

# Memory Protection and Address Translation

Xin Liu

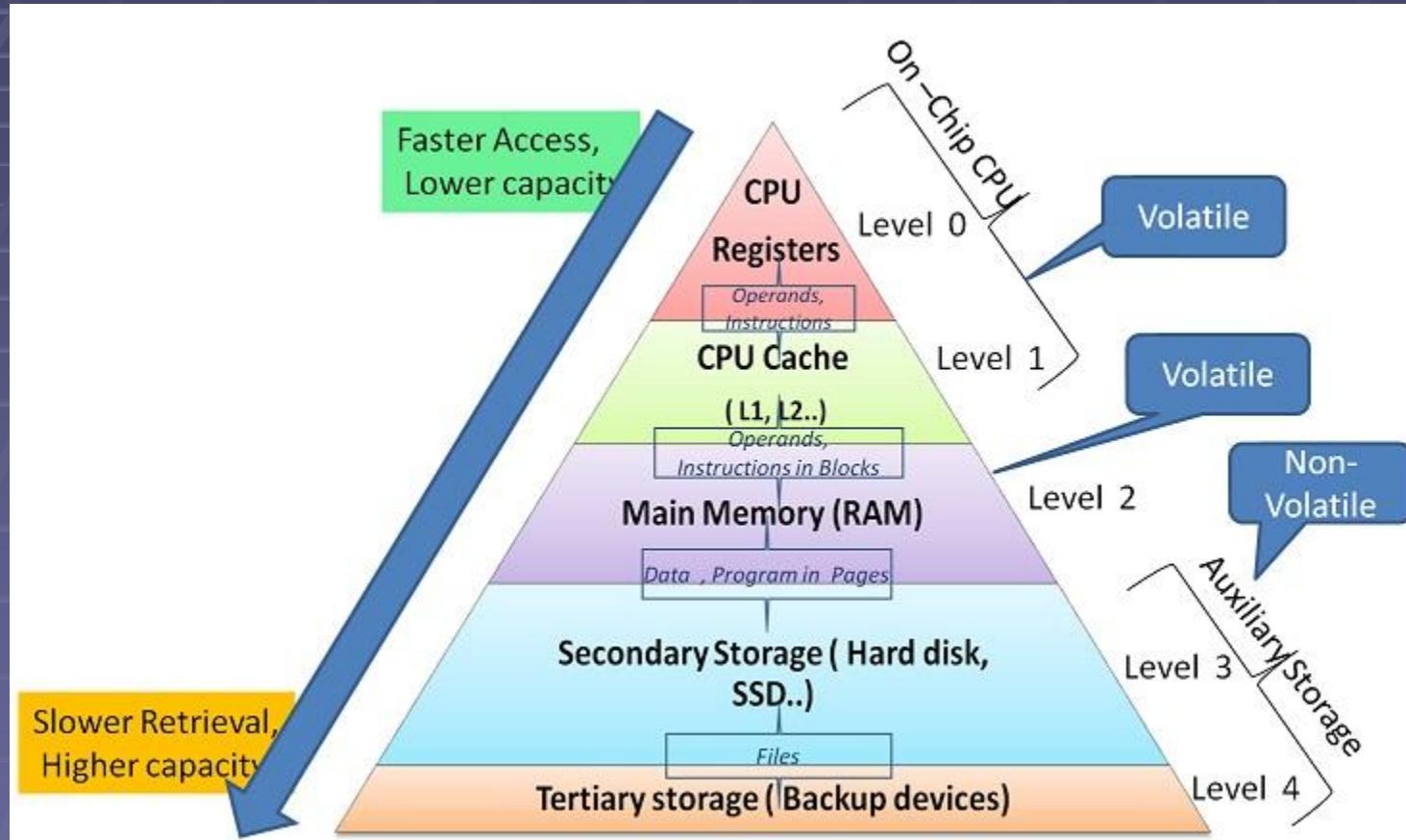
Operating Systems

COP 4610

# This Lecture...

- Different address translation schemes
  - Base-and-bound translation
  - Segmentation
  - Paging
  - Multi-level translation
  - Paged page tables
  - Hashed page tables
  - Inverted page tables
- Dual-mode Operations

# Memory Architecture



# Memory Addresses

- Memory is byte-addressable
  - Each memory address can specify the location of a byte
- `unsigned char memory[MEMORY_SIZE]`
- To access
  - `memory[virtual_memory_address]`

# Up to This Point

- Threads provide the illusion of an infinite number of CPUs
  - On a single processor machine
- Memory management provides a different set of illusions
  - Protected memory
  - Infinite amount of memory
  - Transparent sharing

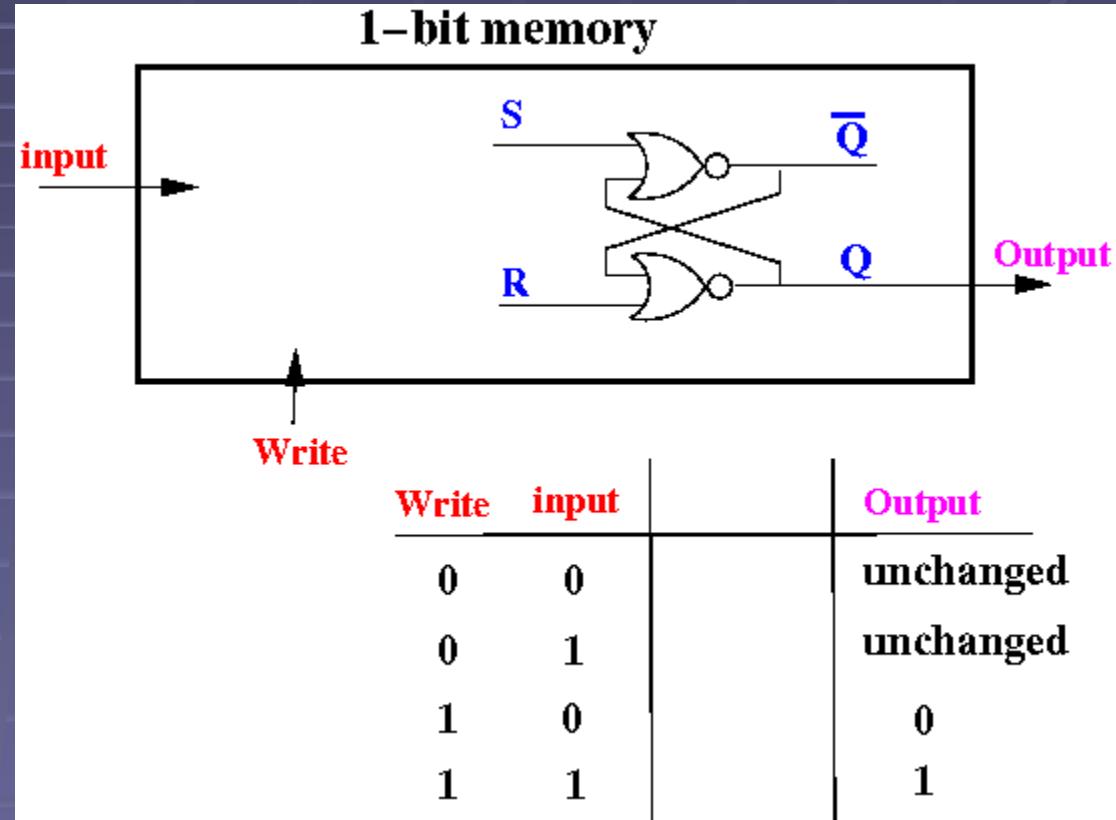
# Physical vs. Virtual Memory

Physical memory

No protection

Limited size

Sharing visible to processes



# Physical vs. Virtual Memory

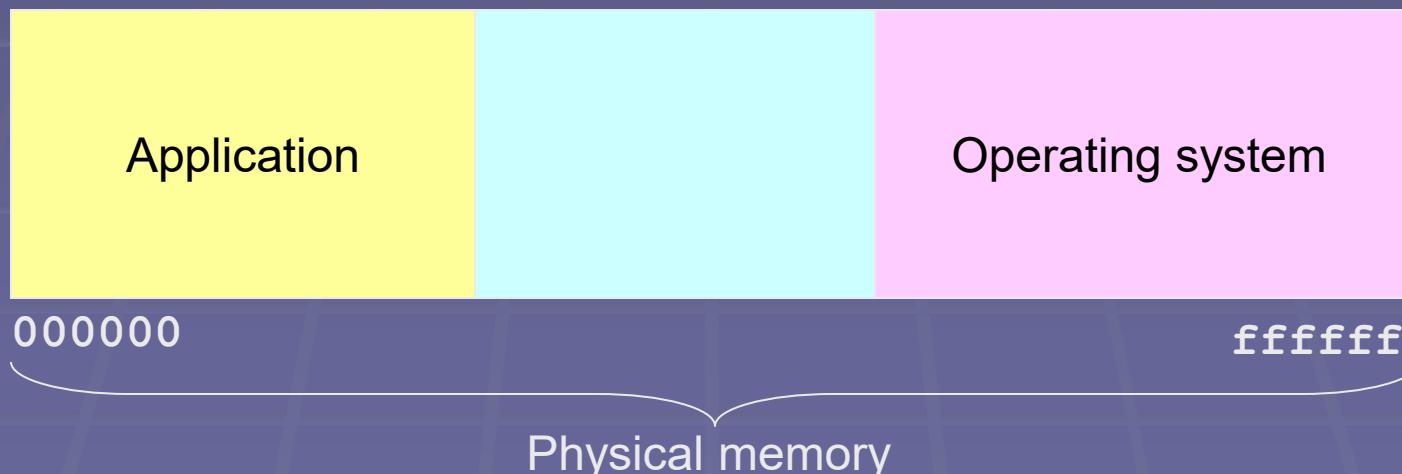
Physical memory	Virtual memory
No protection	Each process isolated from others and from OS
Limited size	Illusion of infinite memory
Sharing visible to processes	Each process cannot tell if memory is shared

# Memory Organizations

- Simplest: ***uniprogramming without memory protection***
  - Each application runs within a hardwired range of physical memory addresses
- One application runs at a time
  - Application can use the same physical addresses every time, across reboots
- E.g., Embedded System

# Uniprogramming Without Memory Protection

- Applications typically use the lower memory addresses
- An OS uses the higher memory addresses
- An application can address any physical memory location (may cause an OS crash)

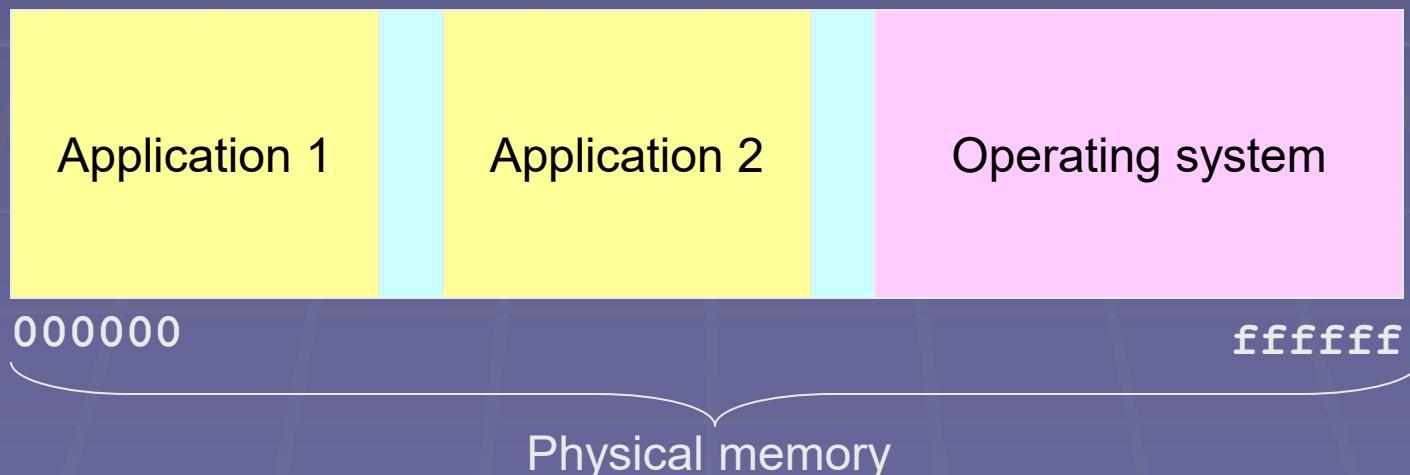


# ***Multiprogramming Without Memory Protection***

- When a program is copied into memory, a ***linker-loader*** alters the code of the program (e.g., loads, stores, and jumps)
  - To use the address of where the program lands in memory

# Multiprogramming Without Memory Protection

- Bugs in any program can cause other programs to crash, even the OS



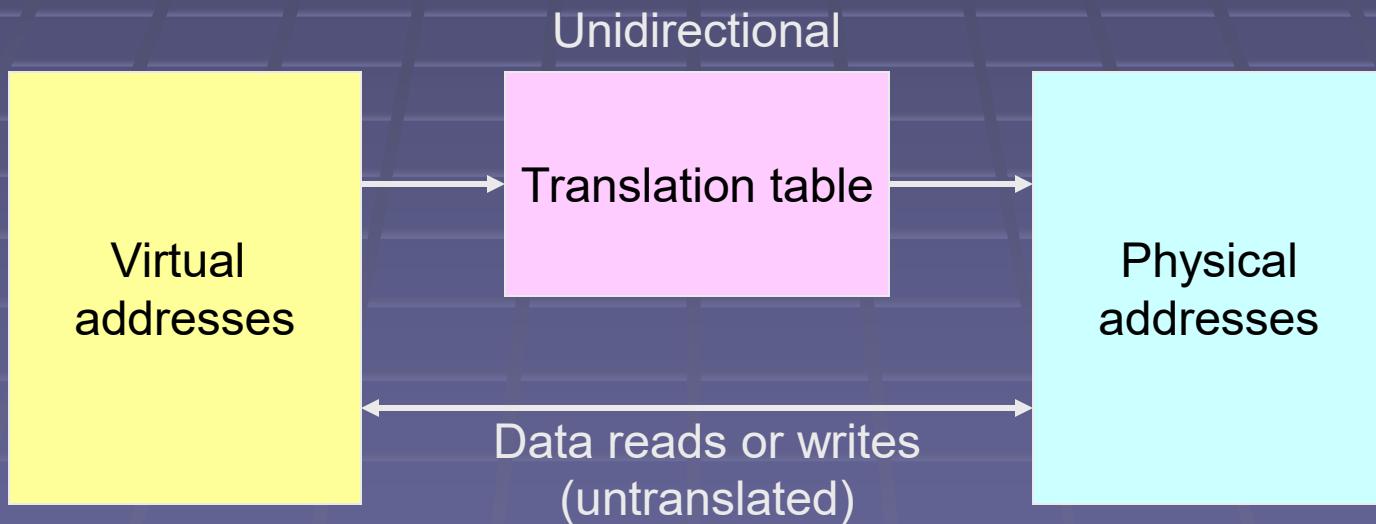
# ***Multiprogrammed OS With Memory Protection***

- ***Memory protection*** keeps user programs from crashing one another and the OS
- Two hardware-supported mechanisms
  - Address translation
  - Dual-mode operation

# Address Translation

- Each process is associated with an **address space**, or all the *physical* addresses a process can touch
- However, each process believes that it owns the entire memory, starting with the *virtual* address 0
- The missing piece is a translation table
  - Translate every memory reference from virtual to physical addresses

# Address Translation Visualized



# More on Address Translations

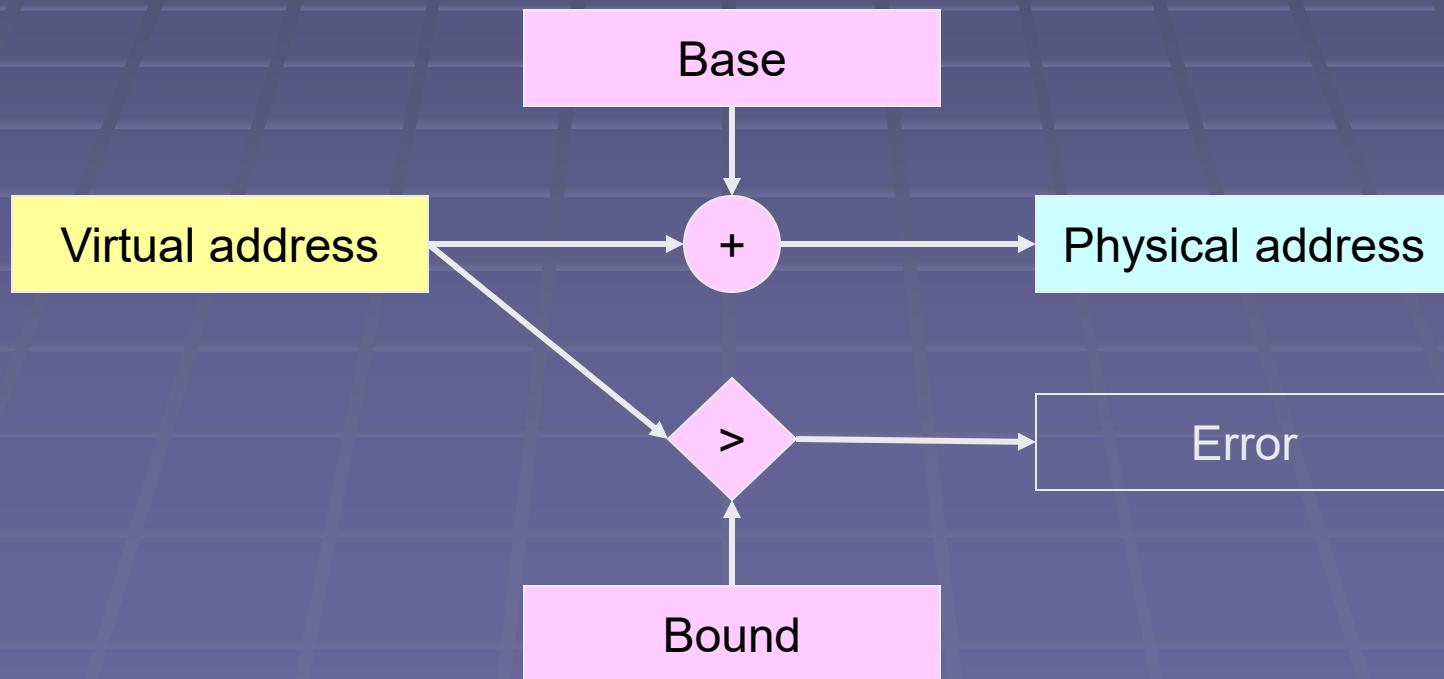
- Translation provides protection
  - Processes cannot talk about other processes' addresses, nor about the OS addresses
  - OS uses physical addresses directly
    - No translations

