Operating system concepts

Memory management

Memory management

- ***RAM*** (Random Access Memory) is subject to memory management subsystem (MM),
  - RAM is a (temporary) data storage many times faster than other storage media (e.g. hard disk)
  - other storage media is used as:
    - permanent storage (disk, cd-rom, …) for files
    - supplementary storage (when there is not enough RAM)

- Everything in computer system passes through memory
  - operating system code and data must first be loaded into memory at system startup
  - to start programs, they must be loaded into memory first
  - input and output data for programs are loaded or created in memory (and loaded or stored to other media)
  - data cache (from slower devices) is placed in memory

Storage for operating system code and data

- Must be protected from user programs (threads)

- Should be available only to operating system operations (mostly only for kernel functions)

- Preferably always resident in RAM
  - it's frequently used
  - kernel functions must be fast
    - while in kernel function, interrupts are disabled!
    - if some data or code must be loaded for kernel operations, this will significantly prolong function duration

- Main OS data parts: data for handling threads/processes, memory, I/O management, network subsystem/data, file system tables/data/cache, …

Storage for programs

- Program loaded into memory becomes a *process*

- Memory used by the process is divided into:
  - code segment – program instructions (processor instr.)
  - data segment – variables, heap (dynamic memory)
  - stack – required for many purposes (e.g. subroutine calls)

- For process management, *process descriptor* must be created in kernel space
  - contains all data for process: ID, priority, scheduling policy, used memory locations, file descriptors used by process, I/O caches, …

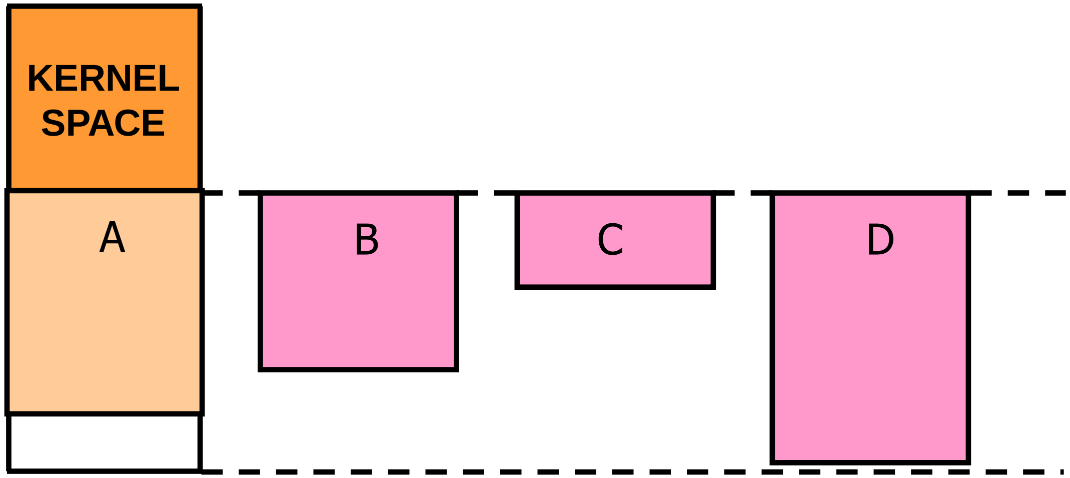
- Processes must be/should be protected from each other

Memory management problems

- How to organize memory layout?
  - Where to put kernel/program data/code/…
- How to protect memory segments
  - from unprivileged usage (e.g. from software errors, from malicious code)
  - access from another process
- What if the whole program can't fit into memory?
- What are hardware requirements for MM?
- *Dynamic memory allocation* – operations like *malloc* and *free* (*new, delete*) are not covered in this presentation!

Simple systems

- Simple systems:
  a) single program (OS and application are coupled)
  b) single program at a time systems
    - e.g. simple devices as handhelds, mobile phones, …
- Divide memory into (b) only):
  - ☐ OS part
    - load OS data and code, and reserve space for rest
  - ☐ application part
    - load program code, data and create stack



Simple systems

- When changing programs (processes) “big” context switch occurs:
    - one program is removed form memory (and stored on other media if not finished)
    - other program is loaded in memory
- Usable ***only*** when very long context switching time is acceptable
    - only for *simple systems*
- Other systems require that more than one program resides in memory (at least their essential parts)

Memory management requirements

- OS and (some) programs must be in memory

- More than one program should be simultaneously present in memory
  - if required, “big” context switch can happen “in the background”, e.g. performed by DMA device

- Processes should be separated – protected from each other
  - threads from same process can share its process address space – use it for communication…
  - threads from different processes should be separated

Memory management requirements

- Mechanisms for executing programs that do not fit completely in available memory
- Fragmentation should be minimal
- Hardware requirements should be minimal
- Transparent for programmers – don’t require special procedures for using memory

