INFO 324 Operating Systems II

Ahmad FAOUR

Memory Management

Memory Management

Memory – a linear array of bytes

- Holds O.S. and programs (processes)
- Each cell (byte) is named by a unique memory address

Recall, processes are defined by an *address space*, consisting of text, data, and stack regions

Process execution

- CPU fetches instructions from the text region according to the value of the program counter (PC)
- Each instruction may request additional operands from the data or stack region

Addressing Memory

Cannot know ahead of time where in memory a program will be loaded!

Compiler produces code containing embedded addresses

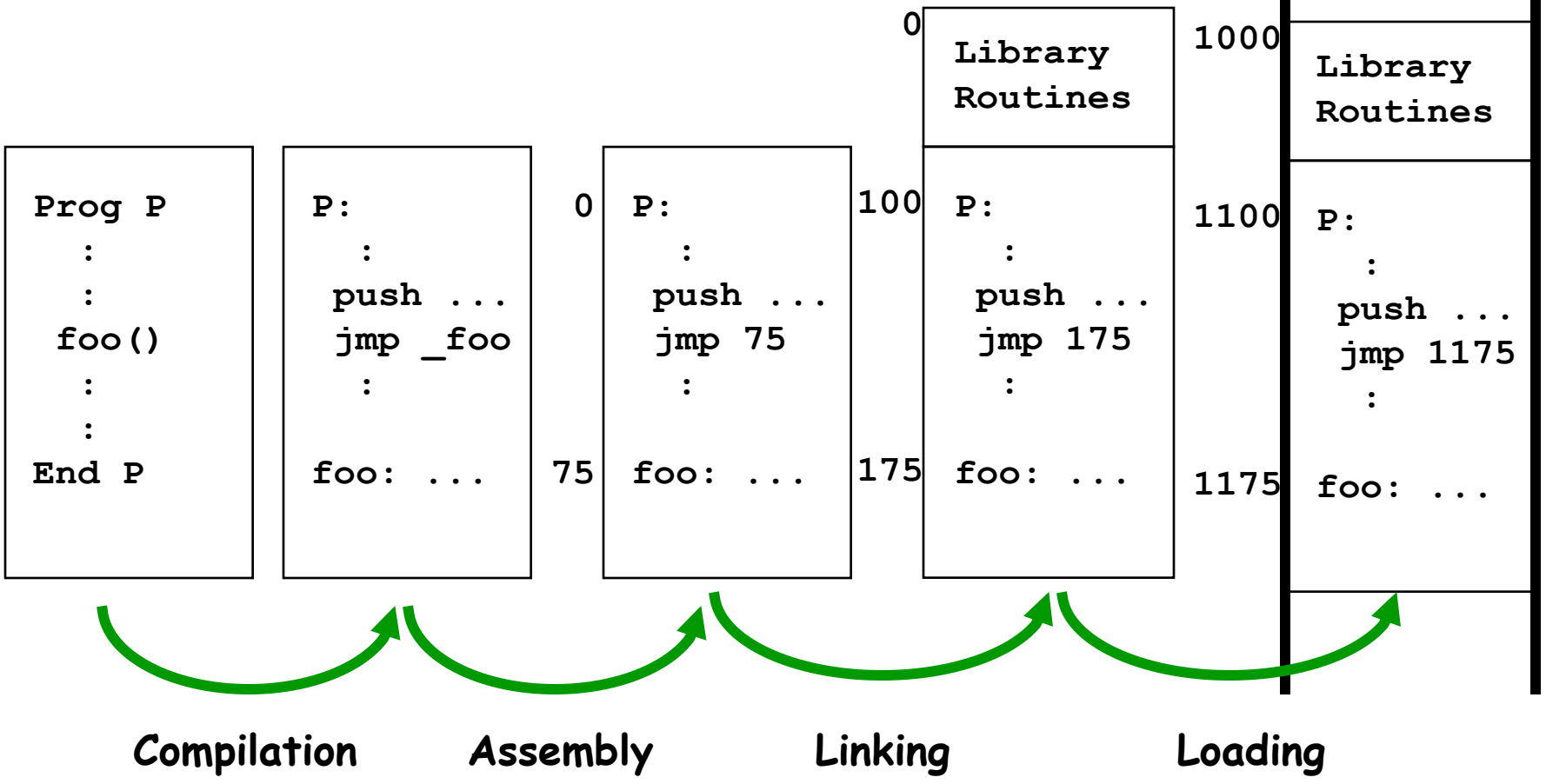
these addresses can't be absolute (physical addresses)

Linker combines pieces of the program

Assumes the program will be loaded at address 0

We need to bind the compiler/linker generated addresses to the actual memory locations

Relocatable Address Generation



Address Binding

Address binding

- fixing a physical address to the logical address of a process' address space

Compile time binding

- if program location is fixed and known ahead of time

Load time binding

- if program location in memory is unknown until run-time AND location is fixed

Execution time binding

- if processes can be moved in memory during execution
- Requires hardware support!

Base and Limit Registers

Simple runtime relocation scheme

- Use 2 registers to describe a partition

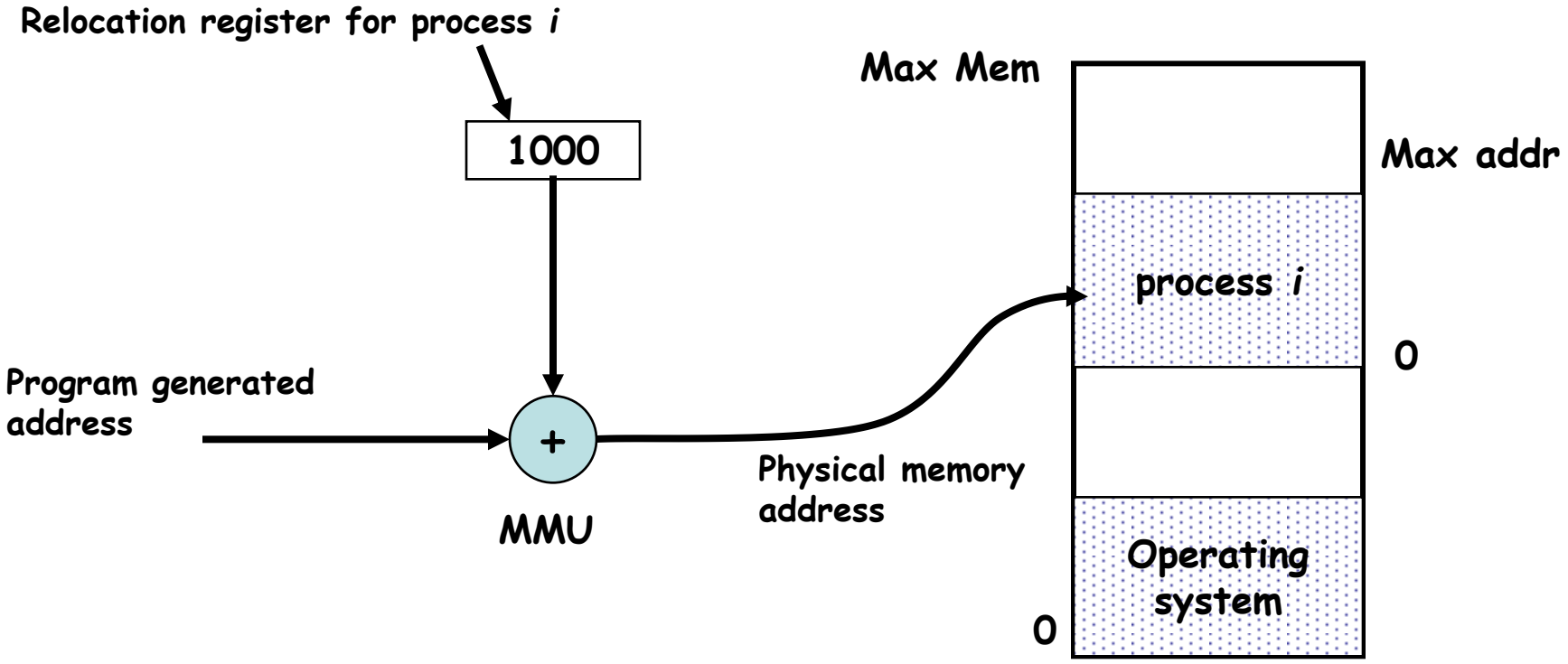
For every address generated, at runtime...

- Compare to the limit register (& abort if larger)
- Add to the base register to give physical memory address

Dynamic Relocation

Memory Management Unit (MMU)

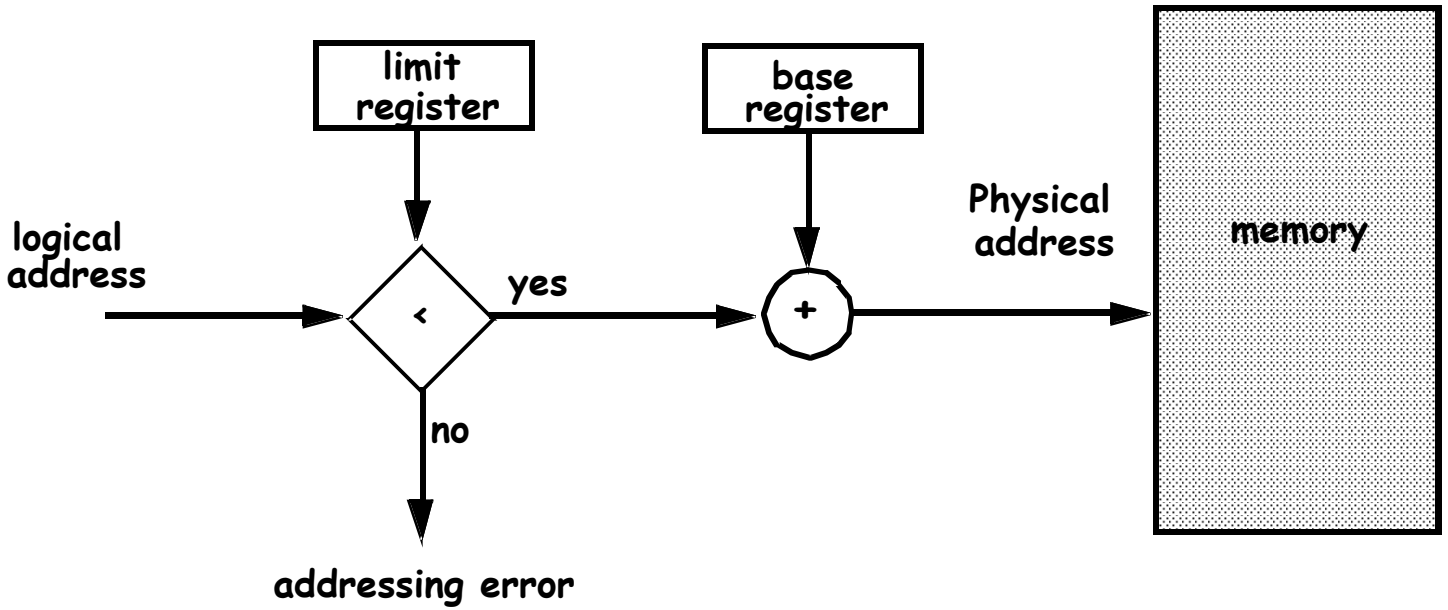
- Dynamically converts logical to physical address
- Contains base address register for running process



Protection

Memory protection

- Base register gives starting address for process
- Limit register limits the offset accessible from the relocation register

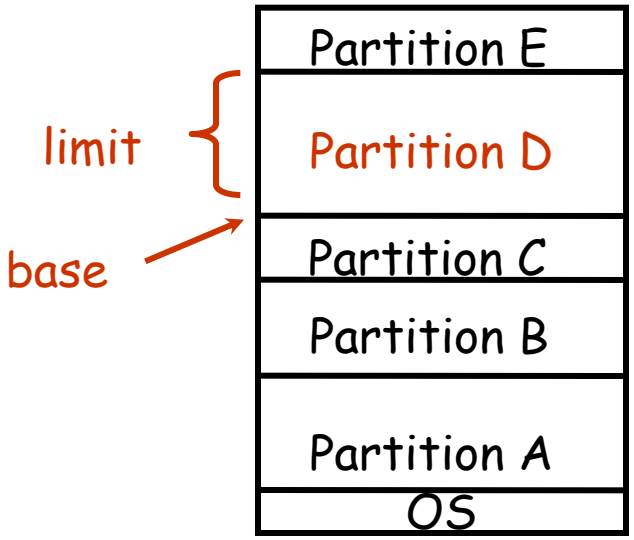


Multiprogramming

Multiprogramming: a separate partition per process

What happens on a context switch?

- Store process base and limit register values
- Load new values into base and limit registers



Swapping

When a program is running...

- The entire program must be in memory
- Each program is put into a single partition

When the program is not running...

- May remain resident in memory
- May get "*swapped*" out to disk

Over time...

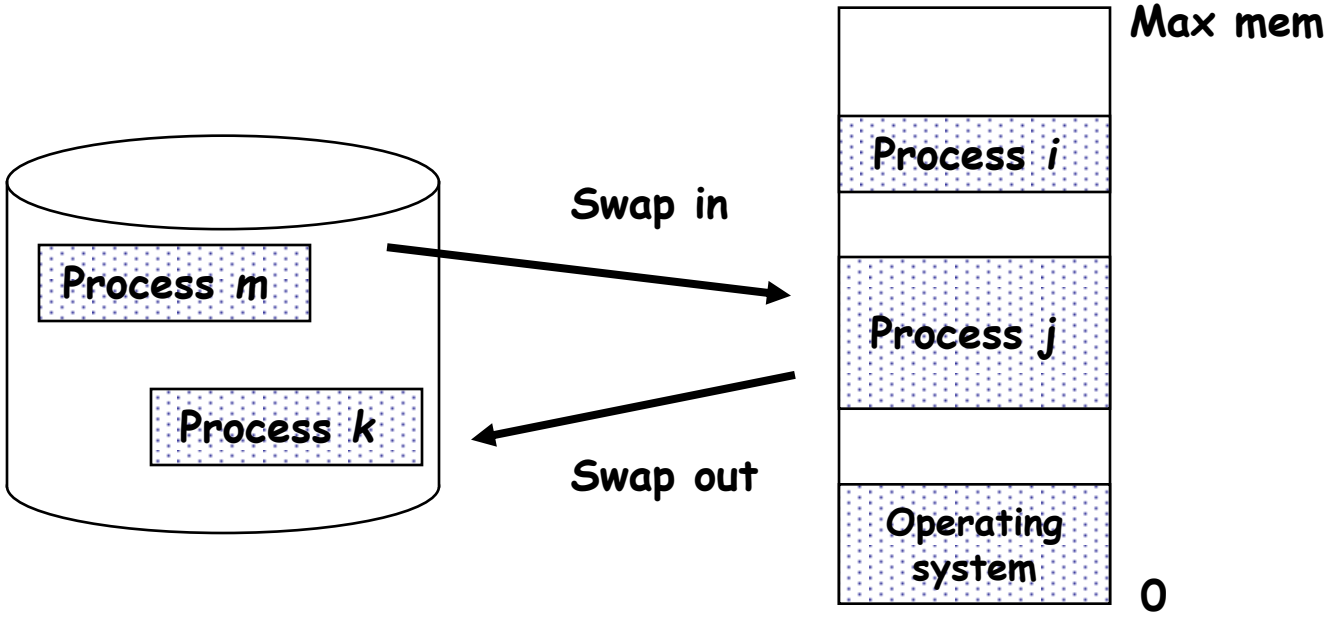
- Programs come into memory when they get swapped in
- Programs leave memory when they get swapped out