# Assumptions

- 32-bit machines
- 1-GB RAM max

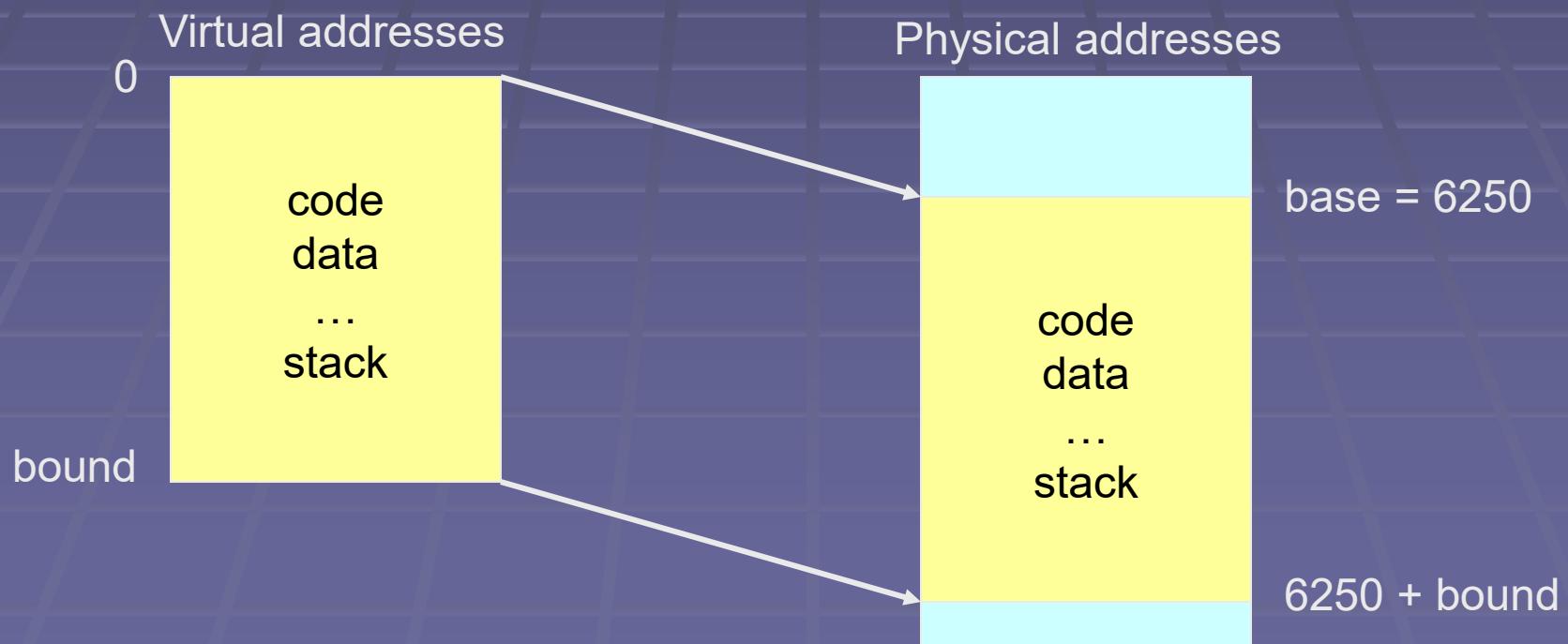
# *Base-and-Bound Translation*

- Each process is loaded into a contiguous region of physical memory
- Processes are protected from one another



# Base-and-Bound Translation

- Each process “thinks” that it owns a dedicated machine, with memory addresses from 0 to bound



# Base-and-Bound Translation

- An OS can move a process around
  - By copying bits
  - Changing the base and bound registers

# Pros/Cons of Base-and-Bound Translation

- + Simplicity
- + Speed
- ***External fragmentation:*** memory wasted because the available memory is not contiguous for allocation
- Difficult to share programs
  - Each instance of a program needs to have a copy of the code segment

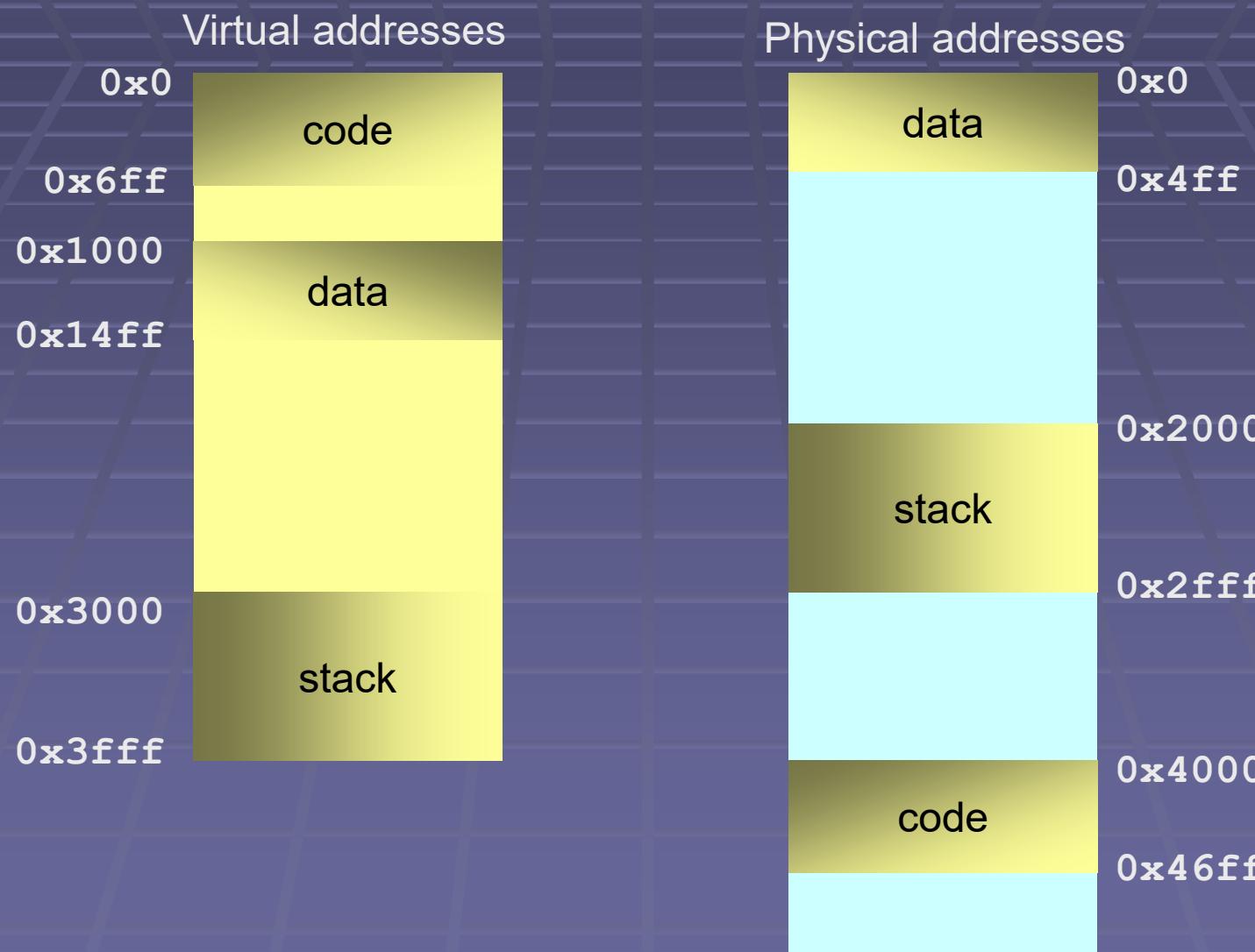
# Pros/Cons of Base-and-Bound Translation

- Memory allocation is complex
  - Need to find contiguous chunks of free memory
    - **First fit:** Use the first free memory region that is big enough
    - **Best fit:** Use the smallest free memory region
    - **Worst fit:** Use the largest free memory region
  - Reorganization involves copying
- Does not work well when address spaces grow and shrink dynamically

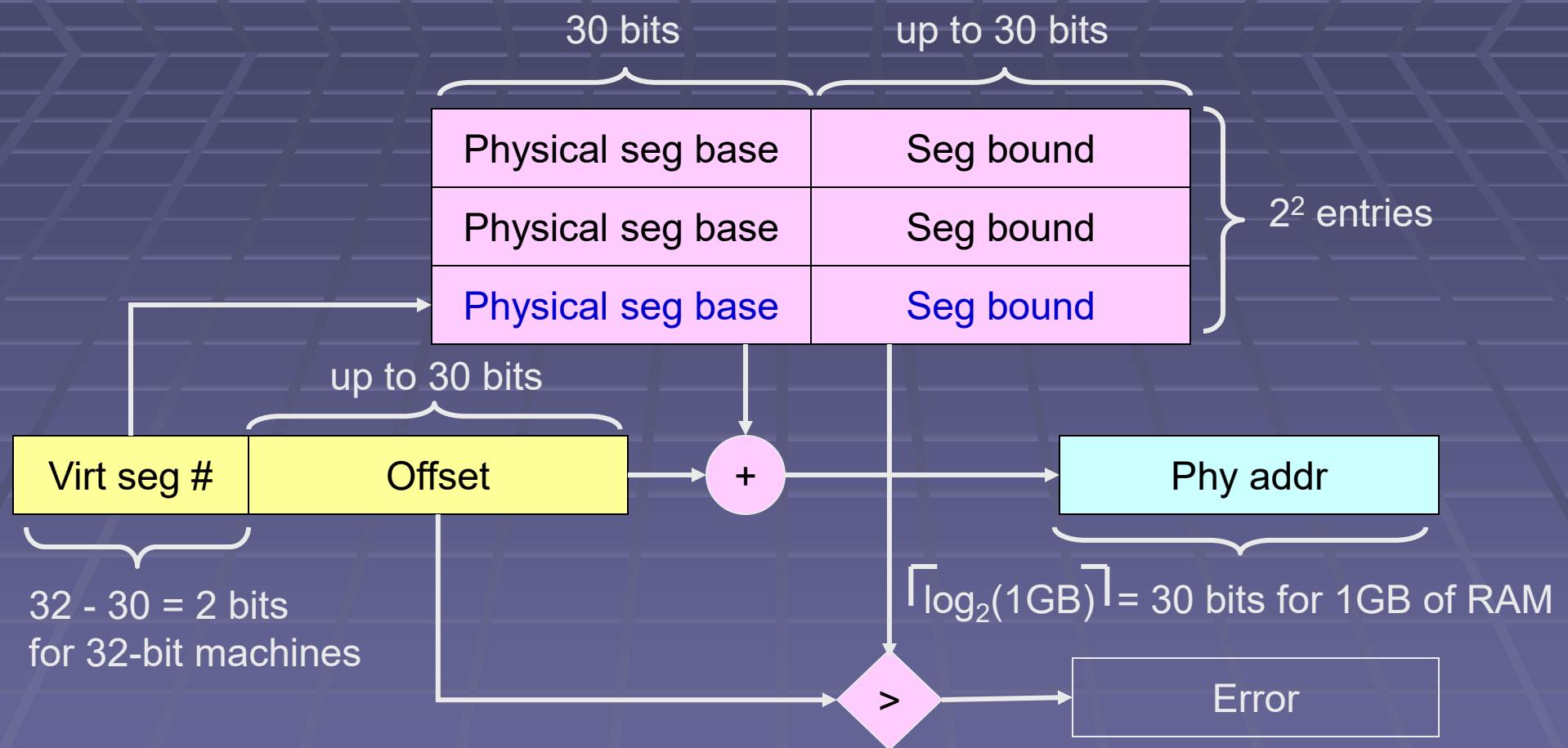
# Segmentation

- **Segment:** a logically contiguous memory region
- **Segmentation-based translation:** use a table of base-and-bound pairs

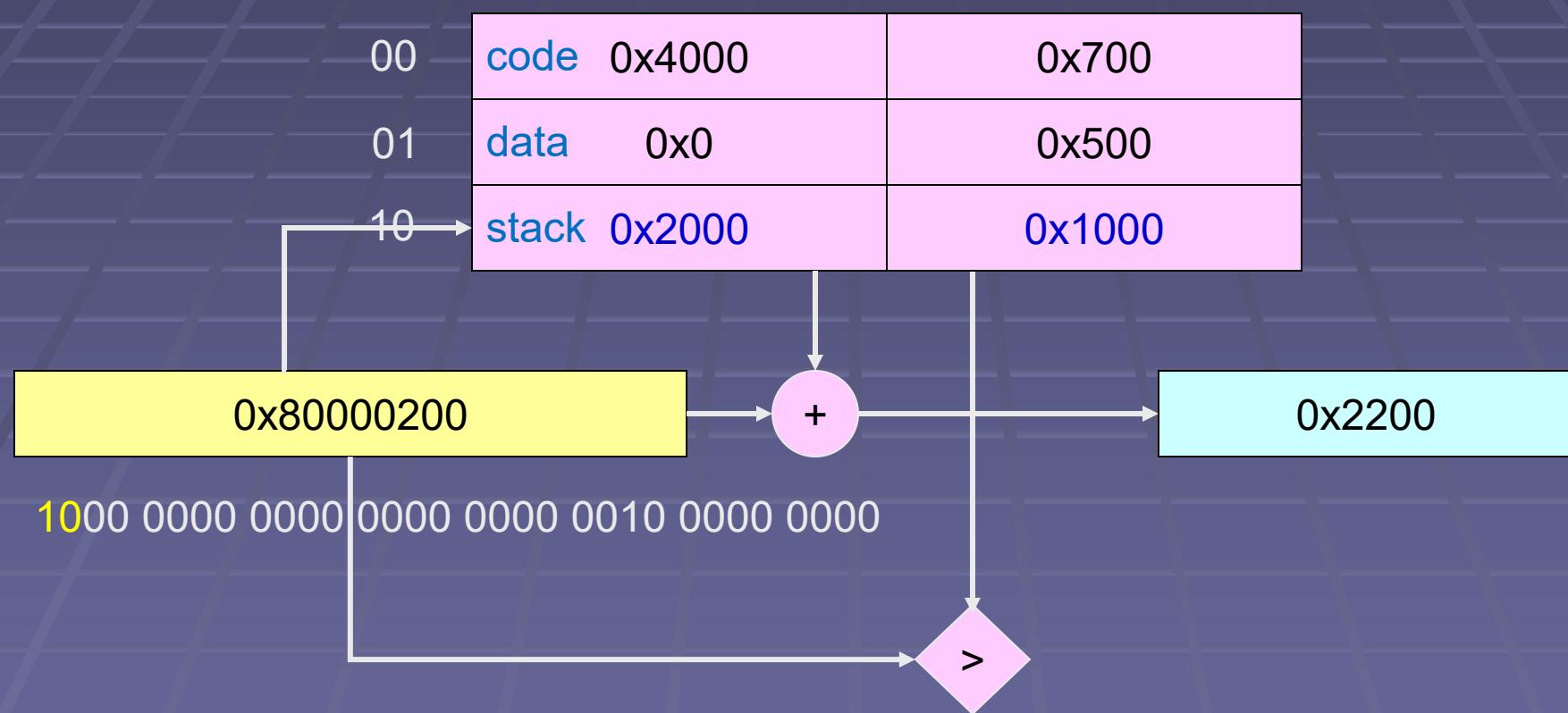
# Segmentation Illustrated



# Segmentation Diagram



# Segmentation Diagram



# Segmentation Diagram

00	code 0x4000	0x700
01	data 0x0	0x500
10	stack 0x2000	0x1000

0x40000100

+

?

>

# Segmentation Diagram

00	code 0x4000	0x700
01	data 0x0	0x500
10	stack 0x2000	0x1000

0x00000800

+

?

>

# Segmentation Translation

- $\text{virtual\_address} = \text{virtual\_segment\_number}:\text{offset}$
- $\text{physical\_base\_address} = \text{segment\_table}[\text{virtual\_segment\_number}]$
- $\text{physical\_address} = \text{physical\_base\_address} + \text{offset}$

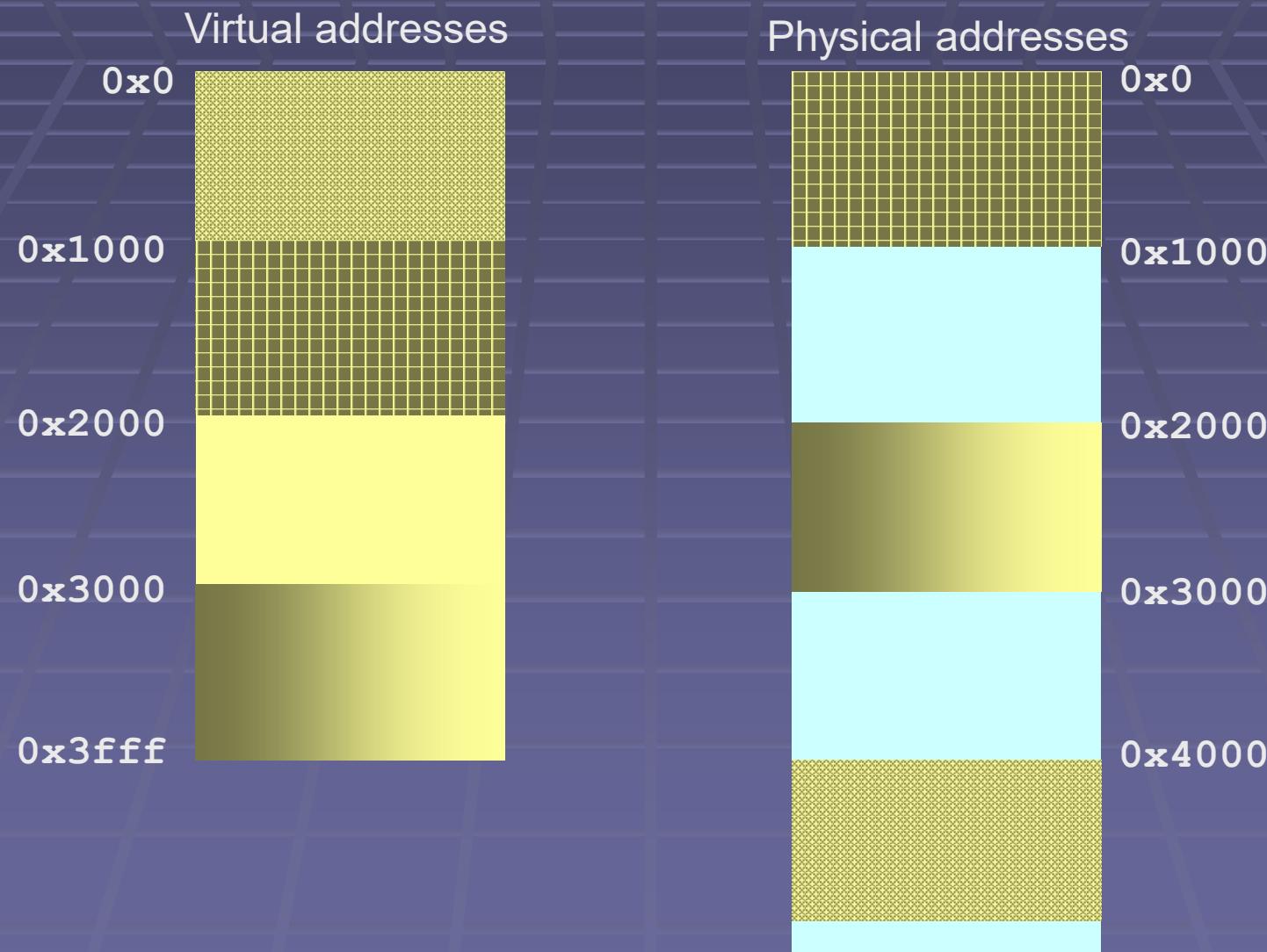
# Pros/Cons of Segmentation

- + Easier to grow/shrink individual segments
- + Finer control of segment accesses
  - e.g., read-only for shared code segment
  - Recall the semantics of fork()...
- + More efficient use of physical space
- + Multiple processes can share the same code segment
- Memory allocation is still complex
  - Requires contiguous allocation

