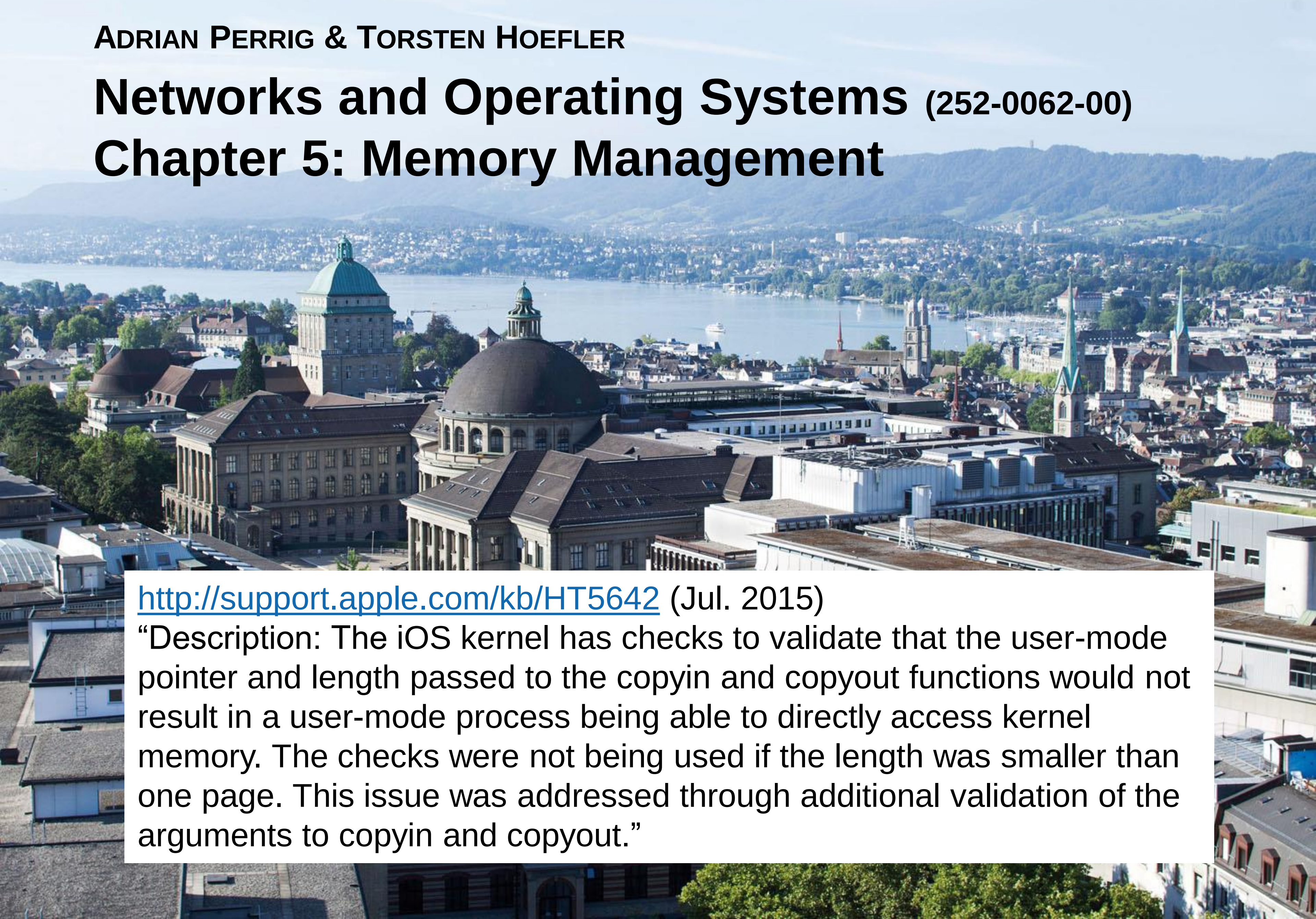


ADRIAN PERRIG & TORSTEN HOEFLER

Networks and Operating Systems (252-0062-00)
Chapter 5: Memory Management

http://support.apple.com/kb/HT5642 (Jul. 2015)
"Description: The iOS kernel has checks to validate that the user-mode pointer and length passed to the copyin and copyout functions would not result in a user-mode process being able to directly access kernel memory. The checks were not being used if the length was smaller than one page. This issue was addressed through additional validation of the arguments to copyin and copyout."

Oldskool: `signal()`

```
void (*signal(int sig, void (*handler)(int))) (int);
```

- **Unpacking this:**
  - A handler looks like
    `void my_handler(int);`
  - Signal takes two arguments…
    *An integer (the signal type, e.g., **`SIGPIPE`**)*
    A pointer to a handler function
  - … and returns a pointer to a handler function
    The previous handler,

- **“Special” `handler` arguments:**
  - **`SIG_IGN`** (ignore), **`SIG_DFL`** (default), **`SIG_ERR`** (error code)

Unix signal handlers

- **Signal handler can be called at *any time!***
- **Executes on the current user stack**
  - If process is in kernel, may need to retry current system call
  - Can also be set to run on a different (alternate) stack

⇒ User process is in *undefined* state when signal delivered

Implications

- **There is very little you can safely do in a signal handler!**
  - Can’t safely access program global or static variables
  - Some system calls are *re-entrant*, and can be called
  - Many C library calls cannot (including **`_r`** variants!)
  - Can sometimes execute a **`longjmp`** if you are careful
  - With **`signal`**, cannot safely change signal handlers…

- **What happens if another signal arrives?**

Multiple signals

- If multiple signals of the *same* type are to be delivered, Unix will ***discard all but one.***

- If signals of *different* types are to be delivered, Unix will deliver them ***in any order.***

- Serious concurrency problem:
  How to make sense of this?

A better `signal()` POSIX `sigaction()`

```
#include <signal.h>

int sigaction(int signo,
              const struct sigaction *act,
              struct sigaction *oldact);
```

New action for signal **`signo`**

Previous action is returned

```
struct sigaction {
      void (*sa_handler)(int);
      sigset_t    sa_mask;
      int         sa_flags;
      void (*sa_sigaction)(int, siginfo_t *, void *);
};
```

Signal handler

Signals to be blocked in this handler (cf., **`fd_set`**)

More sophisticated signal handler (depending on flags)

Signals as upcalls

- **Particularly specialized (and complex) form of *Upcall***
  - Kernel RPC to user process

- **Other OSes use upcalls much more heavily**
  - Including Barrelfish
  - "Scheduler Activations": dispatch every process using an upcall instead of return

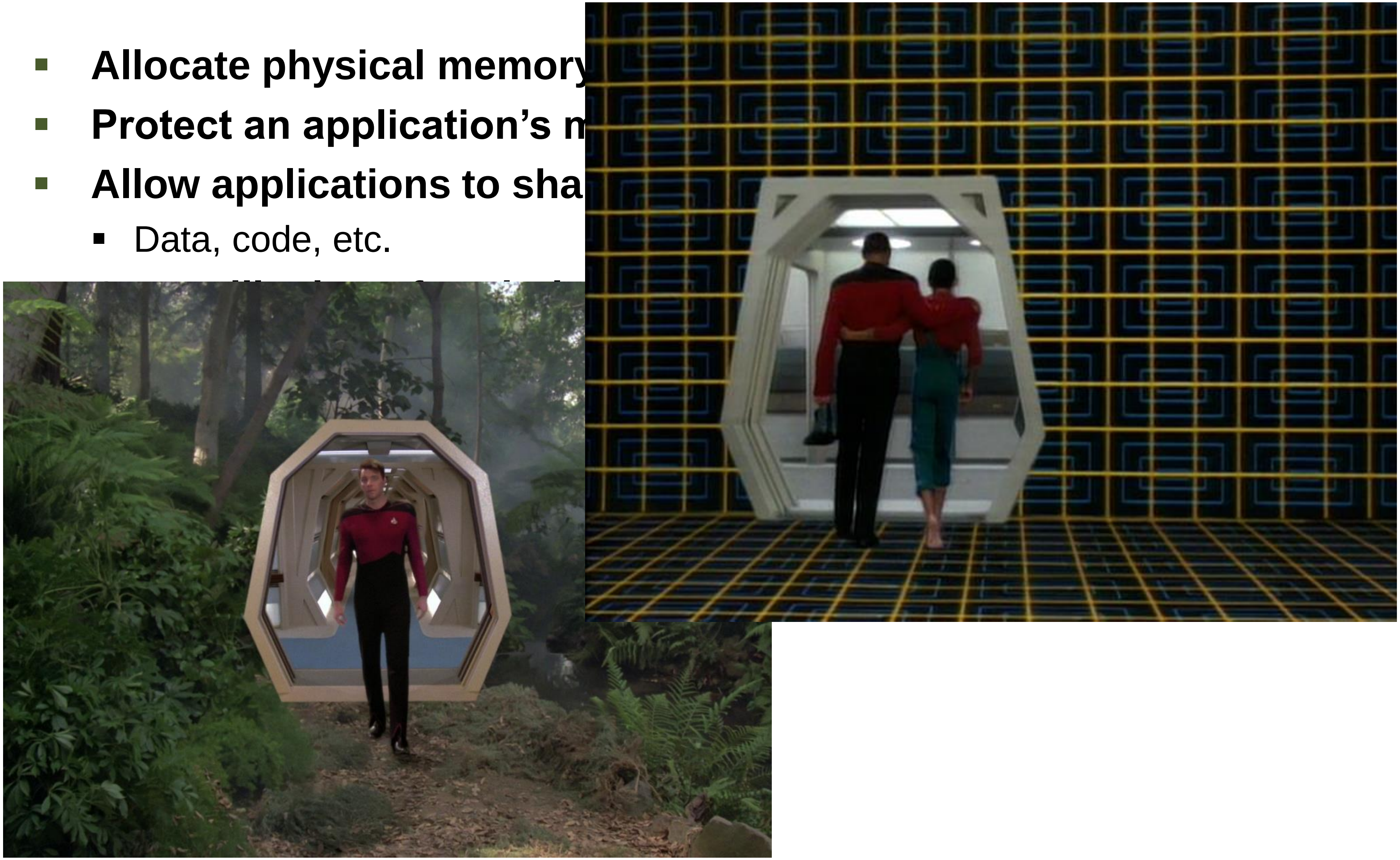
- ***Very* important structuring concept for systems!**

