
Address Spaces, Segments, Pages

16 January 2025
Lecture 11

Slides adapted from John Kubiatowicz (UC Berkeley)

Concept Review

Deadlock

Starvation

Deadlock Graph

Deadlock conditions

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

Deadlock detection
algorithm

Bankers Algorithm

Topics for Today

- Address Spaces and Segmentation
 - Loading and Translating
 - Segments
 - Fragmentation
- Page tables

Today's concepts

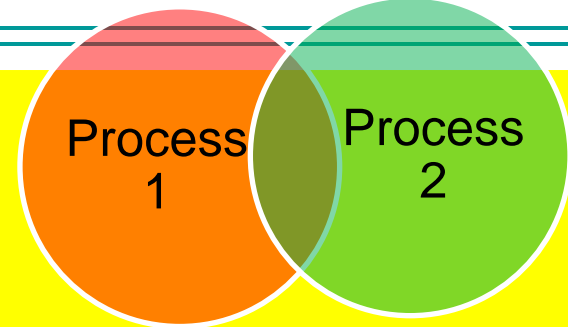


Virtualizing Resources

- Physical Reality: Different Processes/Threads share the same hardware
 - Need to multiplex CPU (Just finished: scheduling)
 - Need to multiplex use of Memory (Today)
- Why worry about memory sharing?
 - The complete working state of a process and/or kernel is defined by its data in memory (and registers)
 - Consequently, cannot just let different threads of control use the same memory
 - Physics: Two different pieces of data cannot occupy the same locations in memory
 - Probably don't want different threads to even have access to each other's memory (protection)

Important Aspects of Memory Multiplexing

Translation
 $a \rightarrow b$



Controlled
Overlap

Protection



Translation: $a \rightarrow b$

Ability to translate accesses from one address space (**virtual**) to a different one (**physical**)

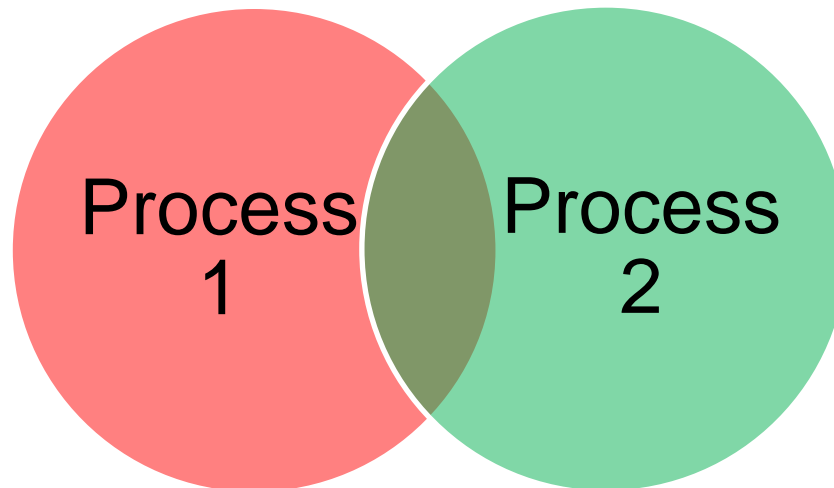
When translation exists, **processor uses virtual addresses**
physical memory uses physical addresses

Side effects:

- Can be used to **avoid overlap**
- Can be used to give **uniform view** of memory to programs

Controlled Overlap

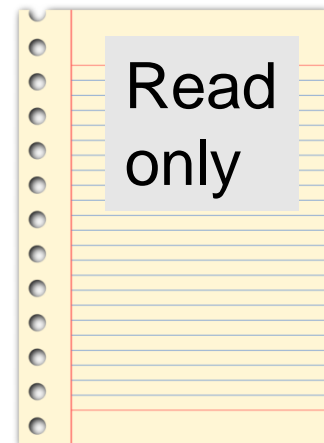
- Separate state of threads **should not collide** in physical memory.
 - Unexpected overlap causes chaos!
- Want to be able to overlap when desired for **communication**
 - Also debugging...



Protection



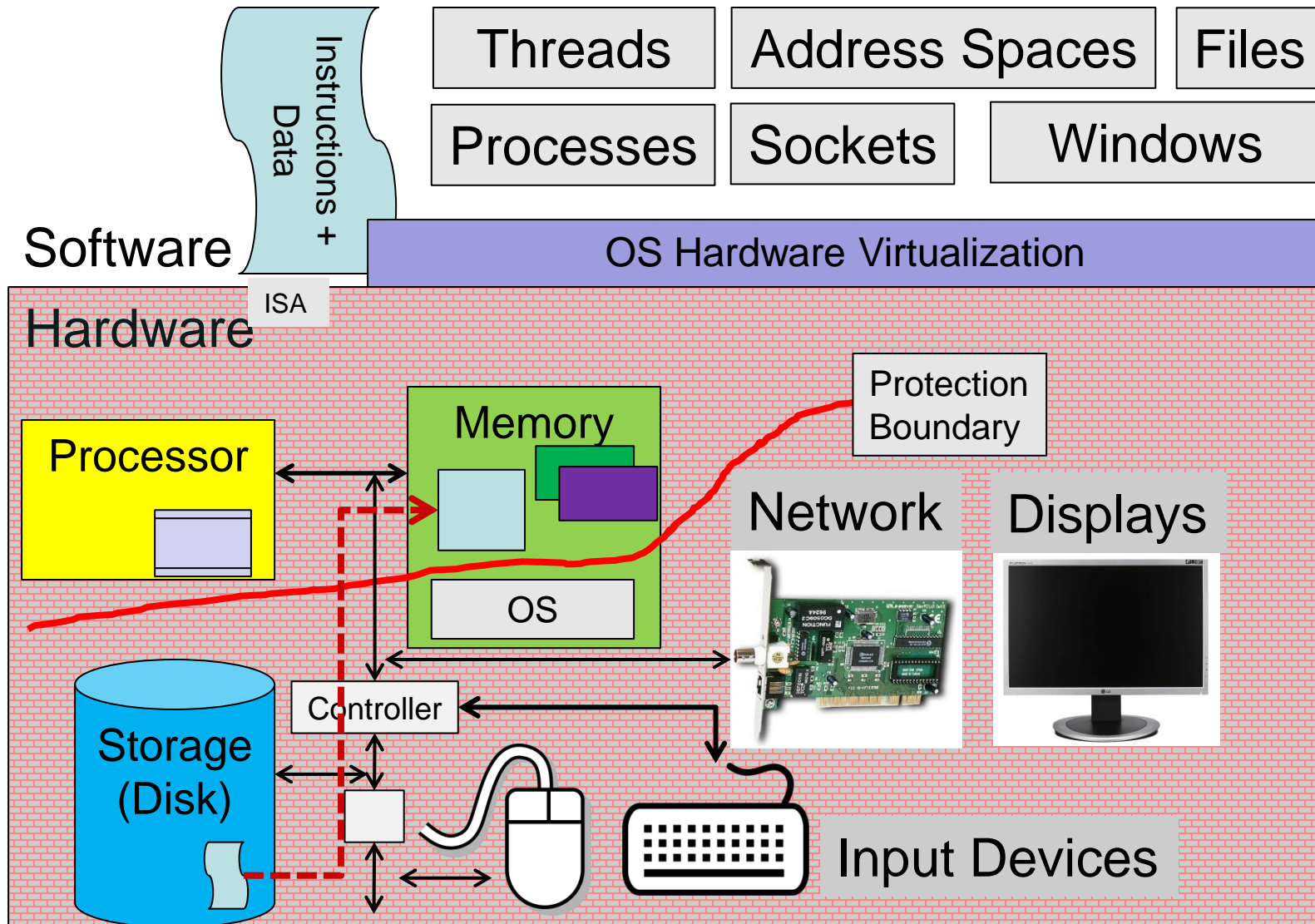
- Prevent access to **private memory** of other processes
- Different pages of memory can be given special behavior (Read Only, Invisible to user programs, etc).
- Kernel data protected from User programs
- Programs protected from themselves



So Far

- Address Spaces and Segmentation
 - Loading and Translating
 - Segments
 - Fragmentation
- Page tables

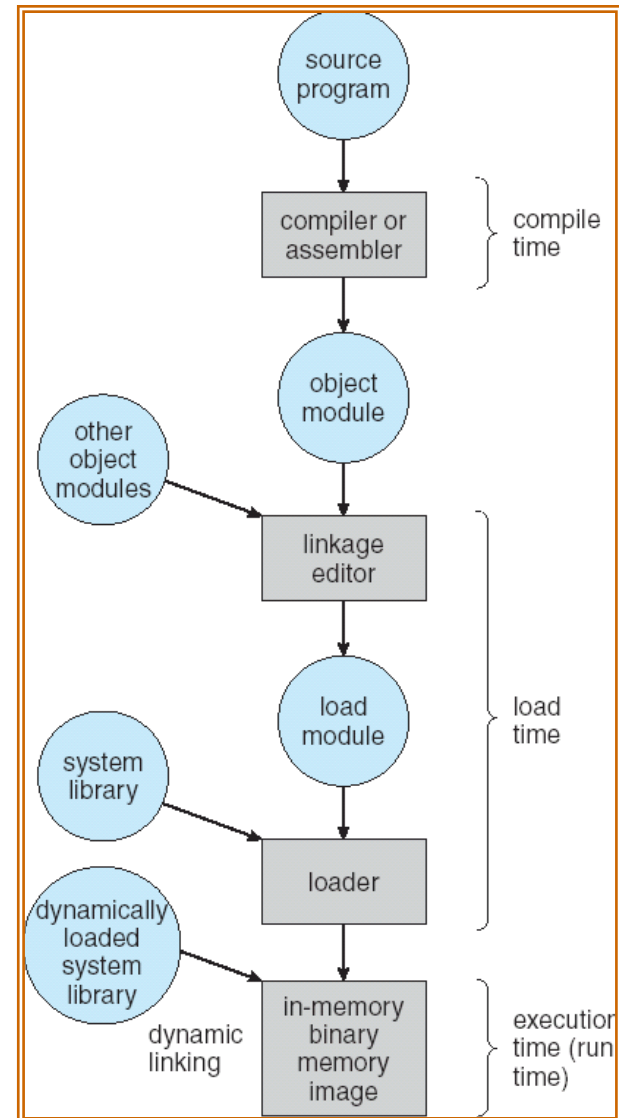
Recall: Loading



Recall a bit of the old days →
Relocating loaders

Multi-step Processing of a Program for Execution

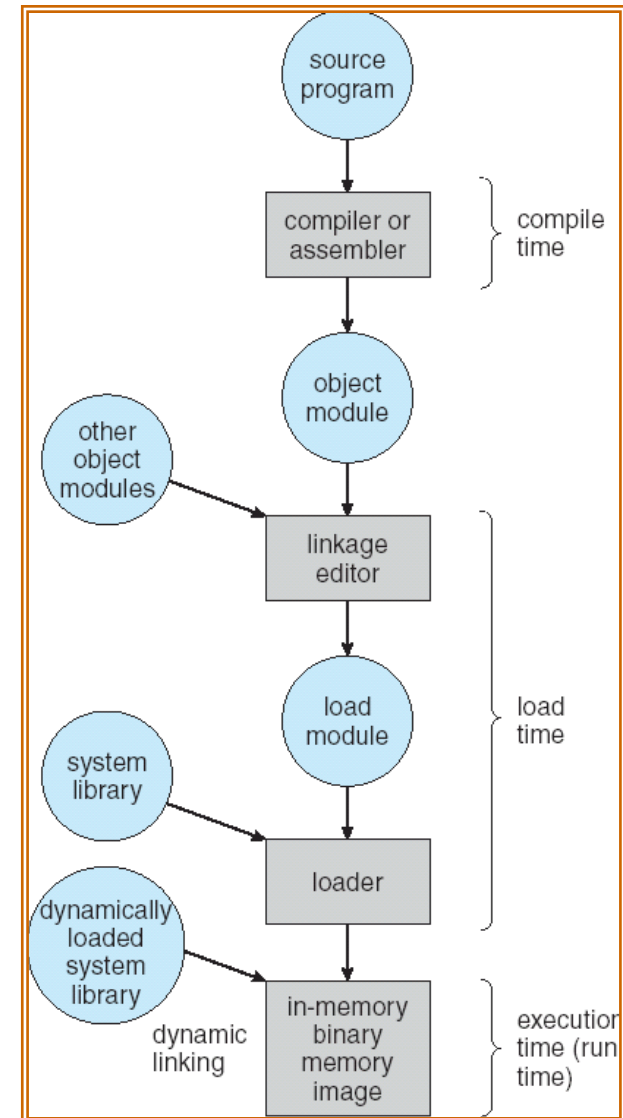
- Preparation of a program for execution involves components at:
 - **Compile time** (ex. gcc)
 - **Link/Load time** (UNIX ld does link)
 - **Execution time** (ex. dynamic libs)
- Addresses can be bound to final values anywhere in this path
 - Depends on hardware support
 - Also depends on operating system



