Memory Protection and Address Translation

Xin Liu

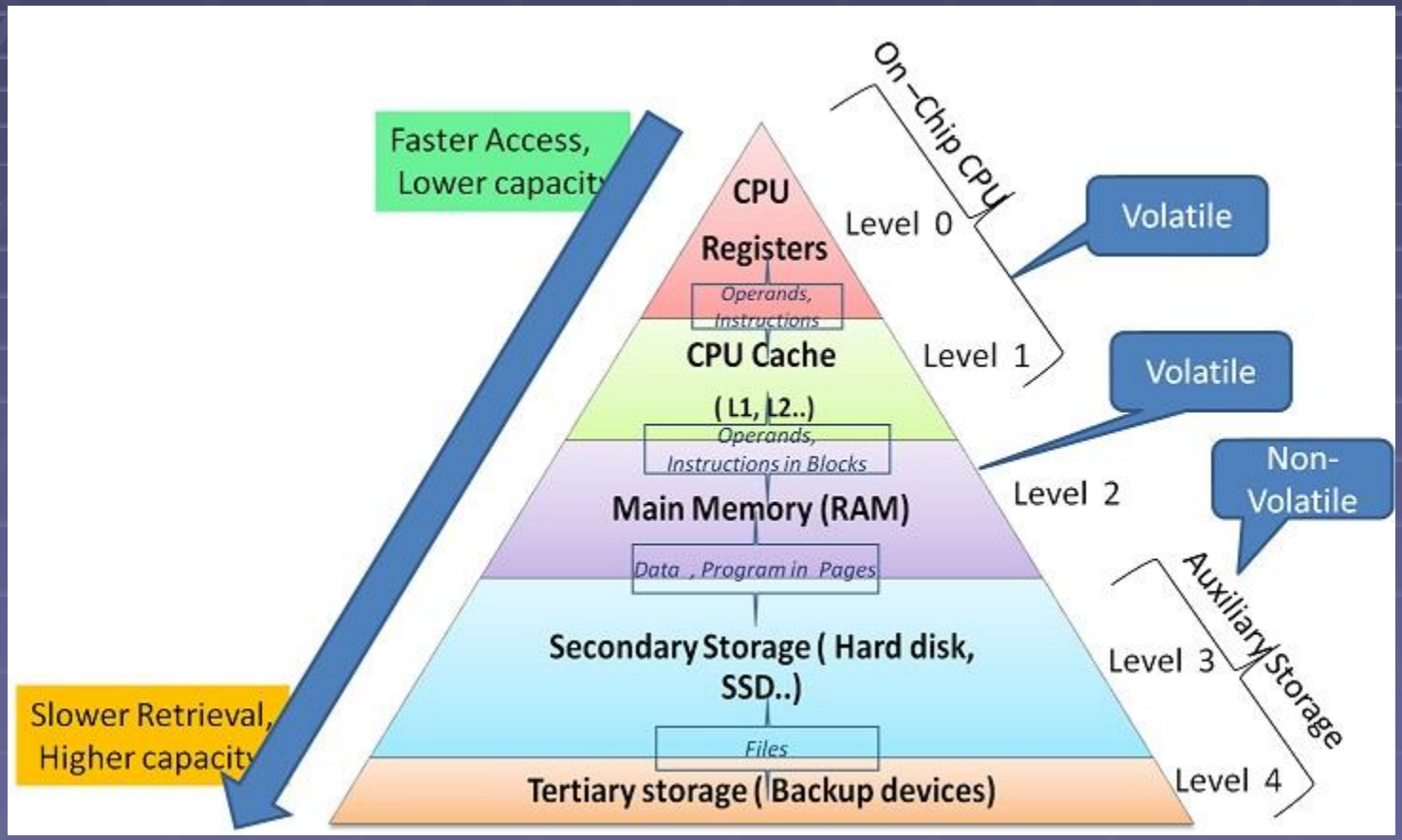
Operating Systems

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

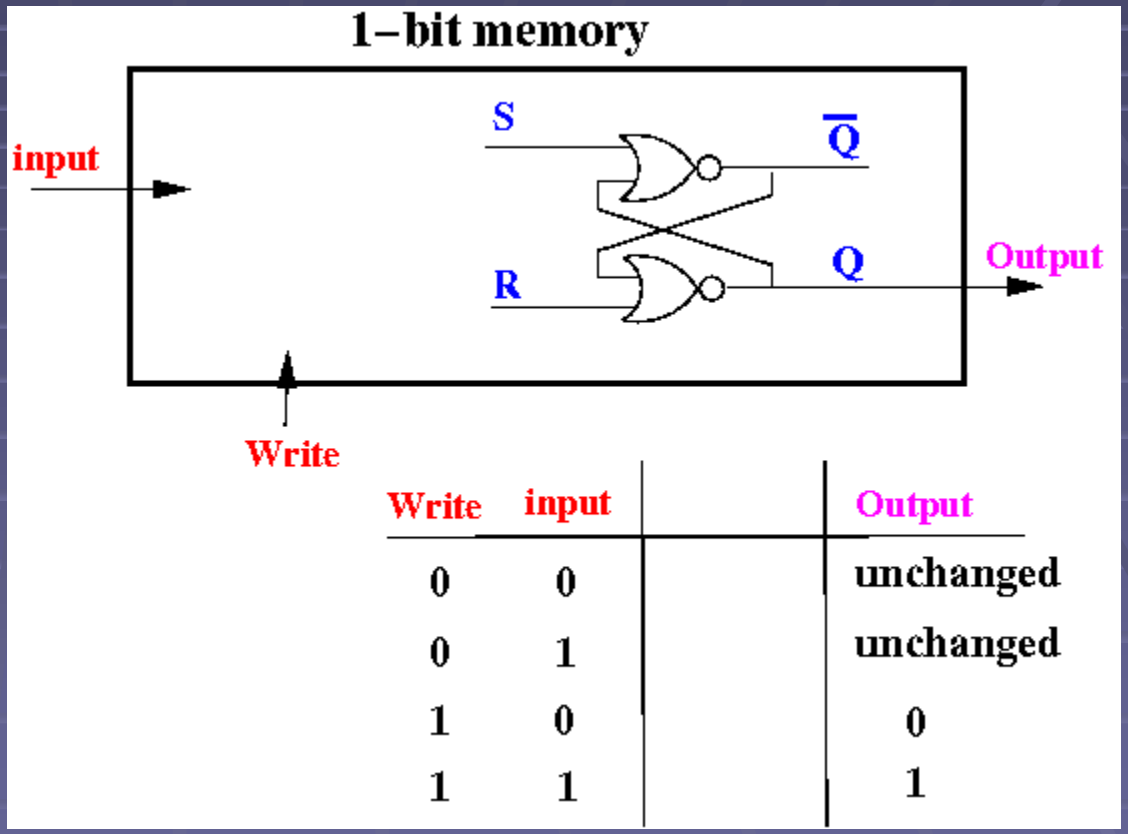
- Memory is byte-addressable
  - Each memory address can specify the location of a byte
- unsigned char memory[MEMORY_SIZE]
- To access
  - memory[virtual_memory_address]

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



Write	input	Output
0	0	unchanged
0	1	unchanged
1	0	0
1	1	1

Physical vs. Virtual Memory

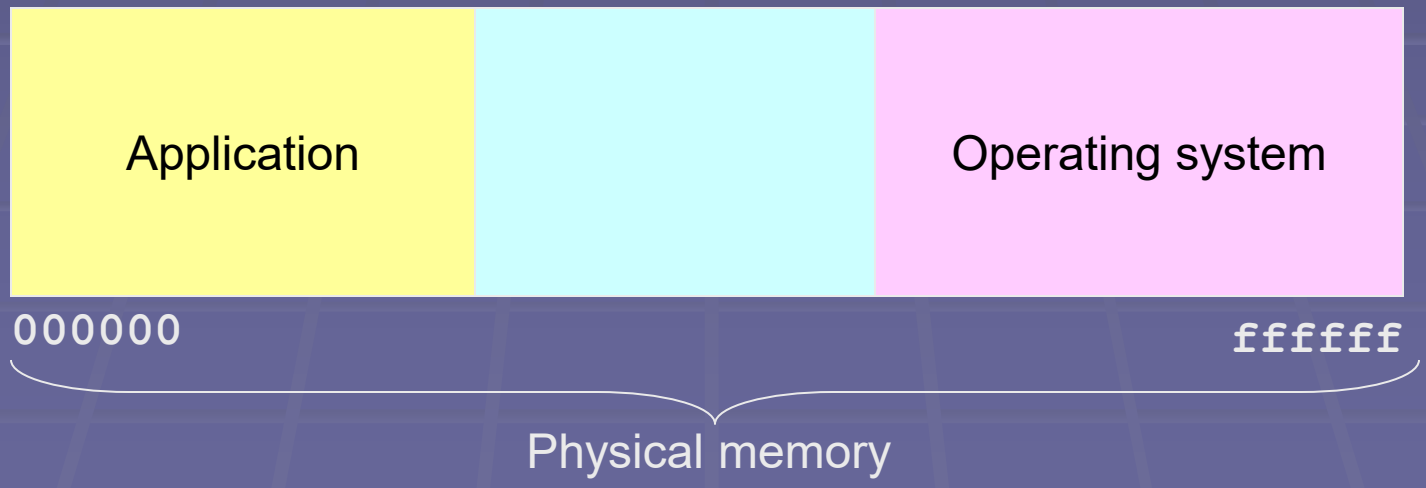
Physical memory	Virtual memory
No protection	Each process isolated from others and from OS
Limited size	Illusion of infinite memory
Sharing visible to processes	Each process cannot tell if memory is shared

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)

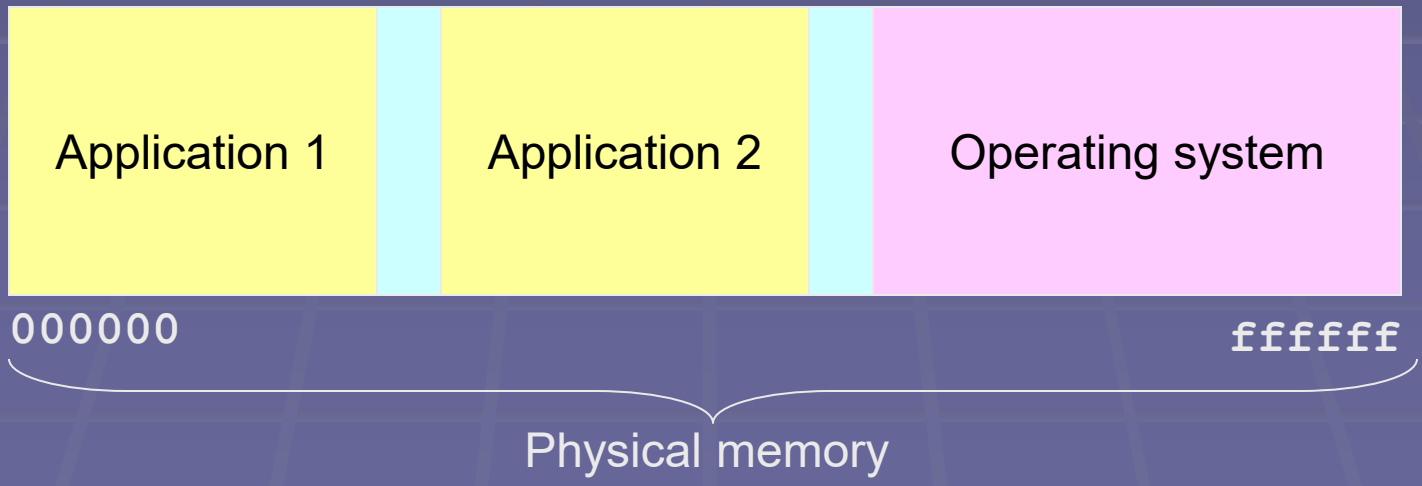


Multiprogramming Without Memory Protection

- 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



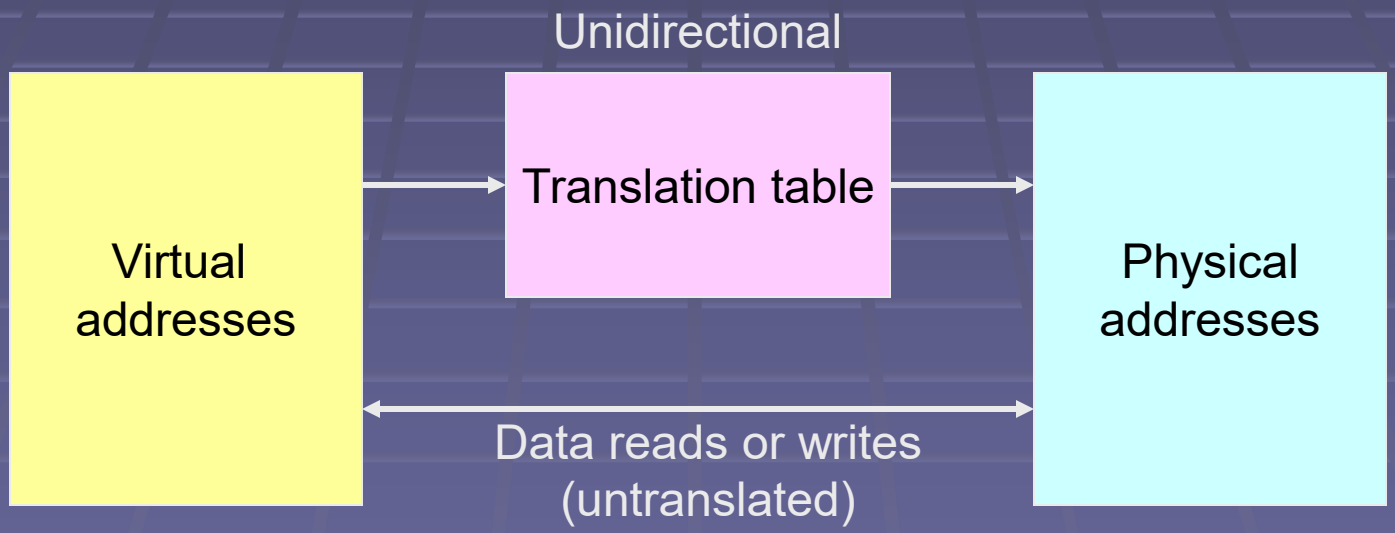
Multiprogrammed OS With Memory Protection

- ***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

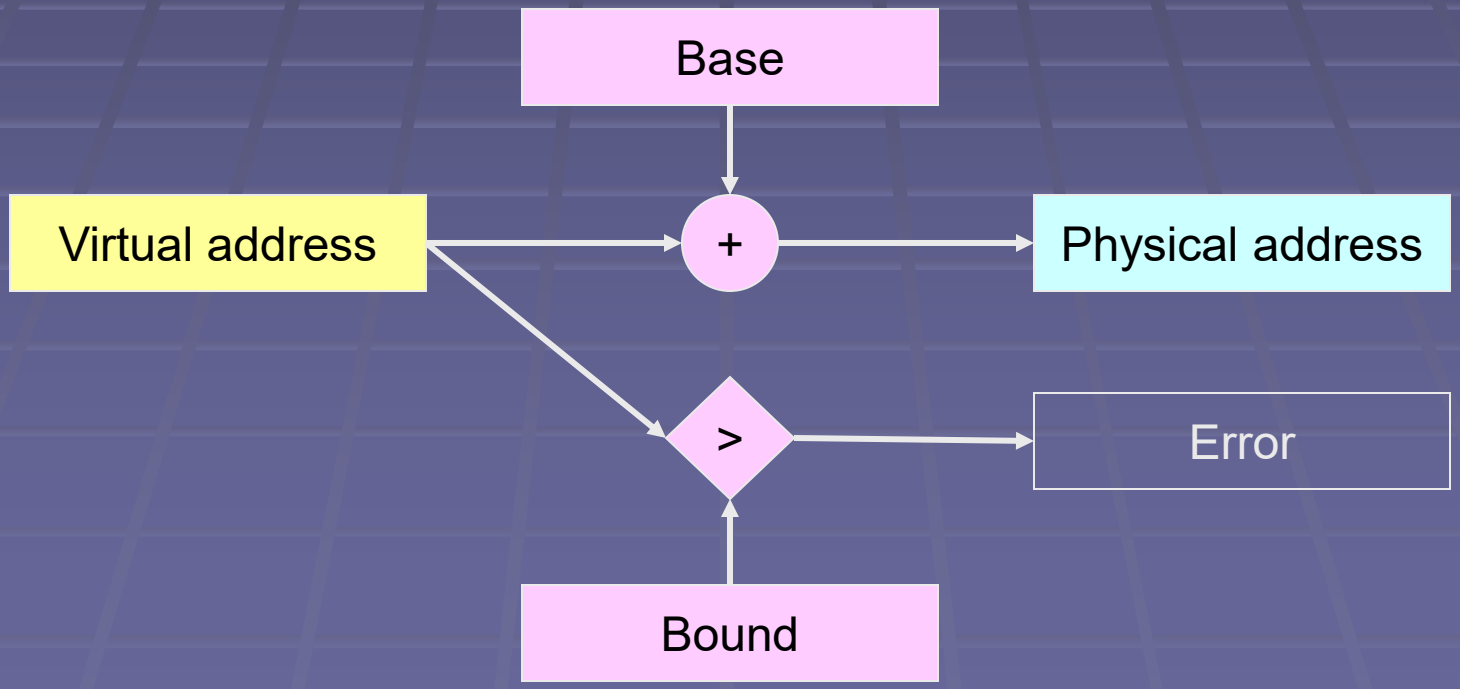
- Translation provides protection
  - Processes cannot talk about other processes' addresses, nor about the OS addresses
  - OS uses physical addresses directly
    - No translations

