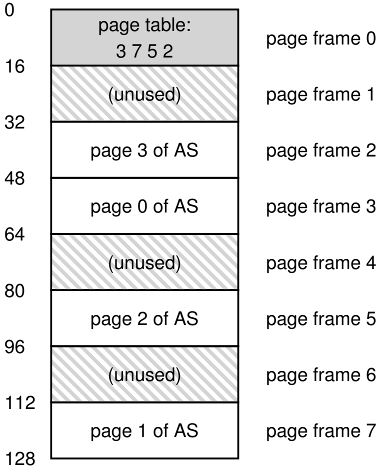


CSL 301

OPERATING SYSTEMS

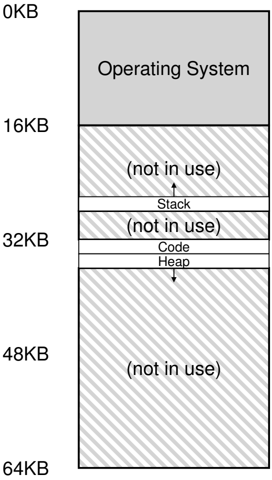
Lecture 10

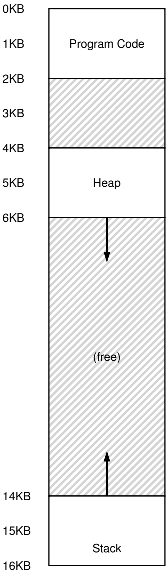
Segmentation (Contd.) + Paging

Instructor

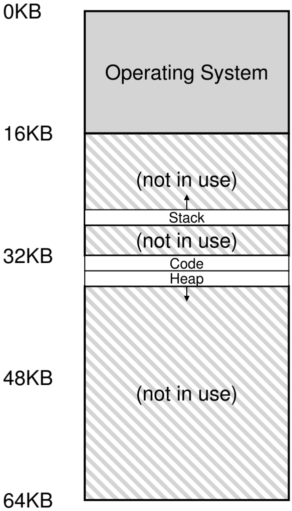
Dr. Dhiman Saha

Recall Segmentation





Pros so far



- Pieces of the address space are relocated into physical memory as the system runs
- Implies huge savings of physical memory w.r.t single base/bound register approach
- Note **unused space** saved between heap and stack **need no longer be allocated**

Idea

Share certain memory segments between address spaces

- For instance **code sharing**

- Hardware support: **protection bits**

Segment	Base	Size	Grows Positive?	Protection
Code	32K	2K	1	Read-Execute
Heap	34K	2K	1	Read-Write
Stack	28K	2K	0	Read-Write

- New task: Hardware has to check if a particular access is permissible

Coarse Grained Vs Fine-Grained

Fine/Coarse Grained

Granularity of segmentation

Coarse-grained

- Code/Stack/Heap
- Relatively large segments
- Hence coarse-grained

Fine-grained

- Idea of smaller segments
- Many segments $\implies$ more H/W support

- Idea of **segment table** (stored in memory)

How does this help?

With fine-grained segments, the OS could better learn about which segments are in use and which are not.

OS Support

What should the OS do on a context switch?

- The segment registers must be saved and restored

Second issue is managing free space in physical memory

- The OS has to be able to find space in physical memory for the segments of a new process

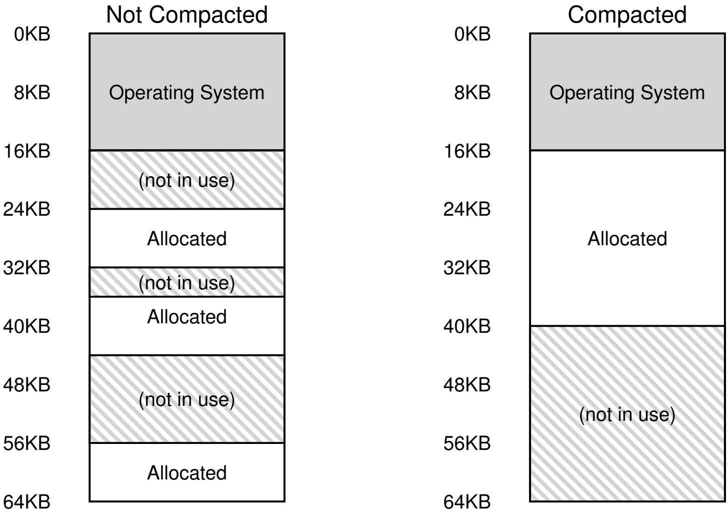
Note

We have a number of segments per process, and each segment might be a different size.