Swapping

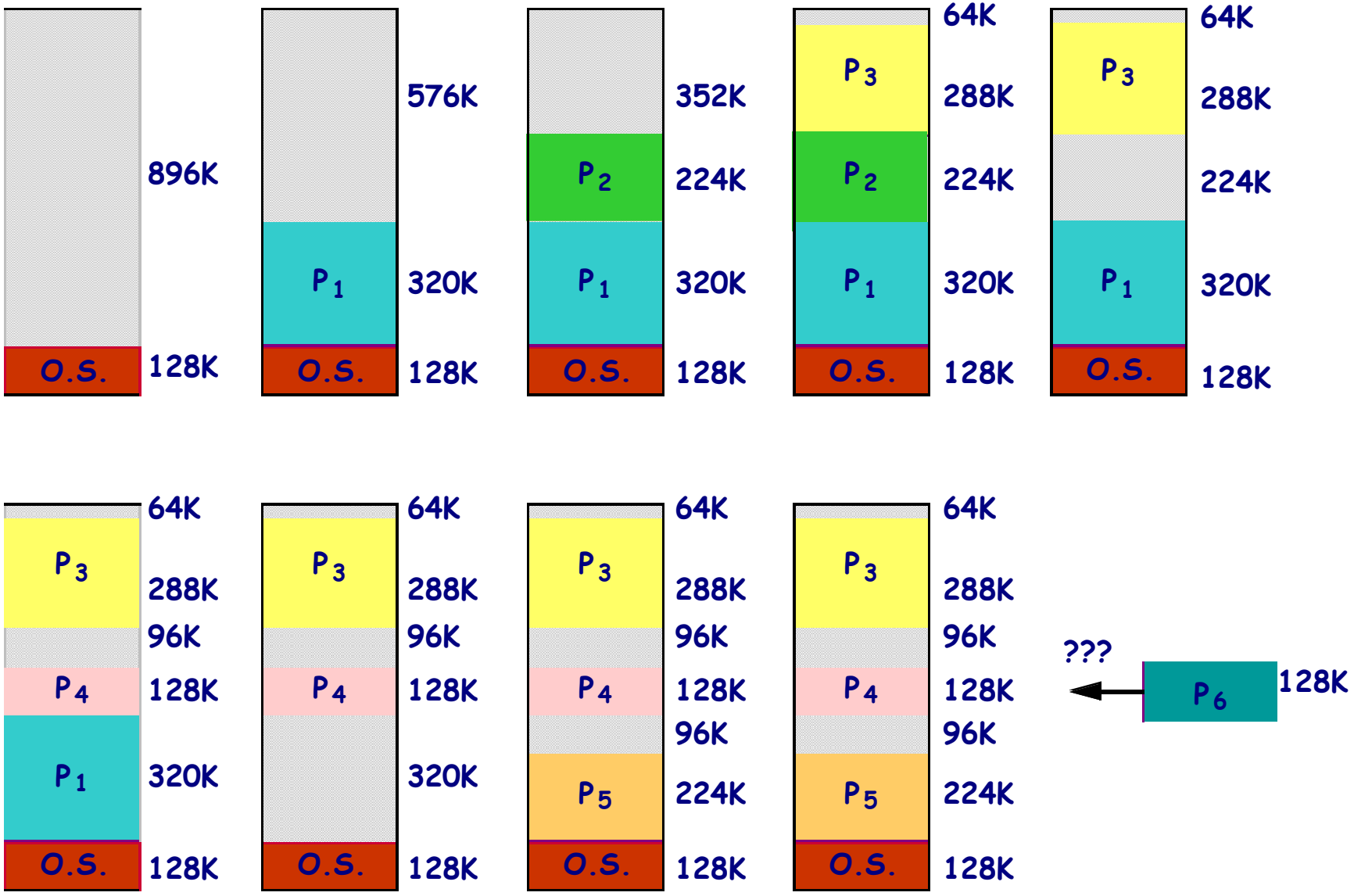
Benefits of swapping:

Allows multiple programs to be run concurrently

… more than will fit in memory at once



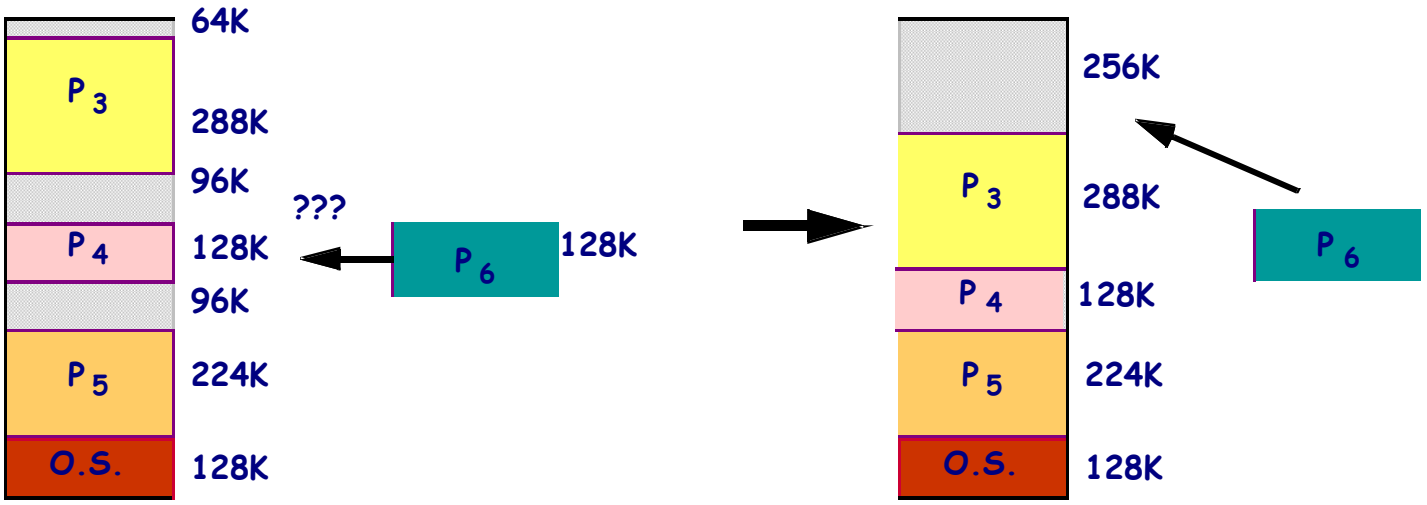
Fragmentation


896K
O.S.
128K
576K
P1
320K
O.S.
128K
352K
P2
224K
P1
320K
O.S.
128K
64K
P3
288K
P2
224K
P1
320K
O.S.
128K
64K
P3
288K
224K
P1
320K
O.S.
128K
64K
P3
288K
96K
P4
128K
P1
320K
O.S.
128K
64K
P3
288K
96K
P4
128K
320K
O.S.
128K
64K
P3
288K
96K
P4
128K
96K
P5
224K
O.S.
128K
64K
P3
288K
96K
P4
128K
96K
P5
224K
O.S.
128K
???
P6
128K

Dealing With Fragmentation

Compaction – from time to time shift processes around to collect all free space into one contiguous block

- Memory to memory copying overhead
- Memory to disk to memory for compaction via swapping!



How Big Should Partitions Be?

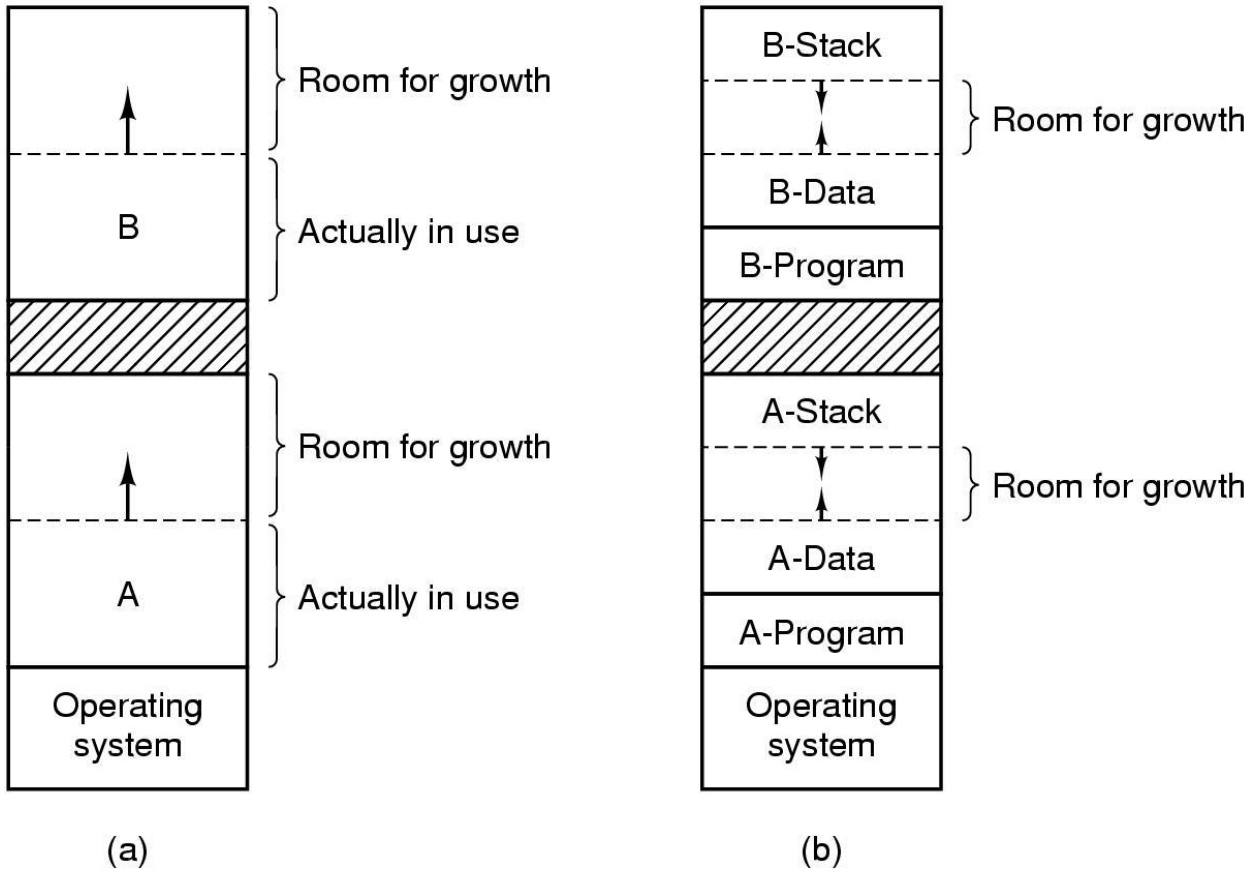
Programs may want to grow during execution

- More room for stack, heap allocation, etc

Problem:

- If the partition is too small, programs must be moved
- Requires copying overhead
- Why not make the partitions a little larger than necessary to accommodate “some” cheap growth?

Allocating Extra Space Within



Management Data Structures

Each chunk of memory is either

- Used by some process or unused (free)

Operations

- Allocate a chunk of unused memory big enough to hold a new process
- Free a chunk of memory by returning it to the free pool after a process terminates or is swapped out

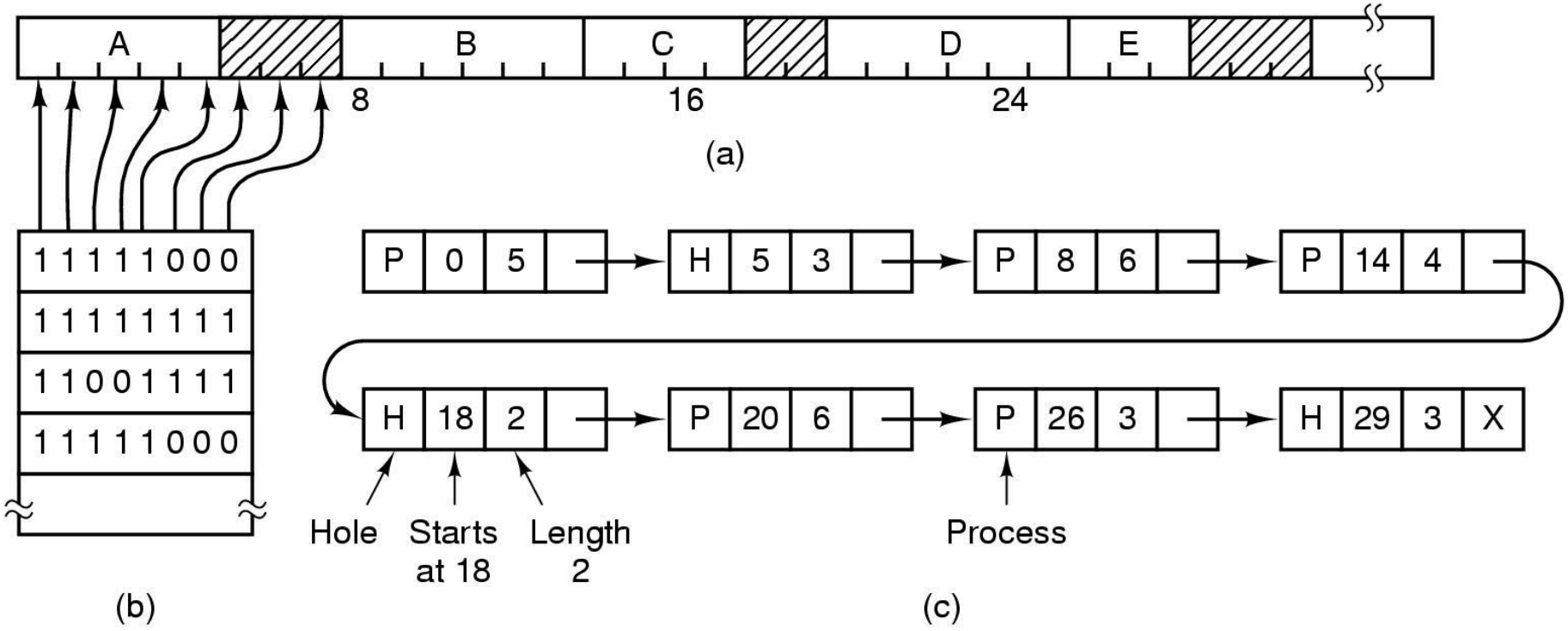
Management With Bit Maps

Problem - how to keep track of used and unused memory?