# Paging

- ***Paging-based translation***: memory allocation via fixed-size chunks of memory, or ***pages***
- Uses a ***bitmap*** to track the allocation status of memory pages
- Translation granularity is a page

# Paging Illustrated

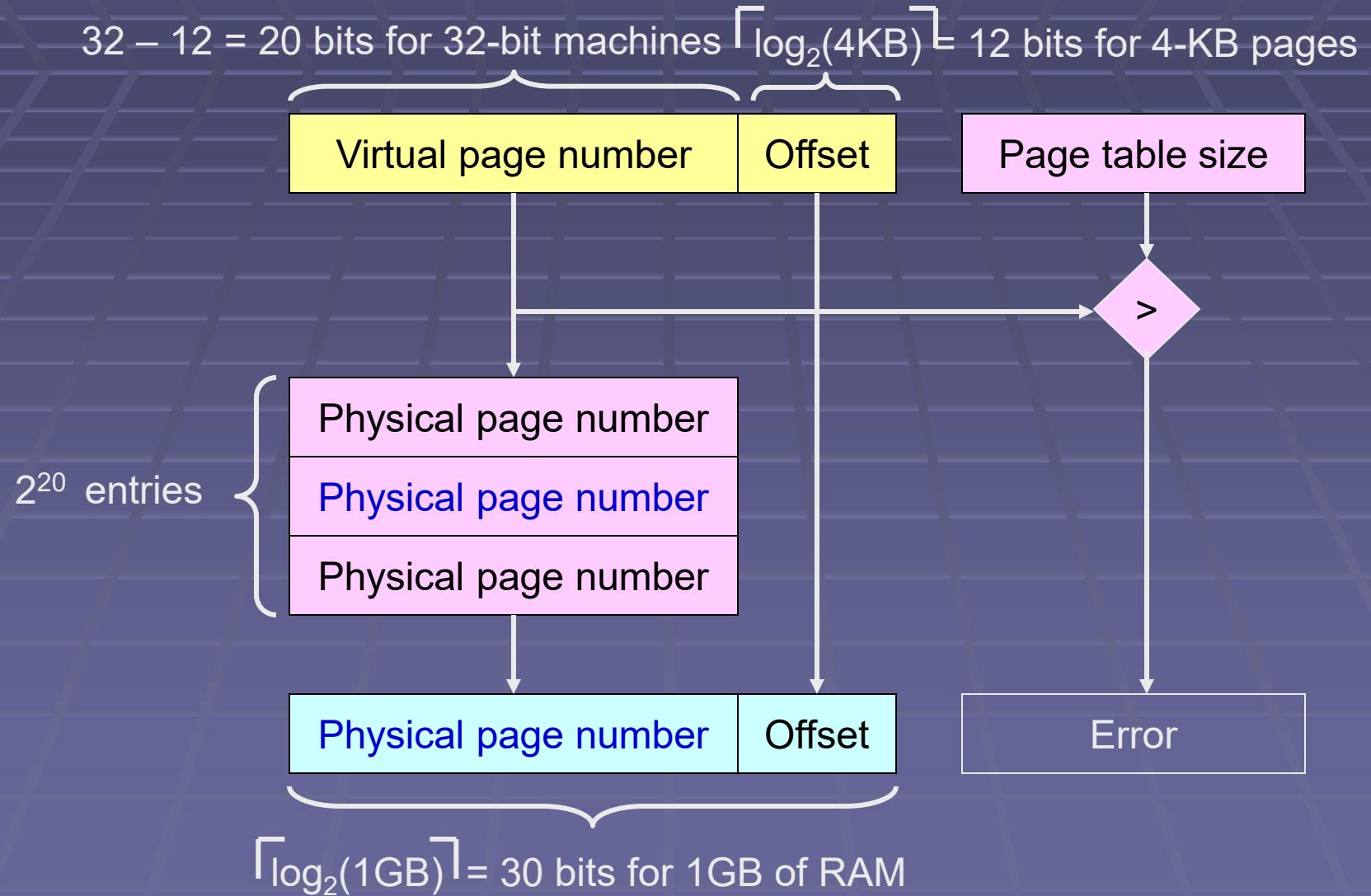


# Paged Memory Acces

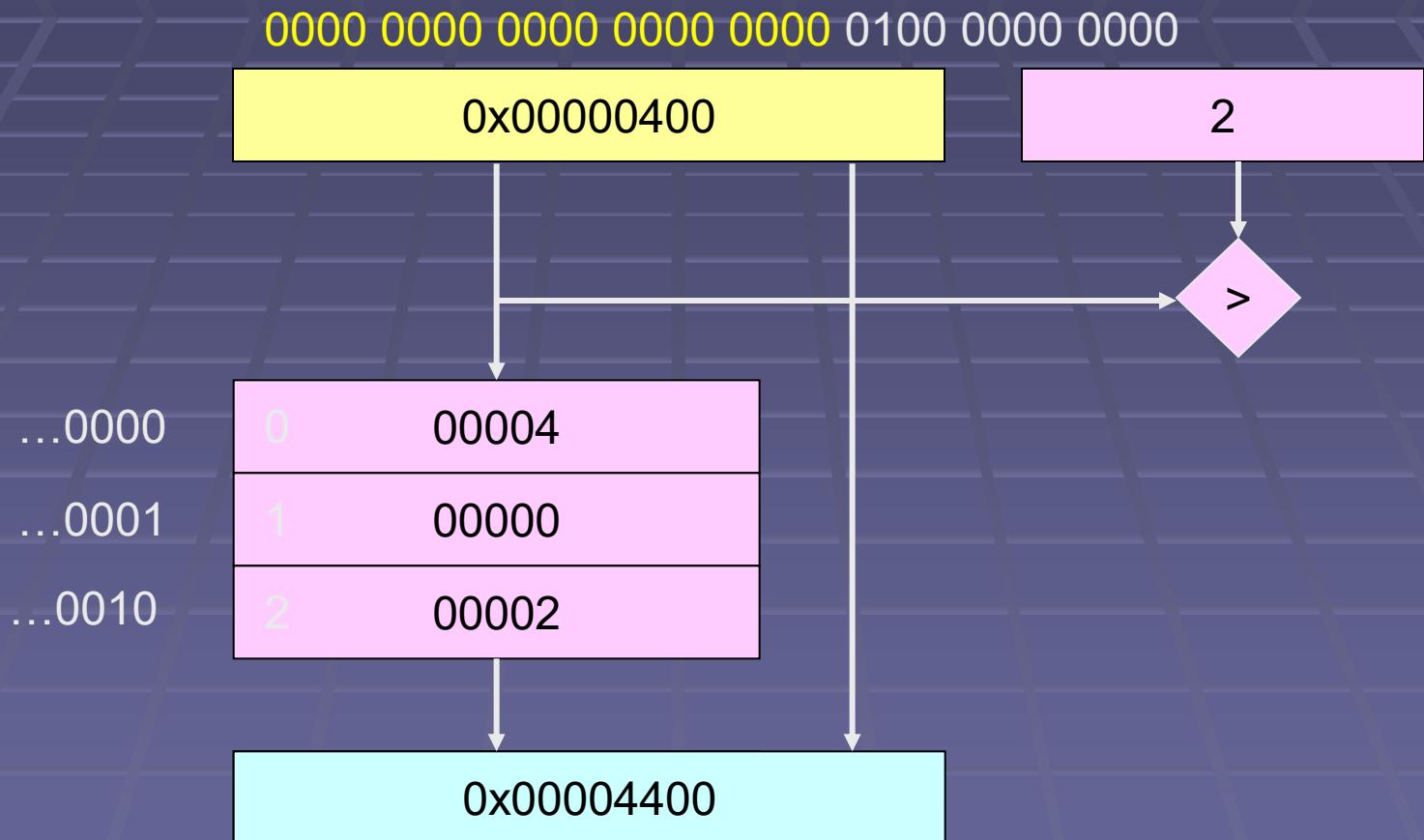
unsigned char memory[N\_PAGES][PAGE\_SIZE]

- To access
  - memory[virtual\_page\_number][page\_offset]

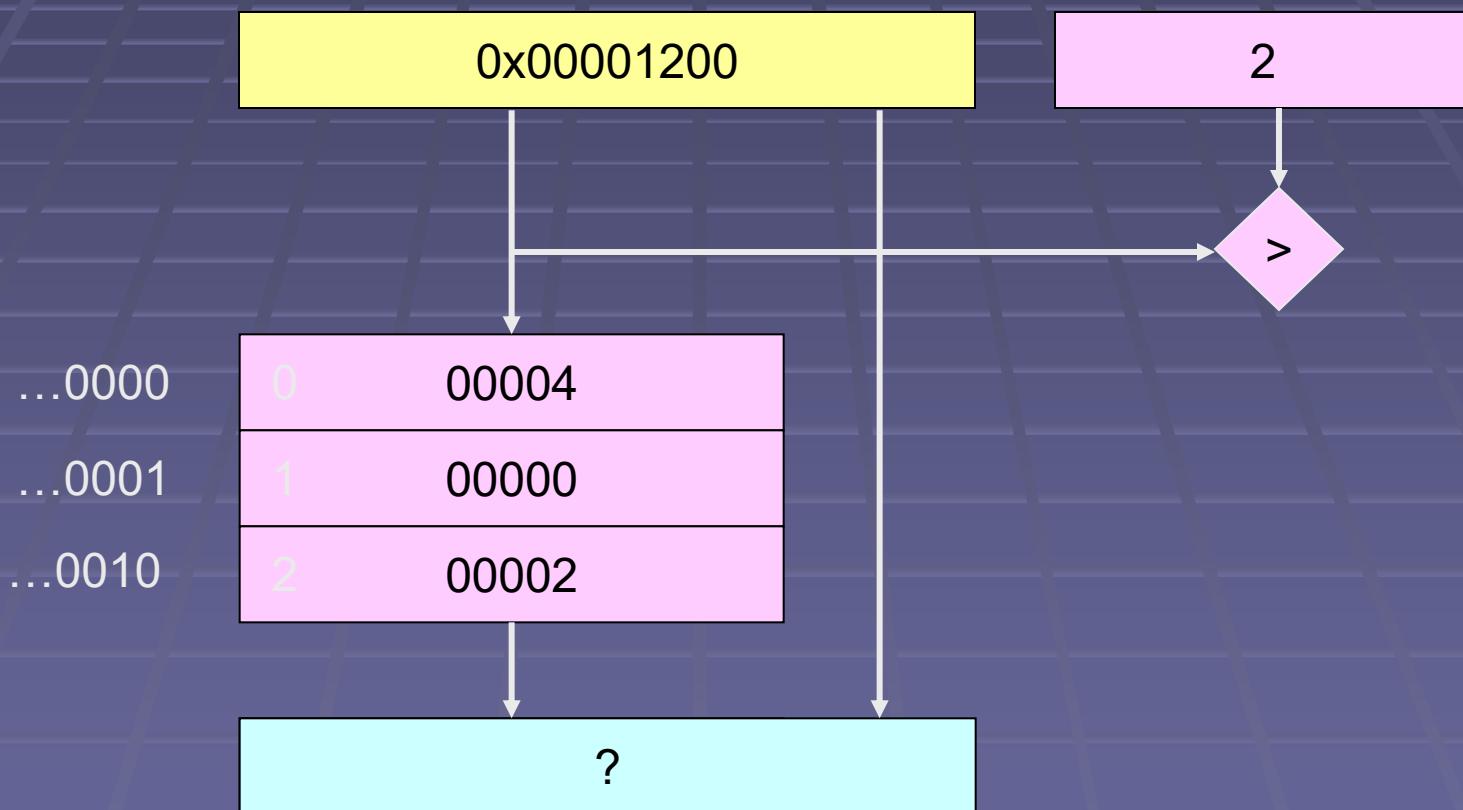
# Paging Diagram



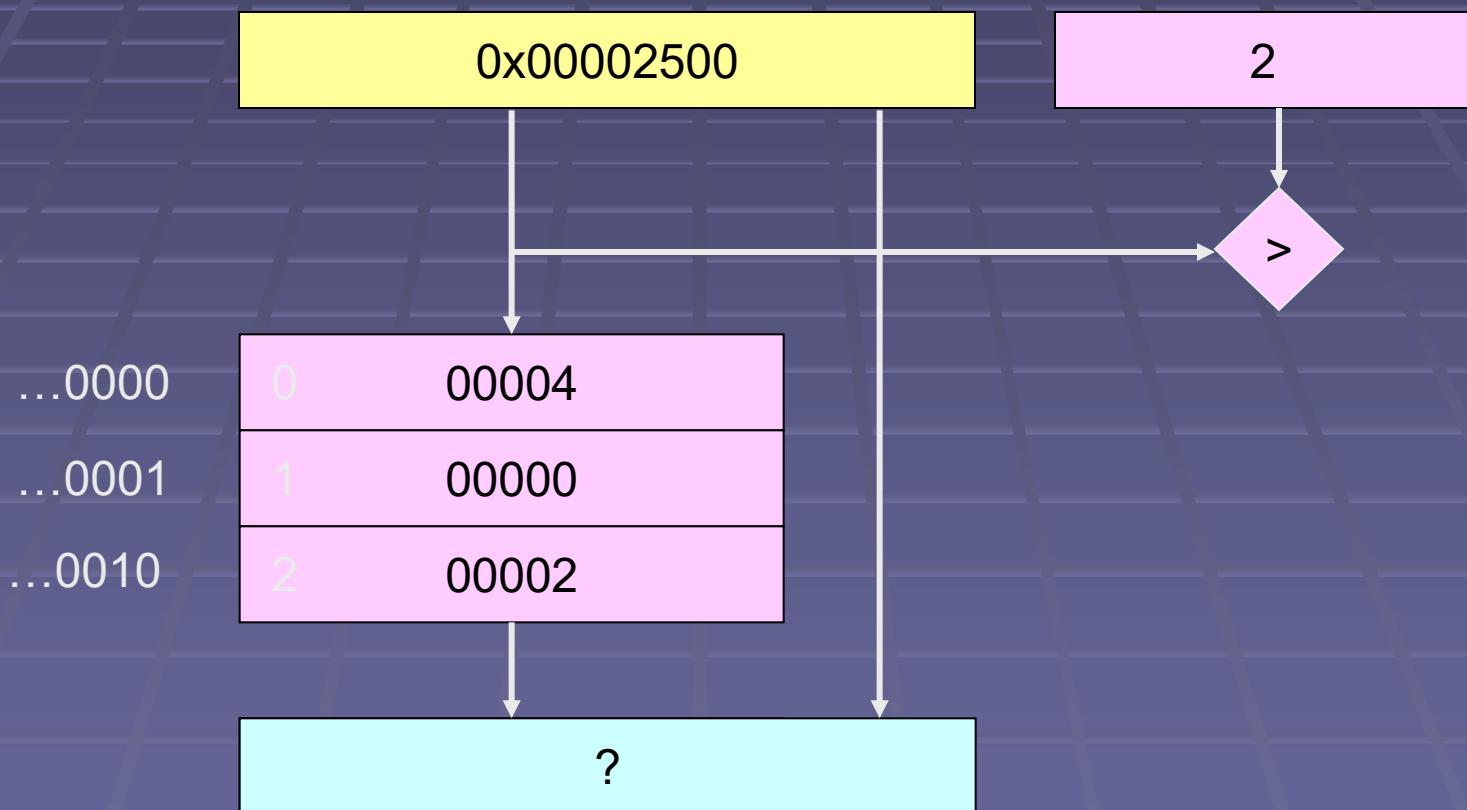
# Paging Example



# Paging Example



# Paging Example



# Paging Translation

- $\text{virtual\_address} = \text{virtual\_page\_number}:\text{offset}$
- $\text{physical\_page\_number} = \text{page\_table}[\text{virtual\_page\_number}]$
- $\text{physical\_address} = \text{physical\_page\_number}:\text{offset}$