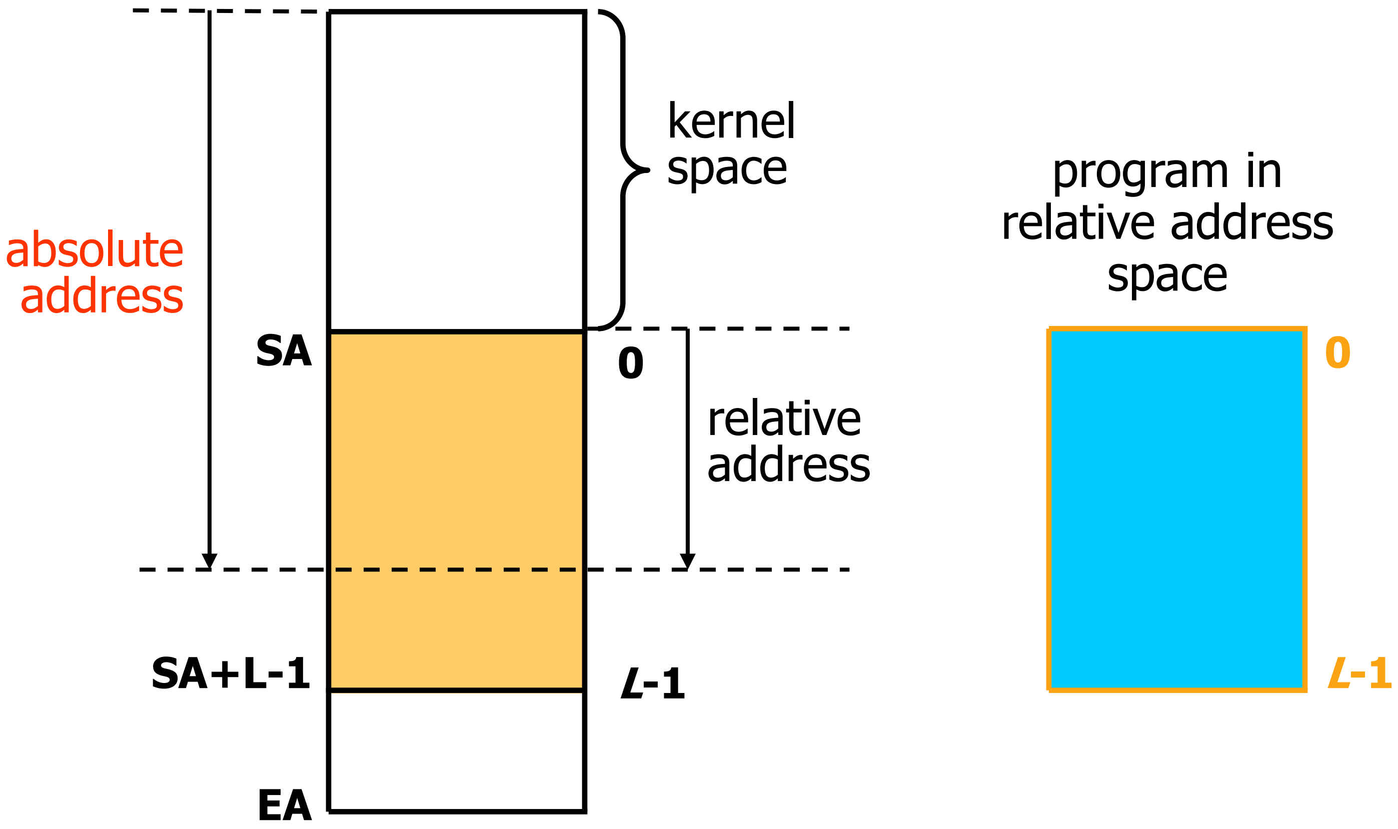
Memory management techniques

- Static memory management
  - divide memory in partitions (one for each application)
  - may use hardware support (for protection)

- Dynamic memory management
  - use dynamic memory allocation for process placement
  - *require* hardware support (for address translation)

- Virtual memory; paging
  - divide programs into *pages*
  - divide memory into *frames*
    - (frame size and page size is equal)
  - load pages into frames
  - translate relative address (from process perspective) to physical address using tables and **hardware** translators

Relative and absolute (physical) addresses



Relative – absolute addressing, example

program (on hard drive, before starting, relative addresses)

```
   0  (start)
      .
  20  LDR R0, (100)
  24  LDR R1, (104)
  28  ADD R2, R0, R1
  32  STR R2, (120)
  34  B 80
      .
      .
  80  CMP R0, R3
      .
      .
 100  DD 5
 104  DD 7
      .
 120  DD 0
```

process (loaded at start address = 1000, absolute addresses)

```
1000  (start)
      .
1020  LDR R0, (1100)
1024  LDR R1, (1104)
1028  ADD R2, R0, R1
1032  STR R2, (1120)
1034  B 1080
      .
      .
1080  CMP R0, R3
      .
      .
1100  DD 5
1104  DD 7
      .
1120  DD 0
      .
1500  (top of stack)
```

process (loaded somewhere, but still in relative addresses)

```
   0  (start)
      .
  20  LDR R0, (100)
  24  LDR R1, (104)
  28  ADD R2, R0, R1
  32  STR R2, (120)
  34  B 80
      .
      .
  80  CMP R0, R3
      .
      .
 100  DD 5
 104  DD 7
      .
 120  DD 0
      .
 500  (top of stack)
```