Technique 1 - Bit Maps

- A long bit string
- One bit for every chunk of memory
  - 1 = in use
  - 0 = free
- Size of allocation unit influences space required
  - Example: unit size = 32 bits
    - overhead for bit map: 1/33 = 3%
  - Example: unit size = 4Kbytes
    - overhead for bit map: 1/32,769

Management With Bit Maps



Management With Linked Lists

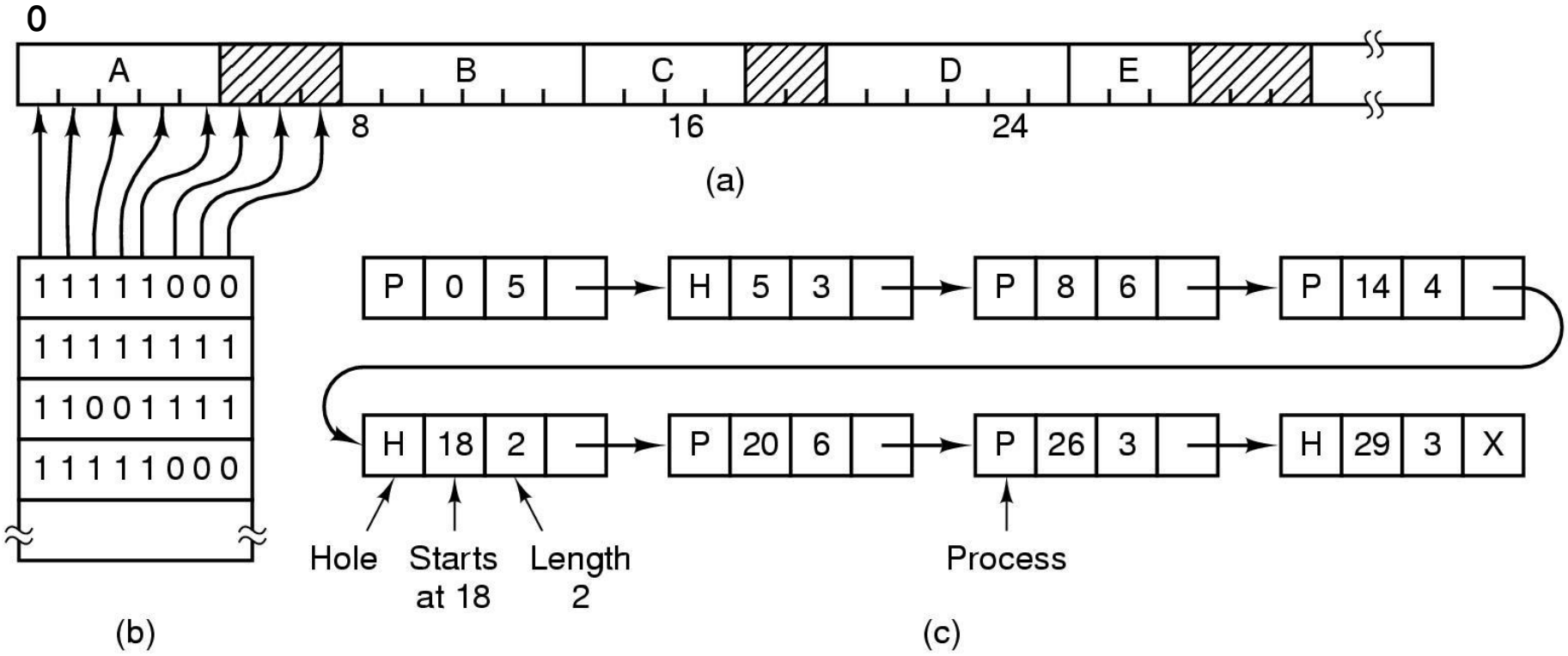
Technique 2 - Linked List

Keep a list of elements

Each element describes one unit of memory

- Free / in-use Bit (“P=process, H=hole”)
- Starting address
- Length
- Pointer to next element

Management With Linked Lists



Management With Linked Lists

Searching the list for space for a new process

- First Fit
- Next Fit
  - Start from current location in the list
- Best Fit
  - Find the smallest hole that will work
  - Tends to create lots of really small holes
- Worst Fit
  - Find the largest hole
  - Remainder will be big
- Quick Fit
  - Keep separate lists for common sizes

Fragmentation Revisited

Memory is divided into partitions

Each partition has a different size

Processes are allocated space and later freed

After a while memory will be full of small holes!

- No free space large enough for a new process even though there is enough free memory in total

If we allow free space within a partition we have fragmentation

External fragmentation = unused space between partitions

Internal fragmentation = unused space within partitions

Solutions to Fragmentation

Compaction requires high copying overhead

Why not allocate memory in non-contiguous equal fixed size units?

- No external fragmentation!
- Internal fragmentation < 1 unit per process

How big should the units be?

- The smaller the better for internal fragmentation
- The larger the better for management overhead

The key challenge for this approach

How can we do secure dynamic address translation?

Non-Contiguous Allocation (Pages)

Memory divided into fixed size page frames

- Page frame size = 2^n bytes
- Lowest n bits of an address specify byte offset in a page

But how do we associate page frames with processes?

- And how do we map memory addresses within a process to the correct memory byte in a page frame?

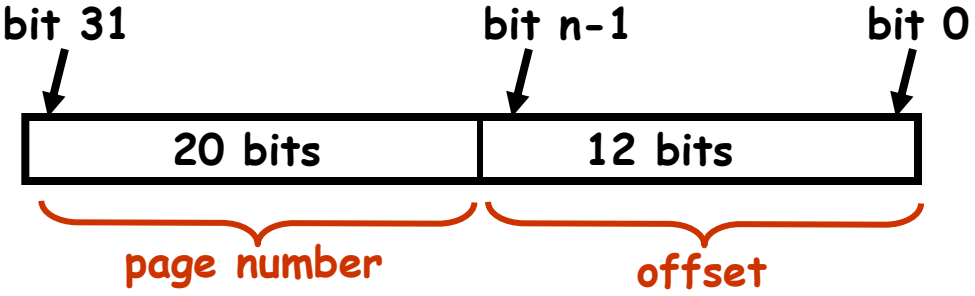
Solution – address translation

- Processes use virtual addresses
- CPU uses physical addresses
- Hardware support for virtual to physical address translation

Virtual Addresses

Virtual memory addresses (what the process uses)

- Page number plus byte offset in page
- Low order n bits are the byte offset
- Remaining high order bits are the page number



Example: 32 bit virtual address

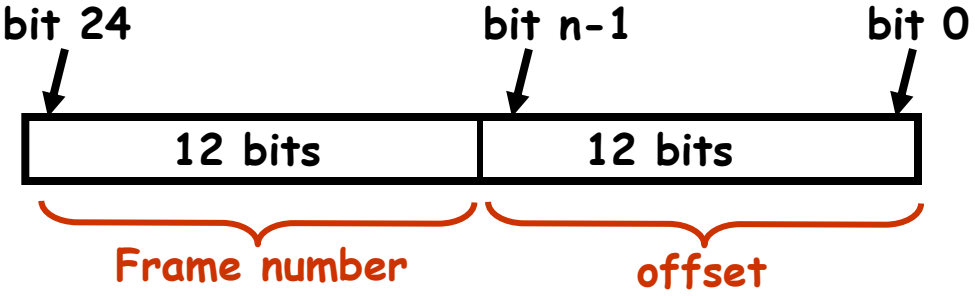
Page size = 2^{12} = 4KB

Address space size = 2^{32} bytes = 4GB

Physical Addresses

Physical memory addresses (what the CPU uses)

- Page "frame" number plus byte offset in page
- Low order n bits are the byte offset
- Remaining high order bits are the frame number



Example: 24 bit physical address

Frame size = 2^{12} = 4KB

Max physical memory size = 2^{24} bytes = 16MB

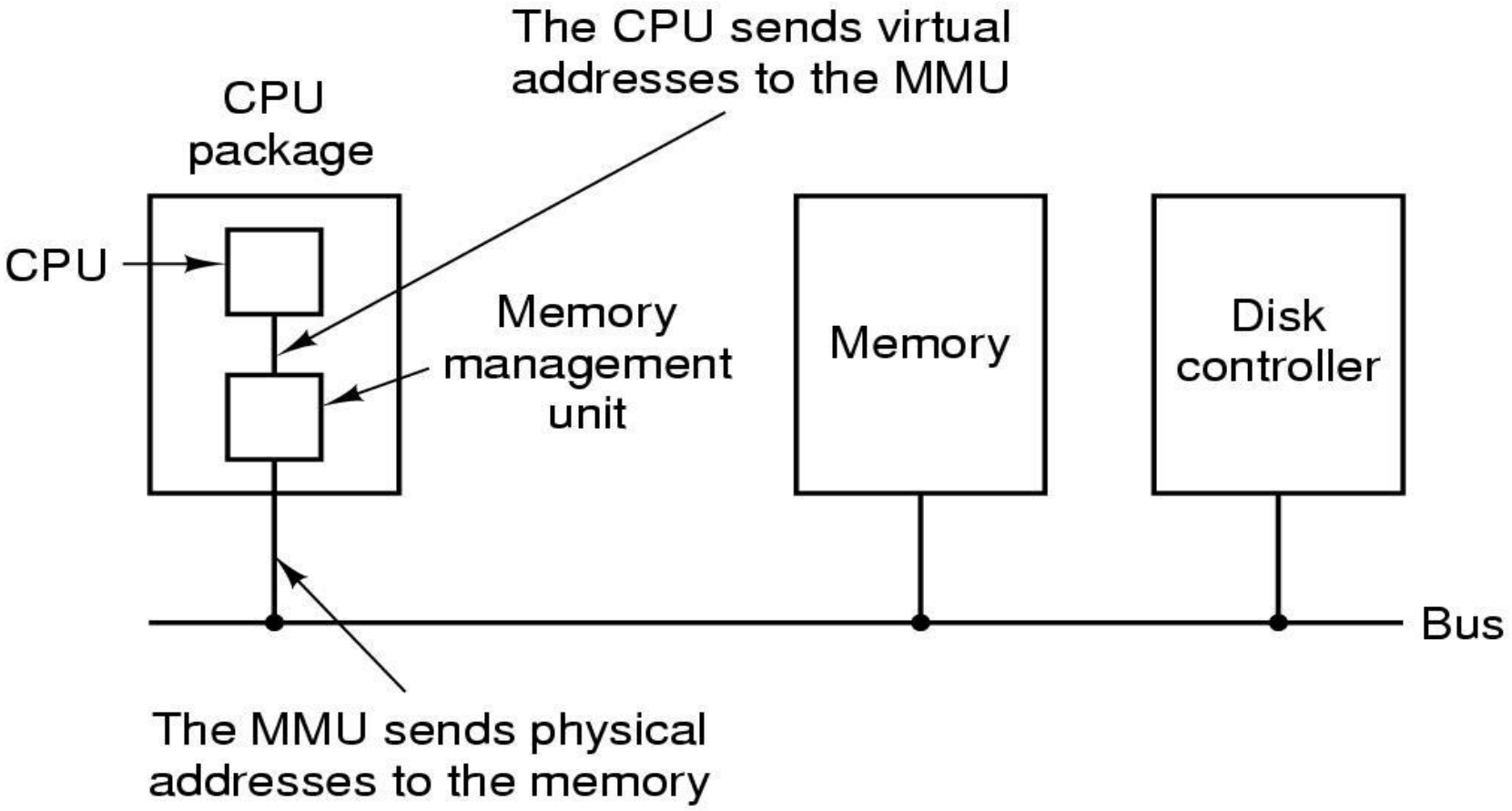
Address Translation

Hardware maps page numbers to frame numbers

Memory management unit (MMU) has multiple registers for multiple pages

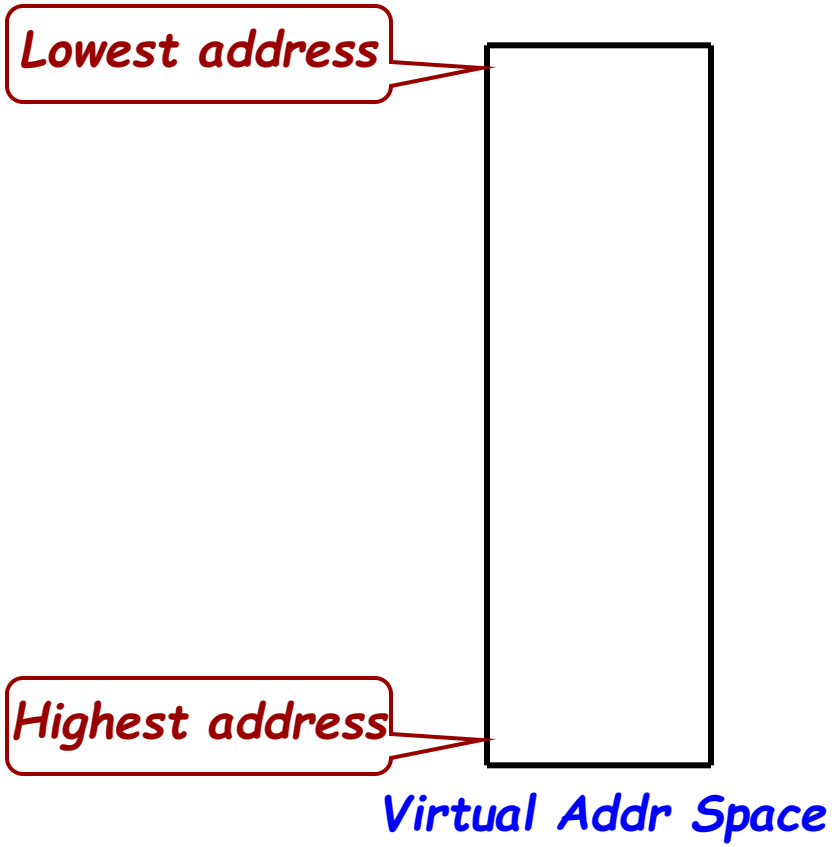
- Like a base register except its value is substituted for the page number rather than added to it
- Why don’t we need a limit register for each page?