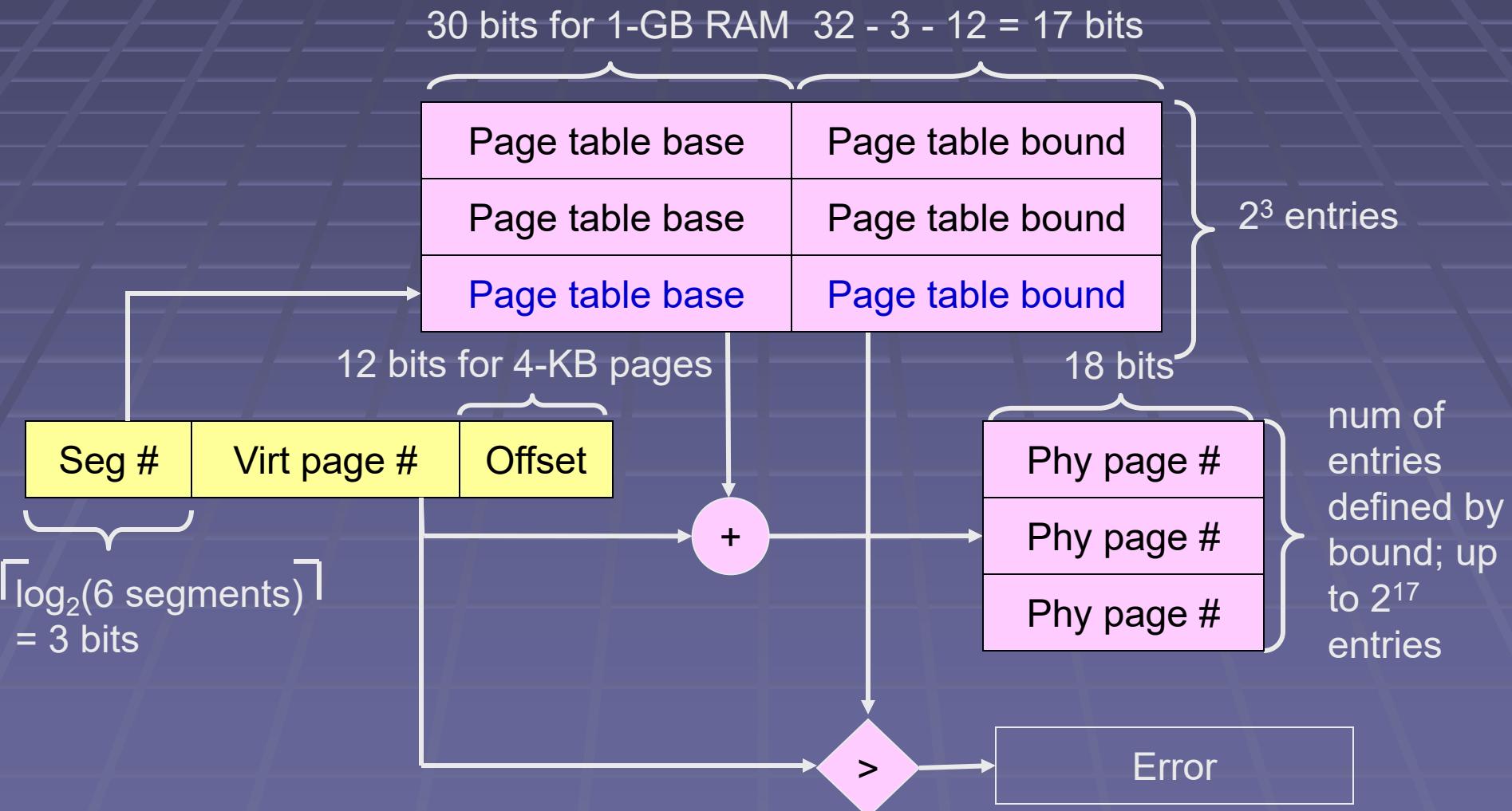
# Pros and Cons of Paging

- + Easier memory allocation
- + Allows code sharing
- ***Internal fragmentation:*** allocated pages are not fully used
- Page table can potentially be very large
  - 32-bit architecture with 1-KB pages can require 4M table entries

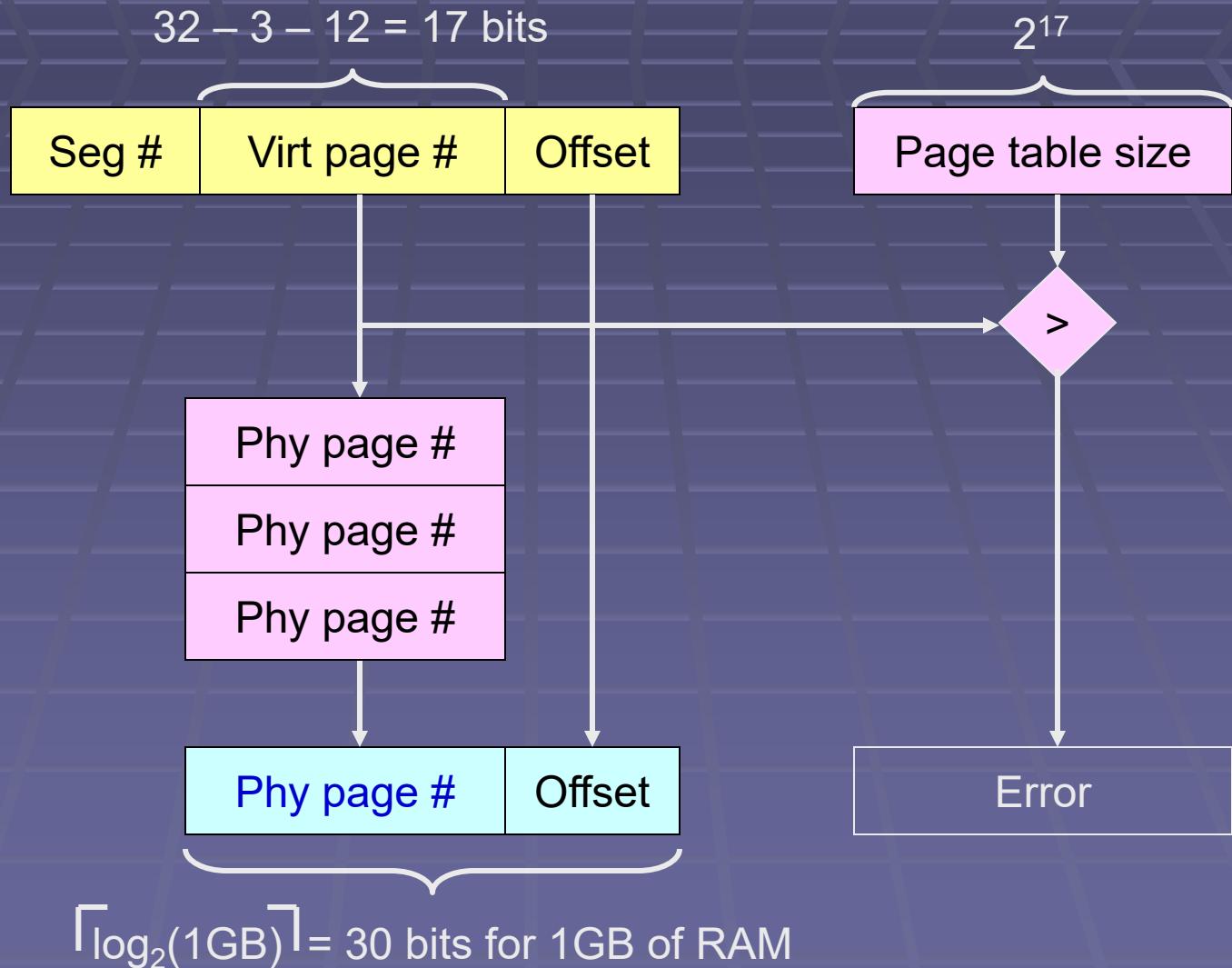
# Multi-Level Translation

- ***Segmented-paging translation***: breaks the page table into segments
- ***Paged page tables***: Two-level tree of page tables

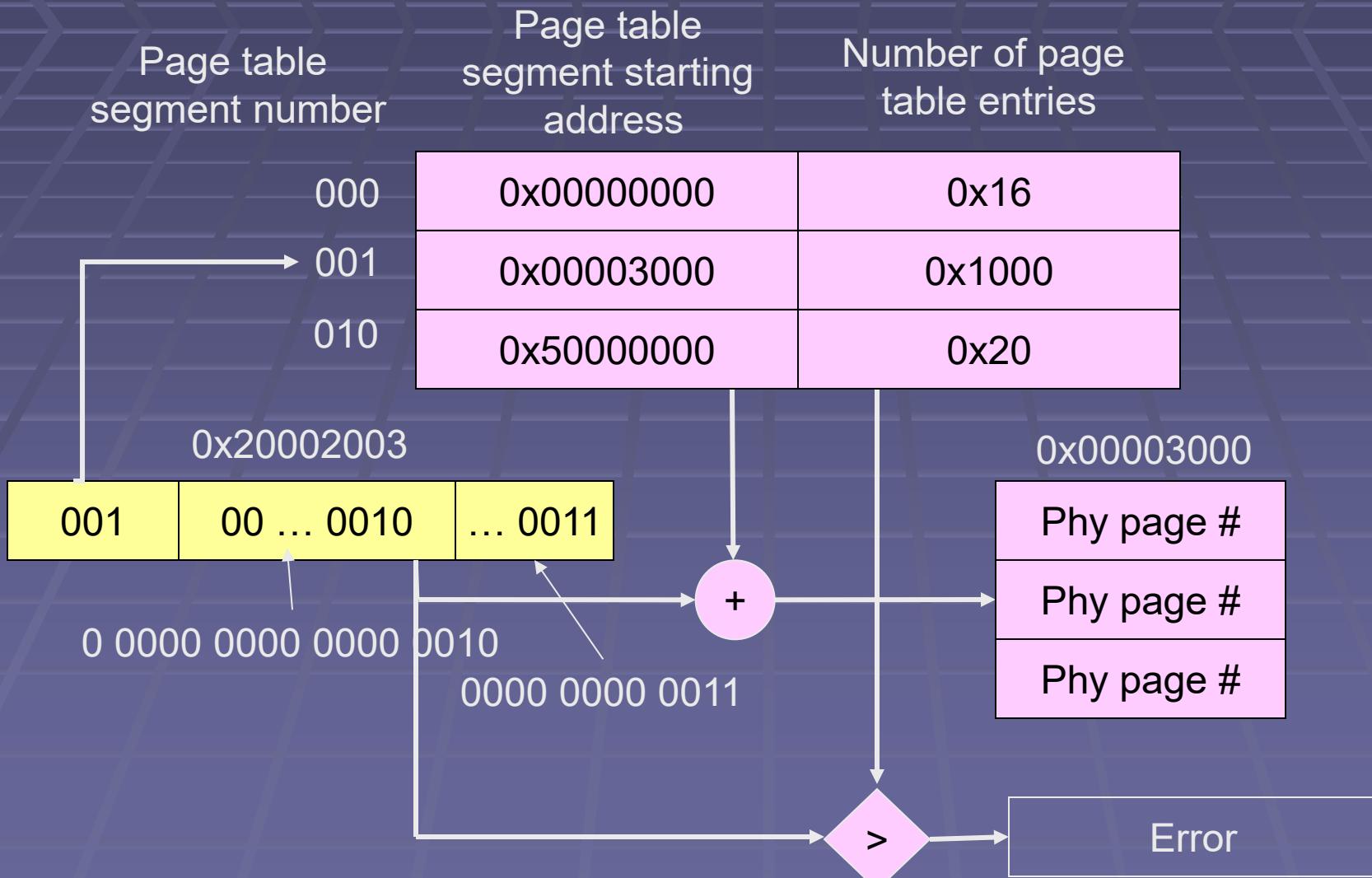
# Segmented Paging



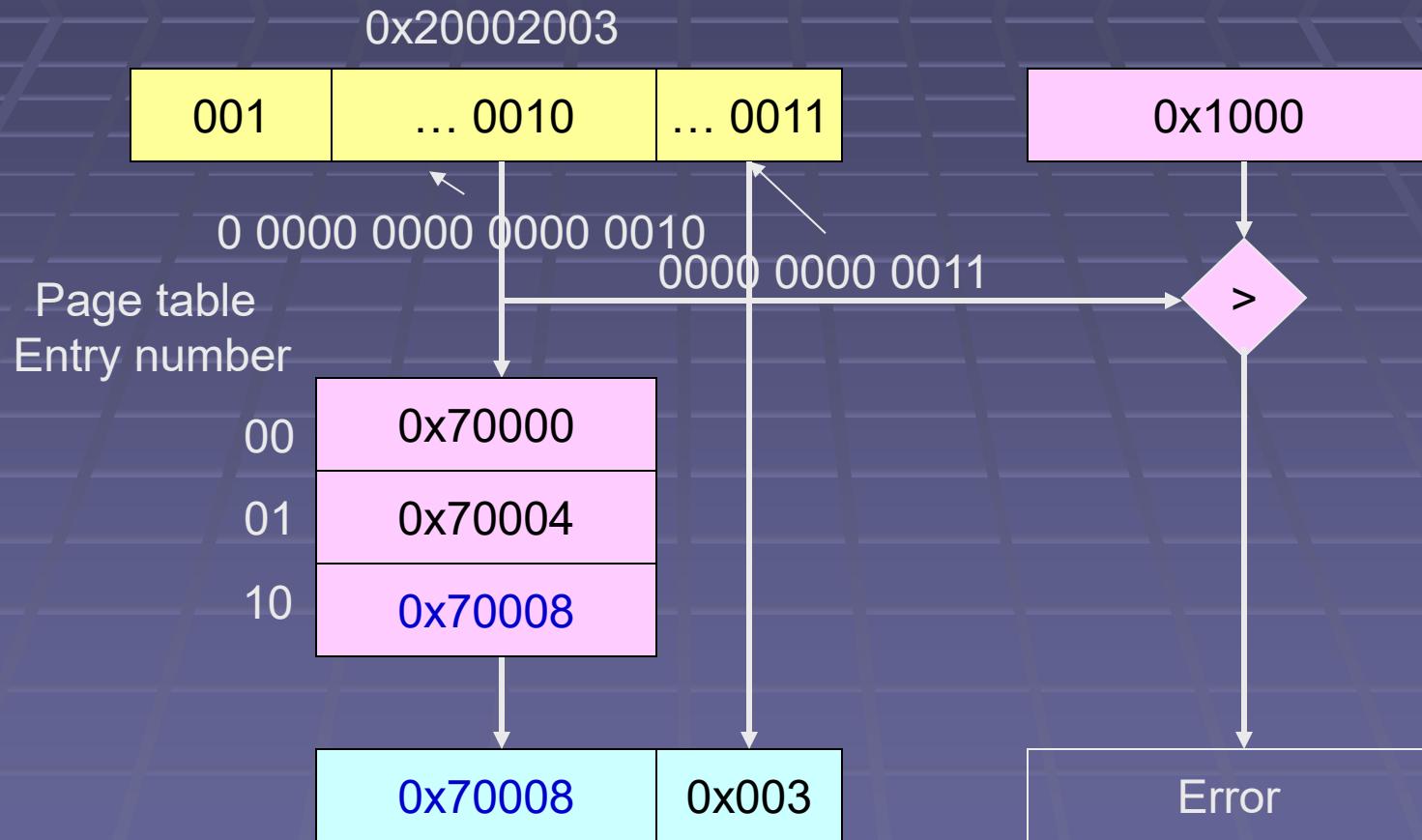
# Segmented Paging



# Segmented Paging Example



# Segmented Paging



# Segmented Paging Example

Page table  
segment number

Page table  
segment starting  
address

Number of page  
Table entries

000

0x00000000

0x16

001

0x00003000

0x1000

010

0x50000000

0x20

0x21002005

0x00003000

?

?

?

?

?

