CS 134:
Operating Systems

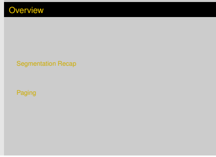
More Memory Management

2013-05-19

CS34

CS 134:
Operating Systems
More Memory Management

Overview



Overview

Segmentation Recap

Paging

Segmentation (Recap)

Logical address consists of the pair

<segment-number, offset>

Example

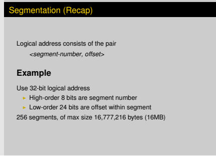
Use 32-bit logical address

- High-order 8 bits are segment number
- Low-order 24 bits are offset within segment

256 segments, of max size 16,777,216 bytes (16MB)

2013-05-19

CS34
└Segmentation Recap
    └Segmentation (Recap)


Segmentation (Recap)

Logical address consists of the pair

<segment-number, offset>

Example

Use 32-bit logical address

- High-order 8 bits are segment number
- Low-order 24 bits are offset within segment

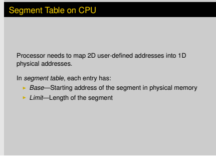
256 segments, of max size 16,777,216 bytes (16MB)

Segment Table on CPU

Processor needs to map 2D user-defined addresses into 1D physical addresses.

In *segment table*, each entry has:

- *Base*—Starting address of the segment in physical memory
- *Limit*—Length of the segment



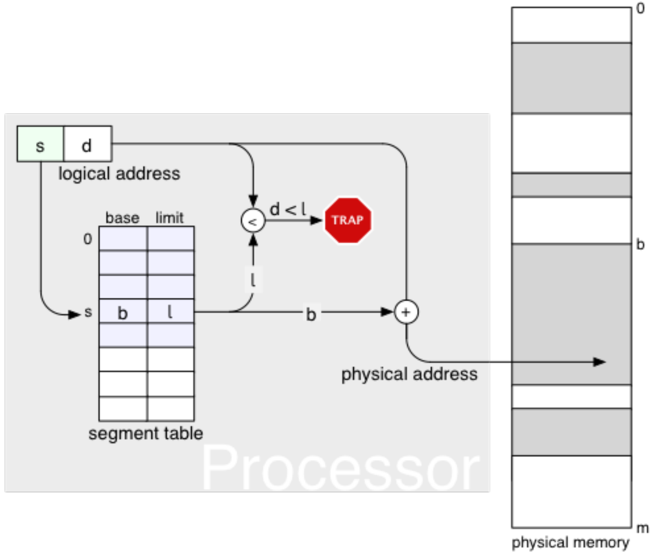
Segment Table on CPU

Processor needs to map 2D user-defined addresses into 1D physical addresses.

In *segment table*, each entry has:

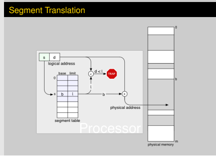
- *Base*—Starting address of the segment in physical memory
- *Limit*—Length of the segment

Segment Translation



2013-05-19

CS34
└Segmentation Recap
└Segment Translation



Segmentation Architecture

- Relocation
  - Dynamic
  - By segment table
- Sharing
  - Shared segments
  - Same segment number
- Allocation
  - First fit/best fit
  - External fragmentation

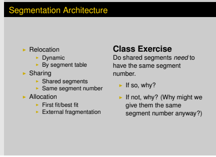
Class Exercise

Do shared segments *need* to have the same segment number.

- If so, why?
- If not, why? (Why might we give them the same segment number anyway?)

2013-05-19

CS34
└Segmentation Recap
└Segmentation Architecture


Segmentation Architecture

- Relocation
  - Dynamic
  - By segment table
- Sharing
  - Shared segments
  - Same segment number
- Allocation
  - First fit/best fit
  - External fragmentation

Class Exercise
Do shared segments *need* to have the same segment number.

- If so, why?
- If not, why? (Why might we give them the same segment number anyway?)