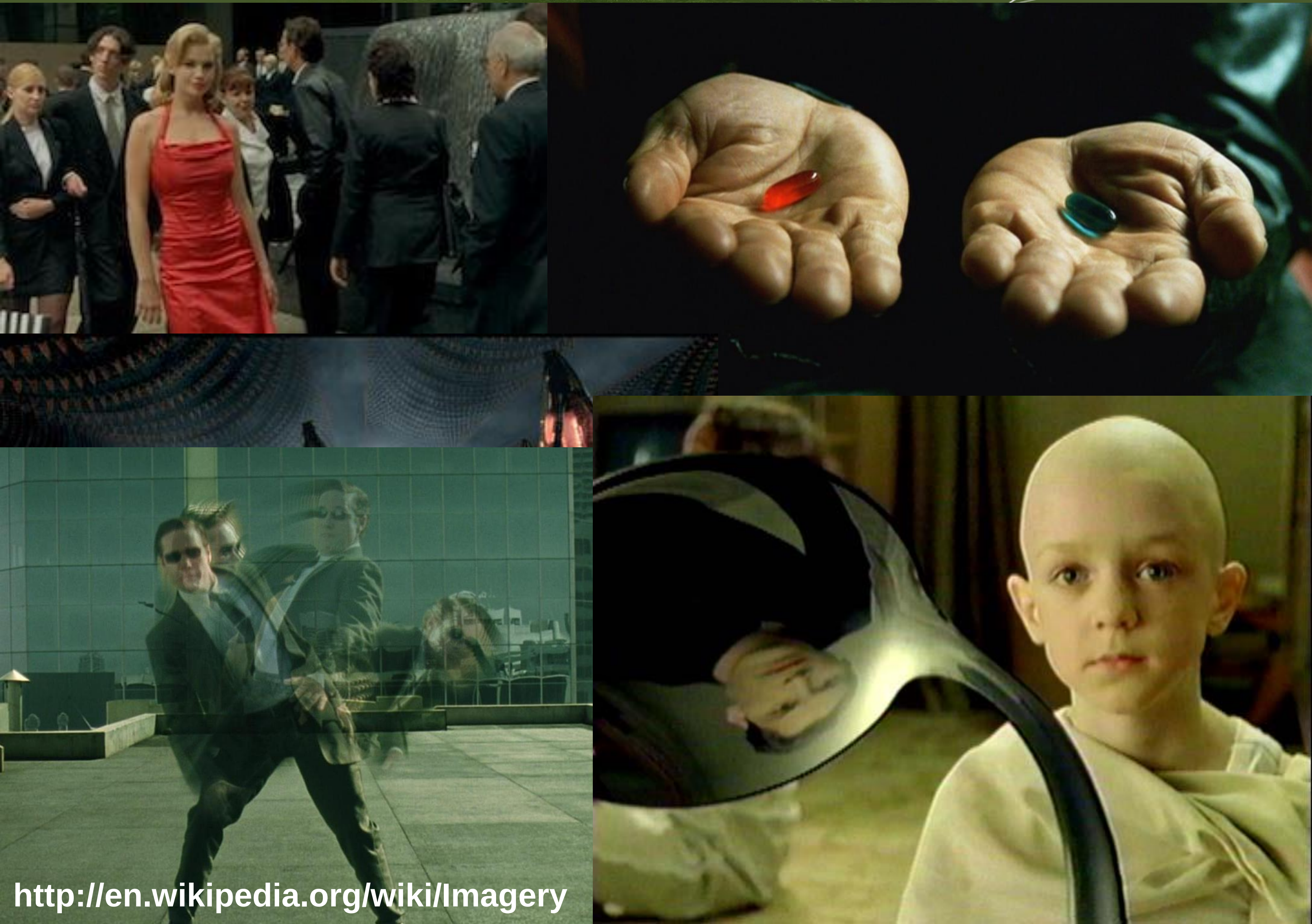
Our Small Quiz

- **True or false (raise hand)**
  - Mutual exclusion on a multicore can be achieved by disabling interrupts
  - Test and set can be used to achieve mutual exclusion
  - Test and set is more powerful than compare and swap
  - The CPU retries load-linked/store conditional instructions after a conflict
  - The best spinning time is 2x the context switch time
  - Priority inheritance can prevent priority inversion
  - The receiver never blocks in asynchronous IPC
  - The sender blocks in synchronous IPC if the receiver is not ready
  - A pipe file descriptor can be sent to a different process
  - Pipes do not guarantee ordering
  - Named pipes in Unix behave like files
  - A process can catch all signals with handlers
  - Signals always trigger actions at the signaled process
  - One can implement a user-level tasking library using signals
  - Signals of the same type are buffered in the kernel

Goals of Memory Management

- **Allocate physical memory**
- **Protect an application's m**
- **Allow applications to sha**
  - Data, code, etc.





http://en.wikipedia.org/wiki/Imagery

In CASP last semester we saw:

- **Assorted uses for virtual memory**
- **x86 paging**
  - Page table format
  - Translation process
  - Translation lookaside buffers (TLBs)
  - Interaction with caches
- **Performance implications**
  - For application code, e.g., matrix multiply

What's new this semester?

- **Wider range of memory management hardware**
  - Base/limit, segmentation
  - Inverted page tables, etc.
- **How the OS uses the hardware**
  - Demand paging and swapping
  - Page replacement algorithms
  - Frame allocation policies

Terminology

- **Physical address: address as seen by the memory unit**
- **Virtual or Logical address: address issued by the processor**
  - Loads
  - Stores
  - Instruction fetches
  - Possible others (e.g., TLB fills)…

Memory management

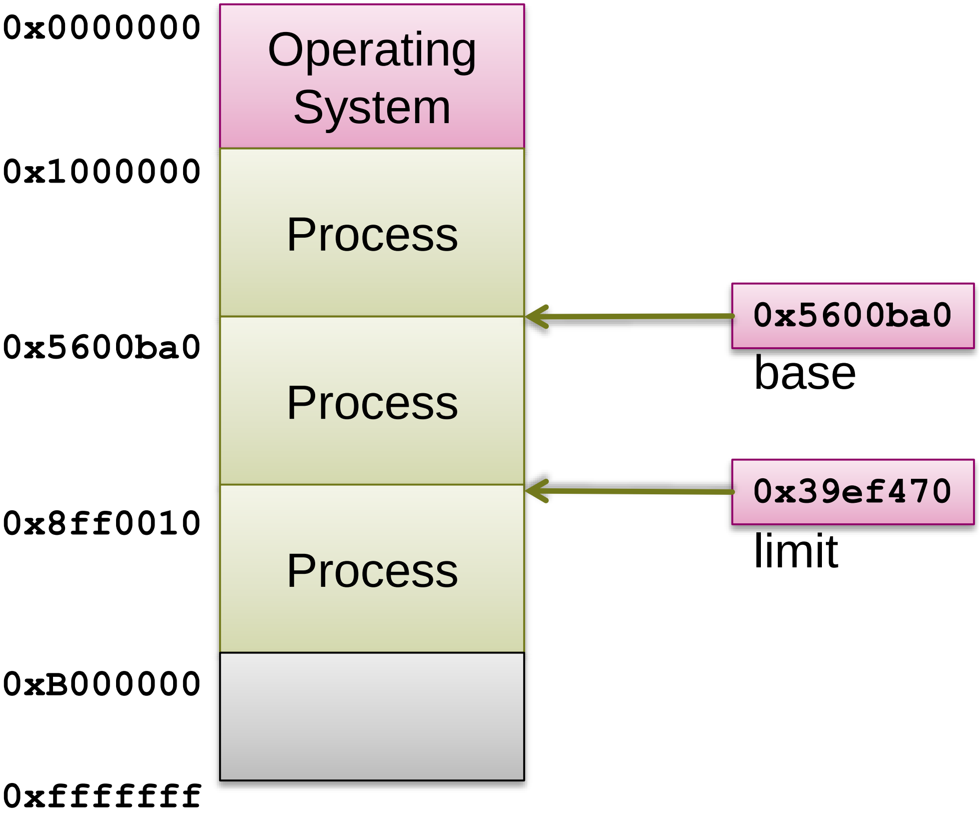
1. **Allocating physical addresses to applications**
2. **Managing the name translation of virtual addresses to physical addresses**
3. **Performing access control on memory access**

