Chapter 3

Memory Management

Part 1

Outline of Chapter 3

- Basic memory management
- Swapping
- Virtual memory

in this file

- Page replacement algorithms
- Modeling page replacement algorithms
- Design issues for paging systems
- Implementation issues
- Segmentation

The Memory Hierarchy

Ideally programmers want memory that is...

Large
Fast
Non volatile

Memory Hierarchy

Small amount of fast, expensive memory -- cache
Some medium speed, medium priced -- main memory
Gigabytes of slow, cheap storage -- disk

Memory Manager manages the hierarchy

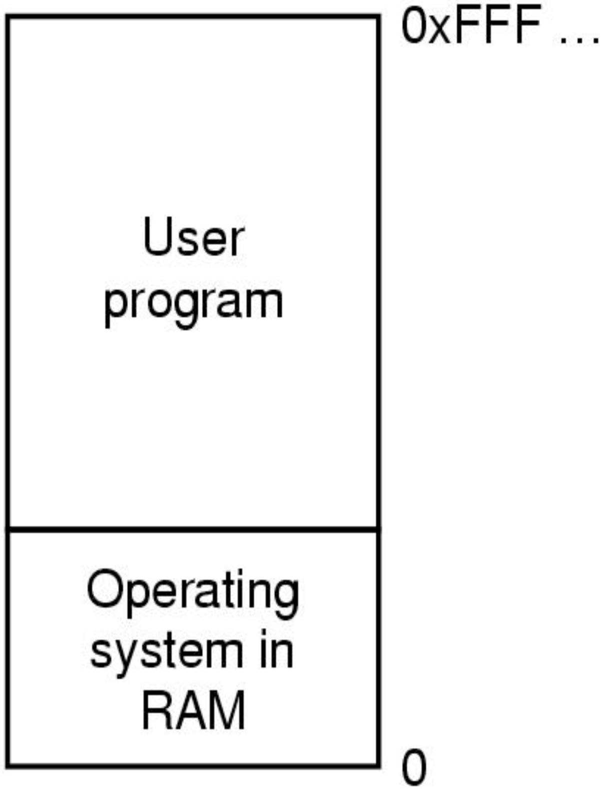
Simplest Memory Organization

Monoprogramming

- One user program at a time
- Plus the OS

No protection

Simplest Memory Organization

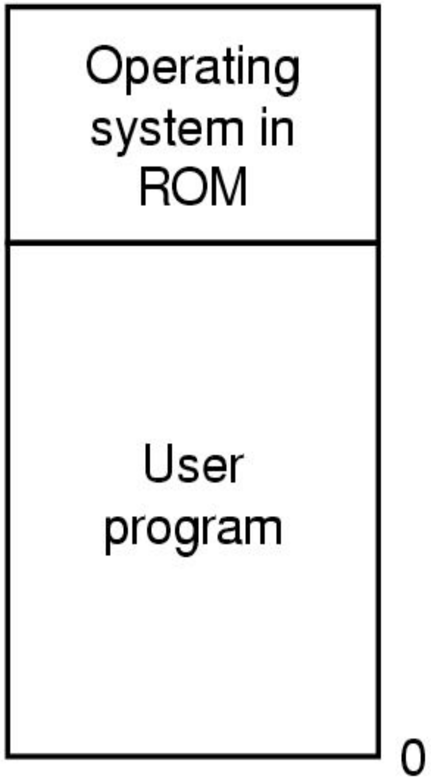

0xFFF …

User program

Operating system in RAM

0

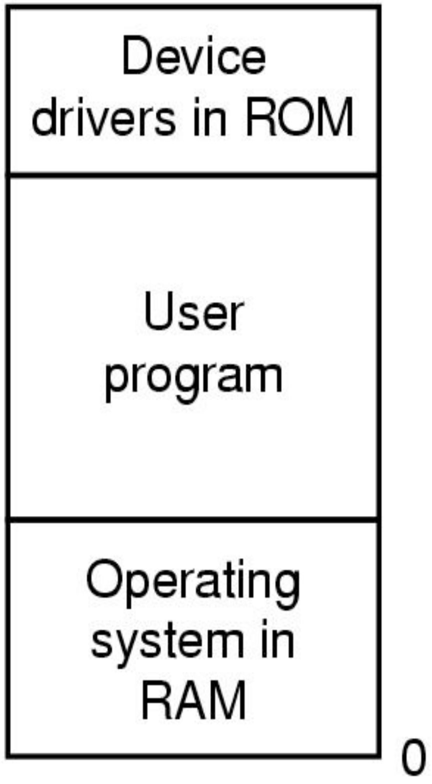
Mainframes


Operating system in ROM

User program

0

Palm OS


Device drivers in ROM

User program

Operating system in RAM

0

MS Dos

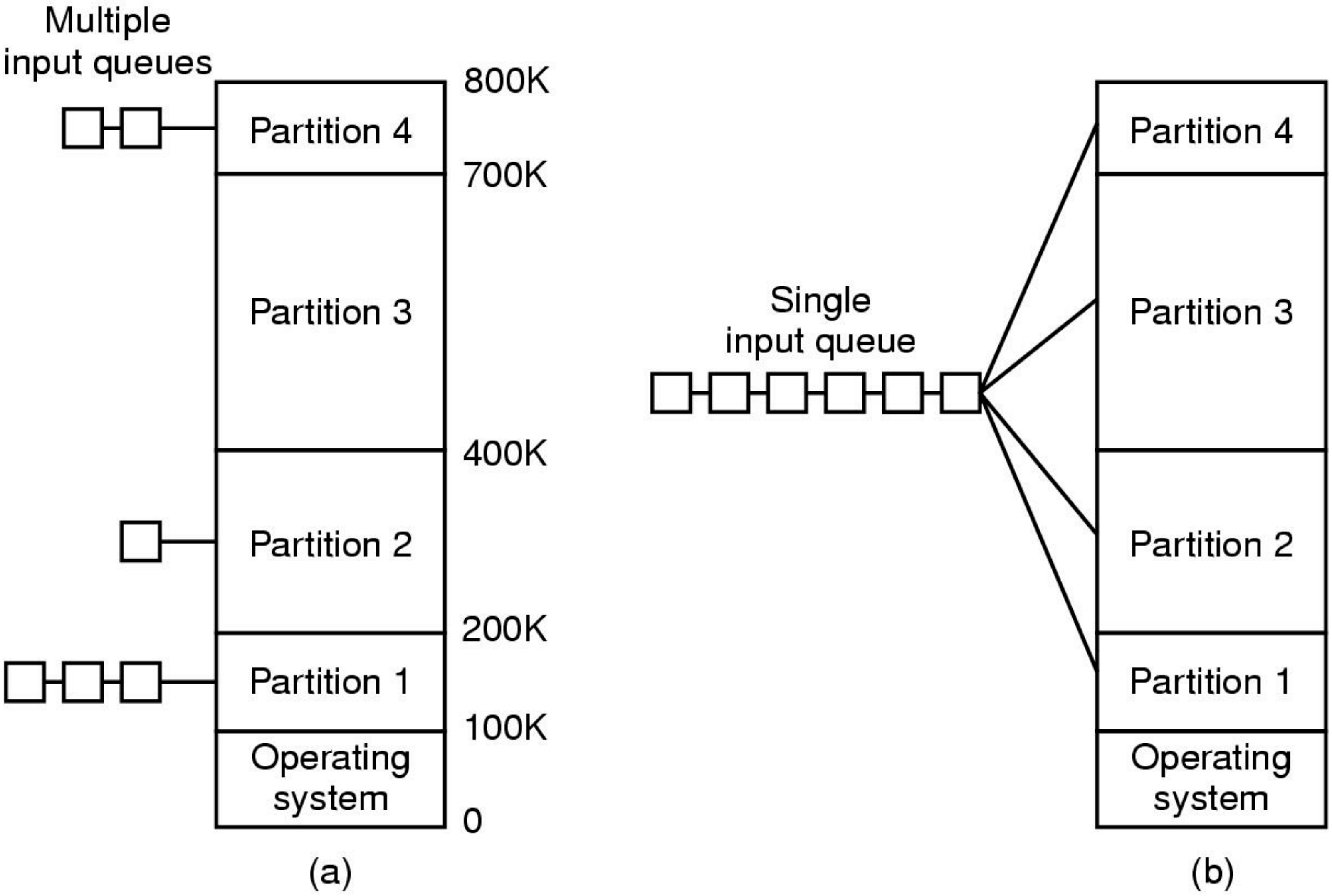
Multiprogramming with Fixed Partitions

Main memory divided into “partitions”
    Done once, e.g., at startup

To run a program...
    Select a partition
        (Must find one that is large enough)
    Put program into partition

Not enough memory for all runnable programs?
    The “input queue”
        A list of programs waiting to be run

Multiprogramming with Fixed Partitions



Modelling Multiprogramming

Assume each process...

20% - computing

80% - waiting on I/O

Each process spends some fraction of time waiting

$p = .8$

Many processes are running

N = Degree of multiprogramming (e.g., N = 5 procs)

The probability that all processes are waiting for I/O

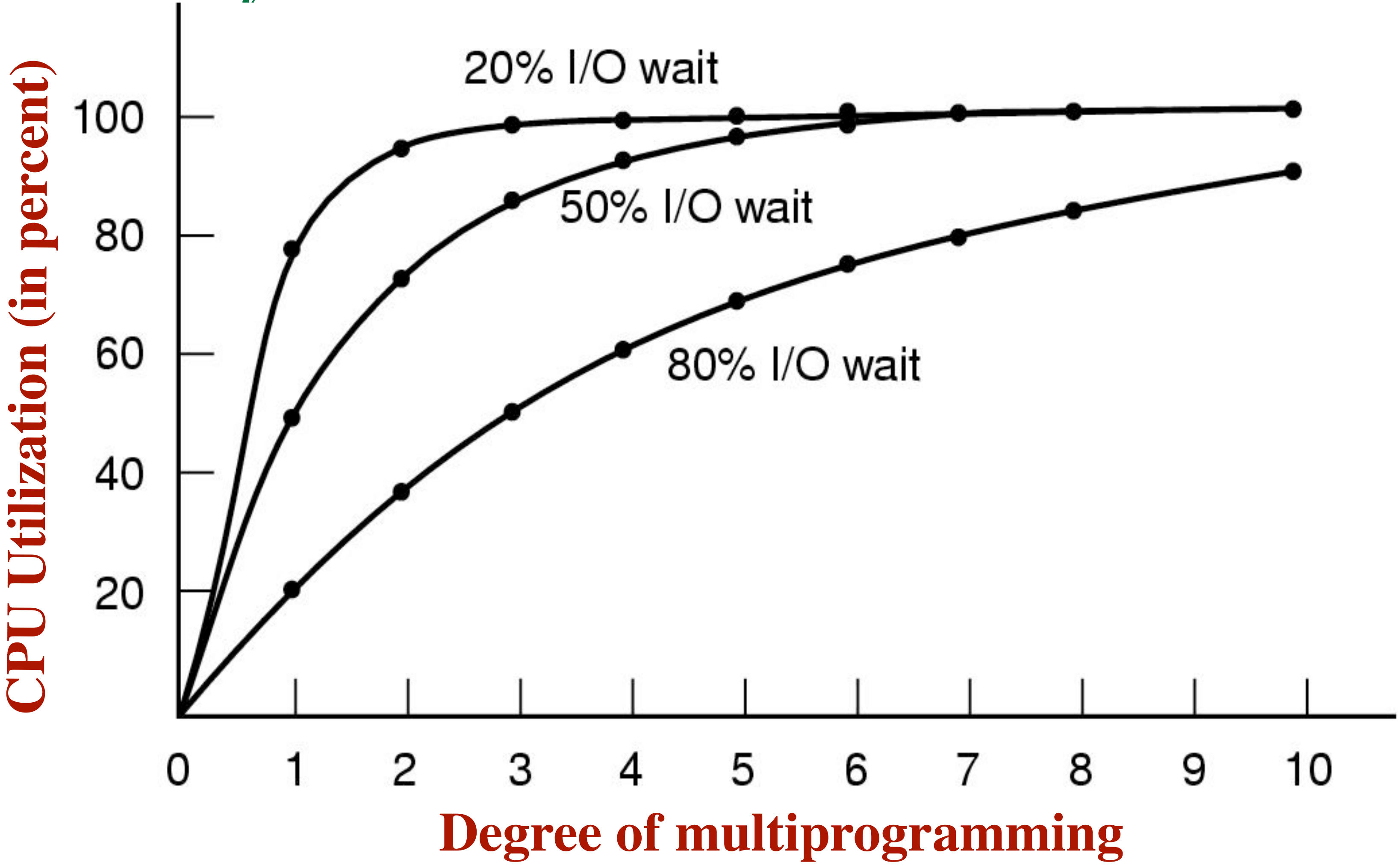
p^N

CPU Utilization

$1 - p^N$

CPU Utilization

CPU utilization as a function of number of processes in memory:



Analysis of System Performance

Example:

4 jobs
Each job has 20% compute and 80% wait time
Each job arrives at a different time

Job	Arrival Time	Total CPU minutes needed
1	10:00	4
2	10:10	3
3	10:15	2
4	10:20	2

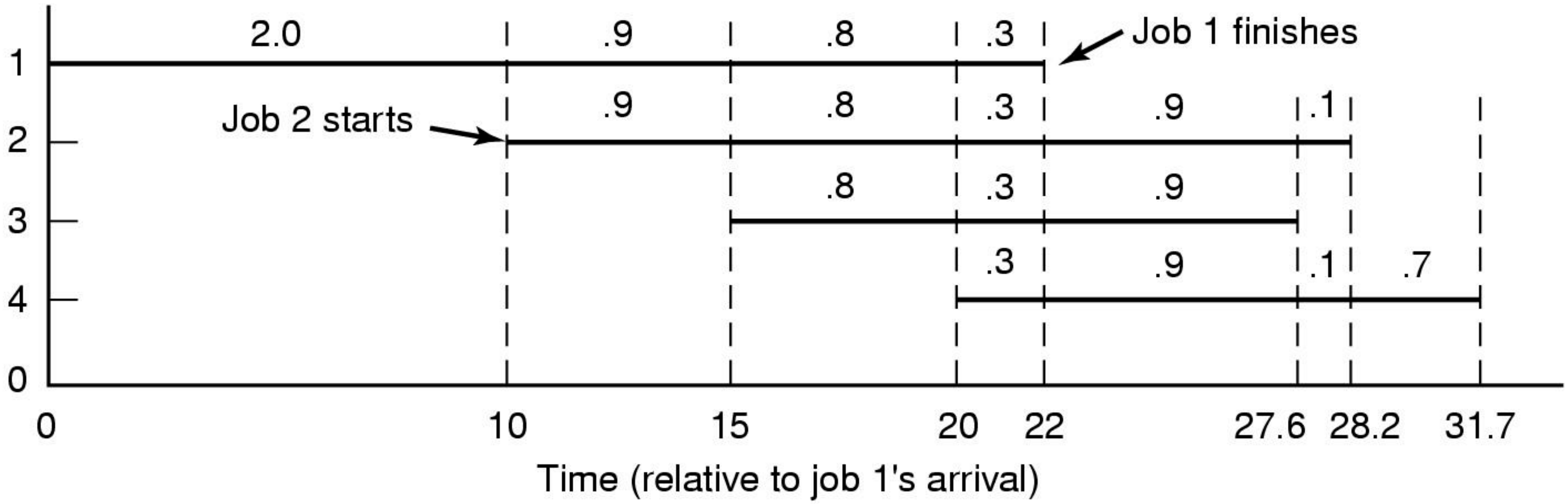
Analysis of System Performance

Job	Arrival time	CPU minutes needed
1	10:00	4
2	10:10	3
3	10:15	2
4	10:20	2

(a)

	# Processes			
	1	2	3	4
CPU idle	.80	.64	.51	.41
CPU busy	.20	.36	.49	.59
CPU/process	.20	.18	.16	.15

(b)



(c)

Relocation and Protection

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

Compiler produces code containing embedded addresses.
These addresses cannot be absolute!

Linker combines pieces of the program.
Assumes the program will be loaded at address 0.

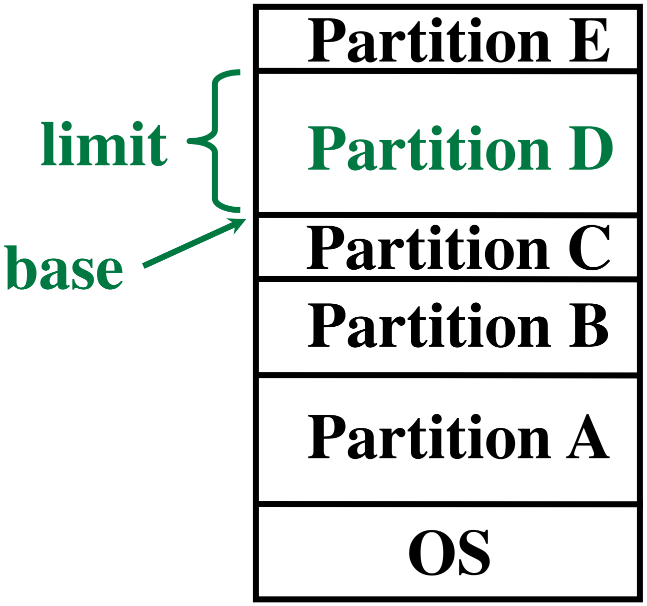
Option 1:
Modify the addresses at load-time

Option 2:
Modify the addresses at run-time

Protection:
Keep program A out of program B's partition

Base and Limit Registers

- The simplest scheme
- These 2 registers describe a partition.
- Every address generated at runtime...
  - Compare to the limit register (& abort if larger)
  - Add to the base register to give physical memory address
- Multiprogramming
  - Each process is in a partition
  - Context switch?
    - Load new values into
      - base and limit



Swapping

When a program is running...

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

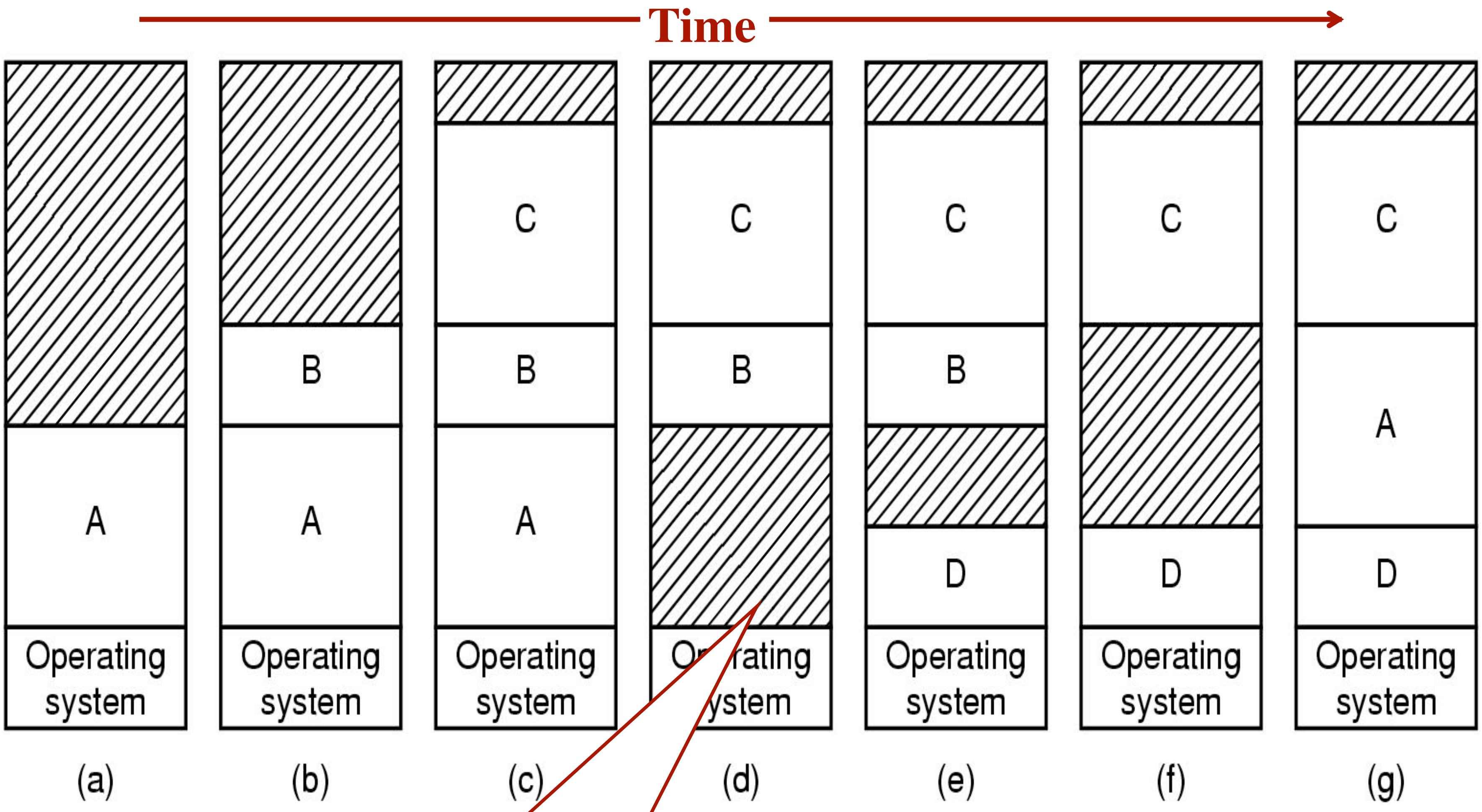
When the program is not running...