Segmentation Architecture

Class Exercise

Does our segmentation scheme capture the *difference* between code and data segments?

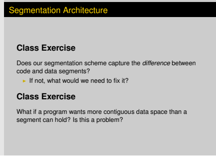
- If not, what would we need to fix it?

Class Exercise

What if a program wants more contiguous data space than a segment can hold? Is this a problem?

2013-05-19

CS34
└─Segmentation Recap
  └─Segmentation Architecture


Segmentation Architecture

Class Exercise

Does our segmentation scheme capture the *difference* between code and data segments?

- If not, what would we need to fix it?

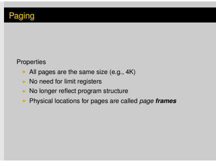
Class Exercise

What if a program wants more contiguous data space than a segment can hold? Is this a problem?

Paging

Properties

- All pages are the same size (e.g., 4K)
- No need for limit registers
- No longer reflect program structure
- Physical locations for pages are called *page* ***frames***



Paging

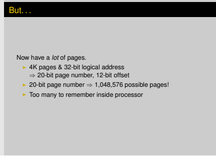
Properties

- All pages are the same size (e.g., 4K)
- No need for limit registers
- No longer reflect program structure
- Physical locations for pages are called *page* ***frames***

But. . .

Now have a *lot* of pages.

- 4K pages & 32-bit logical address
  $\Rightarrow$ 20-bit page number, 12-bit offset
- 20-bit page number $\Rightarrow$ 1,048,576 possible pages!
- Too many to remember inside processor



But. . .

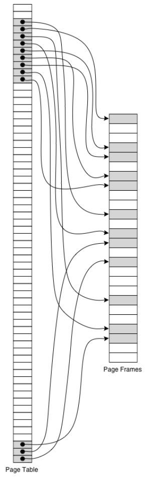
Now have a *lot* of pages.

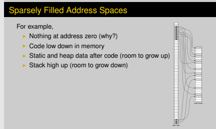
- 4K pages & 32-bit logical address
  $\Rightarrow$ 20-bit page number, 12-bit offset
- 20-bit page number $\Rightarrow$ 1,048,576 possible pages!
- Too many to remember inside processor

Sparsely Filled Address Spaces

For example,

- Nothing at address zero (why?)
- Code low down in memory
- Static and heap data after code (room to grow up)
- Stack high up (room to grow down)

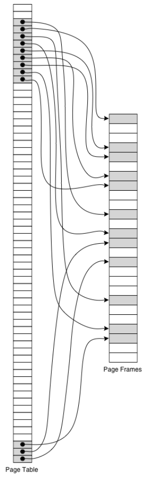


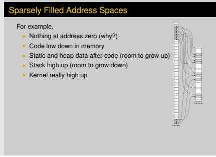

Sparsely Filled Address Spaces
For example,
- Nothing at address zero (why?)
- Code low down in memory
- Static and heap data after code (room to grow up)
- Stack high up (room to grow down)

Sparsely Filled Address Spaces

For example,

- Nothing at address zero (why?)
- Code low down in memory
- Static and heap data after code (room to grow up)
- Stack high up (room to grow down)
- Kernel really high up




Sparsely Filled Address Spaces
For example,
- Nothing at address zero (why?)
- Code low down in memory
- Static and heap data after code (room to grow up)
- Stack high up (room to grow down)
- Kernel really high up

Sparsely Filled Address Spaces

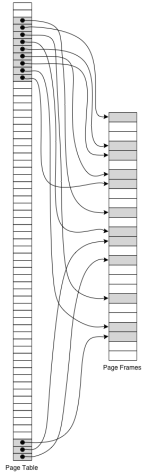
For example,

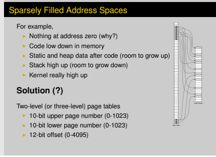
- Nothing at address zero (why?)
- Code low down in memory
- Static and heap data after code (room to grow up)
- Stack high up (room to grow down)
- Kernel really high up

Solution (?)

