Lecture M1: Virtual Memory Overview

****

Review -- 1 min

~~Dual mode operation~~

- ~~Implementation basics: supervisor mode, handler address, save/restore state~~
- ~~Three ways to invoke OS~~
- ~~Higher level abstractions on these primitives~~
  - ~~System call~~
  - ~~Context switch, scheduling~~
  - ~~Interprocess communication~~
  - ~~Virtual machine monitor~~

~~Boot~~

Process = address space + 1 or more threads

****

Outline - 1 min

Basic story --
-- simple mechanism
-- lots of powerful abstractions/uses

Virtual memory abstraction
What is an address space?
How is it implemented?: Translation
Sharing: **segmentation**
Sharing + simple allocation: **paging**
sharing + simple + scalable: **multi-level paging** (, **paged segmentation**, **paged paging**, …)
OR **inverted page table**

Quantitative measures:

Space overhead: internal v. external fragmentation, data structures
Time overhead: AMAT: average memory access time

Crosscutting theme: to make sure you understand these things, think about what TLB, kernel data structures are needed to implement it

****
Preview - 1 min
****

Outline/Preview

Historical perspective/motivation
Mechanism: translation
Use 1: protection + sharing + relocation
-- principles/basic approaches
-- case studies; cost models

Use 2: paging to disk

****
Lecture - 35 min
****

1. Overview/preview: Process abstraction

Prev lecture spoke informally about process. Next two chunks of class – memory and concurrency – will define key abstractions in much more detail. As overview/context, define process [INSERT Section 4 lecture 2 HERE]

2. Virtual memory abstraction

Reality v.	abstraction
Physical memory	Virtual memory
No protection address space	**protection** – each program isolated
Limited size	**expansion** -- infinite memory
sharing of physical frames	**relocation**— everyone thinks they are loaded at addr "0"; can put anything from 0..2^32-1 or 2^64-1
Easy to share data between	**sharing** –

ability to share code, data programs

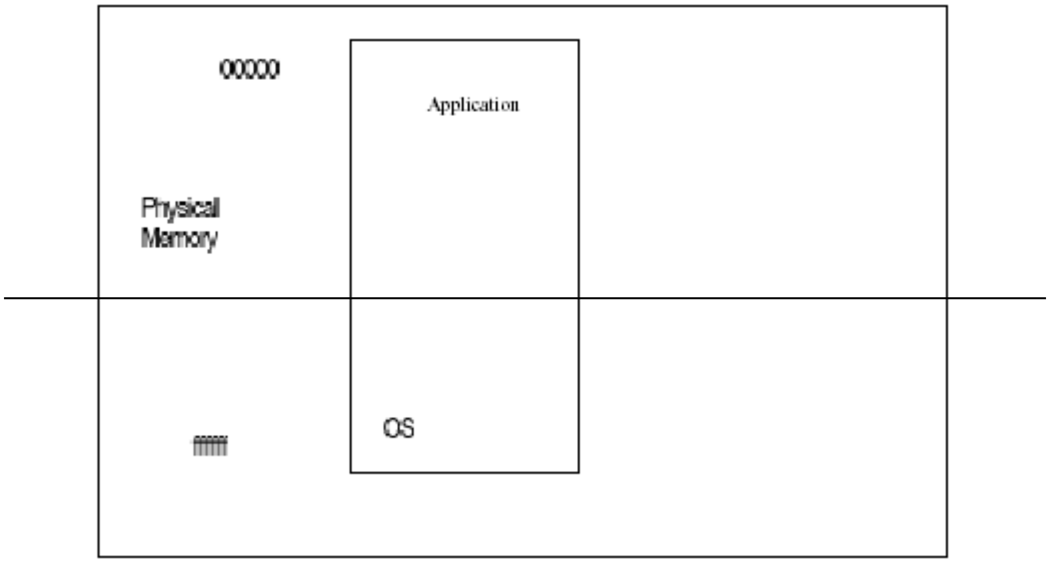
That all sounds great, but how do I do it?

2. ~~Historical perspective: Operating system organizations~~

~~2.1 Uniprogramming w/o protection~~

~~Personal computer OS's~~
~~Application always runs at the same place in physical memory~~
~~Each application runs one at a time~~
~~->give illusion of dedicated machine by giving reality of dedicated machine~~