Multi-step Processing of a Program for Execution

- Dynamic Libraries

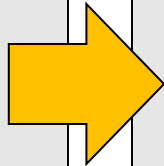
- Linking postponed until execution
- Small piece of code, *stub*, used to locate appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes routine



Binding of Instructions and Data to Memory

Process View of Memory

```
data1: dw      32
      ...
start: lw      r1,0(data1)
      jal     checkit
loop:  addi    r1, r1, -1
      bnz     r1, loop
      ...
checkit: ...
```



Physical View of Memory

```
0x0300 00000020
...
0x0900 8C2000C0
0x0904 0C000280
0x0908 2021FFFF
0x090C 14200242
...
0x0A00
```

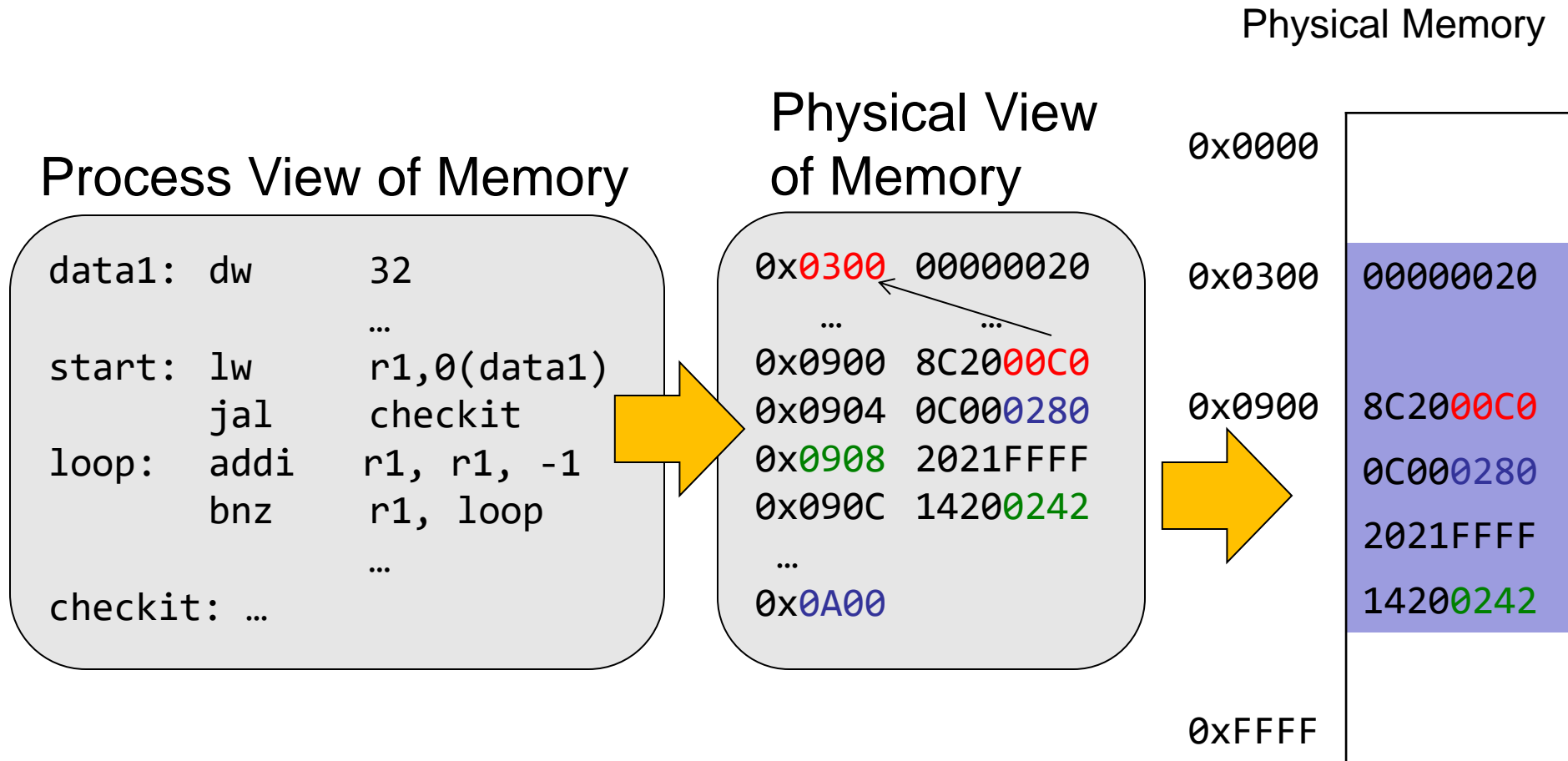
Assume 4 Byte words

0x300 = 4 x 0x0C0

0x0C0 = 0000 1100 0000

0x300 = 0011 0000 0000

Binding of Instructions and Data to Memory



Second copy of the program

Process View of Memory

```
data1: dw      32
...
start: lw       r1,0(data1)
      jal      checkit
loop:  addi     r1, r1, -1
      bnz      r1, loop
...
checkit: ...
```

Physical View of Memory

```
0x0300 00000020
...
0x0900 8C2000C0
0x0904 0C000280
0x0908 2021FFFF
0x090C 14200242
...
0x0A00
```

Physical Memory

0x0000

0x0900

App X

0xFFFF

What do we do with the addresses? Need translation!

Second copy of the program

Process View of Memory

```
data1: dw      32
      ...
start: lw      r1,0(data1)
      jal     checkit
loop:  addi    r1, r1, -1
      bnz     r1, loop
      ...
checkit: ...
```

Physical View of Memory

```
0x1300 00000020
...
0x1900 8C2004C0
0x1904 0C000680
0x1908 2021FFFF
0x190C 14200642
...
0x1A00
```

Physical Memory

0x0000	
0x0900	App X
0x1300	00000020
0x1900	8C2004C0 0C000680 2021FFFF 14200642
0xFFFF	

- One of many possible translations!
- When does translation take place?
 - Compile time, Link/Load time, or Execution time?

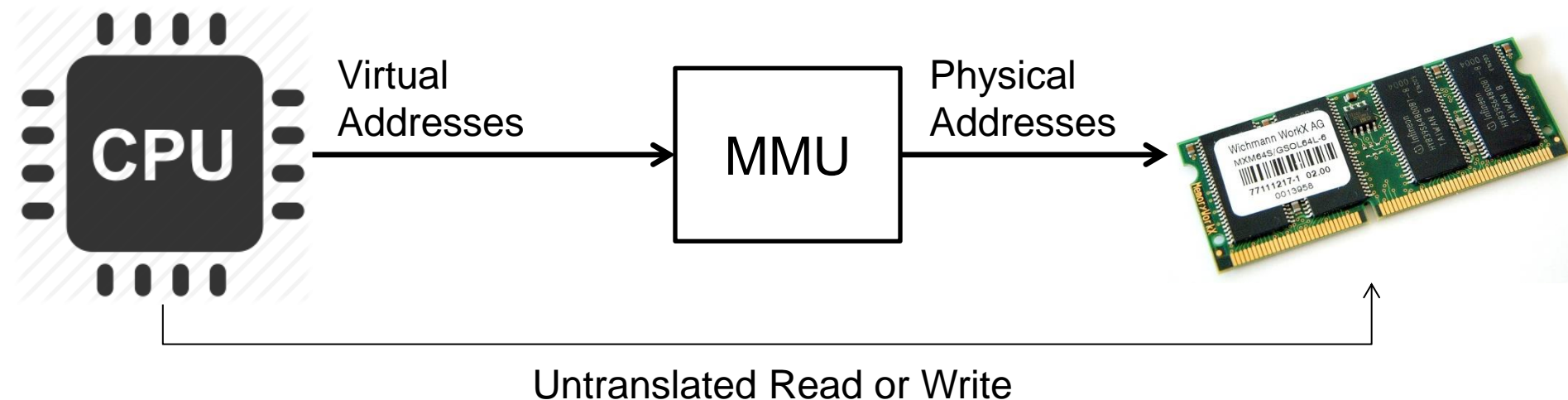
So Far

- Address Spaces and Segmentation
 - Loading and Translating
 - Segments
 - Fragmentation
- Page tables

General Address translation (Review)

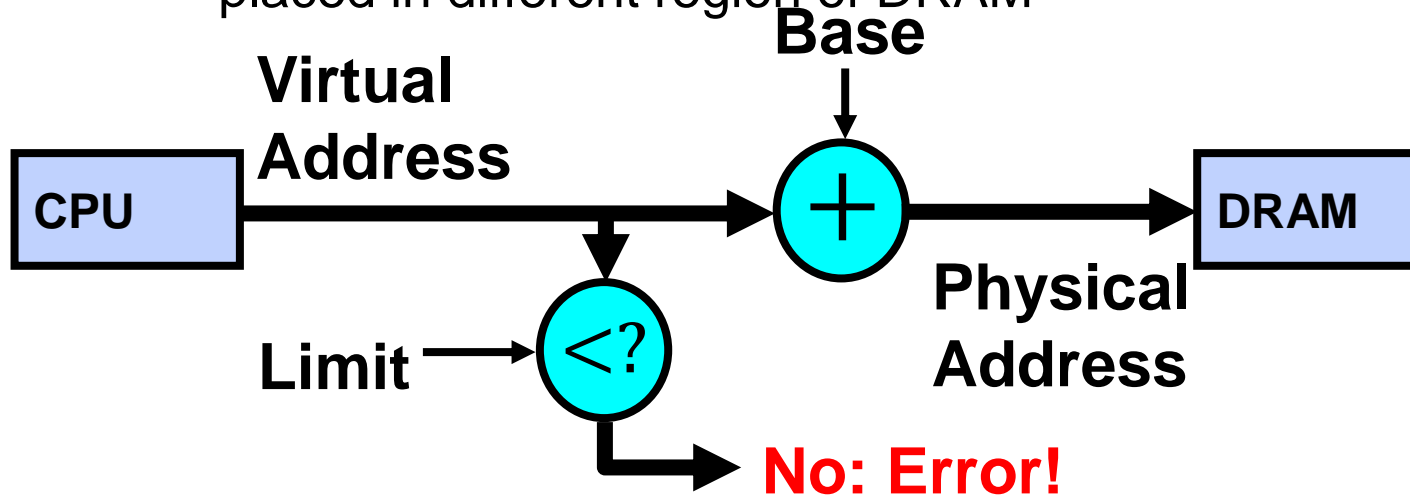
- **Address Space:** All the addresses and state a process can touch
 - Each process and kernel have different address spaces
- Therefore → Two views of memory:
 - View from the **CPU** (what program sees, virtual memory)
 - View from **memory** (physical memory)
 - **Translation box (MMU)** converts between the two views
- Translation makes it **much easier** to implement protection
 - If **Job A** cannot even gain access to **Job B**'s data, **no way** for A to adversely affect B
- With translation, every program can be linked/loaded into the **same region of user address space**

General Address translation (Review)

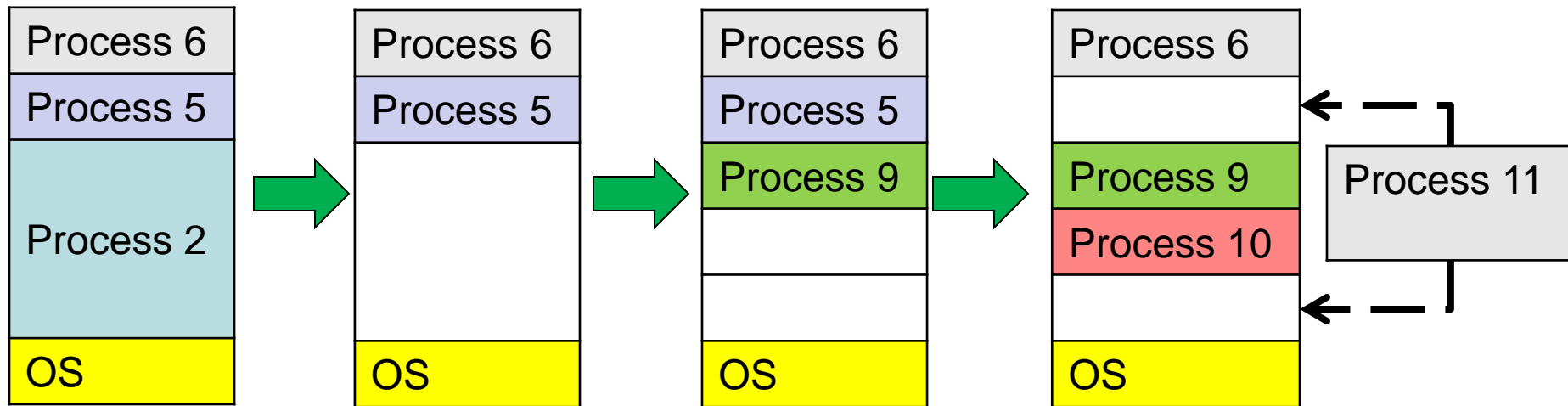


Simple Example: Base and Bounds (CRAY-1)

