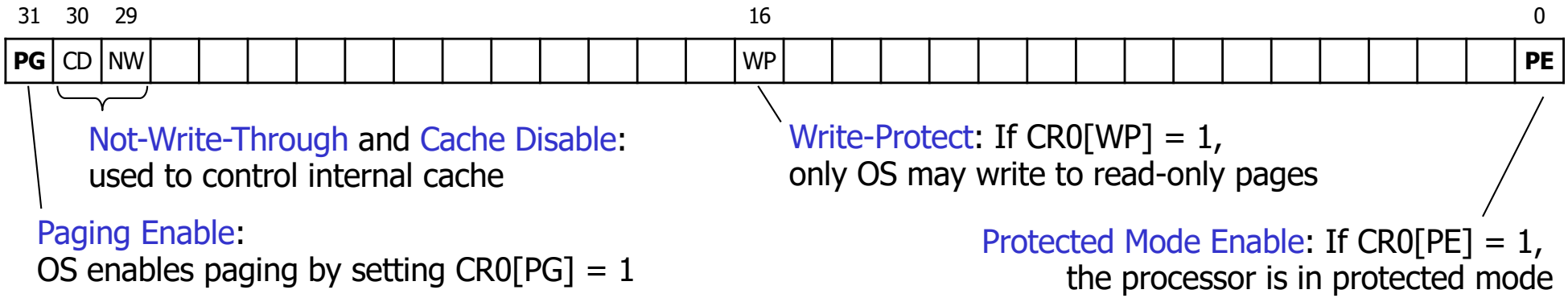
32-bit (Pentium)

Paging on Pentium

- The executing process has a 4 GB address space (2^{32}) – viewed as 1M (2^{20}) 4 KB (2^{12}) pages
  - The 4 GB address space is divided into 1 K page groups (pointed to by the 1 level table – page directory)
  - Each page group has 1 K 4 KB pages (pointed to by the 2 level tables – page tables)
- Mass storage space is also divided into 4 KB blocks of information
- Control registers: used to change/control general behavior (e.g., interrupt control, switching the addressing mode, **paging control**, etc.)

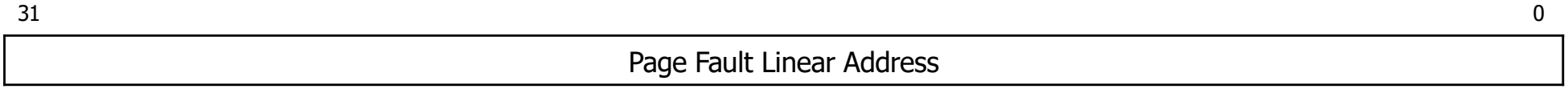
Control Registers used for Paging on Pentium

- Control register 0 (CR0):



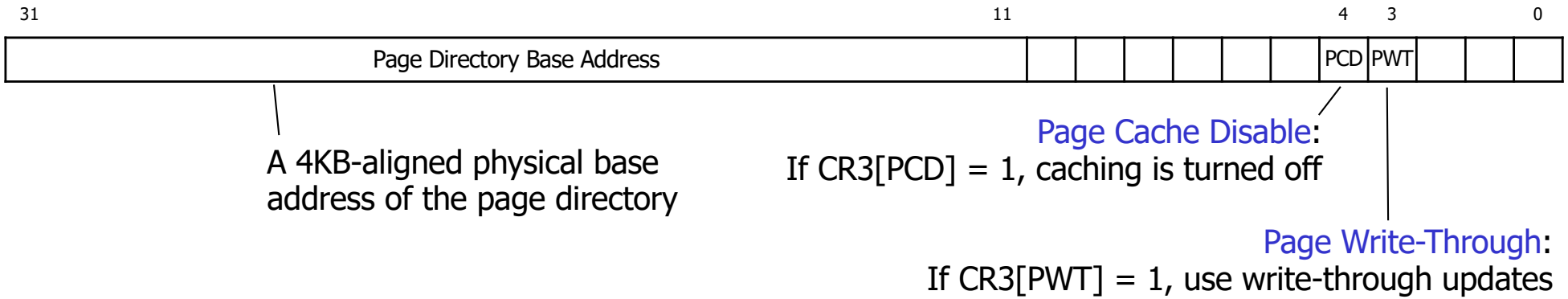
- Control register 1 (CR1) – does not exist, returns only zero

- Control register 2 (CR2)
  - only used if CR0[PG]=1 & CR0[PE]=1

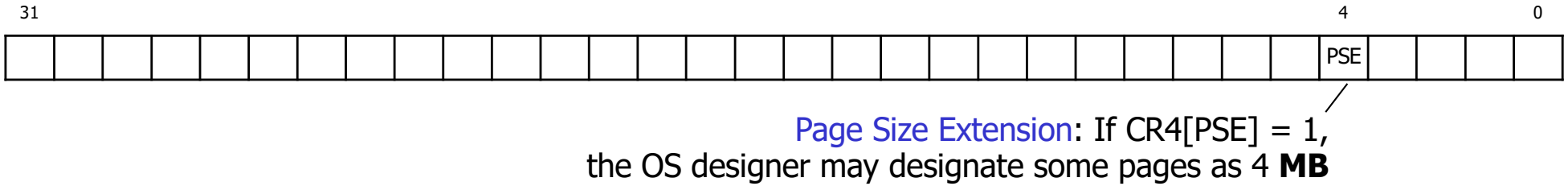


Control Registers used for Paging on Pentium

- Control register 3 (CR3) – page directory base address:
  - only used if CR0[PG]=1 & CR0[PE]=1