- **Functions 2 & 3 usually involve the hardware Memory Management Unit (MMU)**

Simple scheme: partitioned memory

Base and Limit Registers

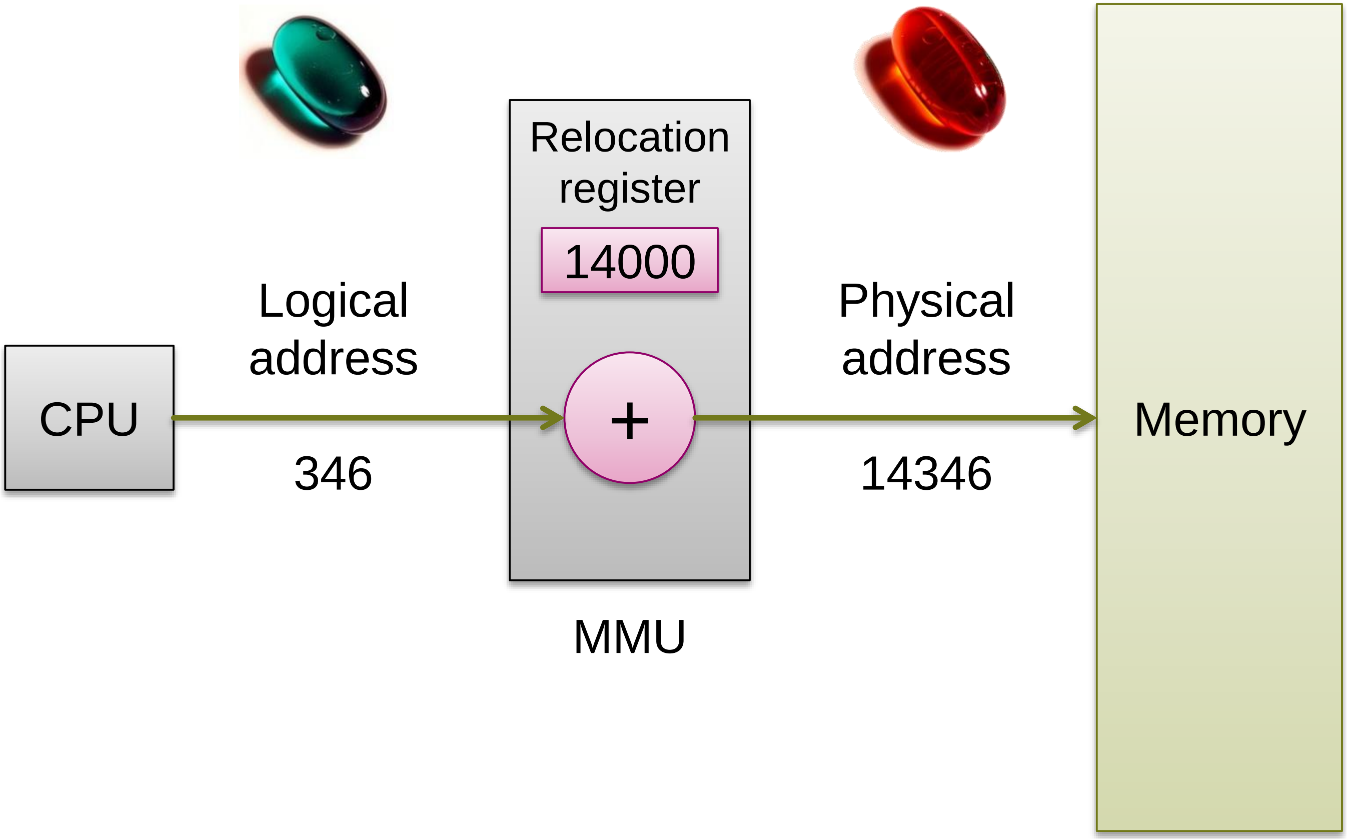
- A pair of base and limit registers define the logical address space



Issue: address binding

- **Base address isn’t known until load time**
- **Options:**
  1. Compiled code must be *completely* **position-independent**, or
  2. **Relocation Register** maps compiled addresses dynamically to physical addresses

Dynamic relocation using a relocation register

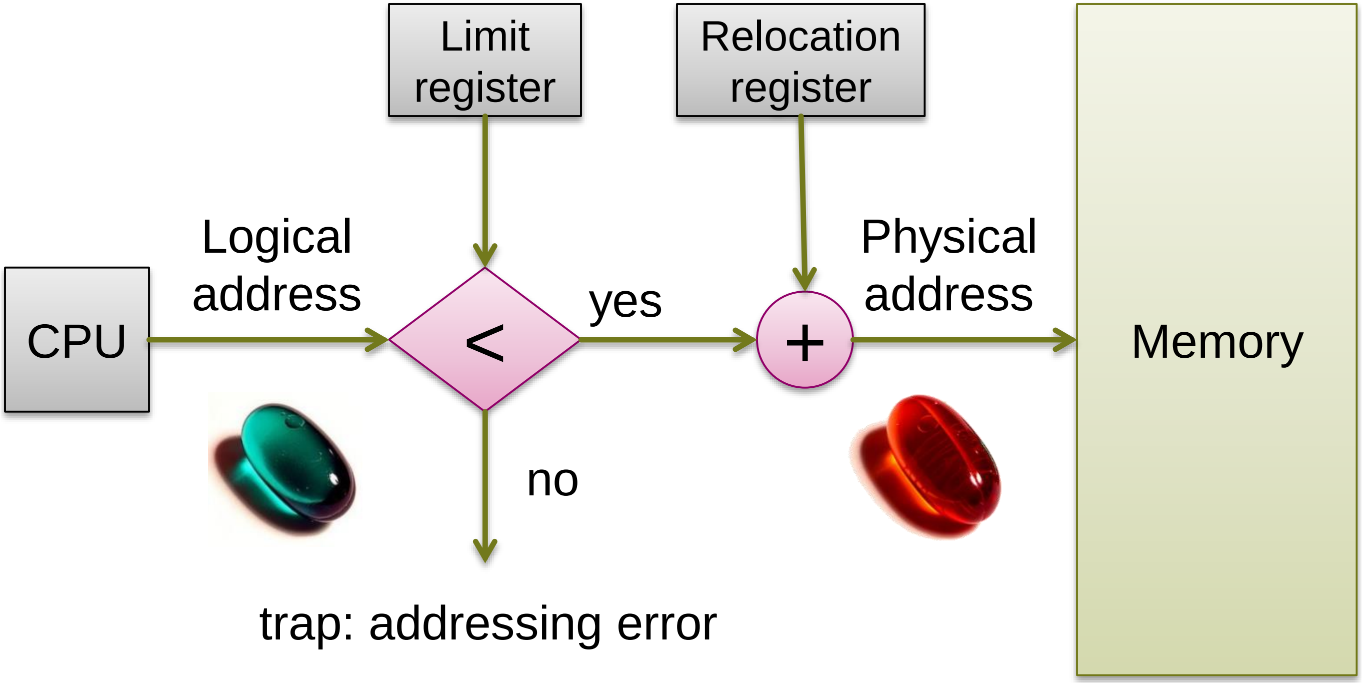


Contiguous allocation

- **Main memory usually into two partitions:**
  - Resident OS, usually in low memory with interrupt vector
  - User processes in high memory
- **Relocation registers protect user processes from**
  1. each other
  2. changing operating-system code and data
- **Registers:**
  - ***Base register*** contains value of smallest physical address
  - ***Limit register*** contains range of logical addresses

    each logical address must be less than the limit register
  - MMU maps logical address dynamically

Hardware Support for Relocation and Limit Registers



Base & Limit summary

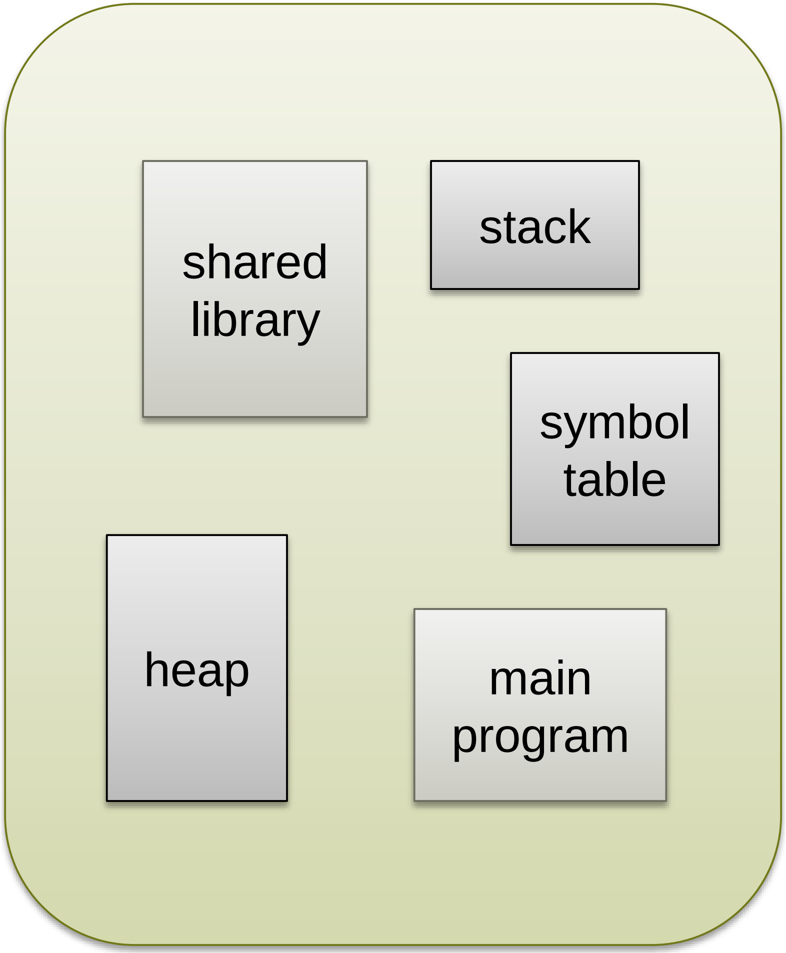
- **Simple to implement (addition & compare)**
- **Physical memory fragmentation**
- **Only a single contiguous address range**
  - How to share data between applications?
  - How to share program text (code)?
  - How to load code dynamically?
  - Total logical address space ≤ physical memory