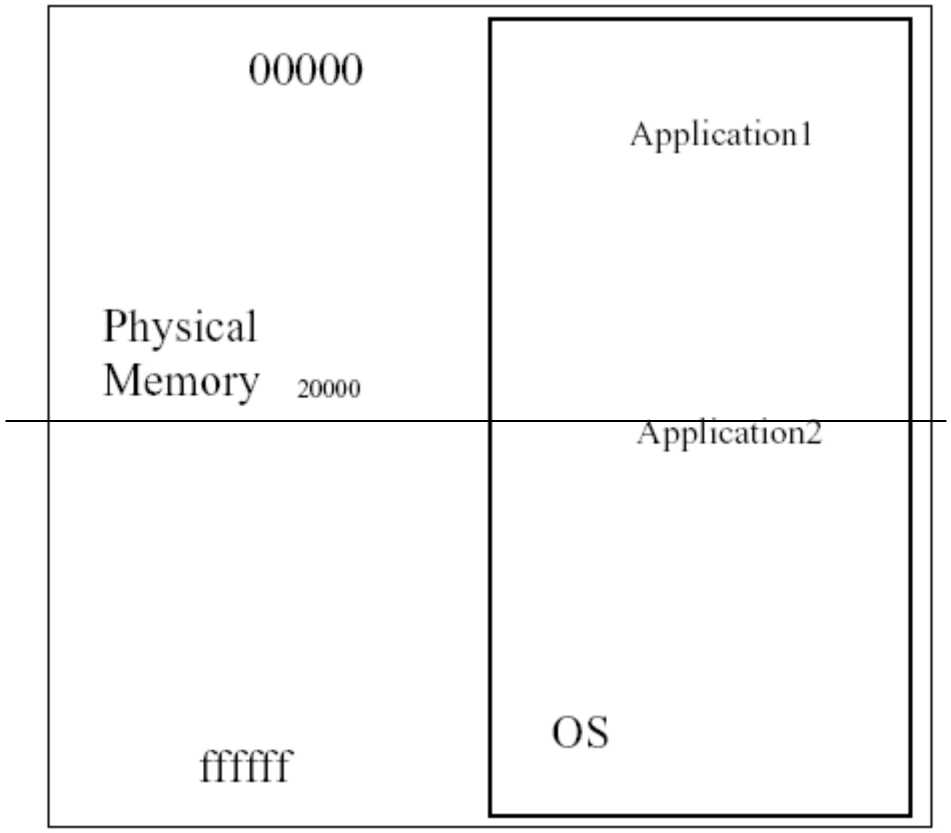
~~Example: load application into low memory, OS into high memory~~
~~Application can address any physical memory location~~



2. Multiprogramming w/o protection: Linker-loader

~~2.2 Multiprogramming w/o protection: Linker-loader~~

~~Can multiple programs share physical memory without hardware translation?~~
~~Yes: when copy program into memory, change its addresses (loads, stores, jumps) to use the addresses where program lands in memory. This is called a~~ **linker-loader**~~. Used to be very common.~~



~~Unix ld does the linking portion of this (despite its name deriving from loading): compiler generates each .o file with code that starts at address 0.~~

~~How do you create an executable from this? Scan through each .o, changing addresses to point to where each module goes in larger program (requires help from compiler to say where all the relocatable addresses are stored.)~~
~~With linker-loader – no protection: bugs in any program can cause other programs to crash or even the OS.~~

~~e.g.,~~

~~.long foo 42~~
~~…~~
~~bar:~~
~~load r3, foo~~
~~…~~
~~jmp bar~~

~~3 ways to make foo and bar location-independent:~~

~~(1) relative (v. absolute) address~~
~~(2) relocation register~~
~~(3) loader changes absolute address at load time~~

2.3 Multiprogrammed OS with protection

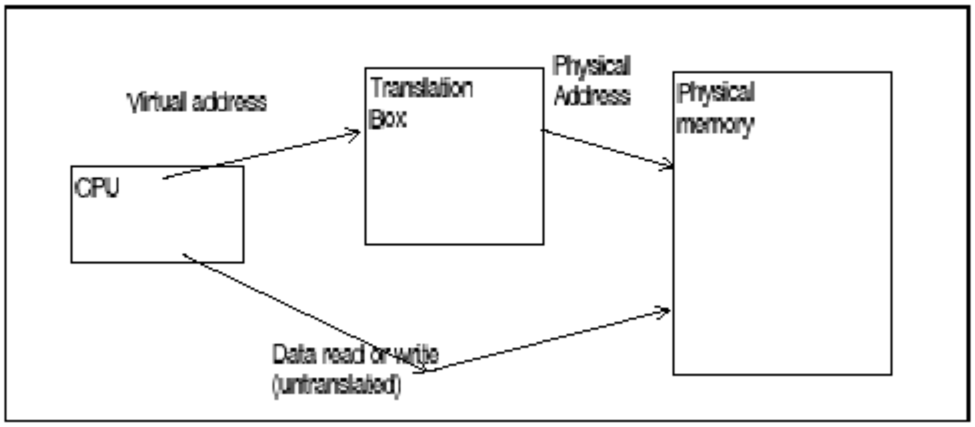
Goal of protection:

- keep user program from crashing OS
- keep user programs from crashing each other

How is protection implemented?
Hardware support:
1) address translation
2) dual mode operation: kernel v. user mode