Two-level (or three-level) page tables

- 10-bit upper page number (0-1023)
- 10-bit lower page number (0-1023)
- 12-bit offset (0-4095)




Sparsely Filled Address Spaces

For example,
- Nothing at address zero (why?)
- Code low down in memory
- Static and heap data after code (room to grow up)
- Stack high up (room to grow down)
- Kernel really high up

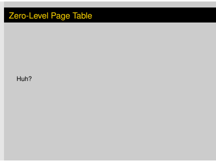
Solution (?)

Two-level (or three-level) page tables
- 10-bit upper page number (0-1023)
- 10-bit lower page number (0-1023)
- 12-bit offset (0-4095)

Zero-Level Page Table

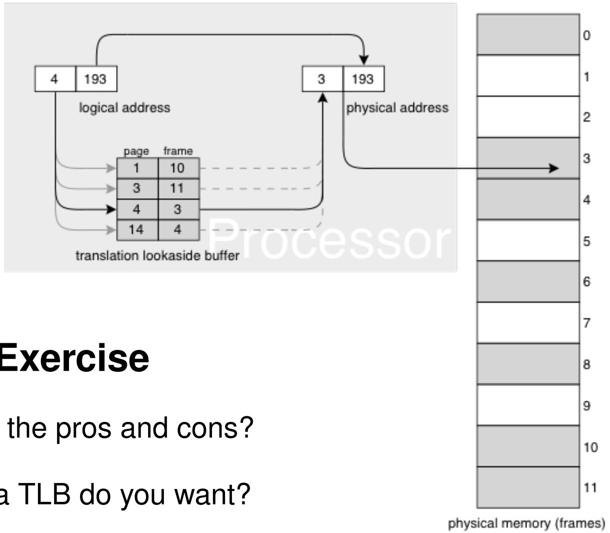
Huh?

CS34
└─Paging
  └─Zero-Level Page Table


Zero-Level Page Table

Huh?

Zero-Level Page Table



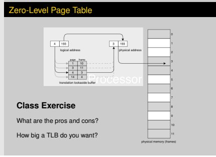
Class Exercise

What are the pros and cons?

How big a TLB do you want?

2013-05-19

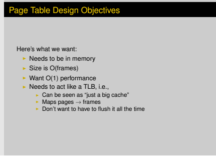
CS34
└─Paging
└─Zero-Level Page Table



Page Table Design Objectives

Here's what we want:

- Needs to be in memory
- Size is O(frames)
- Want O(1) performance
- Needs to act like a TLB, i.e.,
    - Can be seen as "just a big cache"
    - Maps pages $\rightarrow$ frames
    - Don't want to have to flush it all the time



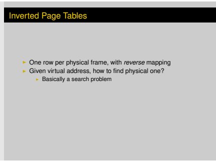
Page Table Design Objectives

Here's what we want:

- Needs to be in memory
- Size is O(frames)
- Want O(1) performance
- Needs to act like a TLB, i.e.,
    - Can be seen as "just a big cache"
    - Maps pages $\rightarrow$ frames
    - Don't want to have to flush it all the time

Inverted Page Tables

- One row per physical frame, with *reverse* mapping
- Given virtual address, how to find physical one?
  - Basically a search problem


Inverted Page Tables

- One row per physical frame, with *reverse* mapping
- Given virtual address, how to find physical one?
  - Basically a search problem

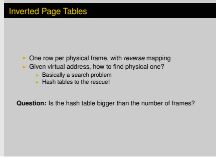
Inverted Page Tables

- One row per physical frame, with *reverse* mapping
- Given virtual address, how to find physical one?
  - Basically a search problem
  - Hash tables to the rescue!

Question: Is the hash table bigger than the number of frames?