Static memory management - partitions

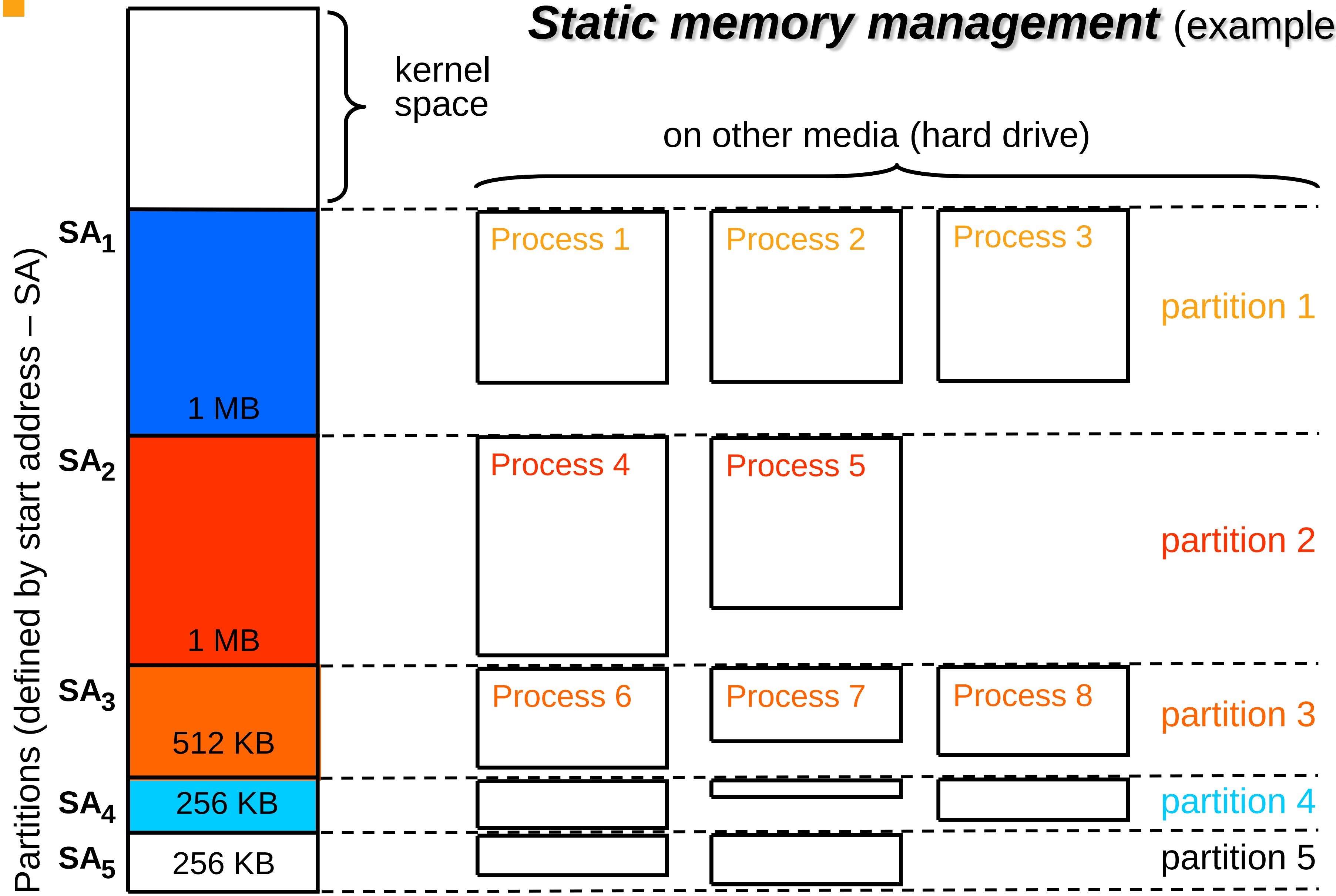
- Memory reserved for programs is divided into *partitions* with same or different sizes

- For every partition a set of programs are prepared on secondary storage (processes)
  - or just one which is permanently loaded into it

- If active process (from one partition) is finished or blocked – another one from other partition is activated
  - while the other is running, “big” context switch can be performed in first partition, e.g. using DMA
    - no “down” time for the processor!

- No special hardware support is required!

Static memory management (example)

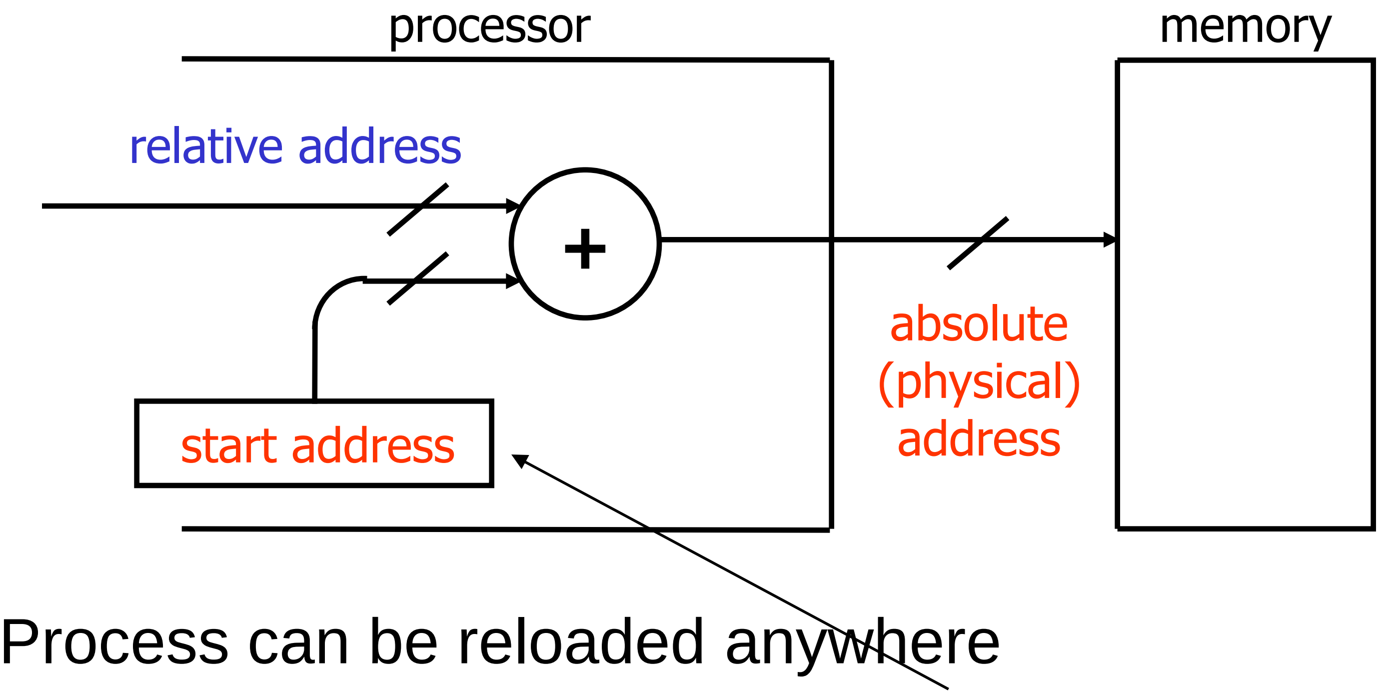


Static memory management problems

- Protection isn’t available
  - no guarantees if hardware support isn’t present
- Fragmentation
  - *internal* – some programs may not use all partition space
  - *external* – all processes allocated to the same partition may be blocked – still, the partition can’t be used by other processes!
- Can’t execute processes that don’t fit in memory (in the largest partition)