3. Address translation

address space – literally, all the addresses a program can touch. All the state a program can affect or be affected by
Idea: restrict what a program can do by restricting what it can touch.
Fundamental rule of CS: all problems can be solved with a level of indirection



Translation box = abstraction for now.
Reality -- some combination of HW and SW

Level of indirection gives you

- protection

- No way for a program to even talk about another program's addresses; no way to touch OS code or data
- Translation box can implement *protection bits* – e.g., allow read but not write

• relocation (transparent sharing of memory)
P1's address 0 can be different than P2's address 0
Your program can put anything it wants in its address space 0..2^64-1

• Share data between programs if you want
P1's address 0xFF00 can point to same data as P2's address 0xFF00 (or P2's 0xAA00)

Notice
CPU (and thus your program) always see/work with VAs (and doesn't care about PAs)
Memory sees PAs (and doesn't care about VAs)

Think of memory in two ways
View from CPU – what program sees; virtual memory
View from memory – physical memory
Translation is implemented in hardware; controlled in software.

5. Implementing protection, relocation

want: programs to coexist in memory
need: mapping from
<pid, virtual addr> → <physical address>
Many different mappings; use odd-seeming combination of techniques for historical and practical reasons →seems confusing

"practical reasons" -- mainly that translation is critical to performance, so data structures get tightly optimized; data structures get split between HW and SW; ...

Remember that all of these algorithms are just arranging some simple techniques in different ways

Basics:

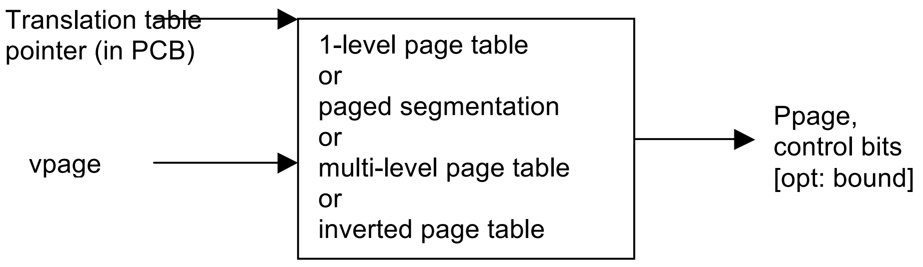
segment maps variable-sized range of contiguous virtual addresses to a range of contiguous physical addresses

page maps fixed size range of contiguous virtual addresses to a fixed sized range of contiguous virtual addresses

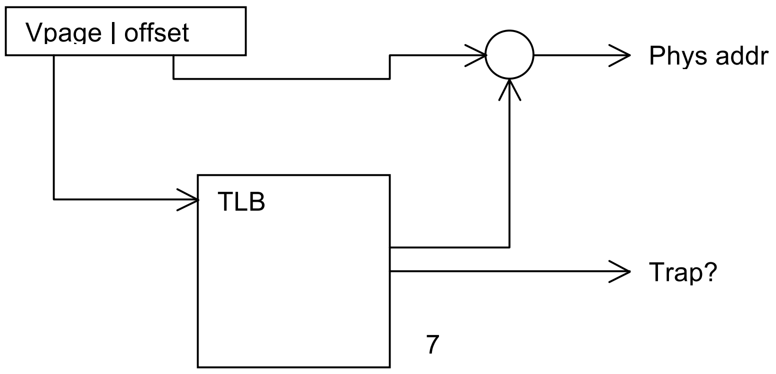
need data structures to lookup page/segment mapping given a virtual address
  <segment #> → segment info {base, size}
  <page #> → page info {base}

Again, data structures seem confusing – **base+bounds**, **segment table, page table, paged segmentation, multi-level page table, inverted page table** -- but we're just doing a lookup, and there aren't that many data structures that are used for lookup:
  (pointer)
  array
  tree
  hash table
  {used in various combinations}

Memory data structure is opaque object:


Translation table pointer (in PCB)
vpage
1-level page table or paged segmentation or multi-level page table or inverted page table
Ppage, control bits [opt: bound]

To speed things up, usually (always) add hardware lookup table (e.g., TLB)



[[defer] QUESTION: How will above picture differ for segments?]

Dual mode operation

Can application modify its own translation tables (memory or HW)? No. If it could, it could get access to all physical memory.
has to be restricted somehow
• kernel mode – can do anything (e.g. bypass translation, change translation for a process, etc)
• User mode – each program restricted to touching its own address space
Implementation
SW loaded TLB

- each process has process control block (PCB) in kernel
- PCB includes (pointer to or entire) software translation table (table in kernel memory --> applications cannot alter it)
- TLB miss --> exception
- --> kernel handler reads appropriate entry from currently running process's table and loads entry into TLB
-

X86: hw loaded tlb

- translation data structure is in kernel memory (PCB as above)
- HW register has pointer to this data structure
- TLB miss --> hardware can follow this pointer and load TCB
    - No exception handler in normal case (HW state machine)
    - Drop into OS exception handler if no/bad mapping/permission
- Context switch changes HW register to point to new process's mapping
    - Privileged instruction to load HW register