External Fragmentation

Physical memory quickly becomes **full of little holes of free space**, making it difficult to allocate new segments, or to grow existing ones.

Solution #1: Compaction



The OS cannot (left)/ can (right) satisfy a 20KB request.

Note

- Compaction is memory-intensive
- And what about the CPU activity during compaction?

Free-list management algorithm

- Best-fit
- Worst-fit
- Buddy Algorithm (covered later in the course)

Can you get rid of external fragmentation?

Issues?

- External fragmentation
- Not flexible enough to support our fully generalized, sparse address space

Example

If we have a large but sparsely-used heap all in one logical segment, the entire heap must still reside in memory in order to be accessed.

- Actual address space usage is not taken into account

Which ones have been dealt with so far? Assumptions

1. User's address space is placed contiguously in physical memory.
2. Size of the address space is not too big
3. It is less than the size of physical memory
4. Each address space is exactly the same size

Attempt #5:
Hardware-Based Relocation
Paging

Paging

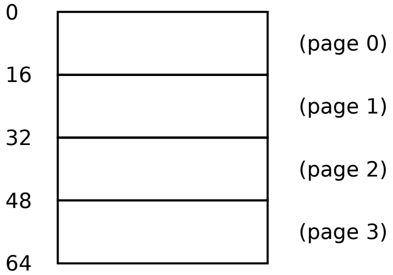
Idea

Divide address space into fixed-sized units, called **pages**.

- Same for physical memory in terms of **page-frames**

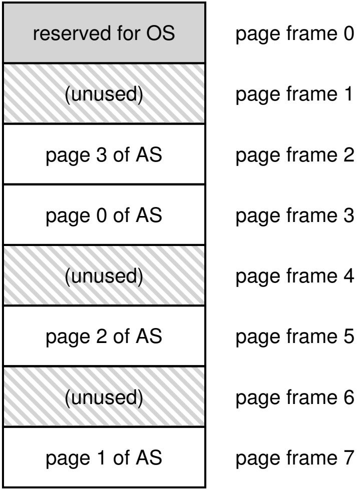
$$\text{page}_{\texttt{virtual-memory}} \equiv \text{page-frame}_{\texttt{physical-memory}}$$

Pages ⟷ Page Frames



0	
	(page 0)
16	
	(page 1)
32	
	(page 2)
48	
	(page 3)
64	

A 64-byte Address Space



reserved for OS	page frame 0
(unused)	page frame 1
page 3 of AS	page frame 2
page 0 of AS	page frame 3
(unused)	page frame 4
page 2 of AS	page frame 5
(unused)	page frame 6
page 1 of AS	page frame 7

A 128-Byte Physical Memory

Memory Virtualization with Paging

- How can we virtualize memory with pages, so as to avoid the problems of segmentation?
- What are the basic techniques?
- How do we make those techniques work well, with minimal space and time overheads?

Advantage

Flexibility

- Better abstraction of the address space
- Does not rely on **how** a process uses the address space
- Recall heap, stack **growth** in segmentation
- **Simplicity** of free-space management

How?

No external fragmentation

The Page Table

Task

To record where each **virtual page** of the **address space** is placed in physical memory

Solution

A **per-process** data structure known as a **page table**

- Basically stores the address translations for each of the virtual pages of the address space
- Note that this a **per-process** feature
- What changes during a context-switch?

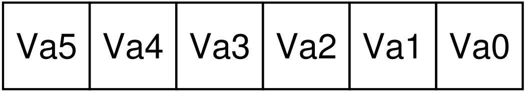
Address-Translation Example

- `movl <virtual address>, %eax`
- Recal: 64 byte address space

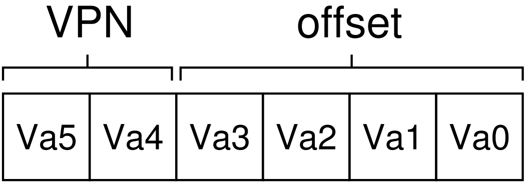
Va5	Va4	Va3	Va2	Va1	Va0

Address-Translation Example

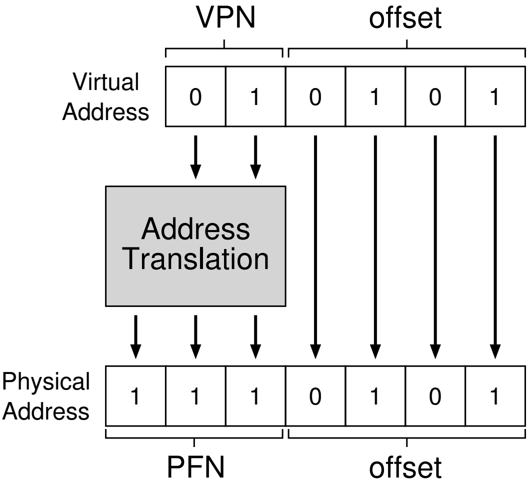
- `movl <virtual address>, %eax`
- Recal: 64 byte address space