Memory Management Unit (MMU)



Virtual Address Spaces

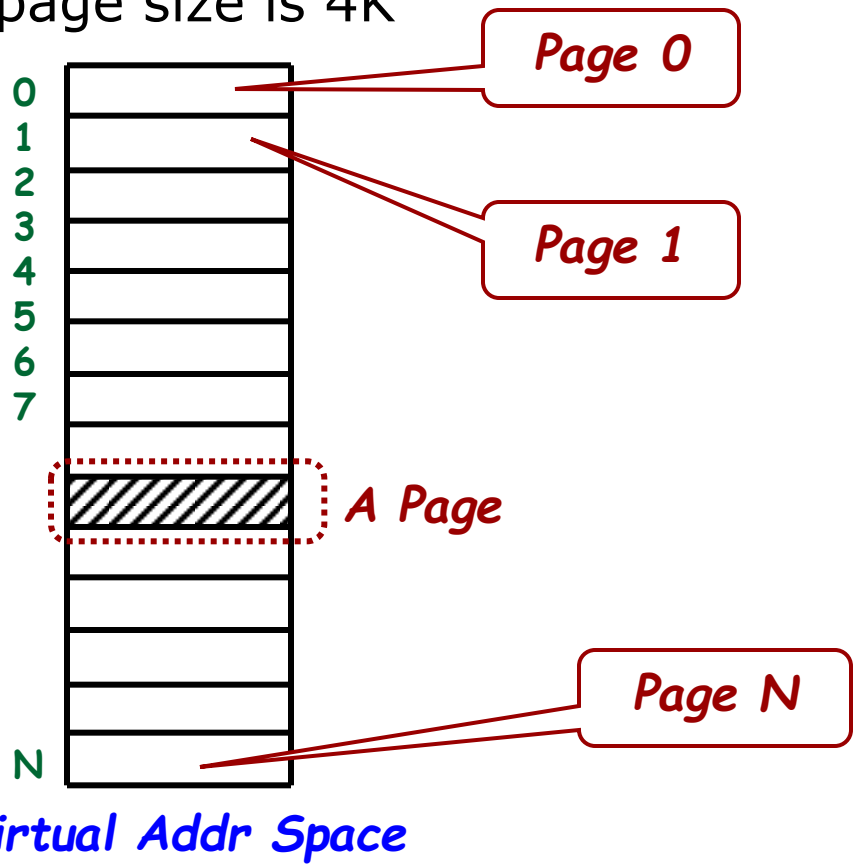
Here is the virtual address space (as seen by the process)



Virtual Address Spaces

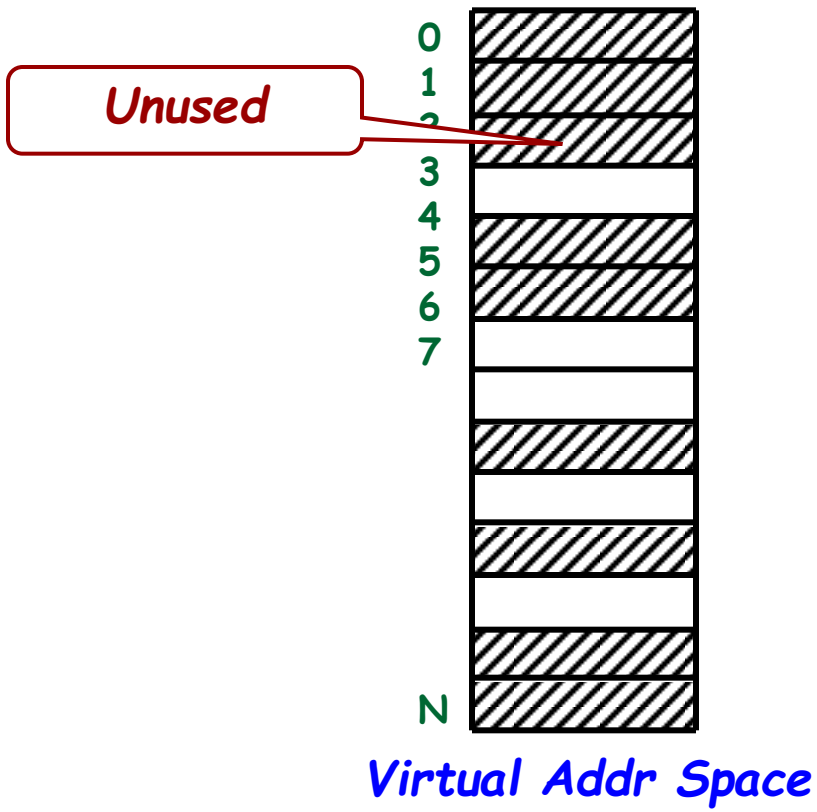
The address space is divided into “pages”

In UNIX, the page size is 4K



Virtual Address Spaces

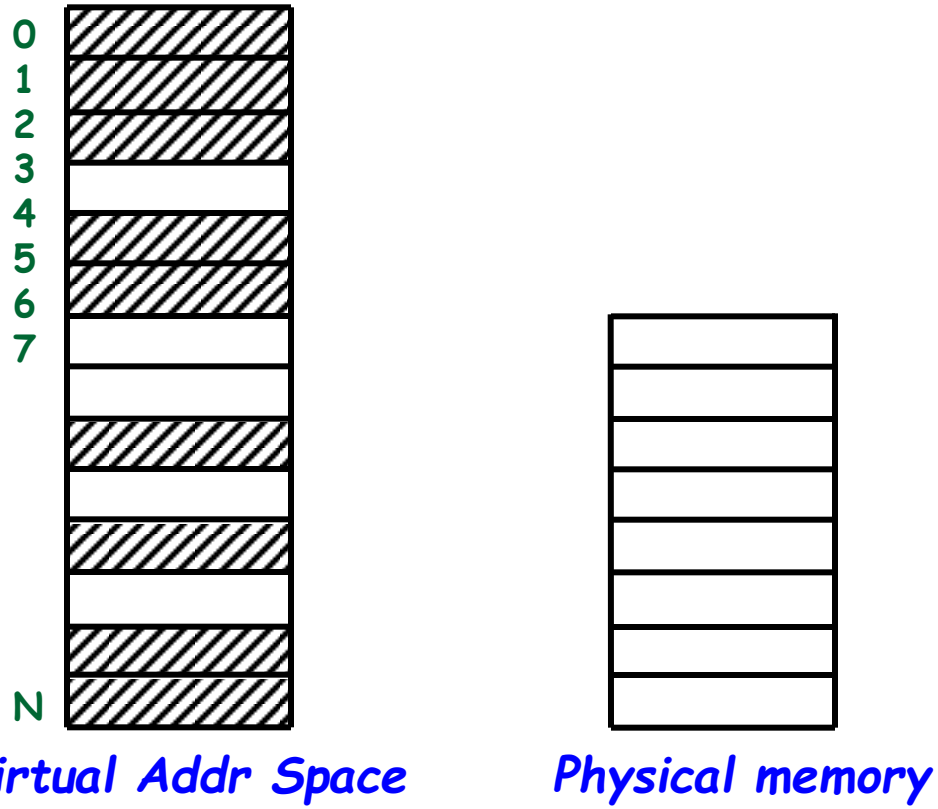
In reality, only some of the pages are used



Physical Memory

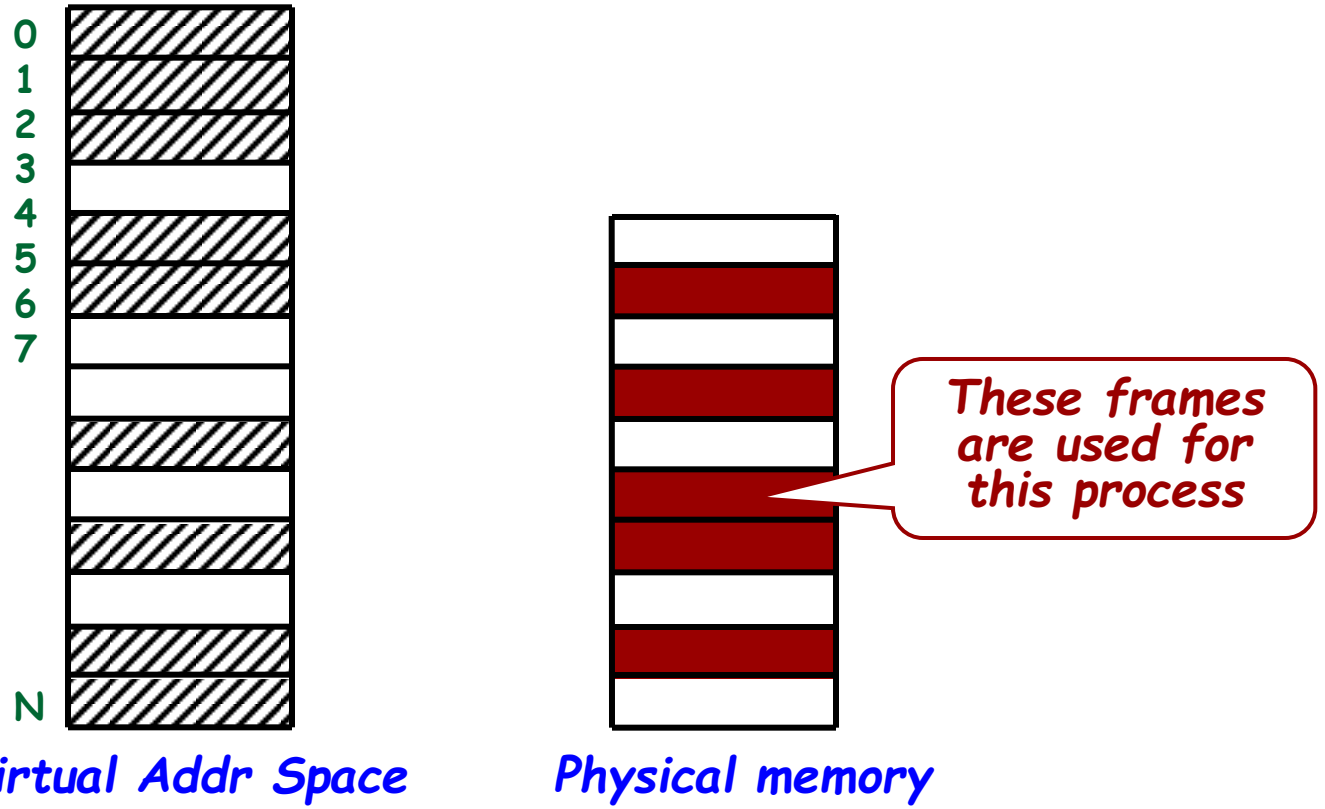
Physical memory is divided into "*page frames*"

(Page size = frame size)