- Base + limit for **dynamic address translation (runtime)**
 - Alter address of every load/store by adding “base”
 - Error if address > limit
- Gives program illusion it is running alone, with memory starting at 0
 - Program gets continuous region of memory
 - Addresses in program do not need relocation when program placed in different region of DRAM



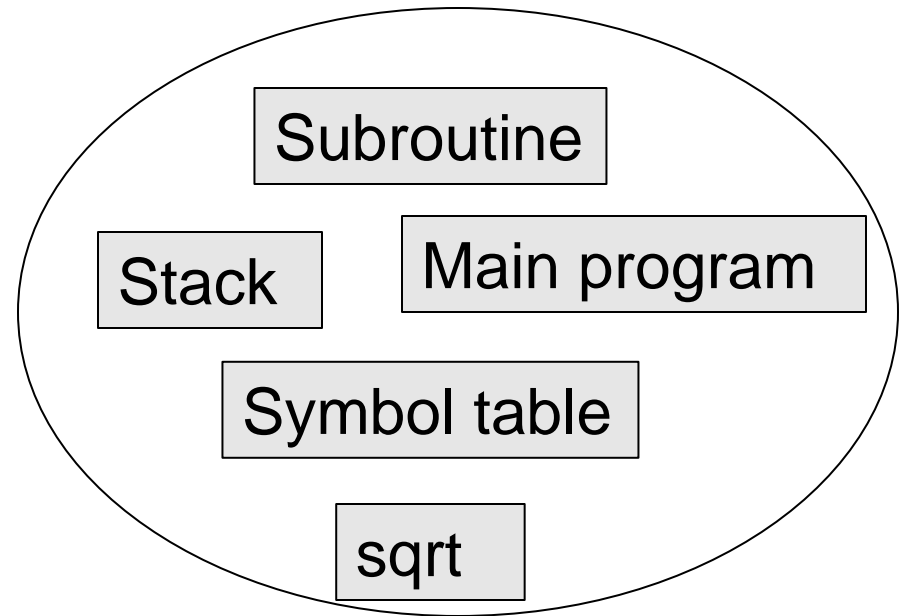
Issues with Simple B&B Method



- **Fragmentation** problem
 - Not every process is the same size
 - Over time, memory space **becomes fragmented**
- Missing support for **sparse address space**
 - Would like to have **multiple chunks per program**
 - Ex: Code, Data, Stack
- **Hard** to do inter-process sharing
 - Want to **share code segments** when possible
 - Want to **share memory between** processes
 - Helped by providing **multiple segments** per process

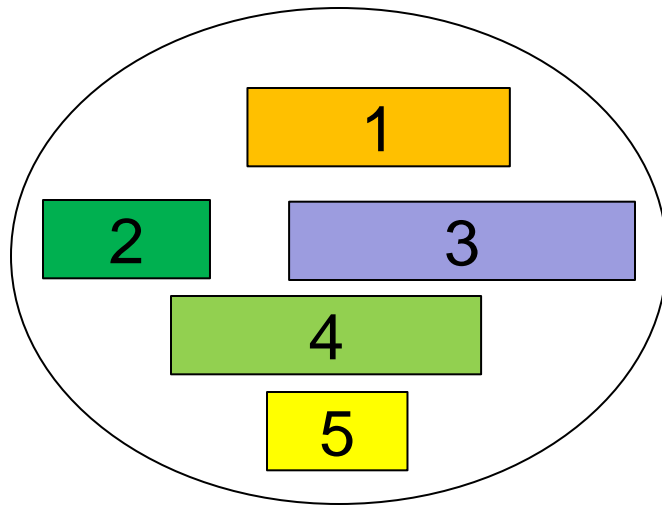
More Flexible Segmentation

- **Logical View:** Multiple separate segments
 - Typical: **Code, Data, Stack**
 - Others: memory sharing, etc
- Each segment is given region of **contiguous memory**
 - Has a **base and limit**
 - Can reside **anywhere** in physical memory

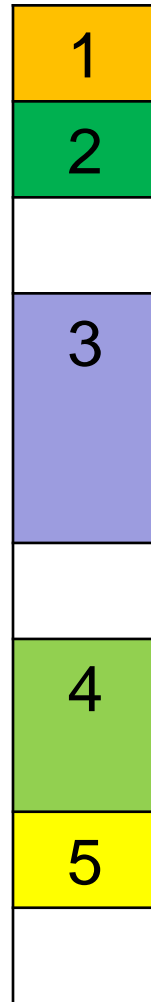


Logical Addresses

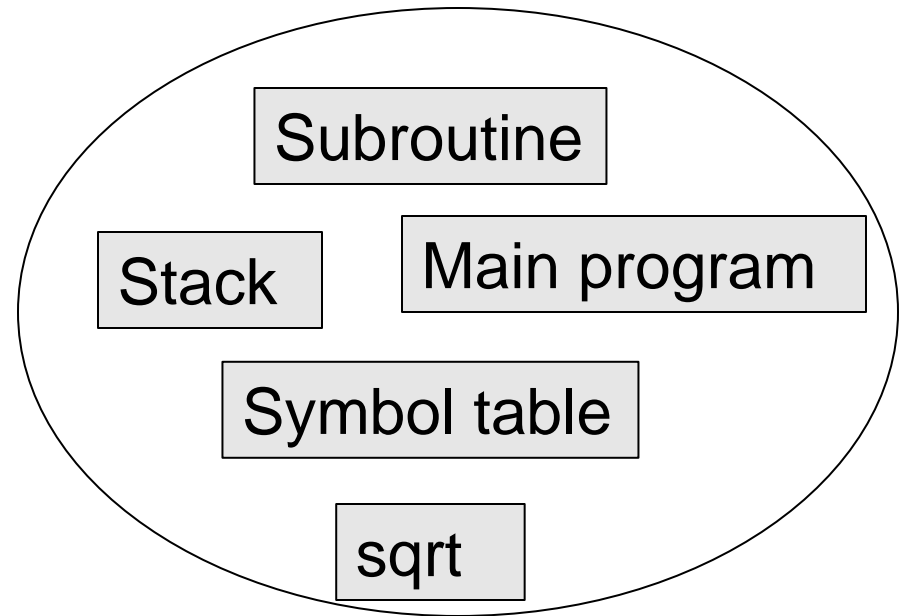
More Flexible Segmentation



User view
of memory space



Physical memory space

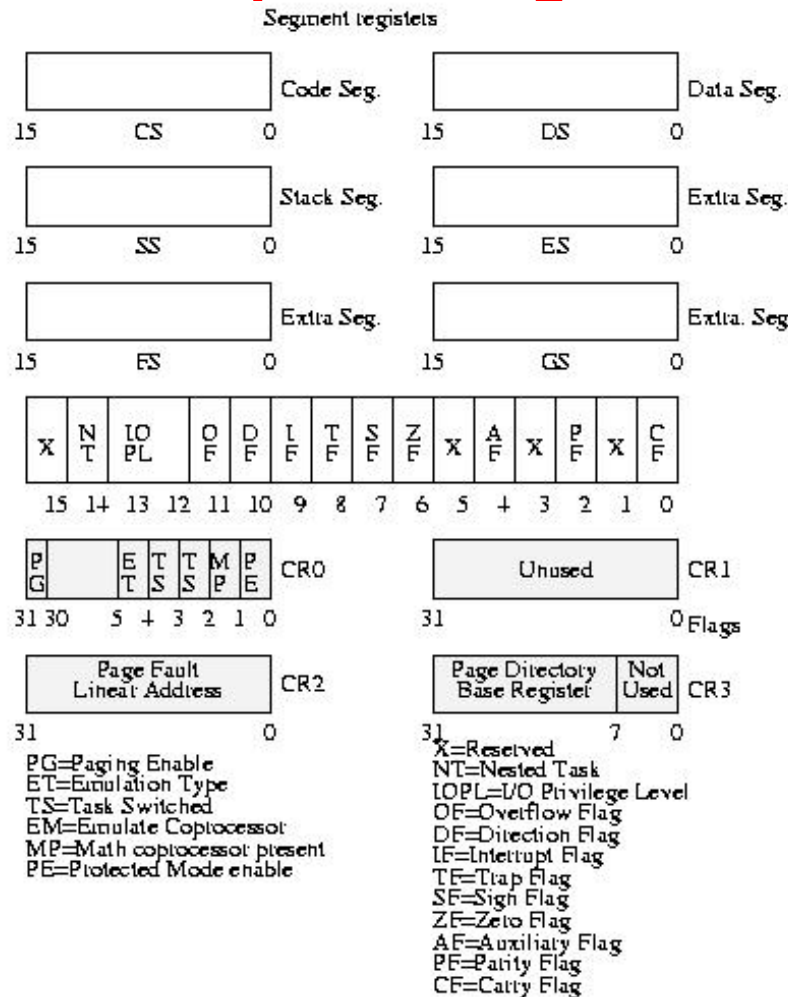


Logical Addresses

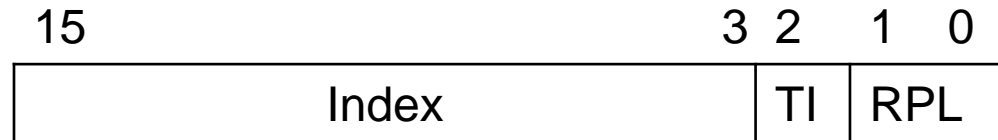
Intel x86 Special Registers



80386 Special Registers



Typical Segment Register Current Priority is RPL of Code Segment (CS)



- RPL: Requestor Privilege Level
- TI: Table Indicator
 - 0 = GDT, 1=LDT
- Index: Index into table
- Protected memory segment selector

Ex: 4 Segments (16b addresses)

Virtual Address Format

Seg	Offset
-----	--------

15 14 13 0
0x0000 SegID=0

0x4000 SegID=1

0x8000

0xC000

Virtual Address Space

0x0000

0x4000

0x4800

0x5C00

0xF000

Physical Address Space

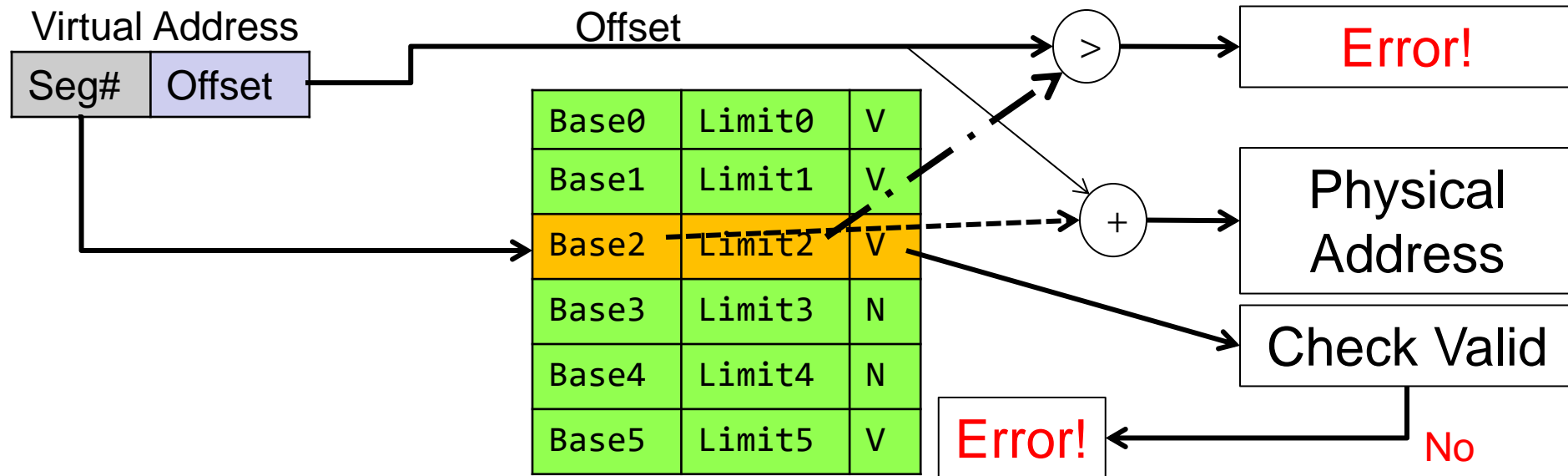
Seg ID#	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000