+

Phy page #

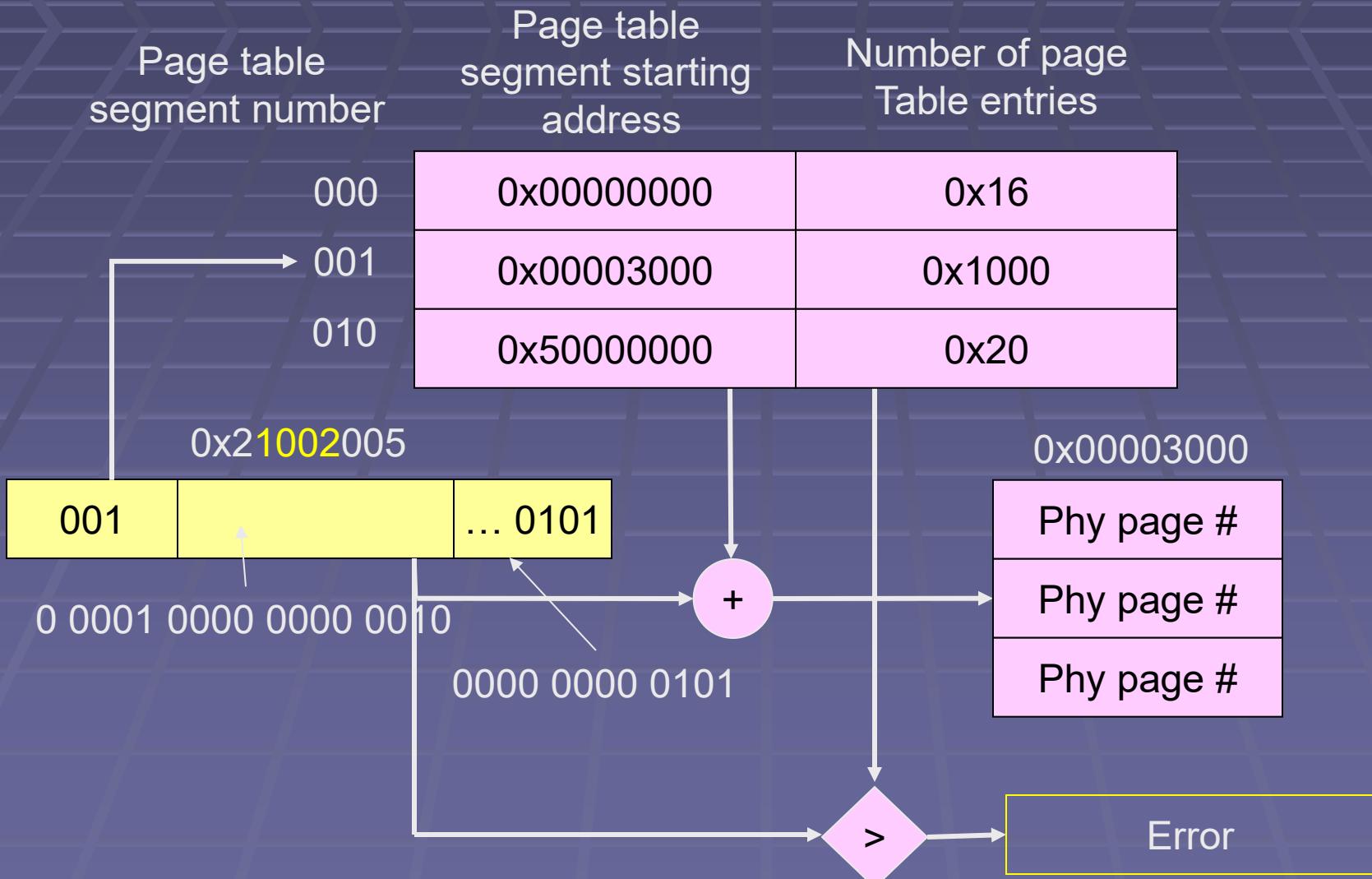
Phy page #

Phy page #

>

Error

# Segmented Paging Example



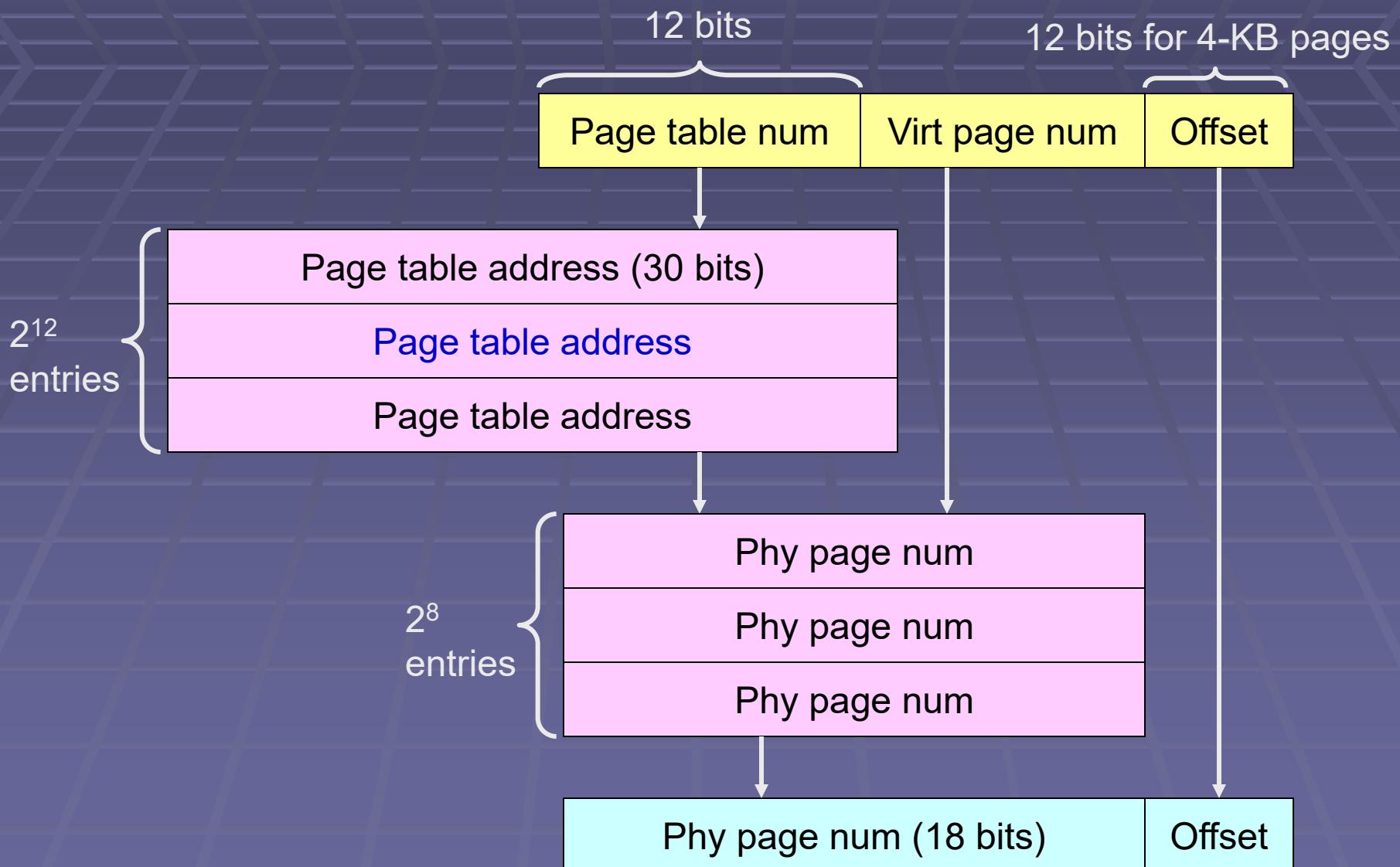
# Segmented Paging Translation

- virtual\_address =  
segment\_number:page\_number:offset
- page\_table =  
segment\_table[segment\_number]
- physical\_page\_number =  
page\_table[virtual\_page\_number]
- physical\_address =  
physical\_page\_number:offset

# Pros/Cons of Segmented Paging

- + Code sharing
- + Reduced memory requirements for page tables
- Higher overhead and complexity
- Page tables still need to be contiguous
- Two lookups per memory reference

# Paged Page Tables



# Paged Page Table Translation

- $\text{virtual\_address} = \text{page\_table\_num}:\text{virtual\_page\_num}:\text{offset}$
- $\text{page\_table} = \text{page\_table\_address}[\text{page\_table\_num}]$
- $\text{physical\_page\_num} = \text{page\_table}[\text{virtual\_page\_num}]$
- $\text{physical\_address} = \text{physical\_page\_num}:\text{offset}$

# Pros/Cons of Paged Page Tables

- + Can be generalized into multi-level paging
- Multiple memory lookups are required to translate a virtual address
  - Can be accelerated with ***translation lookaside buffers (TLBs)***
  - Store recently translated memory addresses for short-term reuses

# Hashed Page Tables

- Physical\_address
  - = hash(virtual\_page\_num):offset
- + Conceptually simple
- Need to handle collisions
- Need one hash table per address space

# Inverted Page Table

- One hash entry per physical page
- `physical_address`  
=  $\text{hash}(\text{pid}, \text{virtual\_page\_num}) : \text{offset}$
- + The number of page table entries is proportional to the size of physical RAM
- Collision handling

# Dual-mode Operation Revisited

- Translation tables offer protection if they cannot be altered by applications
- An application can only touch its address space under the user mode
- HW requires the CPU to be in the kernel mode to modify the address translation tables

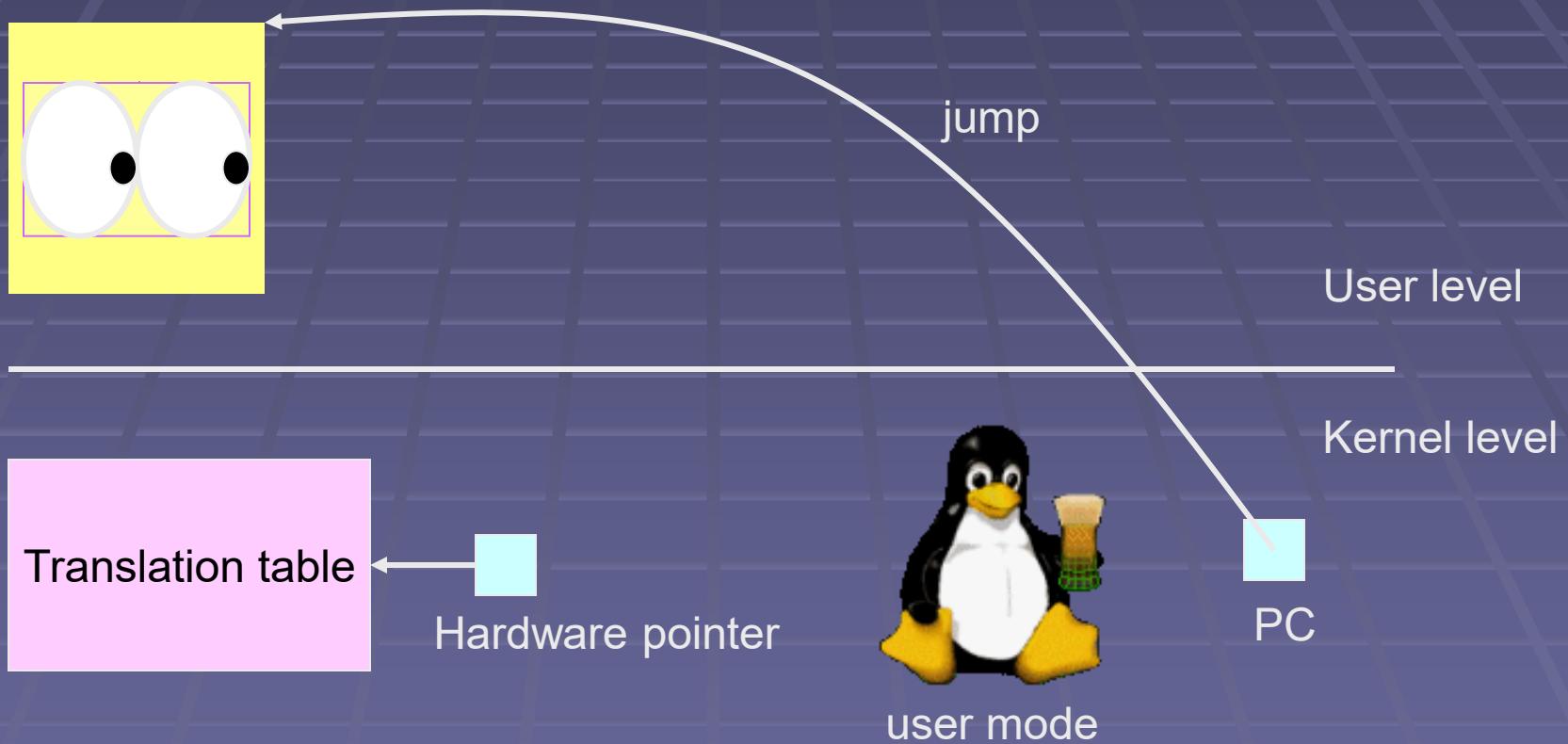
# Details of Dual-mode Operations

- How the CPU is shared between the kernel and user processes
- How processes interact among themselves

# Switching from the Kernel to User Mode

- To run a user program, the kernel
  - Creates a process and initialize the address space
  - Loads the program into the memory
  - Initializes translation tables
  - Sets the HW pointer to the translation table
  - Sets the CPU to user mode
  - Jumps to the entry point of the program

# To Run a Program



# Switching from User Mode to Kernel Mode

- Voluntary
  - **System calls**: a user process asks the OS to do something on the process's behalf
- Involuntary
  - Hardware interrupts (e.g., I/O)
  - Program exceptions (e.g., segmentation fault)

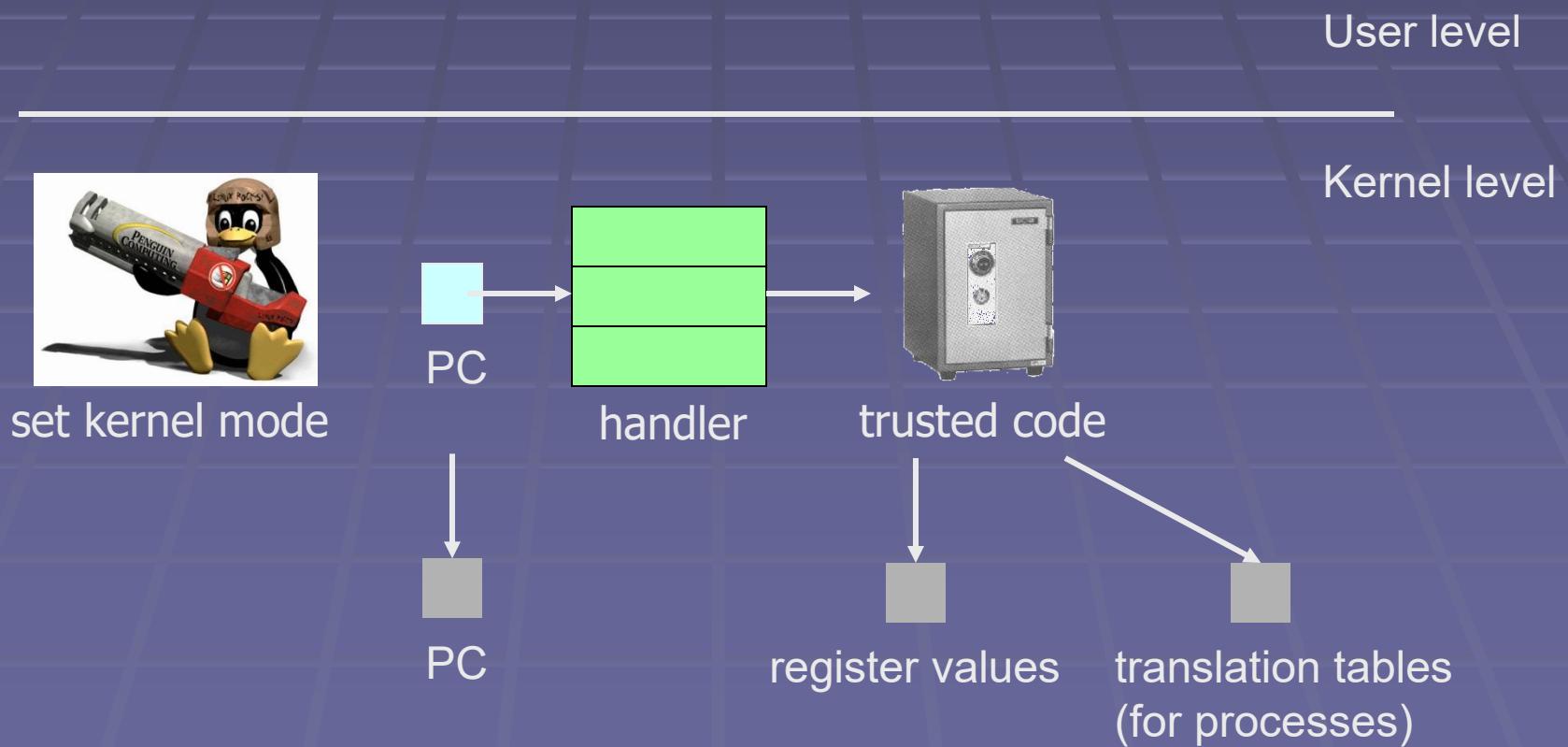
# Switching from User Mode to Kernel Mode

- For all cases, hardware atomically performs the following steps
  - Sets the CPU to kernel mode
  - Saves the current program counter
  - Jumps to the handler in the kernel
    - The handler saves old register values

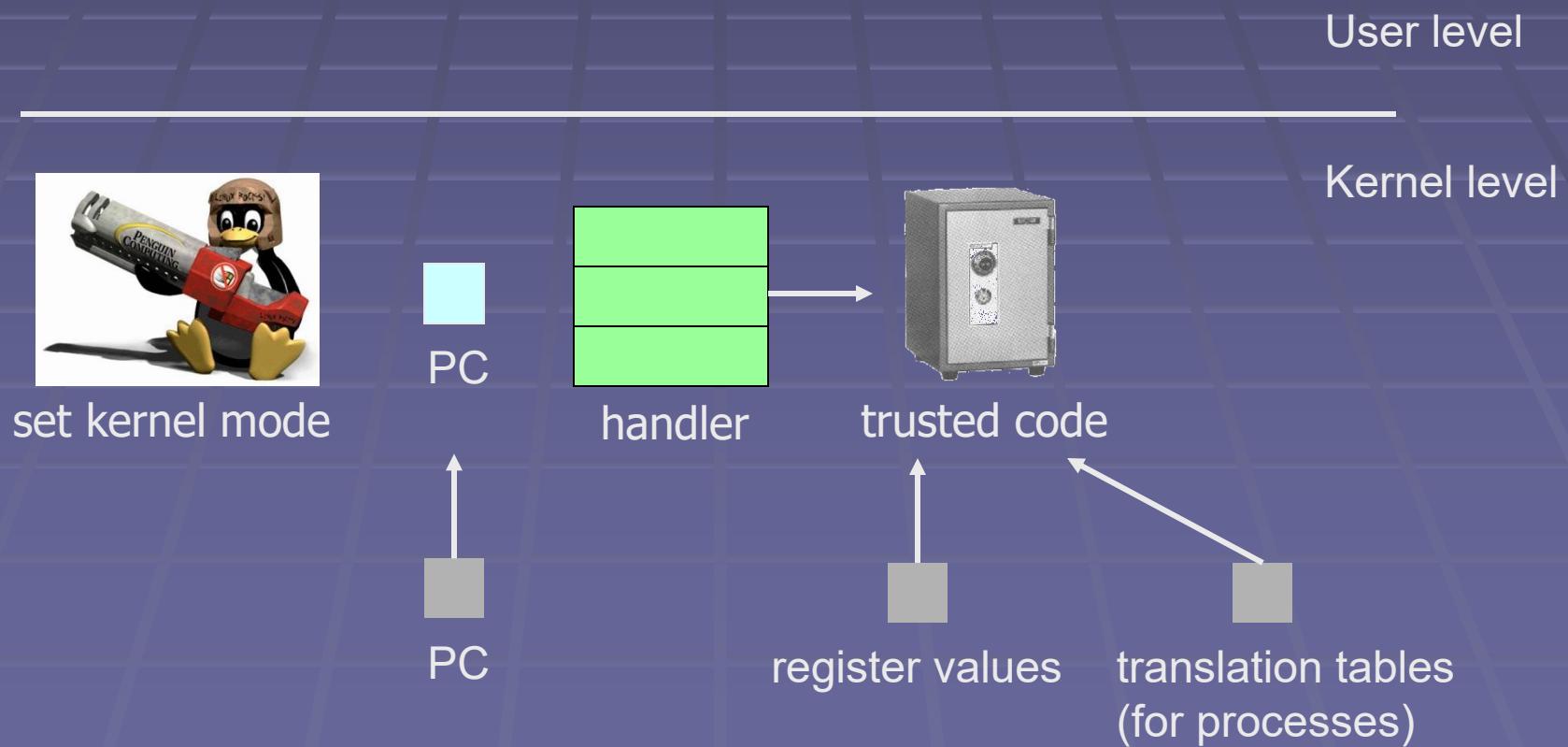
# Switching from User Mode to Kernel Mode

- Unlike context switching among threads, to switch among processes
  - Need to save and restore pointers to translation tables
- To resume process execution
  - Kernel reloads old register values
  - Sets CPU to user mode
  - Jumps to the old program counter