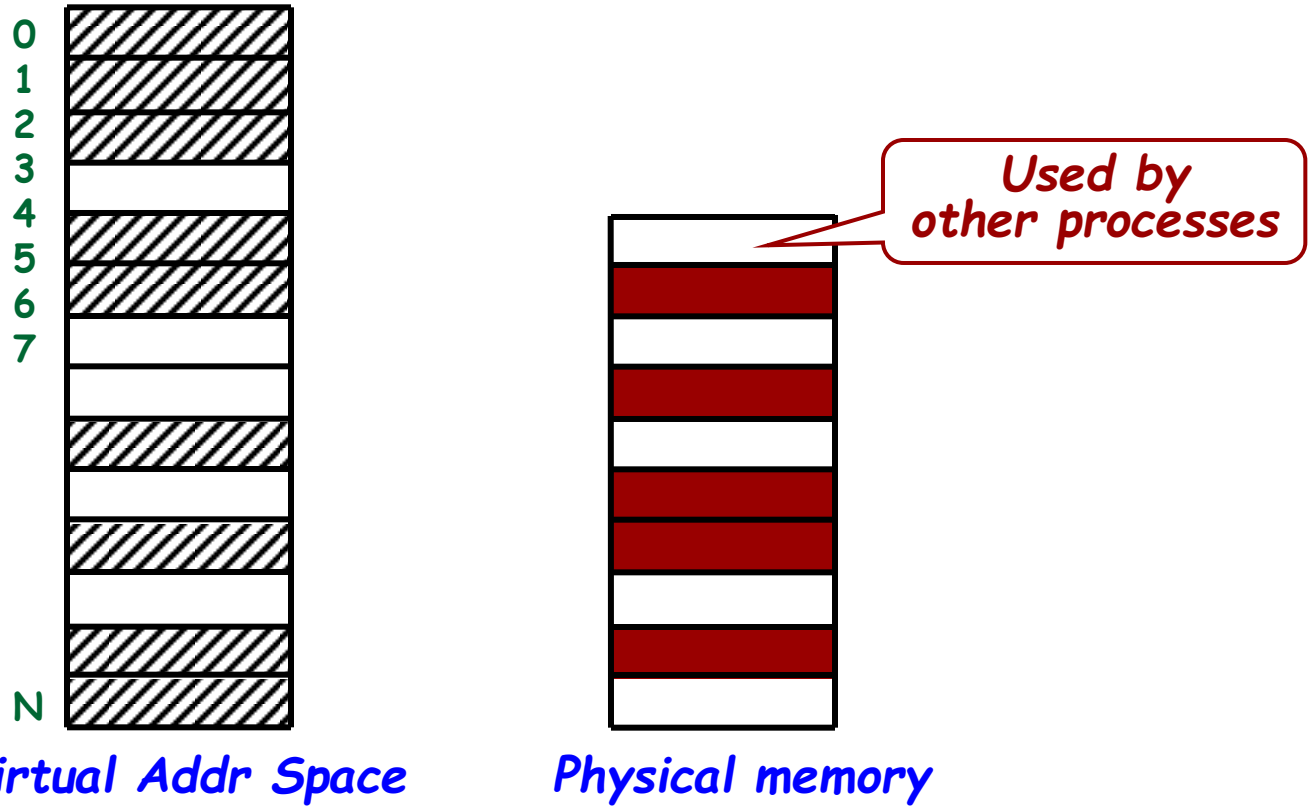
Virtual & Physical Address Spaces

Some frames are used to hold the pages of this process



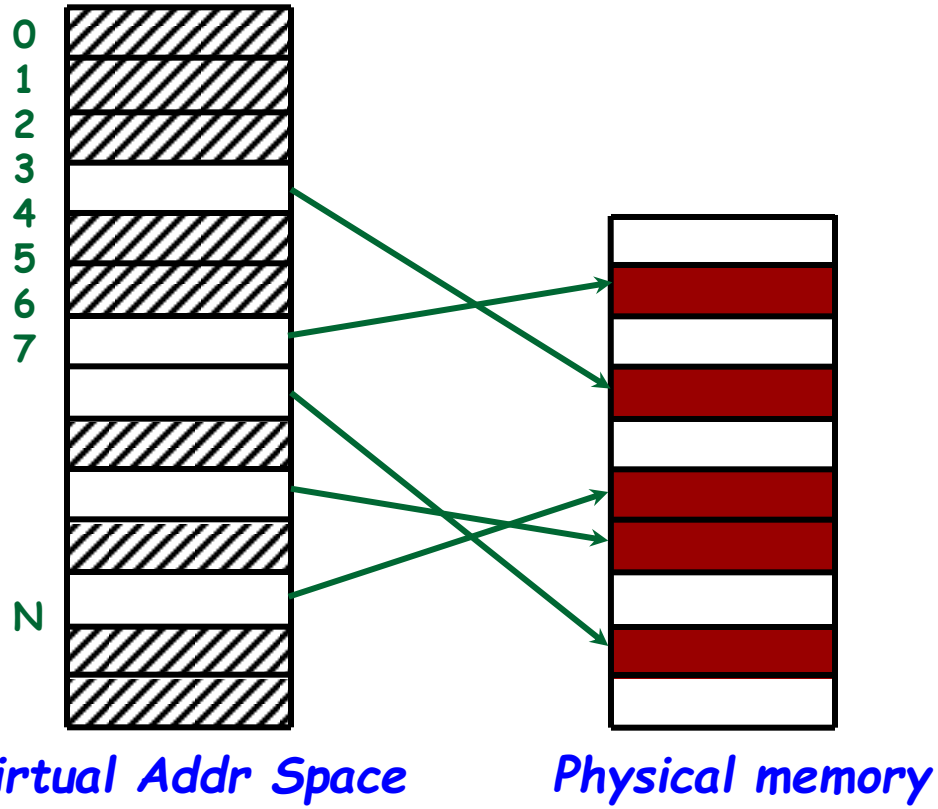
Virtual & Physical Address Spaces

Some frames are used for other processes



Virtual & Physical Address Spaces

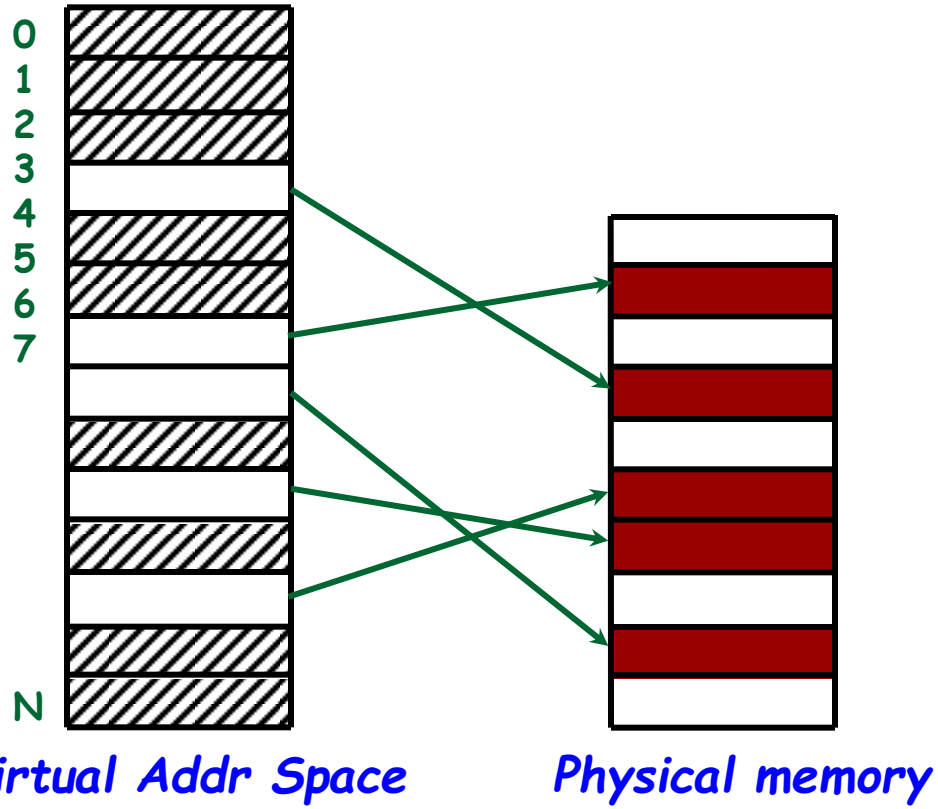
Address mappings say which frame has which page



Page Tables

Address mappings are stored in a *page table* in memory

1 entry/page: is page in memory? If so, which frame is it in?

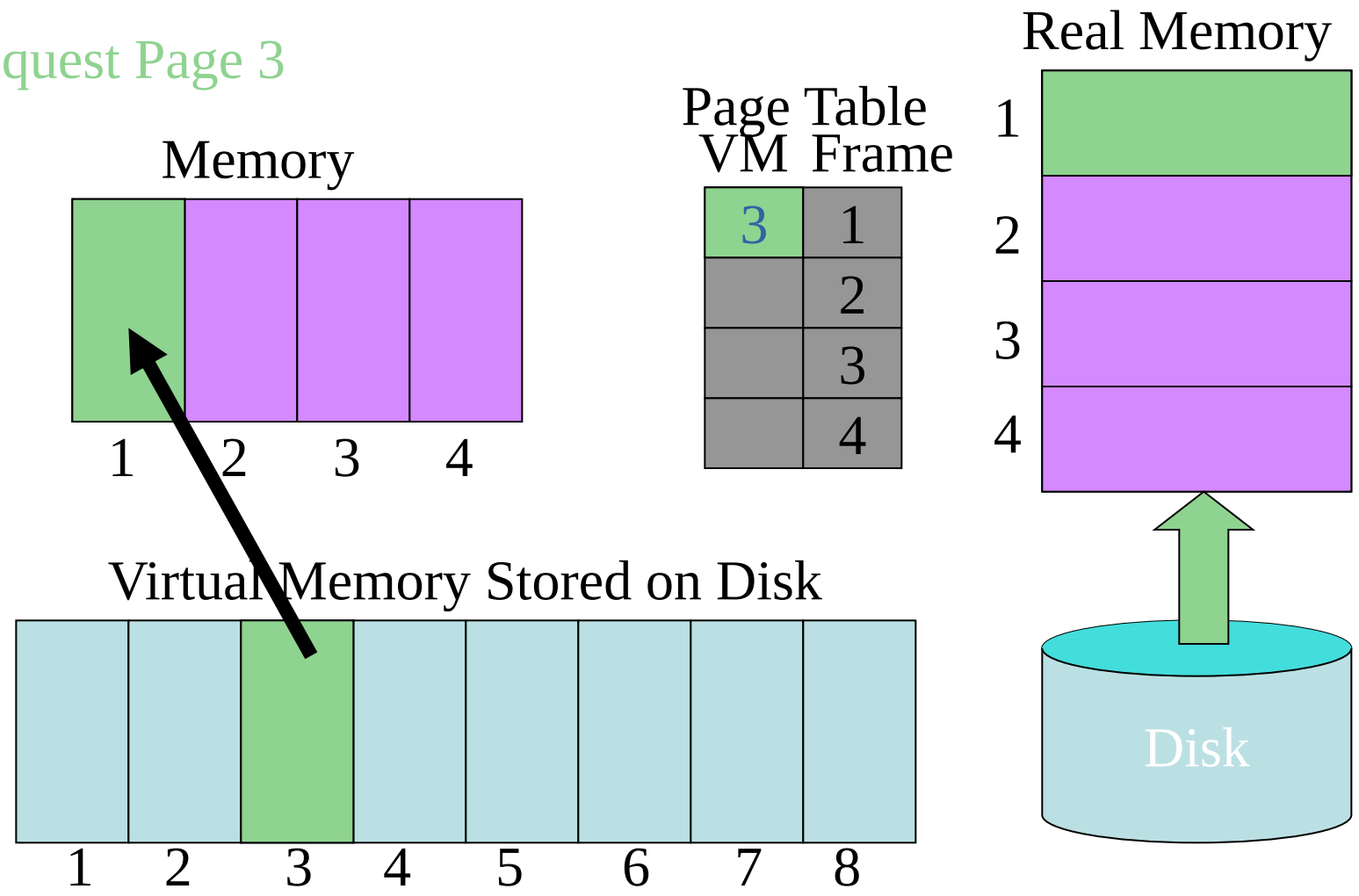


Page Fault

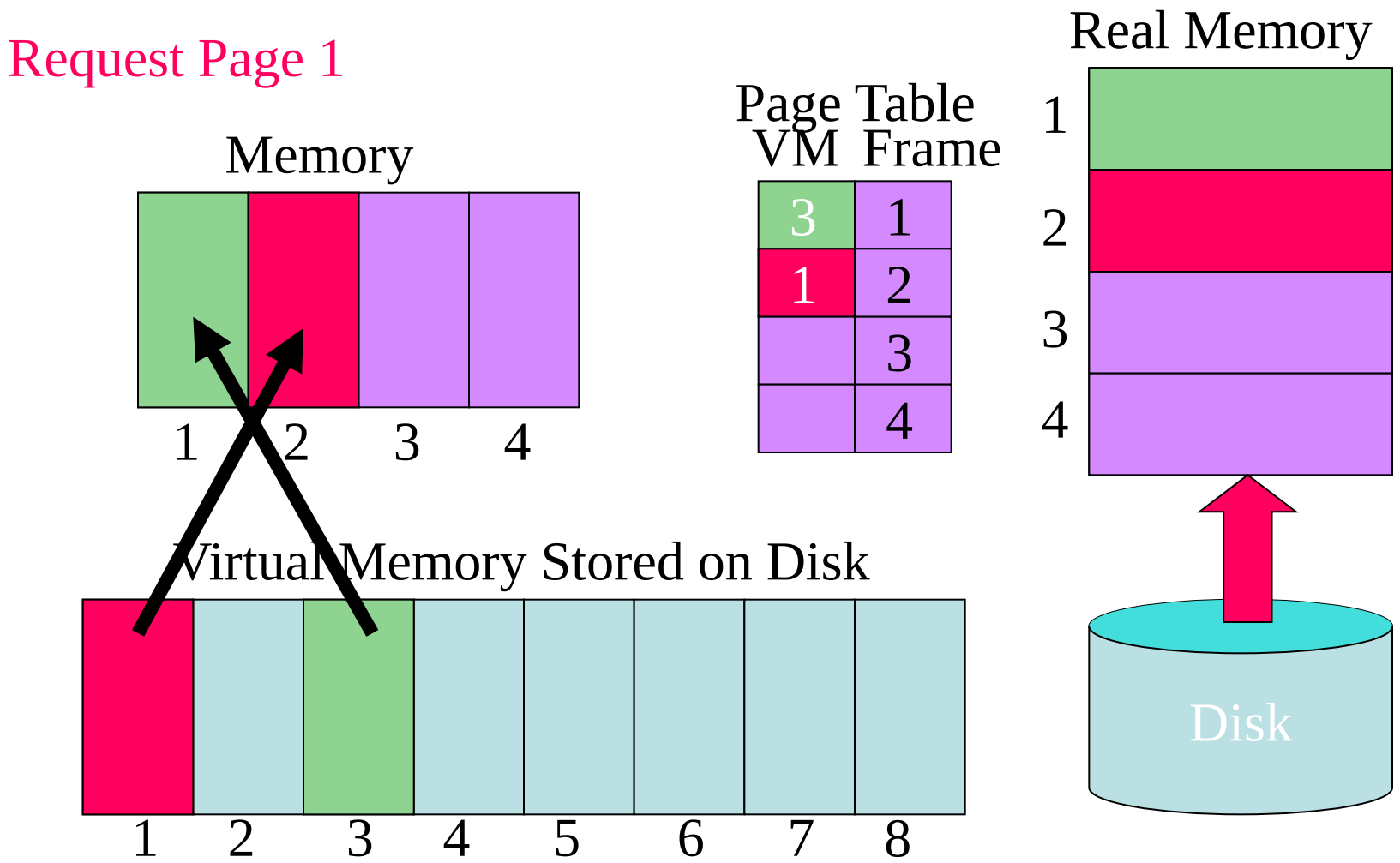
- Access a virtual page that is not mapped into any physical page
  - A fault is triggered by hardware
- Page fault handler (in OS's VM subsystem)
  - Find if there is any free physical page available
    - If no, evict some resident page to disk (swapping space)
  - Allocate a free physical page
  - Load the faulted virtual page to the prepared physical page
  - Modify the page table