Dynamic memory management

- Processes always remain in relative address space
- Requires hardware support:
  - *adder* that will add *start address* to relative address given by the program



- Process can be reloaded anywhere
  - base register must be loaded with *start address* of the memory segment where the process is loaded

Dynamic memory management - summary

- Better than static
  - less internal and external fragmentation
  - process stay in relative address space

- With additional comparators – memory protection
  - basic *memory protection unit*

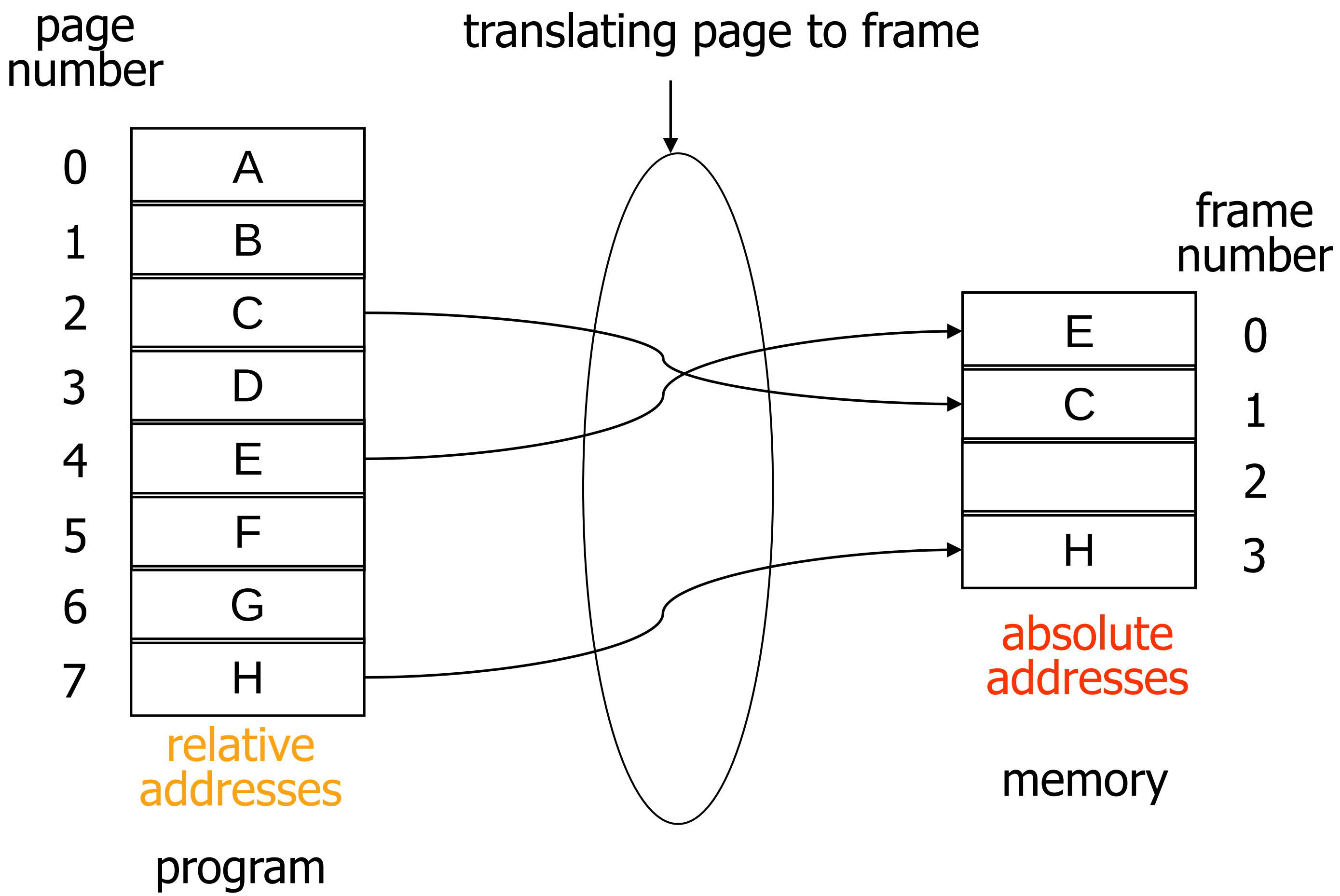
- Problems
  - fragmentation:
    - in dynamic environment memory might become fragmented - a program might not fit into largest available segment because of fragmentation