Assumptions

- 32-bit machines
- 1-GB RAM max

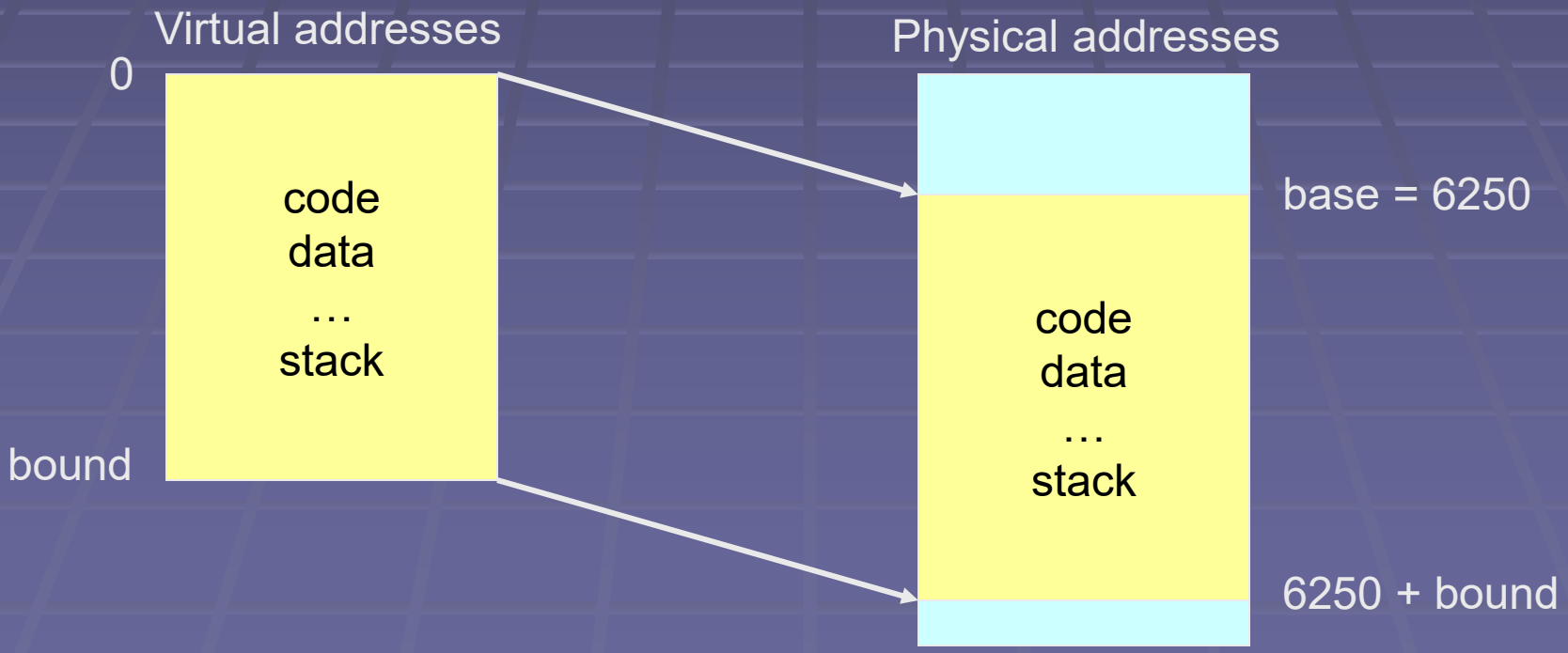
Base-and-Bound Translation

- 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

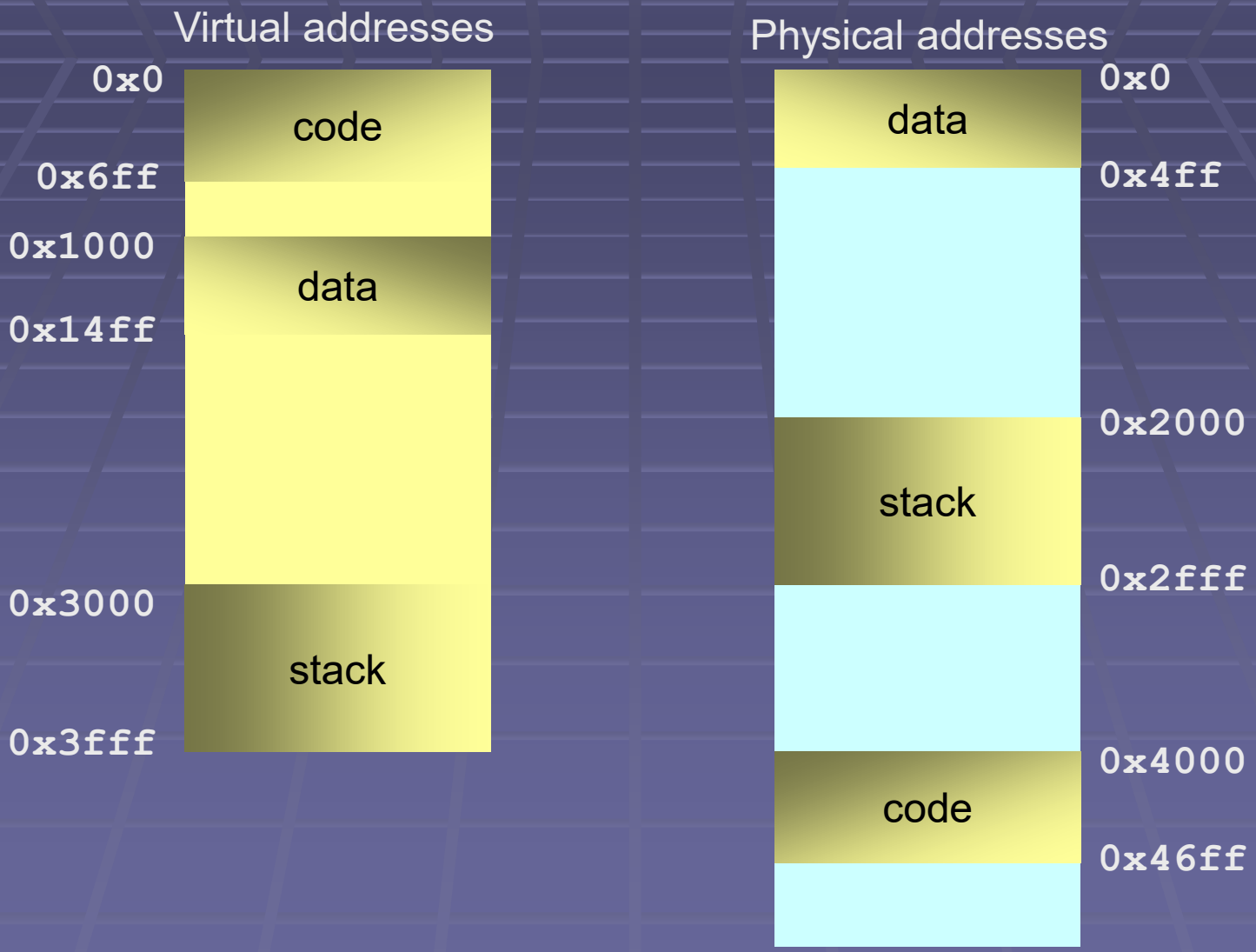
Pros/Cons of Base-and-Bound Translation

- 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

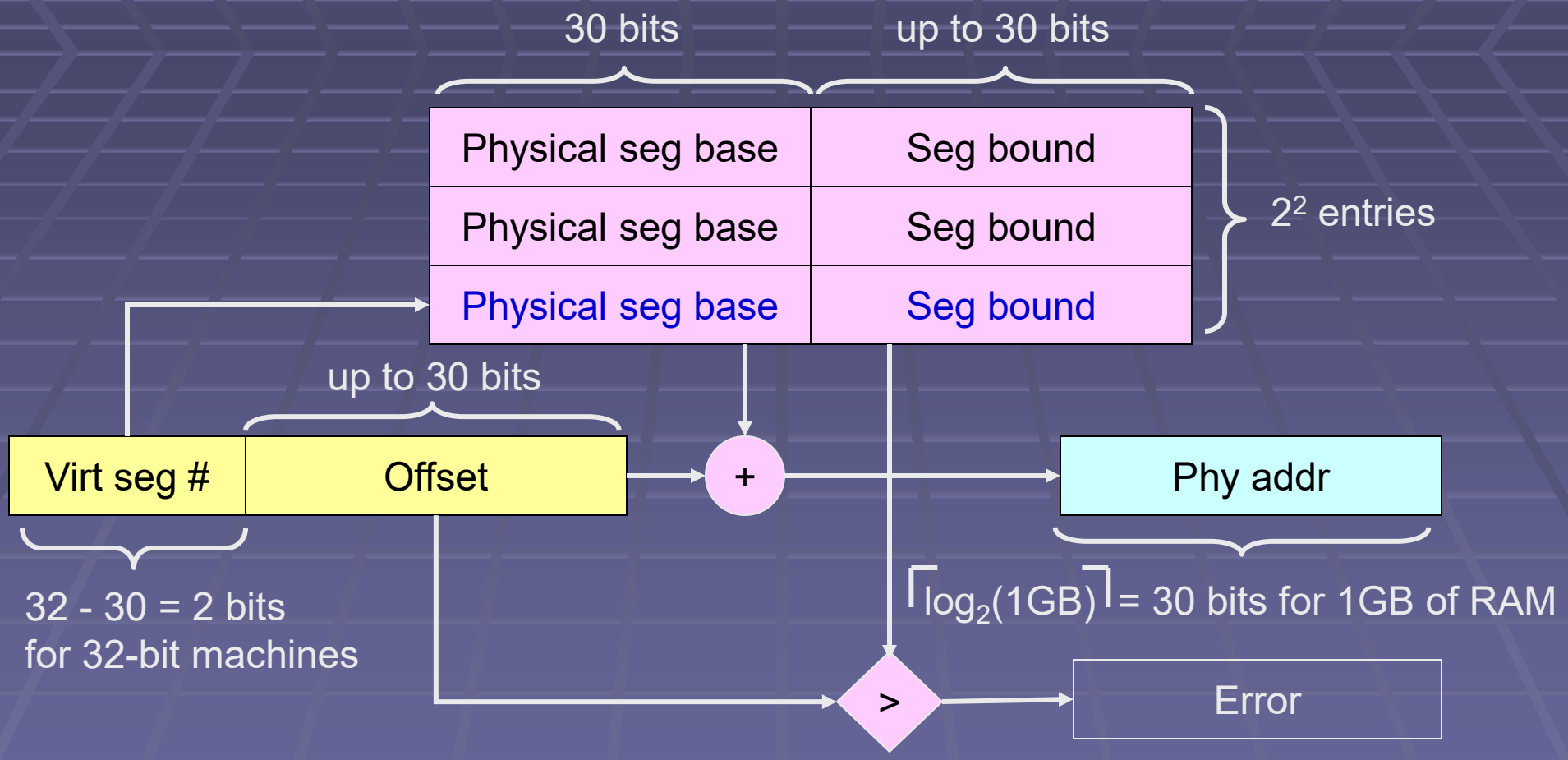
Segmentation

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

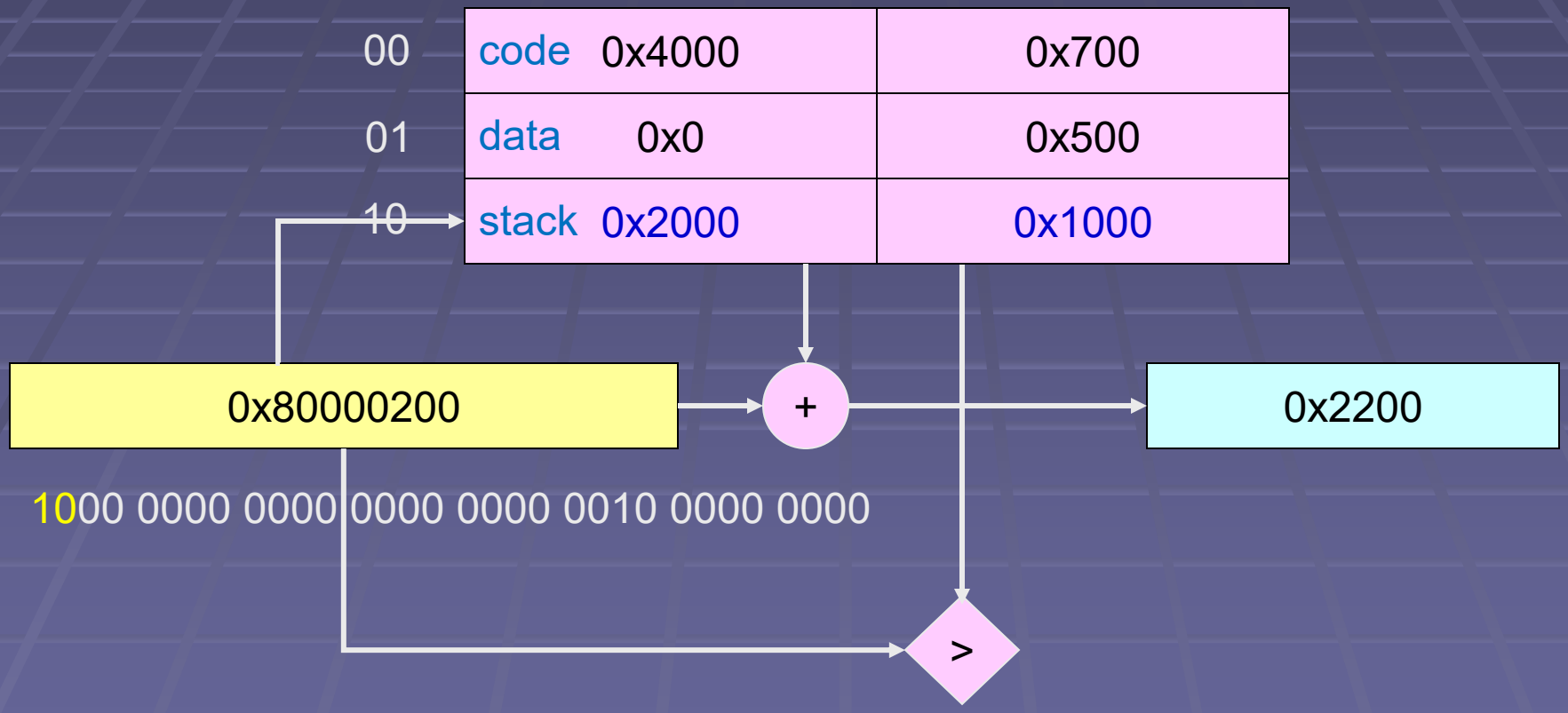
Segmentation Illustrated



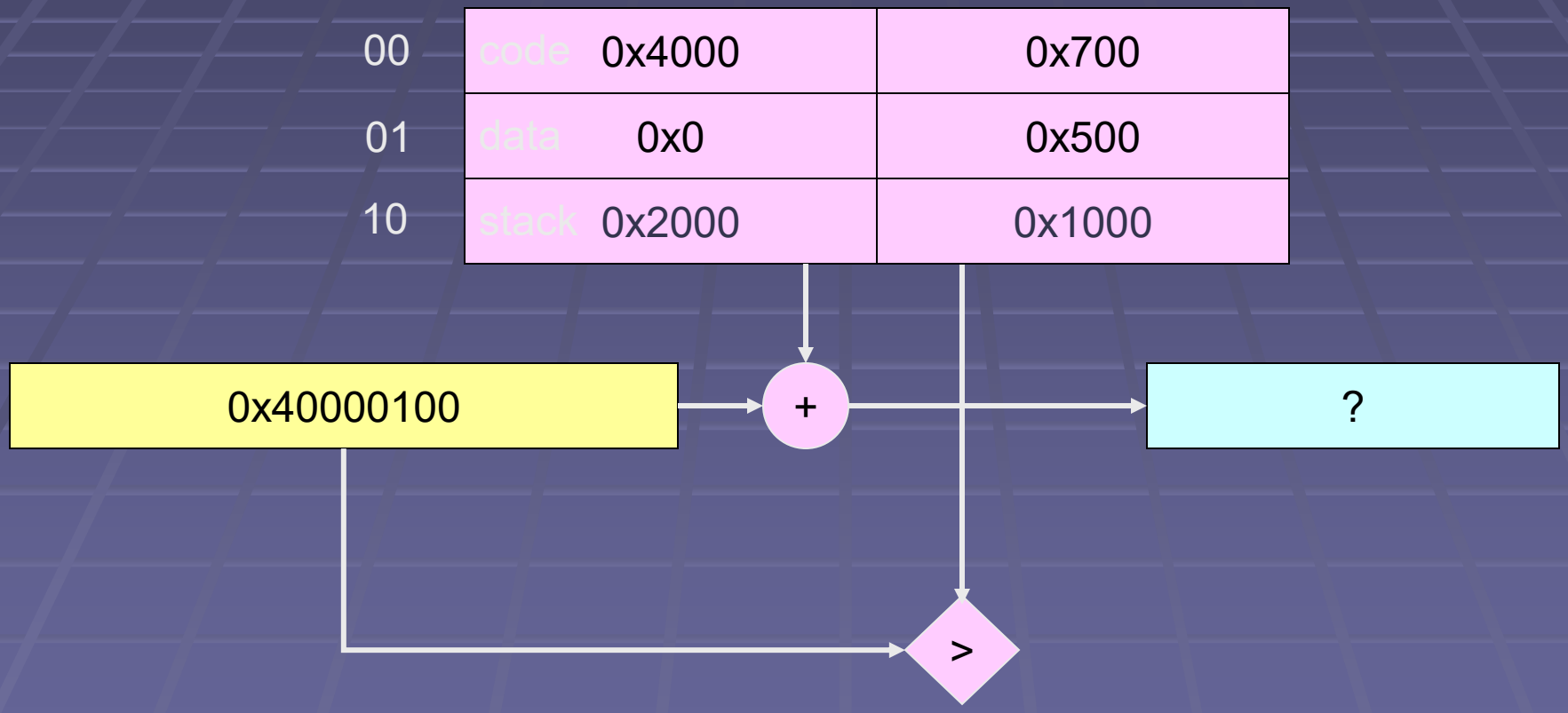
Segmentation Diagram



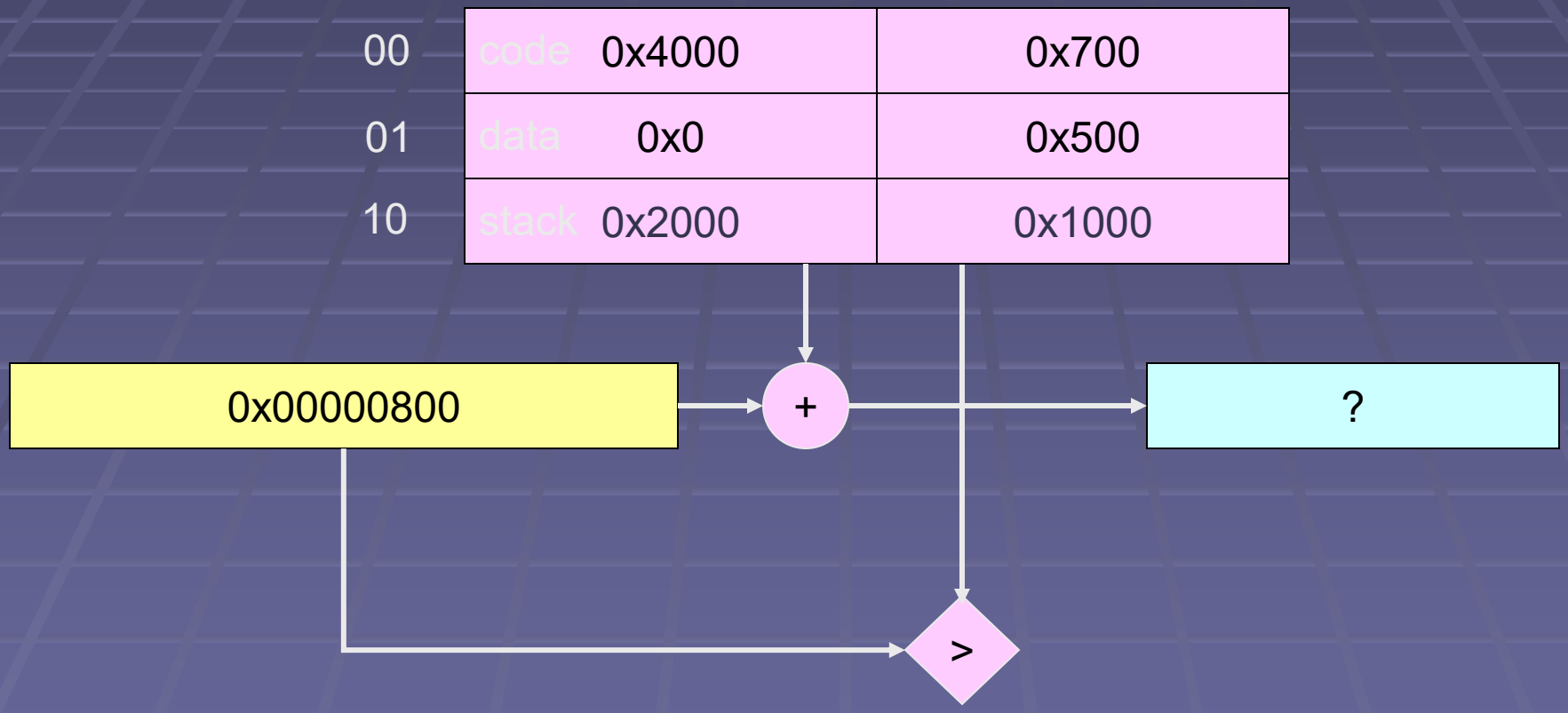
Segmentation Diagram



Segmentation Diagram



Segmentation Diagram



Segmentation Translation

- virtual_address = virtual_segment_number:offset
- physical_base_address = segment_table[virtual_segment_number]
- physical_address = physical_base_address + offset