Might be shared

Space for other apps

Shared with other apps

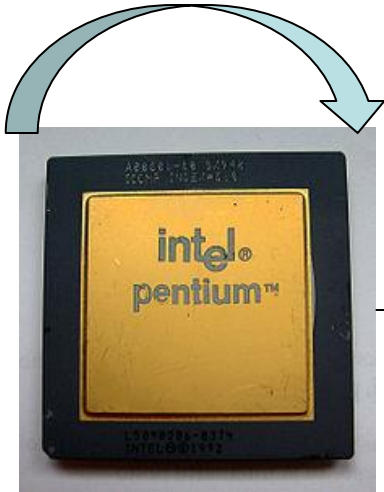
Implementation of Multi-Segment Model



- Segment map resides in processor
 - Segment number mapped into **base/limit pair**
 - Base added to offset to generate physical address
 - Error check** catches offset out of range
- As many chunks of physical memory as entries
 - Segment addressed by **portion of virtual address**
 - However, could be included in instruction instead:
 - x86 Example: `mov [es:bx], ax.`
- What is V/N (valid / not valid)?
 - Can mark segments as invalid; requires check as well

Virtual versus Physical

Virtual addresses



Physical addresses



MMU

Read from
Virtual address

Translate via
Segment Table

Read from
Physical address

Read from
0x240

In: Segment 0
Base: 0x4000

Read from
0x4240

Running more programs than fit in memory: **Swapping**

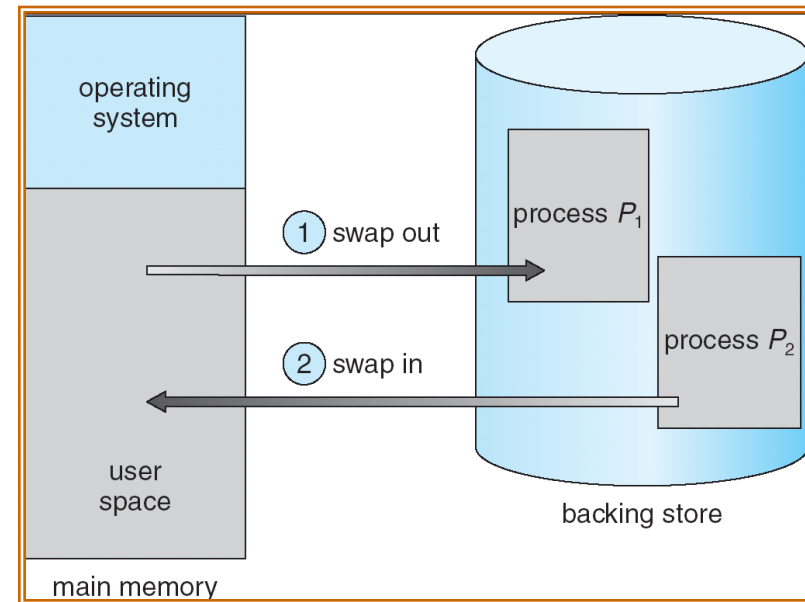
Q: What if not all processes fit in memory?

A: **Swapping**: Extreme form of **Context Switch**

- To make room for next process, some or all the previous **process is moved to disk**
- **Greatly increases** the cost of context switching

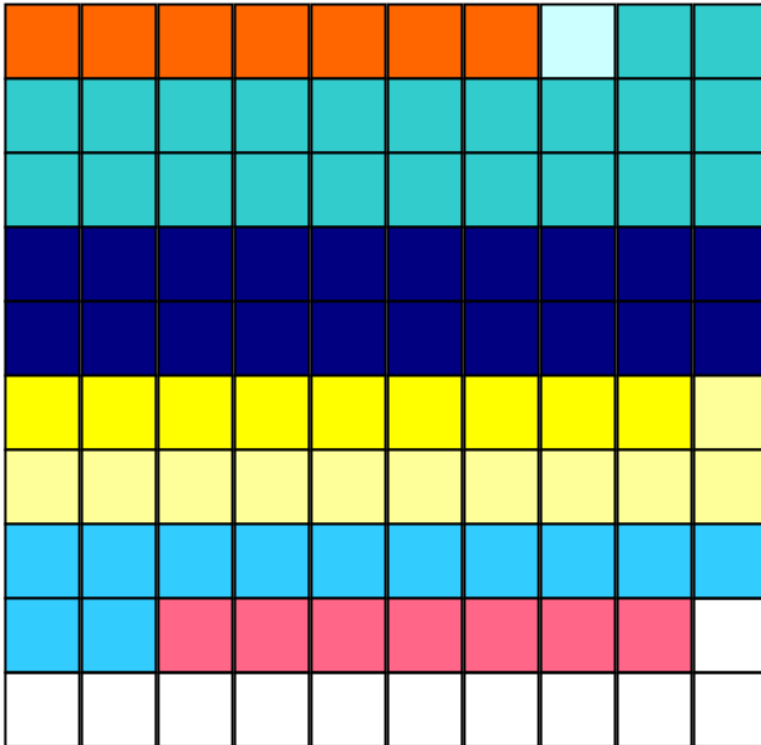
What else could we do?

- Keep only active portions of a process in memory
- Need finer granularity control over physical memory