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

Over time...

- Programs come into memory
- Programs leave memory
  get "swapped out"

Swapping



Shaded regions are unused

Swapping

Programs may want to grow during execution.

More room for stack

More room for heap allocation

...Etc...

Problem:

The partition is too small.

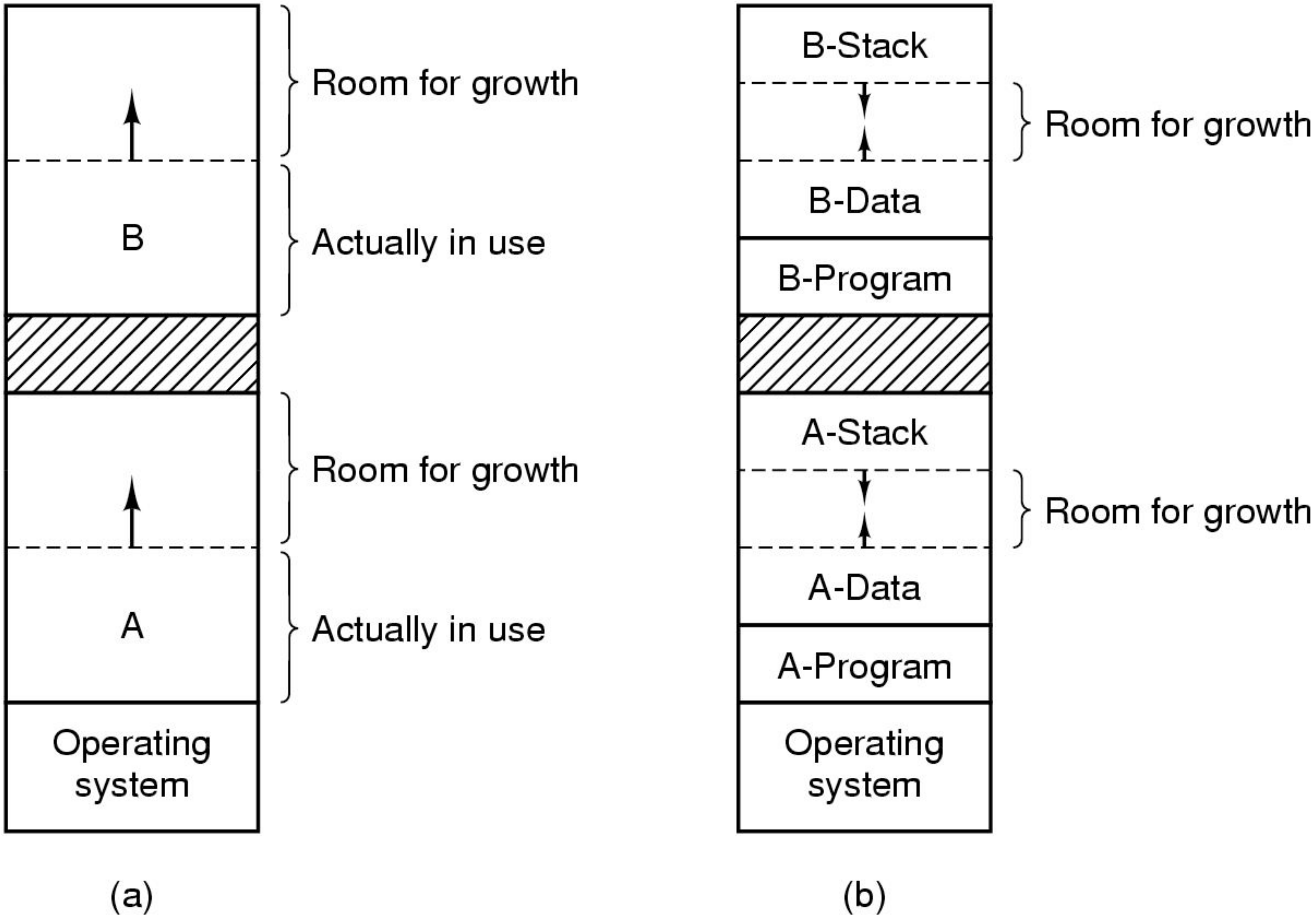
Must move programs around... Ugh!

Idea:

Make the partitions a little larger than necessary.

Can accommodate some growth easily!

Swapping


Room for growth
B
Actually in use
Room for growth
A
Actually in use
Operating system
(a)
B-Stack
Room for growth
B-Data
B-Program
A-Stack
Room for growth
A-Data
A-Program
Operating system
(b)

Managing Memory

Divide main memory into chucks

Bytes <----> pages

Each chunk is either

- **Unused ("free")**
- **Used by some process**

Operations

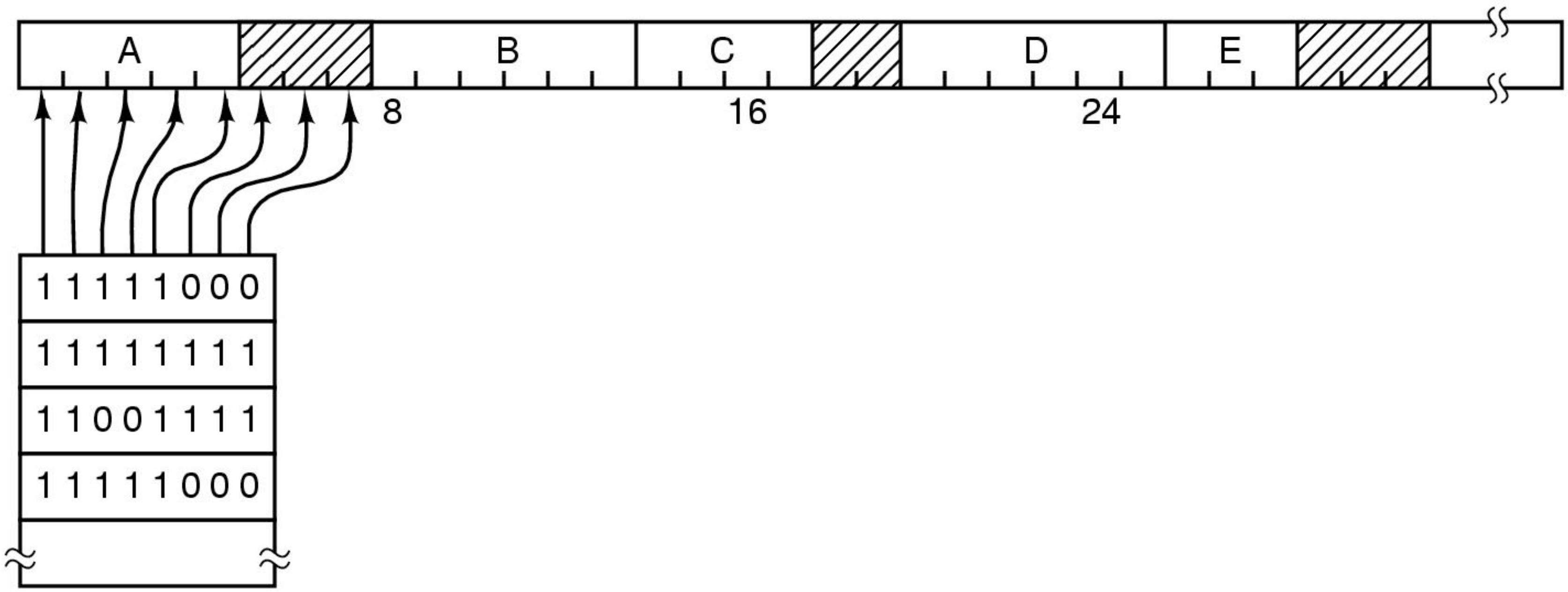
- **Find a chunk of unused memory (big enough to hold a new process)**
- **Return a chunk of memory to the free pool (after a process terminates / is swapped out)**

Managing Memory with Bit Maps

Technique #1: Bit Map

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

Managing Memory with Bit Maps



Managing Memory with Linked Lists

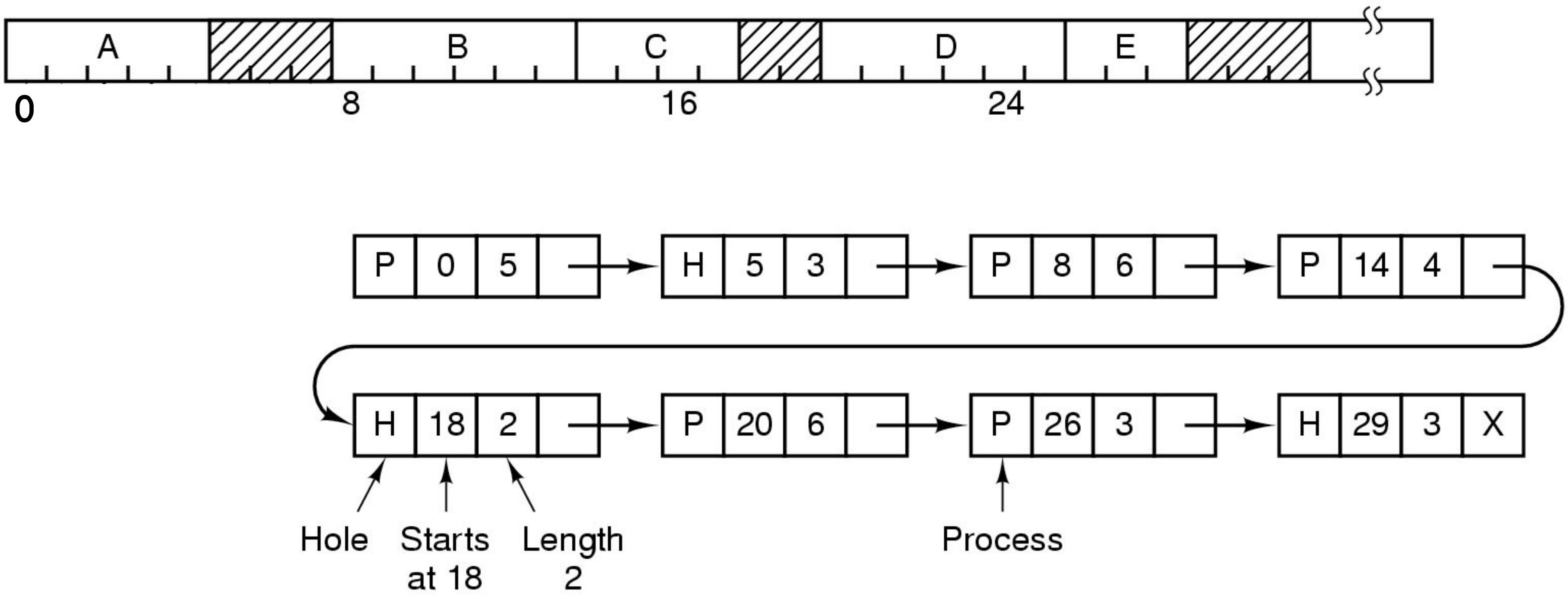
Technique #2: Linked List

Keep a list of elements

Each element describes one chunk of memory

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

Managing Memory with Linked Lists



Merging Holes

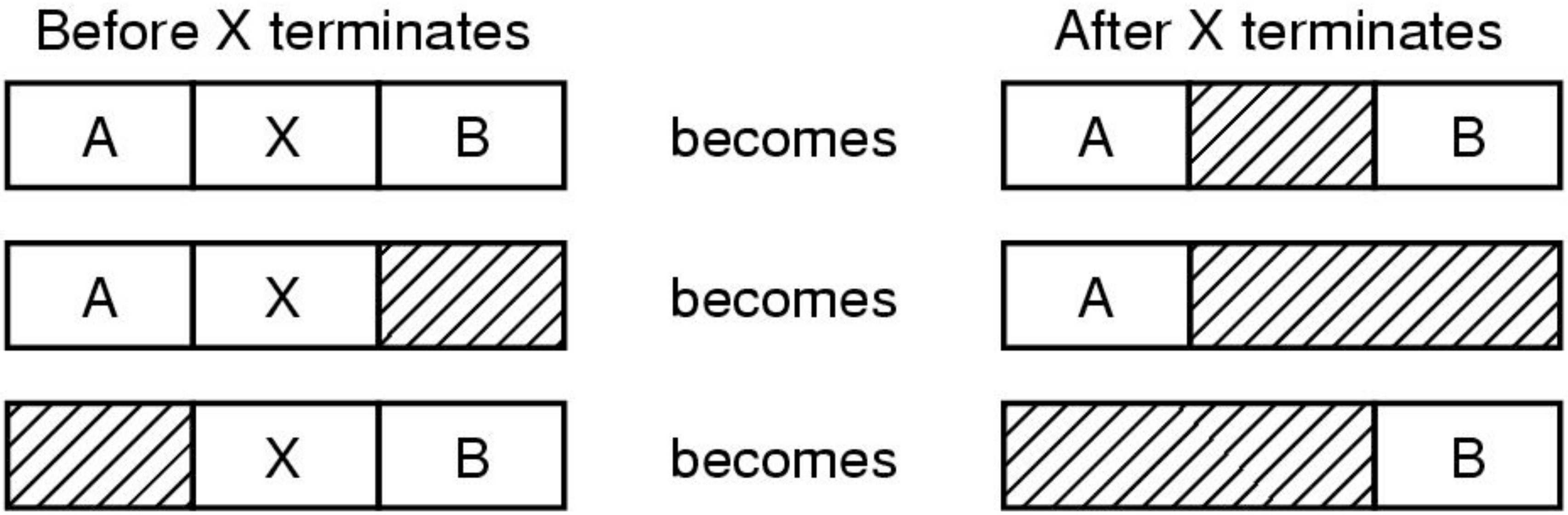
Whenever a chunk of memory is freed…
we want to merge adjacent holes!

Merging Holes

Whenever a chunk of memory is freed...
we want to merge adjacent holes!

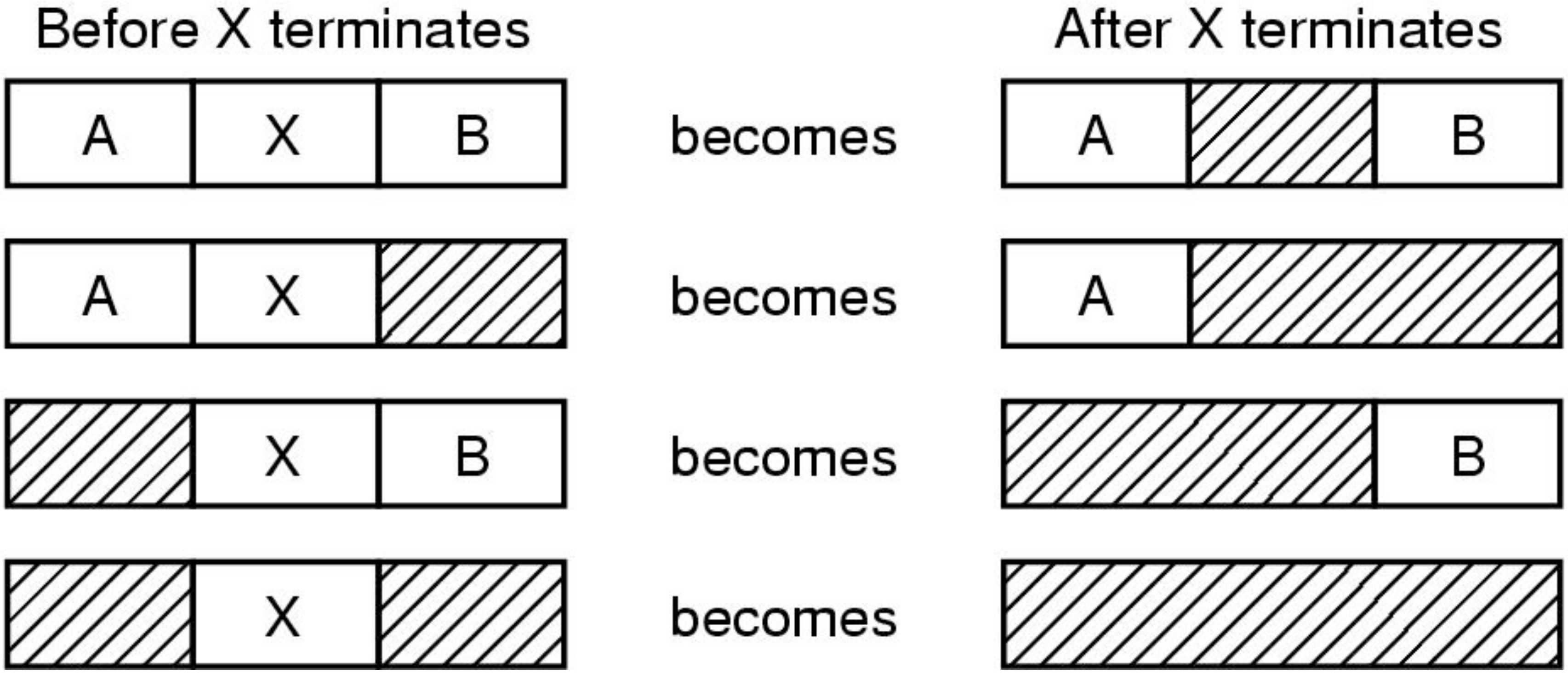
Merging Holes

Whenever a chunk of memory is freed...
we want to merge adjacent holes!



Merging Holes

Whenever a chunk of memory is freed...
we want to merge adjacent holes!



Managing Memory with Linked Lists

Need to find a hole? (...big enough for some new process)
Search the list!

- First Fit
- Next Fit
  Start from current location in the list
  Not as good as first fit
- Best Fit
  Find the smallest hole that will work
  Tends to create lots of little holes
- Worst Fit
  Find the largest hole (remainder will be big)
  No good
- Quick Fit
  Keep separate lists for common sizes

Virtual Memory

With Swapping

- The entire process must be in memory
- Can’t run a program larger than physical memory!

Virtual Memory

With Swapping

- The entire process must be in memory
- Can’t run a program larger than physical memory!

With Virtual Memory

- Put only part of the program in memory.
- Can run a program larger than physical memory!

Virtual Memory

With Swapping

- The entire process must be in memory
- Can’t run a program larger than physical memory!

With Virtual Memory

- Put only part of the program in memory.
- Can run a program larger than physical memory!

The “*working set*” idea:

- Normally, programs do not access all of their memory
- Accesses tend to be concentrated within small regions
- You only really need 16K to run a 16M program

Virtual Memory

With Swapping

- The entire process must be in memory
- Can’t run a program larger than physical memory!

With Virtual Memory

- Put only part of the program in memory.
- Can run a program larger than physical memory!