2013-05-19

CS34
└Paging
└Inverted Page Tables

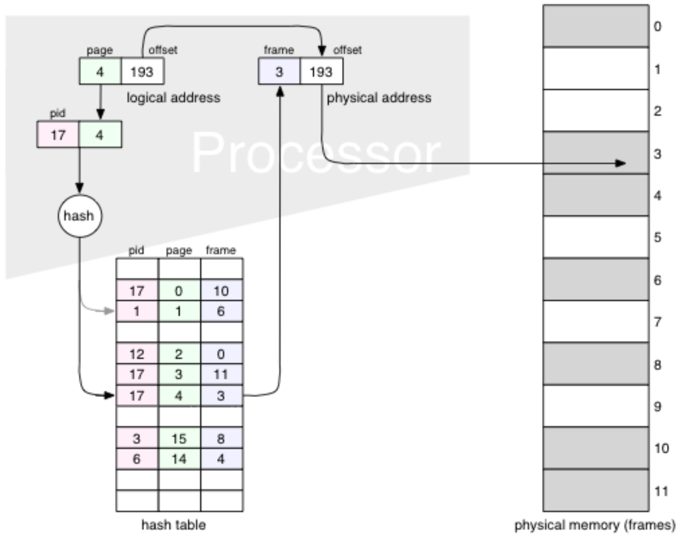


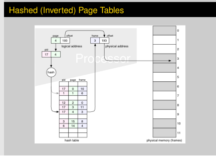
Inverted Page Tables

- One row per physical frame, with *reverse* mapping
- Given virtual address, how to find physical one?
  - Basically a search problem
  - Hash tables to the rescue!

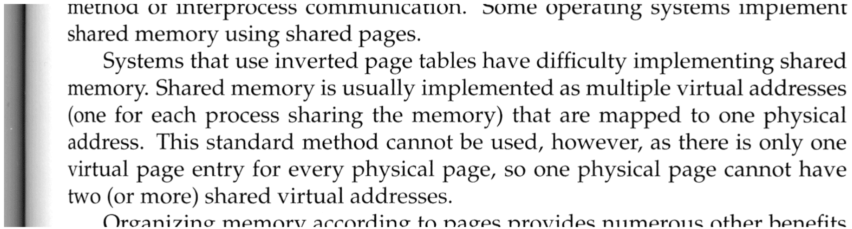
Question: Is the hash table bigger than the number of frames?

Hashed (Inverted) Page Tables





A Question



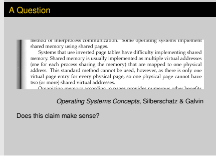
method of interprocess communication. Some operating systems implement shared memory using shared pages.

Systems that use inverted page tables have difficulty implementing shared memory. Shared memory is usually implemented as multiple virtual addresses (one for each process sharing the memory) that are mapped to one physical address. This standard method cannot be used, however, as there is only one virtual page entry for every physical page, so one physical page cannot have two (or more) shared virtual addresses.

Organizing memory according to pages provides numerous other benefits

Operating Systems Concepts, Silberschatz & Galvin

Does this claim make sense?



A Question

method of interprocess communication. Some operating systems implement shared memory using shared pages.

Systems that use inverted page tables have difficulty implementing shared memory. Shared memory is usually implemented as multiple virtual addresses (one for each process sharing the memory) that are mapped to one physical address. This standard method cannot be used, however, as there is only one virtual page entry for every physical page, so one physical page cannot have two (or more) shared virtual addresses.

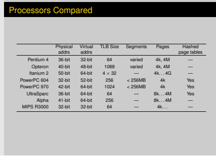
Organizing memory according to pages provides numerous other benefits

Operating Systems Concepts, Silberschatz & Galvin

Does this claim make sense?