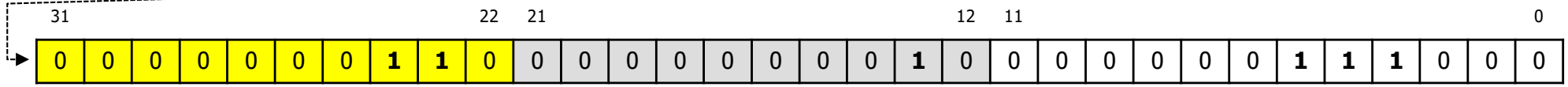


- Control register 4 (CR4):

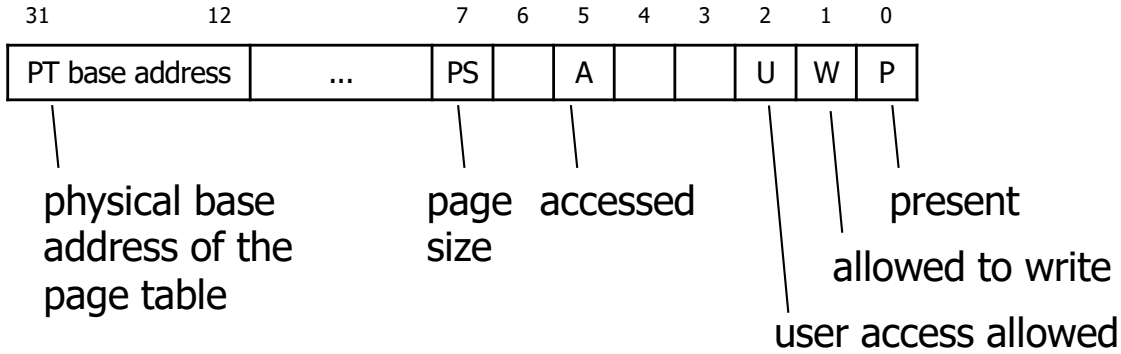


Pentium Memory Lookup

Incoming virtual address (CR2)
(0x1802038, 20979768)

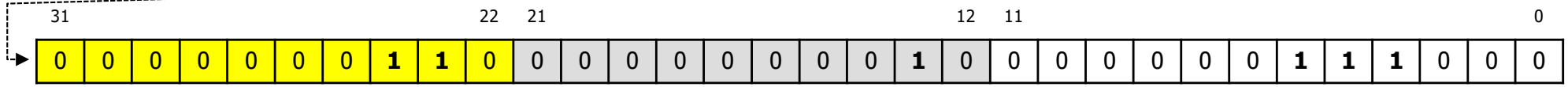


Page directory:

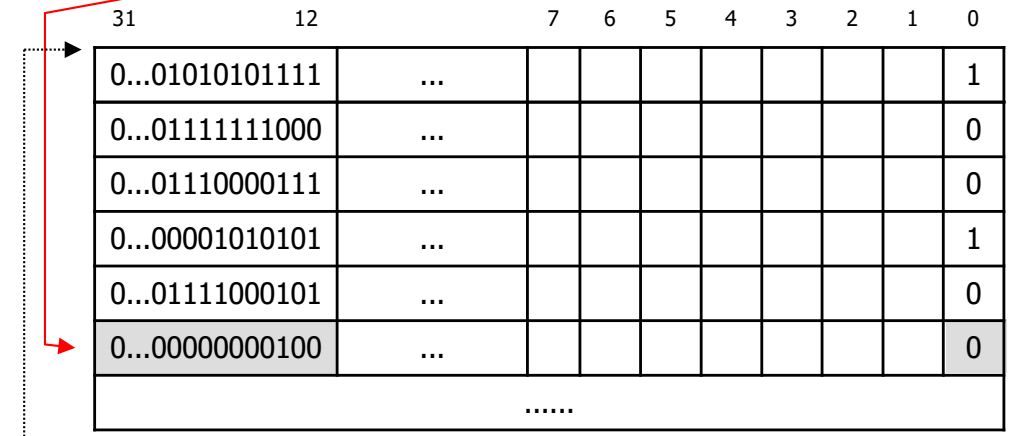


Pentium Memory Lookup

Incoming virtual address (CR2)
(0x1802038, 20979768)



Index to page directory
(0x6, 6)



CR3:

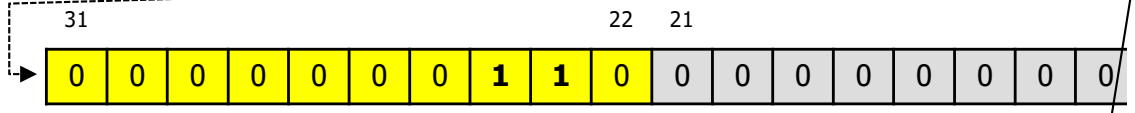
Page Directory Base Address

Page table PF:

1. Save pointer to instruction
2. Move linear address to CR2
3. Generate a PF exception – jump to handler
4. Programmer reads CR2 address
5. Upper 10 CR2 bits identify needed PT
6. Page directory entry is really a mass storage address
7. Allocate a new page – write back if dirty
8. Read page from storage device
9. Insert new PT base address into page directory entry
10. Return and restore faulting instruction
11. Resume operation reading the same page directory entry again – now P = 1

Pentium Memory Lookup

Incoming virtual address (CR2)
(0x1802038, 20979768)



Index to page directory
(0x6, 6)

31 ... 12	...	7	6	5	4	3	2	1	0
0...01010101111	...								1
0...01111111000	...								0
0...01110000111	...								0
0...00001010101	...								1
0...01111000101	...								0
0...00000000100	...								**1**
.....									