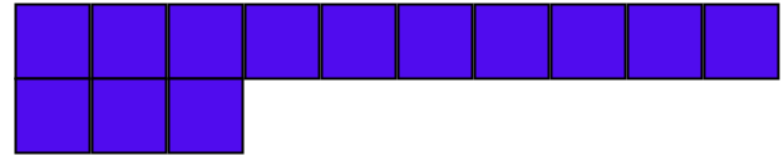
Swapping imagined

In Memory



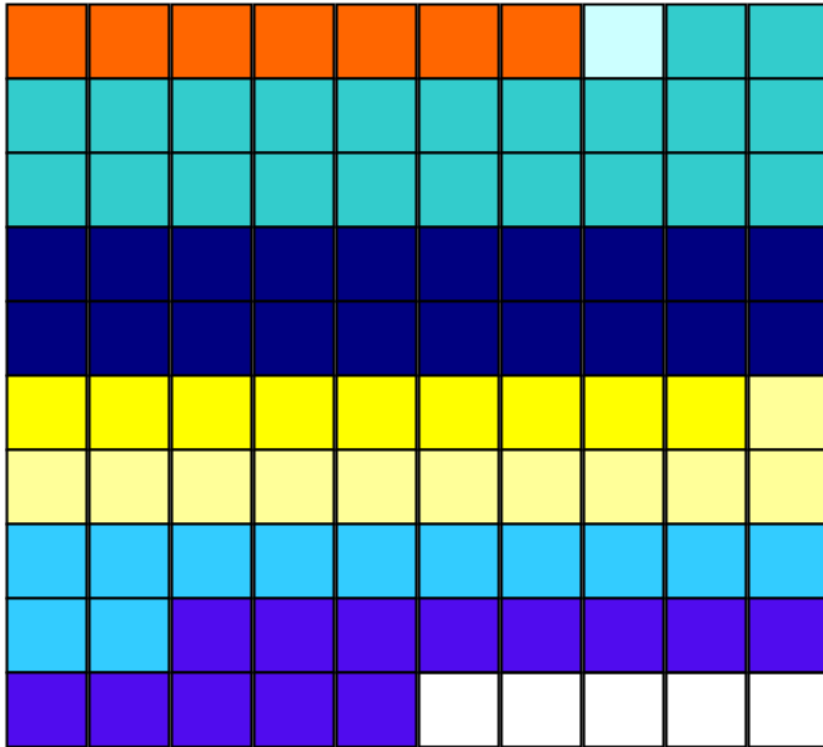
Free = white blocks

Segment to load



Swapping imagined

In Memory

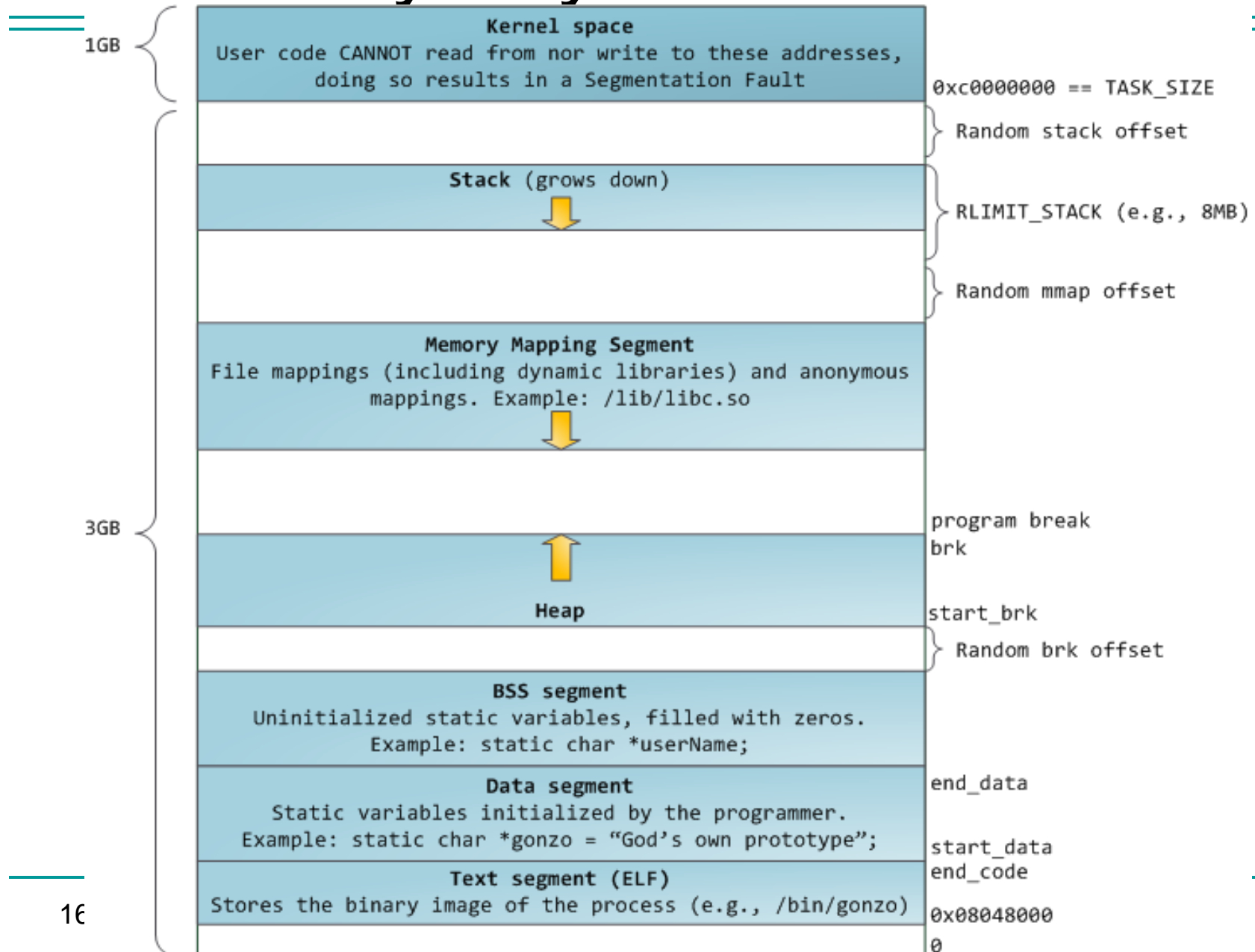


Free = white blocks



Stored on disk

Memory Layout for Linux 32-bit



Problems with Segmentation

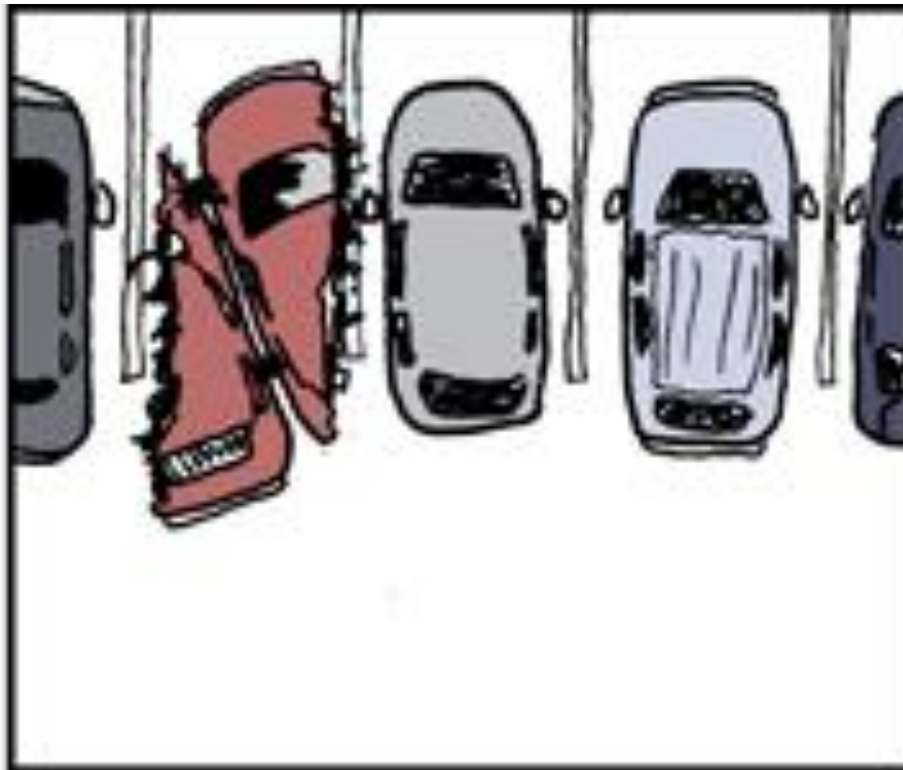
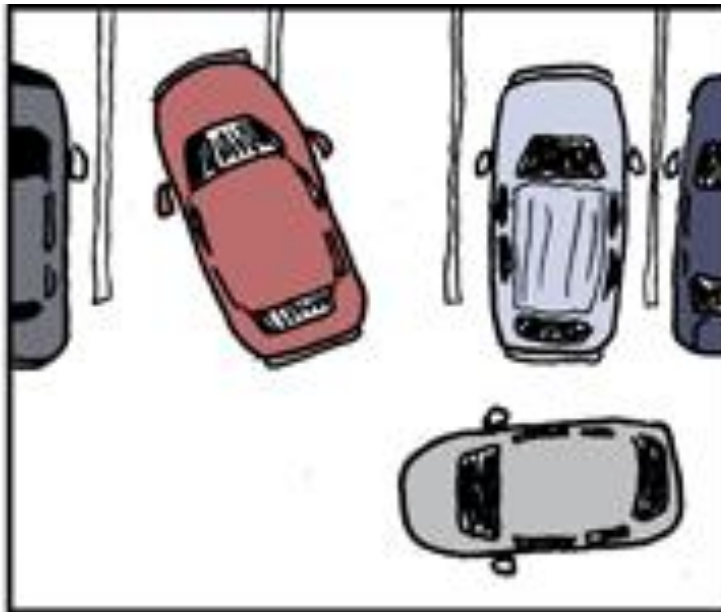
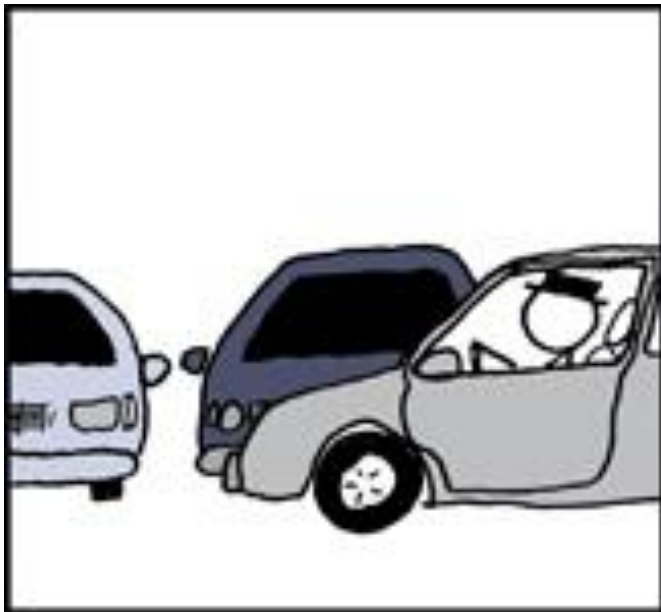
Must fit **variable-sized chunks** into physical memory

May move processes **multiple times** to fit everything

Limited options for **swapping** to disk

Fragmentation: wasted space

- **External**: free gaps between allocated chunks
- **Internal**: don't need all memory within allocated chunks



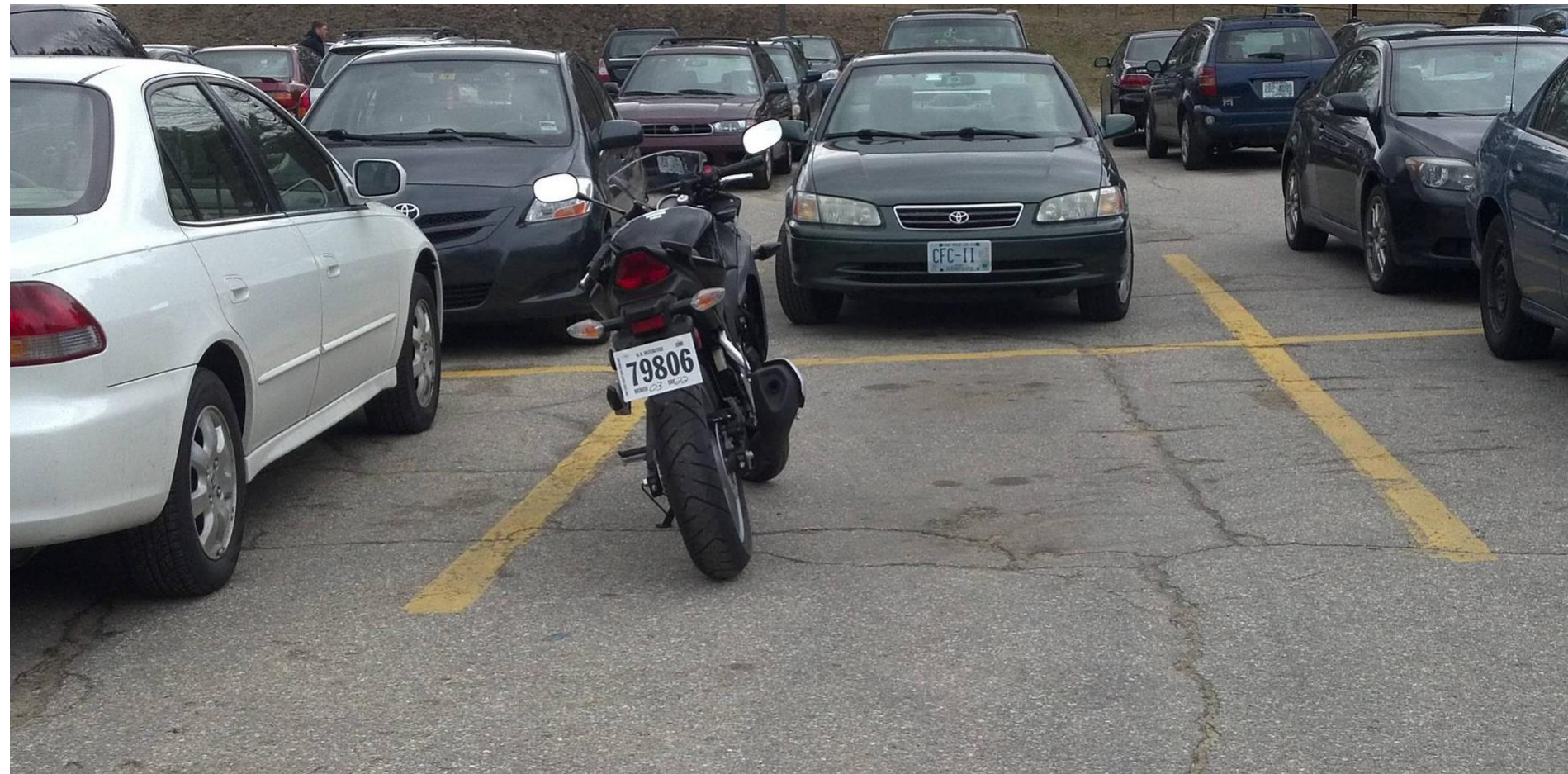


Image source: <http://imgur.com/OFEy7>

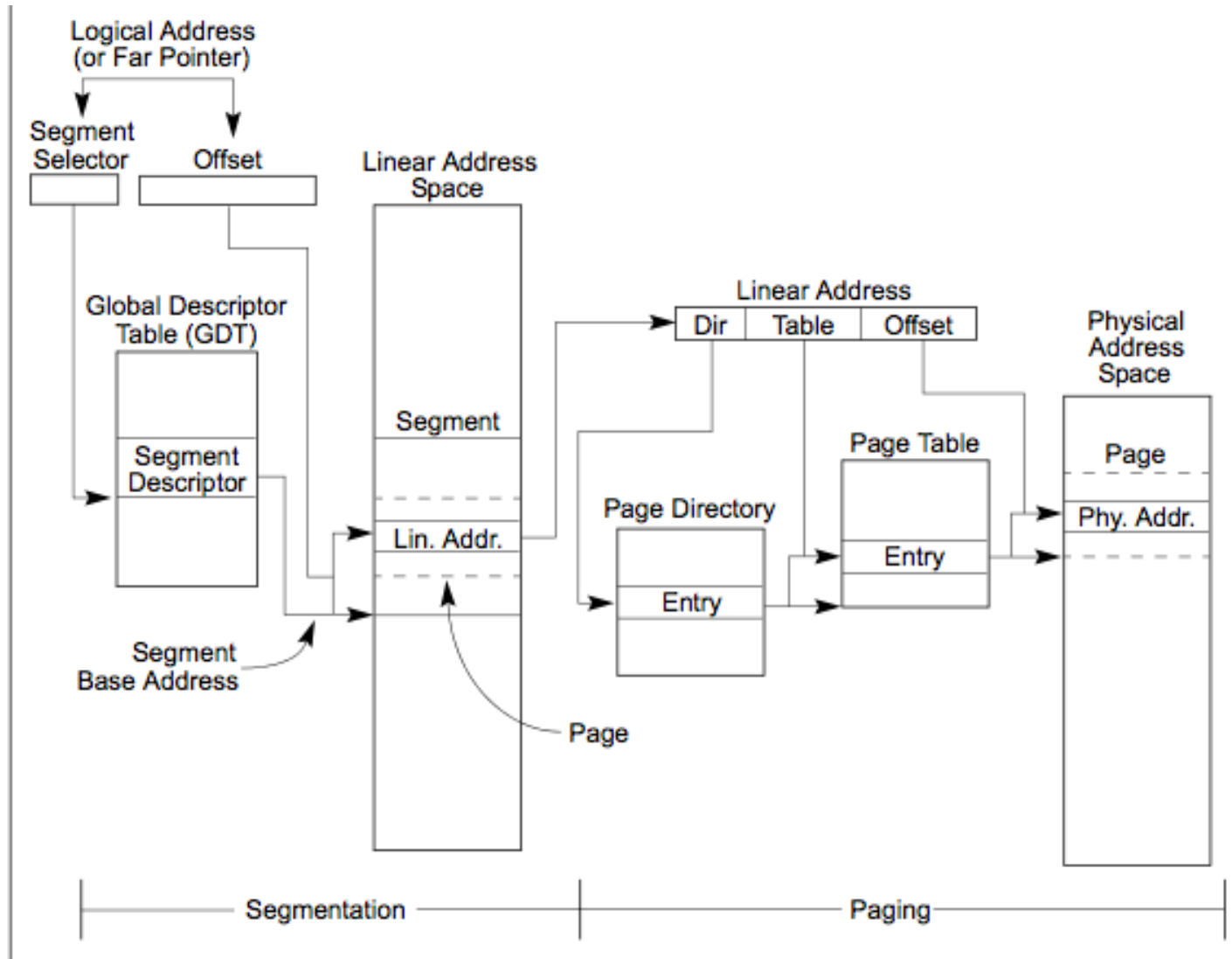
Segmentation Fault

```
user@host:/tmp/segfault$ cat segfault.c
void main() {
    char *str = "Hello, world!";
    *str = 'A';
}
user@host:/tmp/segfault$ gcc segfault.c -o segfault
user@host:/tmp/segfault$ ./segfault
Segmentation fault
neil@snap:/tmp/segfault$
```



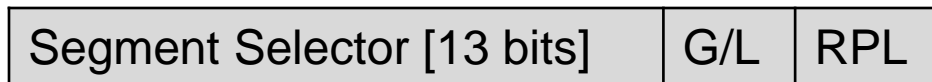
Making it real:

x86 Memory model with segmentation (16/32-bit)



X86 Segment Descriptors (32-bit Protected Mode)

- Segments are either implicit in the instruction (say for code segments) or actually part of the instruction
 - There are 6 registers: **SS, CS, DS, ES, FS, GS**
- What is in a segment register?
 - A *pointer* to the actual segment description:

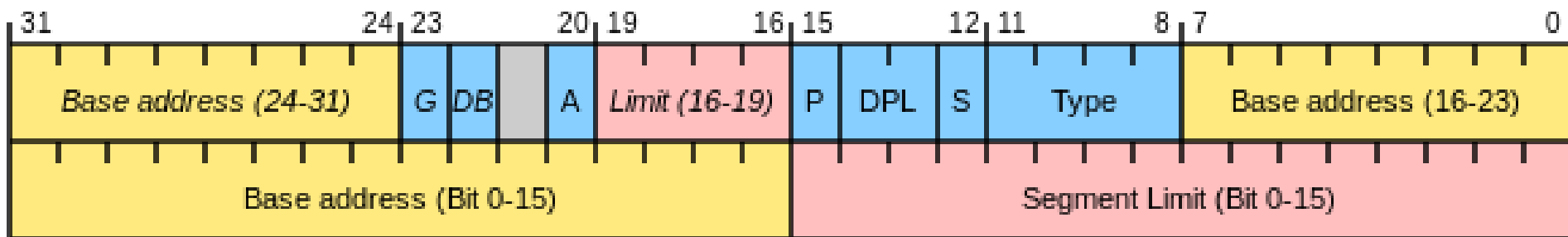


G/L selects between **GDT** and **LDT** tables (global vs local descriptor tables)

- Two registers: **GDTR** and **LDTR** hold pointers to the global and local descriptor tables in memory
 - Includes length of table (for $< 2^{13}$ entries)

X86 Segment Descriptors (32-bit Protected Mode)

- Descriptor format (64 bits):



G: Granularity of segment (0: 16bit, 1: 4KiB unit)

DB: Default operand size (0; 16bit, 1: 32bit)

A: Freely available for use by software

P: Segment present

DPL: Descriptor Privilege Level

S: System Segment (0: System, 1: code or data)

Type: Code, Data, Segment

How are segments used?

- One set of global segments (**GDT**) for everyone, different set of local segments (**LDT**) for every process



In legacy applications (**16-bit mode**):

- Segments provide protection for different components of user programs
- Separate segments for chunks of code, data, stacks
- Limited to **64K** segments

How are segments used?

- One set of global segments (**GDT**) for everyone, different set of local segments (**LDT**) for every process



Modern use in 32-bit Mode:

- Segments “flattened”, i.e. every segment is 4GB in size
- One exception: Use of **GS (or FS)** as a pointer to “**Thread Local Storage**” (**TLS**)
 - A thread can make accesses to TLS like this:
`mov eax, gs(0x0)`

How are segments used?

- One set of global segments (**GDT**) for everyone, different set of local segments (**LDT**) for every process



Modern use in **64-bit** (“long”) mode

- Most segments (SS, CS, DS, ES) have zero base and no length limits
- Only FS and GS retain their functionality (for use in TLS)

So Far

- Address Spaces and Segmentation
 - Loading and Translating
 - Segments
 - Fragmentation
- Page tables

Paging: Physical Memory in Fixed Size Chunks

- Solution to fragmentation from segments?
 - Allocate physical memory in fixed size chunks (**pages**)
 - Every chunk of physical memory is **equivalent**
 - Can use simple vector of bits to handle allocation:
00110001110001101 ... 110010
 - Each bit represents page of physical memory
1 \Rightarrow allocated, 0 \Rightarrow free
- Should pages be as big as our previous segments?
 - **No**: Can lead to lots of internal fragmentation
 - Typically have small pages (1K-16K)
 - **Consequently**: Need multiple pages/segment

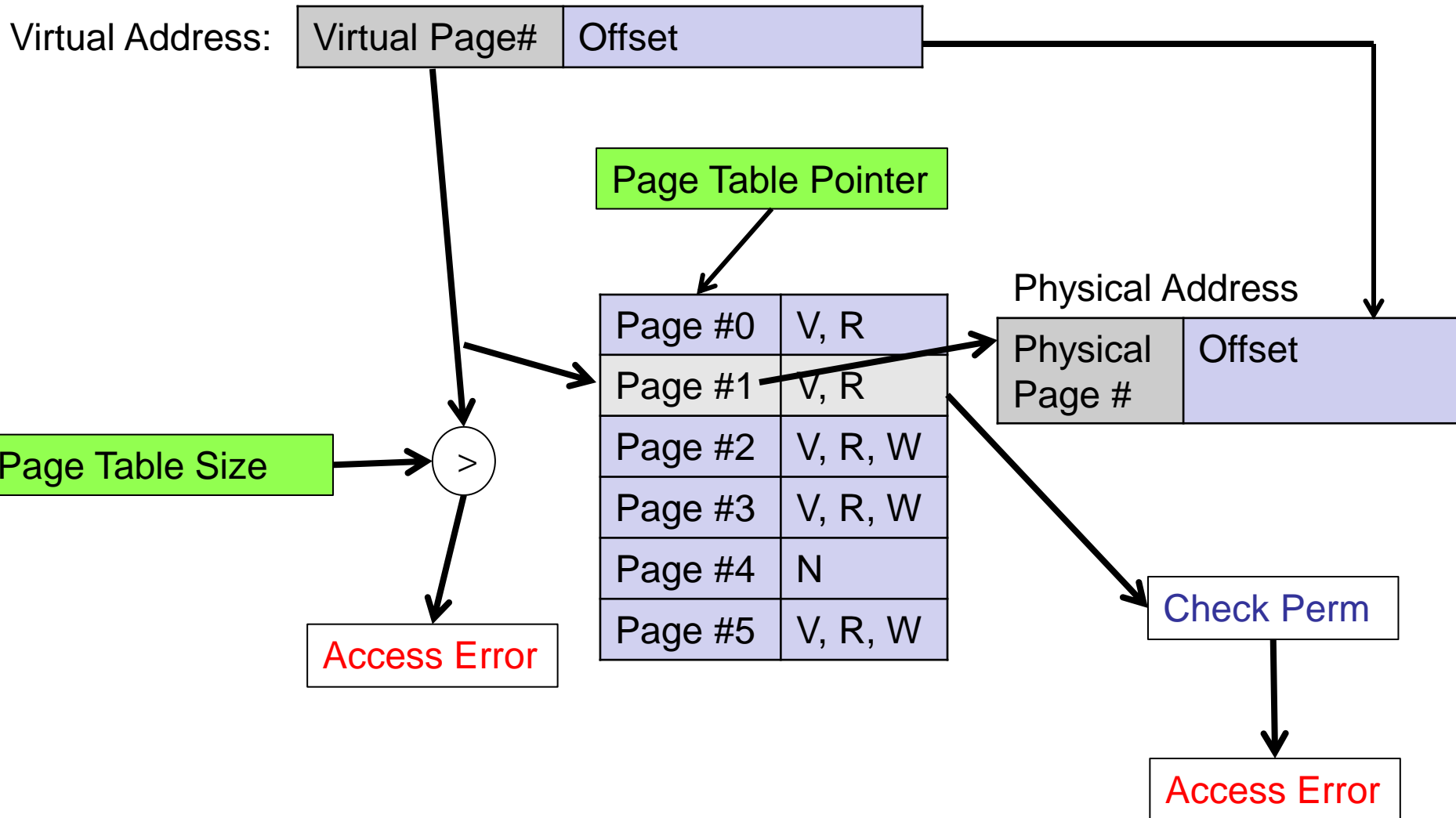
How to Implement Paging?

- Page Table** (One per process)
- Resides in **physical memory**
 - Contains physical page and permission for each virtual page
 - Permissions include: Valid bits, Read, Write, etc

Virtual address mapping

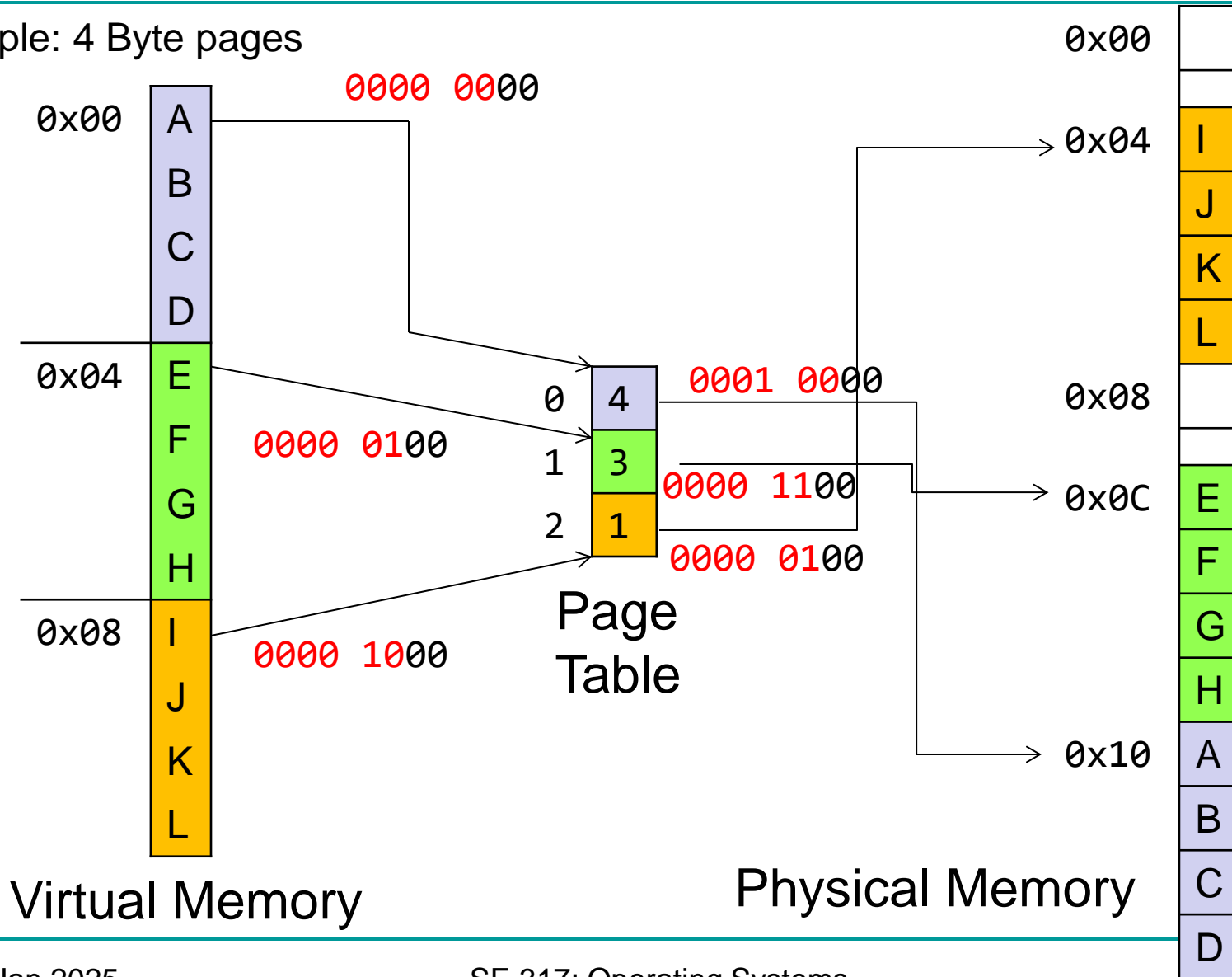
- Offset from Virtual address copied to Physical Address
 - Example: 10 bit offset \Rightarrow 1024 byte pages
- Virtual page # is all remaining bits
 - Example for 32-bits: $32 - 10 = 22$ bits, (4 million entries)
 - Physical page # copied from table into physical address
- **Check Page Table** bounds and permissions

Implementing Paging



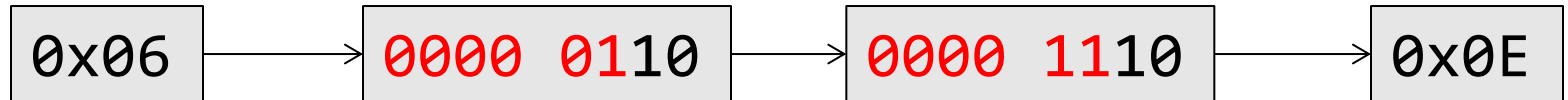
Simple Page Table Example

Example: 4 Byte pages

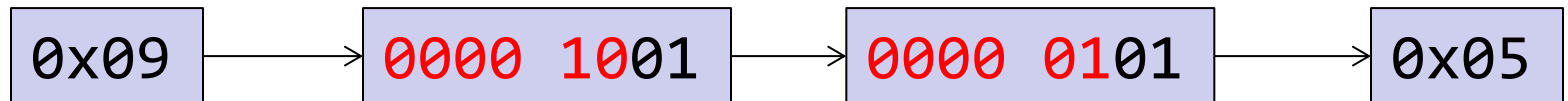


Simple Page Table Example

Virtual Address 1



Virtual Address 2



What about Sharing?