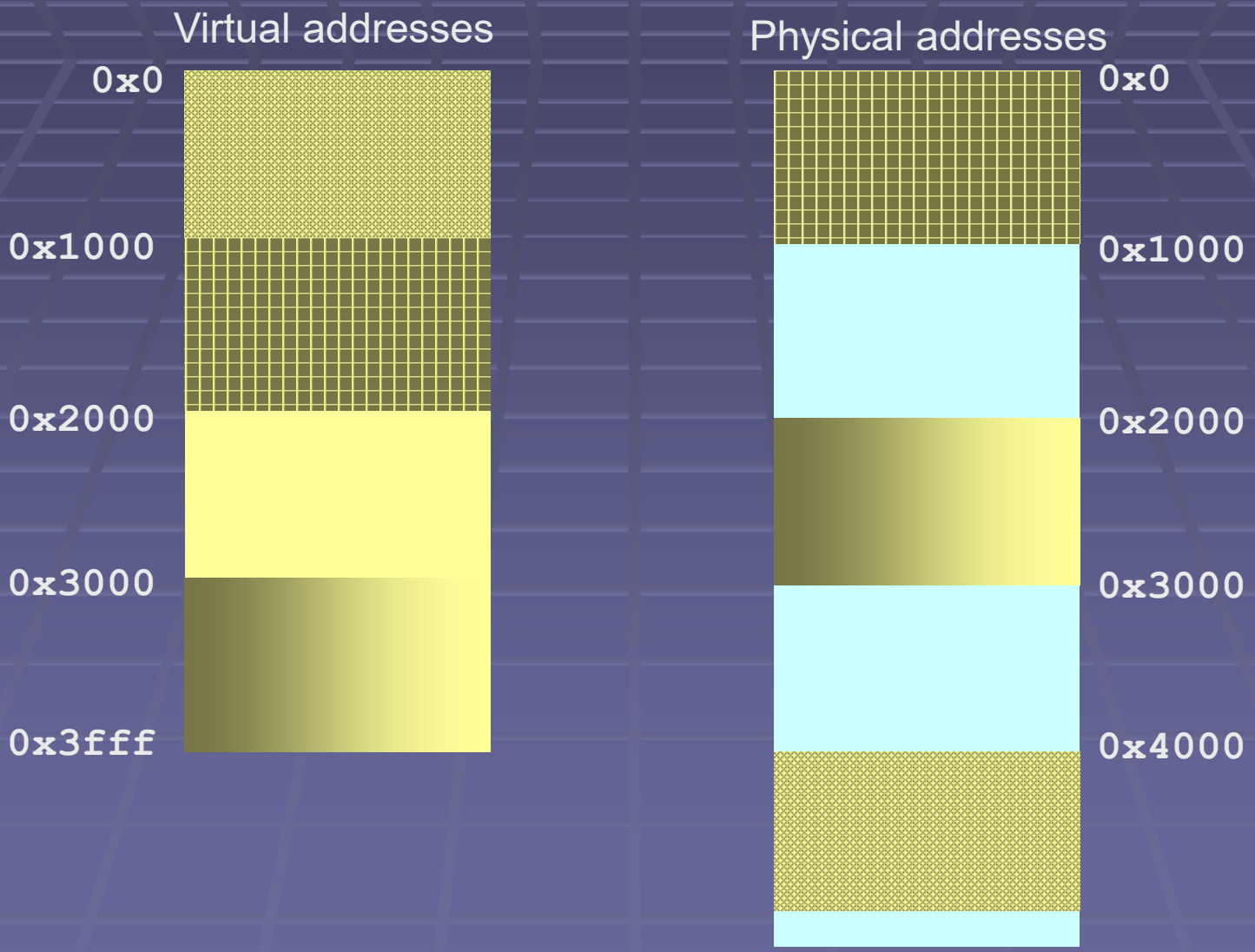
Pros/Cons of Segmentation

+ 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

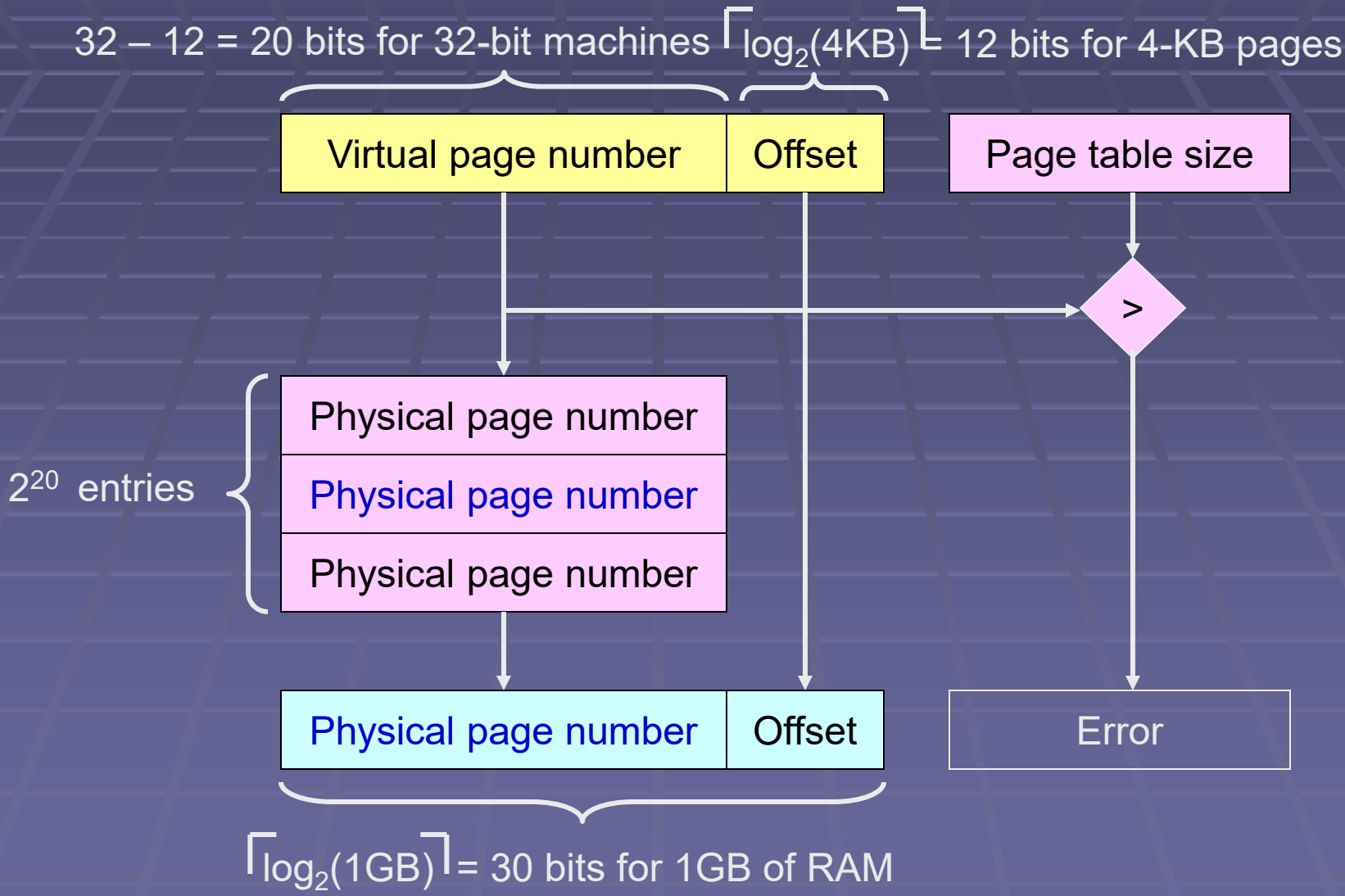


Paged Memory Acces

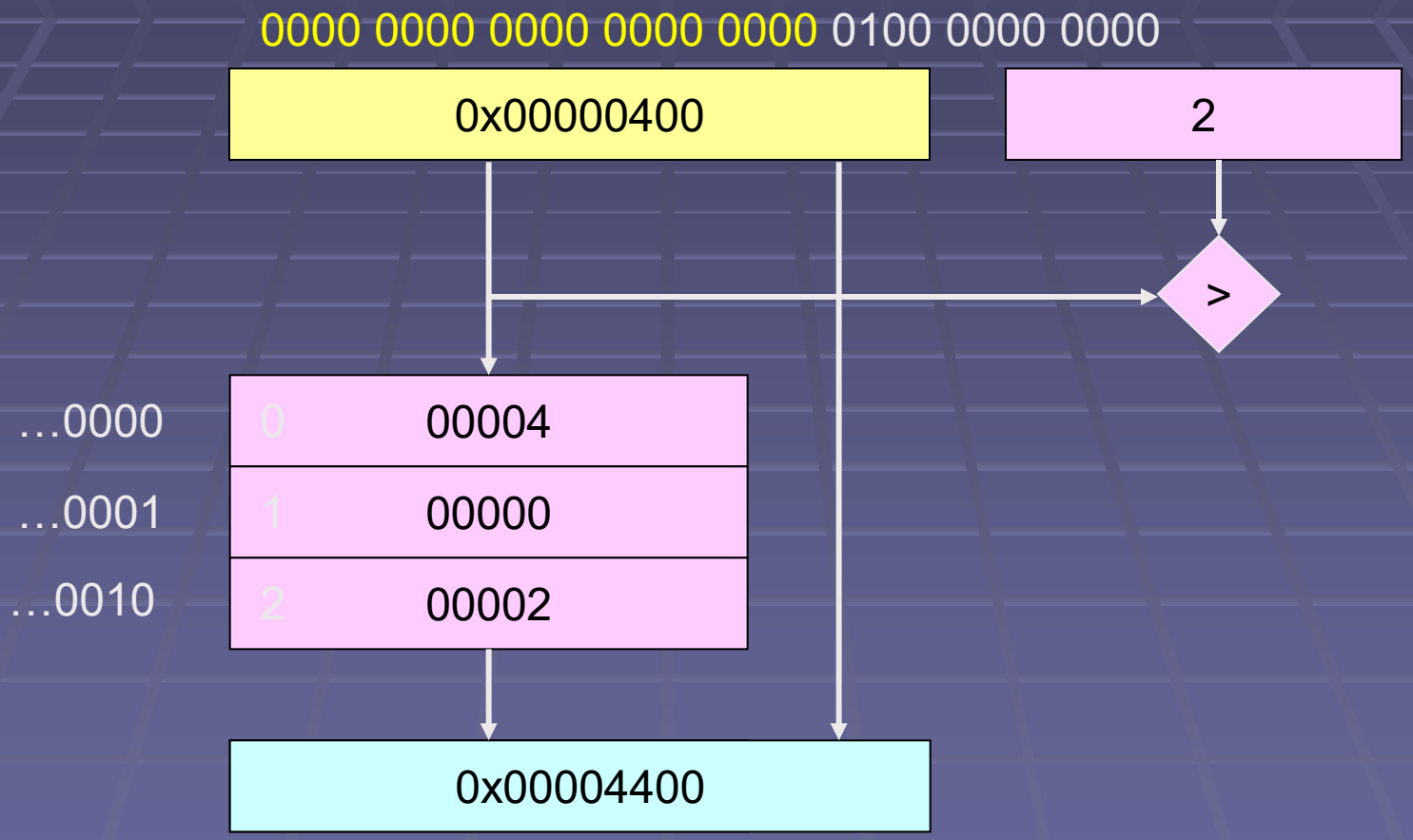
unsigned char memory[N_PAGES][PAGE_SIZE]

- To access
  - memory[virtual_page_number][page_offset]

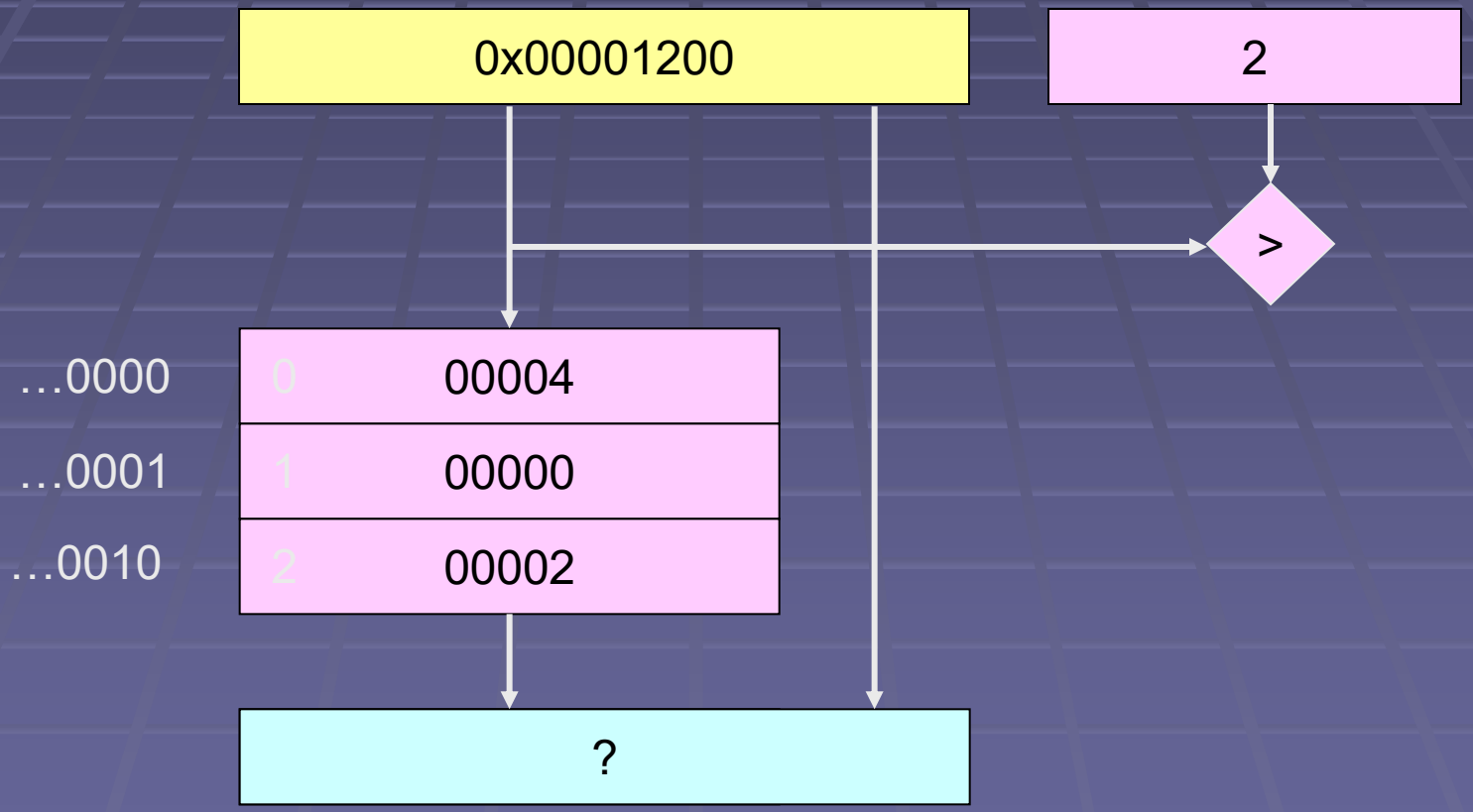
Paging Diagram



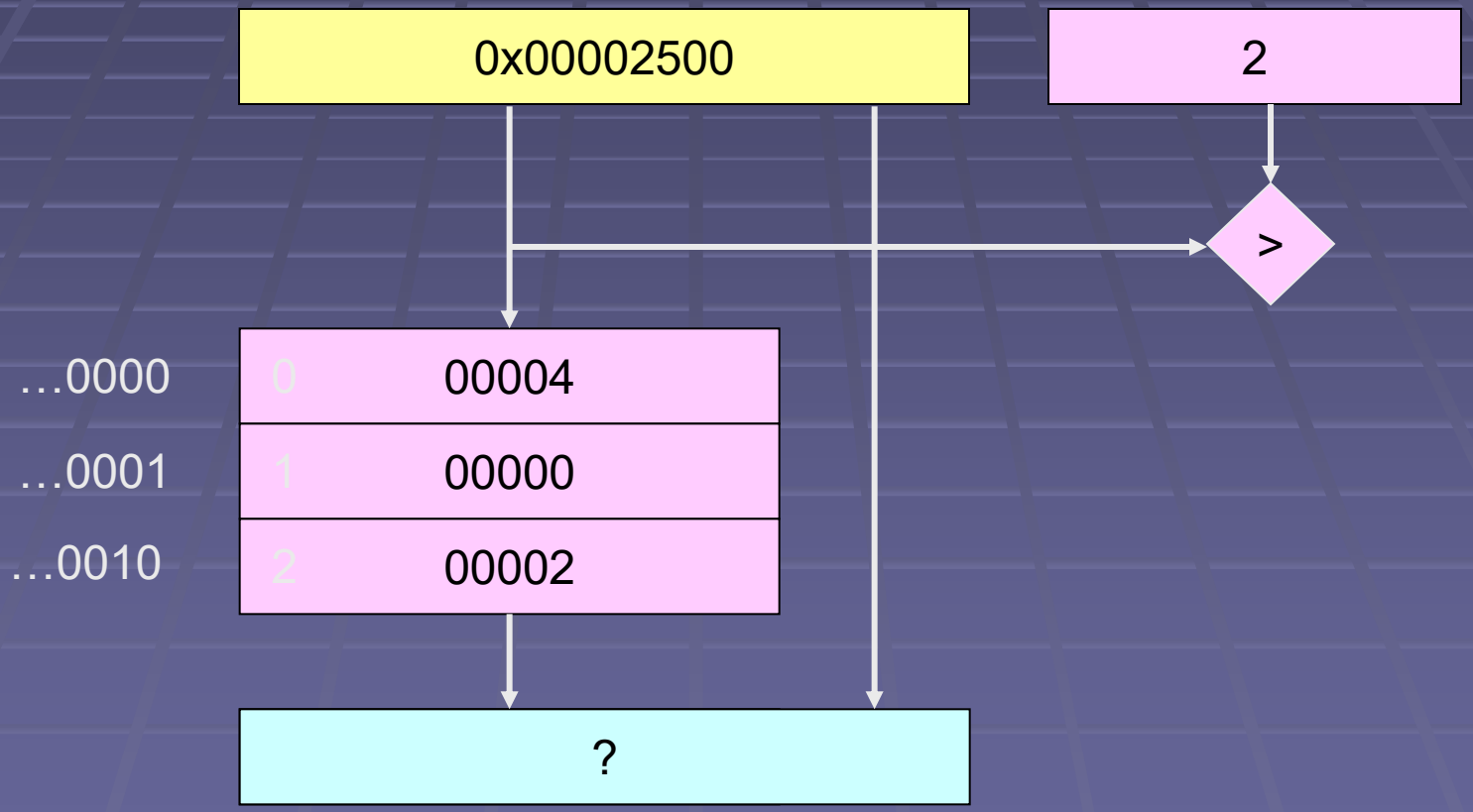
Paging Example



Paging Example



Paging Example



Paging Translation

- virtual_address = virtual_page_number:offset
- physical_page_number = page_table[virtual_page_number]
- physical_address = physical_page_number:offset