CR3:

Page Directory Base Address

Page frame PF:

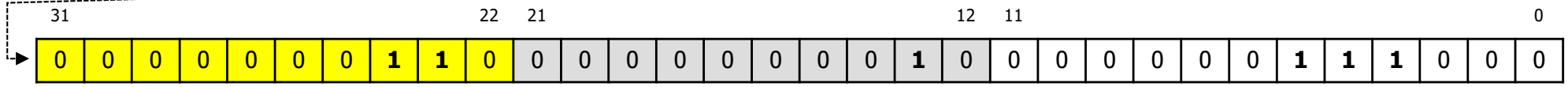
1. Save pointer to instruction
2. Move linear address to CR2
3. Generate a PF exception – jump to handler
4. Programmer reads CR2 address
5. Upper 10 CR2 bits identify needed PT
6. Use middle 10 CR2 bit to determine entry in PT – holds a mass storage address
7. Allocate a new page – write back if dirty
8. Read page from storage device
9. Insert new page frame base address into page table entry
10. Return and restore faulting instruction
11. Resume operation reading the same page directory entry and page table entry again – both now P = 1

Page table:

31 ... 12	...	7	6	5	4	3	2	1	0
0...01010101111	...								1
0...01010100000									0
0...01100110011									1
0...00010000100									1
.....									

Pentium Memory Lookup

Incoming virtual address (CR2)
(0x1802038, 20979768)



Index to page directory
(0x6, 6)

31 ... 12	...	7	6	5	4	3	2	1	0
0...01010101111	...								1
0...01111111000	...								0
0...01110000111	...								0
0...00001010101	...								1
0...01111000101	...								0
0...00000000100	...								**1**
.....									

Index to page table
(0x2, 2)

Page:

Page offset
(0x38, 56)

requested data

CR3:

Page Directory Base Address

31 ... 12	...	7	6	5	4	3	2	1	0
0...01010101111	...								1
0...01010100000									**1**
0...01100110011									1
0...00010000100									1
.....									

Pentium Page Fault Causes

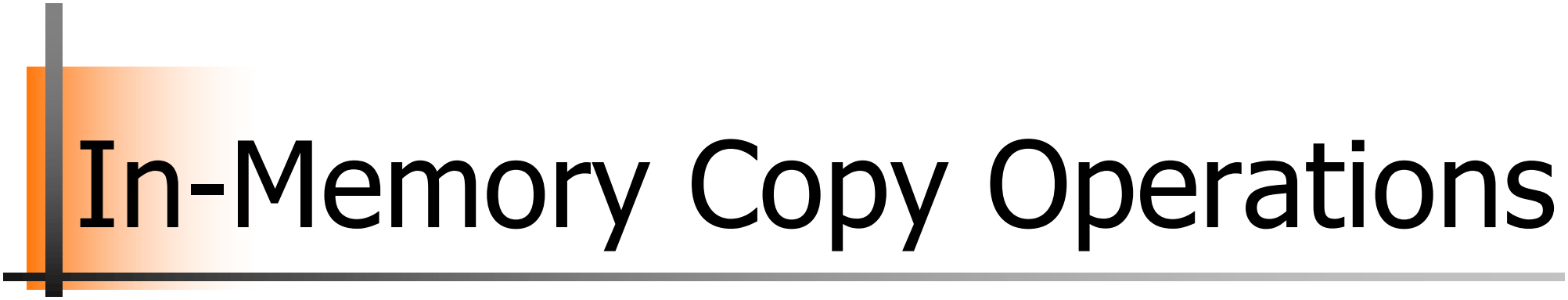
- Page directory entry's P-bit = 0:
  page group's directory (page table) not in memory
- Page table entry's P-bit = 0:
  requested page not in memory
- Attempt to write to a read-only page
- Insufficient page-level privilege to access page table or frame
- One of the reserved bits are set in the page directory or table entry

32-bit (protected/compatibility mode) vs. 64-bit (long mode)

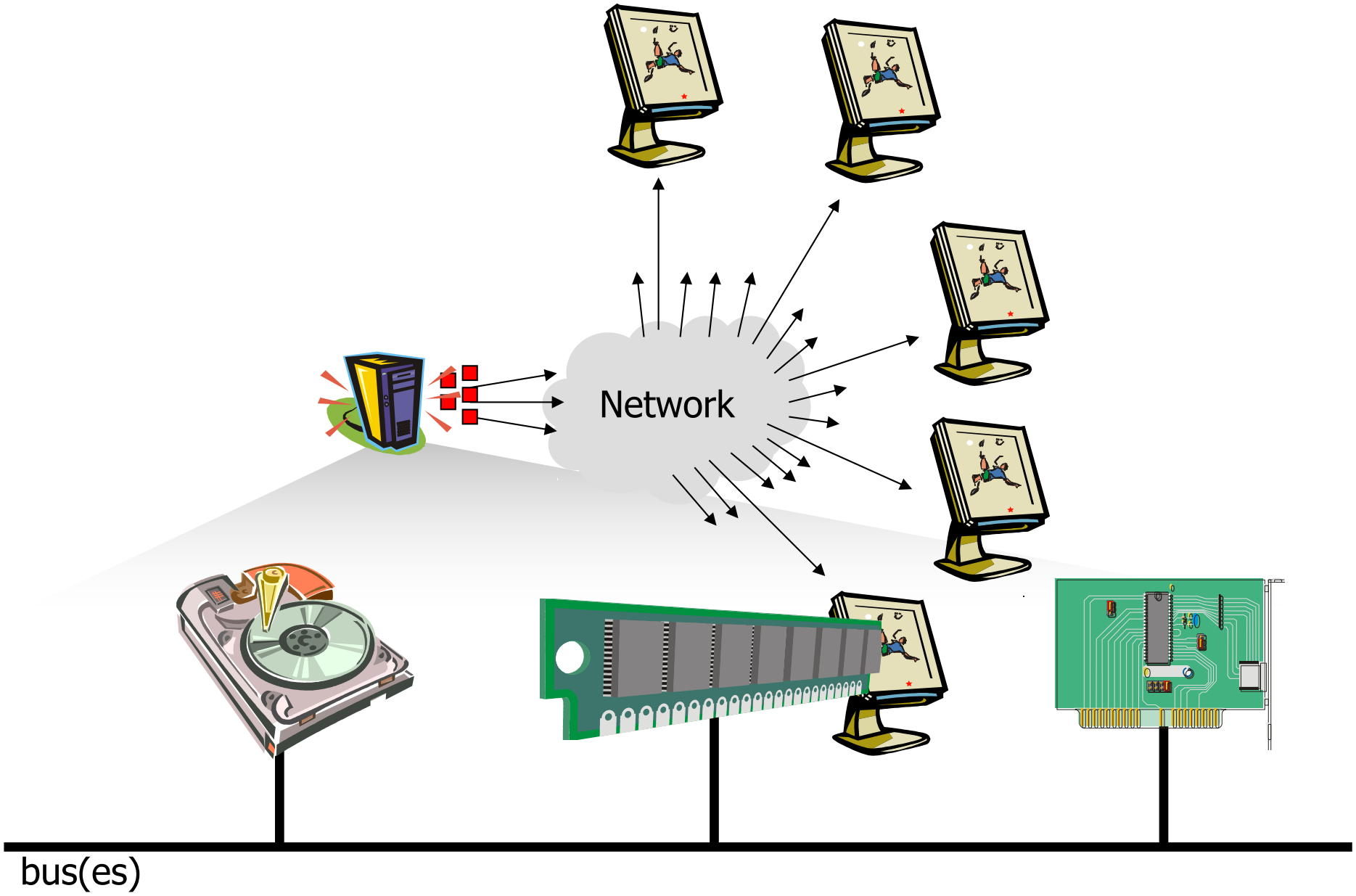
- Virtual address space *could in theory* be 64-bit (16 EB) (vs. 4 GB for 32-bit)
- Processors allow addressing of only a portion of that
- Most common processor implementations allow 48-bit (256 TB)
- An address is now 8 bytes (64-bit) (vs. 4 byte for 32-bit)
- Each 4-KB page can hold 2^9 page-table entries (vs. 2^{10} for 32-bit), need 9 bit index for each level (vs. 10 for 32-bit)