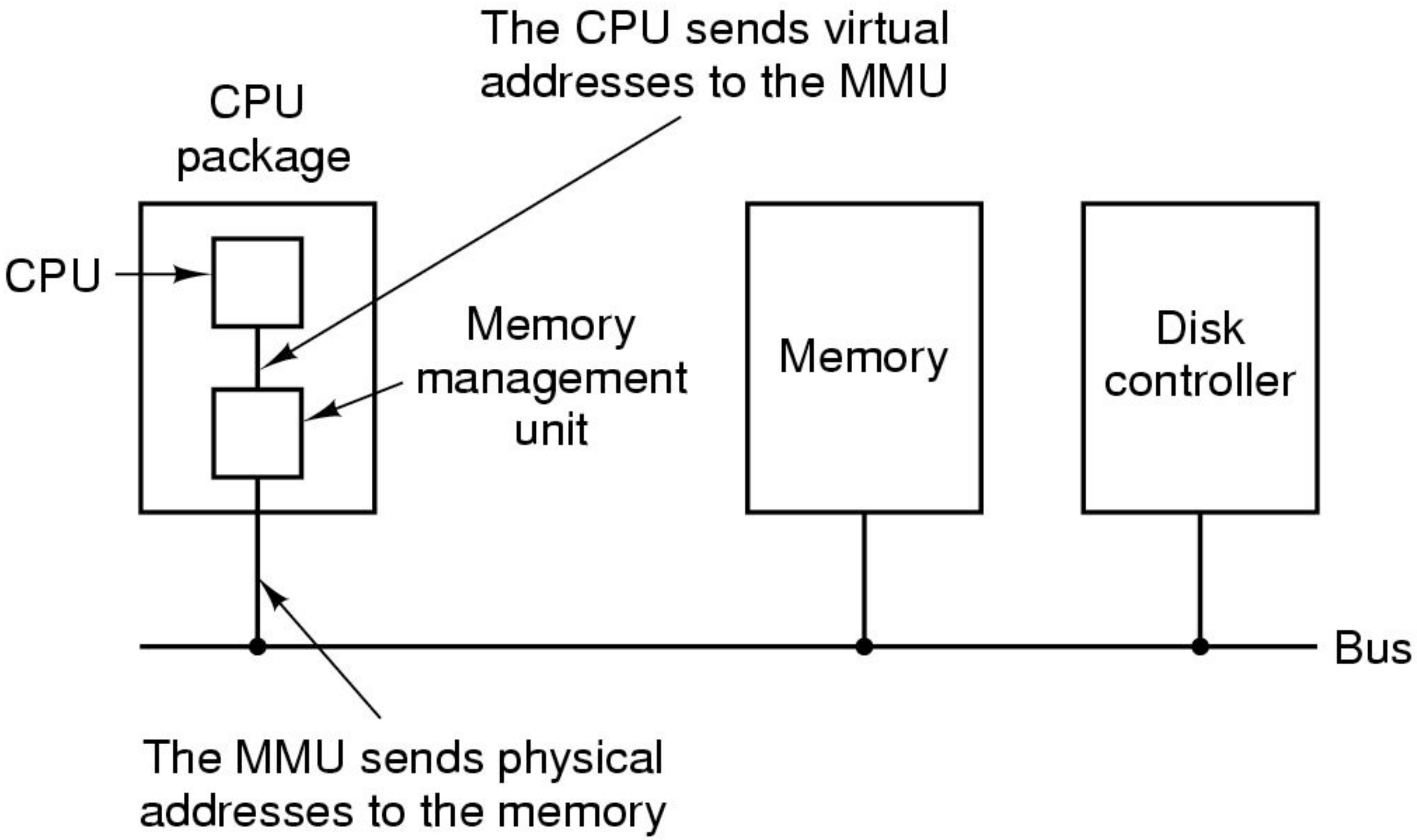
The “*working set*” idea:

- Normally, programs do not access all of their memory
- Accesses tend to be concentrated within small regions
- You only really need 16K to run a 16M program

The real benefit:

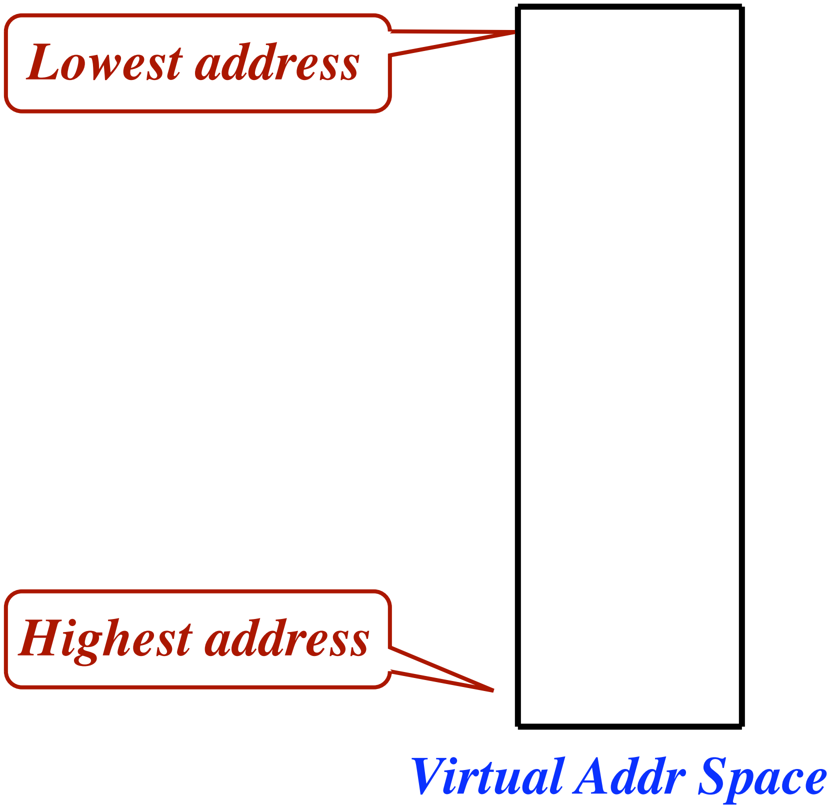
Can get more runnable processes into memory at once!

Memory Management Unit (MMU)

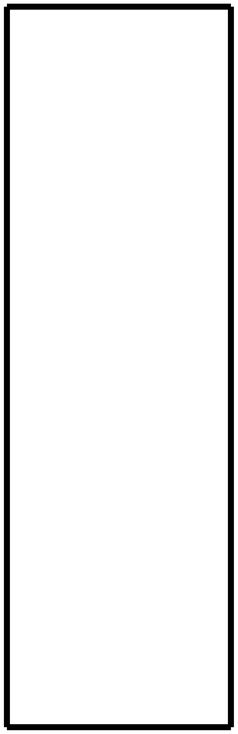


Virtual Address Spaces and the Page Table

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



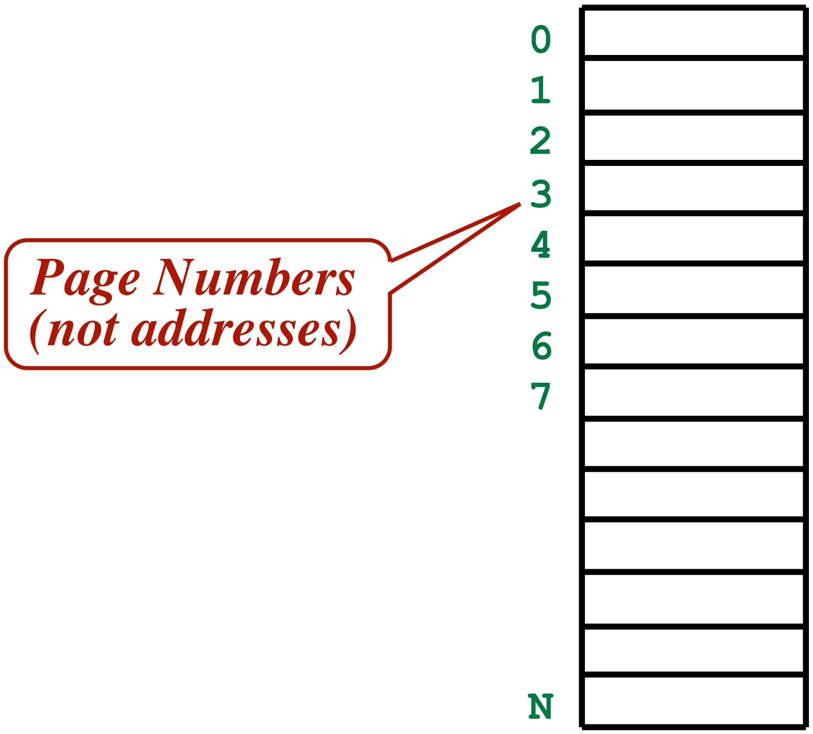
Virtual Address Spaces and the Page Table



Virtual Address Spaces and the Page Table

The address spaces is divided into "pages"

In BLITZ, the page size is 8K

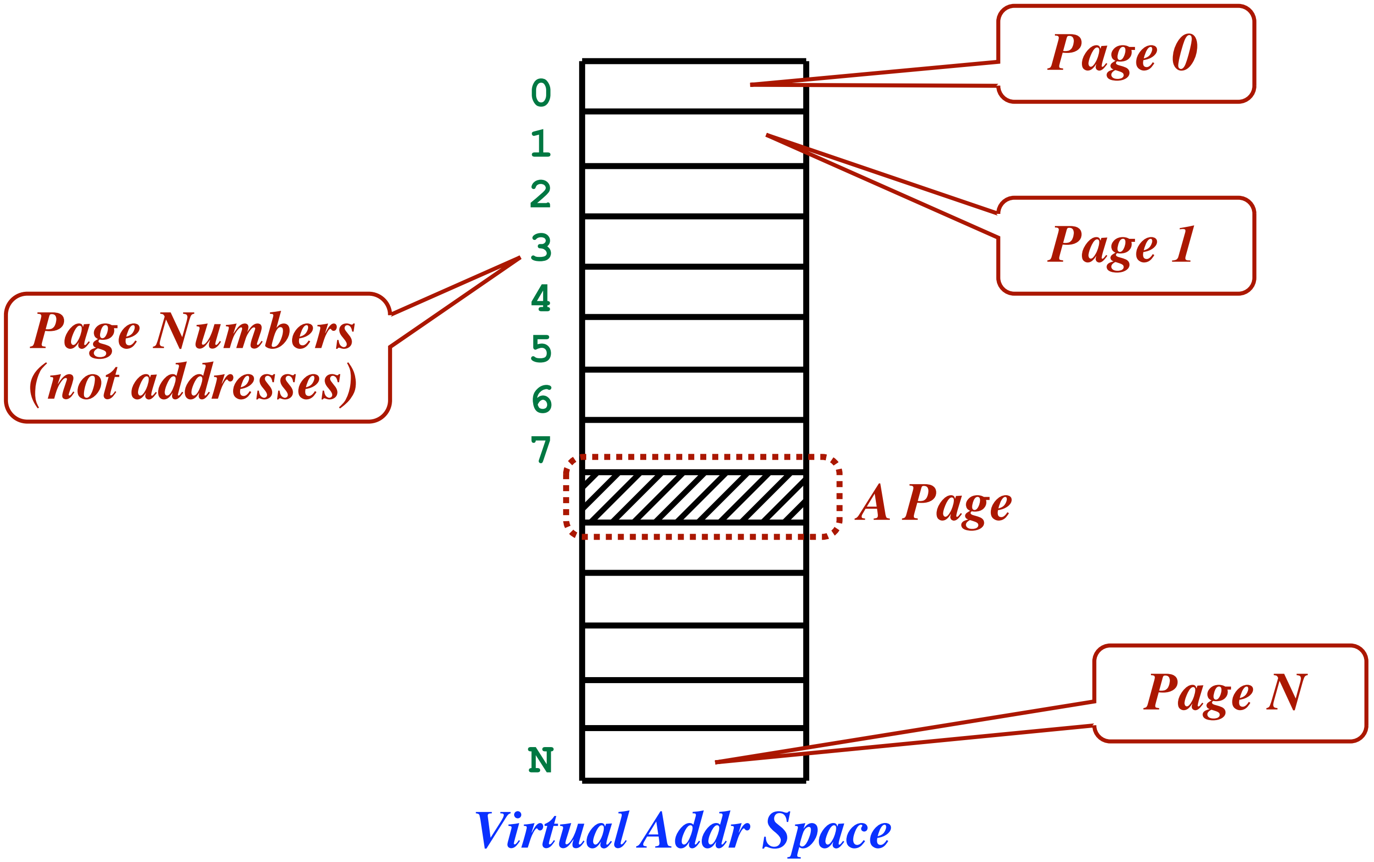


Virtual Addr Space

Virtual Address Spaces and the Page Table

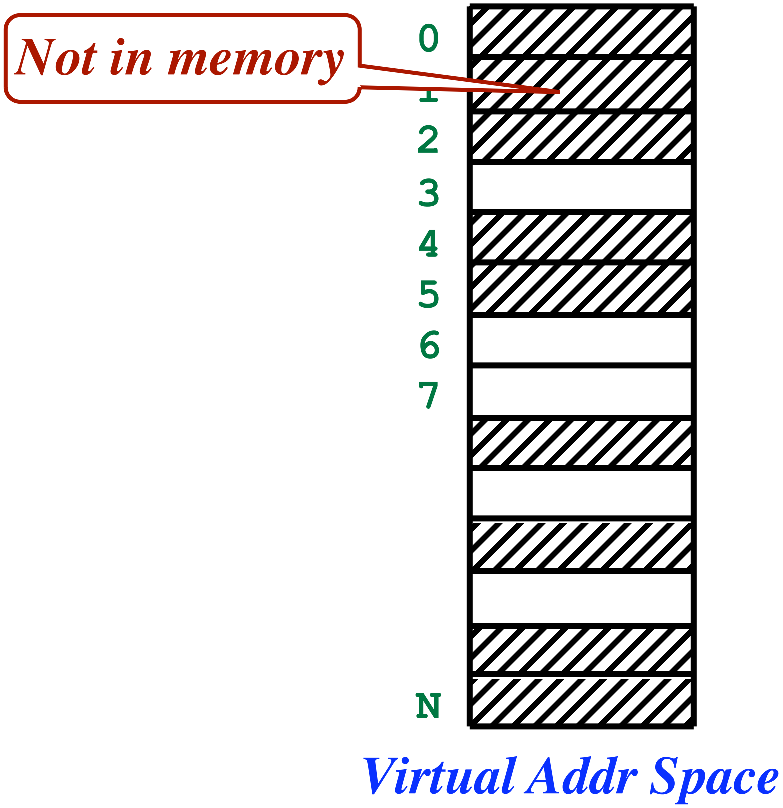
The address spaces is divided into “pages”

In BLITZ, the page size is 8K



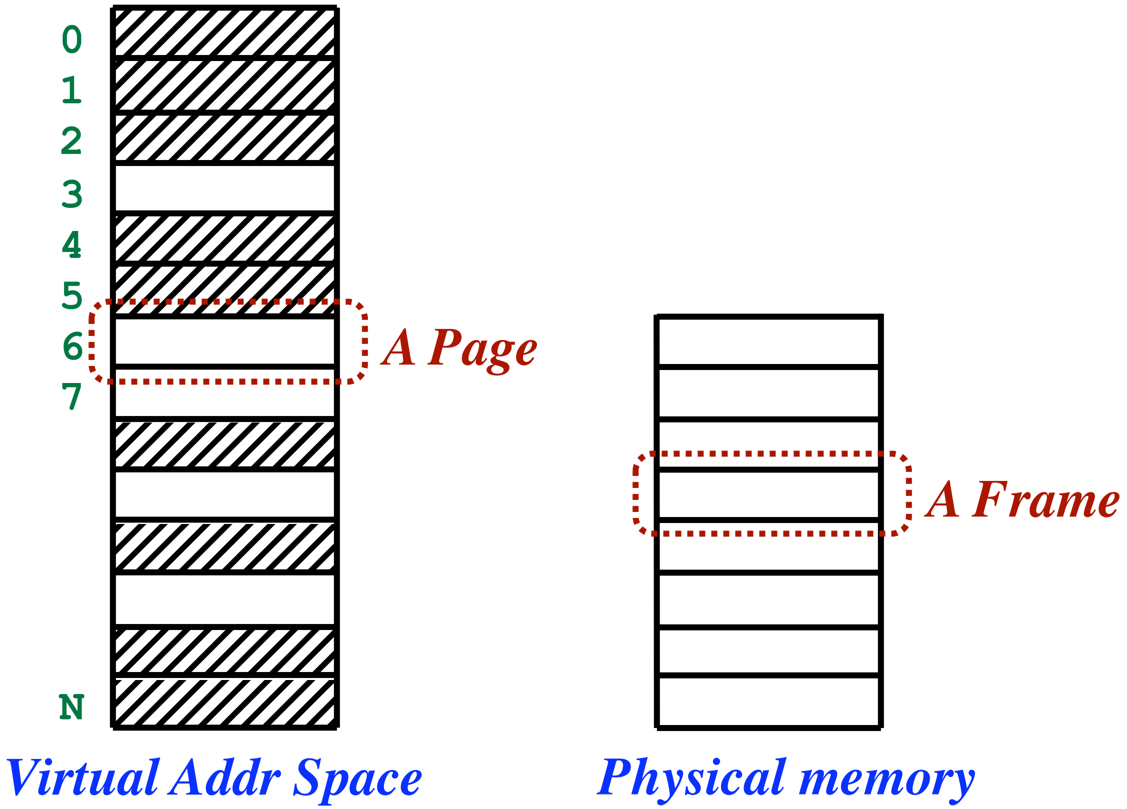
Virtual Address Spaces and the Page Table

In reality, only some of the process's pages are in memory.