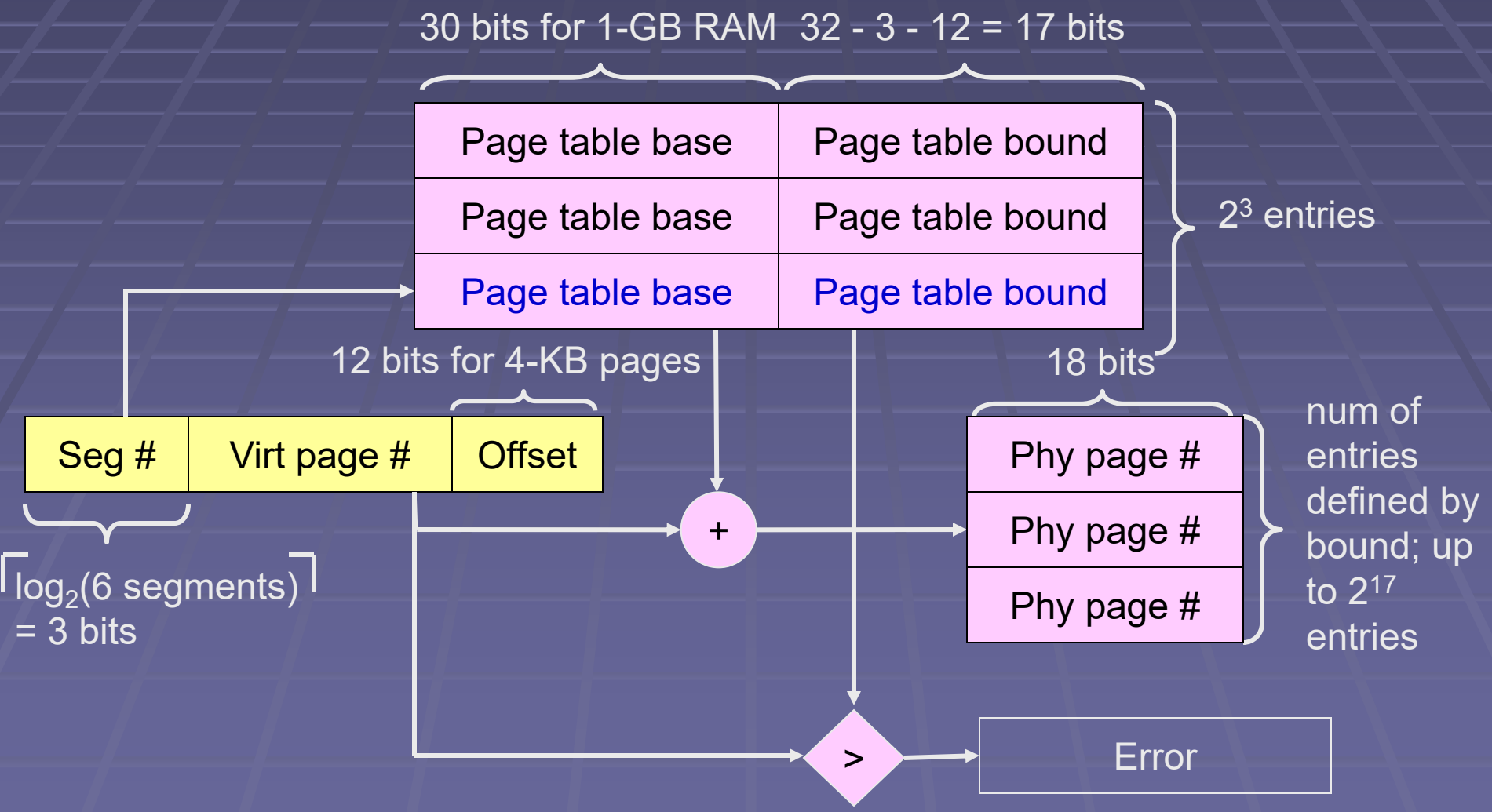
Pros and Cons of Paging

+ 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

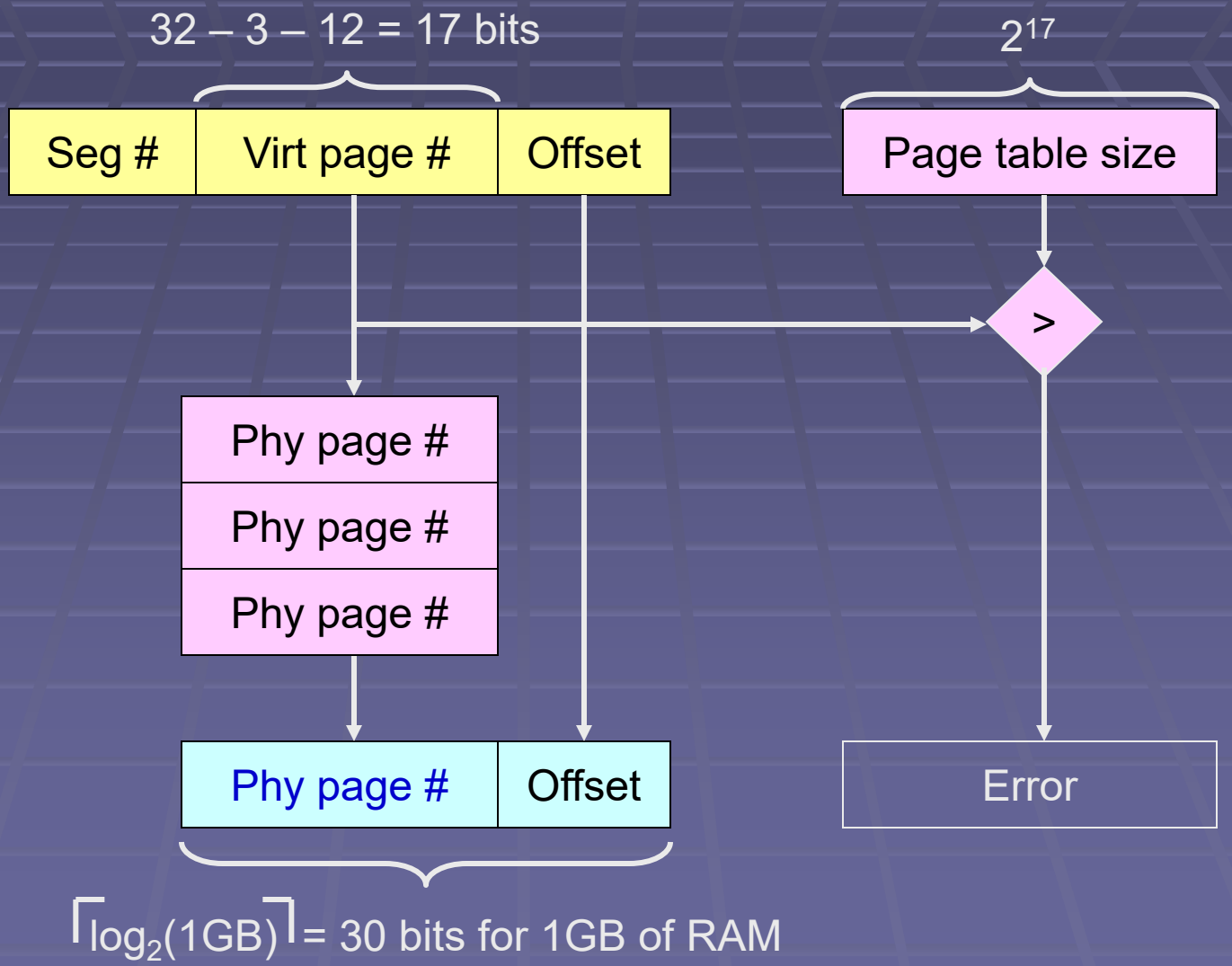
Multi-Level Translation

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

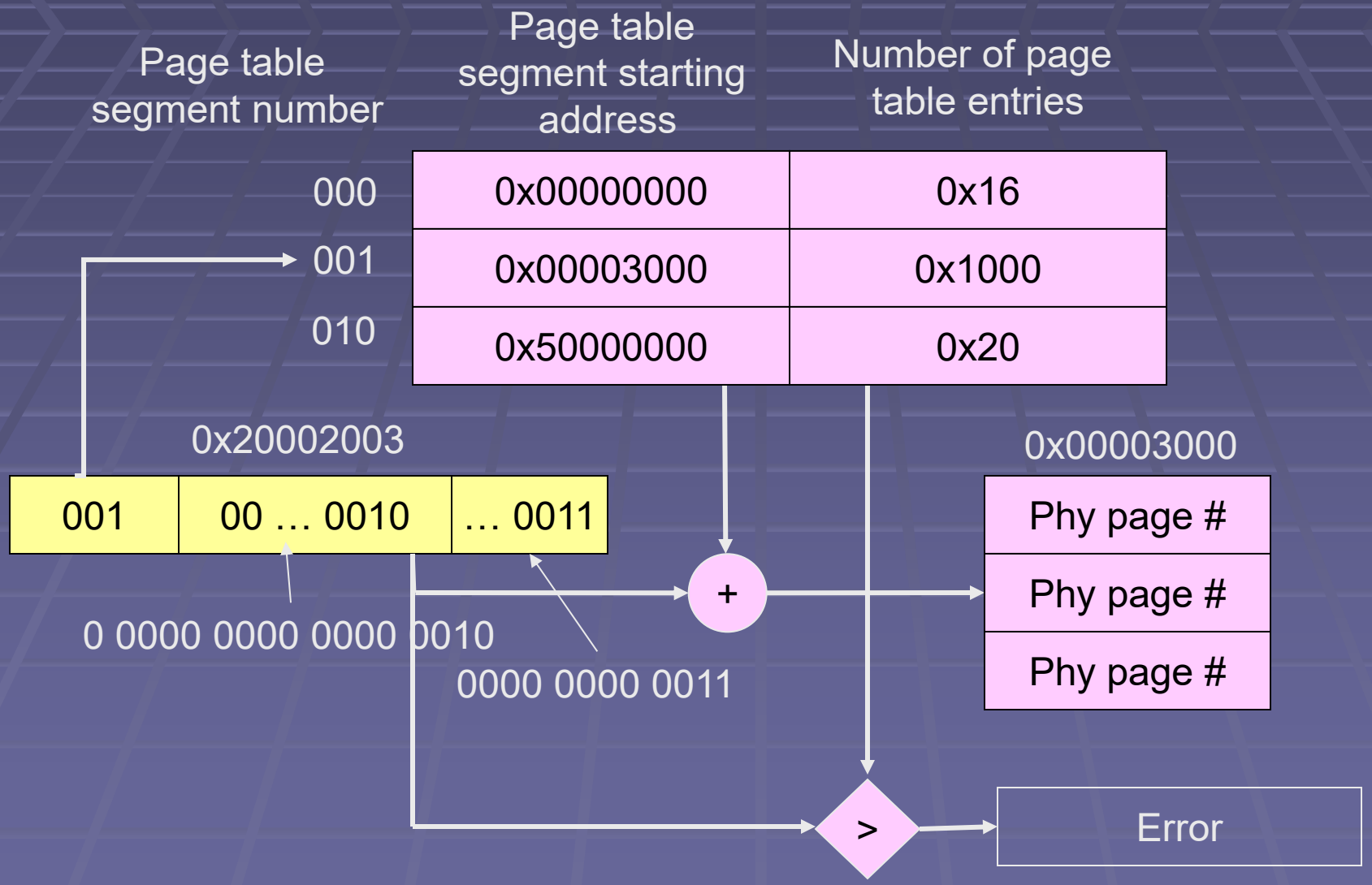
Segmented Paging



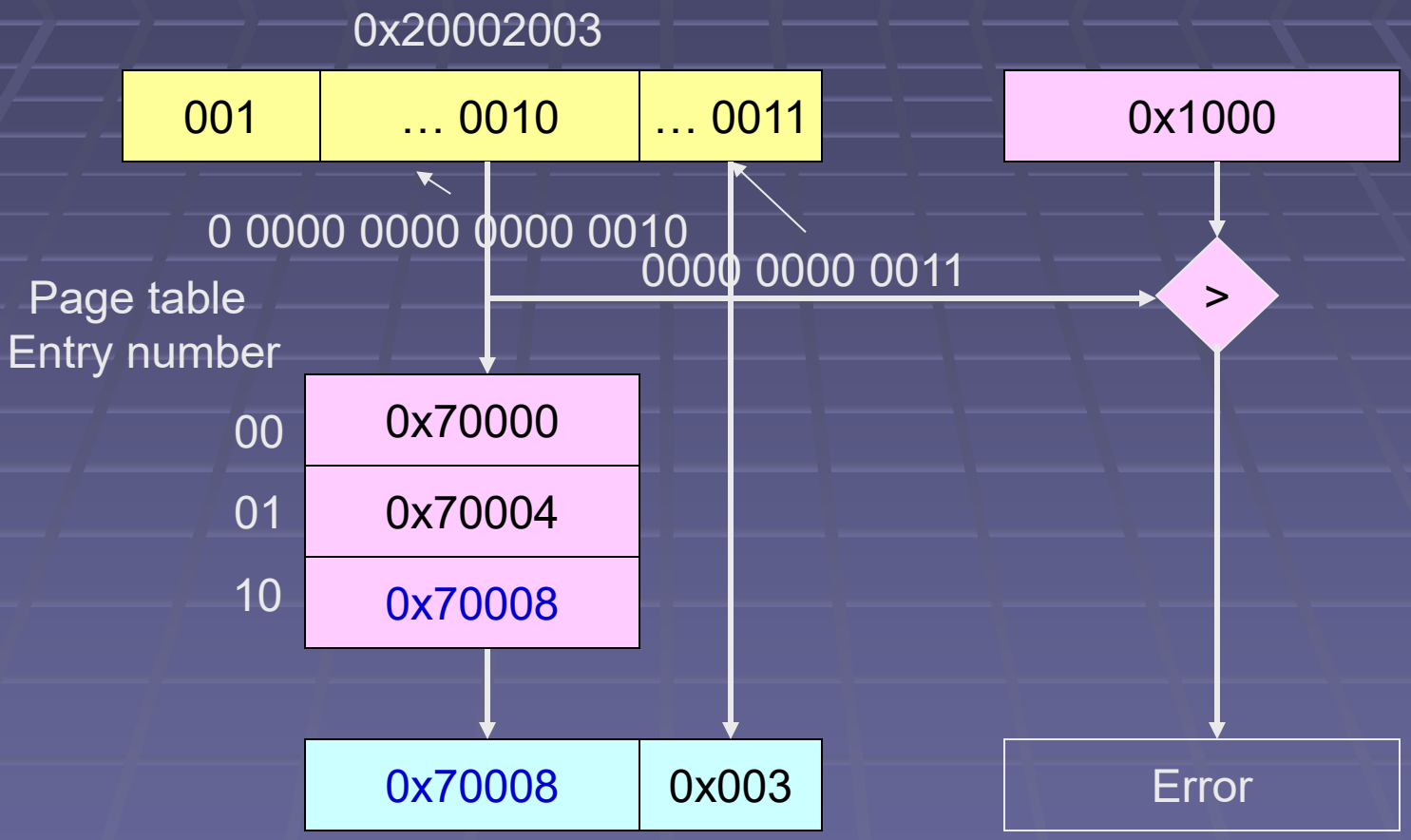
Segmented Paging



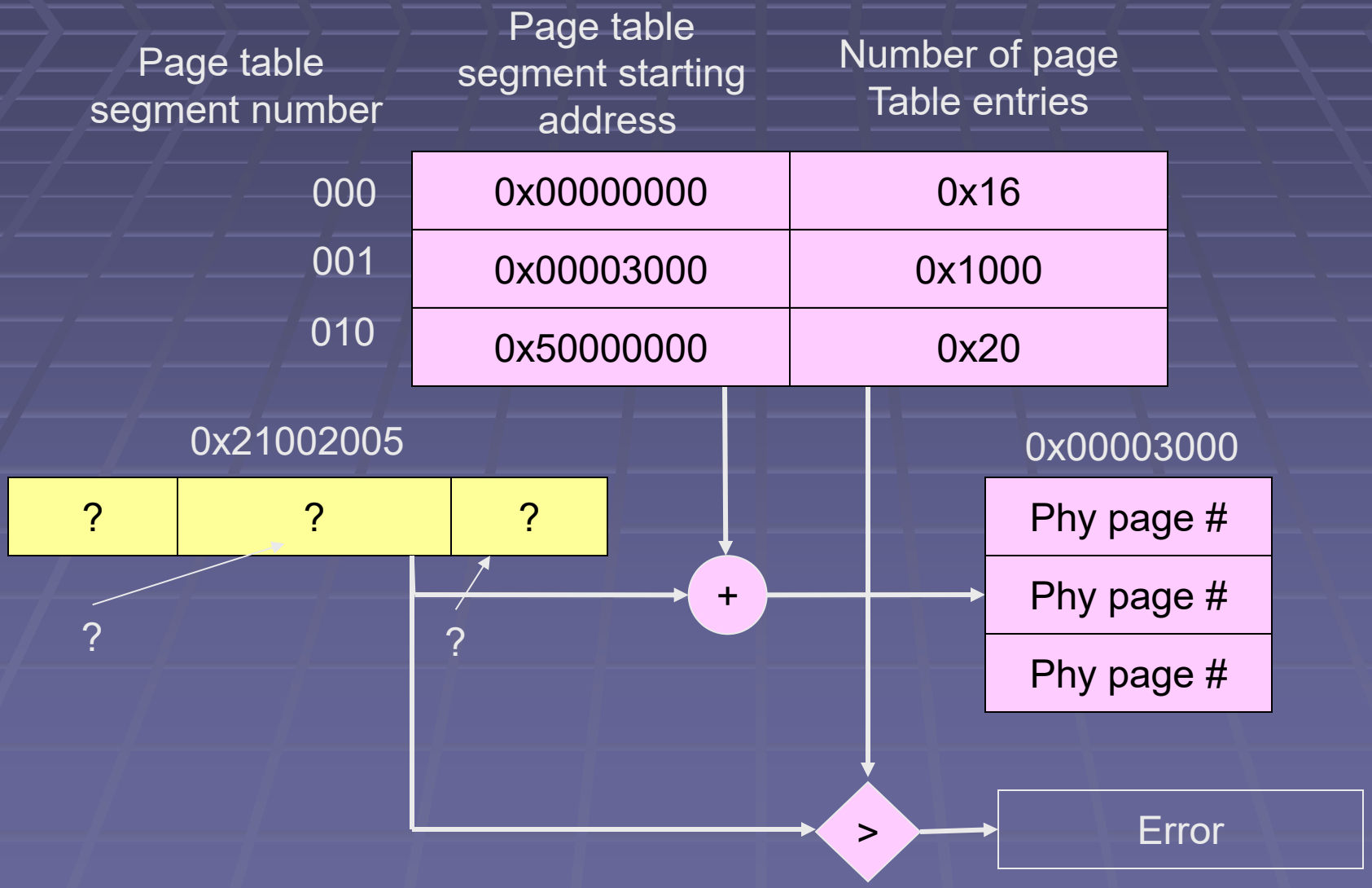
Segmented Paging Example



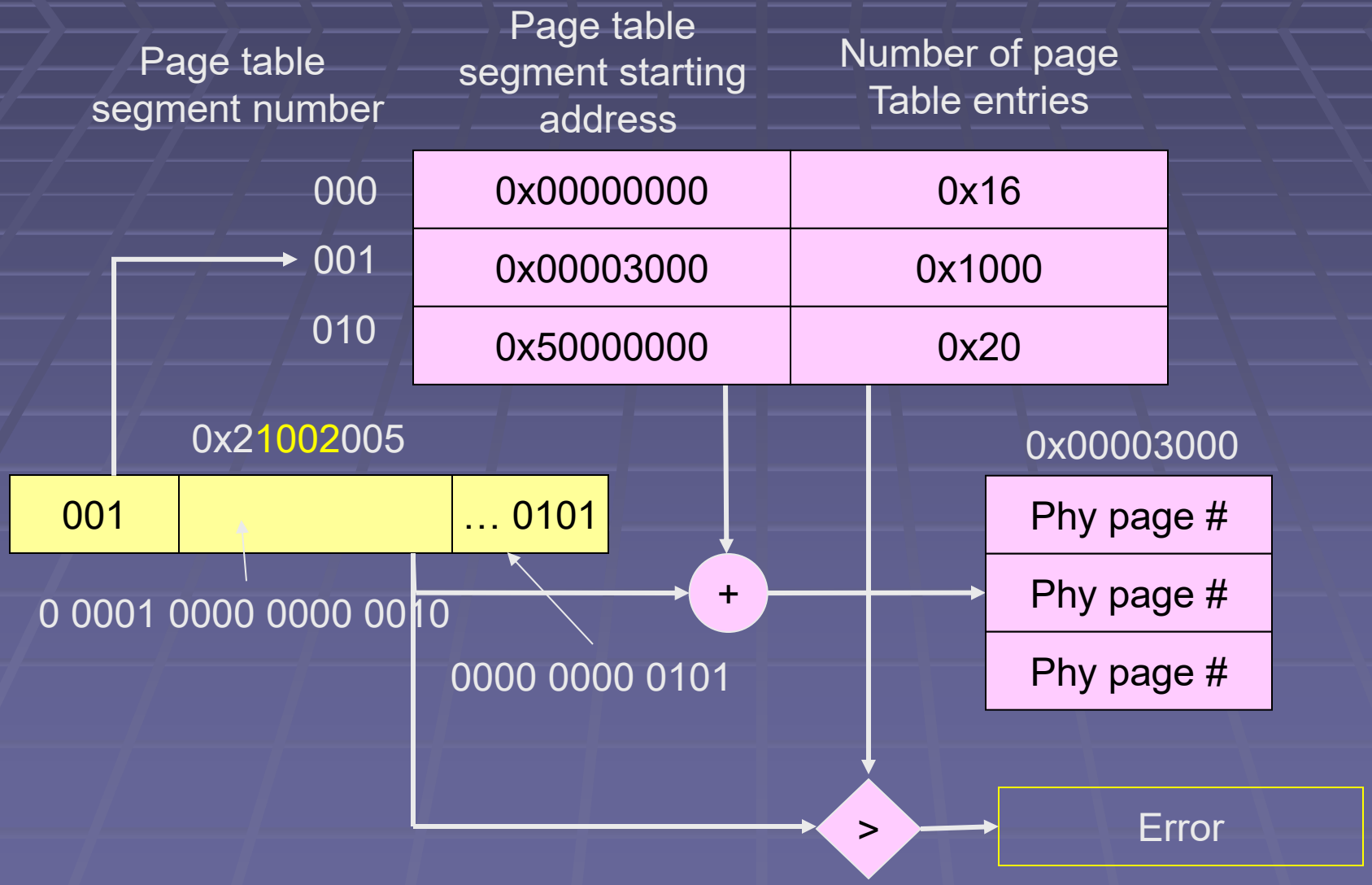
Segmented Paging



Segmented Paging Example



Segmented Paging Example



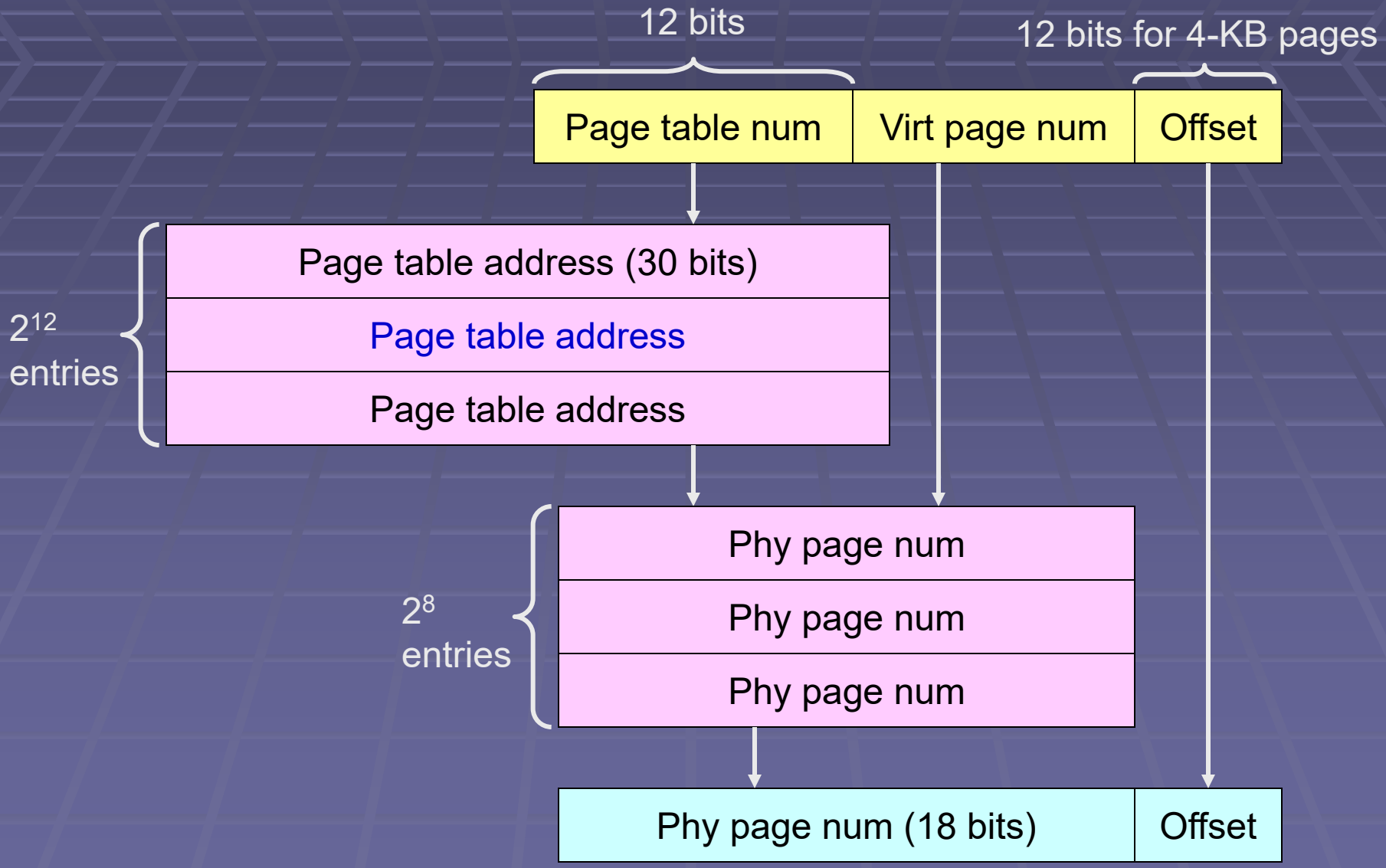
Segmented Paging Translation

- 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

- virtual_address = page_table_num:virtual_page_num:offset
- page_table = page_table_address[page_table_num]
- physical_page_num = page_table[virtual_page_num]