Various kinds of translation schemes

-- start with simplest!
****
Admin - 3 min

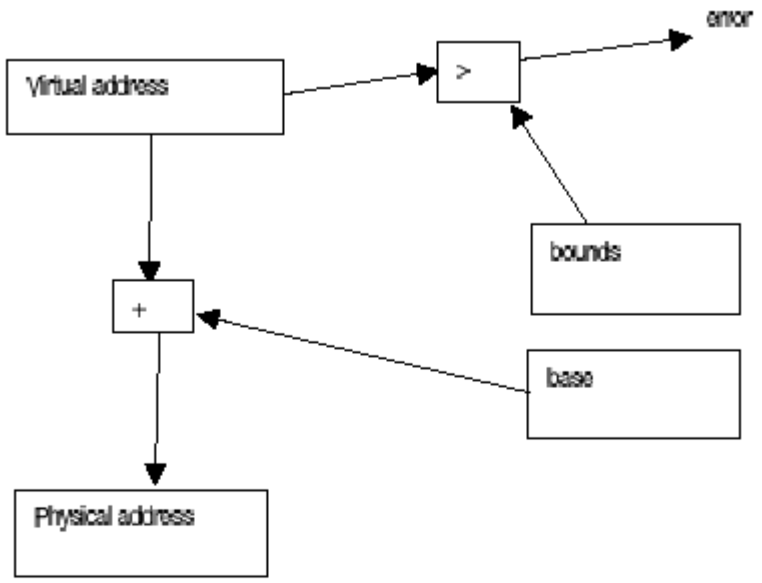
****
Lecture - 35 min
****

ORGANIZATION:
Series of systems
-- historical order
-- each illustrates/introduces a key idea

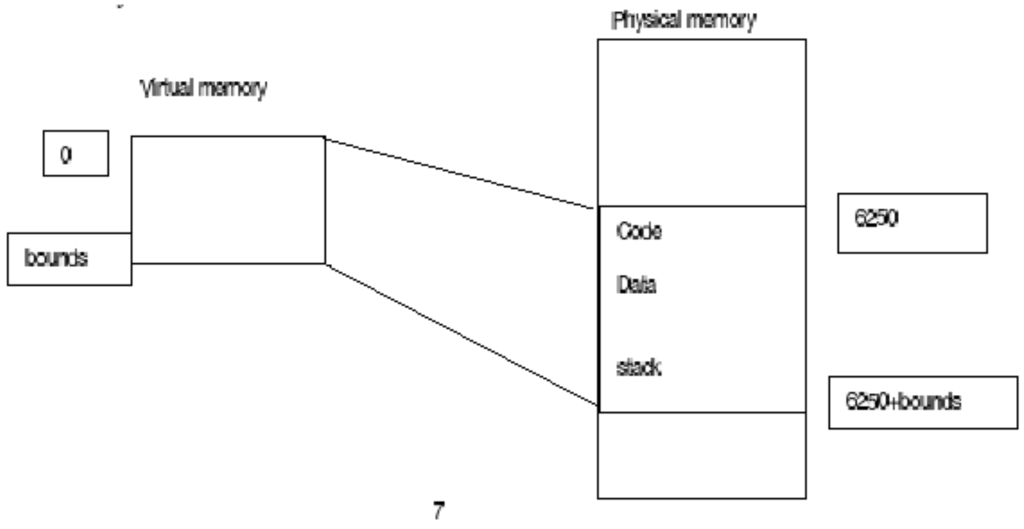
Now: begin discussion of different translation schemes. Remember what they have in common. Start simply.

6. Base and bounds --> Isolation

Each program loaded into contiguous regions of physical memory, but with protection between programs. First built in Cray-1



Program has illusion it is running in its own dedicated machine with memory starting at 0 and going up to <bounds>. Like linker-loader, program gets contiguous region of memory. But unlike linker loader, we have protection: program can only touch locations in physical memory between base and bounds.



Provides level of indirection: OS can move bits around behind program's back.
For instance, if program needs to grow beyond its bounds or if need to coalesce fragments of memory. → stop program, copy bits, change base and bound register, restart.

Implementation

Hardware
-- Add base and bounds registers to CPU (trivial TLB)
Software
-- Add base and bounds to process control block
-- Context switch -- change base and bounds registers (Privileged instruction)

Notice:
Only OS can change base and bounds (memory protection for PCB, privileged instruction for register)
Clearly user can't or else lose protection.

Hardware cost:
2 registers
adder, comparator

Plus, slows down hardware b/c need to take time to do add/compare on every memory reference.

QUESTION: How does protection work?, Sharing?