Virtual Address
(Process A):

Virtual Page#	Offset
------------------	--------

Page Table Pointer A

Page #0	V, R
Page #1	V, R
Page #2	V, R, W
Page #3	V, R, W
Page #4	N
Page #5	V, R, W

Page Table Pointer B

Virtual Address
(Process B):

Virtual Page#	Offset
------------------	--------

Page #0	V, R
Page #1	V, R
Page #2	V, R, W
Page #3	V, R, W
Page #4	V, R
Page #5	V, R, W

Shared
Page

This physical page
appears in the
address spaces of
both processes

Virtual Memory View

1111 1111

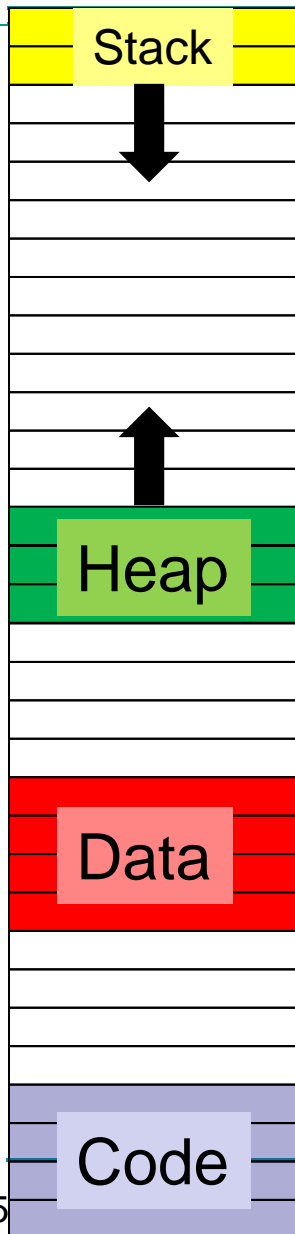
1111 0000

1000 0000

0100 0000

Page # Offset

0000 0000



Page Table

11111	11101
11110	11100
11101	null
11100	null
11011	null
11010	Null
11001	Null
11000	Null
10111	Null
10110	Null
10101	Null
10100	Null
10011	Null
10010	10000
10001	01111
10000	01110
01111	Null
01110	Null
01101	Null
01100	Null
01011	01101
01010	01100
01001	01011
01000	01010
00111	Null
00110	Null
00101	Null
00100	Null
00011	00101
00010	00100
00001	00011
00000	00010

Physical Memory View

1110 1111

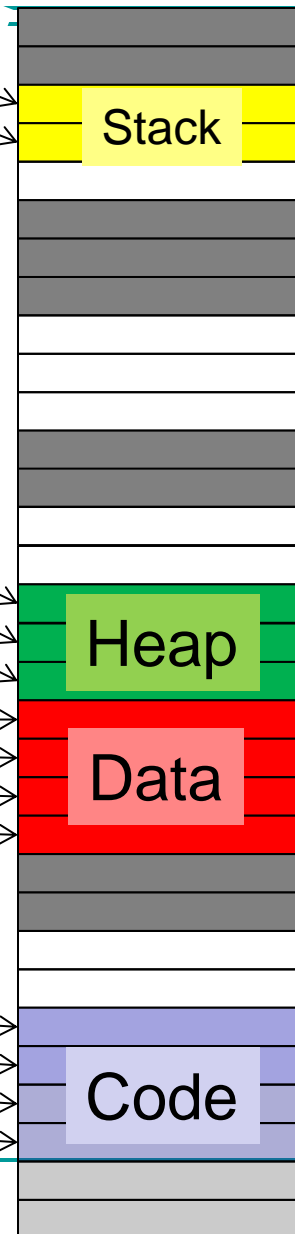
1110 0000

0111 0000

0101 0000

0001 0000

0000 0000



Virtual Memory View

1111 1111

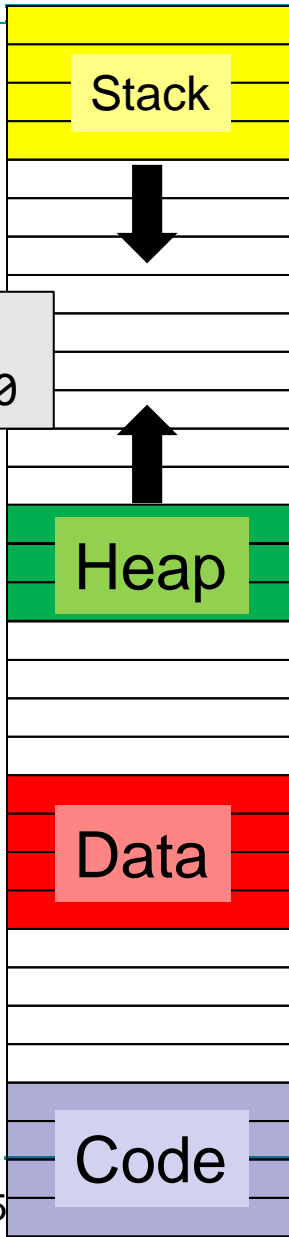
1110 0000

1000 0000

0100 0000

Page # Offset

0000 0000



Page Table

11111	11101
11110	11100
11101	null
11100	null
11011	null
11010	Null
11001	Null
11000	Null
10111	Null
10110	Null
10101	Null
10100	Null
10011	Null
10010	10000
10001	01111
10000	01110
01111	Null
01110	Null
01101	Null
01100	Null
01011	01101
01010	01100
01001	01011
01000	01010
00111	Null
00110	Null
00101	Null
00100	Null
00011	00101
00010	00100
00001	00011
00000	00010

Physical Memory View

1110 1111

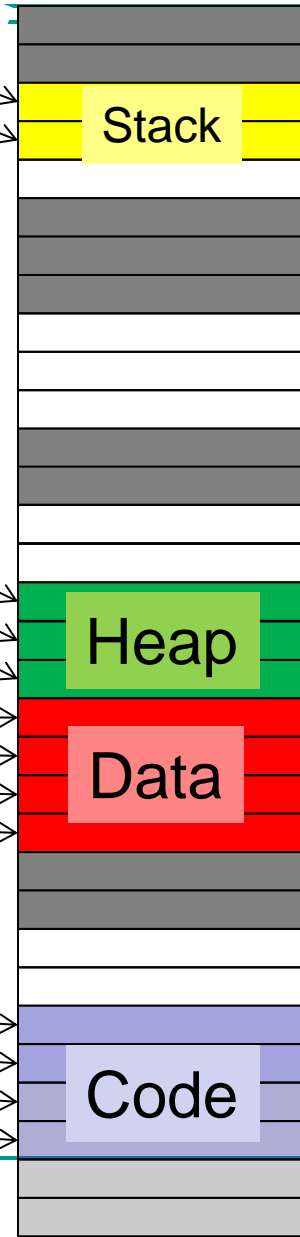
1110 0000

0111 0000

0101 0000

0001 0000

0000 0000



Virtual Memory View

1111 1111

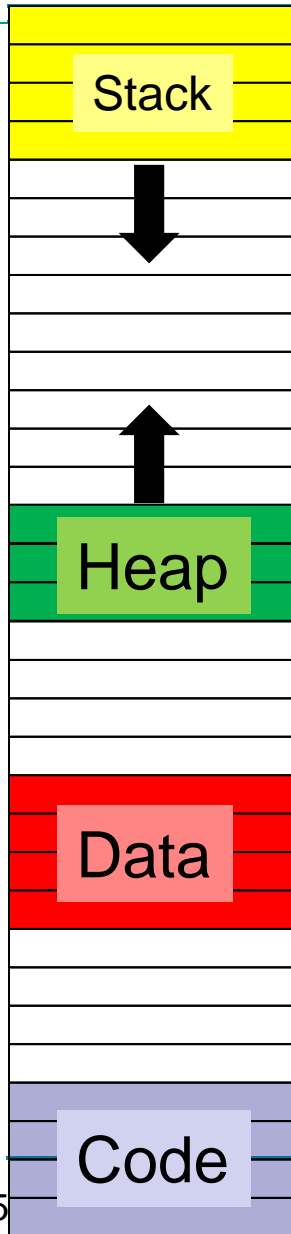
1110 0000

1000 0000

0100 0000

Page # Offset

0000 0000



Page Table

11111	11101
11110	11100
11101	10111
11100	10110
11011	Null
11010	Null
11001	Null
11000	Null
10111	Null
10110	Null
10101	Null
10100	Null
10011	Null
10010	10000
10001	01111
10000	01110
01111	Null
01110	Null
01101	Null
01100	Null
01011	01101
01010	01100
01001	01011
01000	01010
00111	Null
00110	Null
00101	Null
00100	Null
00011	00101
00010	00100
00001	00011
00000	00010

Physical Memory View

1110 1111

1110 0000

0111 0000

0101 0000

0001 0000

0000 0000






Challenge: Table size equal to the **total number of pages** in virtual memory



Image source: <https://www.walldevil.com/2261-artwork-chairs-giant-tables.html>

Page Table Discussion

- What needs to be **switched** on a **context switch**?
 - Page table pointer and limit
- Analysis
 -  – **Pros**
 - Simple memory allocation
 - Easy to Share
 -  – **Con**: What if address space is **sparse**?
 - E.g. on UNIX, **code** starts at 0, **stack** starts at $(2^{31} - 1)$.
 - With 1K pages, need 2 million page table entries!
 -  – **Con**: What if table is **really big**?
 - Not all pages used all the time \Rightarrow would be nice to have a **working set** of page table in memory
- How about **combining paging and segmentation**?
 - **Segments with pages** inside them?
 - Need some sort of **multi-level translation**

What is in a Page Table Entry?

- What is in a **Page Table Entry** (or PTE)?
 - Pointer to next-level page table or to actual page
 - Permission bits: valid, read-only, read-write, write-only
- **Example:** Intel x86 architecture PTE:
 - Address in 10, 10, 12-bit offset format
 - Intermediate page tables called “Directories”

Page Frame Number (Physical Page Number)	Free (OS)	0	L	D	A	PCD	PWT	U	W	P
31-12	11-9	8	7	6	5	4	3	2	1	0

What is in a Page Table Entry?

Page Frame Number (Physical Page Number)	Free (OS)	0	L	D	A	PCD	PWT	U	W	P
31-12	11-9	8	7	6	5	4	3	2	1	0

P: Present (same as “valid” bit in other architectures)

W: Writeable

U: User accessible

PWT: Page write transparent: external cache write-through

PCD: Page cache disabled (page cannot be cached)

A: Accessed: page has been accessed recently

D: Dirty (PTE only): page has been modified recently

L: L=1 \Rightarrow 4MB page (directory only).