- Virtual address:

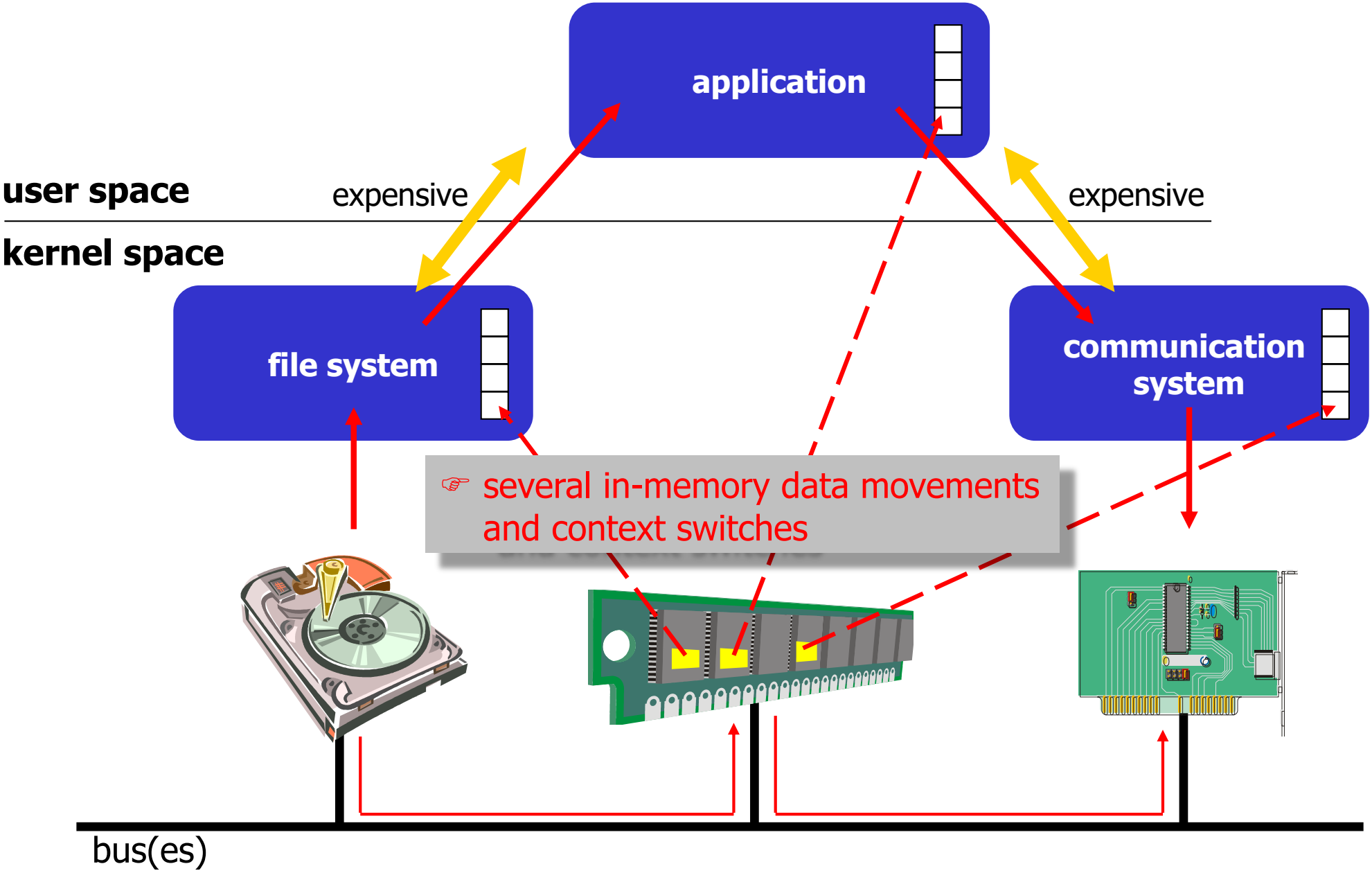
63-48	47-39	38-30	29-21	20-12	11-0
unused	page map level 4	page directory pointer index	page directory index	page table index	page offset