Segmentation

Segmentation

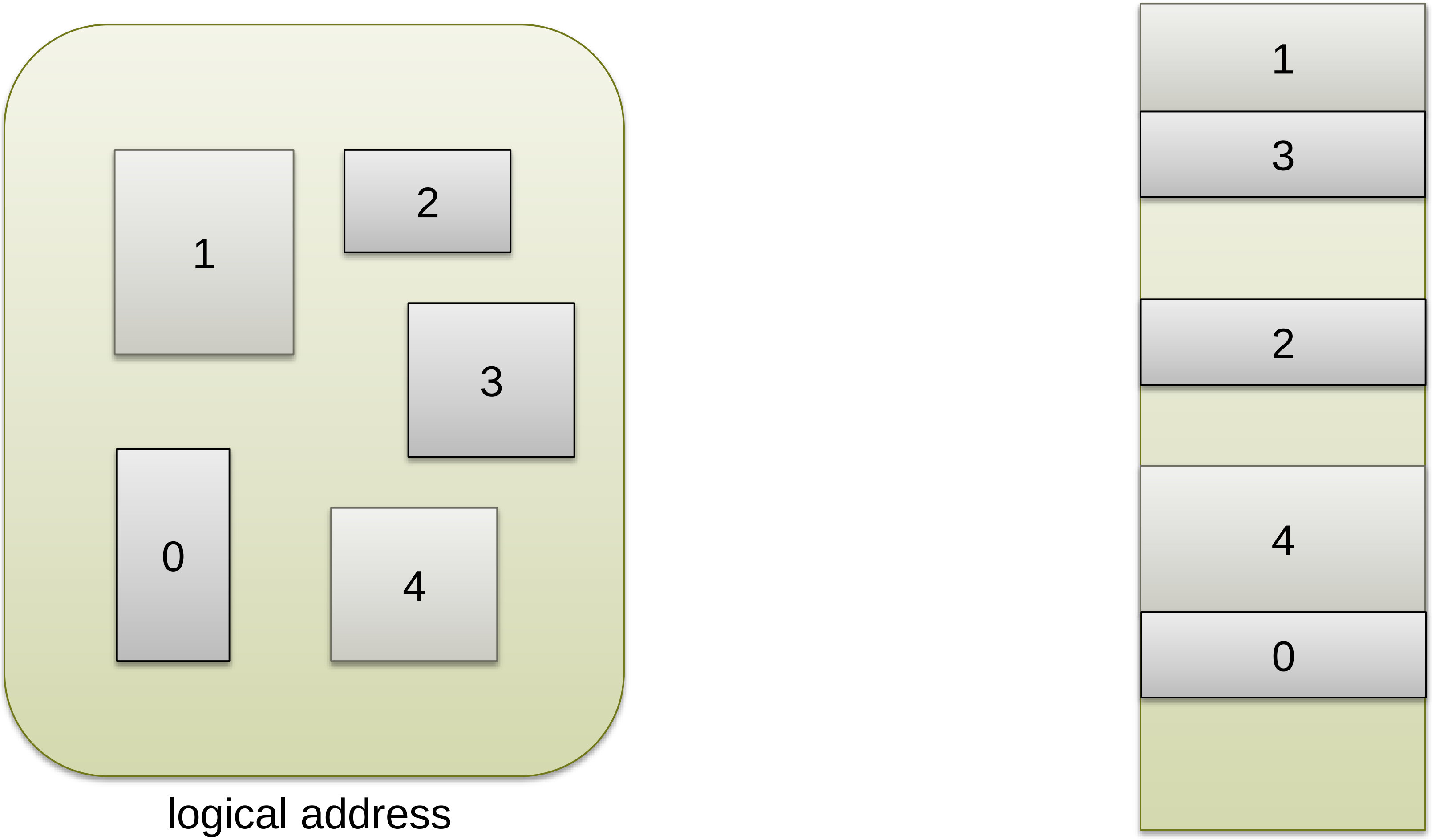
- **Generalize base + limit:**
  - Physical memory divided into *segments*
  - Logical address = (segment id, offset)
- **Segment identifier supplied by:**
  - Explicit instruction reference
  - Explicit processor segment register
  - Implicit instruction or process state

User’s View of a Program



logical address

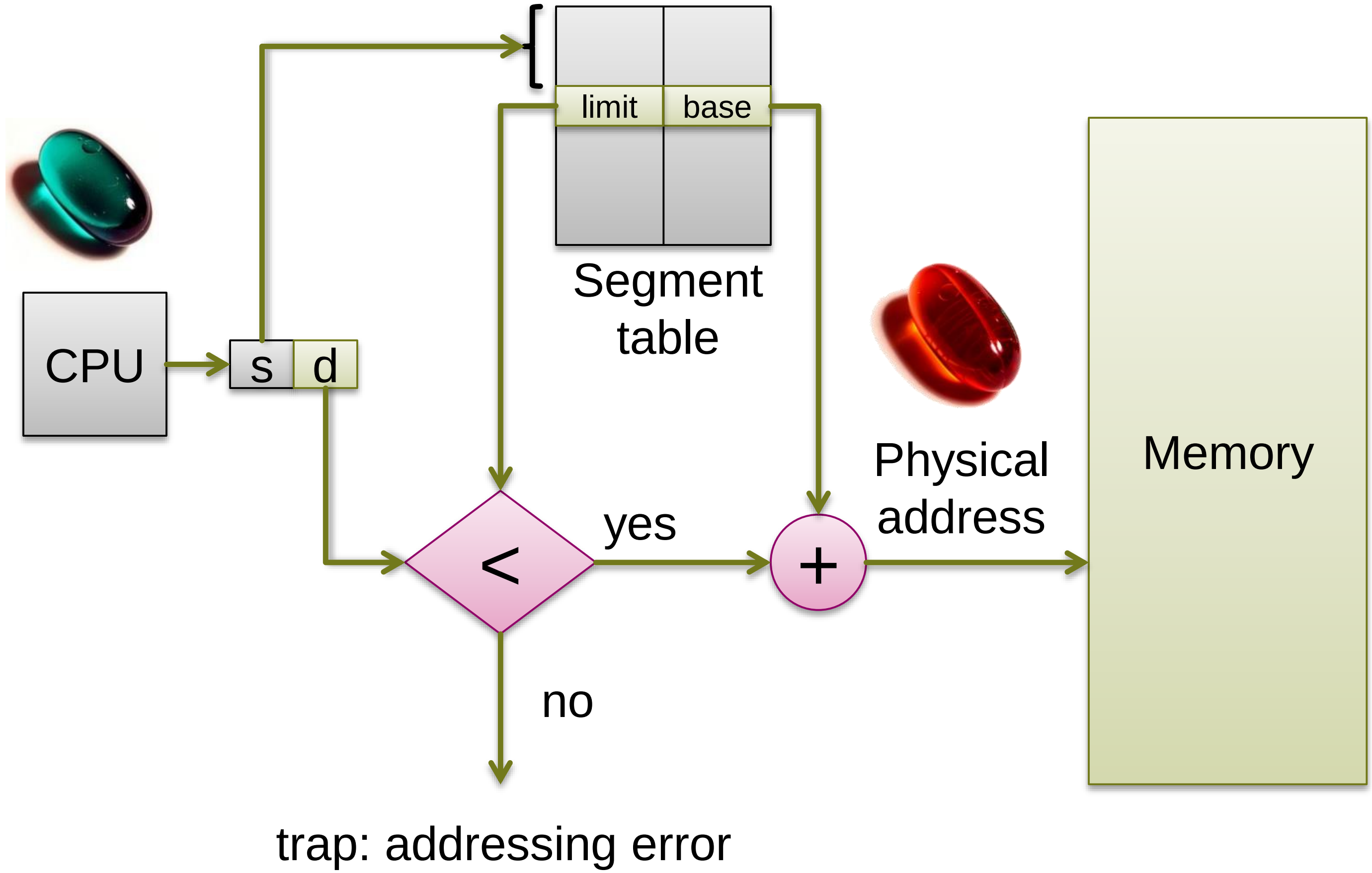
Segmentation Reality



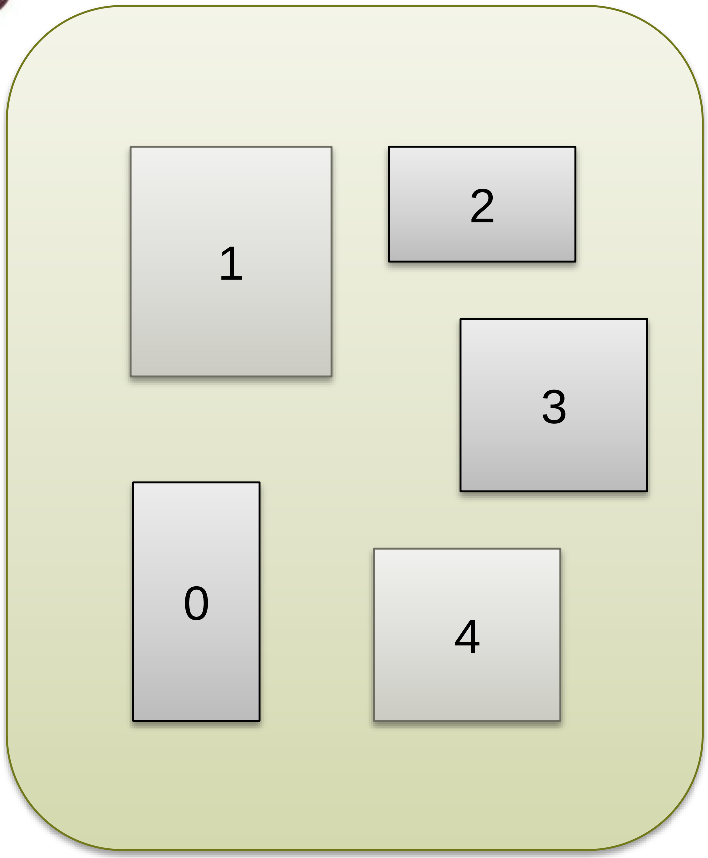
logical address

physical memory

Segmentation Hardware



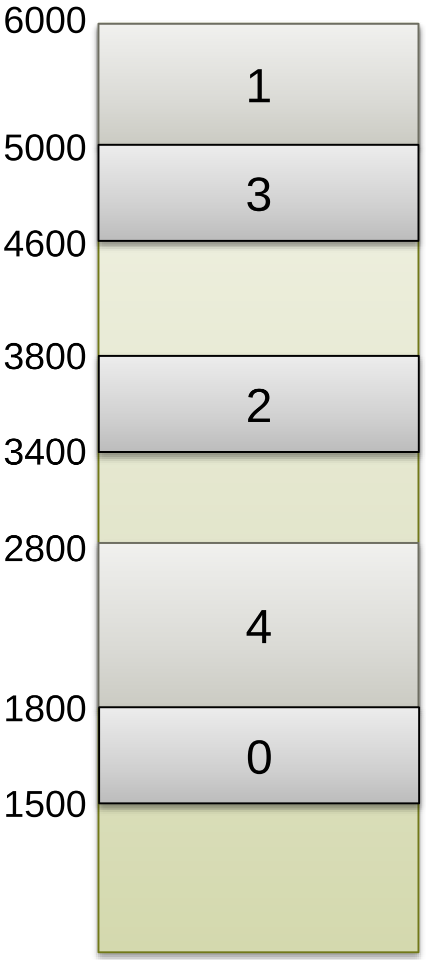
Segmentation Reality



logical address

	limit	base
0	300	1500
1	1000	5000
2	400	3400
3	400	4600
4	1000	1800

segment table



physical memory

Segmentation Architecture

- **Segment table – each entry has:**
  - **base** – starting physical address of segment
  - **limit** – length of the segment
- **Segment-table base register (STBR)**
  - Current segment table location in memory
- **Segment-table length register (STLR)**
  - Current size of segment table