- physical_address = physical_page_num: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
  = hash(pid, virtual_page_num):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

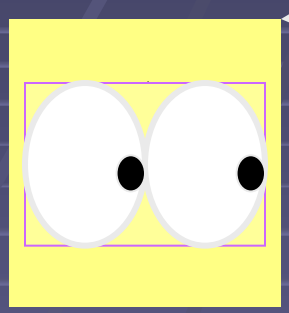
Details of Dual-mode Operations

- 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



jump

User level

Kernel level

Translation table

Hardware pointer

user mode

PC

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)

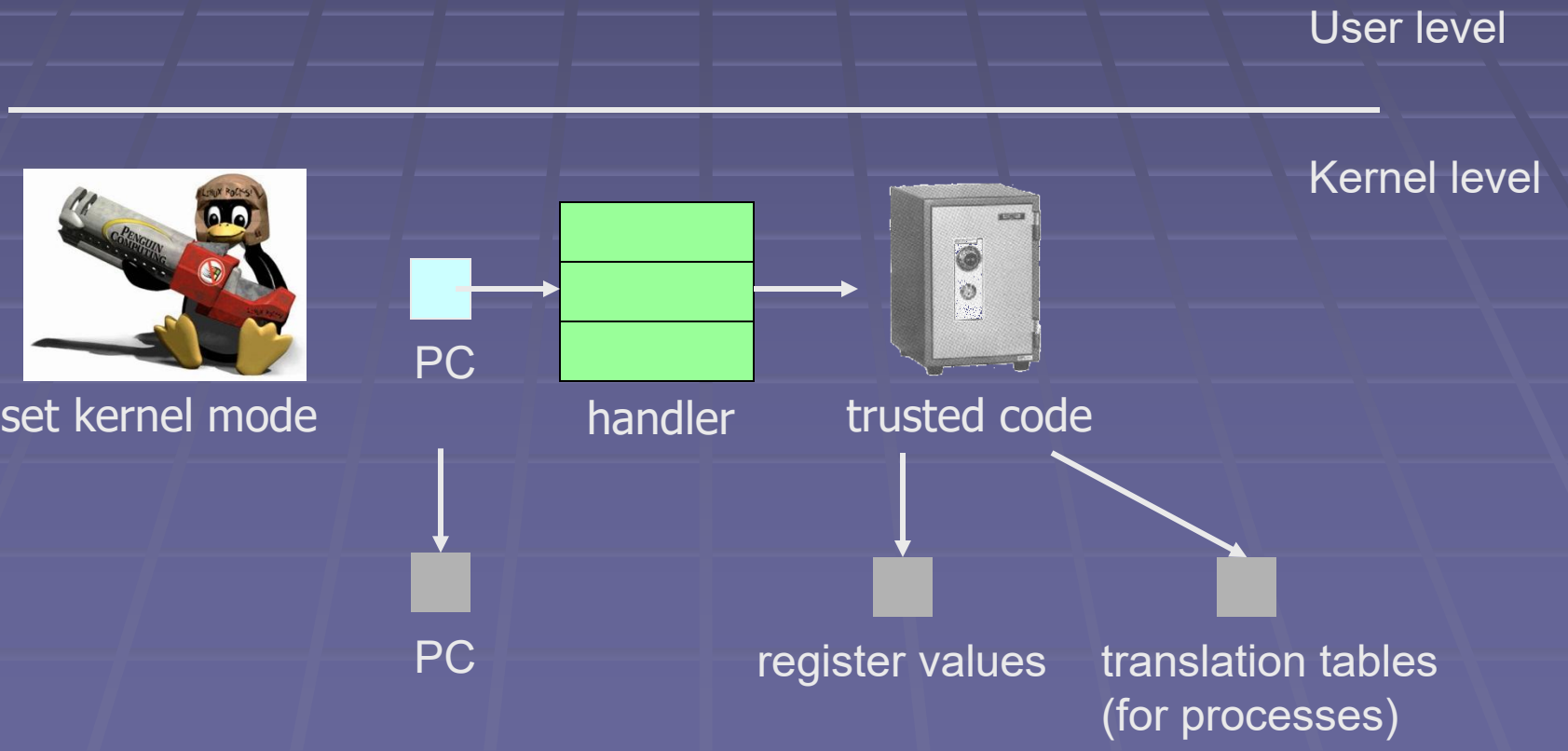
Switching from User Mode to Kernel Mode

- 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

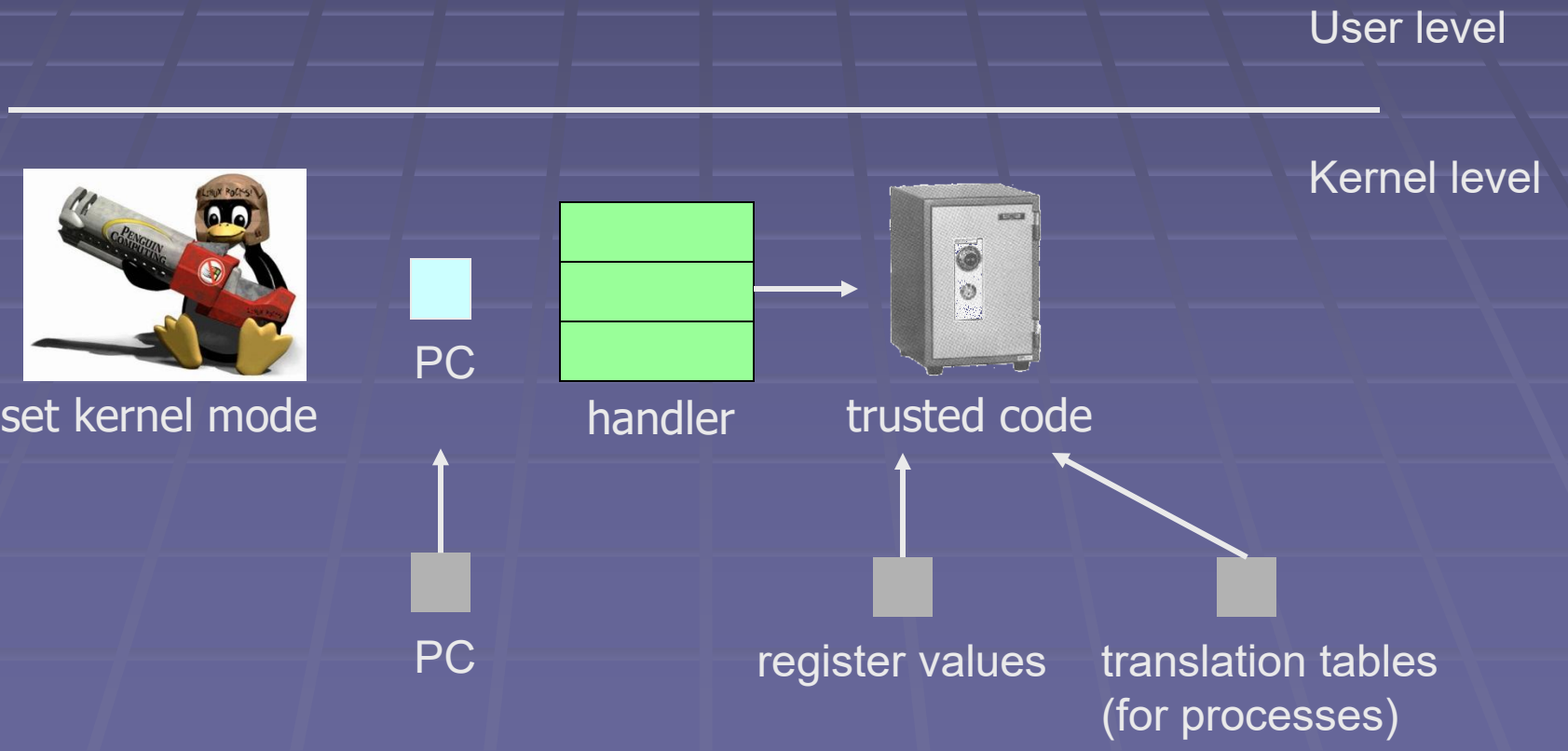
Switching from User Mode to Kernel Mode

- 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

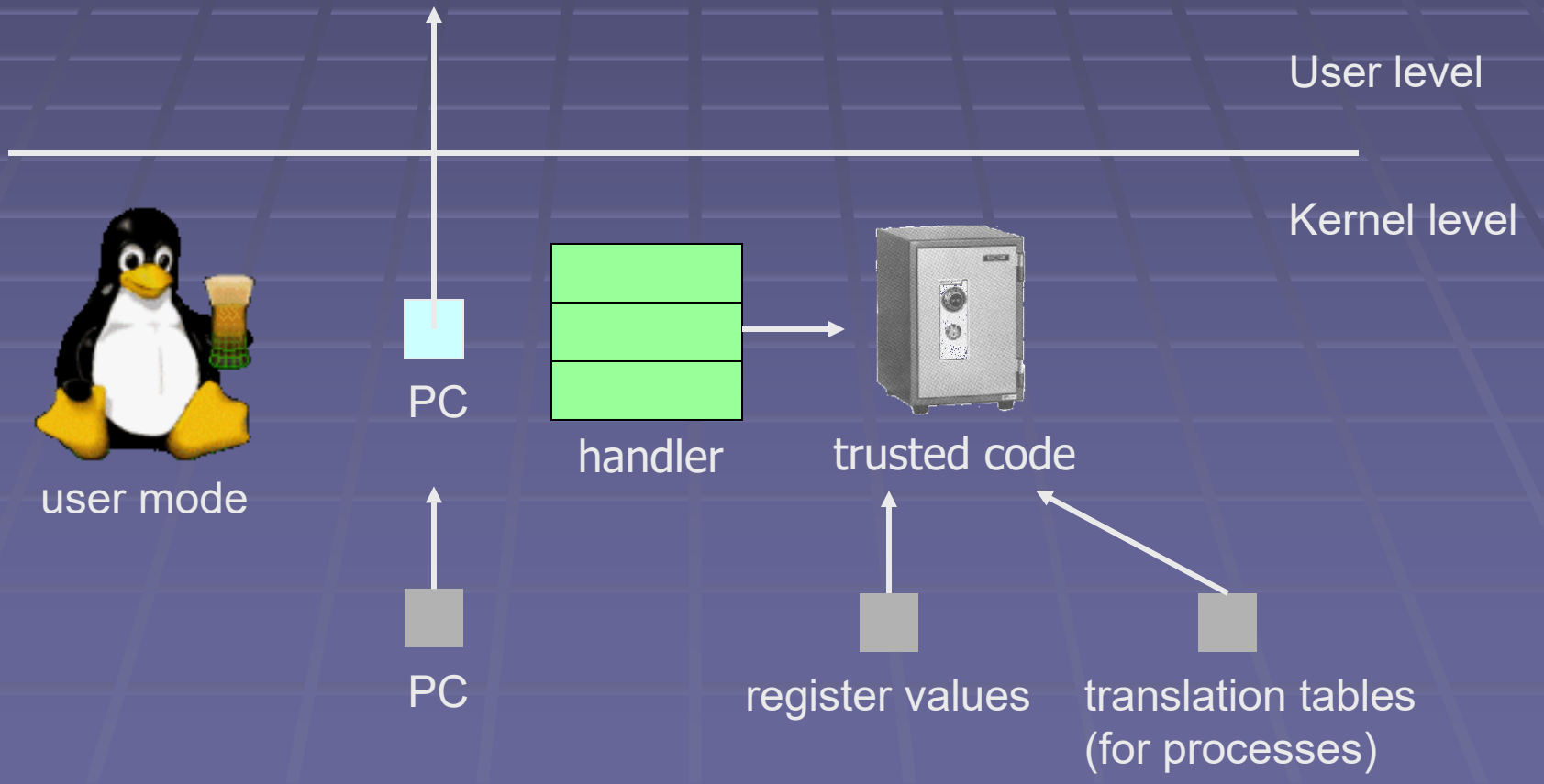
User → Kernel


User level
Kernel level
set kernel mode
PC
handler
trusted code
PC
register values
translation tables
(for processes)