In-Memory Copy Operations

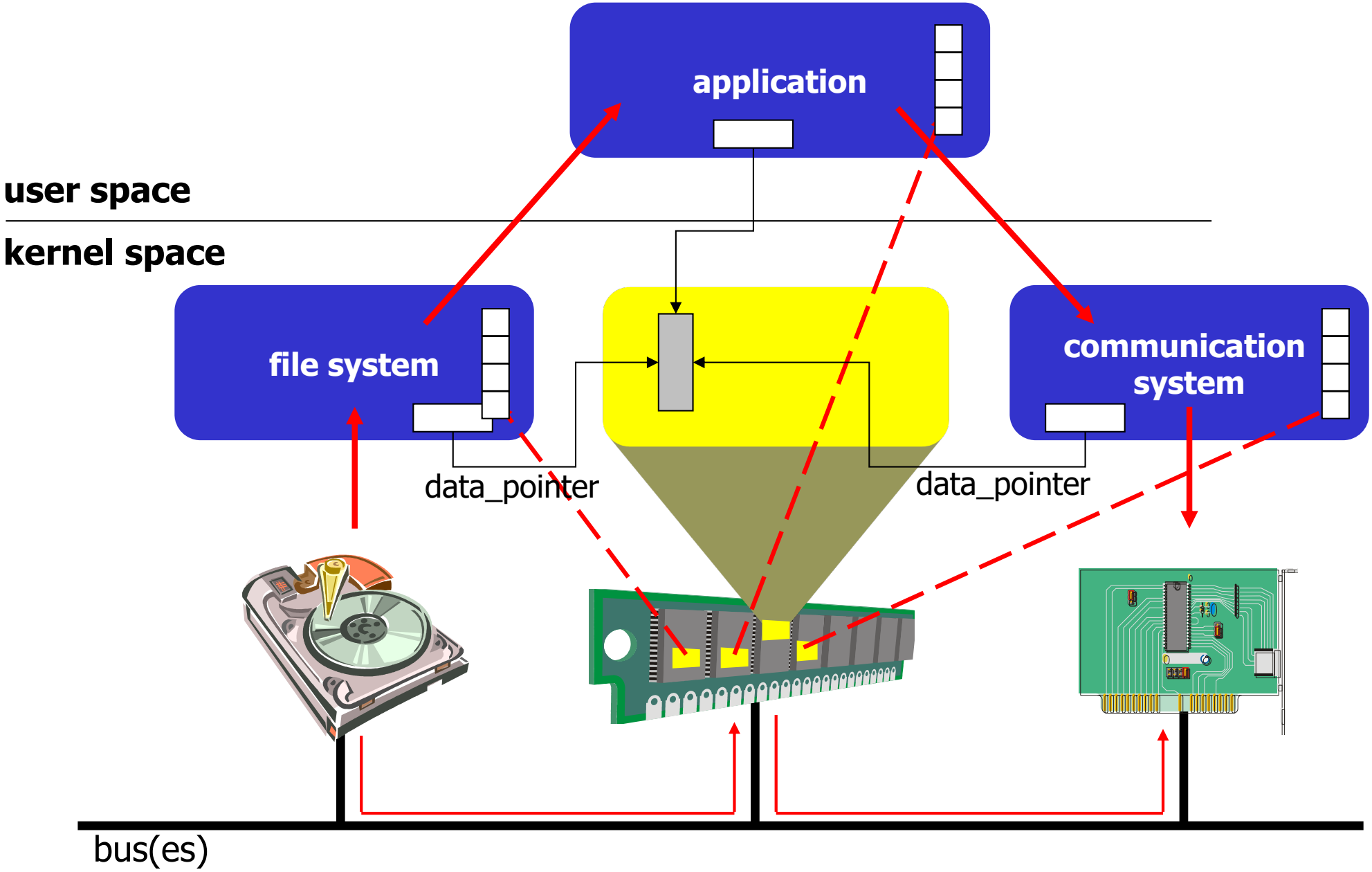
Delivery Systems



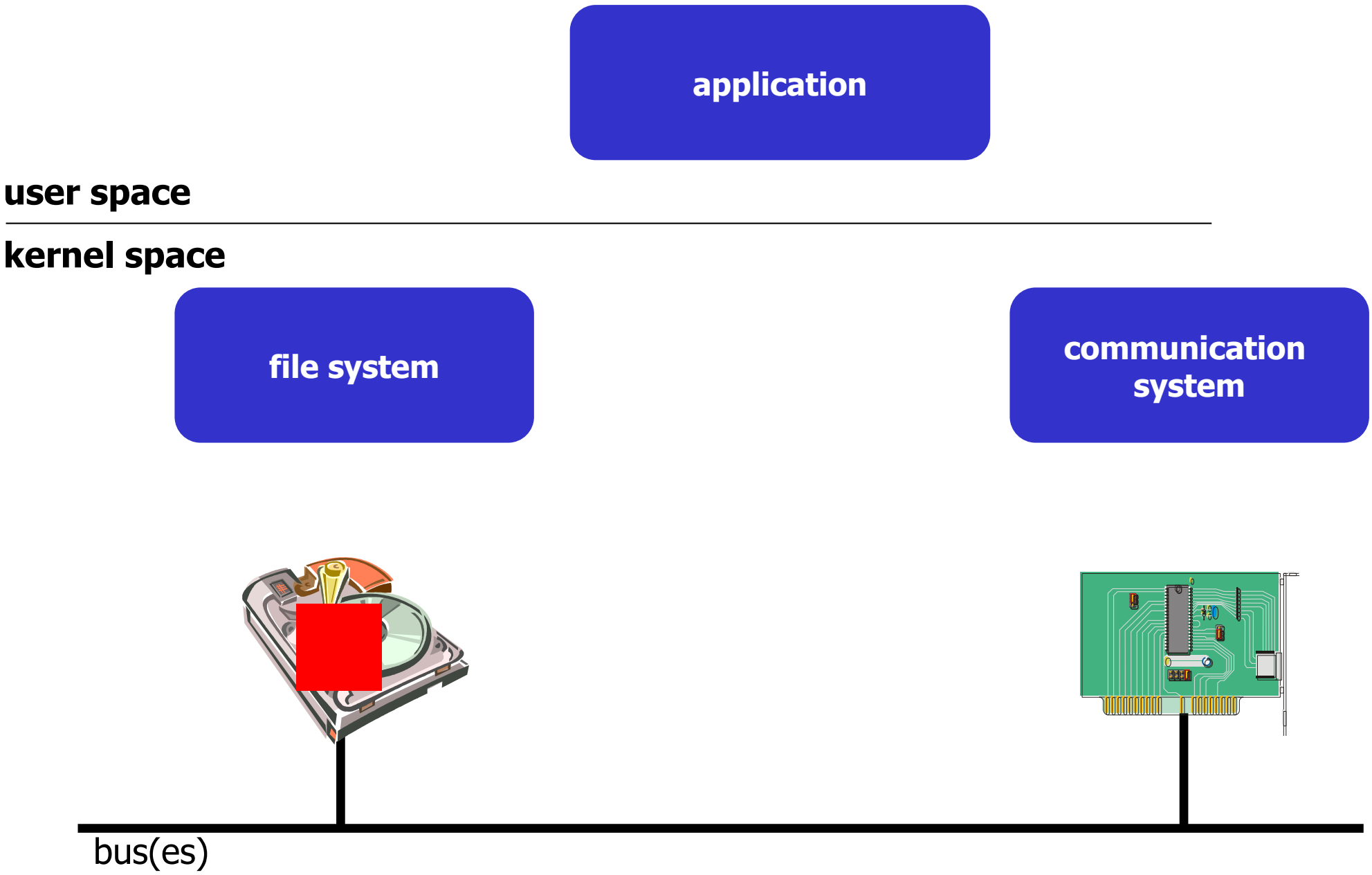
Delivery Systems



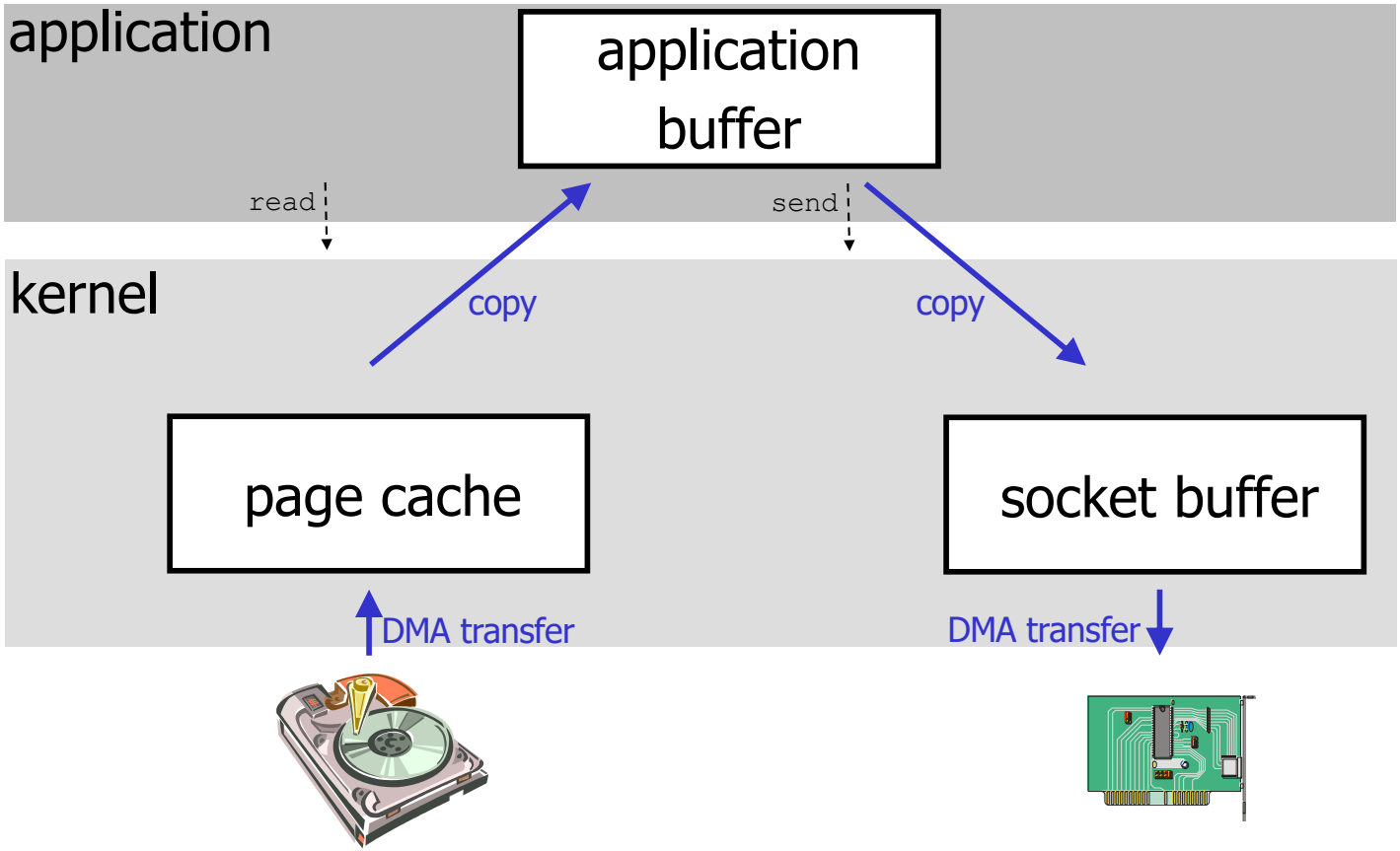
Zero-Copy Data Paths



Content Download

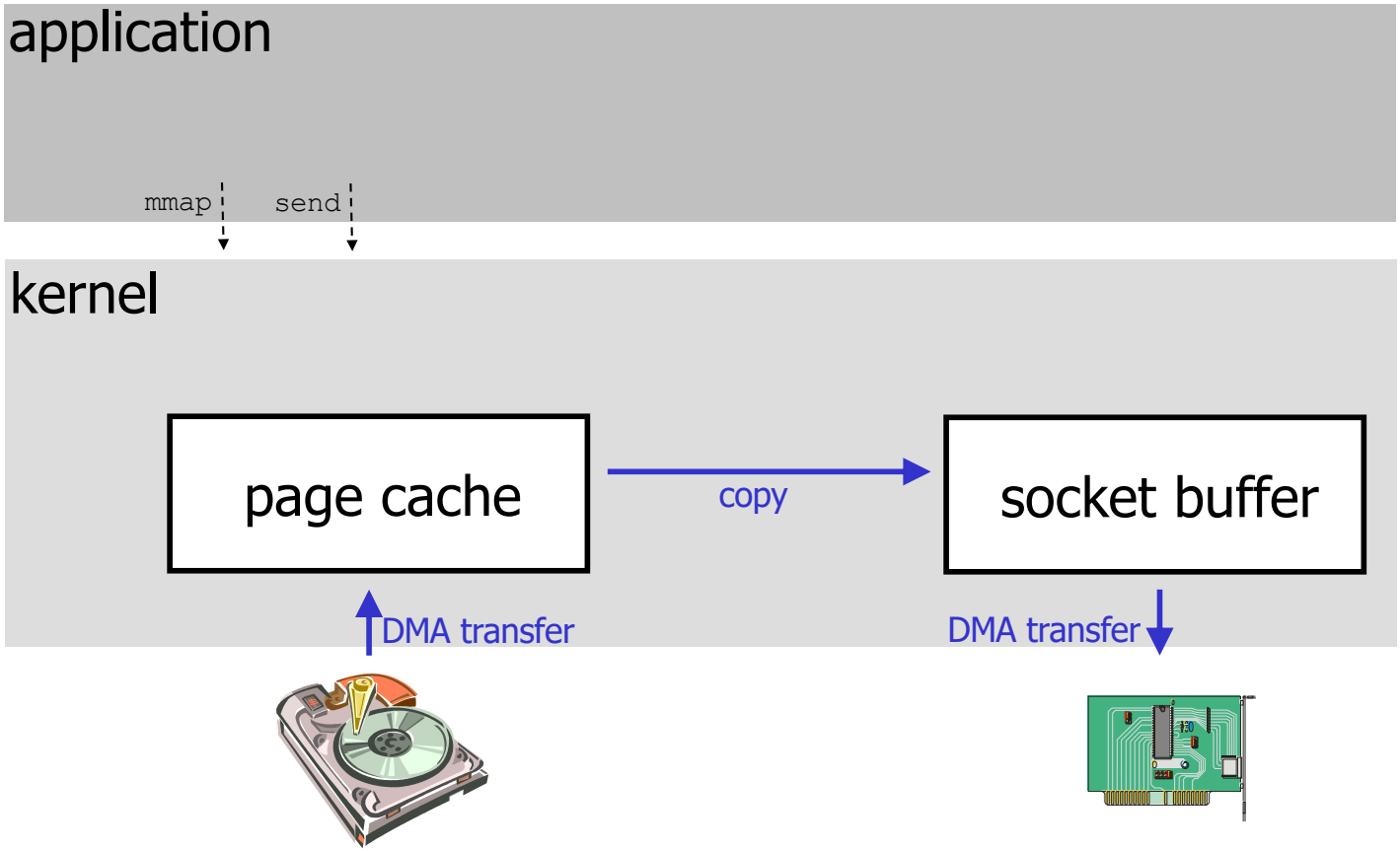


Content Download: `read` / `send`