Evaluation
Base and bounds pros:

+ **protection**
+ simple, fast

Cons:

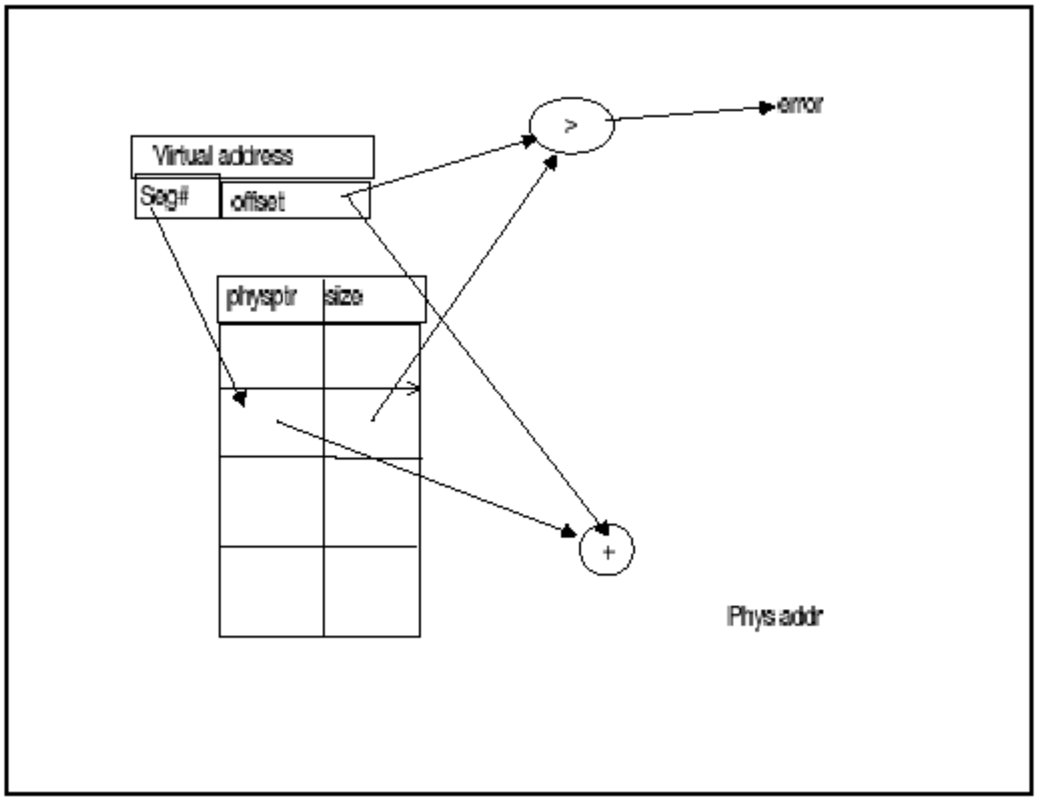
1. **sharing** -- Hard to share between programs
For example, suppose 2 copies of "vi"
Want to share code
Want data and stack to be different.
Cant do this with base and bounds.

2. **relocation** -- Doesn't allow heap, stack to grow dynamically – want to put these
as far apart as possible in virtual memory so they can grow to whatever size is needed.