Processors Compared

	Physical addrs	Virtual addrs	TLB Size	Segments	Pages	Hashed page tables
Pentium 4	36-bit	32-bit	64	varied	4k, 4M	—
Opteron	40-bit	48-bit	1088	varied	4k, 4M	—
Itanium 2	50-bit	64-bit	4×32	—	4k. . . 4G	—
PowerPC 604	32-bit	52-bit	256	< 256MB	4k	Yes
PowerPC 970	42-bit	64-bit	1024	< 256MB	4k	Yes
UltraSparc	36-bit	64-bit	64	—	8k. . . 4M	Yes
Alpha	41-bit	64-bit	256	—	8k. . . 4M	—
MIPS R3000	32-bit	32-bit	64	—	4k. . .	—



Processors Compared

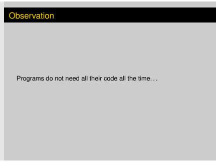
	Physical addrs	Virtual addrs	TLB Size	Segments	Pages	Hashed page tables
Pentium 4	36-bit	32-bit	64	varied	4k, 4M	—
Opteron	40-bit	48-bit	1088	varied	4k, 4M	—
Itanium 2	50-bit	64-bit	4×32	—	4k. . . 4G	—
PowerPC 604	32-bit	52-bit	256	< 256MB	4k	Yes
PowerPC 970	42-bit	64-bit	1024	< 256MB	4k	Yes
UltraSparc	36-bit	64-bit	64	—	8k. . . 4M	Yes
Alpha	41-bit	64-bit	256	—	8k. . . 4M	—
MIPS R3000	32-bit	32-bit	64	—	4k. . .	—

Observation

Programs do not need all their code all the time. . .

2013-05-19

CS34
└─Paging
└─Observation



Observation

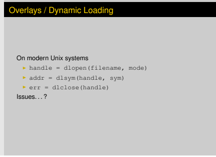
Programs do not need all their code all the time. . .

Overlays / Dynamic Loading

On modern Unix systems

- `handle = dlopen(filename, mode)`
- `addr = dlsym(handle, sym)`
- `err = dlclose(handle)`

Issues. . . ?


Overlays / Dynamic Loading

On modern Unix systems

- `handle = dlopen(filename, mode)`
- `addr = dlsym(handle, sym)`
- `err = dlclose(handle)`

Issues. . . ?

`dlopen` maps a file into the address space and returns an opaque handle.

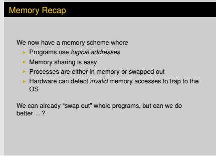
`dlsym` looks up a symbol in a dynamically loaded file.

Memory Recap

We now have a memory scheme where

- Programs use *logical addresses*
- Memory sharing is easy
- Processes are either in memory or swapped out
- Hardware can detect *invalid* memory accesses to trap to the OS

We can already “swap out” whole programs, but can we do better. . . ?



Memory Recap

We now have a memory scheme where

- Programs use *logical addresses*
- Memory sharing is easy
- Processes are either in memory or swapped out
- Hardware can detect *invalid* memory accesses to trap to the OS

We can already “swap out” whole programs, but can we do better. . . ?

Demand Paging

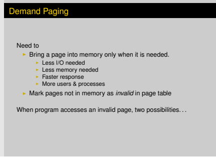
Need to

- Bring a page into memory only when it is needed.
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users & processes
- Mark pages not in memory as *invalid* in page table

When program accesses an invalid page, two possibilities. . .

2013-05-19

CS34
└Paging
└Demand Paging


Demand Paging

Need to

- Bring a page into memory only when it is needed.
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users & processes
- Mark pages not in memory as *invalid* in page table

When program accesses an invalid page, two possibilities. . .

Demand Paging—Hardware Support

Thus,

- Invalid accesses generate a trap
- Need to restart program after the trap
- Must seem like "nothing happened"

Example: The C-code for:

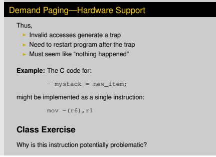
```
--mystack = new_item;
```

might be implemented as a single instruction:

```
mov -(r6),r1
```

Class Exercise

Why is this instruction potentially problematic?