  - still can’t execute processes that don’t fit in memory

Virtual memory (VM) and paging

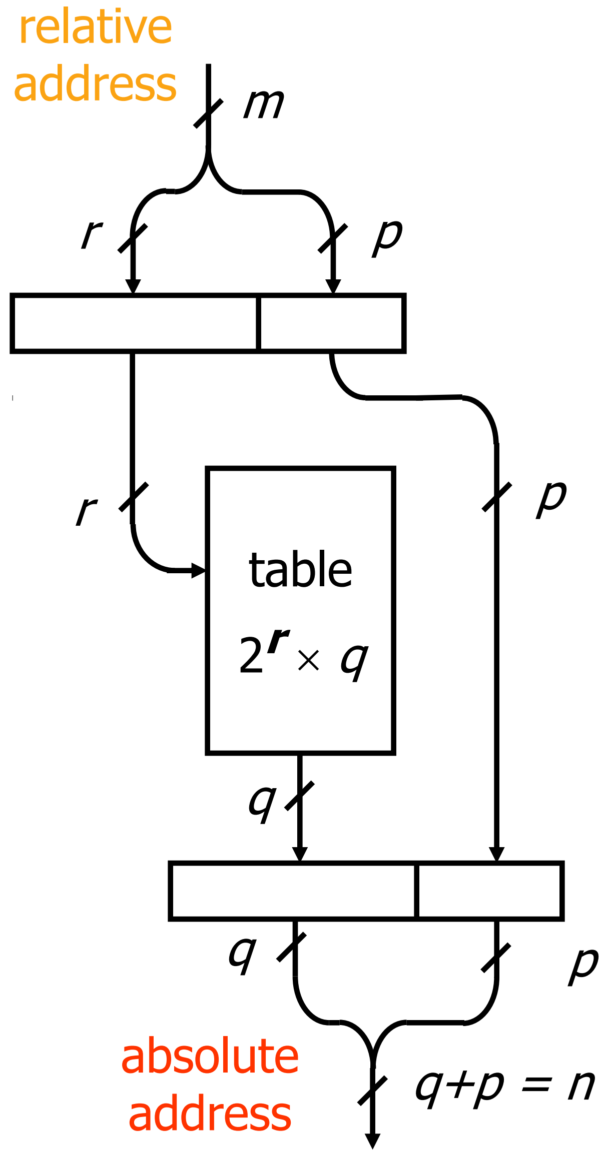
- MM is not simple !
- To solve all problems a sophisticated hardware must be used
- Basic ideas:
  - divide programs into *pages*, memory into *frames*
    - one page fits into one frame
  - load into memory only *parts of programs* that are required
    - at run time *load pages* that are required (demanded)
    - others are stored on secondary storage (hard drive)
    - if more memory is available, all pages are loaded in memory (faster)
  - program uses relative addressing, process *stays relative*
  - protect process and kernel for unintended access by translation mechanism

Virtual memory – concept

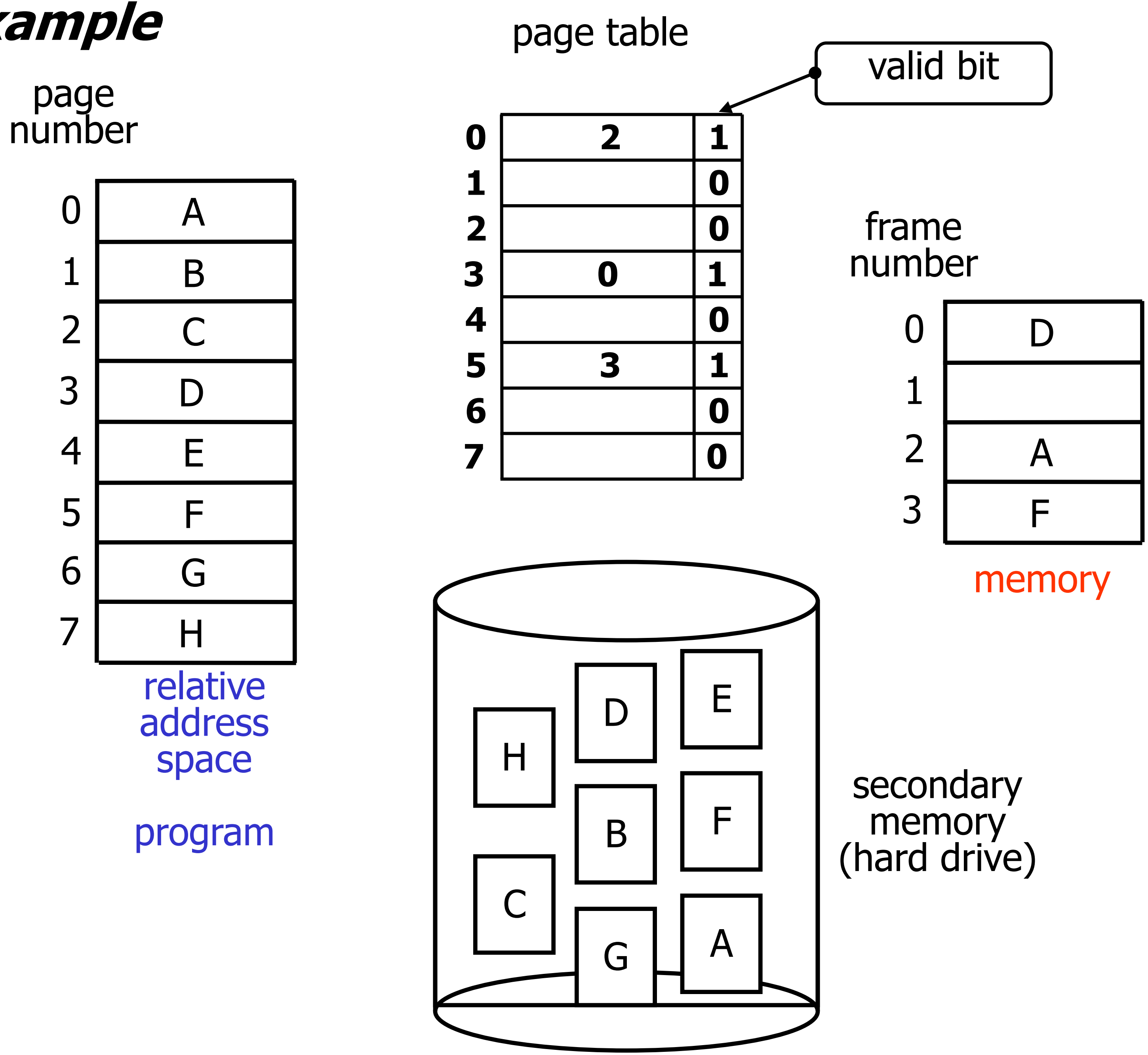


Address translation

- Relative to absolute address
  - relative address length: *m* bits
  - absolute address length: *n* bits
  - generally *m* might be different from *n*
- Relative address consists of:
  - page number: *r* bits
  - location inside page: *p* bits
- Page identification is used to translate page number to *frame number*
  - translation table is used
  - hardware based translation