segment number *s* is legal if *s* < STLR

Segmentation Summary

- **Fast context switch**
  - Simply reload STBR/STLR
- **Fast translation**
  - 2 loads, 2 compares
  - Segment table can be cached
- **Segments can easily be shared**
  - Segments can appear in multiple segment tables
- **Physical layout must still be contiguous**
  - (External) fragmentation still a problem

Paging

Paging

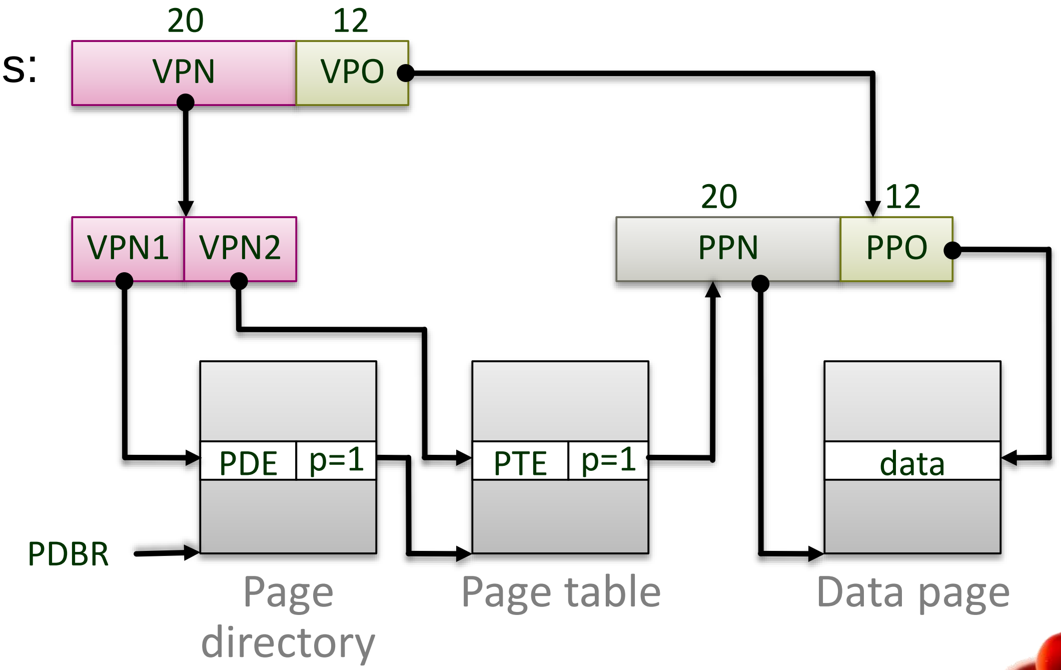
- **Solves contiguous physical memory problem**
  - Process can always fit if there is available free memory
- **Divide physical memory into *frames***
  - Size is power of two, e.g., 4096 bytes
- **Divide logical memory into *pages* of the same size**
- **For a program of *n* pages in size:**
  - Find and allocate *n* frames
  - Load program
  - Set up ***page table*** to translate logical pages to physical frames

Page table jargon

- **Page tables maps VPNs to PFNs**
  - Page table entry = PTE
- ***VPN*** **= Virtual Page Number**
  - Upper bits of virtual or logical address
- ***PFN*** **= Page Frame Number**
  - Upper bits of physical or logical address
- **Same number of bits (usually).**

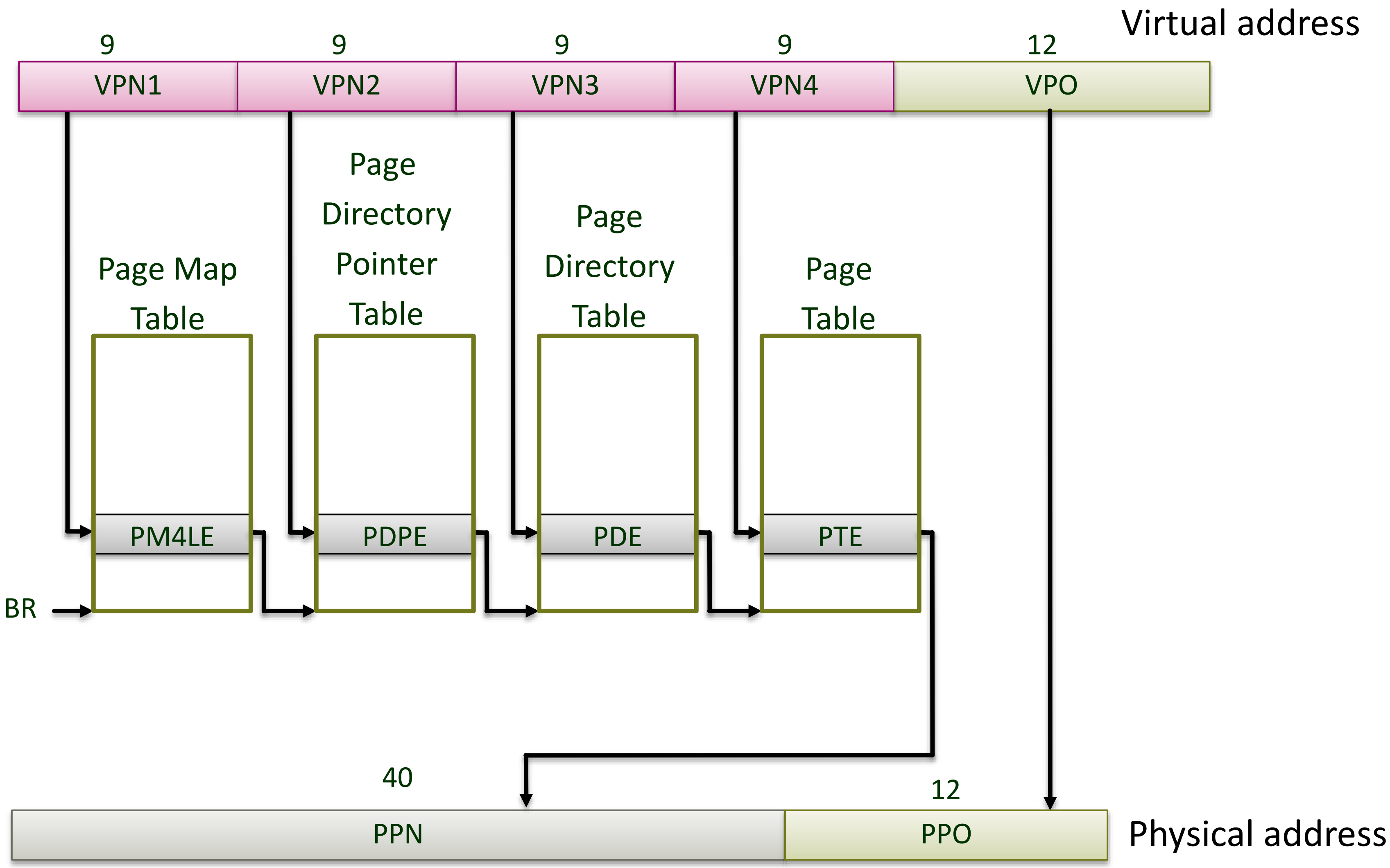
Recall: P6 Page tables (32bit)

Logical address:



- **Pages, page directories, page tables all 4kB**

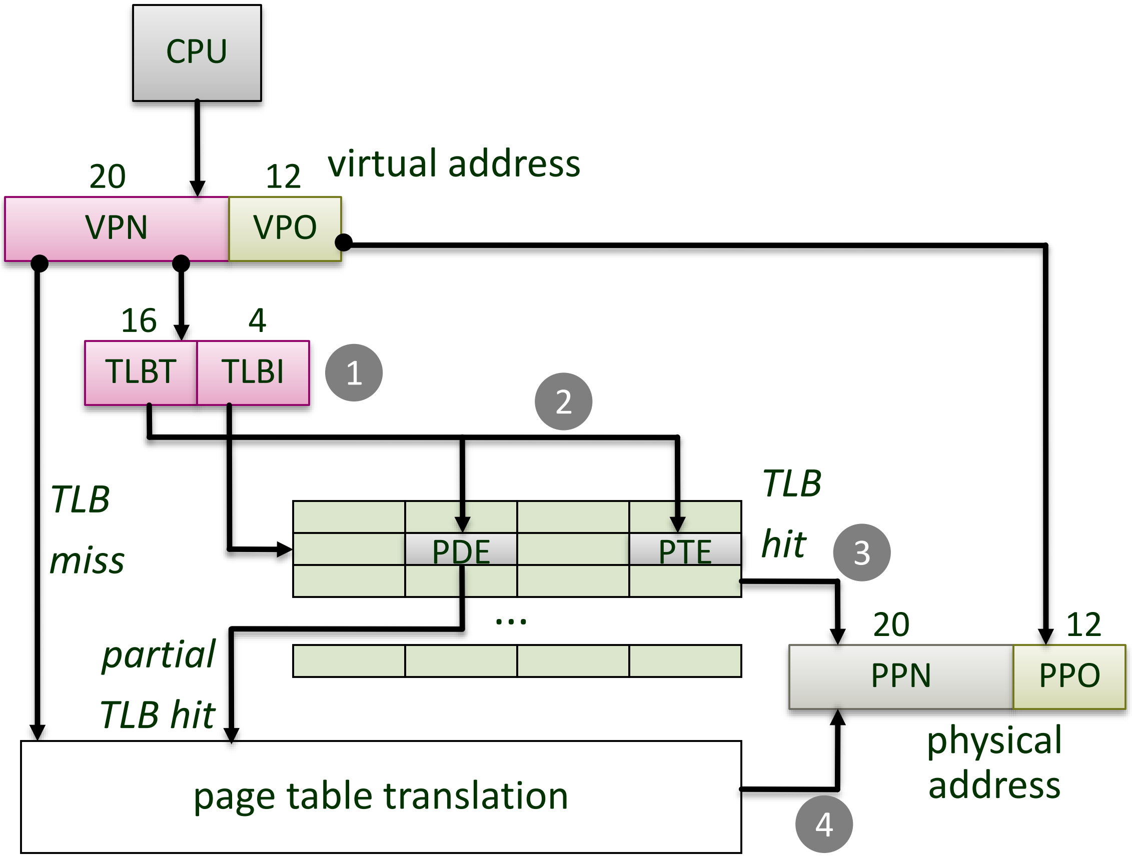
x86-64 Paging



Problem: performance

- **Every logical memory access needs more than two physical memory accesses**
  - Load page table entry → PFN
  - Load desired location
- **Performance ⇒ half as fast as with no translation**
  - Solution: cache page table entries

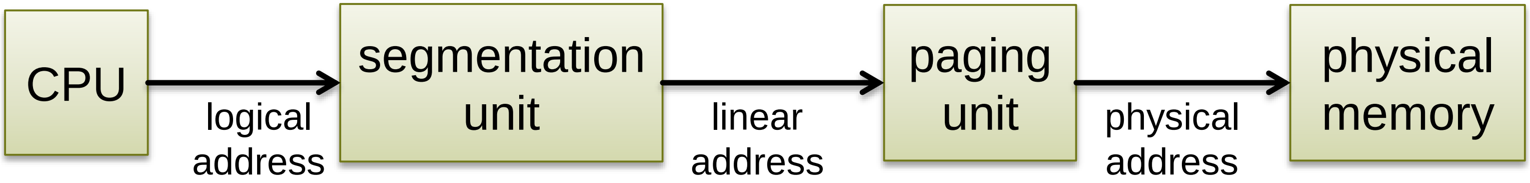
Translating with the P6 TLB



1. **Partition VPN into TLBT and TLBI.**
2. **Is the PTE for VPN cached in set TLBI?**
3. ***Yes:* Check permissions, build physical address**
4. ***No:* Read PTE (and PDE if not cached) from memory and build physical address**

In fact, x86 combines segmentation and paging

- **Segments do (still) have uses**
  - Thread-local state
  - Sandboxing (Google NativeClient, etc.)
  - Virtual machine monitors (Xen, etc.)



Effective Access Time

- **Associative Lookup = ε time units**
- **Assume memory cycle time is 1 time unit**
- **Hit ratio α =**
  - % time that a page number is found in the TLB;
  - Depends on locality and TLB entries (coverage)

Then *Effective Access Time*:

$$\begin{aligned}\text{EAT} &= (1+\varepsilon)\,\alpha + (2+\varepsilon)(1-\alpha) \\ &= 2+\varepsilon-\alpha\end{aligned}$$

Assuming single-level page table.

Exercise: work this out for the P6 2-level table

Page Protection

Memory Protection

- **Associate protection info with each frame**
  - Actually no - with the PTE.
- **Valid-invalid bit**
  - “valid” ⇒ page mapping is “legal”
  - “invalid” ⇒ page is not part of address space, i.e., entry does not exist
- **Requesting an “invalid” address ⇒ “fault”**
  - A “page fault”, or….

Remember the P6 Page Table Entry (PTE)

31 12	11 9	8	7	6	5	4	3	2	1	0
Page physical base address	Avail	G	0	D	A	CD	WT	U/S	R/W	P=1

Page base address: 20 most significant bits of physical page address (forces pages to be 4 KB aligned)

Avail: available for system programmers

G: global page (don’t evict from TLB on task switch)

D: dirty (set by MMU on writes)

A: accessed (set by MMU on reads and writes)

CD: cache disabled or enabled

WT: write-through or write-back cache policy for this page

U/S: user/supervisor

R/W: read/write

P: page is present in physical memory (1) or not (0)

P6 protection bits

31 12	11 9	8	7	6	5	4	3	2	1	0
Page physical base address	Avail	G	0	D	A	CD	WT	U/S	R/W	P=1

Page base address: 20 most significant bits of physical page address (forces pages to be 4 KB aligned)

Avail: available for system programmers

G: global page (don’t evict from TLB on task switch)

D: dirty (set by MMU on writes)

A: accessed (set by MMU on reads and writes)

CD: cache disabled or enabled

WT: write-through or write-back cache policy for this page

U/S: user/supervisor

R/W: read/write

P: page is present in physical memory (1) or not (0)

P bit can be used to trap on *any* access (read or write)

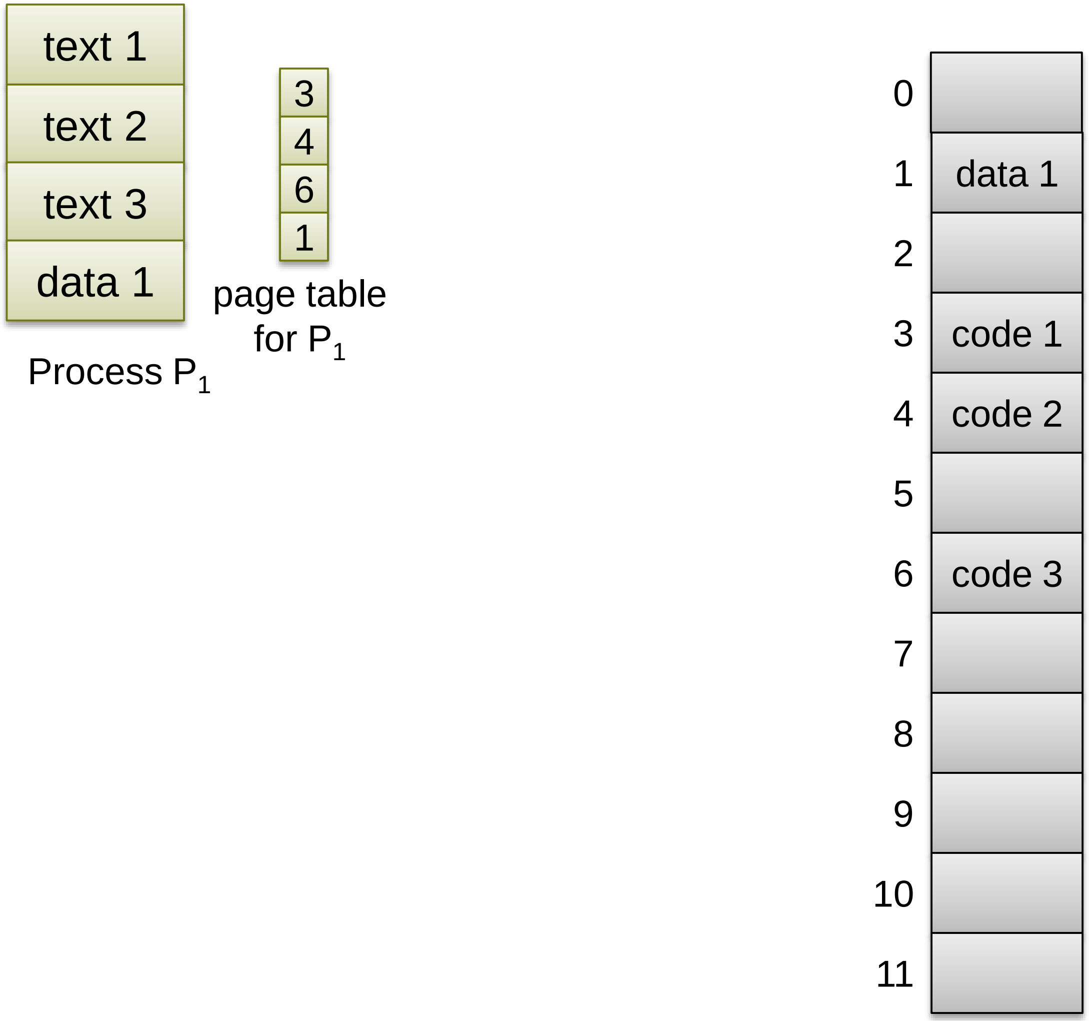
Protection information

- **Protection information typically includes:**
  - Readable
  - Writeable
  - Executable (can fetch to i-cache)
  - *Reference bits* used for demand paging

- ***Observe:* same attributes can be (and are) associated with segments as well**