Demand Paging—Hardware Support

Thus,

- Invalid accesses generate a trap
- Need to restart program after the trap
- Must seem like "nothing happened"

Example: The C-code for:

--mystack = new_item;

might be implemented as a single instruction:

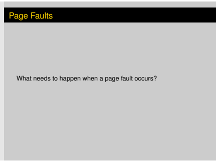
mov -(r6),r1

Class Exercise

Why is this instruction potentially problematic?

Page Faults

What needs to happen when a page fault occurs?

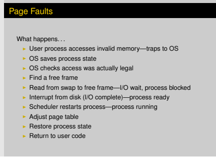

Page Faults

What needs to happen when a page fault occurs?

Page Faults

What happens. . .

- User process accesses invalid memory—traps to OS
- OS saves process state
- OS checks access was actually legal
- Find a free frame
- Read from swap to free frame—I/O wait, process blocked
- Interrupt from disk (I/O complete)—process ready
- Scheduler restarts process—process running
- Adjust page table
- Restore process state
- Return to user code


Page Faults

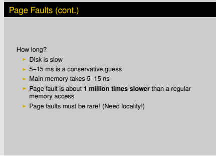
What happens. . .

- User process accesses invalid memory—traps to OS
- OS saves process state
- OS checks access was actually legal
- Find a free frame
- Read from swap to free frame—I/O wait, process blocked
- Interrupt from disk (I/O complete)—process ready
- Scheduler restarts process—process running
- Adjust page table
- Restore process state
- Return to user code

Page Faults (cont.)

How long?

- Disk is slow
- 5–15 ms is a conservative guess
- Main memory takes 5–15 ns
- Page fault is about **1 million times slower** than a regular memory access
- Page faults must be rare! (Need locality!)


Page Faults (cont.)

How long?

- Disk is slow
- 5–15 ms is a conservative guess
- Main memory takes 5–15 ns
- Page fault is about **1 million times slower** than a regular memory access
- Page faults must be rare! (Need locality!)

A “Back of an Envelope Calculation”

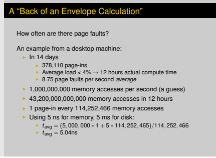
How often are there page faults?

An example from a desktop machine:

- In 14 days
  - 378,110 page-ins
  - Average load < 4% $\rightarrow$ 12 hours actual compute time
  - 8.75 page faults per second *average*
- 1,000,000,000 memory accesses per second (a guess)
- 43,200,000,000,000 memory accesses in 12 hours
- 1 page-in every 114,252,466 memory accesses
- Using 5 ns for memory, 5 ms for disk:
  - $t_{\text{avg}} = (5,000,000 * 1 + 5 * 114,252,465)/114,252,466$
  - $t_{\text{avg}} = 5.04\text{ns}$

2013-05-19

CS34
└Paging
└A “Back of an Envelope Calculation”


A “Back of an Envelope Calculation”

How often are there page faults?

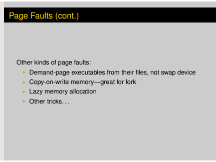
An example from a desktop machine:

- In 14 days
  - 378,110 page-ins
  - Average load < 4% $\rightarrow$ 12 hours actual compute time
  - 8.75 page faults per second *average*
- 1,000,000,000 memory accesses per second (a guess)
- 43,200,000,000,000 memory accesses in 12 hours
- 1 page-in every 114,252,466 memory accesses
- Using 5 ns for memory, 5 ms for disk:
  - $t_{\text{avg}} = (5,000,000 * 1 + 5 * 114,252,465)/114,252,466$
  - $t_{\text{avg}} = 5.04\text{ns}$

Page Faults (cont.)

Other kinds of page faults:

- Demand-page executables from their files, not swap device
- Copy-on-write memory—great for fork
- Lazy memory allocation
- Other tricks. . .


Page Faults (cont.)

Other kinds of page faults:

- Demand-page executables from their files, not swap device
- Copy-on-write memory—great for fork
- Lazy memory allocation
- Other tricks. . .

What kind of other tricks? Well, for example, debugging and tracing; VM translation; buffer overflow prevention.