Example



page number

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

relative address space

program

page table

valid bit

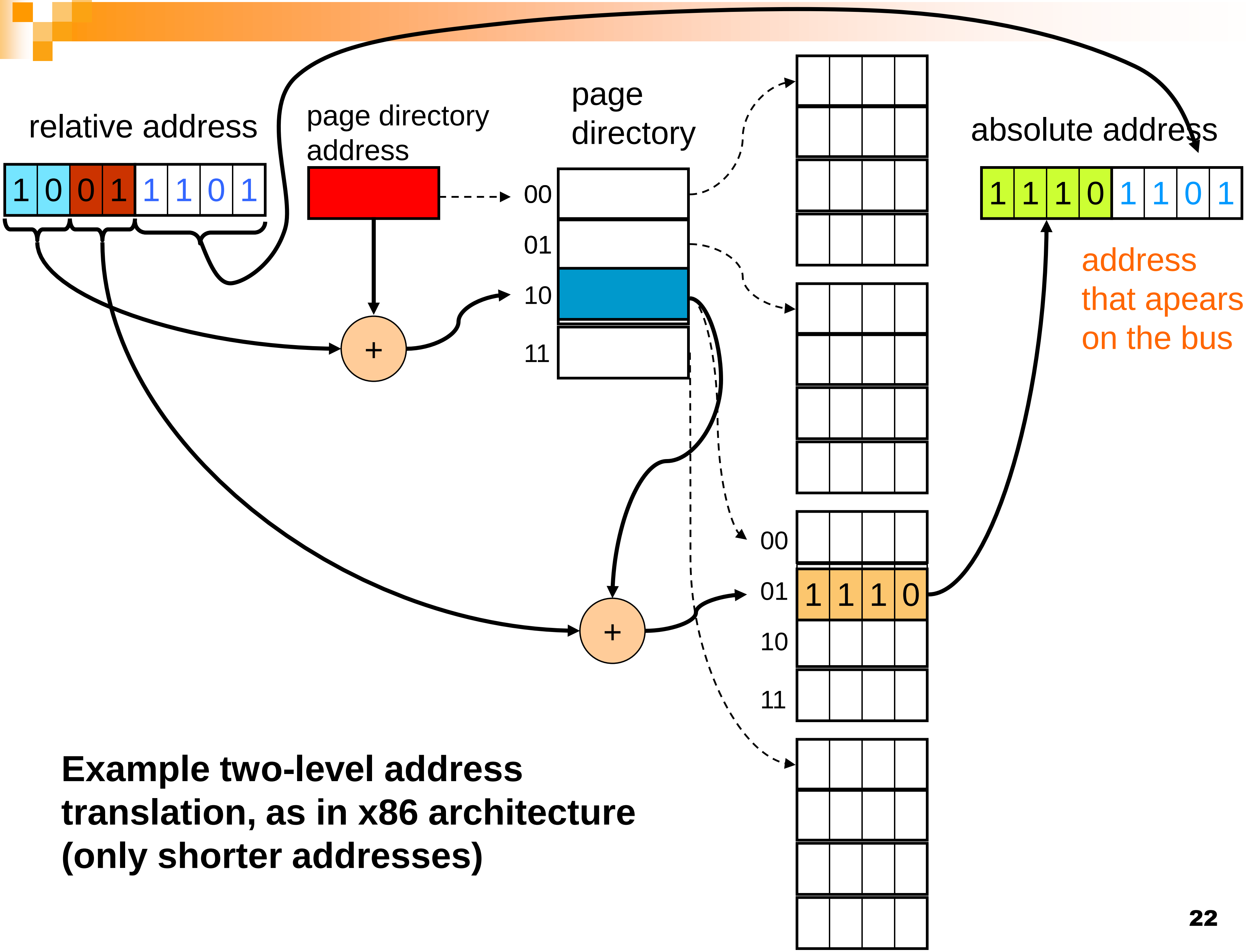
0	**2**	**1**
1		**0**
2		**0**
3	**0**	**1**
4		**0**
5	**3**	**1**
6		**0**
7		**0**