Bottom 22 bits of virtual address serve as offset

Examples of how to use a PTE

- How do we use the PTE?
 - **Invalid PTE** can imply different things:
 - Region of address space is **actually invalid**
 - or**
 - Page/directory is just somewhere else than memory (load it?)
 - Validity checked first
 - OS can use other (say) 31 bits for location info
- **Usage Example: Demand Paging**
 - Keep only active pages in memory
 - Place others on disk and mark their PTEs invalid

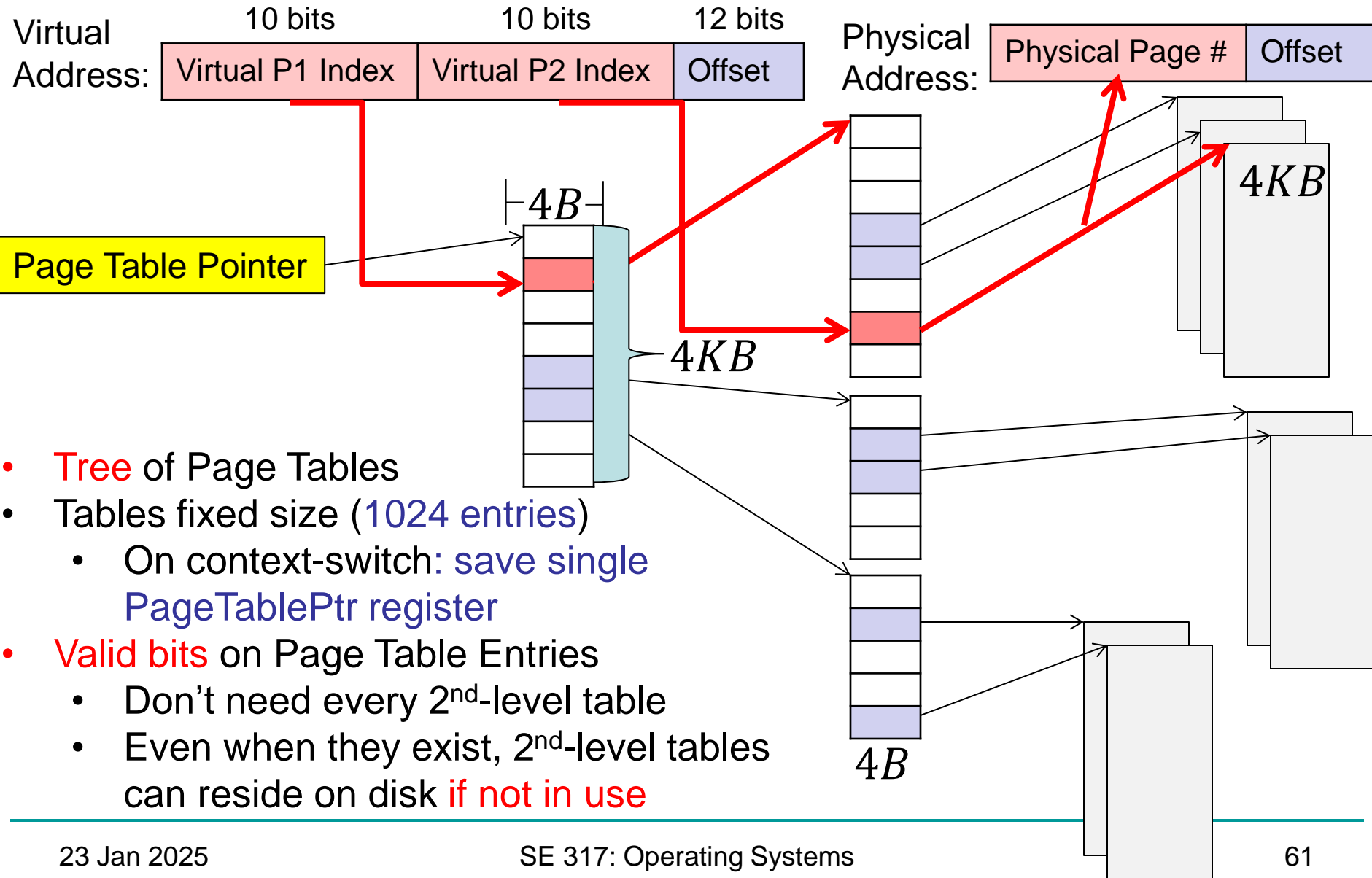
Examples of how to use a PTE

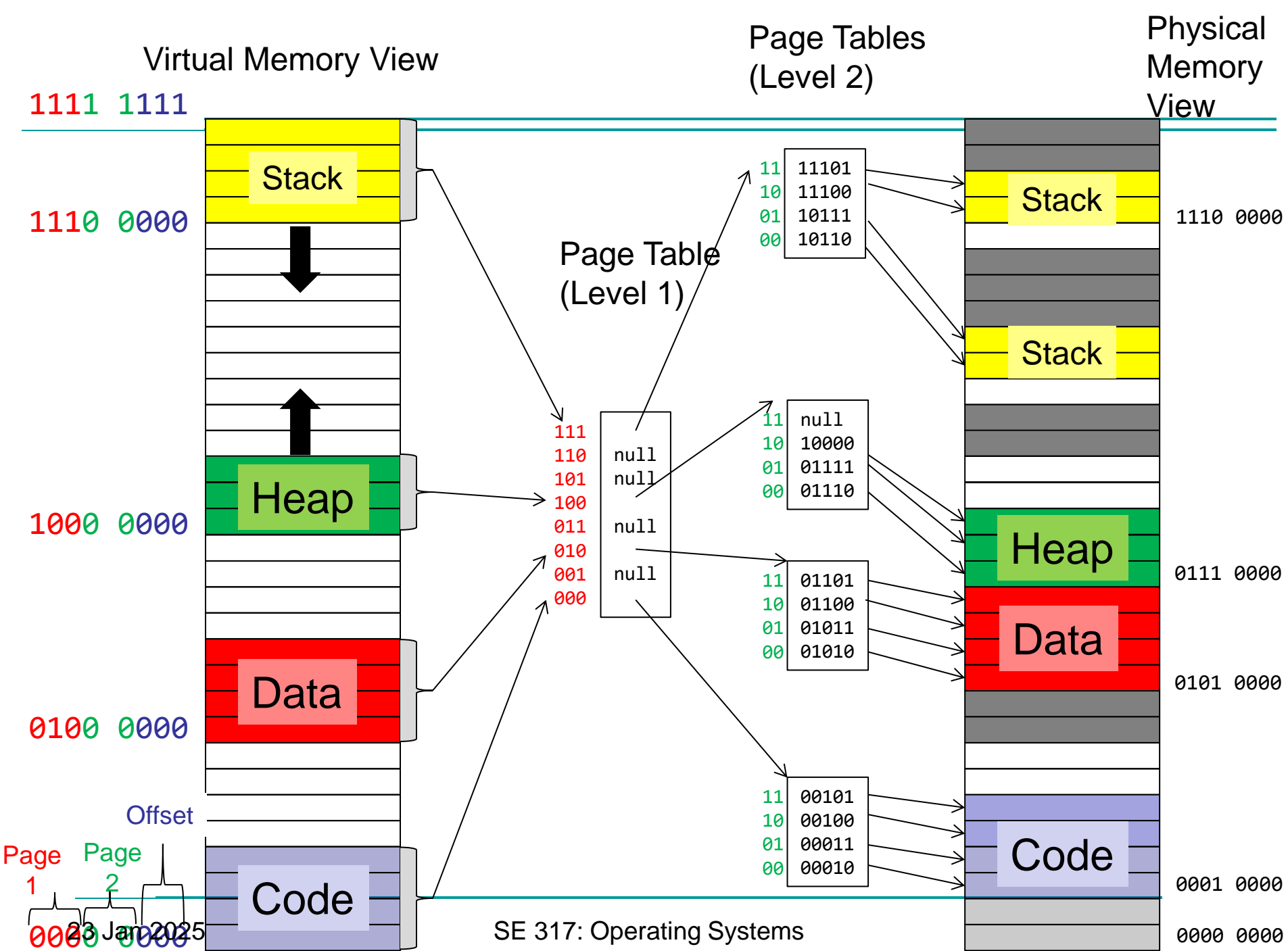
- **Usage Example: Copy on Write**
 - UNIX fork gives *copy* of parent address space to child
 - Address spaces disconnected after child created
 - How to do this cheaply?
 - Make copy of parent's page tables (point at same memory)
 - Mark entries in both sets of page tables as read-only
 - Page fault on write creates two copies
- **Usage Example: Zero Fill On Demand**
 - New data pages must carry no information (say be zeroed)
 - Mark PTEs as invalid; page fault on use gets zeroed page
 - Often, OS creates zeroed pages in background

How can we make the page table closer to the memory space we actually need?



Fix for sparse address space: The two-level page table

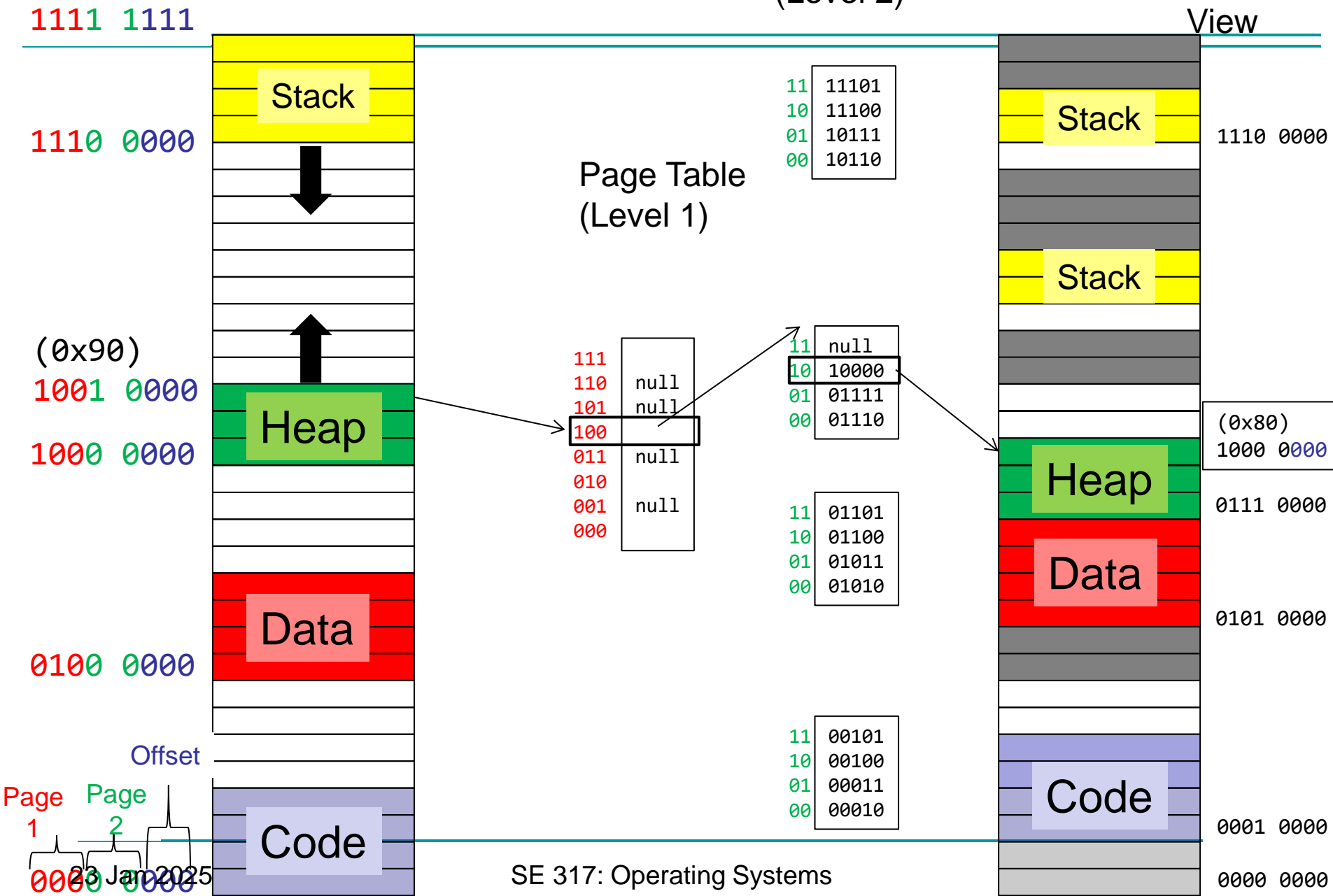




Virtual Memory View

Page Tables (Level 2)

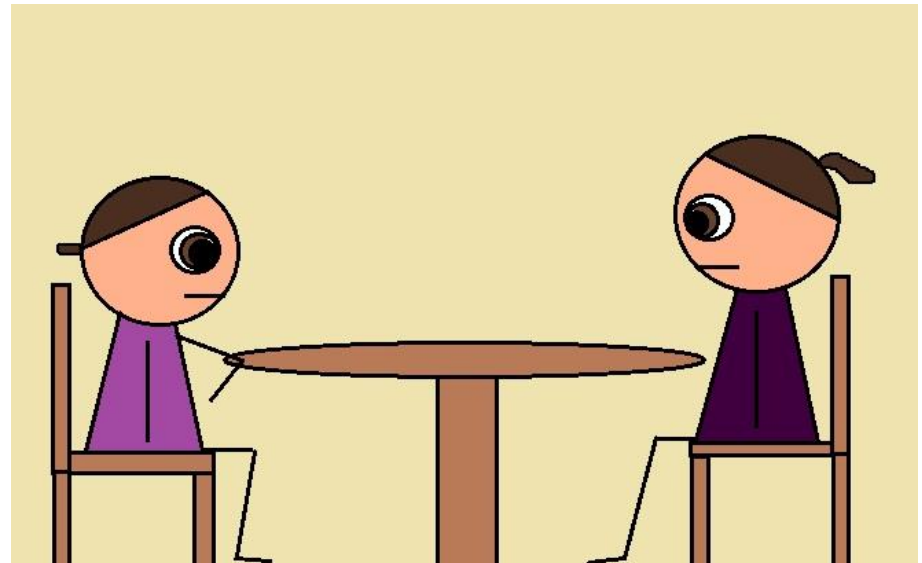
Physical Memory View



Result

In best case, total size of page tables \approx number of pages **used** by program **virtual memory**.

Requires two additional memory access!



Conclusion

- Address Spaces and Segmentation
 - Loading and Translating
 - Segments
 - Fragmentation
- Page tables