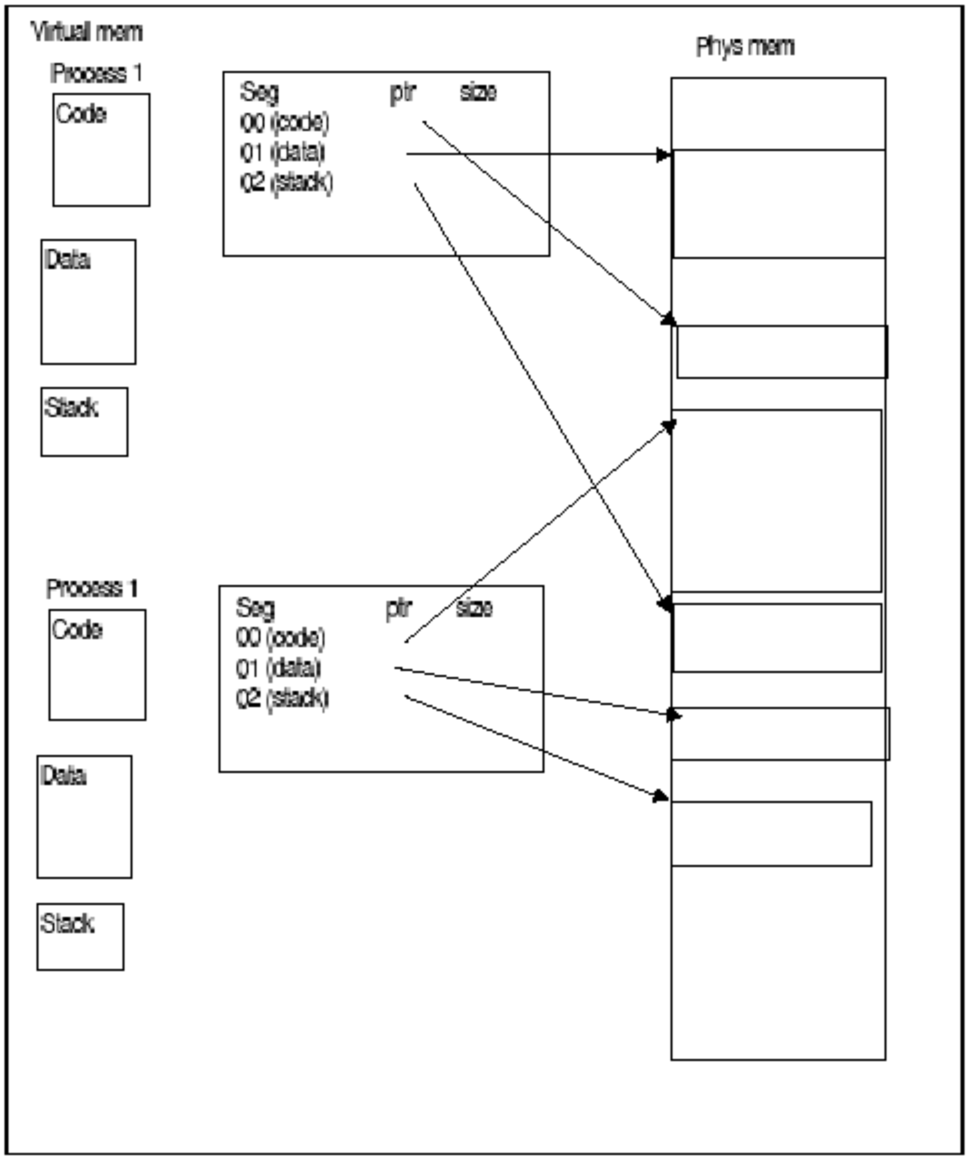
3. Complex memory allocation
see text: *First fit, best fit, buddy system*. Particularly bad if want address space
to grow dynamically (e.g the heap)
In worst case have to shuffle large chunks of memory to fit new Program

7. Segmentation --> Sparse Address Space

segment – variable sized region of contiguous memory
Idea is to generalize base and bounds by allowing a **table** of base and bounds pairs.



View of memory:



This should seem a bit strange: the virtual address space has gaps in it! Each segment gets mapped to contiguous locations in physical memory , but may be gaps between segments.

But a correct program will never address gaps: if it does, trap to kernel and core dump. (Minor exception: stack, heap can grow. UNIX, sbrk() increases size of heap segment. For stack, just take fault; system automatically increases size of stack.)

Detail: need protection mode in segmentation table. For example, code segment would be read-only (only execution and loads are allowed). Data and stack segments would be read/write (stores allowed.)

Implementation
Hardware:
Simple TLB:
Typically, segment table stored in CPU not in memory because it's small.

Software
-- What gets added to PCB?
-- What must be saved/restored on context switch?

QUESTION: How does protection work?, Sharing?

Segmentation pros and cons:

+ **Protection**
+ **relocation** efficient for sparse addr spaces
+ **sharing** easy to share whole segment (example: code segment)
detail: need protection mode bit in segment table – don't let program modify code segment
- complex memory allocation
first fit, best fit, etc
what happens when a segment grows?

8. Paging --> Fixed allocation; TLB

makes memory allocation simple
memory aloc can use a **bitmap**

0 0 1 1 0 1 0 0 0 1 1 1 1 0 0 0 1 0 0 1 0 1

Each bit represents 1 page of physical memory – 1 means allocated 0 means free

Much simpler allocation than base&bounds or segmentation

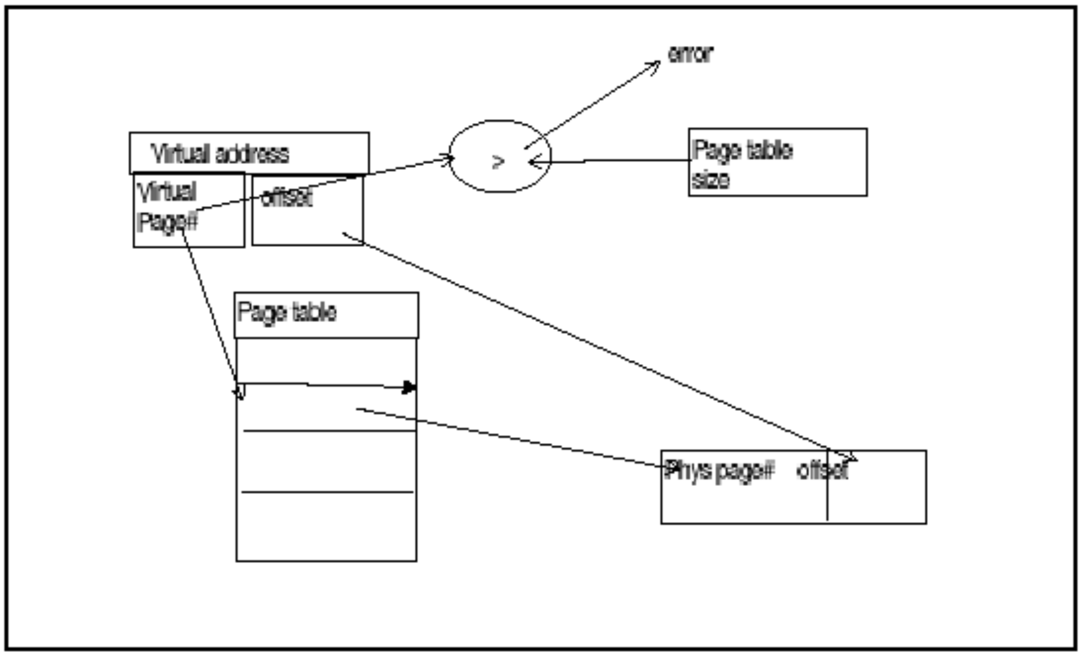
OS controls mapping: any page of virtual memory can go to any page in physical memory.