frame number

0	D
1	
2	A
3	F

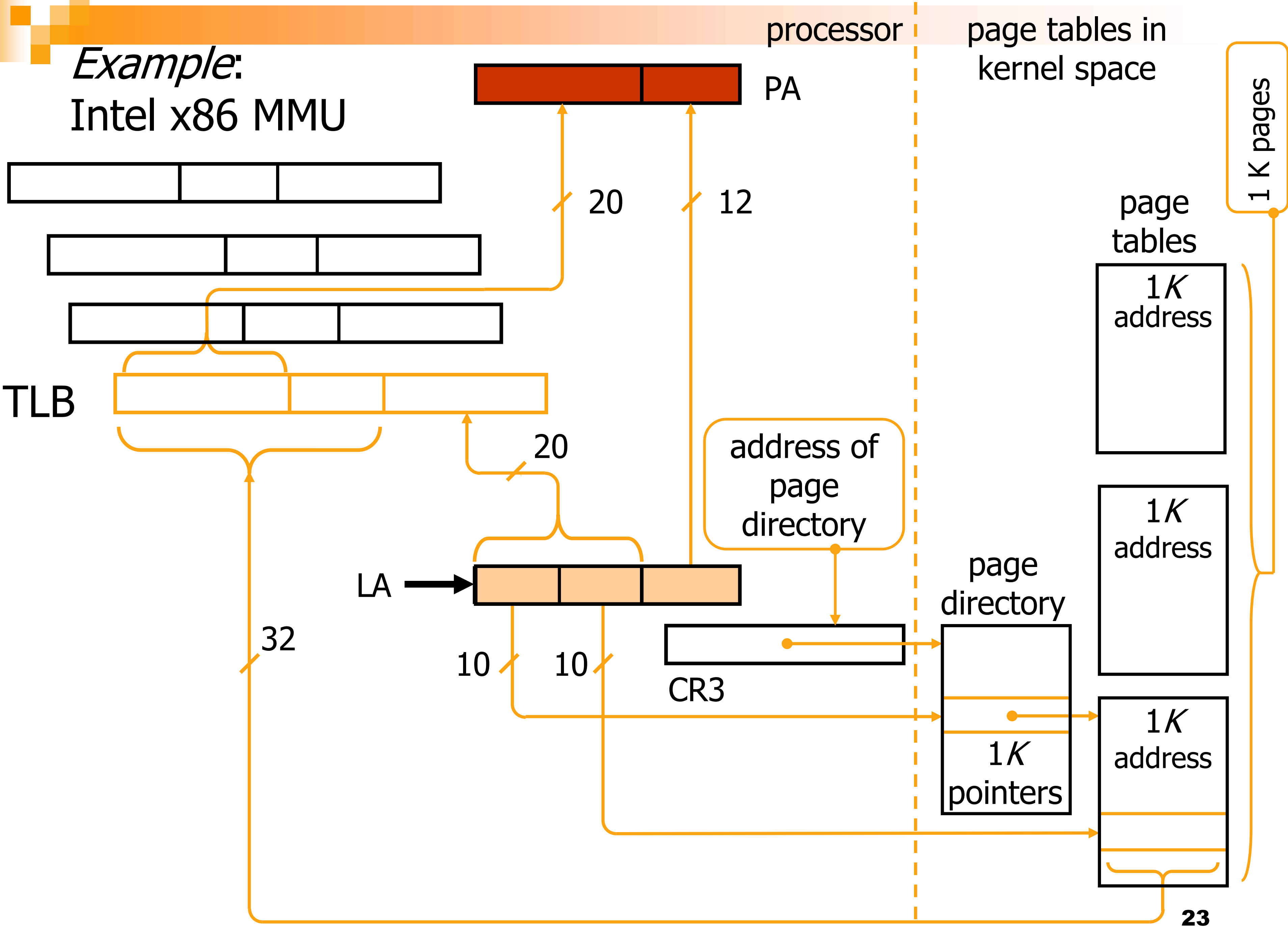
memory

secondary memory (hard drive)



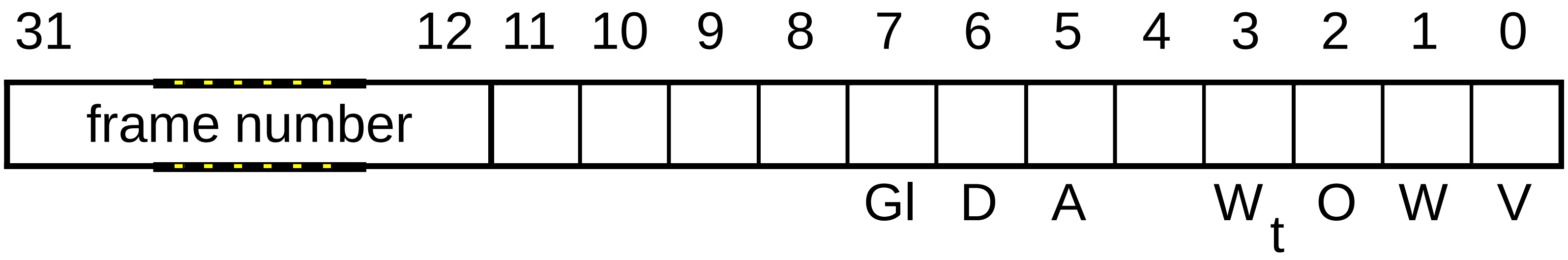
Example two-level address translation, as in x86 architecture (only shorter addresses)

Example: Intel x86 MMU



Page table

- For every page used by the process there is a row in the process page table:
  - map page to frame: frame number
  - flags; example for x86:



31 12	11	10	9	8	7	6	5	4	3	2	1	0
frame number												
					Gl	D	A		W_t	O	W	V

Flags:

V	present	A	accessed
W	write protect	D	dirty
O	for operating system	Gl	global
W_t	write throught		

  - secondary storage address (if not in memory)