Paging Request

Request Page 3

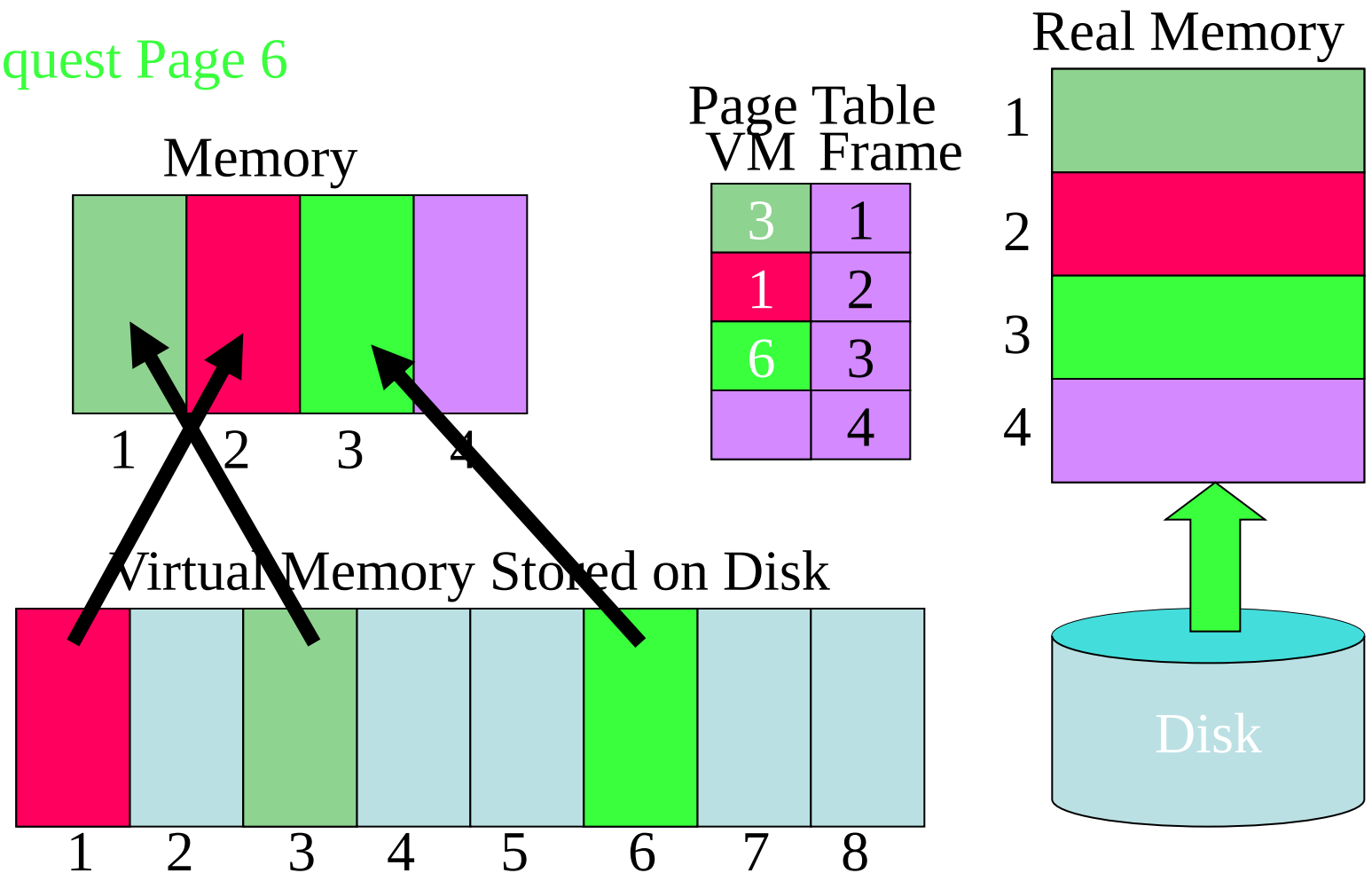


Paging

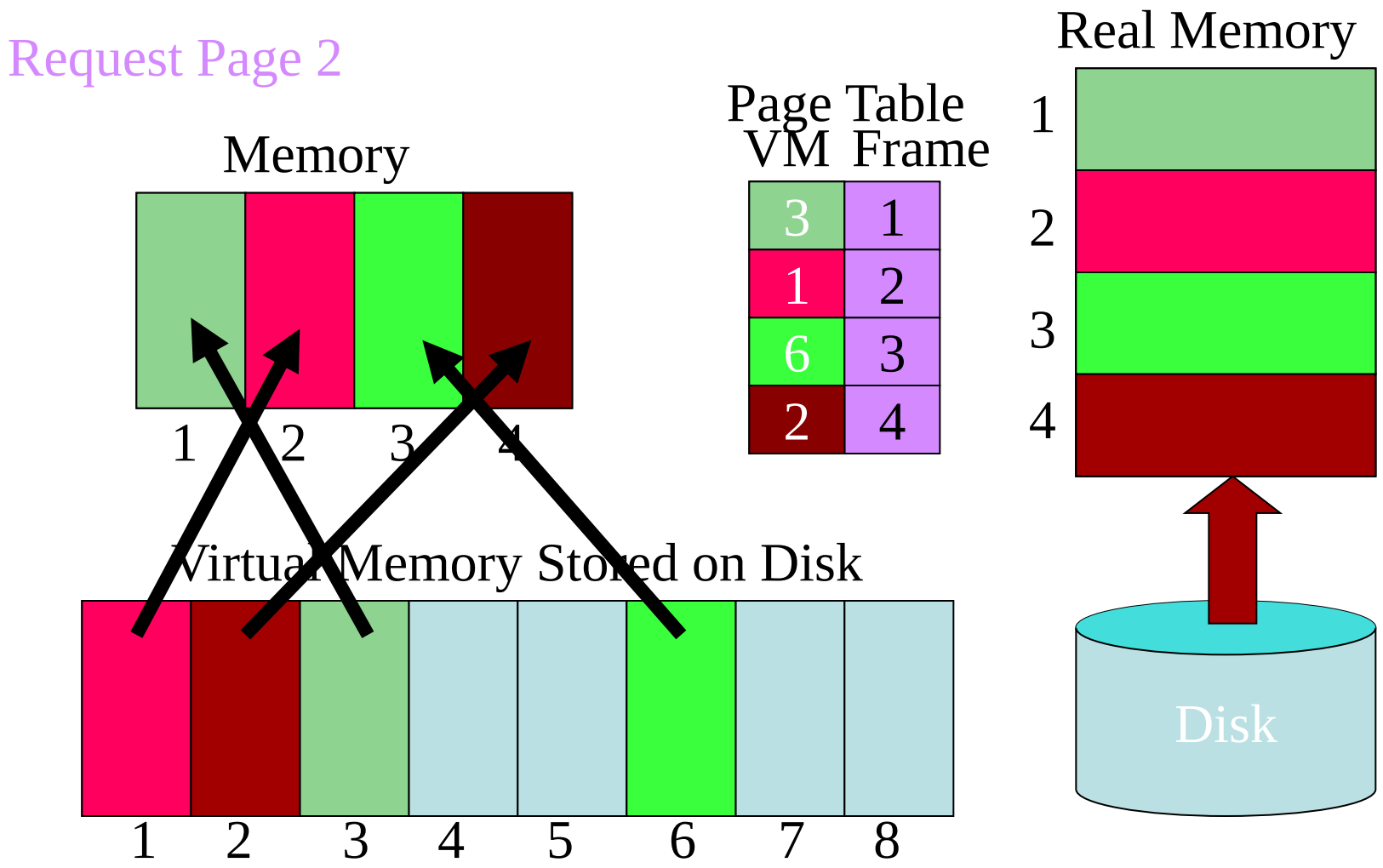

Request Page 1
Memory
1
2
3
4
Page Table
VM Frame
3 1
1 2
3
4
Real Memory
1
2
3
4
Virtual Memory Stored on Disk
1
2
3
4
5
6
7
8
Disk

Paging

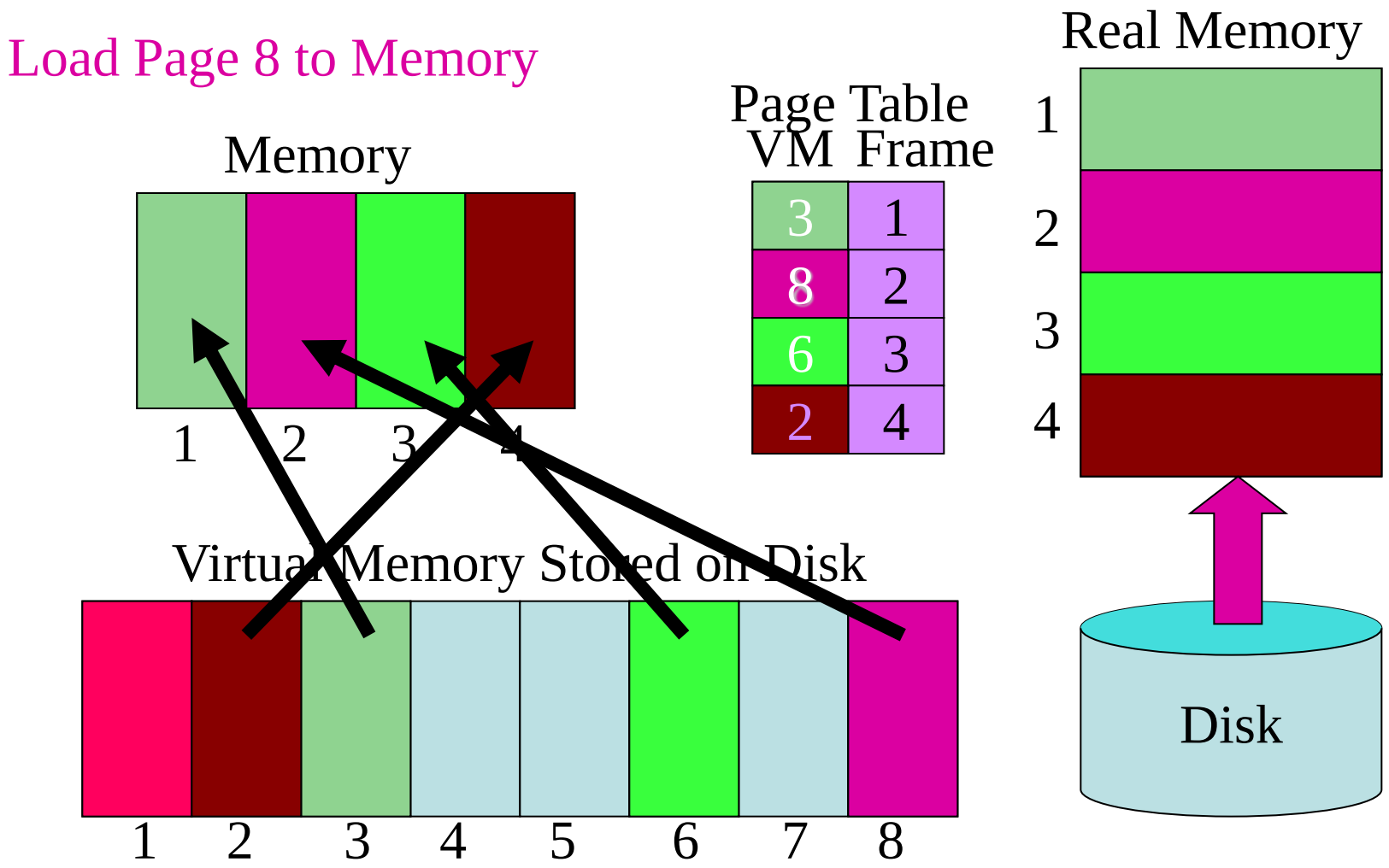
Request Page 6



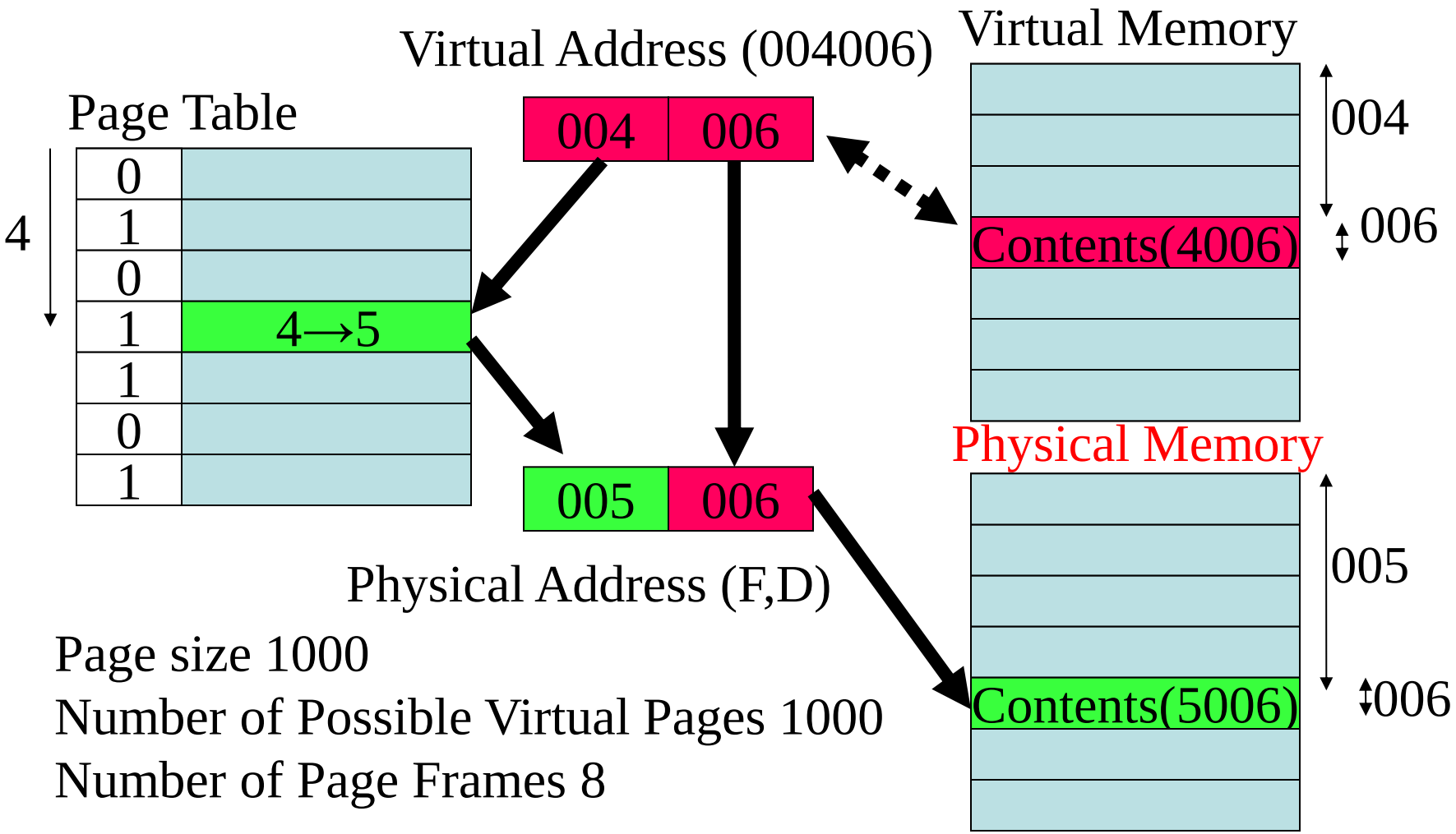
Paging


Request Page 2
Memory
1
2
3
4
Page Table
VM Frame
3 1
1 2
6 3
2 4
Real Memory
1
2
3
4
Virtual Memory Stored on Disk
1 2 3 4 5 6 7 8
Disk

Paging


Load Page 8 to Memory
Memory
1
2
3
4
Page Table
VM Frame
3 1
8 2
6 3
2 4
Real Memory
1
2
3
4
Virtual Memory Stored on Disk
1 2 3 4 5 6 7 8
Disk

Page Mapping Hardware



Page size 1000

Number of Possible Virtual Pages 1000

Number of Page Frames 8

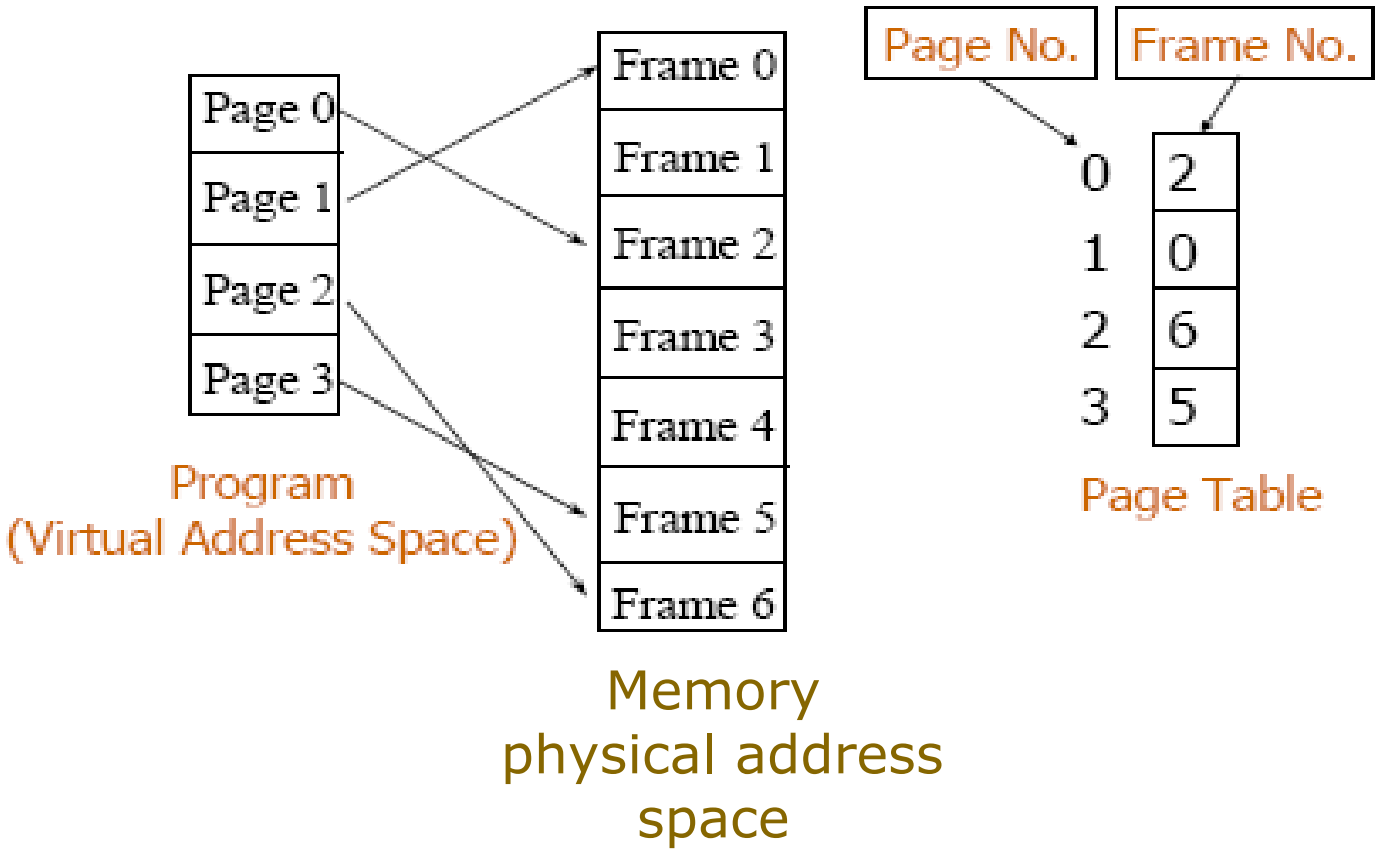
Execution Algorithm - Version 1

Each time a virtual address is generated by the executing program, the following algorithm is used to complete the actual reference.