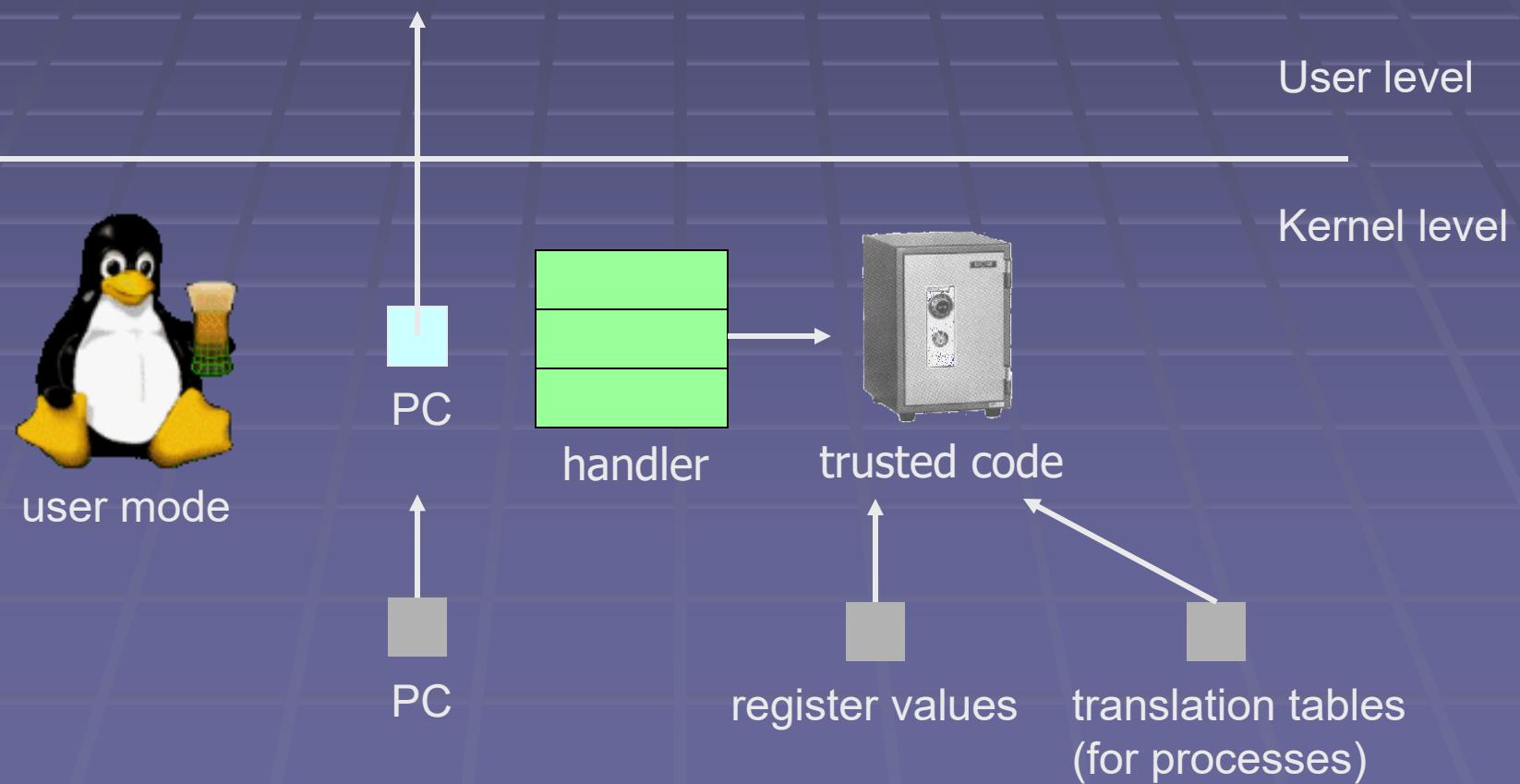
# User → Kernel



# Kernel → User



# Kernel → User



# Communication Between Address Spaces

- Processes communicate among address spaces via ***interprocess communication (IPC)***
  - Byte stream (e.g., `pipe`)
  - Message passing (send/receive)
  - File system (e.g., read and write files)
  - Shared memory
- Bugs can propagate from one process to another

# Interprocess Communication

- Direct
  - `send(P1, message);`
  - `receive(P2, message);`
  - One-to-one communication
- Indirect
  - Mailboxes or ports
  - `send(mailbox_A, message);`
  - `receive(mailbox_A, message);`
  - Many-to-many communication

# Protection Without HW Support

- HW-supported protection can be slow
  - Requires applications be separated into address spaces to achieve fault isolation
- What if your apps are built by multiple vendors? (e.g., Chrome plug-ins)
  - Can we run two programs in the same address space, with safety guarantees?

# Protection via Strong Typing

- Programming languages may disallow the misuse of data structures (casting)
  - e.g., LISP and Java
- Java has its own virtual machines
  - A Java program can run on different HW and OSes

# Protection via Software Fault Isolation

- Compilers generate code that is provably safe
  - e.g., a pointer cannot reference illegal addresses
- With aggressive optimizations, the overhead can be as low as 5%

# Protection via Software Fault Isolation

Original instruction	Compiler-modified version
<code>st r2, (r1)</code>	<code>safe = a legal address</code>
	<code>safe = r1</code>
	Check <code>safe</code> is still legal
	<code>st r2, (safe)</code>

- A malicious user cannot jump to the last line and do damage, since `safe` is a legal address

# Demand Paged Virtual Memory

# Up to this point...

- We assume that a process needs to load all of its address space before running
  - e.g., 0x0 to 0xFFFFFFFF
- Observation: 90% of time is spent on 10% of code

# Demand Paging

- ***Demand paging:*** allows pages that are referenced actively to be loaded into memory
  - Remaining pages stay on disk
  - Provides the illusion of infinite physical memory

# Demand Paging Mechanism

- Page tables sometimes need to point to disk locations (as opposed to memory locations)
- A table entry needs a *present (valid)* bit
  - Present means a page is in memory
  - Not present means that there is a ***page fault***

# Page Fault

- Hardware trap
- OS performs the following steps while running other processes (analogy: firing and hiring someone)
  - Choose a page
  - If the page has been modified, write its contents to disk
  - Change the corresponding page table entry and TLB entry
  - Load new page into memory from disk
  - Update page table entry
  - Continue the thread

# Transparent Page Faults

- ***Transparent*** (invisible) mechanisms
  - A process does not know how it happened
  - It needs to save the processor states and the faulting instruction

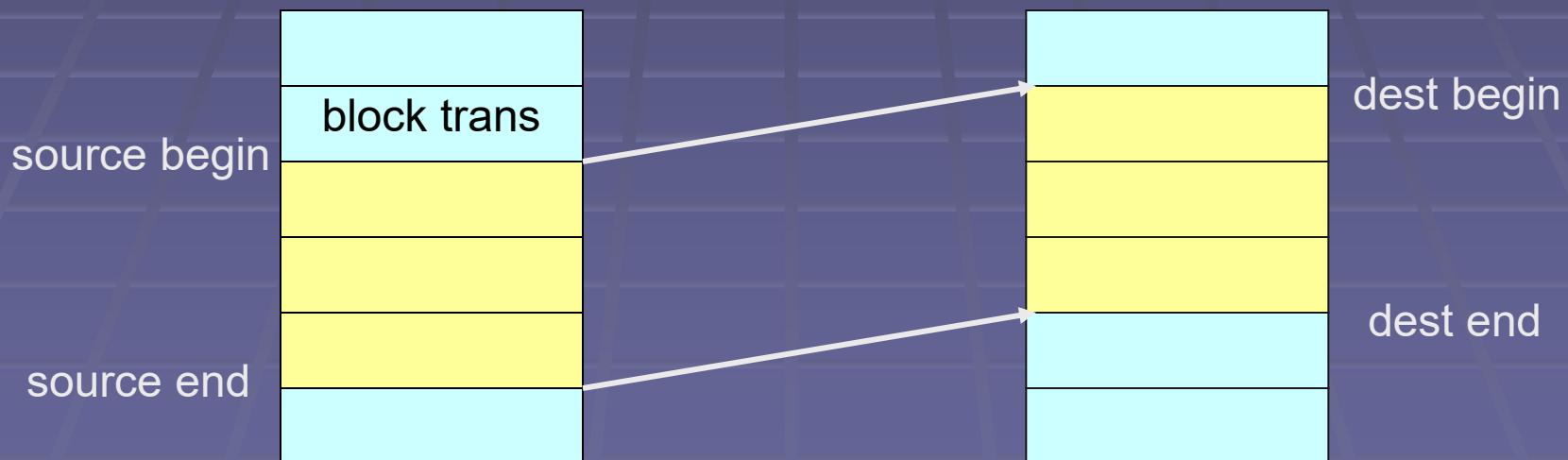
# More on Transparent Page Faults

- An instruction may have side effects
  - Hardware needs to either unwind or finish off those side effects

```
ld r1, x
// page fault, x not in memory
```

# More on Transparent Page Faults

- Hardware designers need to understand virtual memory
  - Unwinding instructions not always possible
  - Example: block transfer instruction



# Page Replacement Policies

- ***Random replacement:*** replace a random page
  - + Easy to implement in hardware (e.g., TLB)
  - May toss out useful pages
- ***First in, first out (FIFO):*** toss out the oldest page
  - + Fair for all pages
  - May toss out pages that are heavily used

# More Page Replacement Policies

- ***Optimal (MIN)***: replaces the page that will not be used for the longest time
  - + Optimal
  - Does not know the future
- ***Least-recently used (LRU)***: replaces the page that has not been used for the longest time
  - + Good if past use predicts future use
  - Tricky to implement efficiently

# More Page Replacement Policies

- ***Least frequently used (LFU)***: replaces the page that is used least often
  - Tracks usage count of pages
    - + Good if past use predicts future use
    - Difficult to replace pages with high counts

# Example

- A process makes references to 4 pages: A, B, E, and R
- Reference stream: BEERBAREBEAR
- Physical memory size: 3 pages

# FIFO

↓

# FIFO

↓

# FIFO

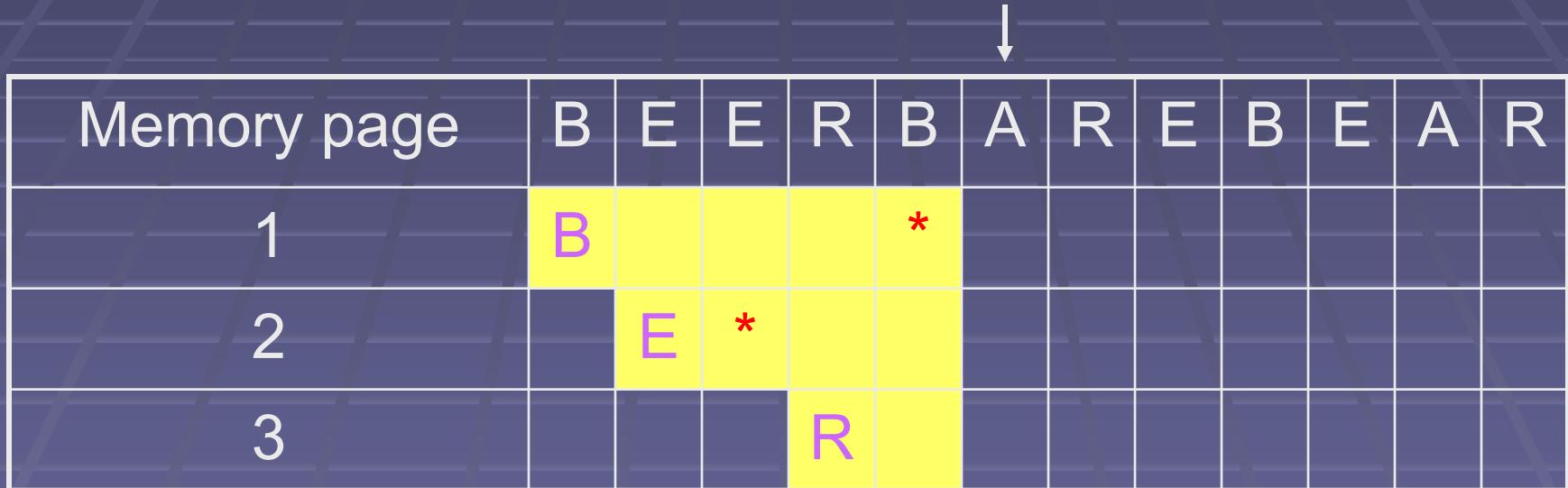
# FIFO

1

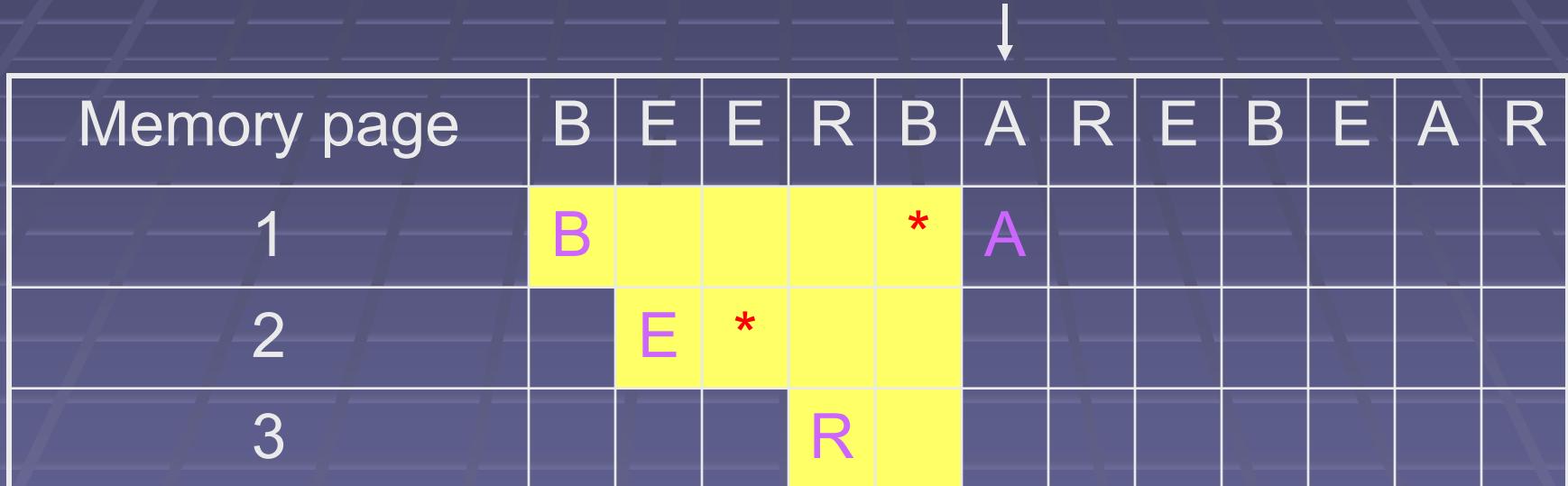
# FIFO

4

# FIFO



# FIFO



# FIFO

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*	A						
2		E	*									
3				R			*					

# FIFO

The diagram illustrates a FIFO (First-In, First-Out) memory system. At the top, the word "BEER BEAR" is written in a sequence of alternating uppercase letters. A vertical arrow points downwards from the letter 'R' in "BEER" towards a 3x10 grid below. The grid is divided into three horizontal rows labeled "Memory page" and "1", "2", "3" on the left, and ten vertical columns. The first column under "Memory page" contains the text "Memory page". The second column contains the letter "B". The third column contains the letter "E". The fourth column contains the letter "E". The fifth column contains the letter "R". The sixth column contains the letter "B". The seventh column contains the letter "A". The eighth column contains the letter "R". The ninth column contains the letter "E". The tenth column contains the letter "B". The eleventh column is empty. Red asterisks (\*) are placed in the following cells: the fourth column of row 1, the fifth column of row 2, the fourth column of row 3, and the seventh column of row 3.

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*	A						
2		E	*						*			
3				R			*					

# FIFO

Memory page    B E E R B A R E B E A R

1	B			*	A				
2		E	*					*	
3				R			*		

# FIFO

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*	A						
2		E	*						*		B	
3				R				*				

# FIFO

The diagram illustrates a FIFO (First-In, First-Out) memory system. At the top, the word "BEER BEAR" is written across three memory pages. Below this, a table shows the state of the memory pages:

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*	A						
2		E	*						*		B	
3				R				*				

A vertical arrow points downwards from the bottom of the third row to the bottom of the fourth row, indicating the direction of the write pointer. The letters in the table represent the characters of the words "BEER BEAR". Red asterisks (\*) are placed in specific cells to highlight certain states or errors.

# FIFO

The diagram illustrates a FIFO (First-In, First-Out) memory system. At the top, the word "BEER BEAR" is written in a sequence of alternating uppercase letters. A vertical arrow points downwards from the letter 'R' in "BEER" towards the memory grid, indicating the direction of data flow or the position of the write pointer.

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*	A						
2		E	*						*	B		
3				R				*			E	

# FIFO

The diagram illustrates a FIFO (First-In, First-Out) memory system. At the top, the word "BEER BAR BEAR" is displayed, with each character representing a page in memory. Below this, a grid shows the state of three memory pages (Page 1, Page 2, and Page 3). The characters in the grid are color-coded: blue for 'B', red for 'E', green for 'R', and purple for 'A'. Red asterisks (\*) indicate free or invalid memory slots.

Memory page	B	E	E	R	B	A	R	B	E	R	B	A	R
1	B				*	A							*
2		E	*						*	B			
3				R			*				E		

# FIFO

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*	A					*	
2		E	*						*	B		
3				R		*				E		

# FIFO

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*	A					*	R
2		E	*					*		B		
3			R			*				E		

# FIFO

- 7 page faults

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*	A				*	R	
2		E	*					*	B			
3				R			*			E		

# FIFO

- 4 compulsory cache misses

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	<i>B</i>				*	<i>A</i>				*	<i>R</i>	
2		<i>E</i>	*					*	<i>B</i>			
3				<i>R</i>			*			<i>E</i>		

# Compulsory Misses vs. Page Faults

- Compulsory misses
  - Can occur at various levels of a memory hierarchy
    - L1, L2, L3, main memory
- Page faults
  - Occur when a page is not in the main memory

**MIN**

1

# MIN

1

**MIN**

↓

**MIN**

↓

# MIN



Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*	A						
2		E	*									
3				R				*				

# MIN

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*	A						
2		E	*							*		
3				R			*					

# MIN

Memory page    B E E R B A R E B E A R

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*	A						
2		E	*						*			
3				R				*				



# MIN



Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*	A						
2		E	*						*			
3				R			*		B			

# MIN



Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*	A						
2		E	*						*		*	
3				R			*		B			

# MIN



Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*	A					*	
2		E	*						*	*		*
3				R			*		B			

# MIN

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*	A				*	R	
2		E	*						*	*		
3				R			*		B			

# MIN

- 6 page faults

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*	A				*	R	
2		E	*					*	*	*		
3				R			*		B			

# LRU

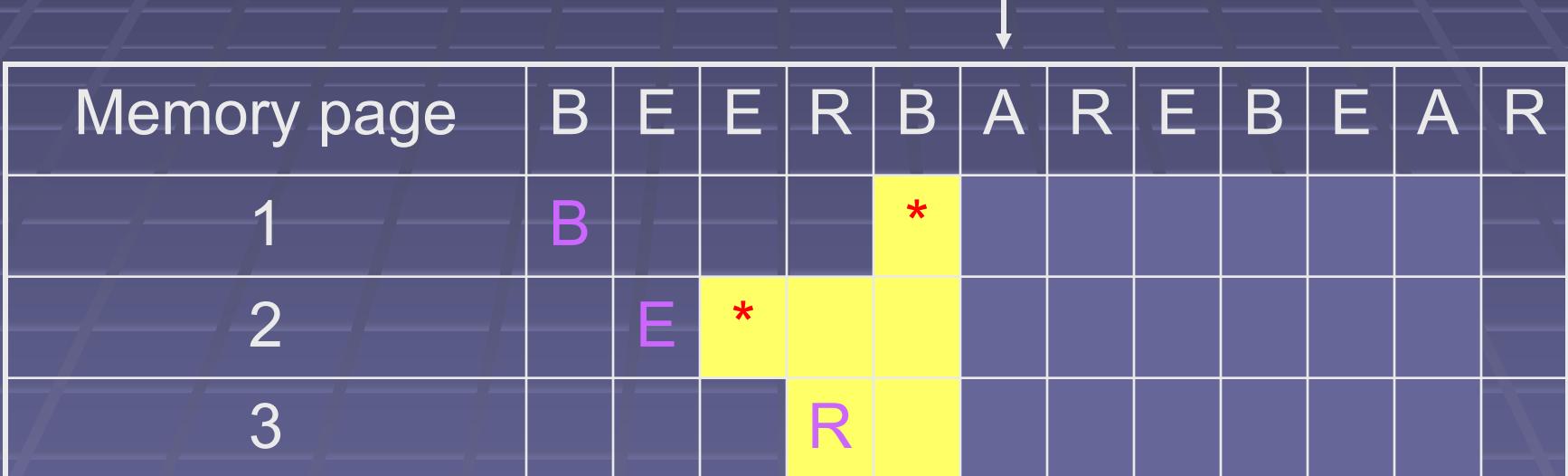
1

# LRU



Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B											*
2		E		*								
3					R							

# LRU



# LRU

↓

Memory page	B	E	E	R	B	A	R	E	B	E	A
1	B				*						
2		E	*					A			
3					R						

# LRU

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*							
2		E	*			A						
3				R				*				

# LRU

Memory page    B E E R B A R E B E A R

1	B				*				
2		E	*			A			
3			R				*		

# LRU

The diagram illustrates the LRU page replacement algorithm. At the top, the word "BEER BAR BEAR" is displayed, with an arrow pointing down to a 3x10 grid below. The grid represents memory pages, with rows labeled 1, 2, and 3 from top to bottom, and columns labeled by the characters in the word.