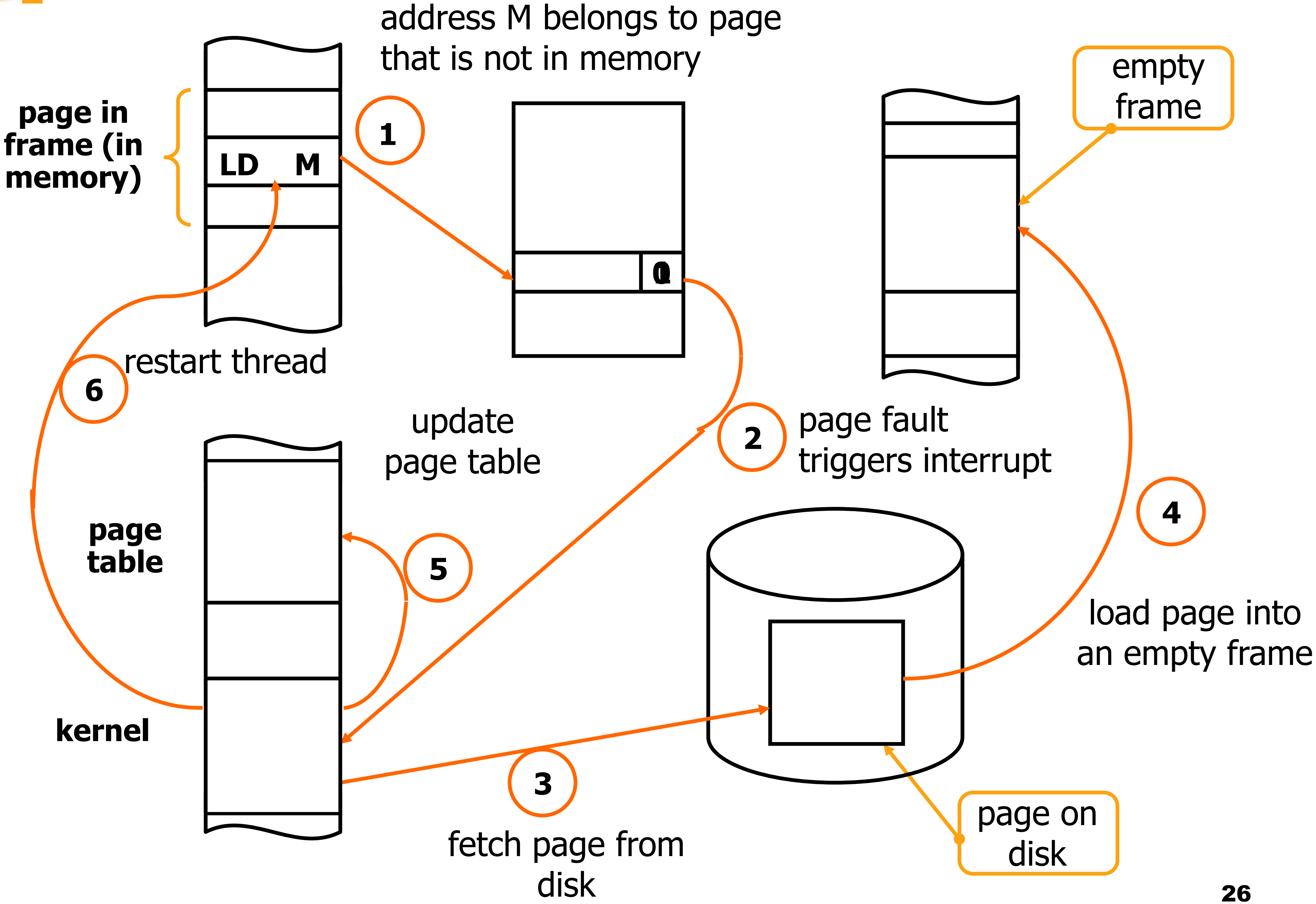
Page fault

- When requested address is not in memory – corresponding page is not in memory, ***page fault*** occurs

- Page fault triggers an interrupt
  - in interrupt processing the required page is loaded in memory and page table is updated
  - instruction that caused page fault is then *repeated*

- Page fault is costly for faulting process
  - access time for page on disk is measured in milliseconds (comparing to micro/nano seconds for accessing memory!)

- Demand paging
  - load pages only when they are required



address M belongs to page that is not in memory

page in frame (in memory)

empty frame

page fault triggers interrupt

restart thread

update page table

page table

kernel

load page into an empty frame

fetch page from disk

page on disk

Page replacement

- When all frames are in use and page faults occurs, some frame must be emptied and loaded by requested page
    - which frame? how to choose?
- An approximation to LRU (least recently used) algorithm is often used
    - remove pages that are not used recently
        - probability that they will soon be requested is less than for others (based on a typical application behavior)
    - *clock algorithm* (also known as *second chance algorithm*) is mostly used
        - flag A (*accessed*) from frame descriptors is checked in a circular manner
        - if A is zero (not accessed recently), replace it
        - otherwise, set it to zero and *move to next frame* (give a second chance to recently used frames)

Clock algorithm *example*

Page table

0	0 99	2	1
1	100 199	0	1
2	200 299		0
3	300 399	6	1
4	400 499		0
5	500 599		0

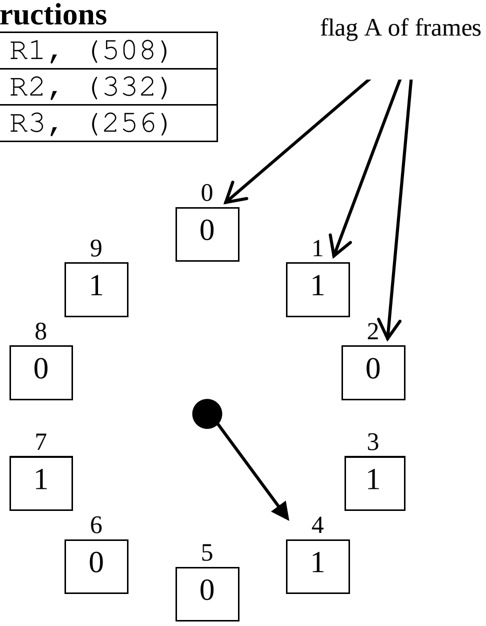
frames

0	1
1	x
2	0
3	x
4	x
5	x
6	3
7	x
8	x
9	x

x – page from other process

Instructions

`1: LDR R1, (508)`
`2: LDR R2, (332)`
`3: LDR R3, (256)`



Virtual memory - summary

- Memory management unit (MMU) is required
- Operating system and MMU handle memory translation
- Benefits:
  - no fragmentation
  - process protection
    - processes are separated, each in its own address space (virtual and physical)
  - large programs can be executed using demand paging
- Disadvantages:
  - cost (additional space on processor chip is required for MMU)
  - slowdown (if frequent page faults occur)
- All general operating systems support VM
  - Real-Time and embedded system are exceptions

Programming for Virtual memory systems

- In theory no program change/preparation is required
  - memory management is transparent for program – completely managed by operating system and MMU

- But VM awareness can significantly improve program performance:
  - page faults are very expensive – avoid them!

  - principle is “simple”: manipulate with data in sequential manner, avoid random data access
    - **principle of locality**: *temporal and spatial locality*

    - same principle will **benefit** from all **cache mechanisms** embedded in hardware and software components, from disk to L1 cache !!!

When programming for Real-Time

- Use API for locking particular pages in memory

- E.g. POSIX:
  - lock memory segment:
    - `int mlock (const void * addr, size_t len);`

    `http://www.opengroup.org/onlinepubs/9699919799/functions/mlock.html`

  - lock whole process (and more):
    - `int mlockall (int flags);`

    `http://www.opengroup.org/onlinepubs/9699919799/functions/mlockall.html`

- E.g. Win32
  - `VirtualLock (lpAddress, dwSize);`

  `http://msdn.microsoft.com/en-us/library/aa366895(VS.85).aspx`

  - `SetProcessWorkingSetSize (hProcess, Min, Max)`

  `http://msdn.microsoft.com/en-us/library/ms686234(v=VS.85).aspx`