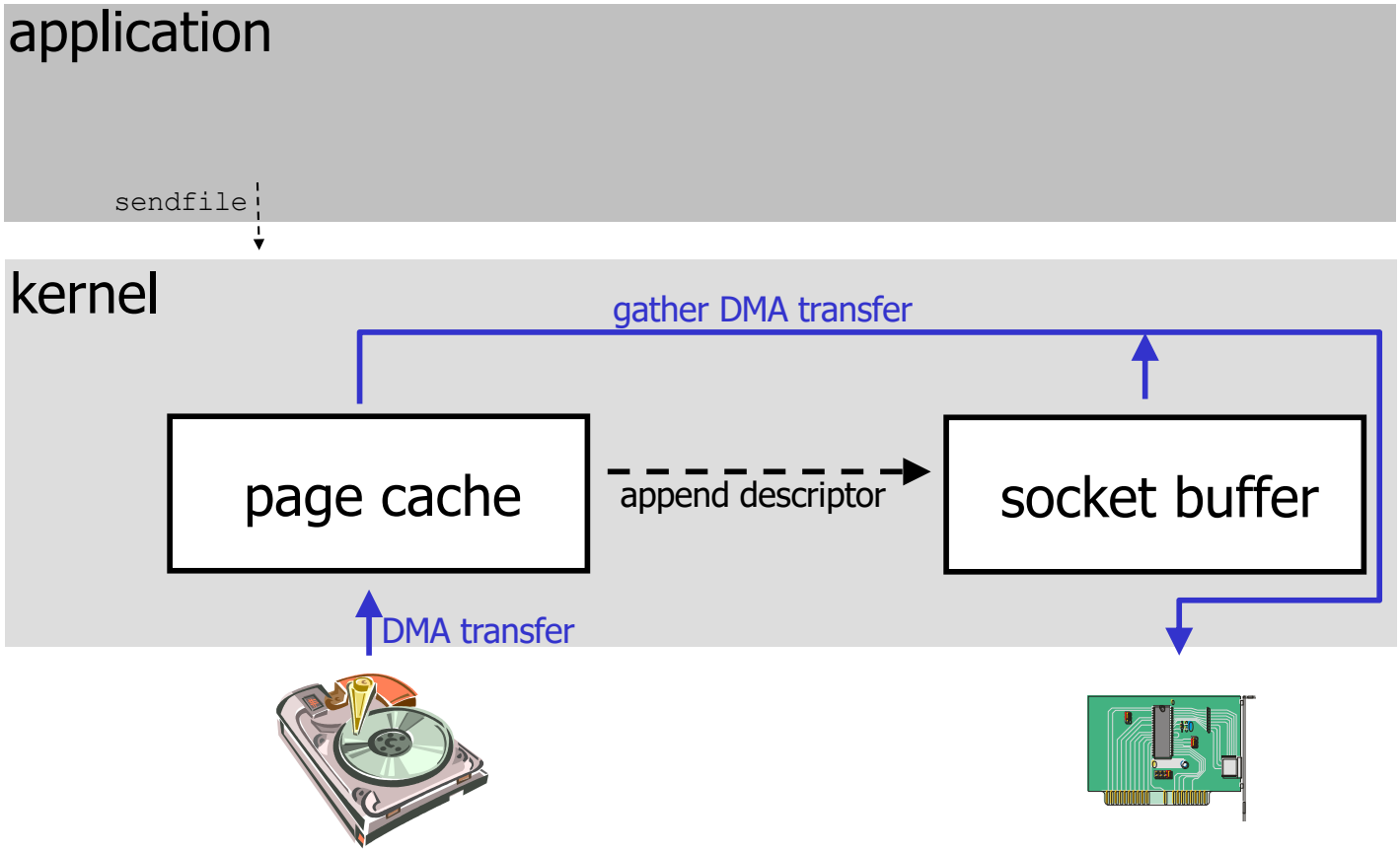
- **2*n*** copy operations
- **2*n*** system calls

Content Download: mmap / send



- ***n*** copy operations
- **1 + *n*** system calls

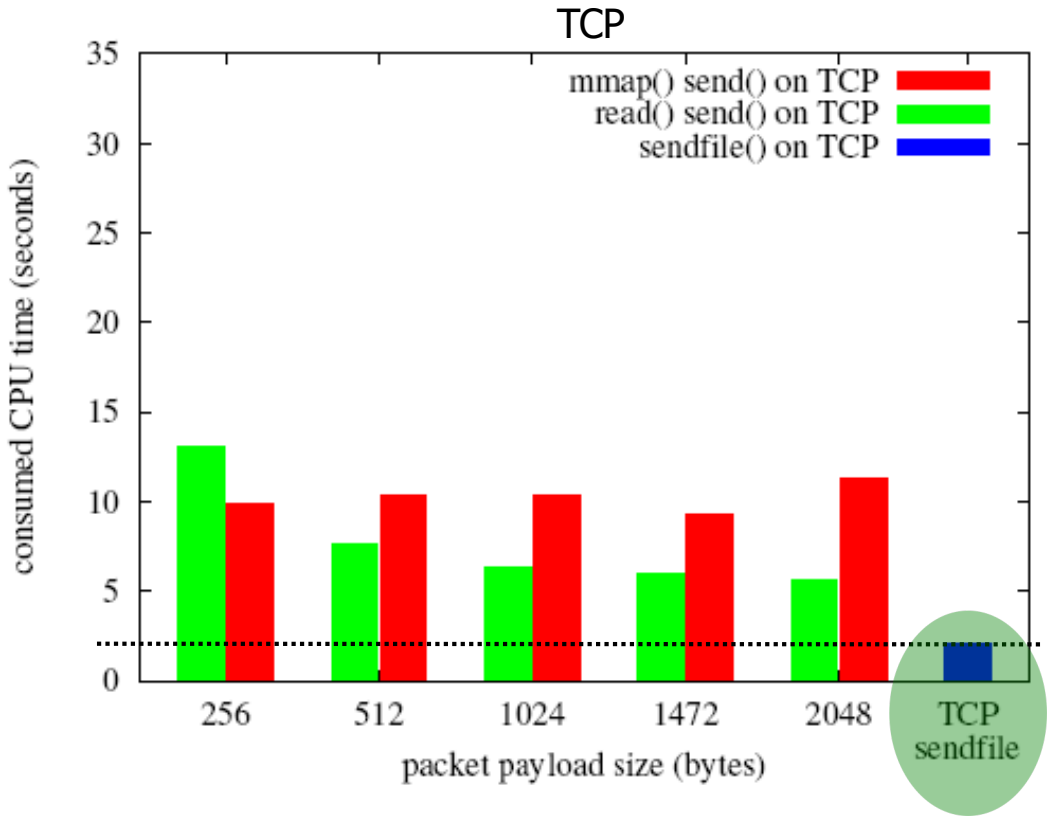
Content Download: `sendfile`



- ***0*** copy operations
- **1** system calls

Content Download: Results

- Tested transfer of 1 GB file on Linux 2.6



Summary

- Memory management is concerned with managing the systems' memory resources
  - allocating space to processes
  - protecting the memory regions
  - in the real world
    - programs are loaded dynamically
    - physical addresses are not known to program – dynamic address translation
    - program size at run-time is not known to kernel
- Each process usually has text, data and stack segments
- Systems like Windows and Unix use virtual memory with paging
- Many issues when designing a memory component