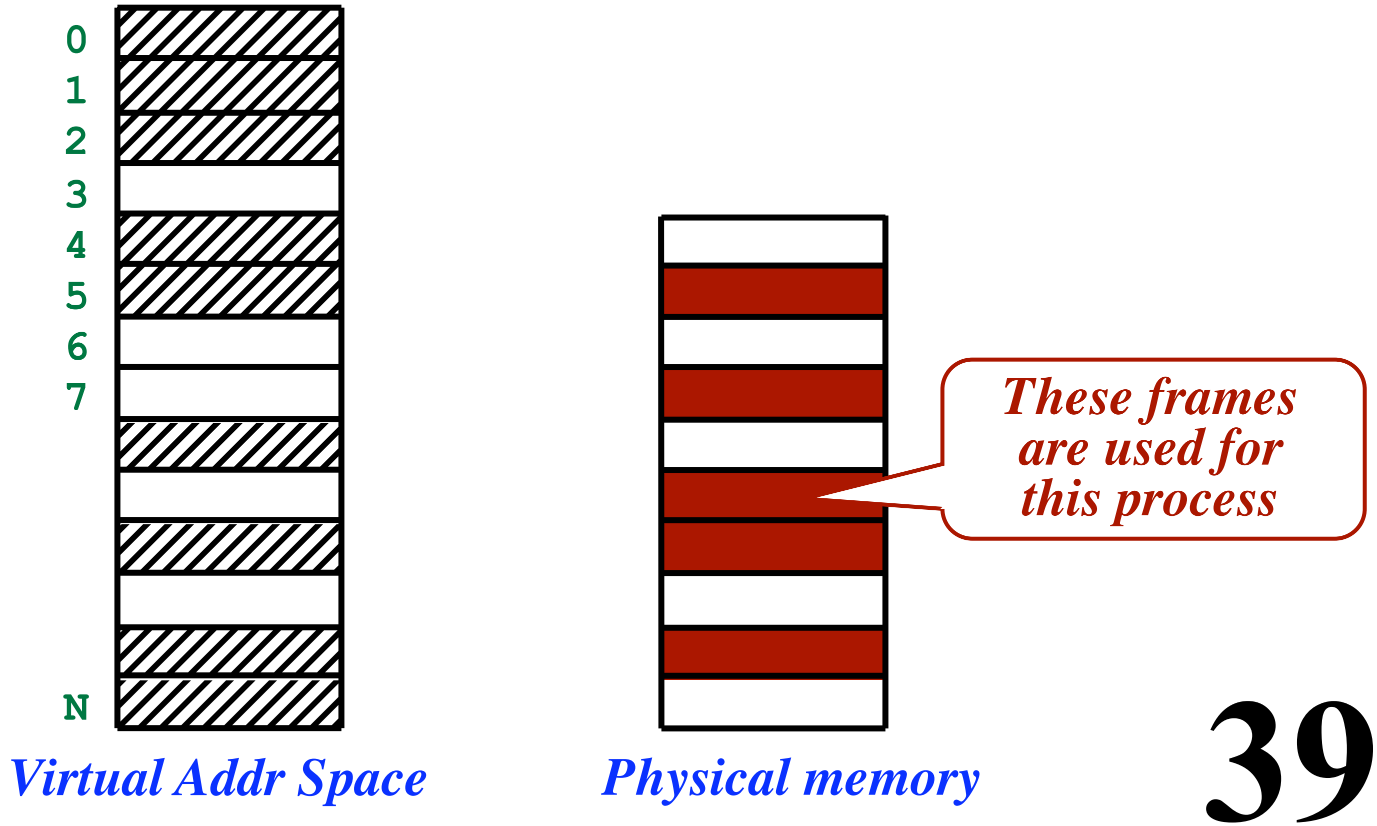
Virtual Address Spaces and the Page Table

Physical memory is divided into "*page frames*"
(Page size = frame size)



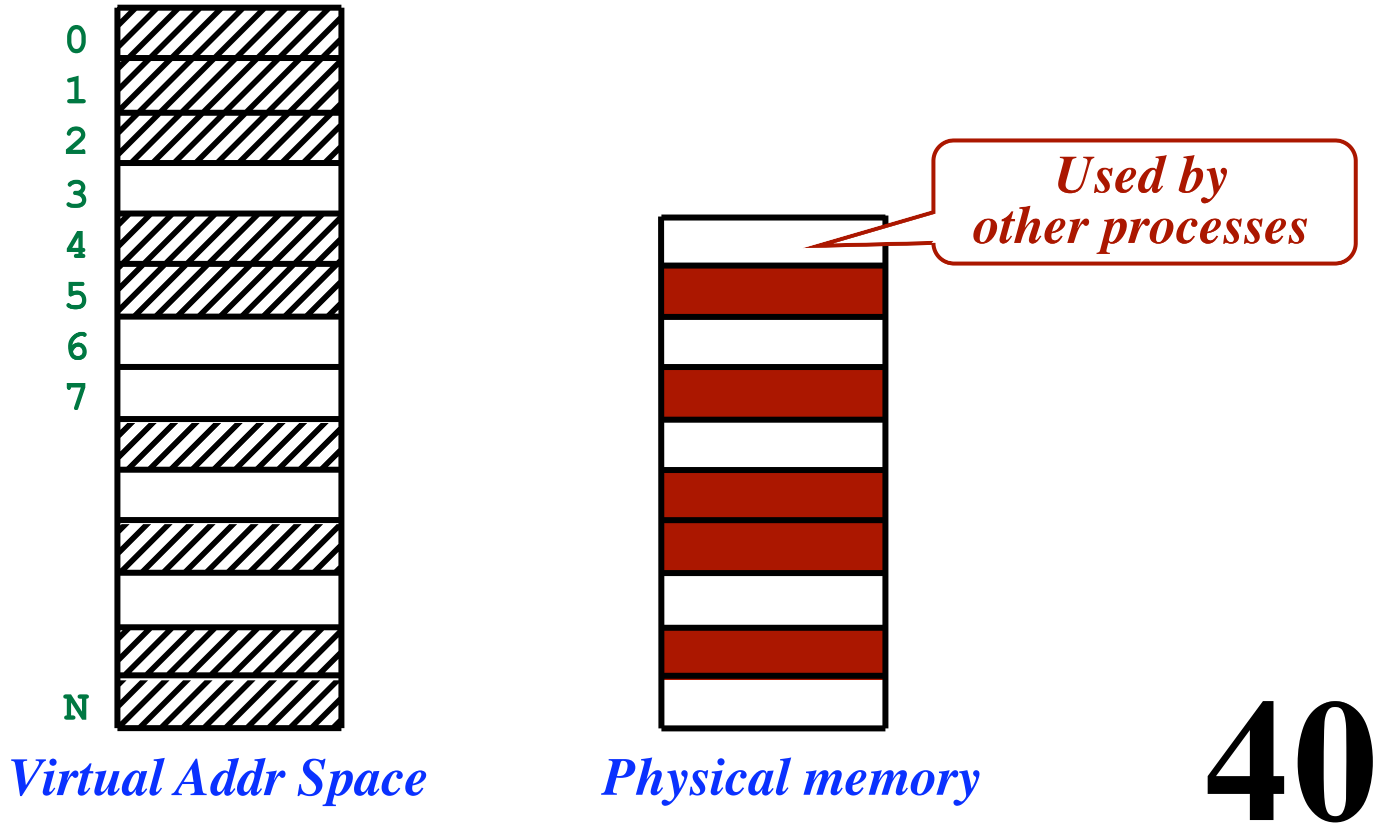
Virtual Address Spaces and the Page Table

Some page frames are used for this process.



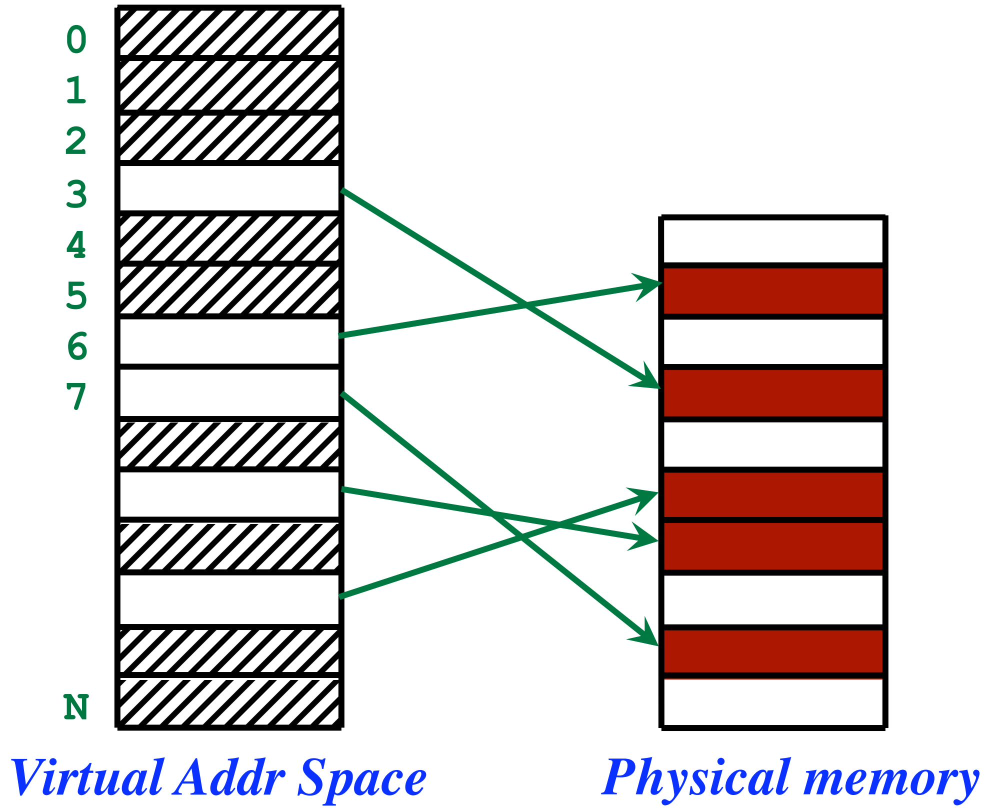
Virtual Address Spaces and the Page Table

Some page frames are used for this process.



Virtual Address Spaces and the Page Table

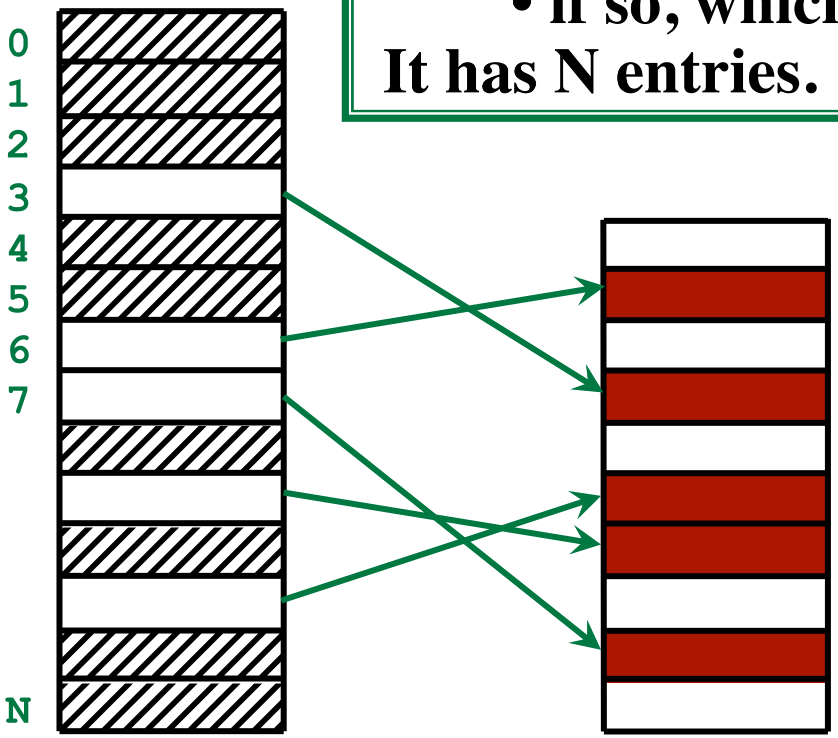
A “mapping” tells which frame holds which page



Virtual Address Spaces and the Page Table

The "*Page Table*" tells
for each page...
- is it in memory
- if so, which frame

It has N entries.



Virtual Addr Space

Physical memory

The BLITZ Architecture

Page Size

8 Kbytes

The BLITZ Architecture

Page Size

8 Kbytes

Logical Addresses (“virtual addresses”)

24 bits --> 16 Mbyte virtual address space

An address:

The BLITZ Architecture

Page Size

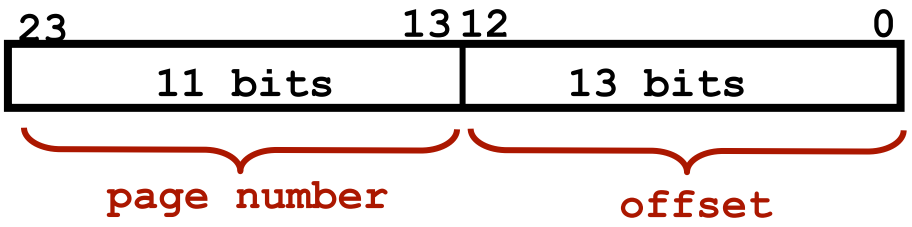
  8 Kbytes

Logical Addresses ("virtual addresses")

  24 bits --> 16 Mbyte virtual address space

  2K Pages --> 11 bits

An address:



23 13 12 0

11 bits | 13 bits

page number

offset

The BLITZ Architecture

Page Size

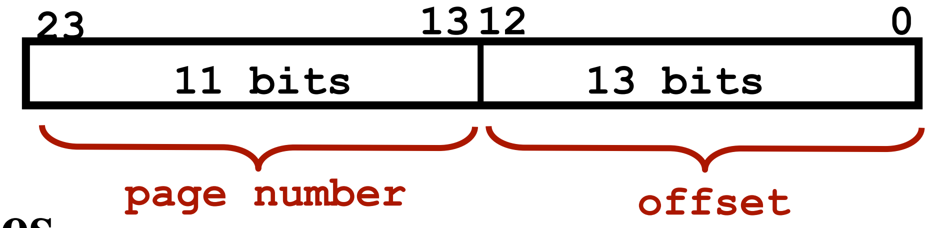
  8 Kbytes

Logical Addresses ("virtual addresses")

  24 bits --> 16 Mbyte virtual address space

  2K Pages --> 11 bits

An address:



Physical Addresses

  32 bits --> 4 Gbyte installed memory (max)

The BLITZ Architecture

Page Size

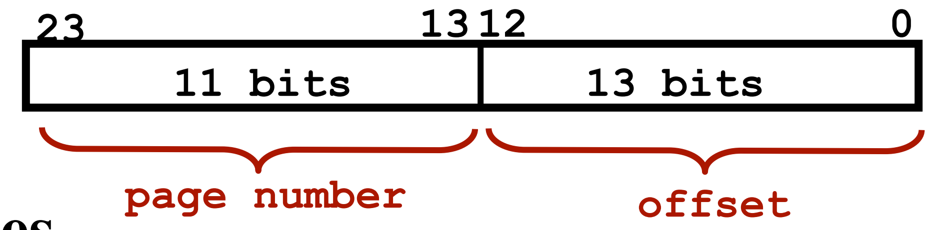
8 Kbytes

Logical Addresses ("virtual addresses")

24 bits --> 16 Mbyte virtual address space

2K Pages --> 11 bits

An address:



23 13 12 0

11 bits | 13 bits

page number

offset

Physical Addresses

32 bits --> 4 Gbyte installed memory (max)

512K Frames --> 19 bits

31 13 12 0

19 bits | 13 bits

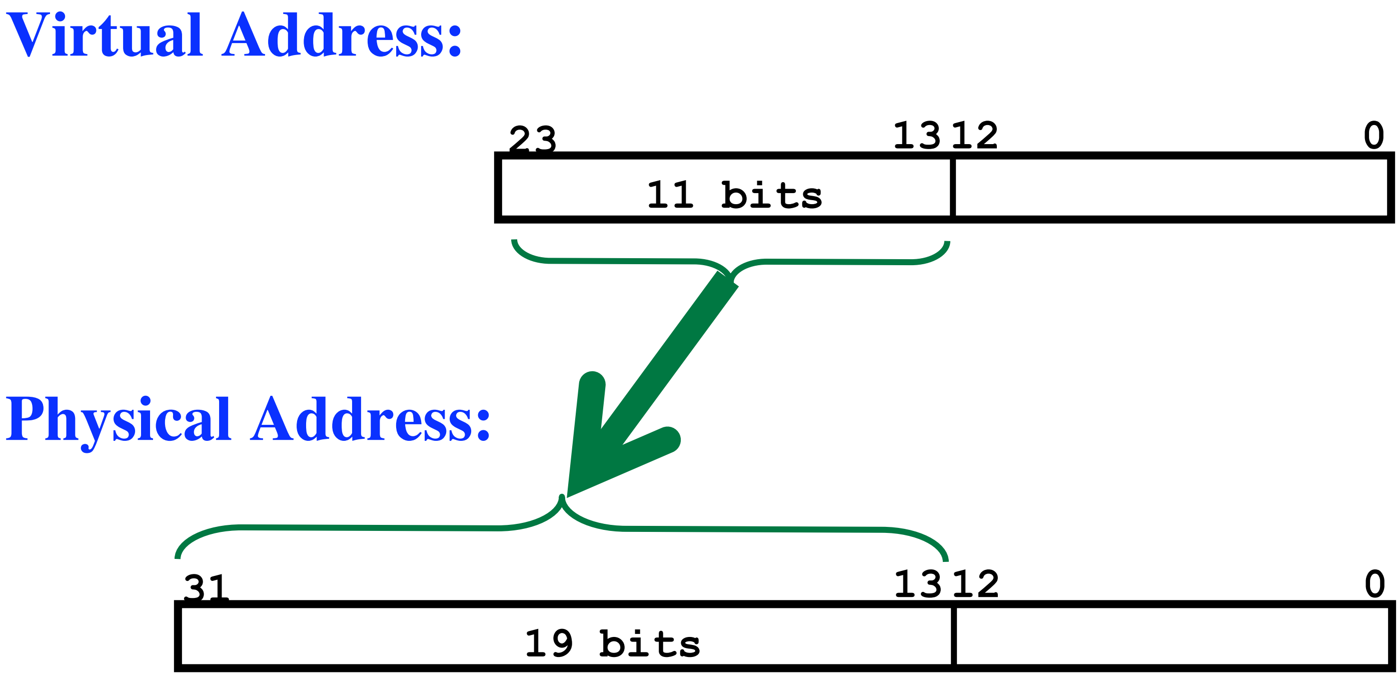
frame number

offset

The BLITZ Architecture

The Page Table Mapping:

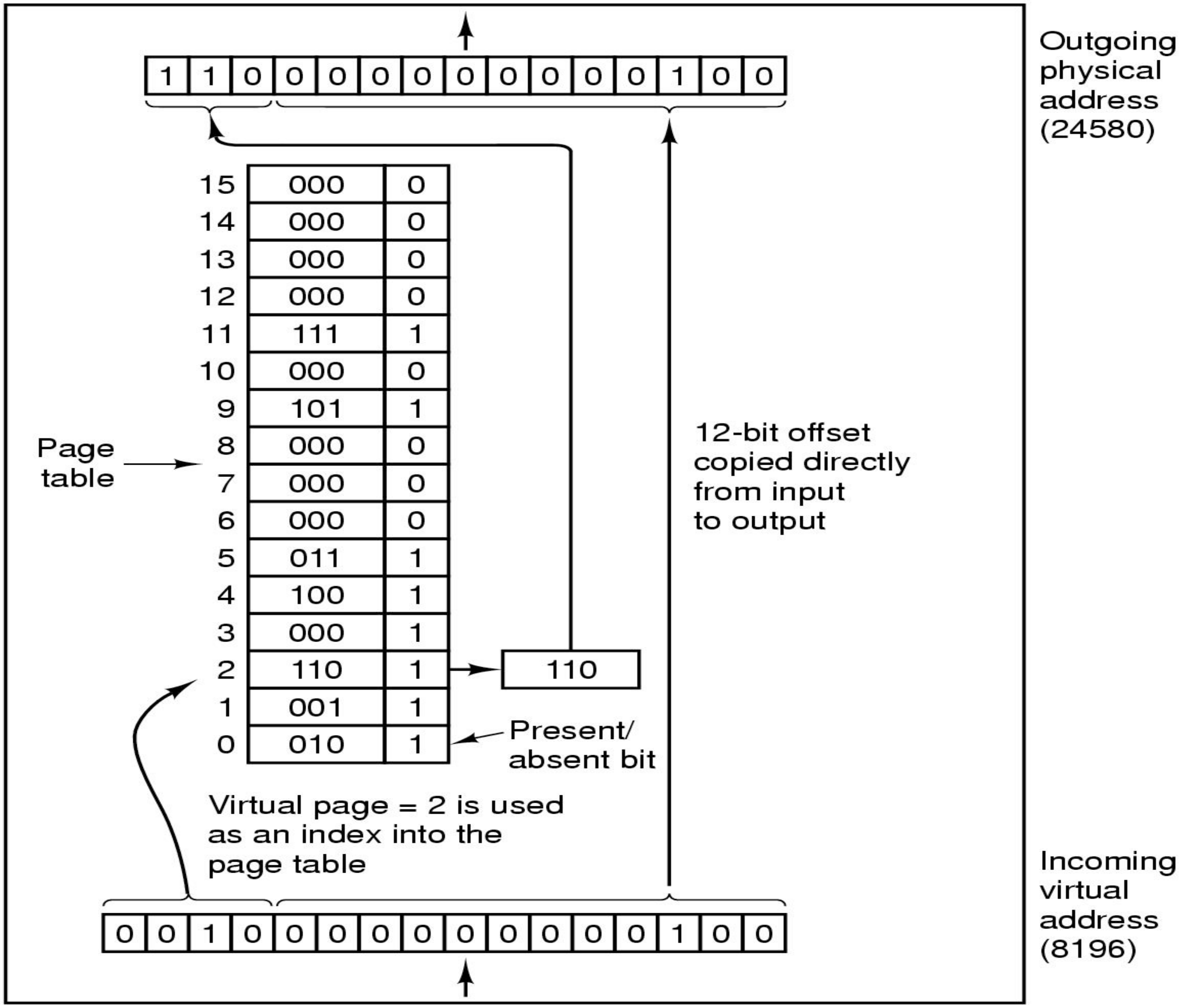
Page --> Frame



Virtual Address:

Physical Address:

Example from Textbook



15	000	0
14	000	0
13	000	0
12	000	0
11	111	1
10	000	0
9	101	1
8	000	0
7	000	0
6	000	0
5	011	1
4	100	1
3	000	1
2	110	1
1	001	1
0	010	1

The Page Table

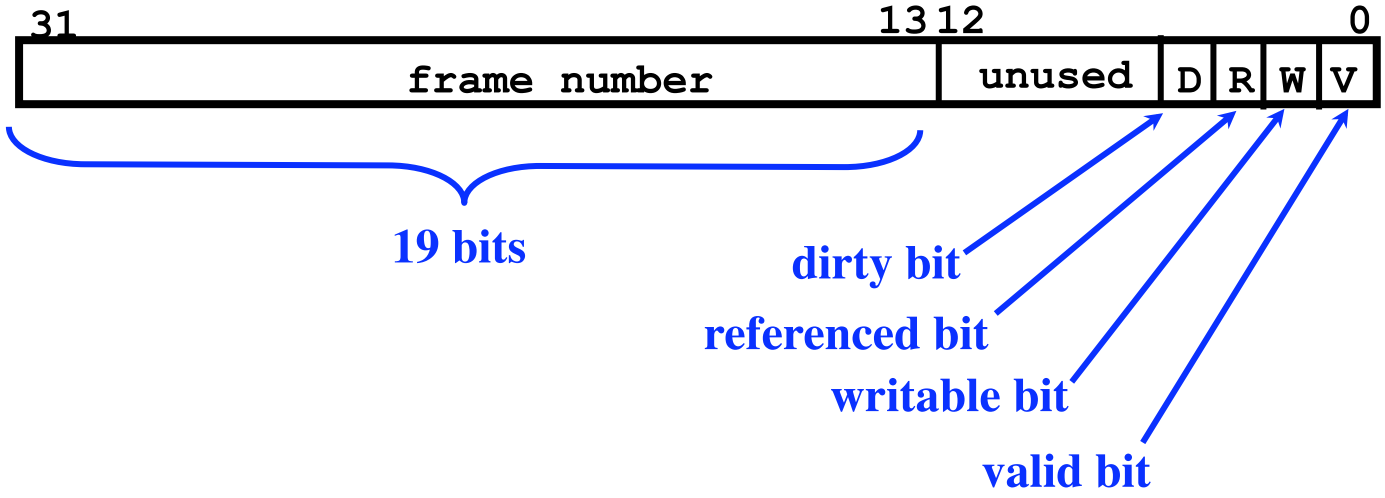
An array of “*page table entries*”
    Kept in memory

2K pages in a virtual address space?
    ---> 2K entries in the table

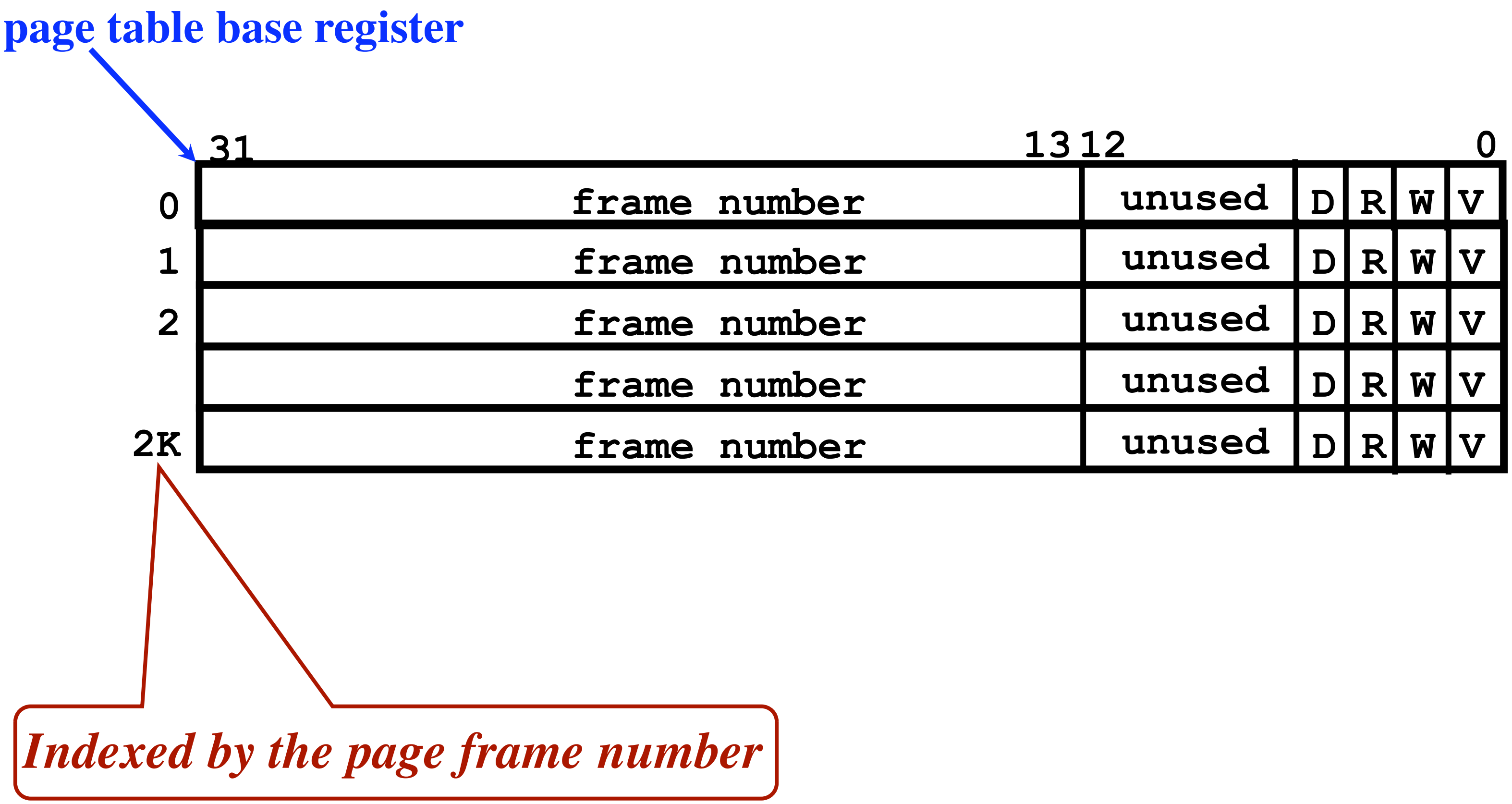
Each entry is 4 bytes long

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

The Page Table



The Page Table

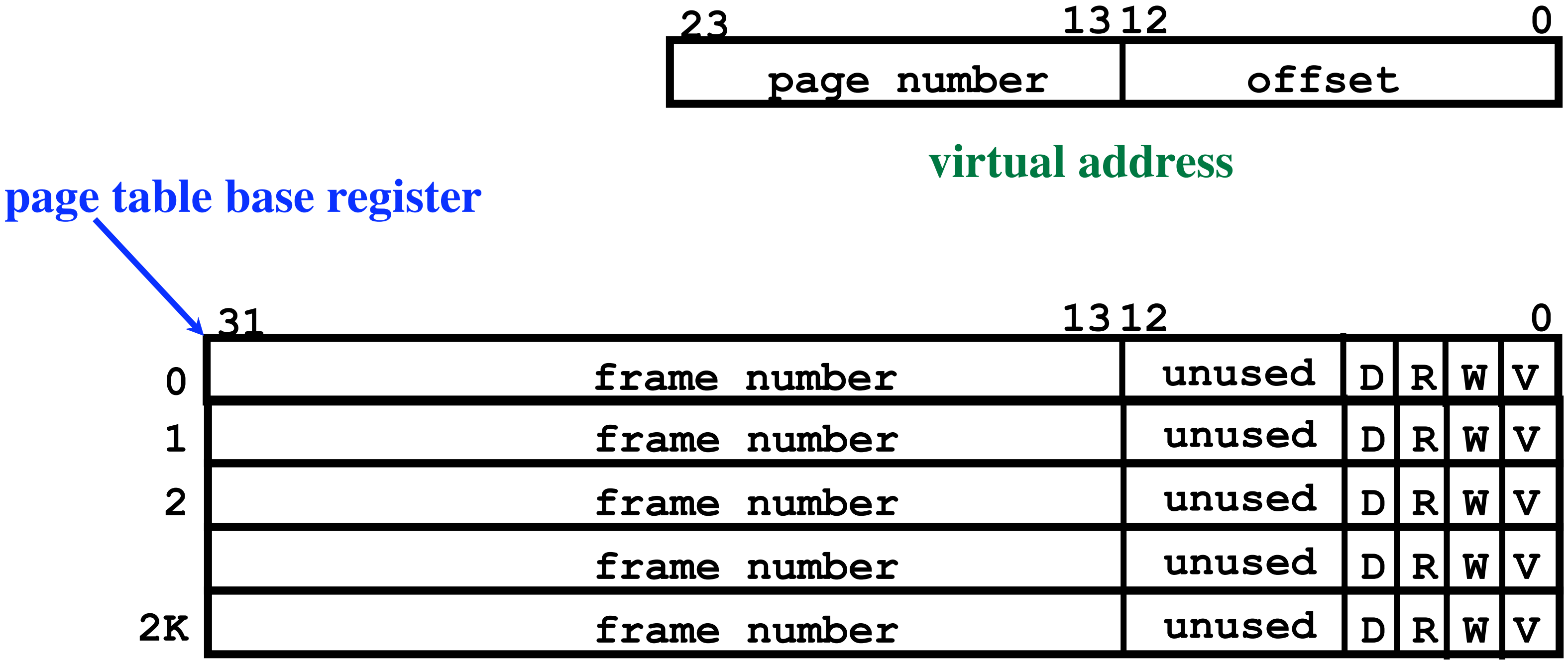


page table base register

	31 13	12				0
0	frame number	unused	D	R	W	V
1	frame number	unused	D	R	W	V
2	frame number	unused	D	R	W	V
	frame number	unused	D	R	W	V
2K	frame number	unused	D	R	W	V

Indexed by the page frame number

The Page Table



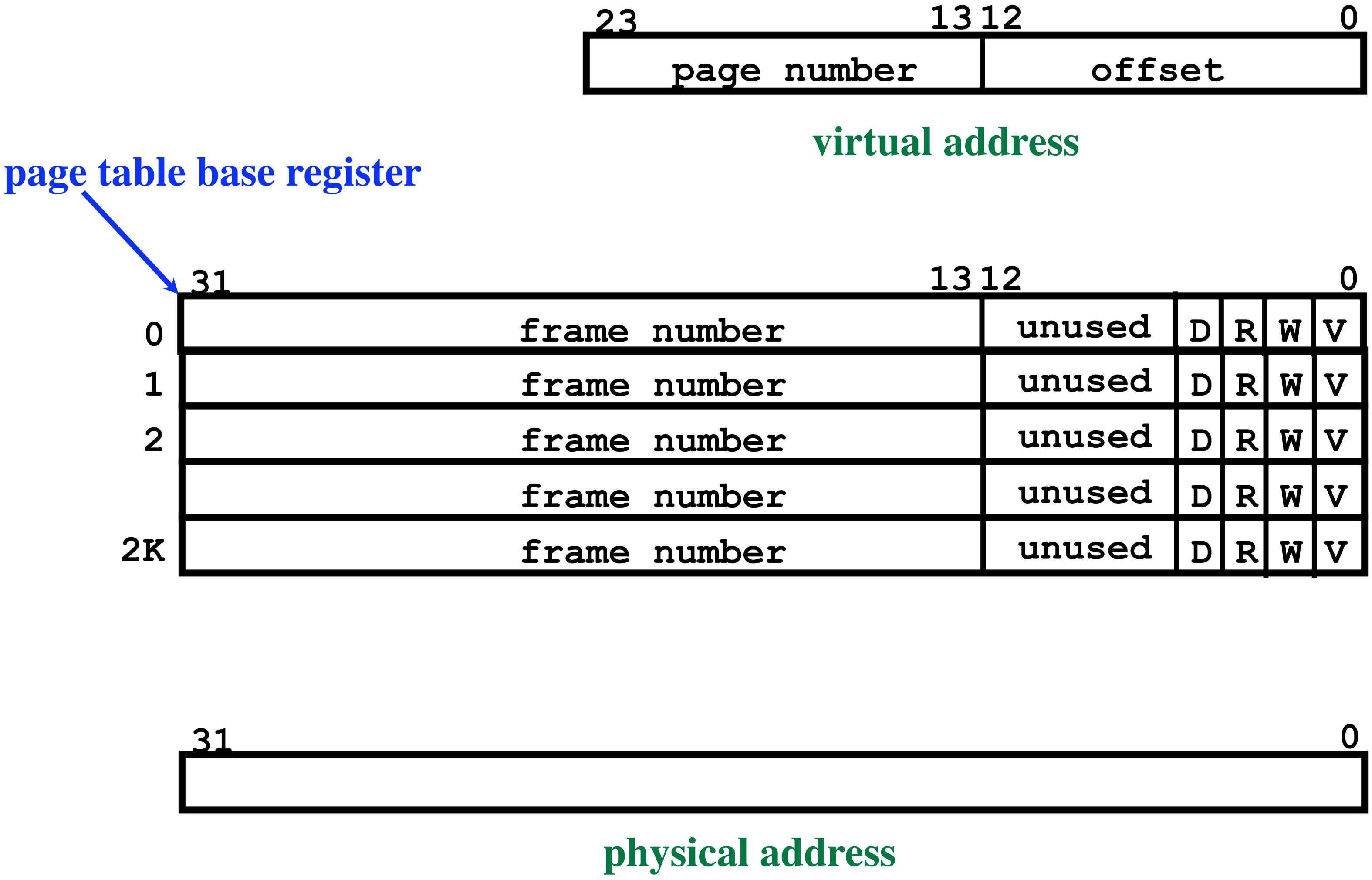
23 … 13	12 … 0
page number	offset

virtual address

page table base register

	31 … 13	12 …				0
0	frame number	unused	D	R	W	V
1	frame number	unused	D	R	W	V
2	frame number	unused	D	R	W	V
	frame number	unused	D	R	W	V
2K	frame number	unused	D	R	W	V