Kernel → User


User level
Kernel level
set kernel mode
PC
handler
trusted code
PC
register values
translation tables
(for processes)

Kernel → User


User level
Kernel level
user mode
PC
handler
trusted code
PC
register values
translation tables
(for processes)

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(P_1, message);
  - receive(P_2, 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
`st r2, (r1)`	`safe` = a legal address
	`safe = r1`
	Check `safe` is still legal
	`st r2, (safe)`

- 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

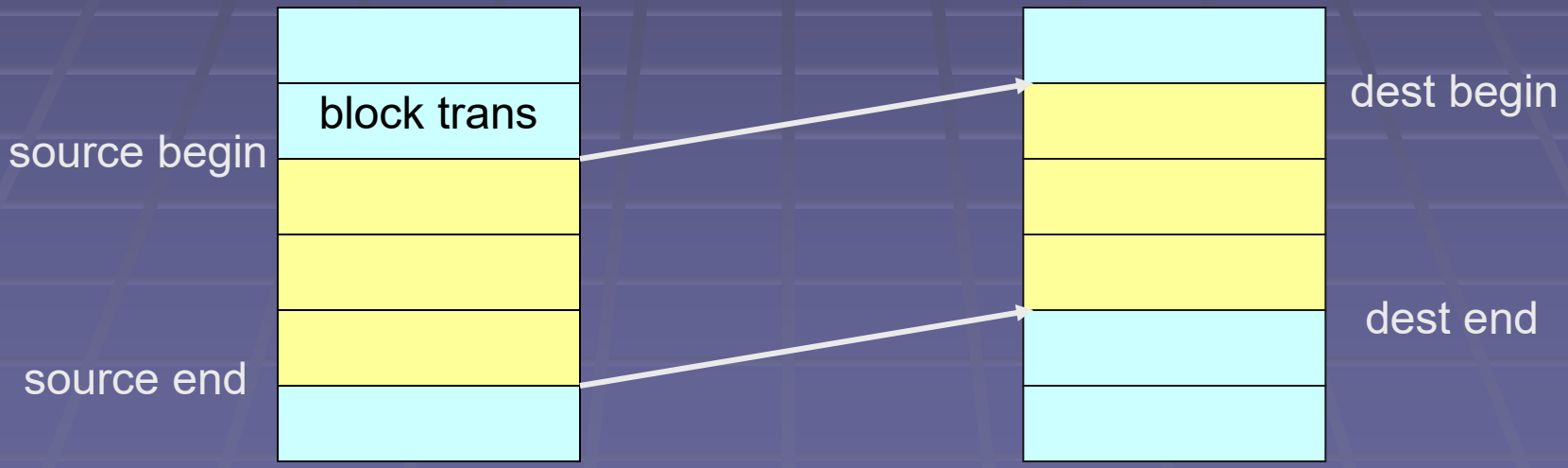
More on Transparent Page Faults

- 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

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

FIFO

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

FIFO

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

FIFO

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

FIFO

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

FIFO

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

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

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

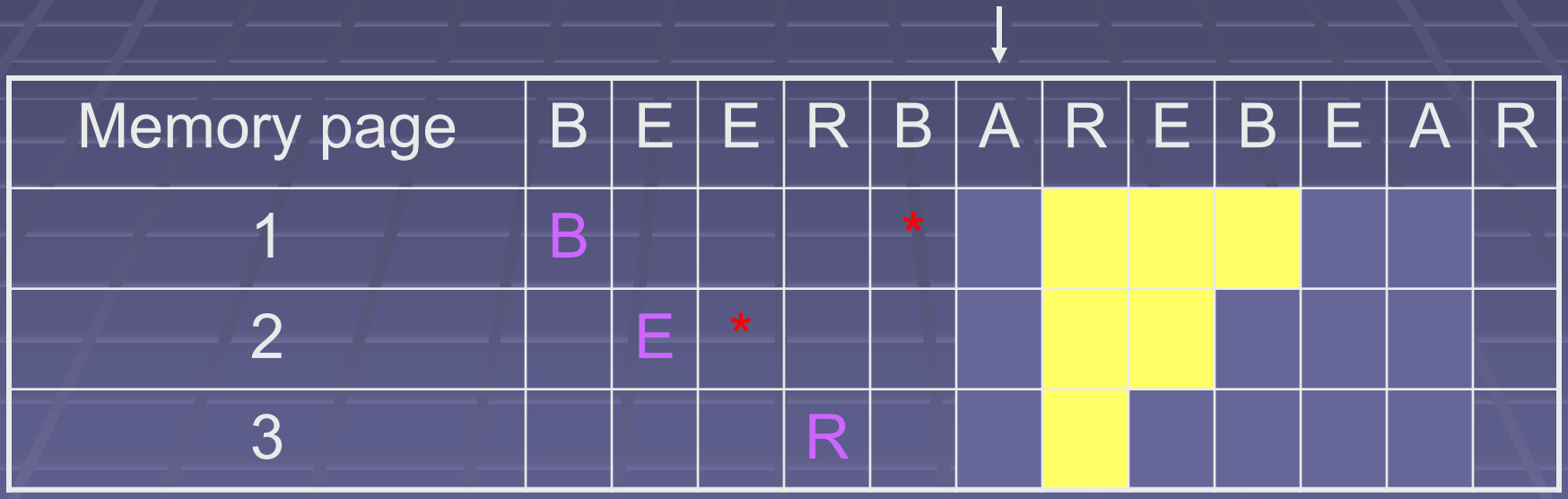
MIN

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

MIN

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

MIN



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

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

LRU

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

LRU

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

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				*			E				
2		E	*			A						
3				R			*					

LRU

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

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

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

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

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

LFU

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

LFU

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

LFU

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

LFU

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

LFU

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

LFU

Memory page	B	E	E	R	B	A	R	E	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