(Also -- avoids bad corner cases from variable allocation ?)

View of abstraction [[PICTURE]]

VA space divided into pages PA space divided into pages
[still with "segments"]
[PICTURE] [PICTURE]
--> need a table to map/translate each vpage to ppage [PICTURE]

Logical/conceptual view of implementation:



Protection
Relocation
Sharing

Implementation (reality)
~~each address space has its own page table stored in physical memory~~
~~→need pageTablePtr~~

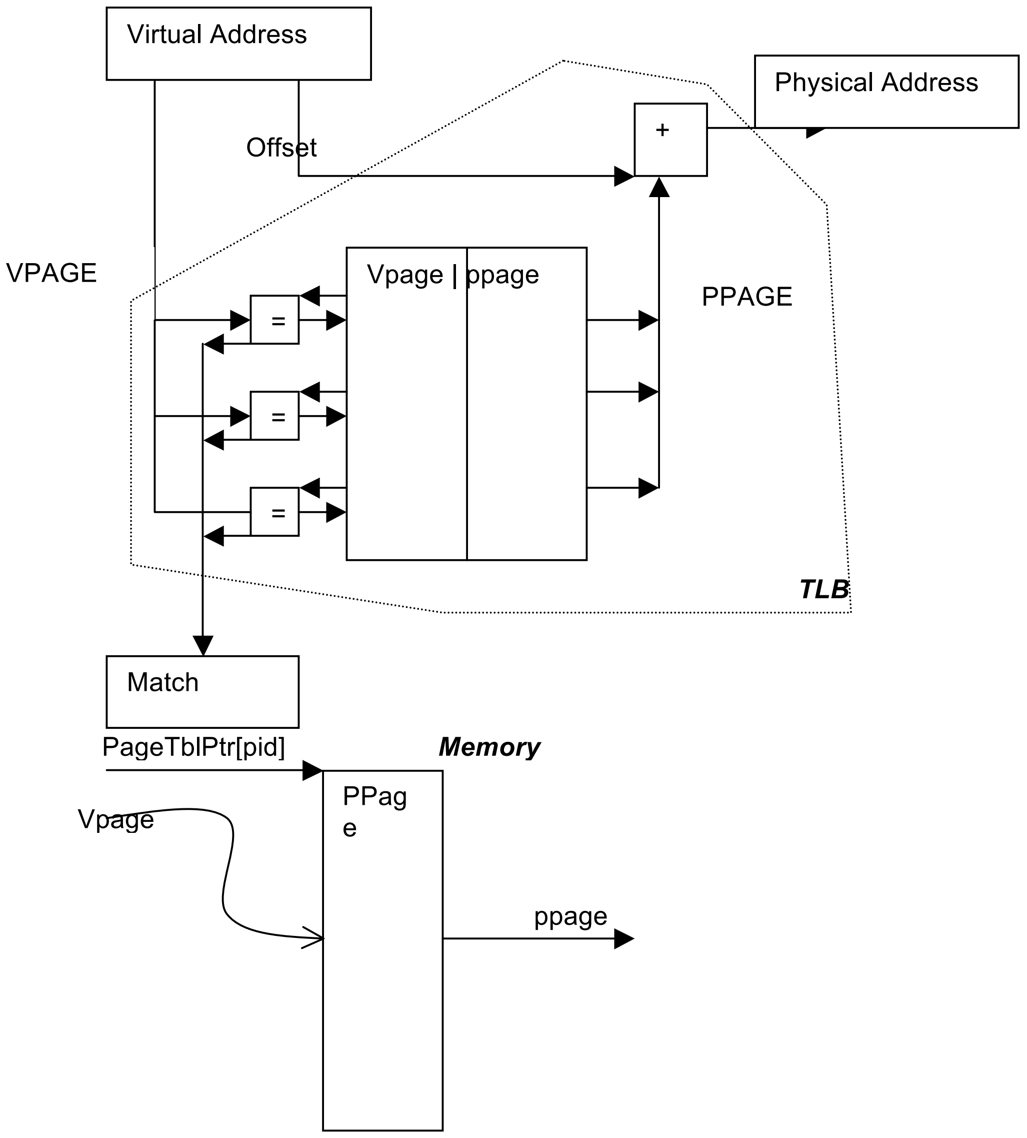
~~pageTablePtr is physical address, not virtual address~~