The Page Table



23 13 12 0

page number | offset

virtual address

page table base register

31 13 12 0

0 frame number unused D R W V

1 frame number unused D R W V

2 frame number unused D R W V

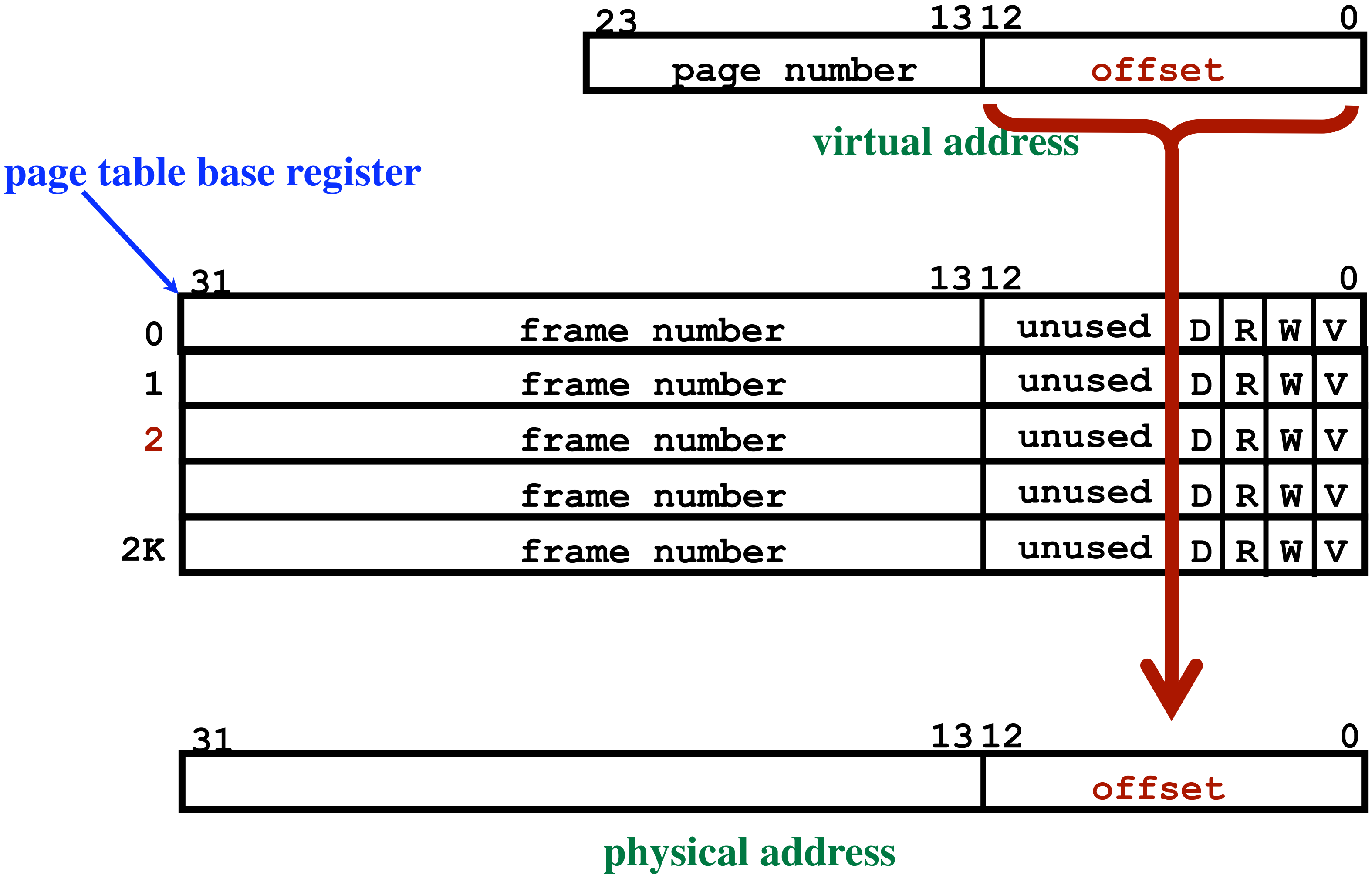
frame number unused D R W V

2K frame number unused D R W V

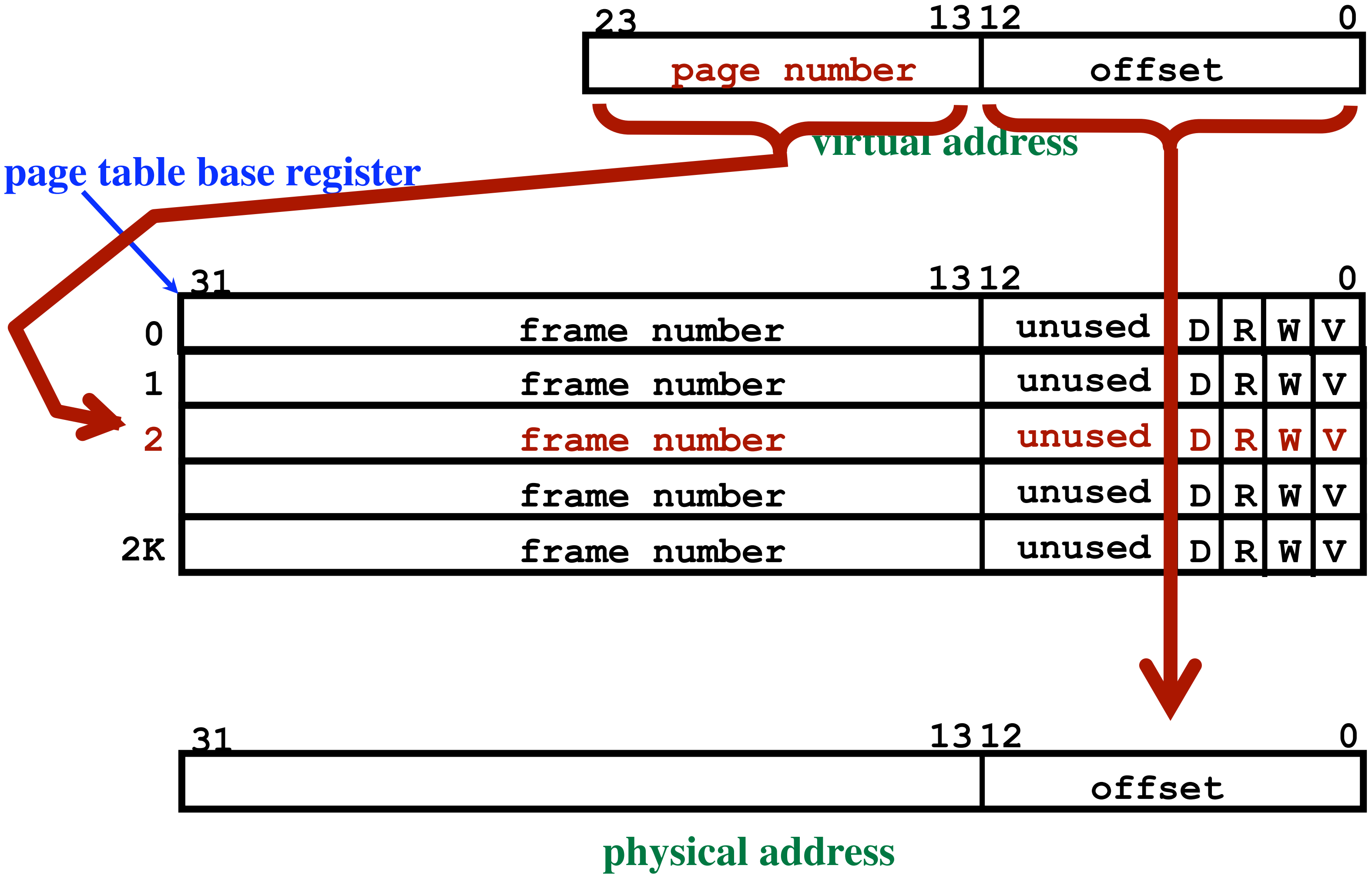
31 0

physical address

The Page Table



The Page Table



23 13 12 0

page number | offset

virtual address

page table base register

31 13 12 0

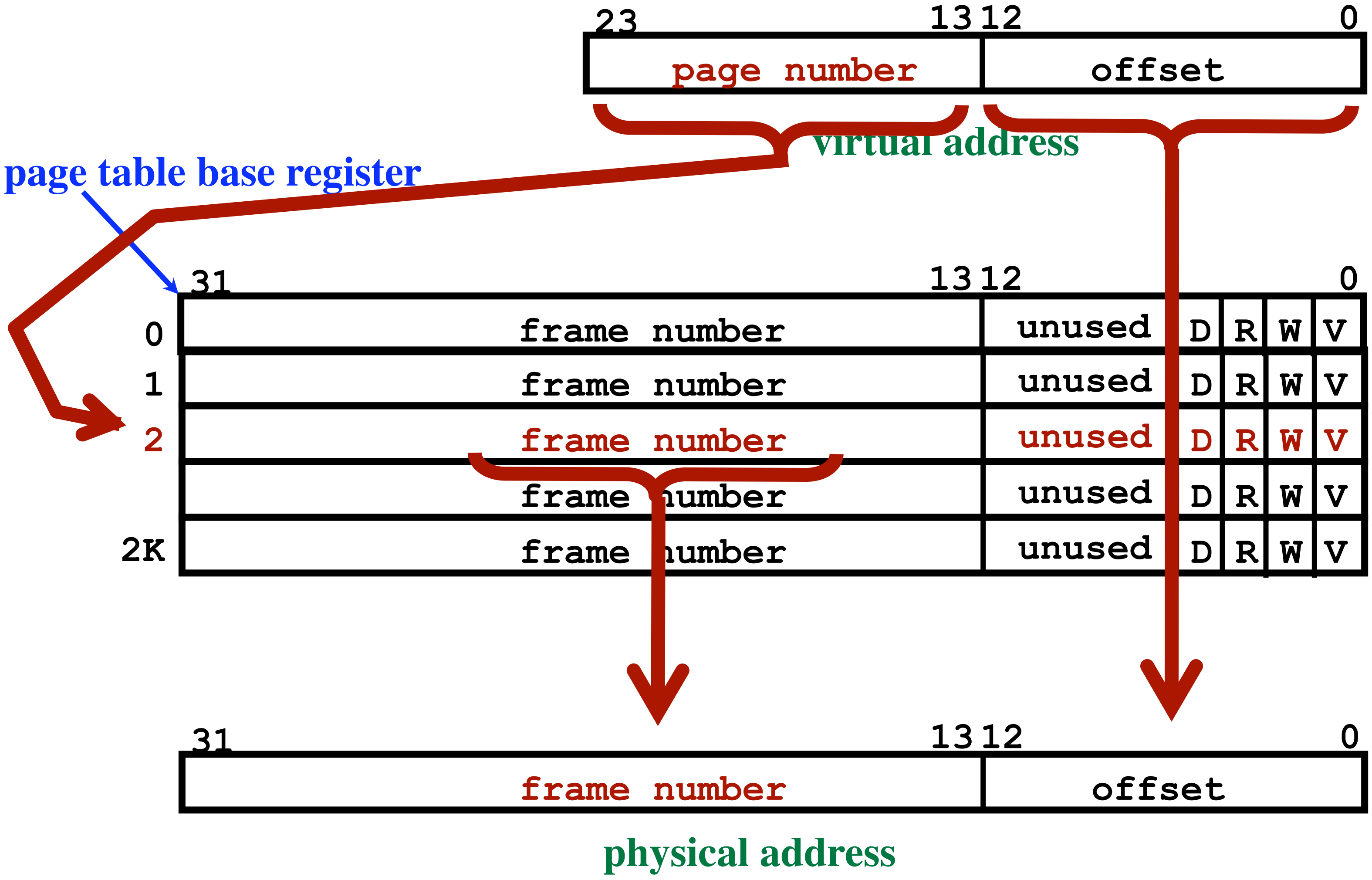
0	frame number	unused	D	R	W	V
1	frame number	unused	D	R	W	V
2	frame number	unused	D	R	W	V
	frame number	unused	D	R	W	V
2K	frame number	unused	D	R	W	V

31 13 12 0

offset

physical address

The Page Table



23 13 12 0

page number | offset

virtual address

page table base register

31 13 12 0

0	frame number	unused	D	R	W	V
1	frame number	unused	D	R	W	V
2	frame number	unused	D	R	W	V
	frame number	unused	D	R	W	V
2K	frame number	unused	D	R	W	V

31 13 12 0

frame number | offset

physical address

The Page Table

Two registers in the CPU

- **Page Table Base Register**
- **Page Table Length Register**

These define the current page table.

(Virtual address space is smaller? Use a smaller table!)

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

Internal Fragmentation

A process will fill several pages.
The last page will be partially full.
    On average, last page will be half full.
This space is wasted, lost!!!

Example: 8K page size
    ---> 4K is wasted for every running process

External Fragmentation

Memory is divided into chunks ("partitions")
Each partition has a different size.
Processes are allocated space and later freed.
After a while...
    Memory will be full of small holes.

External Fragmentation

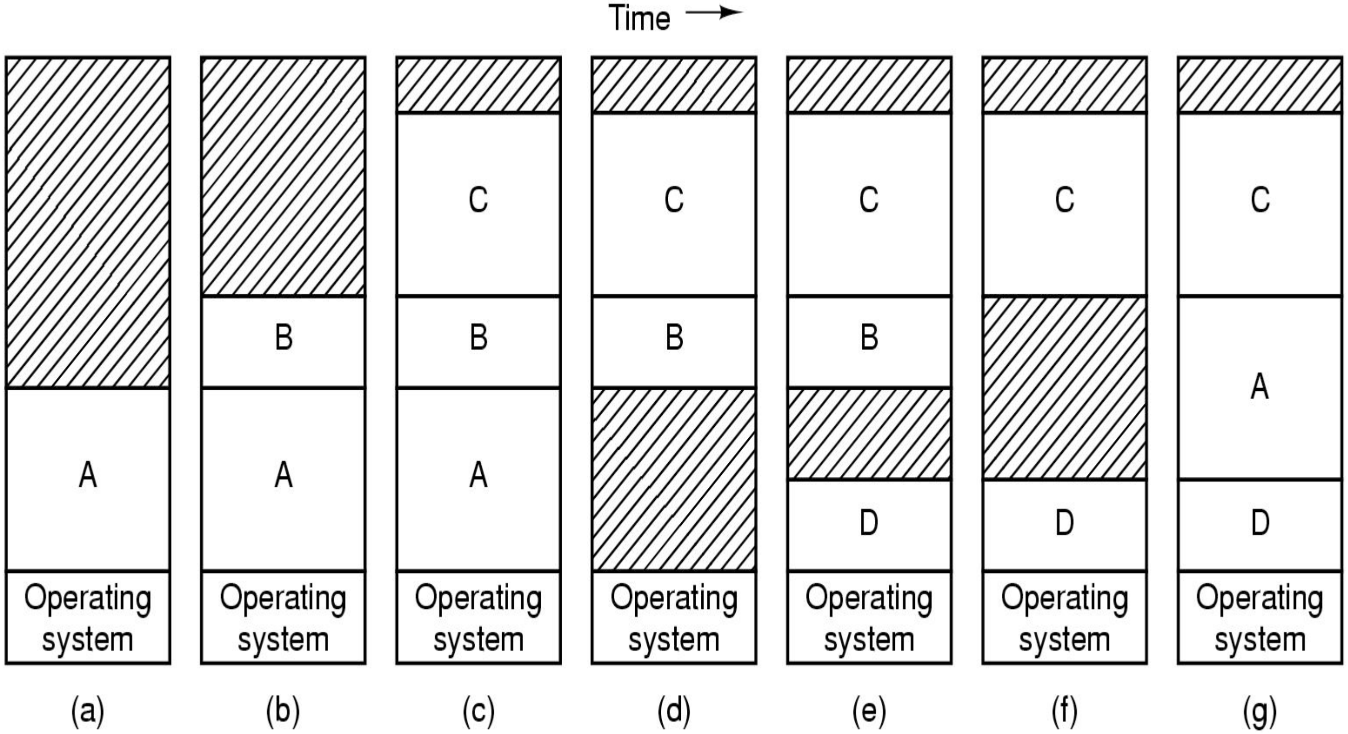
Memory is divided into chunks (“partitions”)

Each partition has a different size.

Processes are allocated space and later freed.

After a while...

Memory will be full of small holes.



Page Size Issues

Choose a large page size

More loss due to internal fragmentation

Assume a process is using 5 regions of memory heavily

... Will need 5 pages, regardless of page size

---> Ties up more memory

Page Size Issues

Choose a large page size

More loss due to internal fragmentation

Assume a process is using 5 regions of memory heavily

... Will need 5 pages, regardless of page size

---> Ties up more memory

Choose a small page size

The page table will become very large

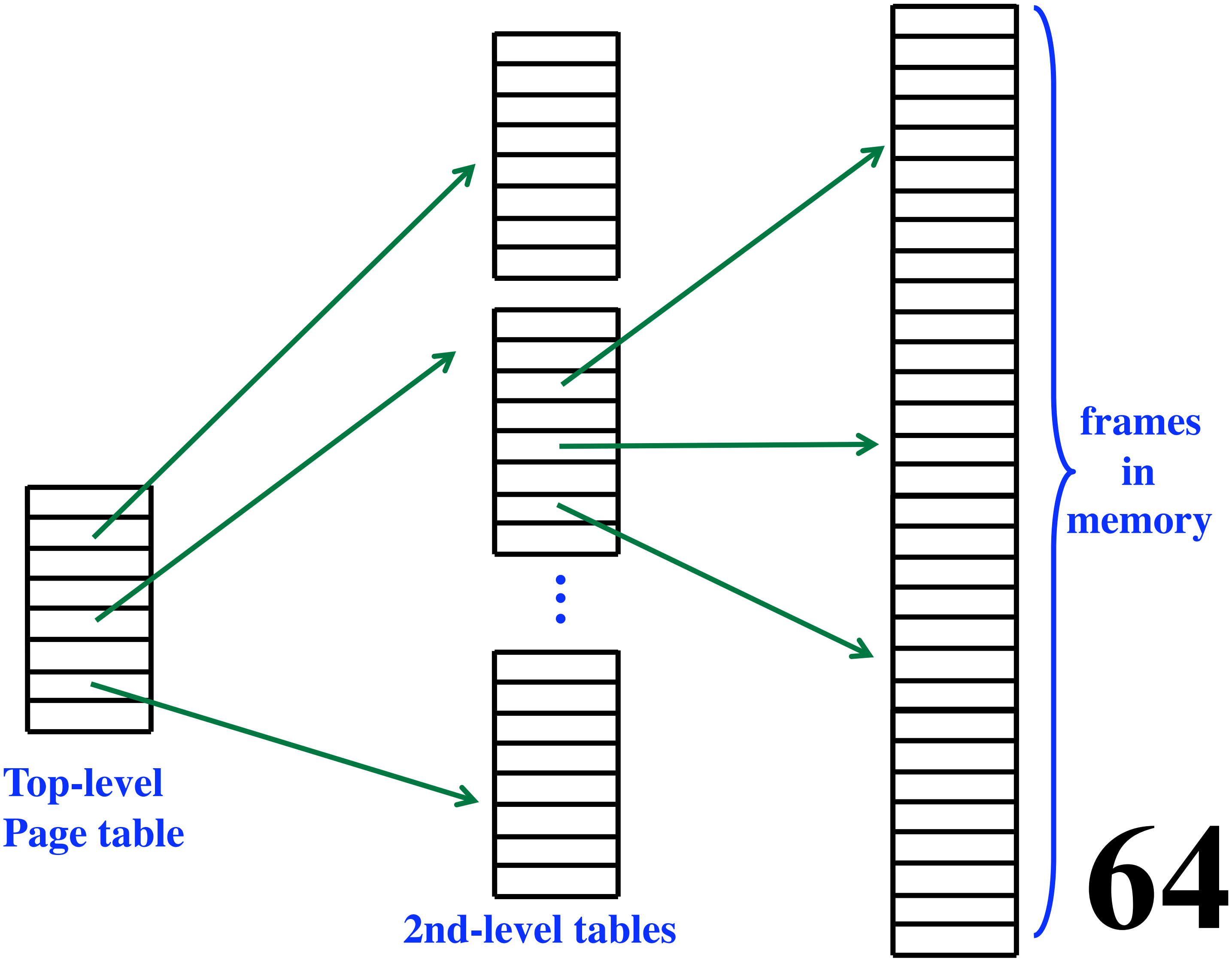
Example:

Virtual Address Space: 4G bytes

Page Size: 4K (e.g., Pentium)

Page table size: 1M entries! (4Mbytes)

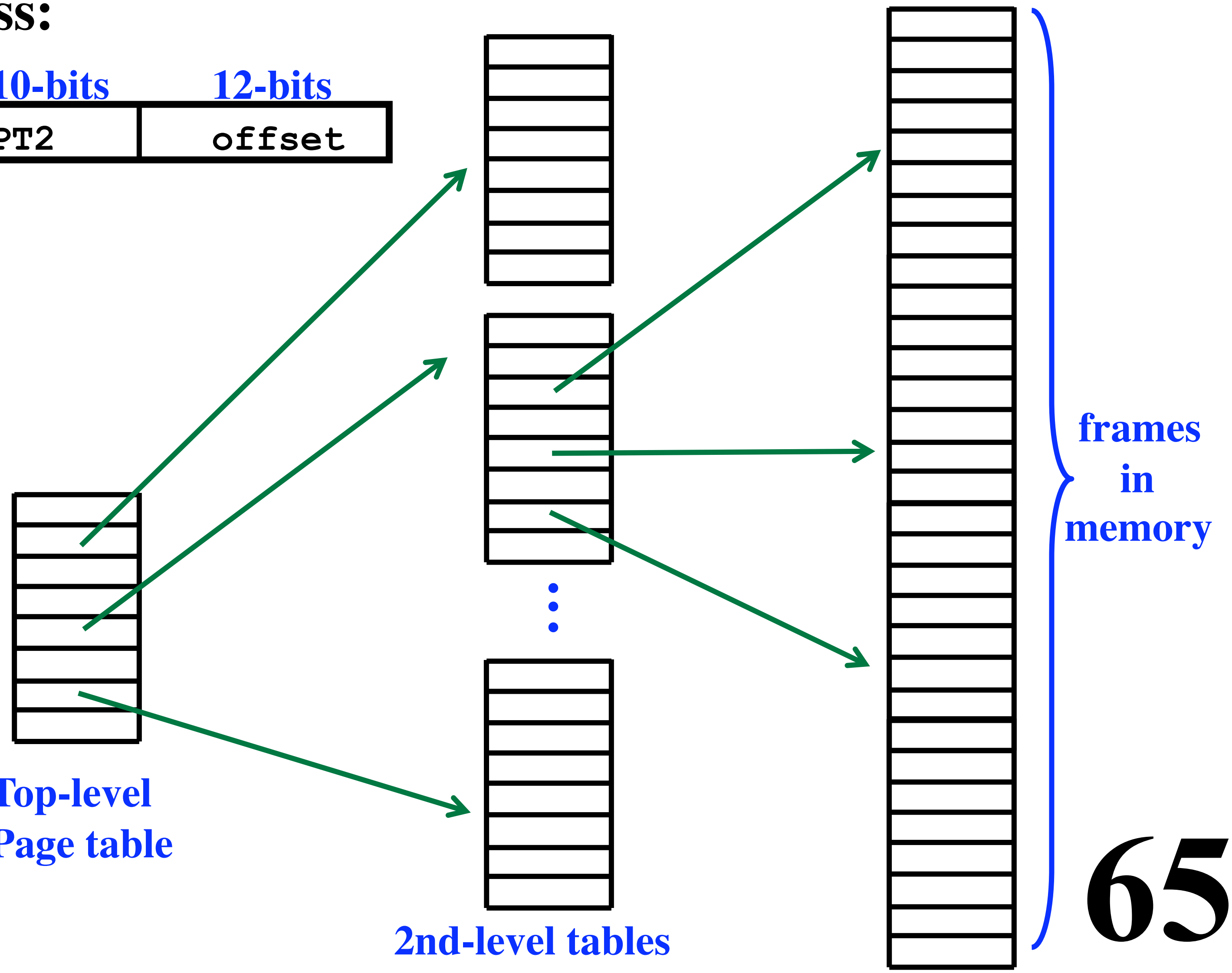
Multi-level Page Tables



Multi-level Page Tables

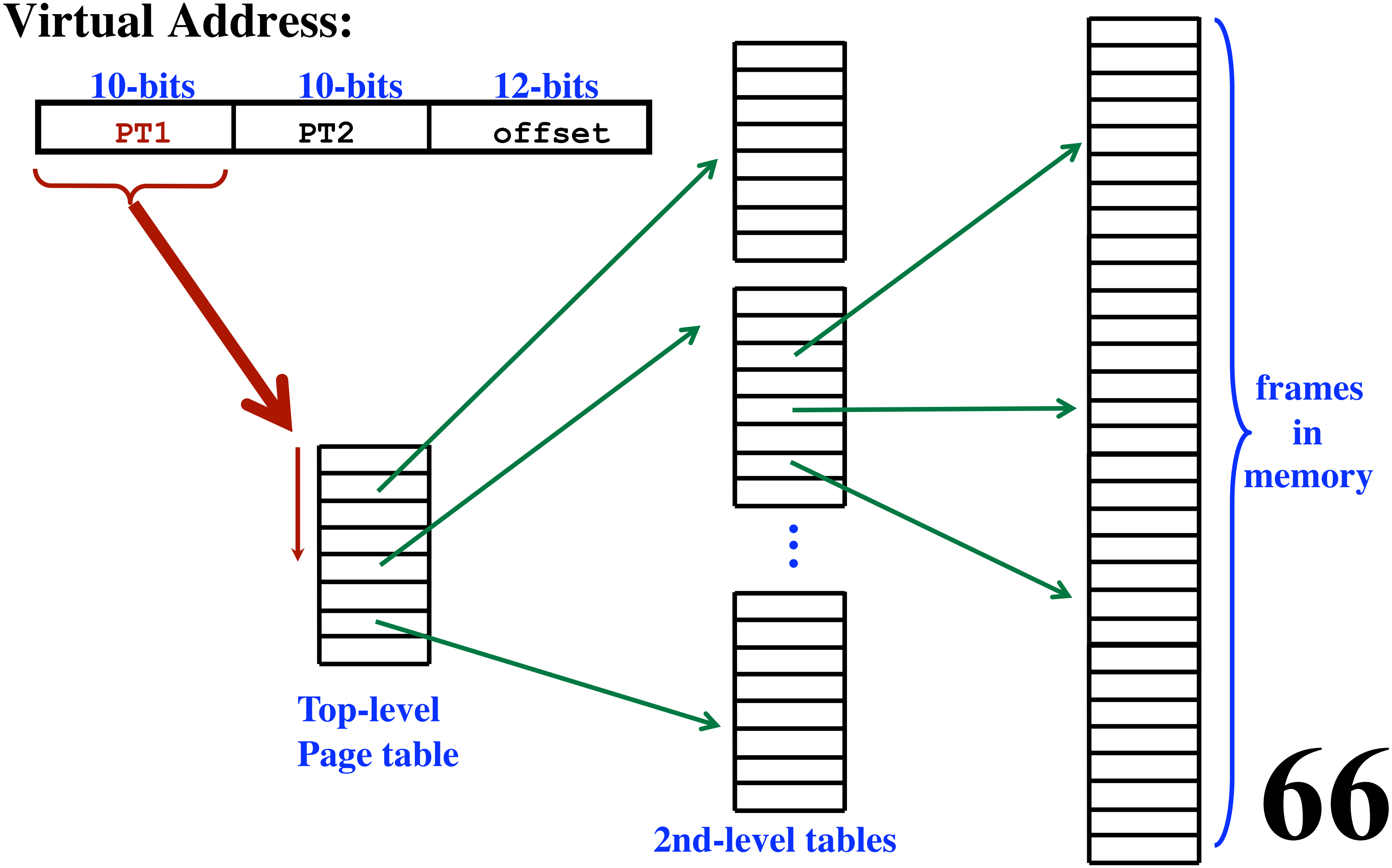
A Virtual Address:

10-bits	10-bits	12-bits
PT1	PT2	offset



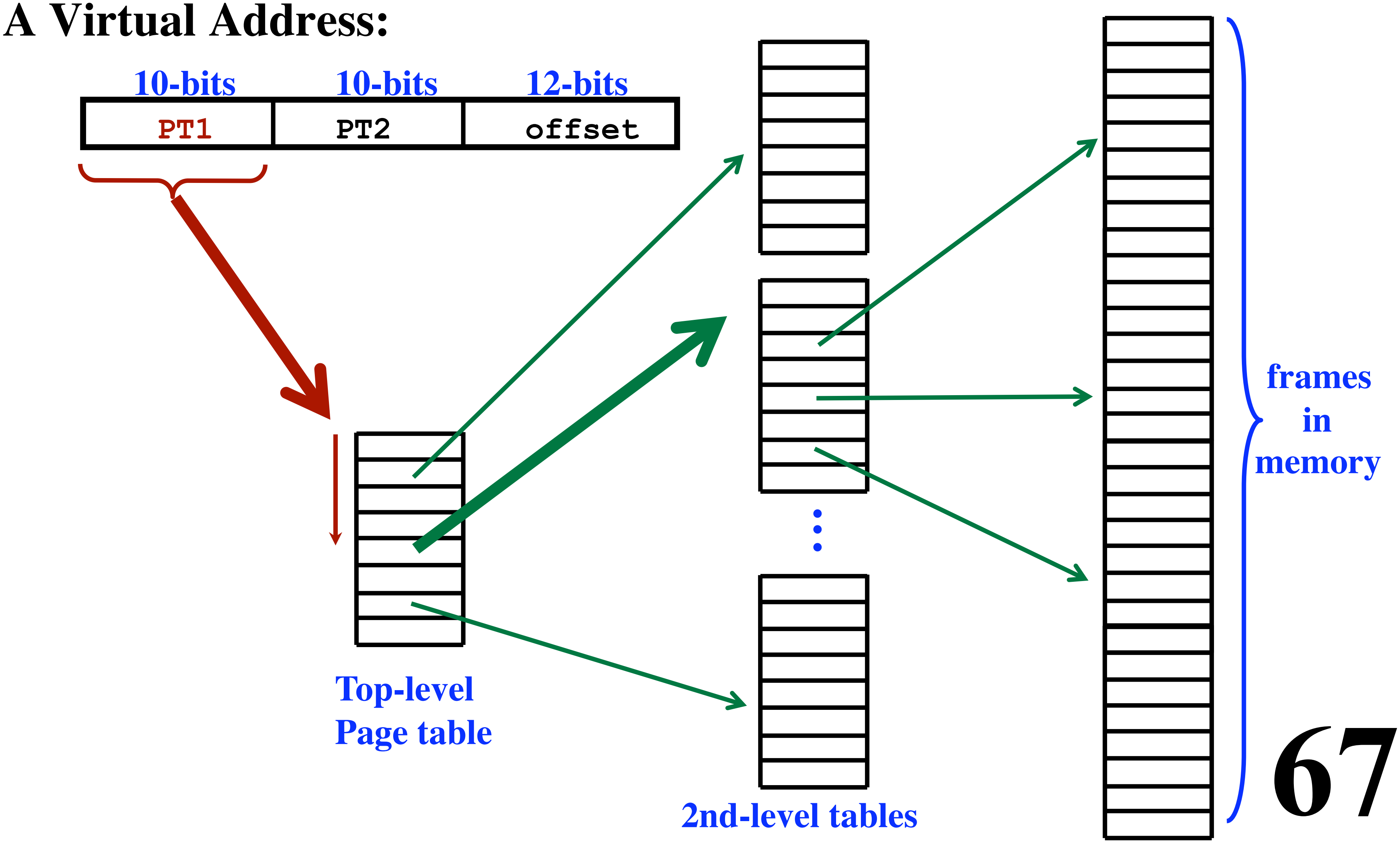
Multi-level Page Tables

A Virtual Address:



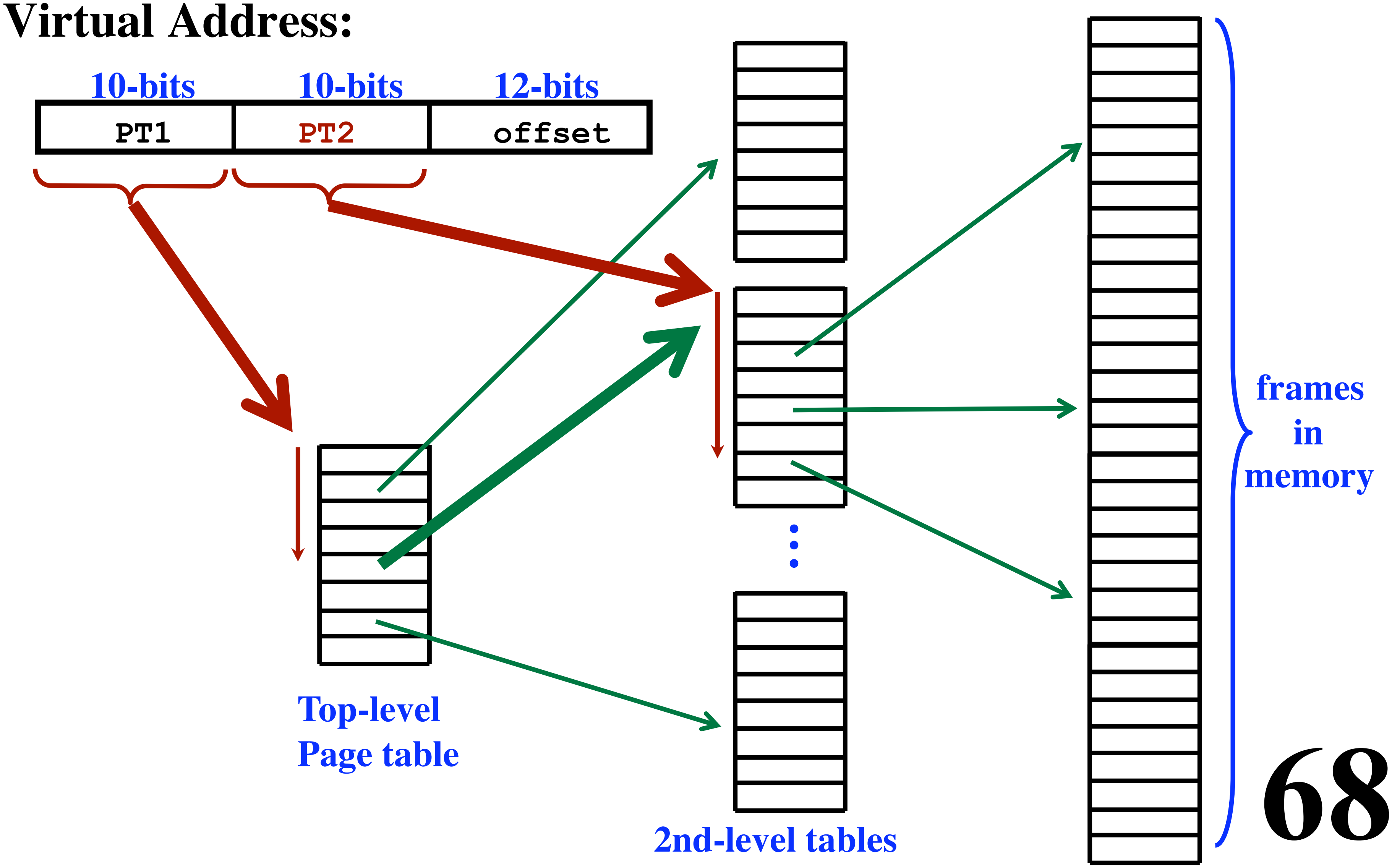
Multi-level Page Tables

A Virtual Address:

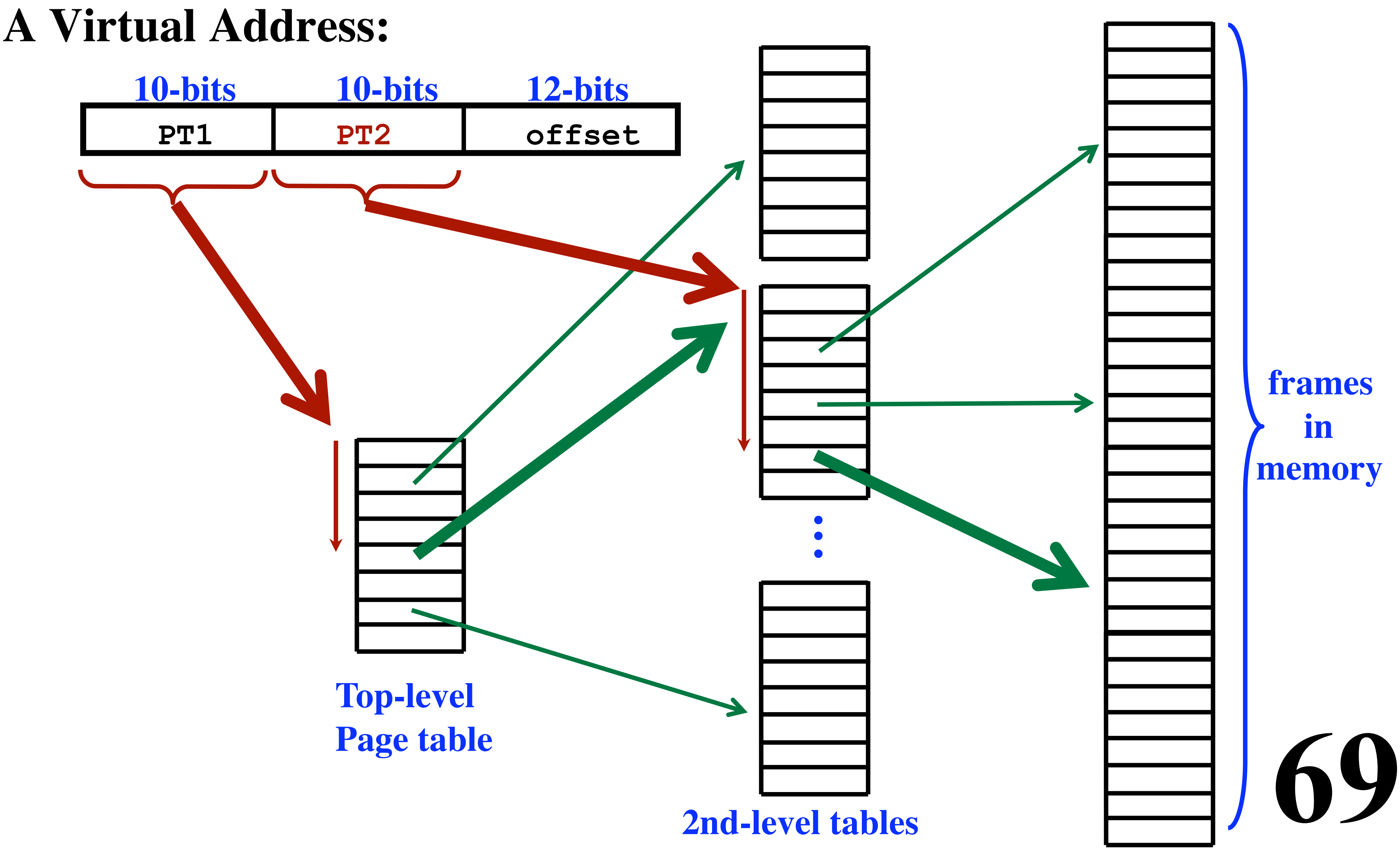


Multi-level Page Tables

A Virtual Address:

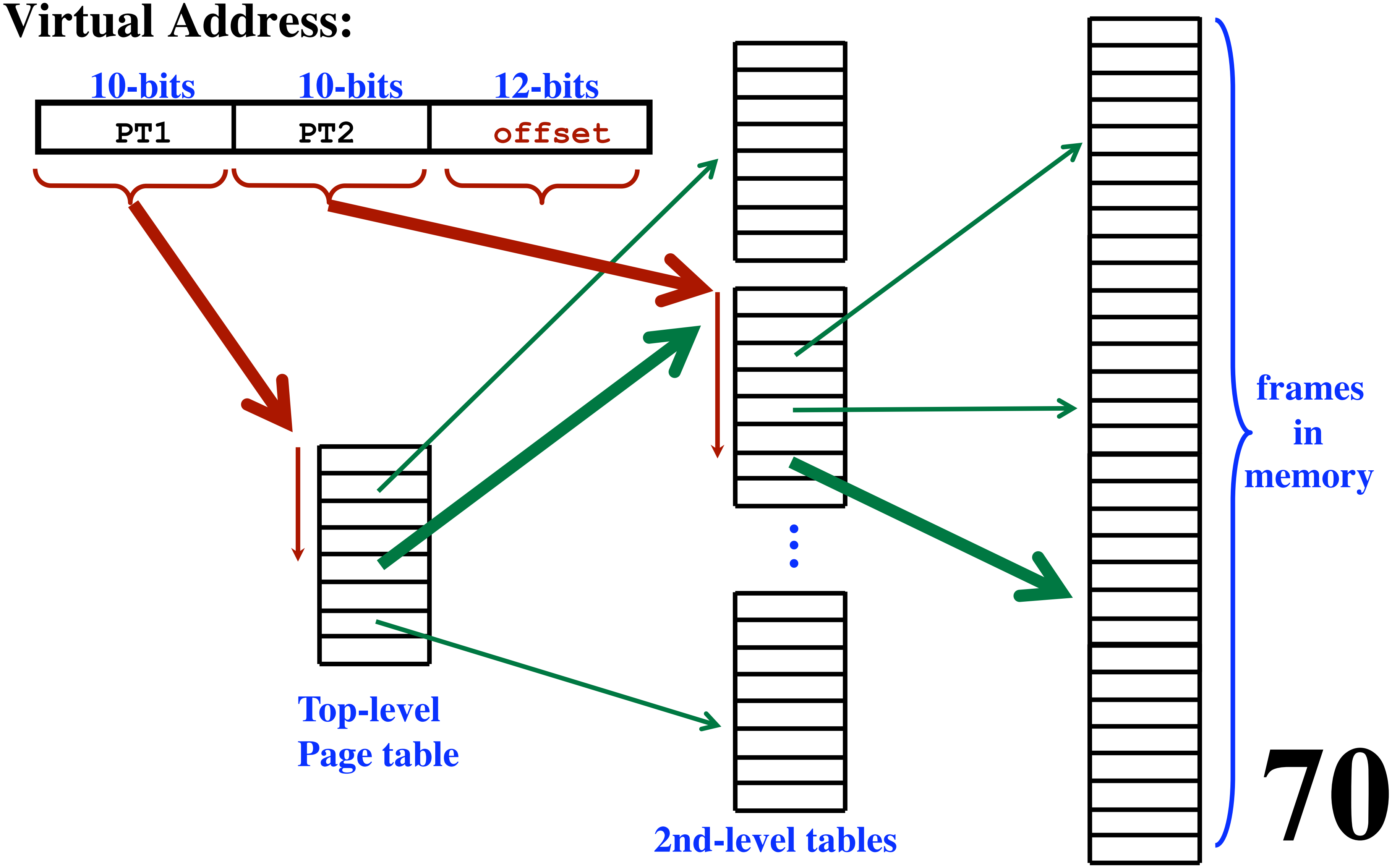


Multi-level Page Tables

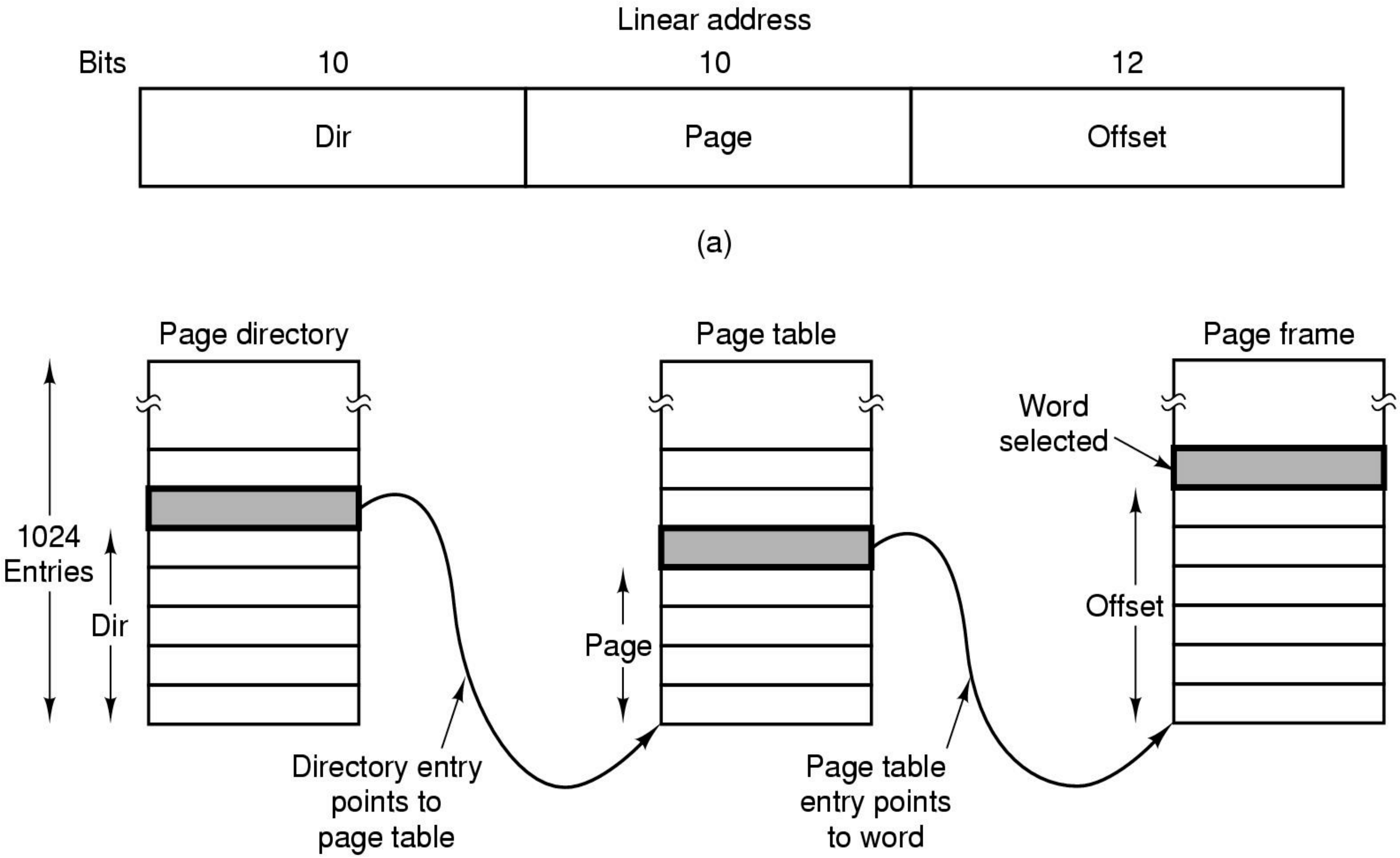


Multi-level Page Tables

A Virtual Address:



Multi-level Page Tables



(a)

(b)

Translation Lookaside Buffers (TLBs)

Problem:

MMU must go to page table for every memory access!