- Virtual Address = VPN + Offset



Address-Translation (Classwork) `movl 21, %eax`



Points to Ponder

- Where are these page tables stored?
- What are the typical contents of the page table?
- How big are the tables?
- Does paging make the system (too) slow?

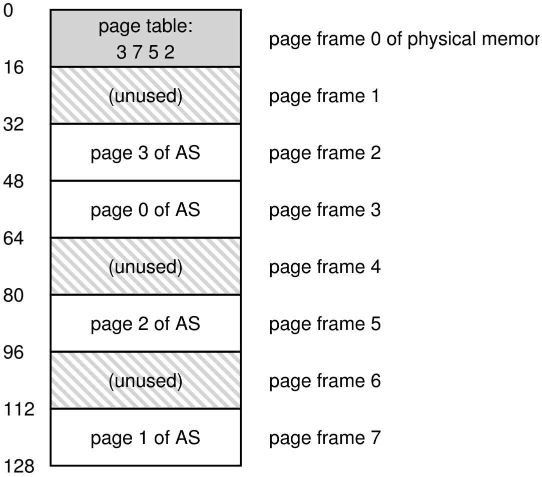
Where are page tables stored?

Should we use hardware support like we have done so far?

Let us do the math:

- Address Space: 32-bit
- Page Size (say): 4KB
- VPN Size: ________________
- Total number of translations: ________________
- Typical **Page Table Entry** Size: 4 bytes
- Page Table Size: ________________
- Multiply by #processes. So where should we store PT?

Where are page tables stored?



Page Table in Kernel Physical Memory

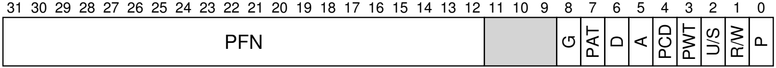
What's Actually In The Page Table?

Page Table Organization

What is the simplest data structure that comes to mind?

PTE Contents:

- Valid bit
- Protection bits
- Present bit
- Dirty bit
- Reference bit (a.k.a. accessed bit) and some other bits
- PFN



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PFN																							G	PAT	D	A	PCD	PWT	U/S	R/W	P

An x86 Page Table Entry (PTE)

Does paging make the system (too) slow?

Let us see what happens? Our Example

```
movl 21, %eax
```

- Step 1: Where is the page table of the process in memory?

Page-table base register

Contains the physical address of the starting location of the page table

- Step 2: To get correct PTE for translation hardware will do:

```
    VPN   = (VirtualAddress & VPN_MASK) >> SHIFT
PTEAddr   = PageTableBaseReg + (VPN * sizeof(PTE))
```

Does paging make the system (too) slow?

- Step 3: The hardware will fetch the PTE from memory, extract the PFN

Note

This is a memory access that the system cannot avoid!!!

- Step 4: The hardware will do the final translation:

```
  offset   = VirtualAddress & OFFSET_MASK
PhysAddr   = (PFN << SHIFT) | offset
```

- Step 5: The hardware can fetch the desired data from memory and put it into register eax.

Accessing Memory With Paging

```
// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT

// Form the address of the page-table entry (PT
PTEAddr = PTBR + (VPN * sizeof(PTE))

// Fetch the PTE
PTE = AccessMemory(PTEAddr)

// Check if process can access the page
if (PTE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.ProtectBits) == False)
    RaiseException(PROTECTION_FAULT)
else
    // Access is OK: form physical address and
    offset   = VirtualAddress & OFFSET_MASK
    PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
    Register = AccessMemory(PhysAddr)
```

Issues?

Space overheads Issue 1

Storing PT in memory wastes valuable memory space.

$>$ Factor 2 slow down Issue 2

For every memory reference, paging requires us to perform one extra memory reference in order to first fetch the translation from the page table

Attempt #6:
Hardware-Based Relocation
Translation-lookaside Buffer

Faster Paging