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					

LFU

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					

LFU

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

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		

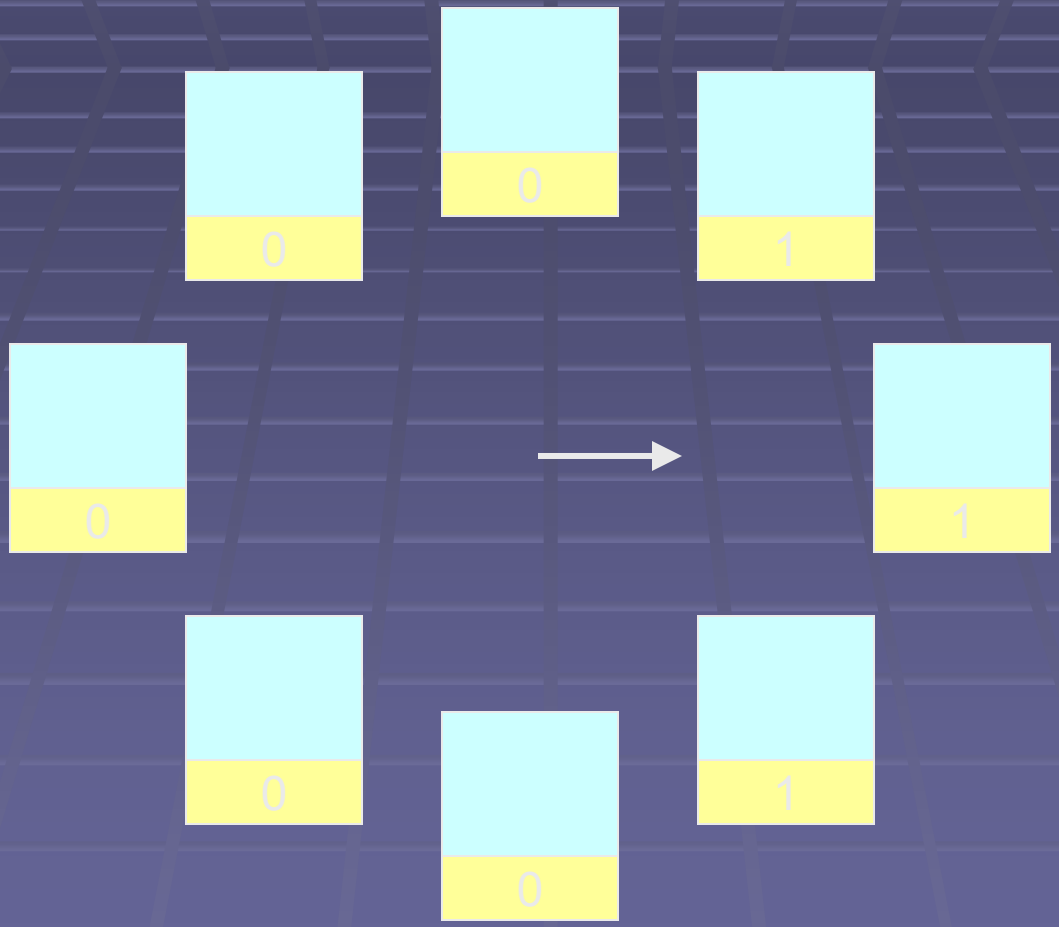
Implementing LRU

- 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

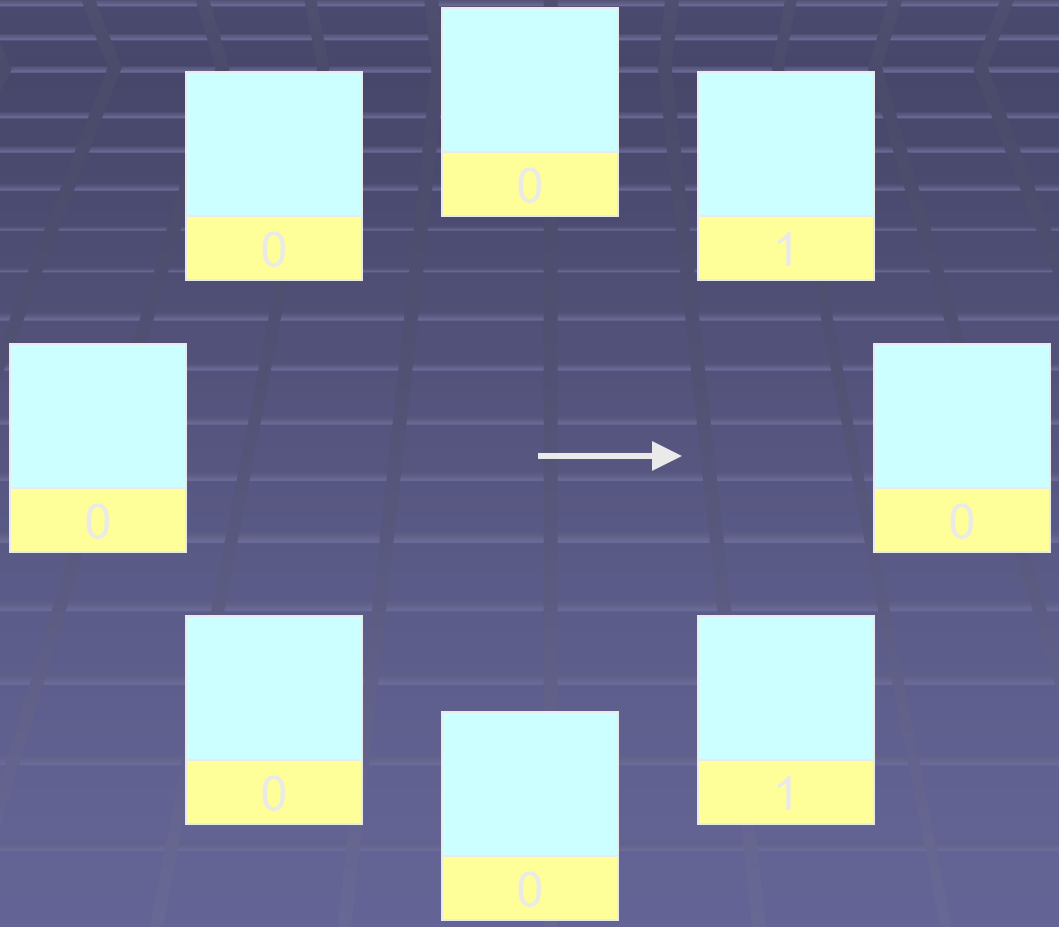
Clock Algorithm

- 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

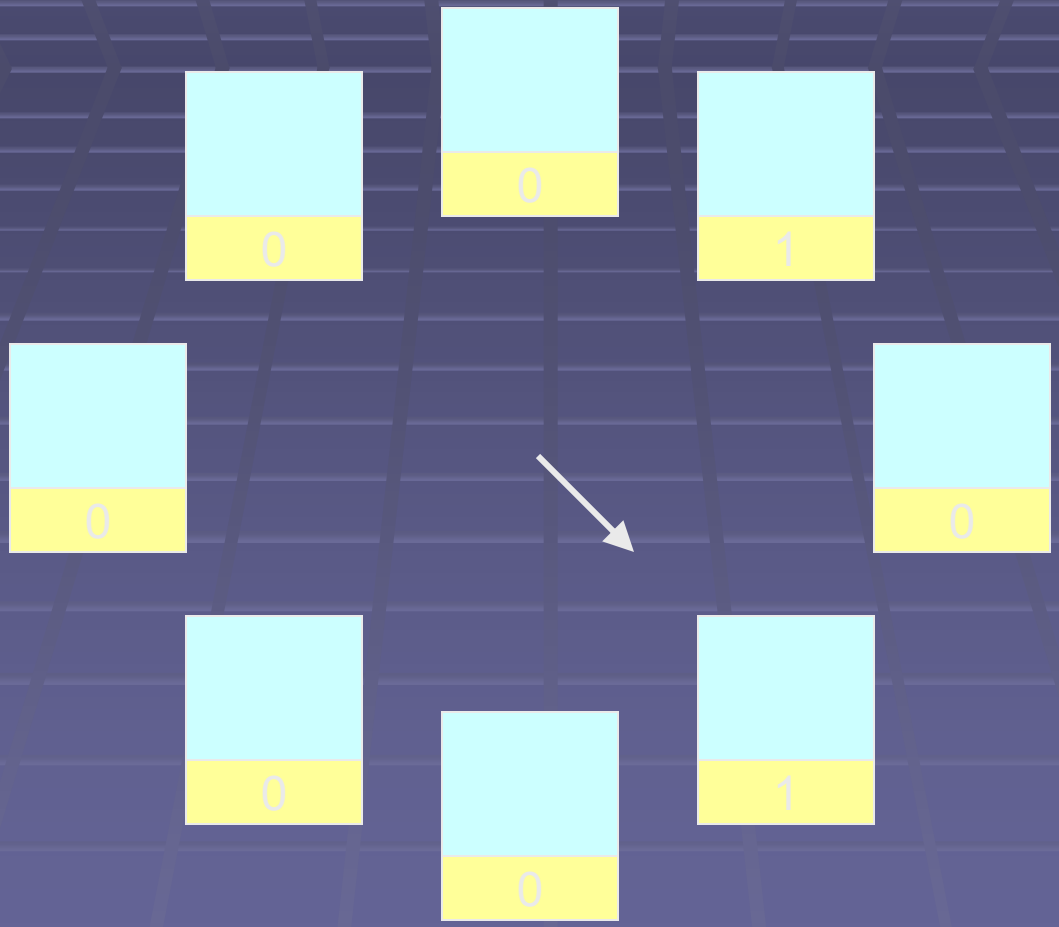
Clock Algorithm



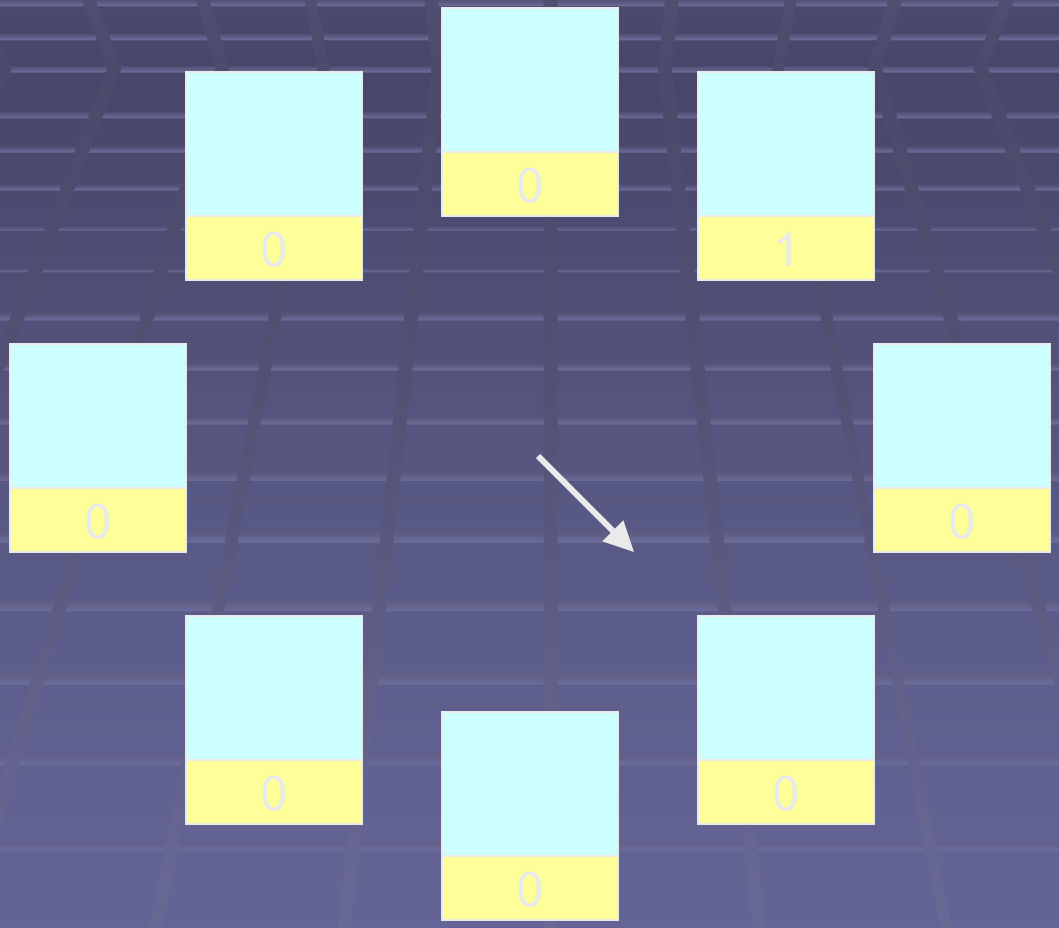
Clock Algorithm



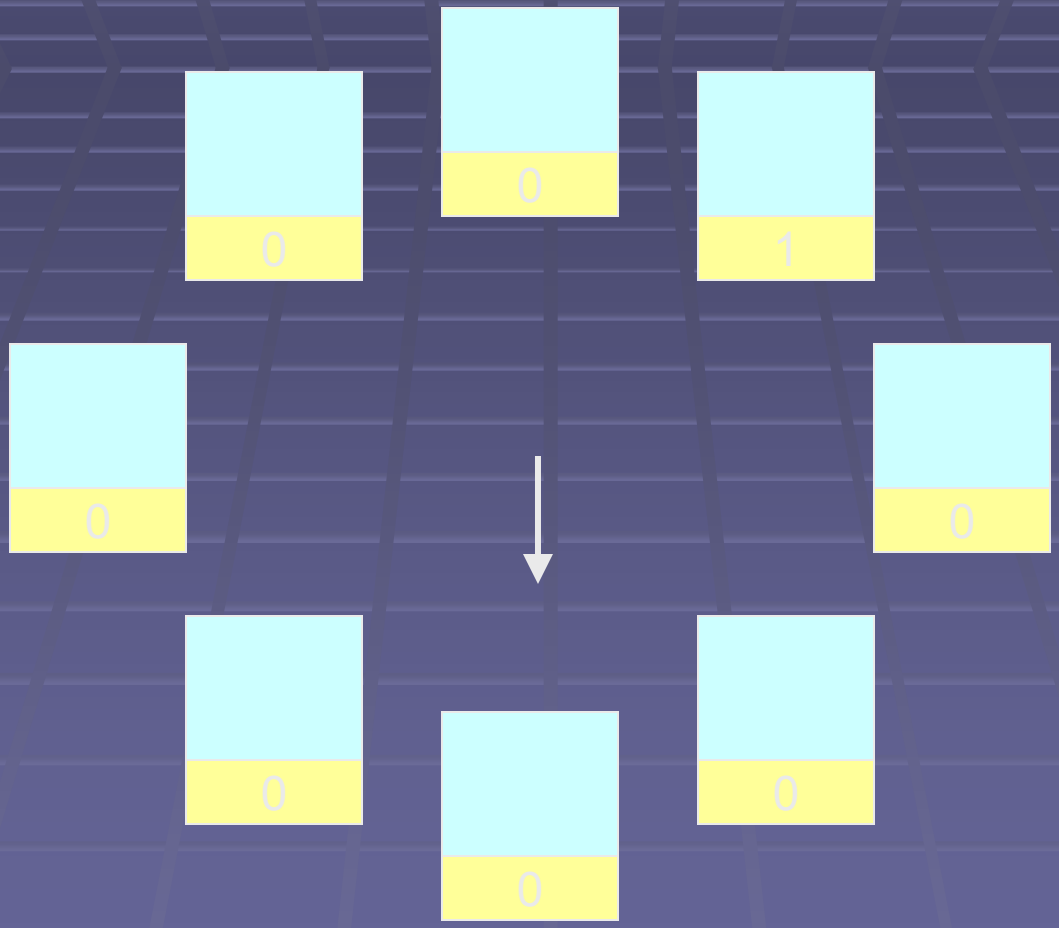