Page sharing

Shared Pages Example

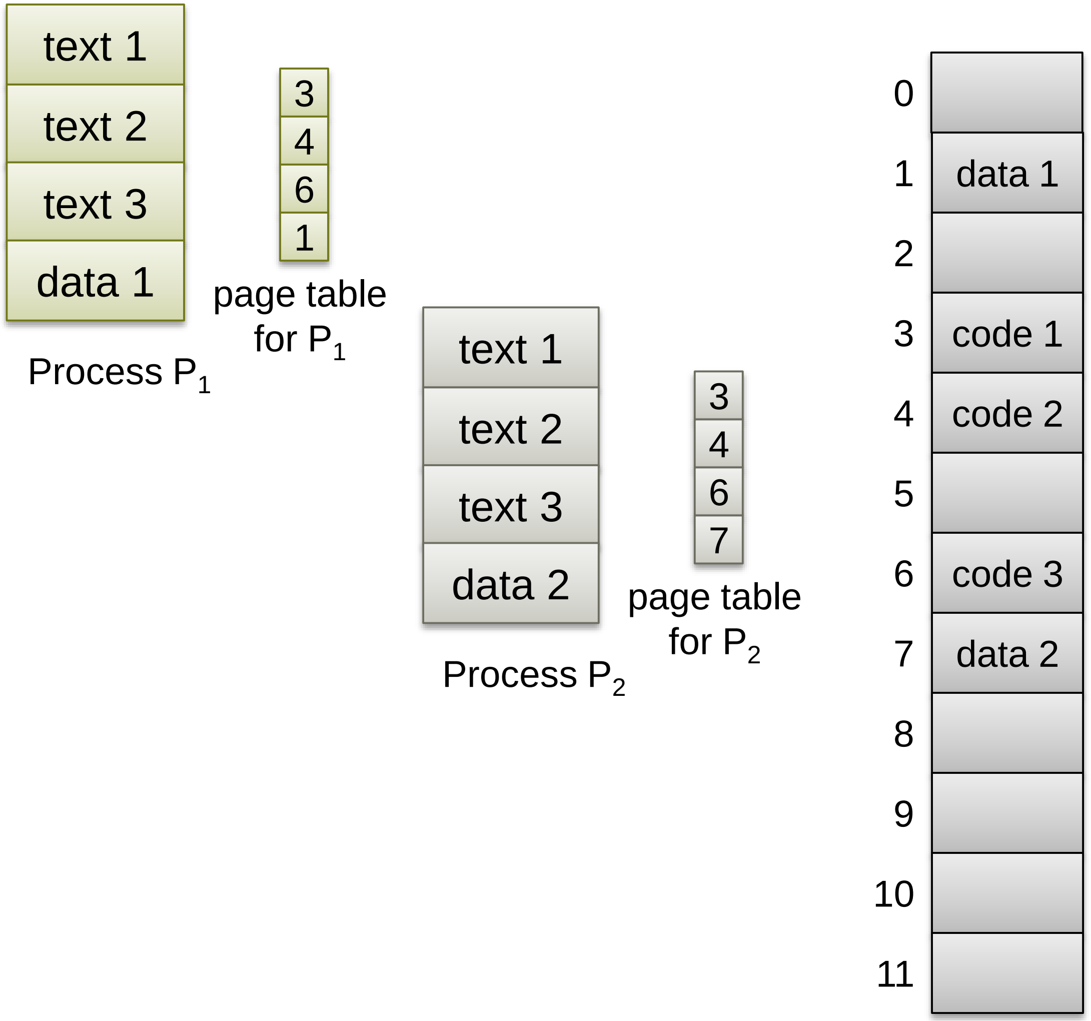


Process P_1
text 1
text 2
text 3
data 1

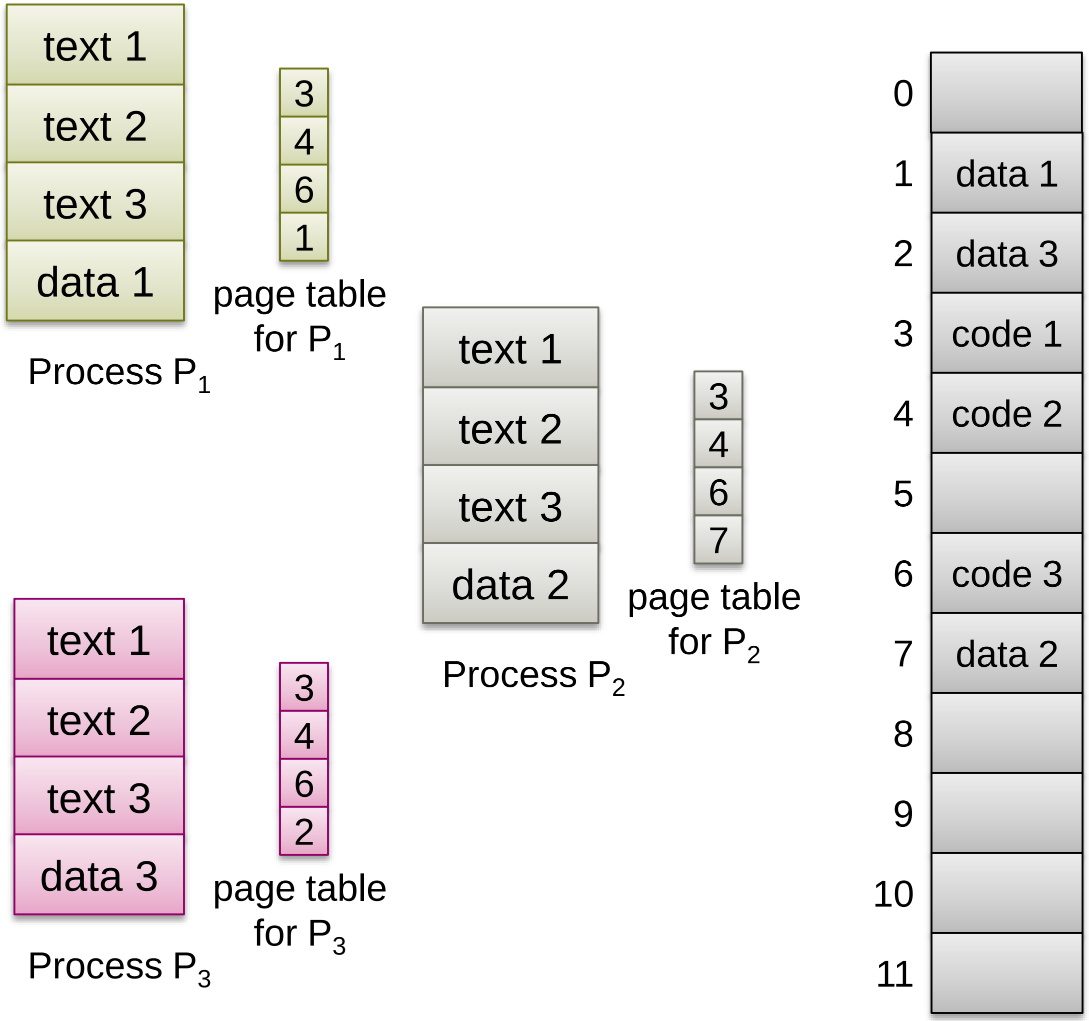
page table for P_1
3
4
6
1

0	
1	data 1
2	
3	code 1
4	code 2
5	
6	code 3
7	
8	
9	
10	
11	

Shared Pages Example



Shared Pages Example



text 1
text 2
text 3
data 1

Process P_1

3
4
6
1

page table for P_1

text 1
text 2
text 3
data 2

Process P_2

3
4
6
7

page table for P_2

text 1
text 2
text 3
data 3

Process P_3

3
4
6
2

page table for P_3

0	
1	data 1
2	data 3
3	code 1
4	code 2
5	
6	code 3
7	data 2
8	
9	
10	
11	

Shared Pages

- **Shared code**
  - One copy of read-only code shared among processes
  - Shared code (often) appears in same location in the logical address space of all processes
  - Data segment is not shared, different for each process
  - But still mapped at same address (so code can find it)

- **Private code and data**
  - Allows code to be relocated anywhere in address space

Per-process protection

- **Protection bits are stored in page table**
  - Plenty of bits available in PTEs
- **⇒ independent of frames themselves**
  - Different processes can share pages
  - Each page can have different protection to different processes
  - Many uses! E.g., debugging, communication, copy-on-write, etc.

Page Table Structures

Page table structures

- **Problem: simple linear page table is too big**
- **Solutions:**
  1. Hierarchical page tables
  2. Virtual memory page tables
  3. Hashed page tables
  4. Inverted page tables