1. The MAR (Memory Address Register) is assumed to contain the virtual address to be referenced.
2. Set VPN (Virtual Page Number) = MAR div pagesize.
3. Set offset = MAR mod pagesize.
4. Set PPN (Physical Page Number) = PT [VPN].
5. Set PADDR (Physical Address) = PPN * pagesize + offset.
6. Reference the memory at PADDR.

Page table

A Small Program and Page Table



Example Execution

Page size is 256, MAR = 600

- VPN = 600 div 256 = 2
- Offset = 600 mod 256 = 88
- PPN = PT [2] = 6
- PADDR = 6 * 256 + 88 = 1624
- Reference memory at physical address 1624.

Extending the Page Table

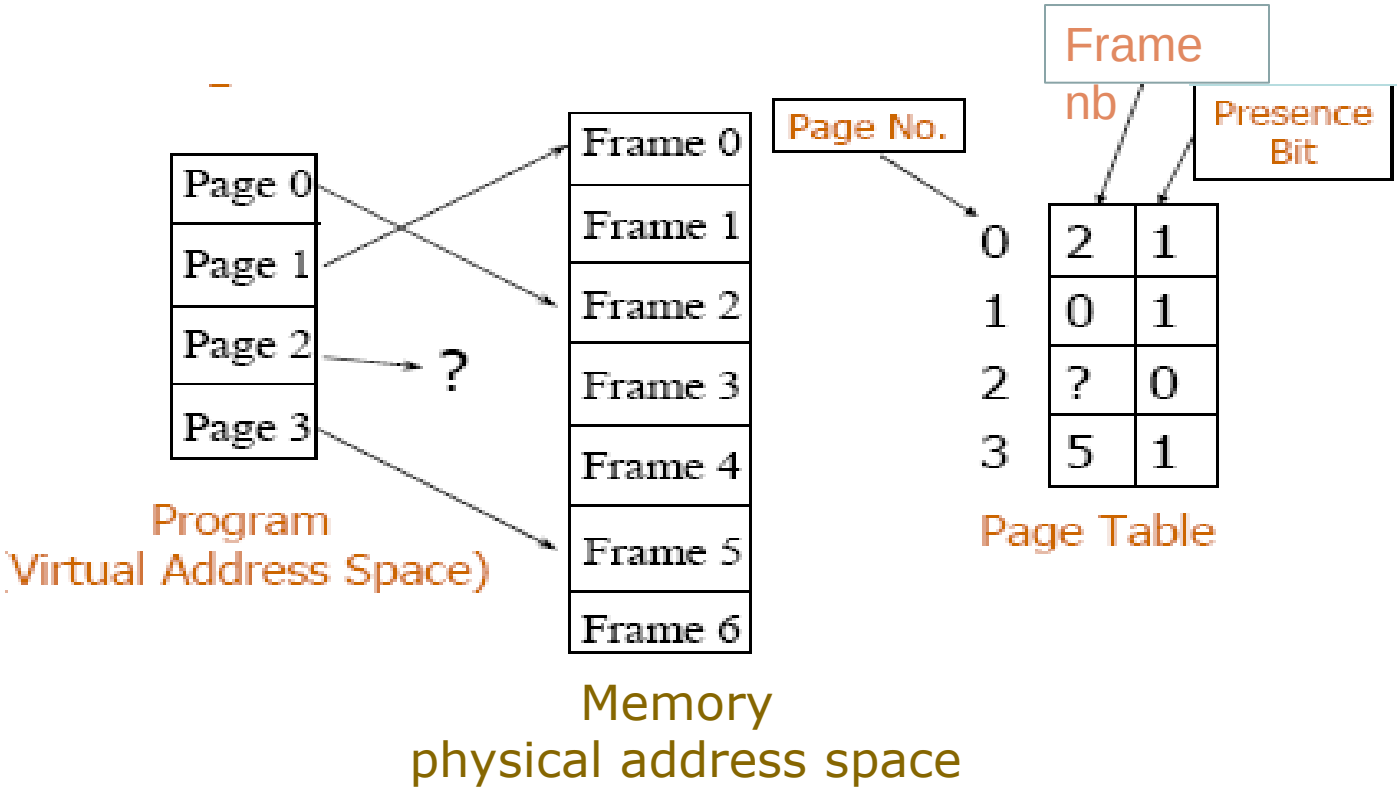
- If a page from a program isn't in primary memory, we must have some technique to detect its absence.
- To achieve this goal, we add an extra field, containing a single bit, to each page table entry.
- This presence bit is set to 1 when a page is present (and the mapping provided by that page table entry is valid). The presence bit is 0 if the page isn't in primary memory.

Execution Algorithm - Version 2

1. The MAR has the virtual address to be referenced.
2. Set VPN = MAR div pagesize.
3. Set offset = MAR mod pagesize.
4. If PT[VPN].presence == 0 then a page fault occurs.

In response, the operating system will
    1. load the needed page into an unused page frame,
    2. set PT[VPN].framenumber to the page frame number, and
    3. set PT[VPN].presence = 1.
5. Set PPN = PT[VPN].framenumber.
6. Set PADDR = PPN * pagesize + offset.
7. Reference the memory at PADDR.

Example with Presence Bits



Example Execution with Presence Bits

1. Page size is 256, MAR = 600
2. VPN = 600 div 256 = 2
3. Offset = 600 mod 256 = 88
4. PT [2].presence = 0, a page fault occurs, and
    1. OS loads page 2 into an unused frame, say frame 3
    2. PT[2].framenumber is set to 3
    3. PT[2].presence is set to 1
5. PPN = PT[2].framenumber = 3
6. PADDR = 3 * 256 + 88 = 856
7. Reference memory at physical address 856.

How Expensive is the Solution?

- Although most programs won't have a virtual address space as large as the physical address space, we still need a page table that can map an arbitrary virtual address.

- Since each virtual address reference requires two physical memory address references, programs will run only half as fast as if paging was not used (based on the time required to reference memory).

Q and A about Paged Memory

Q. How large are page tables?

A. For 32-bit address spaces with a 4K page size (e.g. Intel x86), a million pages are possible, with one entry per page. One page table is needed for each process.

Q. Where is the page table stored?

A. In most systems, the page table is stored in primary memory. Each reference to memory in the virtual address space will require two physical memory references (one to the page table and one for the actual reference).

Address Mappings

Address mappings are stored in a page table in memory

- Typically one page table for each process

Address translation is done by hardware (ie the MMU)

How does the MMU get the address mappings?

- Either the MMU holds the entire page table (too expensive) or it knows where it is in physical memory and goes there for every translation (too slow)
- Or the MMU holds a portion of the page table and knows how to deal with TLB misses
  - MMU caches page table entries
  - Cache is called a translation look-aside buffer (TLB)

Address Mappings & TLB

What if the TLB needs a mapping it doesn't have?

Software managed TLB

- It generates a TLB-miss fault which is handled by the operating system (like interrupt or trap handling)
- The operating system looks in the page tables, gets the mapping from the right entry, and puts it in the TLB

Hardware managed TLB

- It looks in a pre-specified physical memory location for the appropriate entry in the page table
- The hardware architecture defines where page tables must be stored in physical memory
- OS must load current process page table there on context switch!

The UNIX (32 bits) Architecture

Page size

4 Kbytes = 2^{12} bytes -->12 bits offset

Virtual addresses (“logical addresses”)

32 bits --> 4 Gbyte virtual address space

2^{20} Pages --> 20 bits for page number

The UNIX Architecture

Page size

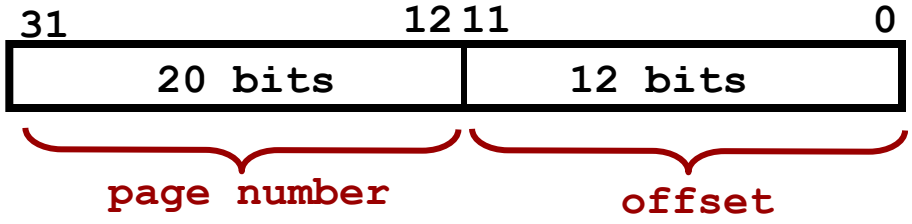
4 Kbytes

Virtual addresses ("logical addresses")

32 bits --> 4 Gbyte virtual address space

2^{20} Pages --> 20 bits for page number

An address:



The UNIX Architecture

Physical addresses

- 32 bits --> 4 Gbyte installed memory (max)
- 2^{20} Frames --> 20 bits for frame number

The UNIX Page Table

An array of "*page table entries*"

Kept in memory

2^{20} pages in a virtual address space?

---> 1M entries in the table

Each entry is 4 bytes long

20 bits	The Frame Number
1 bit	Valid Bit
1 bit	Writable Bit
1 bit	Dirty Bit
1 bit	Referenced Bit
8 bits	Unused (and available for OS algorithms)

The UNIX Page Table

Two page table related registers in the CPU

- Page Table Base Register
- Page Table Length Register

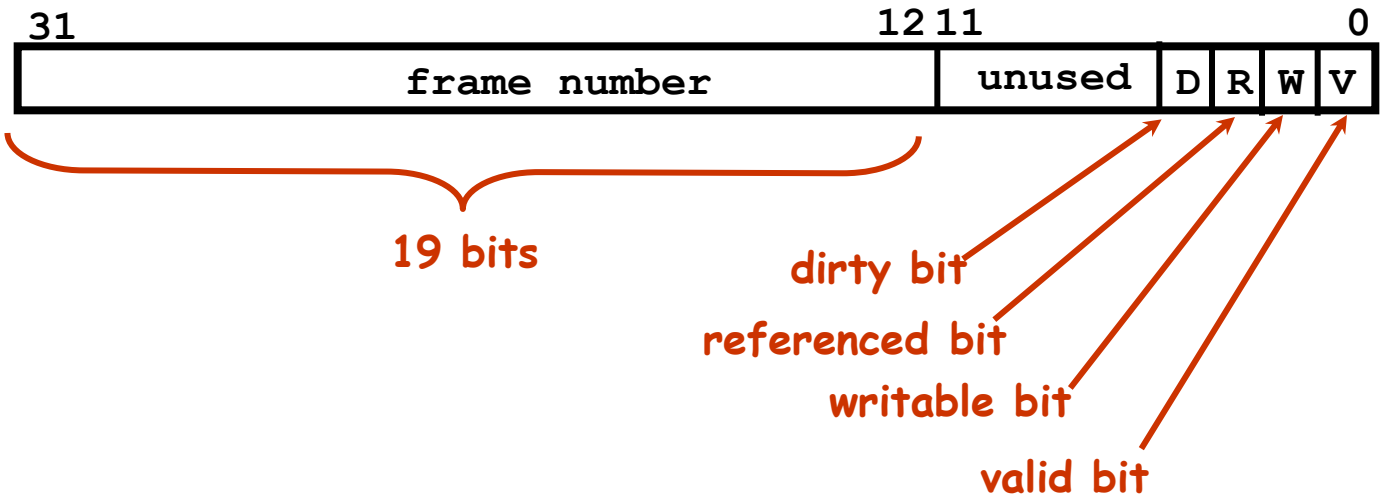
These define the "current" page table

- This is how the CPU knows which page table to use
- Must be saved and restored on context switch

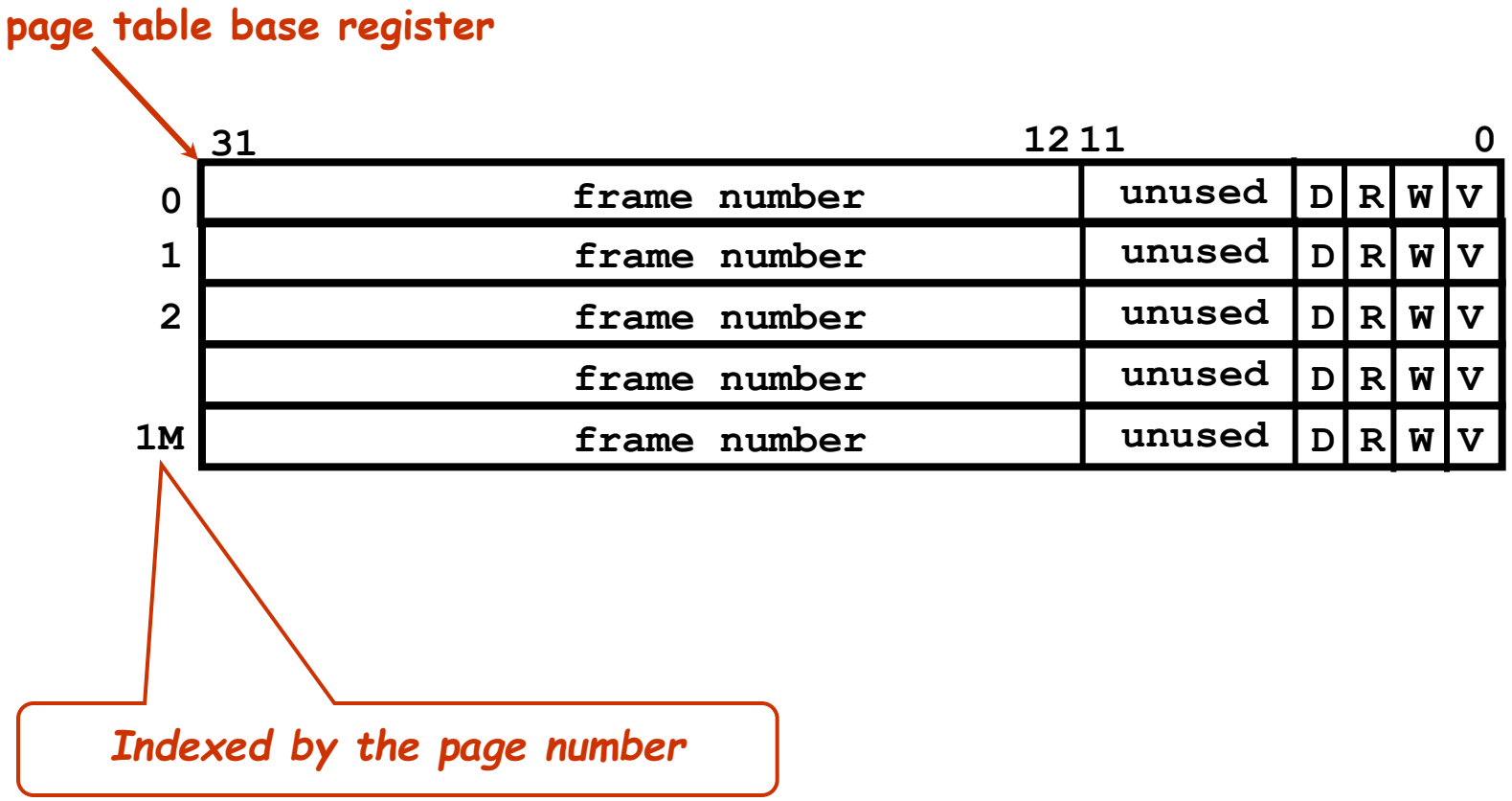
Bits in the CPU status register

- System Mode
- Interrupts Enabled
- Paging Enabled
  - 1 = Perform page table translation for every memory access
  - 0 = Do not do translation