Clock Algorithm



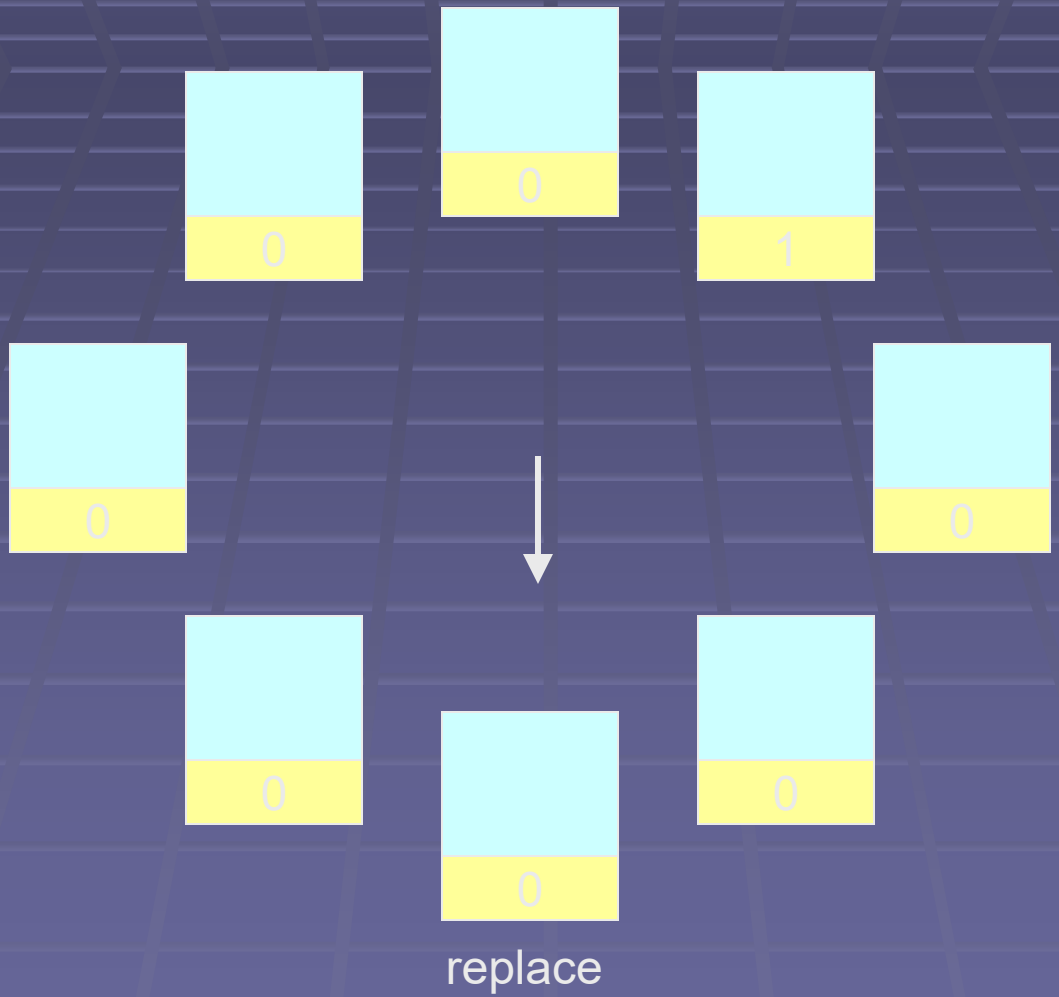
Clock Algorithm



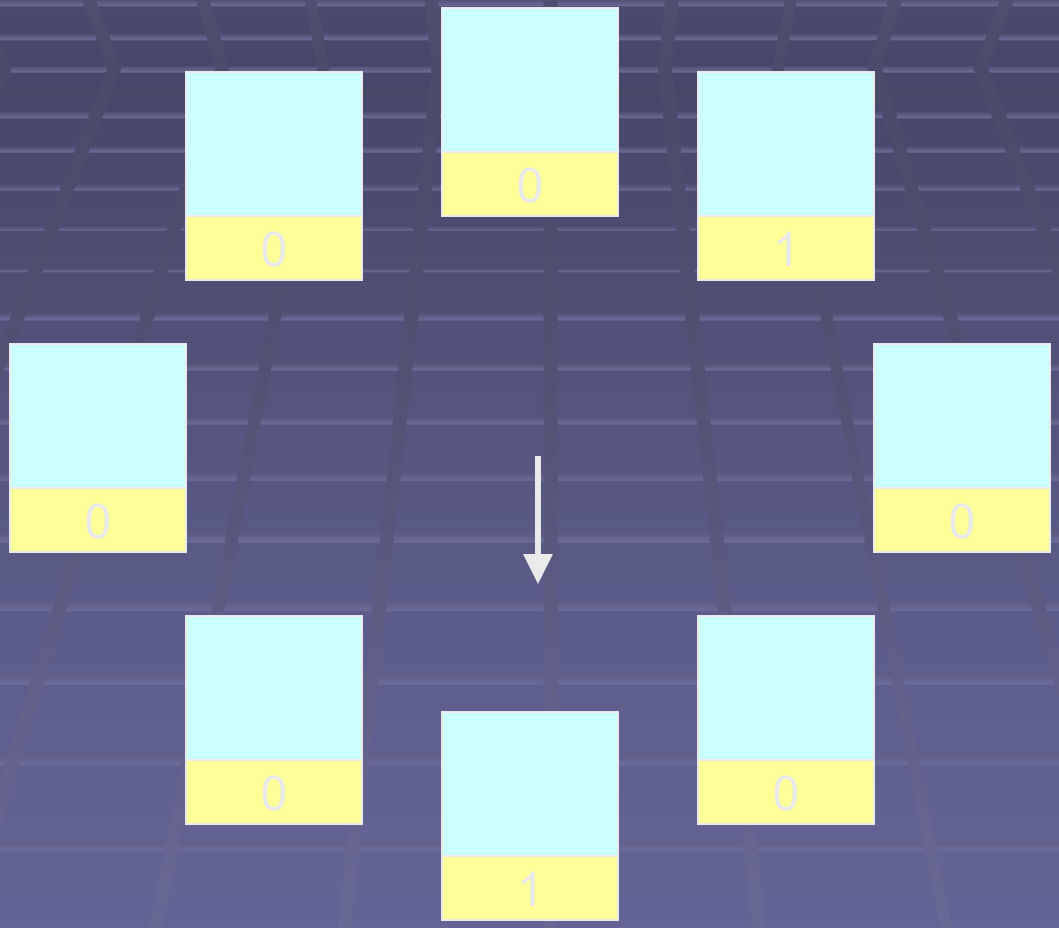
Clock Algorithm



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 → ∞, Nth Chance Algorithm → 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 → ∞, working set → 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

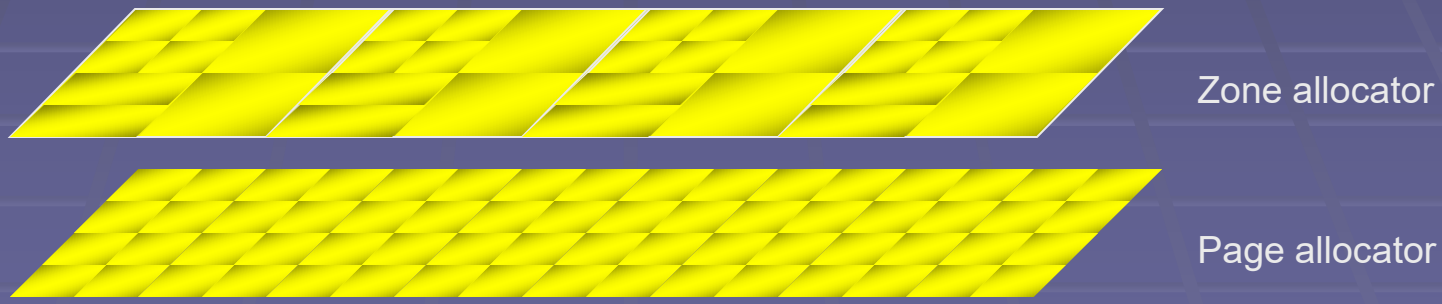
Linux Memory Manager (1)

- Page allocator maintains individual pages

Page allocator

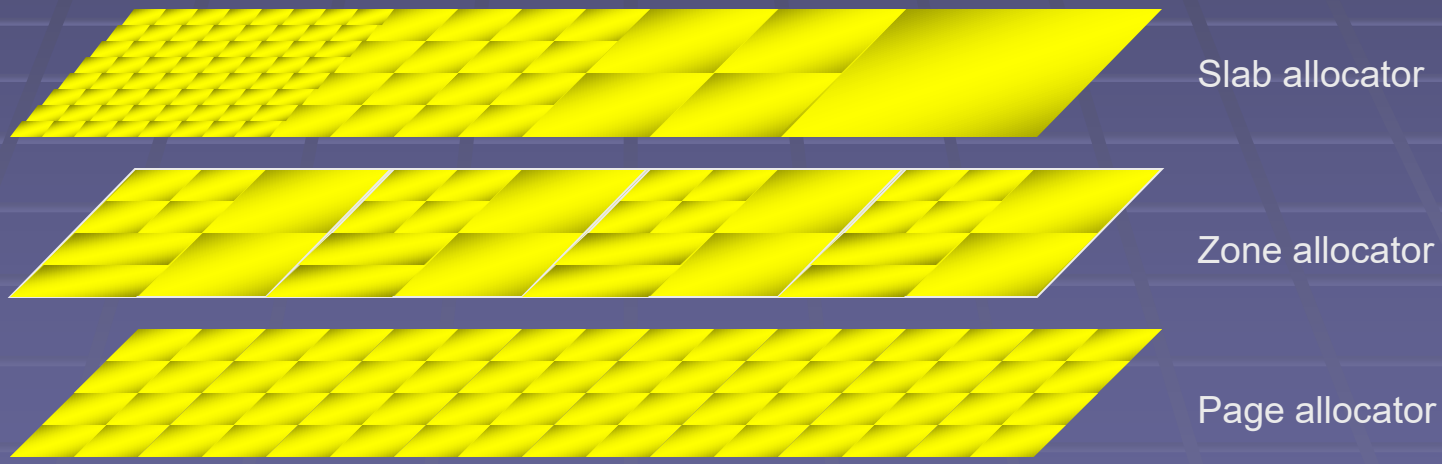
Linux Memory Manager (2)

- 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