Page table structures

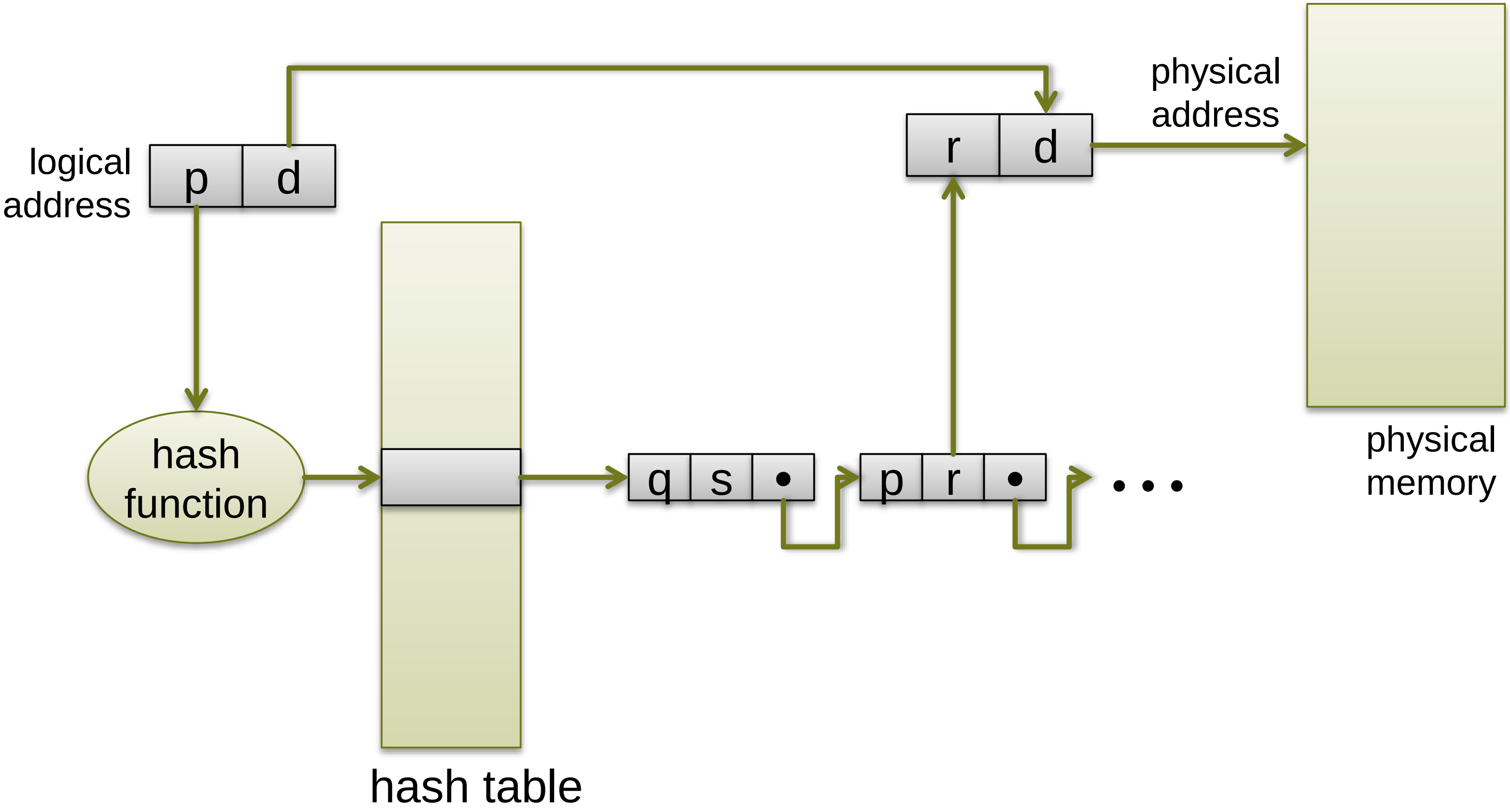
- **Problem: simple linear page table is too big**
- **Solutions:**
  1. Hierarchical page tables
  2. Virtual memory page tables (VAX)
  3. Hashed page tables
  4. Inverted page tables

Saw these last Semester.

#3 Hashed Page Tables

- **VPN is hashed into table**
  - Hash bucket has chain of logical->physical page mappings
- **Hash chain is traversed to find match.**
- **Can be fast, but can be unpredicable**
- **Often used for**
  - Portability
  - Software-loaded TLBs (e.g., MIPS)

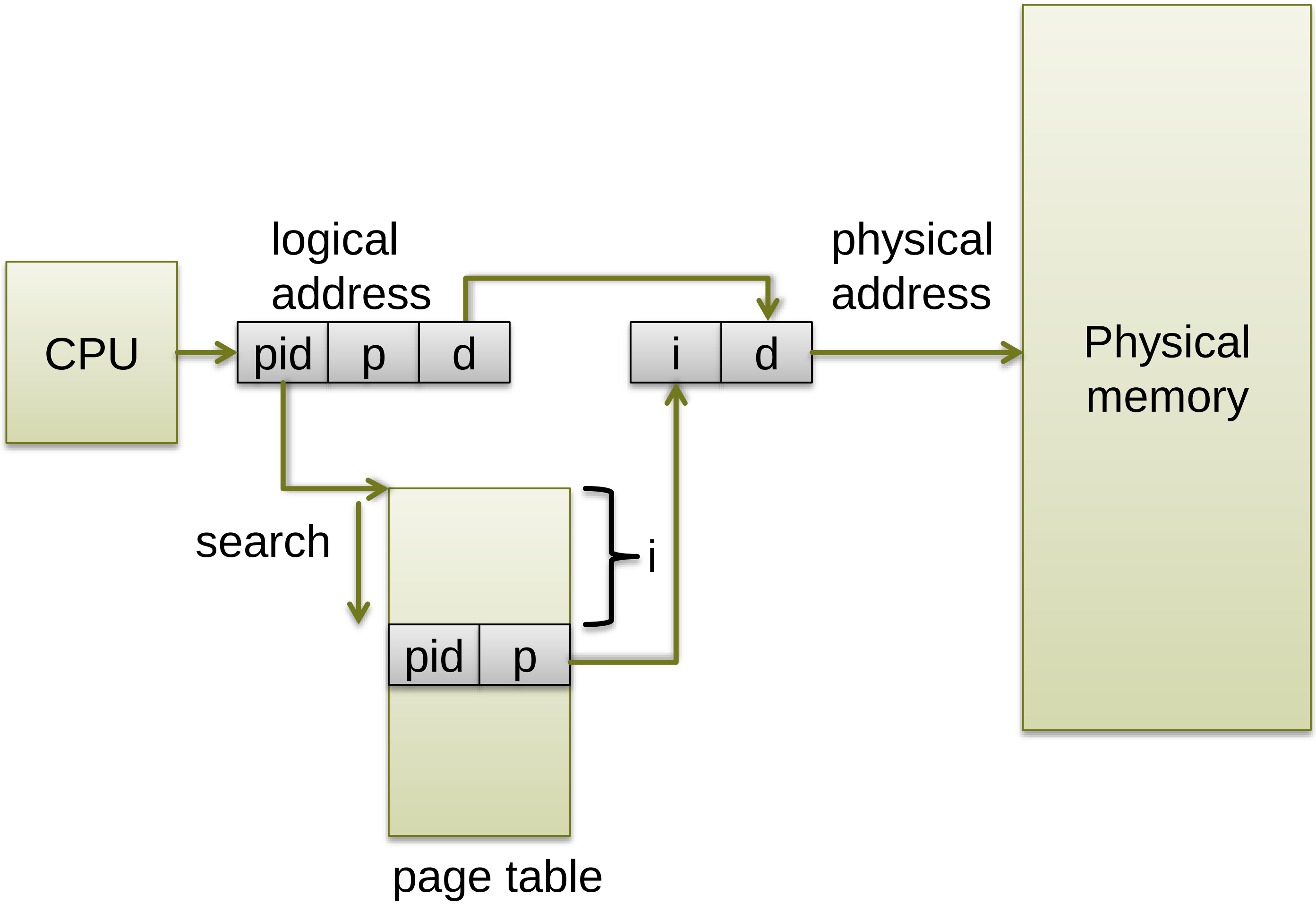
Hashed Page Table



#4 Inverted Page Table

- **One system-wide table now maps PFN -> VPN**
  - One entry for each real page of memory
  - Contains VPN, and which process owns the page
- **Bounds total size of all page information on machine**
  - Hashing used to locate an entry efficiently
- **Examples: PowerPC, ia64, UltraSPARC**

Inverted Page Table Architecture



The need for more bookkeeping

- **Most OSes keep their own translation info**
  - Per-process hierarchical page table (Linux)
  - System wide inverted page table (Mach, MacOS)
- **Why?**
  - Portability
  - Tracking memory objects
  - Software virtual → physical translation
  - Physical → virtual translation

TLB shootdown

TLB management

- **Recall: the TLB is a *cache*.**
- **Machines have many MMUs on many cores**
  ⇒ many TLBs
- **Problem: TLBs should be coherent. Why?**
  - Security problem if mappings change
  - E.g., when memory is reused

TLB management

	Process ID	VPN	PPN	access
Core 1 TLB:	0	0x0053	0x03	r/w
	1	0x20f8	0x12	r/w
Core 2 TLB:	0	0x0053	0x03	r/w
	1	0x0001	0x05	read
Core 3 TLB:	0	0x20f8	0x12	r/w
	1	0x0001	0x05	read

TLB management

	Process ID	VPN	PPN	access
Core 1 TLB:	0	0x0053	0x03	r/w
	1	0x20f8	0x12	r/w
Core 2 TLB:	0	0x0053	0x03	r/w
	1	0x0001	0x05	read
Core 3 TLB:	0	0x20f8	0x12	r/w
	1	0x0001	0x05	read

Change to read only

TLB management

	Process ID	VPN	PPN	access
Core 1 TLB:	0	0x0053	0x03	r/w
	1	0x20f8	0x12	r/w
Core 2 TLB:	0	~~0x0053~~	0x03	r/w
	1	0x0001	0x05	read
Core 3 TLB:	0	0x20f8	0x12	r/w
	1	0x0001	0x05	read

Change to read only

TLB management

	Process ID	VPN	PPN	access
Core 1 TLB:	0	0x0053	0x03	r/w
	1	0x20f8	0x12	r/w
Core 2 TLB:	0	~~0x0053~~	0x03	r/w
	1	0x0001	0x05	read
Core 3 TLB:	0	0x20f8	0x12	r/w
	1	0x0001	0x05	read

Change to read only

Process 0 on core 1 can only continue once shootdown is complete!

Keeping TLBs consistent

1. **Hardware TLB coherence**
   - Integrate TLB mgmt with cache coherence
   - Invalidate TLB entry when PTE memory changes
   - Rarely implemented
2. **Virtual caches**
   - Required cache flush / invalidate will take care of the TLB
   - High context switch cost!
     $\Rightarrow$ Most processors use physical caches
3. **Software TLB shootdown**
   - Most common
   - OS on one core notifies all other cores - Typically an IPI
   - Each core provides local invalidation
4. **Hardware shootdown instructions**
   - Broadcast special address access on the bus
   - Interpreted as TLB shootdown rather than cache coherence message
   - E.g., PowerPC architecture

Tomorrow: demand paging