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*					E		
2		E	*			A						
3				R				*				

# LRU

The diagram illustrates the LRU page replacement algorithm in a memory system with 3 pages and 3 frames. The sequence of pages is BEERBAREREBEAR.

The table shows the state of the memory after each page is processed:

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*						E	
2		E	*			A						
3				R				*				

Annotations:

- An arrow points down to the fourth row, indicating the current page being considered.
- Red asterisks (\*) mark the least recently used pages in each frame.
- Yellow boxes highlight the pages in the frames.

# LRU

The diagram illustrates the LRU page replacement algorithm through a memory grid and a sequence of characters.

**Memory page**    B E E R B A R E B E A R

The memory grid has 3 rows (1, 2, 3) and 10 columns. The sequence of characters above the grid is: B E E R B A R E B E A R.

**Row 1:** Contains B, E, E, R, B, A, R, E, B, E. An arrow points down to the last E.

**Row 2:** Contains B, E, \*, A, \*, B. Red asterisks are placed at the positions of E and A.

**Row 3:** Contains R, \*, \*.

**Yellow Boxes:** Yellow boxes highlight specific cells: Row 1, Column 8 (E); Row 2, Columns 7 and 8 (A and B); Row 3, Columns 7 and 8 (both marked with red asterisks).

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*				E			
2		E	*			A				B		
3			R				*					

# LRU

The diagram illustrates the LRU page replacement algorithm. It shows a sequence of memory pages: BEER BAR E BEAR. A vertical arrow points down to the third page, indicating the current page being accessed.

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*			E		*		
2		E	*			A			B			
3				R			*					

# LRU

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*			E		*		
2		E	*			A			B			
3				R				*				

# LRU

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*			E		*		
2		E	*			A			B			
3				R			*					A

# LRU

The diagram illustrates the LRU page replacement algorithm. It shows a memory page with the string "BEER BAR E BEAR" and a 3-page frame system. The frames are indexed 1, 2, and 3. The table below tracks the state of each frame over time.

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*			E		*		
2		E	*			A			B			
3				R			*				A	

# LRU

The diagram illustrates the LRU page replacement algorithm. It shows a memory page with the string "BEER BAR E BEAR" and a 3-page frame system. The frames are indexed 1, 2, and 3. The page is divided into 12 slots, each corresponding to a character in the string. Red asterisks (\*) indicate which characters are currently in the frames. A yellow arrow points to the bottom-right corner of the third frame, where the letter 'A' is located.

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*			E		*		
2		E	*			A			B			R
3				R			*				A	

# LRU

- 8 page faults

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				*			E		*		
2		E	*			A			B		R	
3				R			*			A		

# LFU

4

# LFU

↓

# LFU

↓

# LFU



Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B											
2		E	2									
3					R							

# LFU

1

# LFU

The diagram illustrates the LFU page replacement algorithm. At the top, the word "LFU" is displayed in large white letters. Below it, a downward-pointing arrow indicates the direction of page replacement. The memory grid consists of 10 columns labeled "B E E R B A R E B E A R". The page table has three rows labeled 1, 2, and 3, corresponding to the memory pages. Row 1 contains the value "B" in the first column. Row 2 contains the values "E" and "2" in the second and third columns respectively. Row 3 contains the values "R" and "A" in the fourth and fifth columns respectively.

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B											
2		E	2									
3				R		A						

# LFU

The diagram illustrates the LFU page replacement algorithm. At the top, the string "BEER BAR BEAR" is displayed, with an arrow pointing down to a table below. The table has three rows labeled "Memory page" and "1", "2", "3" respectively. The columns represent memory pages, with the first column being the page number and the subsequent columns being the page content. The content is arranged as follows:

Memory page	B	E	E	R	B	A	R	B	E	A	R
1	B							2			
2		E	2								
3				R	A	R					

# LFU

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B											2
2		E	2									3
3				R	A	R						

# LFU

The diagram illustrates the LFU page replacement algorithm. It shows a memory page with the string "BEER BAR BEAR" and three pages in memory with their access counts:

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				2				3			
2		E	2						3			
3				R	A	R						

An arrow points to the fourth column of the third row, indicating the page to be replaced because it has the highest frequency count of 3.

# LFU

The diagram illustrates the LFU page replacement algorithm. It shows a memory page with the string "BEER BAR BEAR" and three pages in memory. The first page contains "B" (access count 1), the second page contains "E 2" (access count 2), and the third page contains "R A R" (access count 3). An arrow points down to the fourth page, which is currently empty.

Memory page	B	E	E	R	B	A	R	B	A	E	R
1	B				2				3		
2		E	2					3		4	
3				R	A	R					

# LFU

The diagram illustrates the LFU page replacement algorithm. It shows a sequence of memory pages: BEER BAR E BEAR. A vertical arrow points downwards from the word "LFU" towards the page 3 row. The memory pages are organized into three rows, labeled 1, 2, and 3 from top to bottom. Each row contains a sequence of characters representing memory pages. Red numbers indicate the frequency count of each page. Row 1 contains B (count 1), E (count 2), R (count 2), B (count 3). Row 2 contains E (count 2), R (count 2), A (count 3), R (count 4). Row 3 contains R (count 1), A (count 1), R (count 1), A (count 1).

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				2				3			
2		E	2					3		4		
3				R	A	R					A	

# LFU

The diagram illustrates the LFU page replacement algorithm. It shows a sequence of memory pages: BEER BAR E BEAR. Below this, a grid represents memory pages 1, 2, and 3. Each page has a row of four cells. The first cell contains the page number (1, 2, or 3). The second cell contains the page content (B, E, or R). The third cell contains the access count (2 or 3). The fourth cell contains the page number again (3 or 4). A red arrow points to the bottom right corner of the grid.

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				2				3			
2		E	2					3		4		
3			R		A	R					A	R

# LFU

- 7 page faults

Memory page	B	E	E	R	B	A	R	E	B	E	A	R
1	B				2				3			
2		E	2					3		4		
3				R	A	R				A	R	

# Does adding RAM always reduce misses?

- Yes for LRU and MIN
  - Memory content of  $X$  pages  $\subseteq X + 1$  pages
- No for FIFO
  - Due to modulo math
  - ***Belady's anomaly:*** getting more page faults by increasing the memory size

# Belady's Anomaly

- 9 page faults

Memory page	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					*
2		B			A			*		C		
3			C			B			*		D	

# Belady's Anomaly

- 10 page faults

Memory page	A	B	C	D	A	B	E	A	B	C	D	E
1	A				*		E				D	
2		B				*		A				E
3			C						B			
4				D						C		

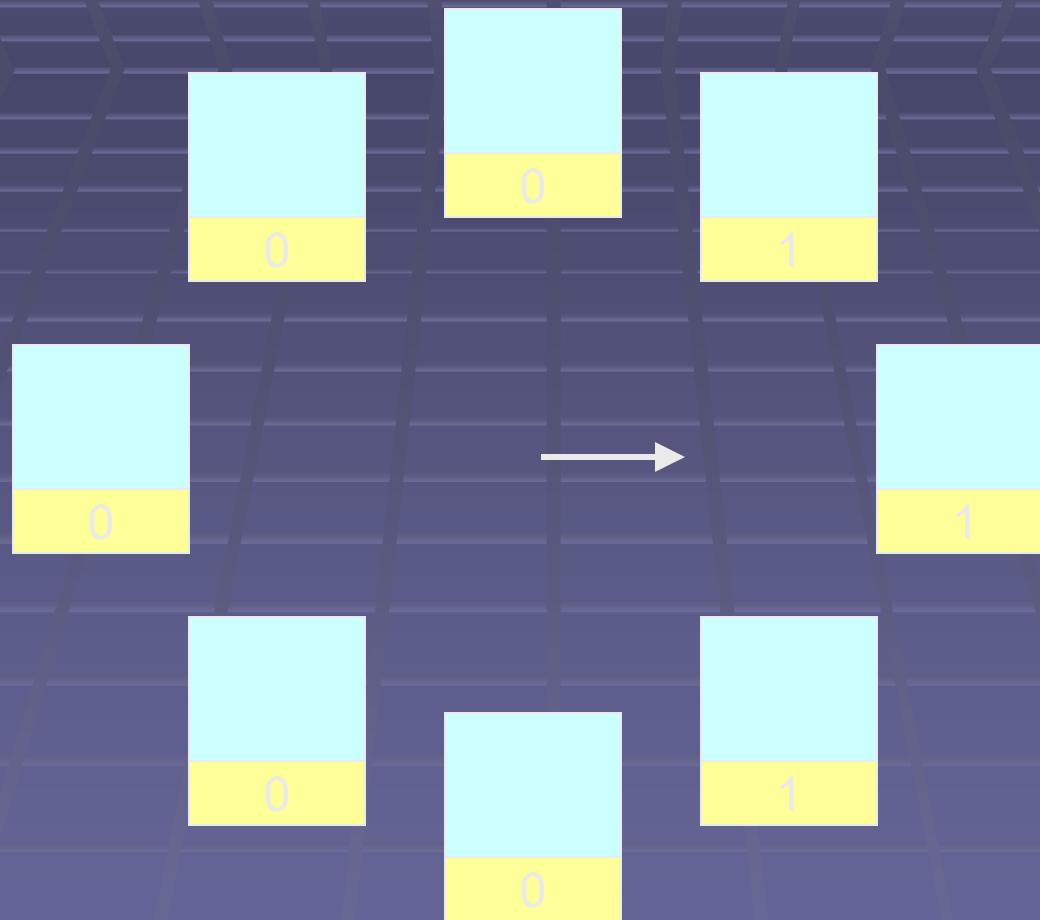
# Implementing LRU

- One way is to require a timestamp on each reference to a cache page
  - Too expensive
- An alternative is to use a stack
  - Whenever a page is referenced, move to the top
  - When needed, discard the bottom page
- Common practice
  - Approximate the LRU behavior

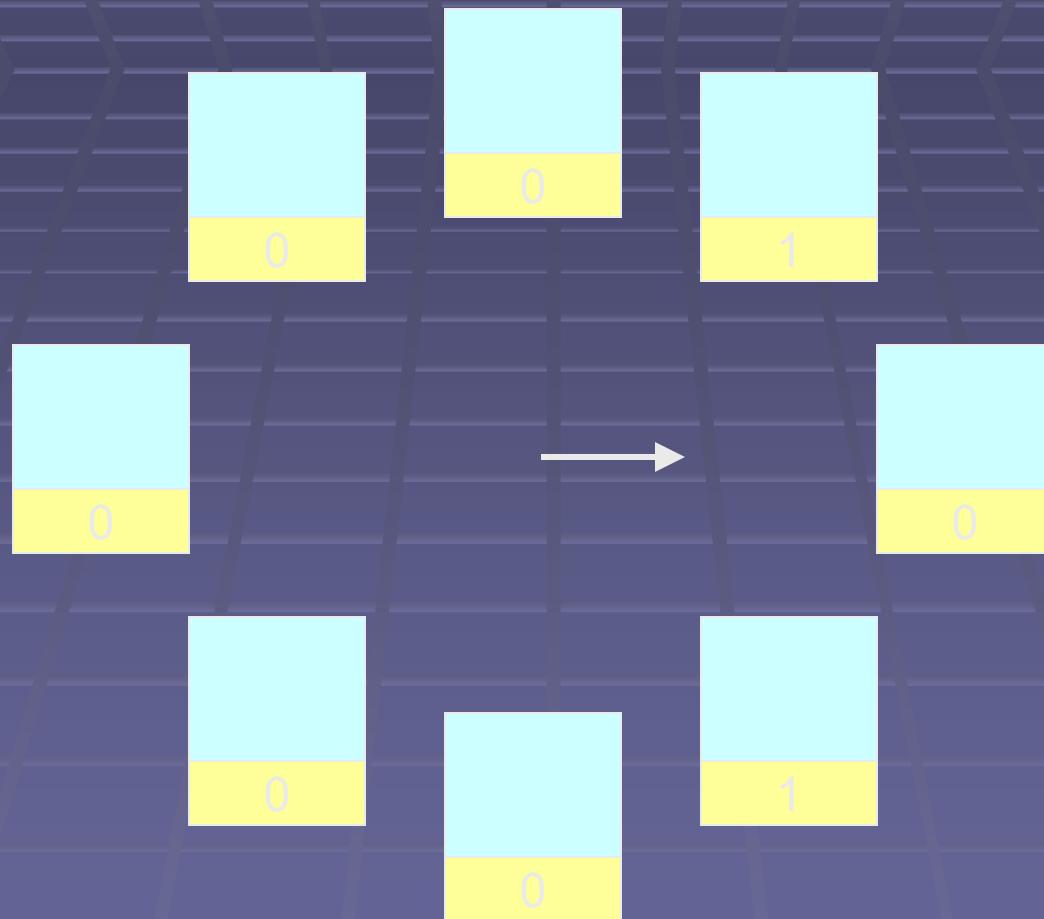
# *Clock Algorithm*

- Replaces an old page, but not the oldest page
- Arranges physical pages in a circle
  - With a clock hand
- Each page has a ***used bit***
  - Set to 1 on reference
  - On page fault, sweep the clock hand
    - If the used bit == 1, set it to 0
    - If the used bit == 0, pick the page for replacement