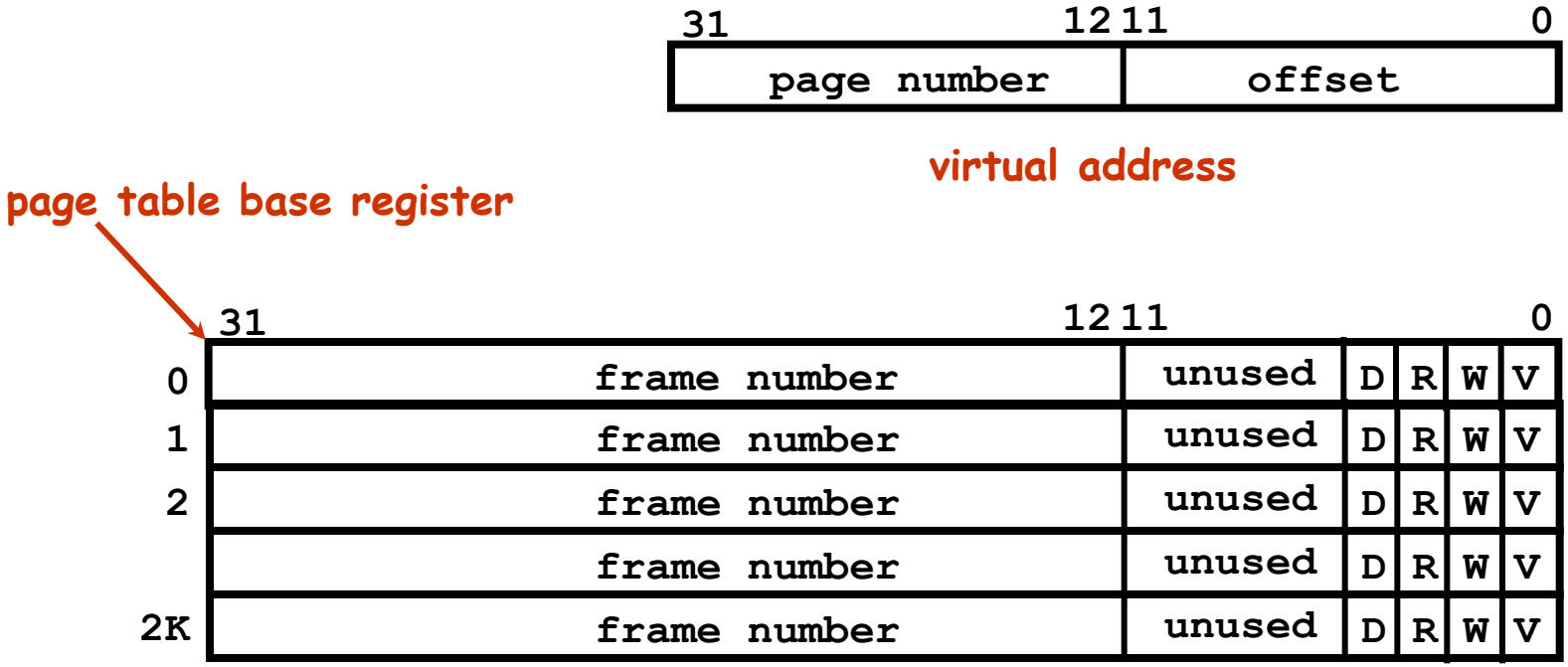
The UNIX Page Table



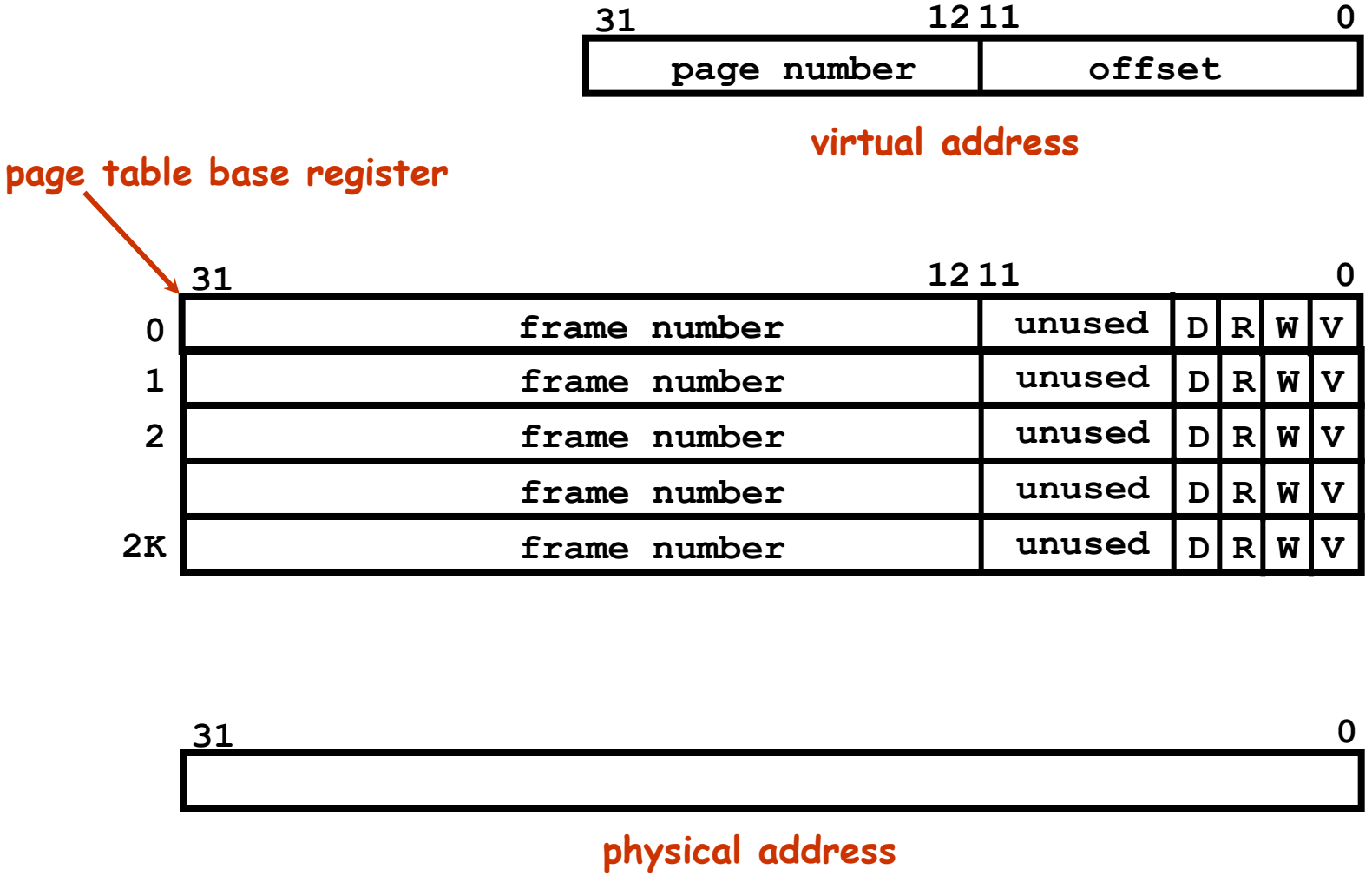
The UNIX Page Table



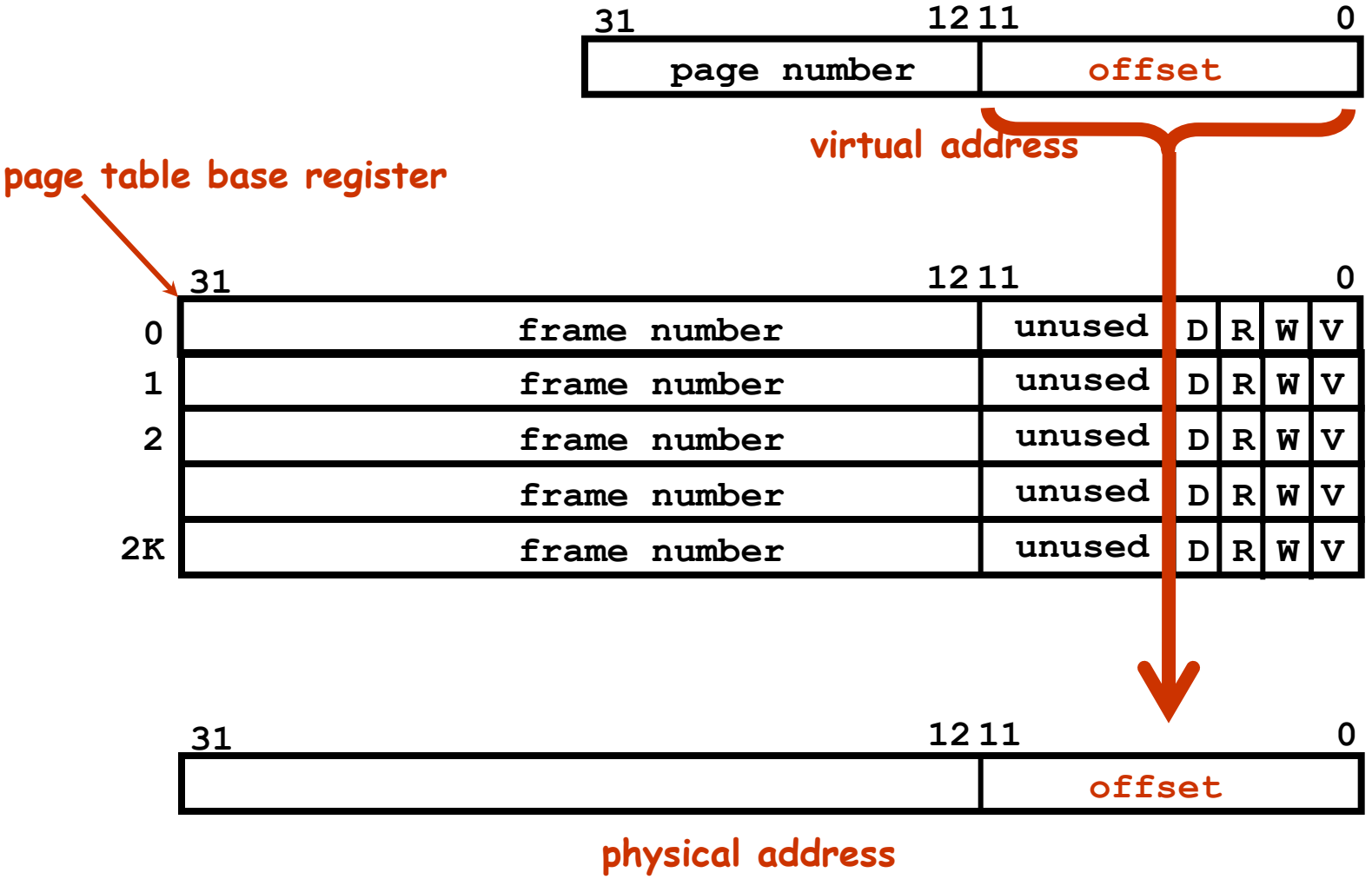
The UNIX Page Table



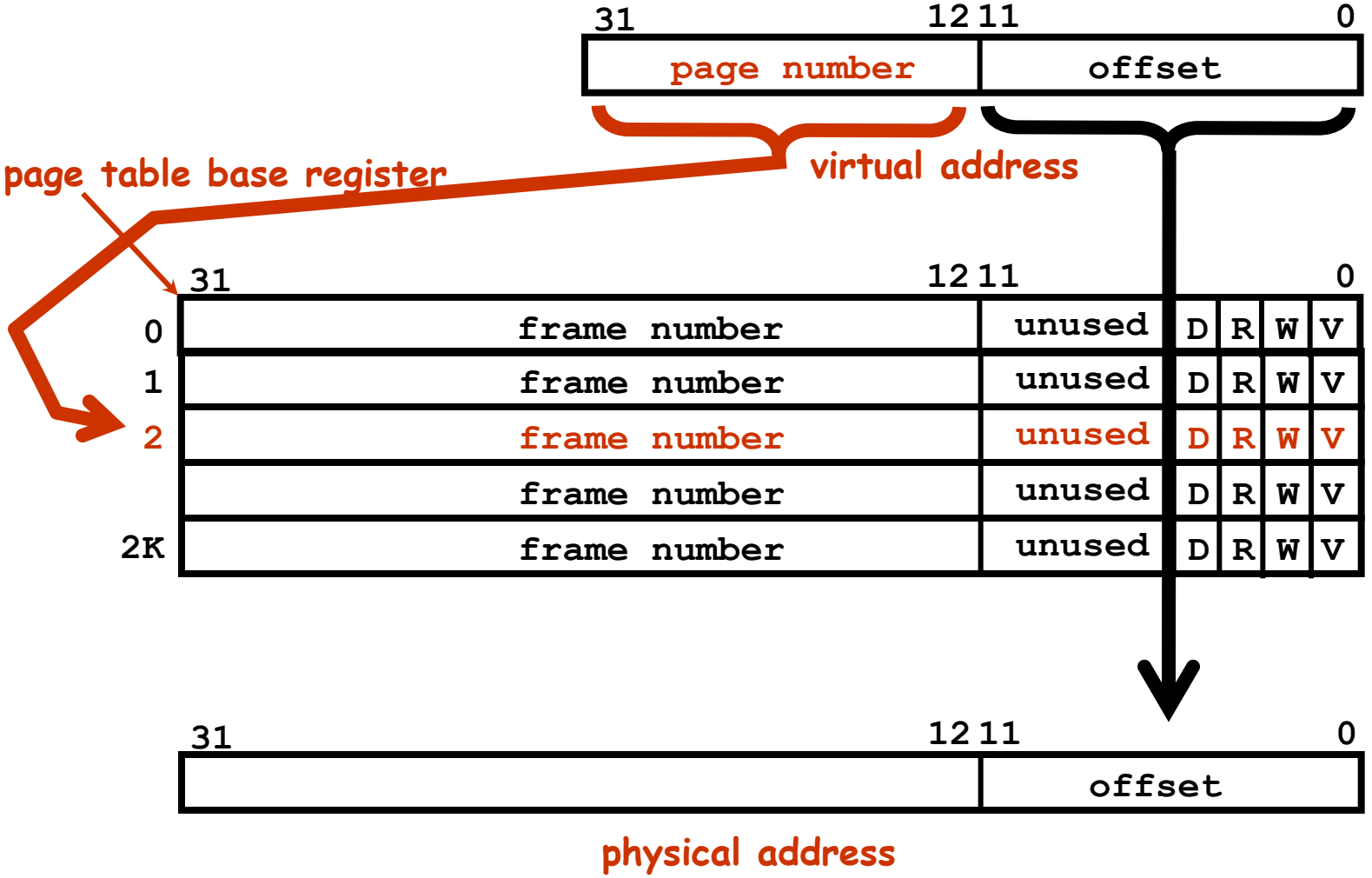
The UNIX Page Table



The UNIX Page Table



The UNIX Page Table



Quiz

What is the difference between a virtual and a physical address?

What is address binding?

Why are programs not usually written using physical addresses?

Why is hardware support required for dynamic address translation?

What is a page table used for?

What is a TLB used for?

How many address bits are used for the page offset in a system with 2KB page size?