Translation Lookaside Buffers (TLBs)

Problem:

MMU must go to page table for every memory access!

Solution:

Cache the page table entries

Hardware cache in the MMU

Small number of entries (e.g., 64)

Each entry contains

- Page Number
- Other stuff from page table entry

Associatively indexed

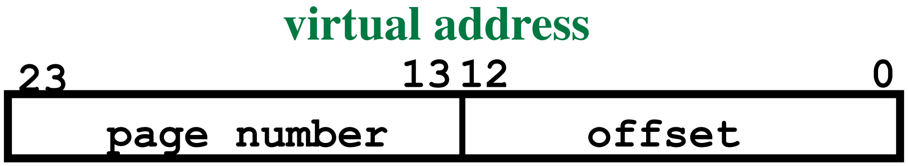
- On page number

Translation Lookaside Buffers (TLBs)

Key

Page Number	Frame Number	Other				
23	37	unused	D	R	W	V
17	50	unused	D	R	W	V
92	24	unused	D	R	W	V
5	19	unused	D	R	W	V
12	6	unused	D	R	W	V

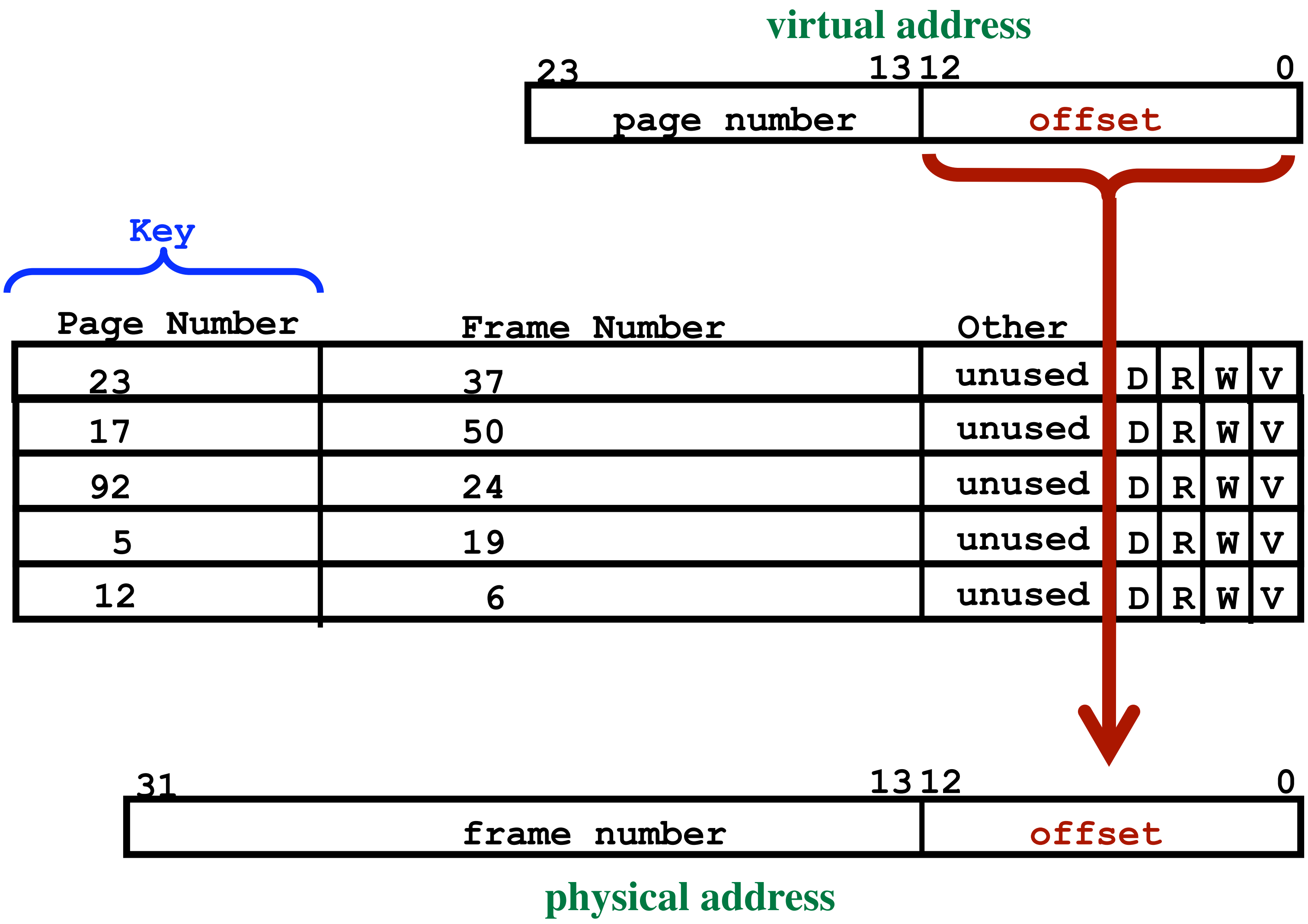
Translation Lookaside Buffers (TLBs)



Key

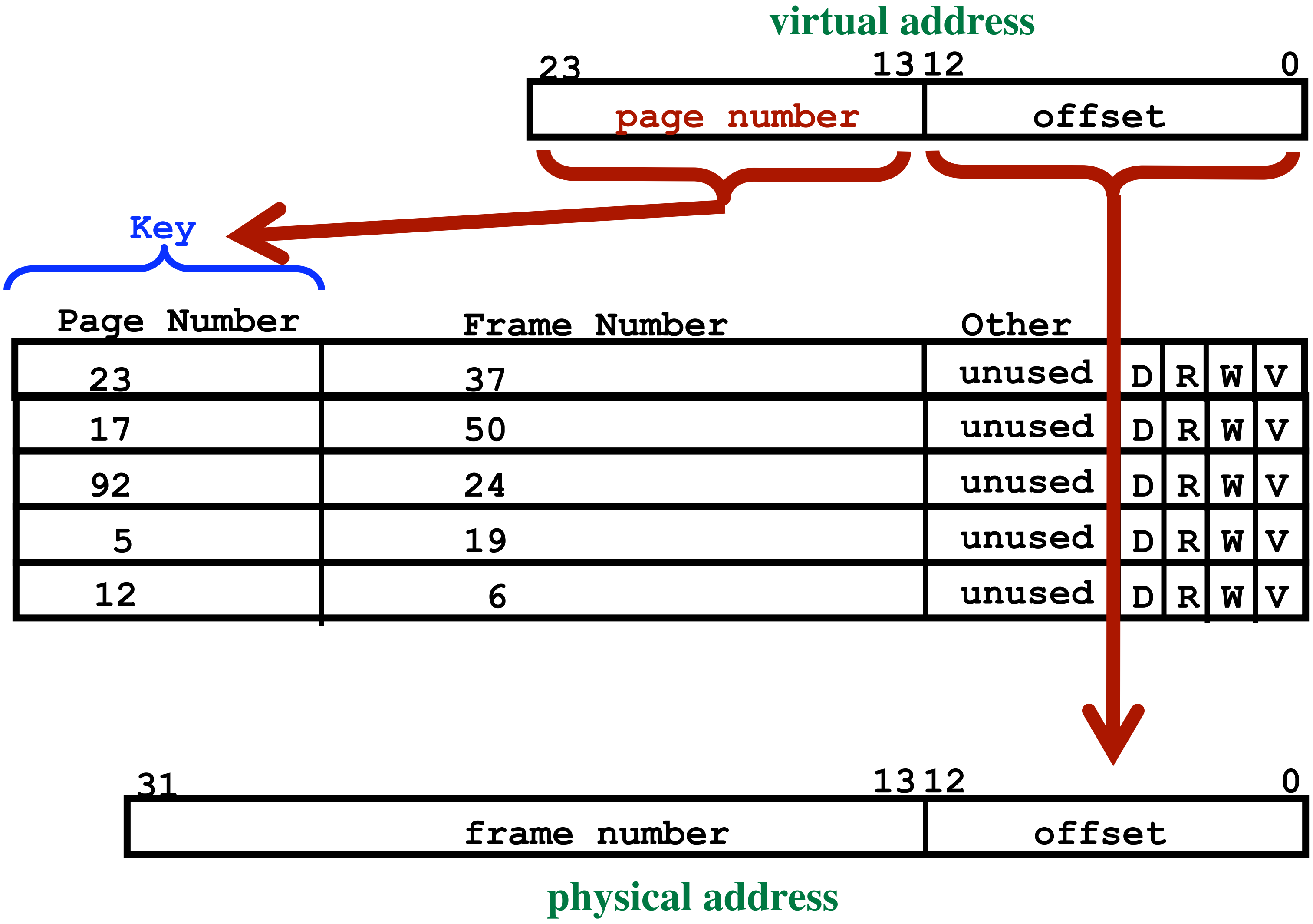
Page Number	Frame Number	Other				
23	37	unused	D	R	W	V
17	50	unused	D	R	W	V
92	24	unused	D	R	W	V
5	19	unused	D	R	W	V
12	6	unused	D	R	W	V

Translation Lookaside Buffers (TLBs)



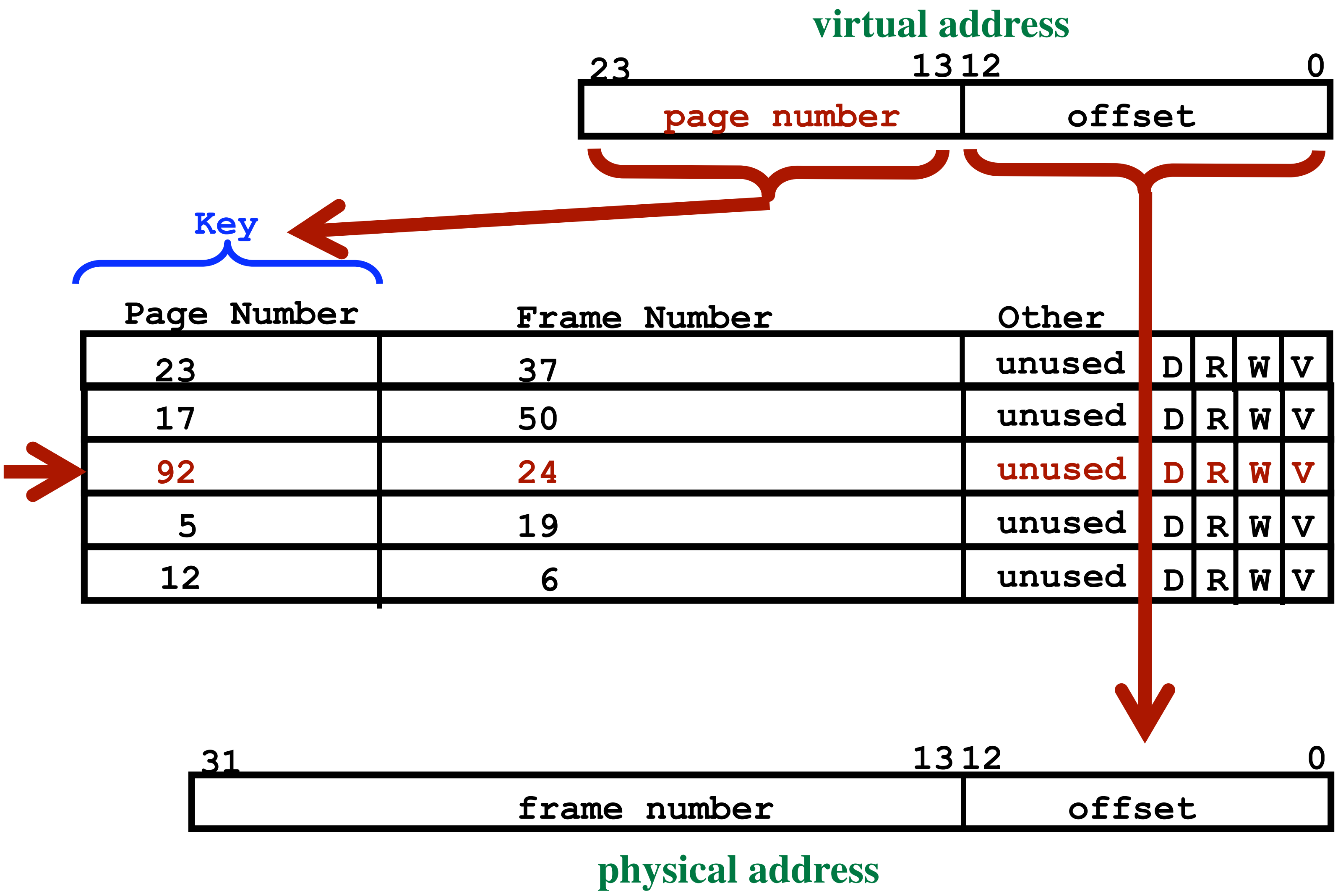
Page Number	Frame Number	Other				
23	37	unused	D	R	W	V
17	50	unused	D	R	W	V
92	24	unused	D	R	W	V
5	19	unused	D	R	W	V
12	6	unused	D	R	W	V

Translation Lookaside Buffers (TLBs)



Page Number	Frame Number	Other				
23	37	unused	D	R	W	V
17	50	unused	D	R	W	V
92	24	unused	D	R	W	V
5	19	unused	D	R	W	V
12	6	unused	D	R	W	V

Translation Lookaside Buffers (TLBs)

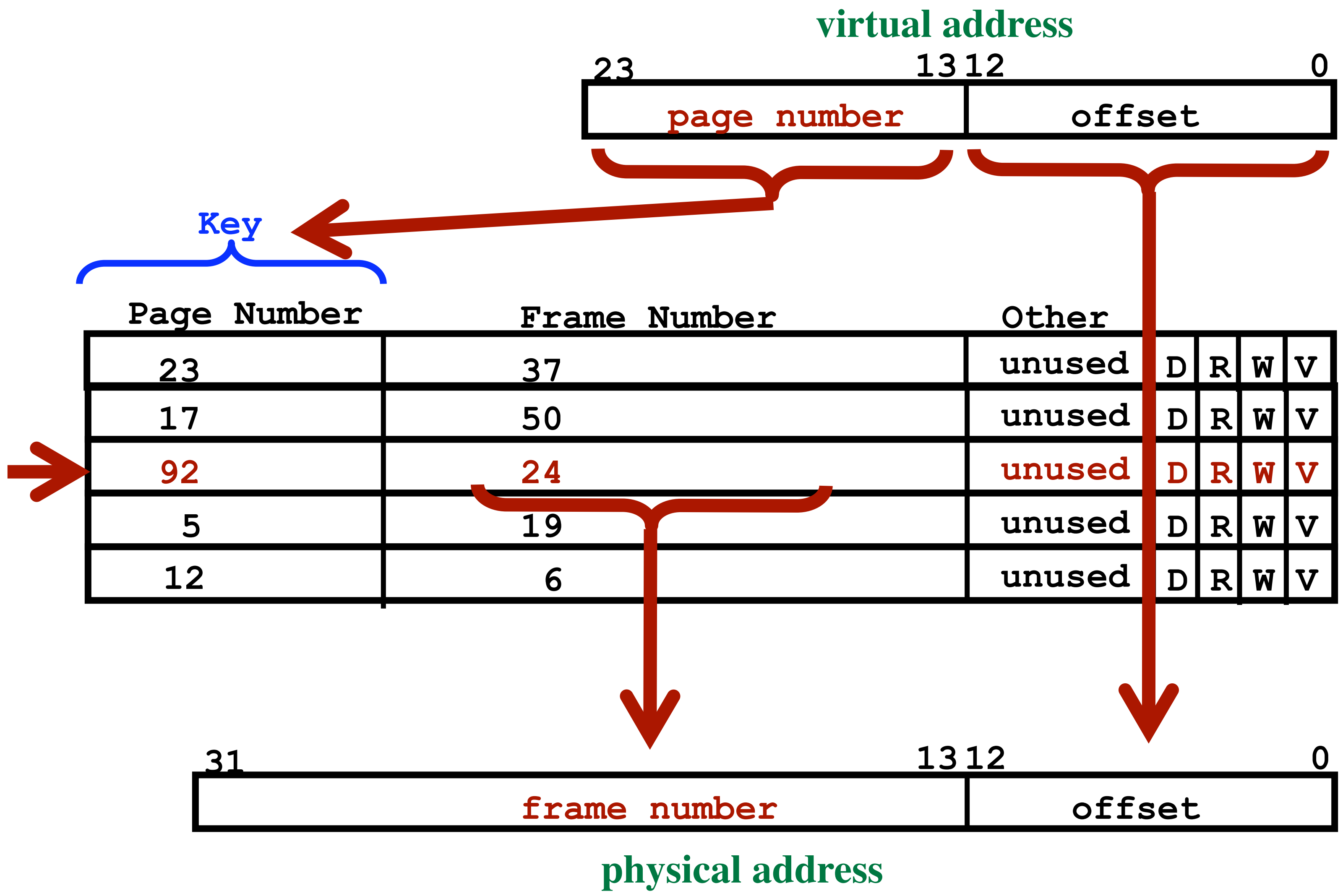


virtual address

Page Number	Frame Number	Other				
23	37	unused	D	R	W	V
17	50	unused	D	R	W	V
92	24	unused	D	R	W	V
5	19	unused	D	R	W	V
12	6	unused	D	R	W	V

physical address

Translation Lookaside Buffers (TLBs)



Page Number	Frame Number	Other				
23	37	unused	D	R	W	V
17	50	unused	D	R	W	V
92	24	unused	D	R	W	V
5	19	unused	D	R	W	V
12	6	unused	D	R	W	V

Translation Lookaside Buffers (TLBs)

What if the entry is not in the TLB?

- Go to page table
- Find the right entry
- Move it into the TLB
- Which entry to replace?
  - Software trap -- Let OS deal with the problem
- Valid Bit
- Page tables become entirely a OS data structure!

Want to do a context switch?

- Must empty the TLB
  - Just clear the “Valid Bit”

64-Bit Virtual Addresses

Assume 4 Kbyte pages (12 bits)

Virtual Space = 2^{52} pages (page table too large!)

Assume 256 Mbyte memory

- Can only have 64K pages in memory

Only need entries for the pages in memory

64-Bit Virtual Addresses

Assume 4 Kbyte pages (12 bits)
Virtual Space = 2^{52} pages (page table too large!)
Assume 256 Mbyte memory
- Can only have 64K pages in memory

Only need entries for the pages in memory

“Inverted Page Table”
- One entry for every *frame* in memory
  - Tells which page is in that frame
- When running the program
  - Given a virtual page, need the frame
  - Search all pages? No
  - Use an indexed, e.g., Hash table

Inverted Page Table