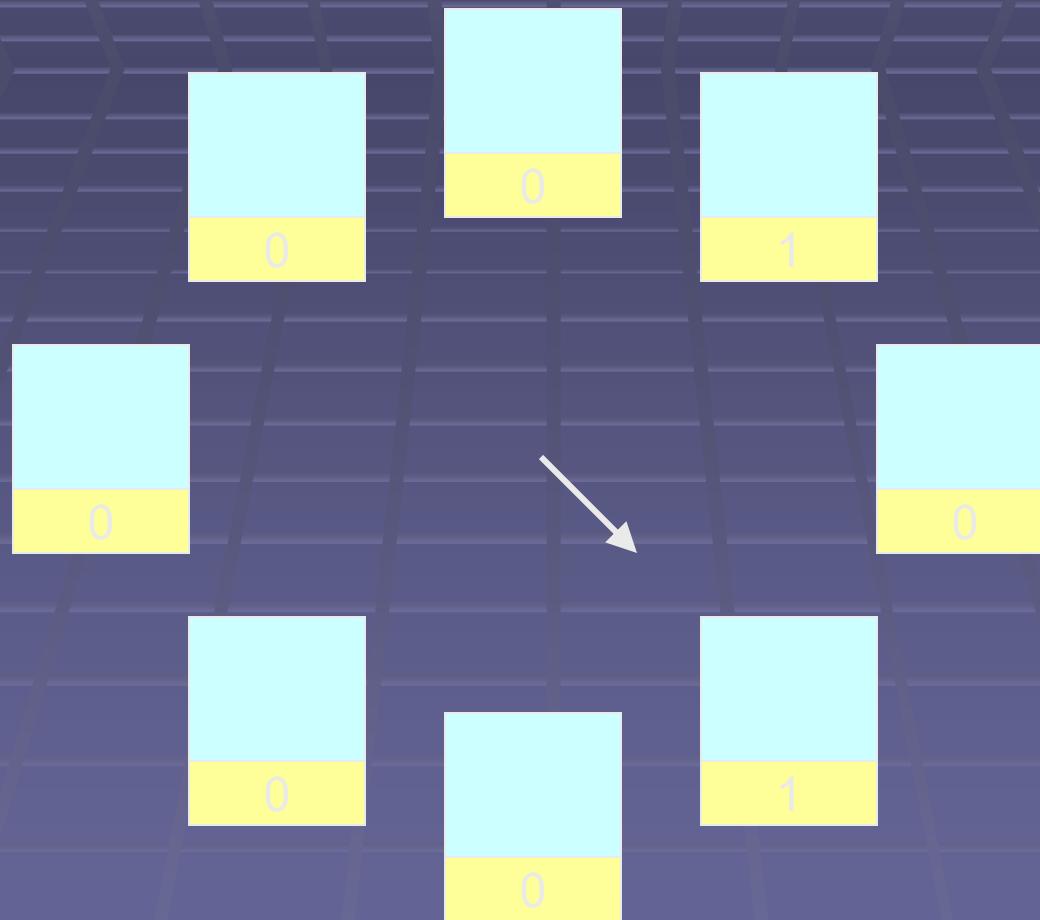
# Clock Algorithm



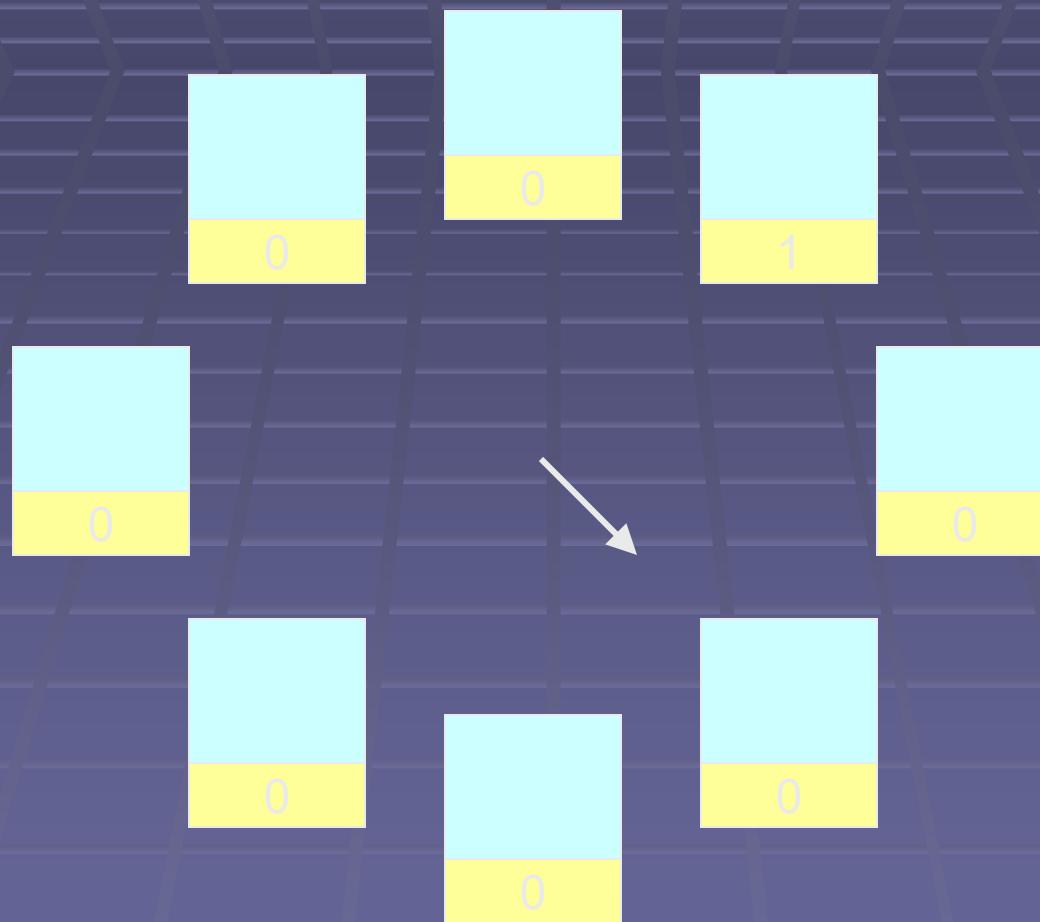
# Clock Algorithm



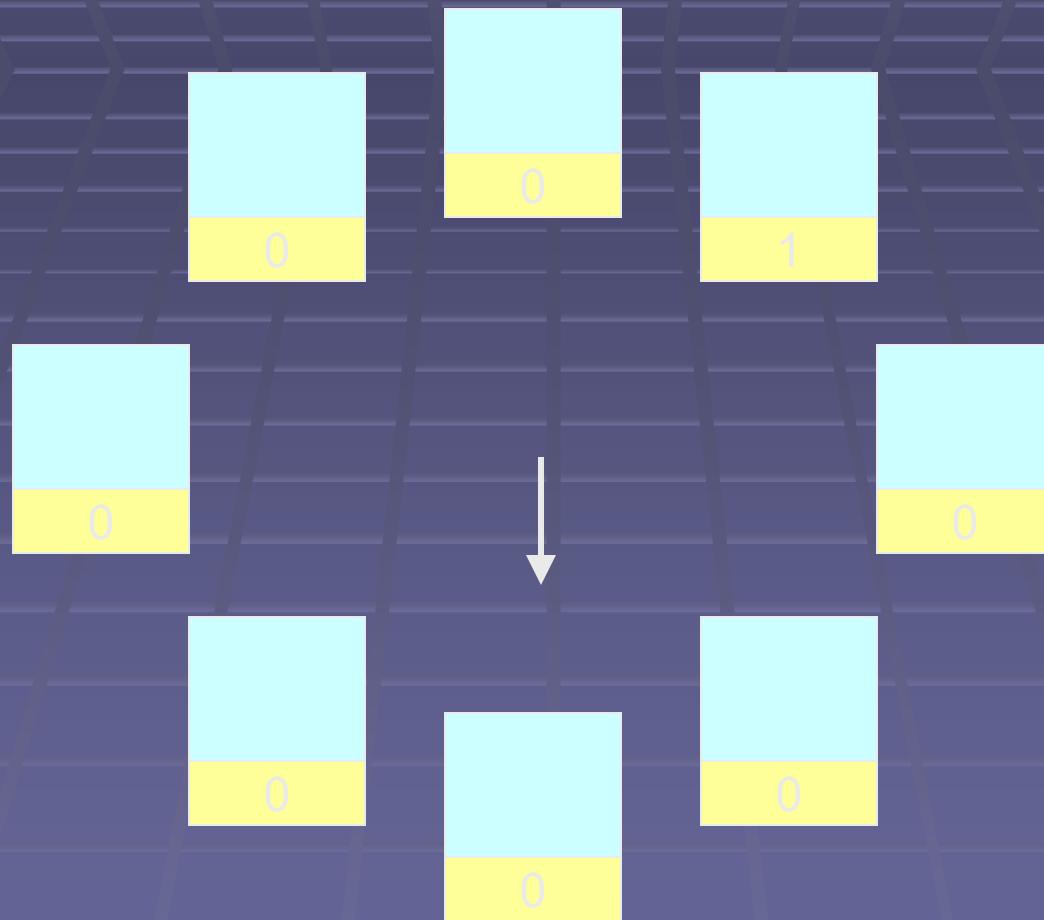
# Clock Algorithm



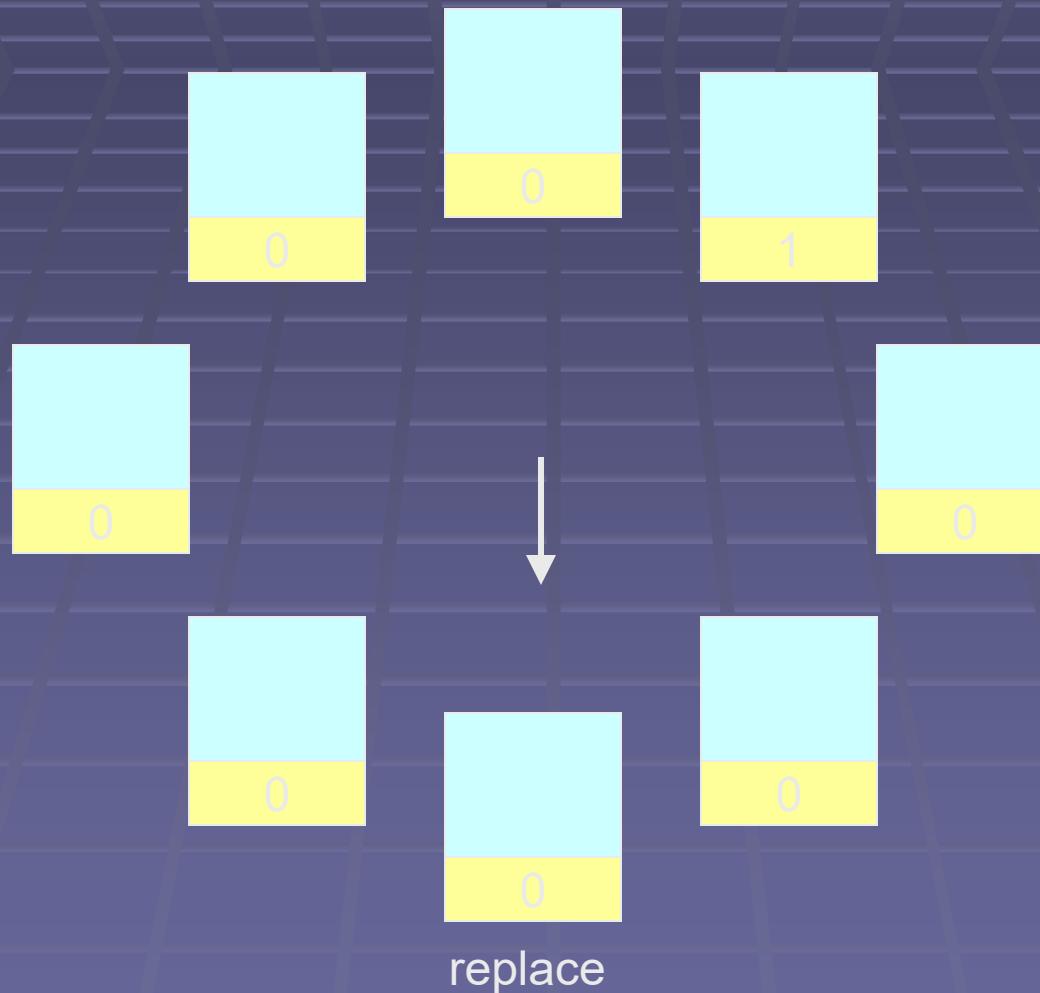
# Clock Algorithm



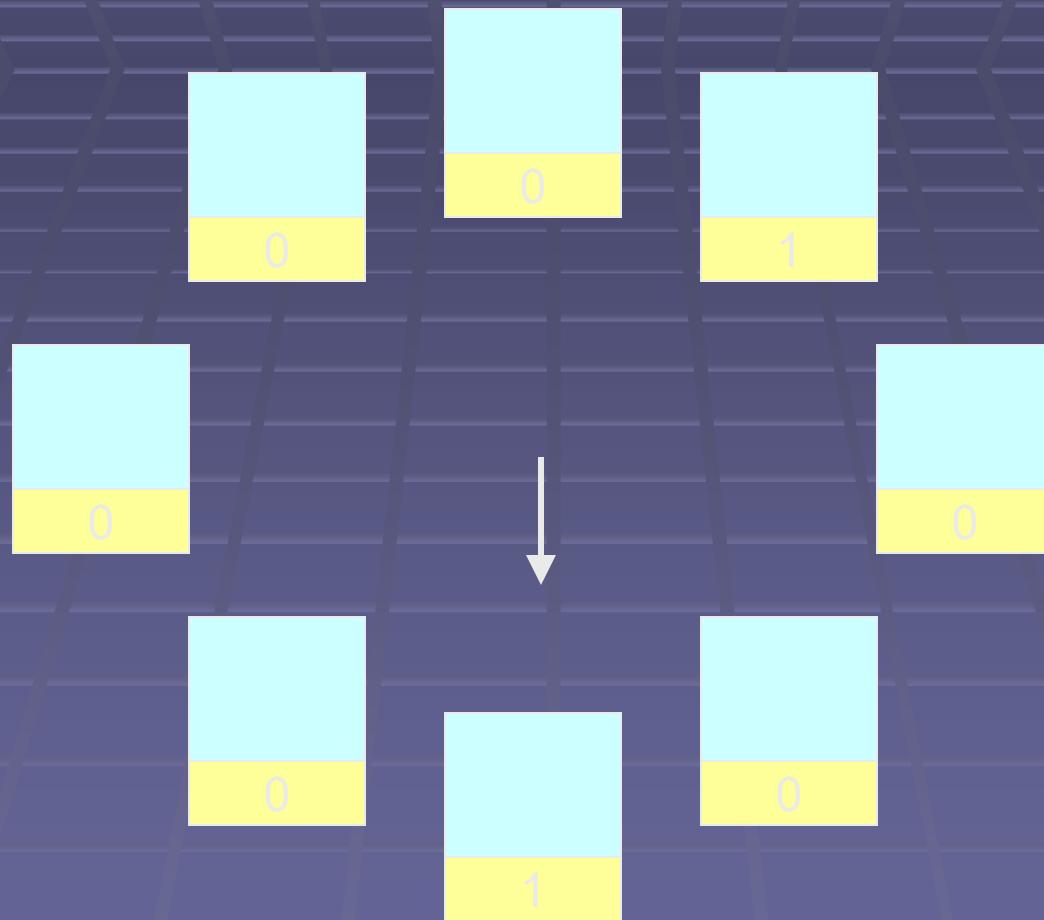
# Clock Algorithm



# Clock Algorithm



# Clock Algorithm



# Clock Algorithm

- The clock hand cannot sweep indefinitely
  - Each bit is eventually cleared
- Slow moving hand
  - Few page faults
- Quick moving hand
  - Many page faults

# *Nth Chance Algorithm*

- A variant of clocking algorithm
  - A page has to be swept N times before being replaced
  - $N \rightarrow \infty$ , Nth Chance Algorithm  $\rightarrow$  LRU
  - Common implementation
    - $N = 2$  for modified pages
    - $N = 1$  for unmodified pages

# States for a Page Table Entry

- **Used bit:** set when a page is referenced; cleared by the clock algorithm
- **Modified bit:** set when a page is modified; cleared when a page is written to disk
- **Valid bit:** set when a program can legitimately use this entry
- **Read-only:** set for a program to read the page, but not to modify it (e.g., code pages)

# *Thrashing*

- Occurs when the memory is overcommitted
  - Pages are still needed are tossed out
- Example
  - A process needs 50 memory pages
  - A machine has only 40 memory pages
  - Need to constantly move pages between memory and disk
- Another example
  - Two processes kick out each other's useful

# Thrashing Avoidance

- Programs should minimize the maximum memory requirement at a given time
  - e.g., matrix multiplications can be broken into sub-matrix multiplications
- OS figures out the memory needed for each process
  - Runs only the computations that can fit in RAM

# *Working Set*

- A set of pages that was referenced in the previous T seconds
  - $T \rightarrow \infty$ , working set  $\rightarrow$  size of the entire process
- Observation
  - Beyond a certain threshold, more memory only slightly reduces the number of page faults

# Working Set

- LRU, 3 memory pages, 12 page faults

Memory page	A	B	C	D	A	B	C	D	E	F	G	H
1	A			D			C			F		
2		B			A			D			G	
3			C			B			E			H

# Working Set

- LRU, 4 memory pages, 8 page faults

Memory page	A	B	C	D	A	B	C	D	E	F	G	H
1	A				*				E			
2		B				*				F		
3			C				*				G	
4				D				*				H

# Working Set

- LRU, 5 memory pages, 8 page faults

Memory page	A	B	C	D	A	B	C	D	E	F	G	H
1	A				*					F		
2		B				*					G	
3			C				*					H
4				D				*				
5									E			

# Global and Local Replacement Policies

- ***Global replacement policy:*** all pages are in a single pool (e.g., UNIX)
  - One process needs more memory
    - Grabs memory from another process that needs less
  - + Flexible
  - One process can drag down the entire system
- ***Per-process replacement policy:*** each process has its own pool of pages

# Linux Memory Manager (1)

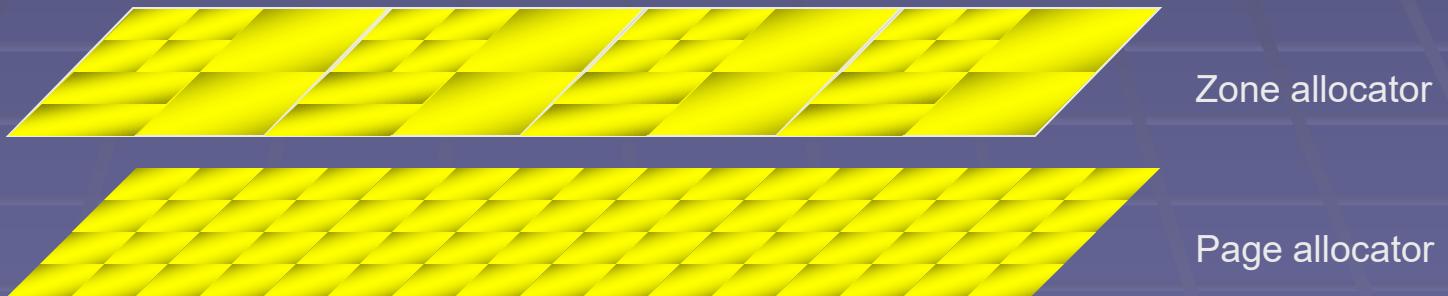
- Page allocator maintains individual pages



Page allocator

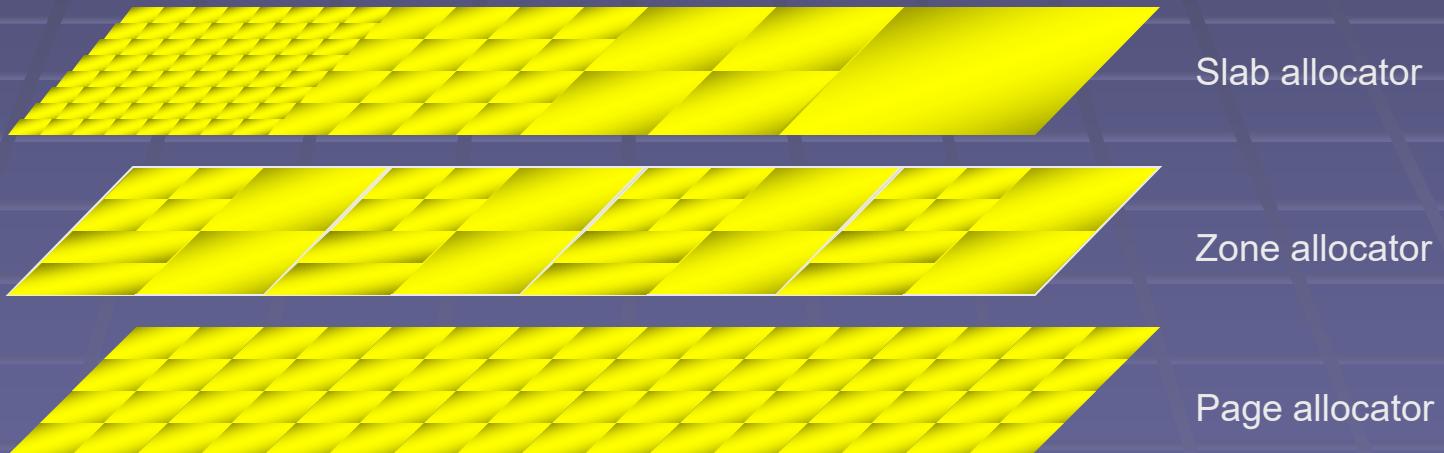
# Linux Memory Manager (2)

- Zone (buddy) allocator allocates memory in power-of-two sizes



# Linux Memory Manager (3)

- Slab allocator groups allocations by sizes to reduce internal memory fragmentation



# After Linux 2.6.24

- SLUB allocator replaced SLAB allocator
  - Moved metadata from SLAB to memory page data structure
  - SLUB does not maintain a per-CPU queue
  - Lower overhead