DA: More complex TLB -- need to split translation between hardware and memory

Problem: Page table could be large
e.g., 256MB process (256 MB VA) with 1KB pages: 256K entries (~1MB)

→ Cannot fit entire page table in TLB (in CPU hardware)

Solution: TLB acts as cache (**real implementation)**



1) each address space/process has its own page table stored in kernel/physical memory
2) Process control block has pageTablePtr
   pageTablePtr is physical address, not virtual address
3) Associative TAG match in TLB hardware

QUESTION: How does this work?

- Hit → translation proceeds
- Miss → memory lookup
  - Either hardware or software controlled

QUESTION: How would each work?

Software TLB miss handling

1) TLB generates trap

2) Drop into OS exception handler and kernel-mode
3) OS does translation (page tables, segmented paging, inverted page table, …)
4) OS loads new entry into TLB and returns from trap

Context switch: Flush TLB
(Or add PID tag to TLB + a CPU register and change PID register on context switch)

Other option: Have HW read page tables/segment tables directly
-- HW includes register pageTablePointer (physical address, not virtual)
-- On TLB miss, HW state machine follows pointer and does lookup in data structure
-- On context switch, change this register (and flush TLB)

- Result of memory lookup: (a) ERROR or (b) translation value

QUESTION: How would you tell the difference?

TLB Design (architecture class review):
Associativity: Fully associative
Replacement: random, LRU, … (SW controlled)

What happens on **context switch**?
Flush TLB
→ new TLB feature – valid bit
QUESTION: what does valid bit mean in TLB? What does valid bit mean in in-memory page table?

1.1 Space Overhead

2 sources of overhead:

1) data structure overhead (e.g., the page table)
2) fragmentation

external – free gaps between allocated chunks
internal – free gaps because don't need all of allocated chunk
segments need to reshuffle segments to avoid external fragmentation
paging suffers from internal fragmentation

How large should a page be?

Key simplification of pages v. segments – fixed size

QUESTION: what if page size is small. For example, vax had a page size of 512 bytes

QUESTION: what if page size is very large? Why not have an infinite page size

Example: What is overhead for paging?

overhead = data structure overhead + fragmentation overhead (internal + external)
*= # entries * size of entry + #"segments" * ½ page size*
*= VA space size / page size * size of entry + #segments * ½ page size*

suppose we have 1MB maximum VA, 1KB page, and 3 segments (program, stack, heap)
*= 2^20 / 2^10 * size of entry + 3 * 2^9*

What is size of entry? Count # of physical pages
E.g. suppose we have a machine with a max 64KB physical memory
64KB = 2^16 bytes = 2^6 pages → need 6 bits per entry to identify physical page

*= 2^10 * 2^6 + 3*2^9 = 2^16 + 3*2^9*

Details: size of entry

a. enough bits for ppage (log2(PA size / page size))
b. should also include control bits (valid, read-only, …)
c. usually word or byte aligned

Suppose we have 1GB physical address space and 1KB pages and 3 control bits, how large is each entry of page table?

2^30 / 2^10 = 2^20 → need 20 bits for ppage
+ 3 control bits = 23 bits
→ either 24 bits (byte aligned entries) or 32 bits (word aligned entries)

~~QUESTION: How does protection work?, Sharing?~~
~~(e.g., address = mmap(file, RO)~~

- ~~how are mappings set up~~

~~■ control bits: valid, read only~~

~~}~~

Evaluation:

Paging

+ simple memory allocation
+ easy to share
- big page tables if sparse address space

Is there a solution that allows simple memory allocation, easy to share memory, **and** efficient for sparse addr spaces?
How about combining segments and paging?

9. Multi-level translation --> A modern approach

Problem -- page table could be huge

QUESTION: what if address space is sparse? For example traditional 32-bit UNIX –
code starts at 0, stack starts at 2^31 - 1

How big is single-level page table for 32-bit VA, 32-bit PA, 1KB page, 6 control bits

-- 2^32 byte virtual address space, 1KB pages --> 2^22 entries --> 2^24 bytes (assuming 4bytes/entry) --> 16MB per table (i.e., 16MB per process)

(And it is worse than these raw numbers suggest -- contiguous memory in kernel!)

How big is single-level page table for 64-bit VA space?
-- 2^64 byte VA space ...

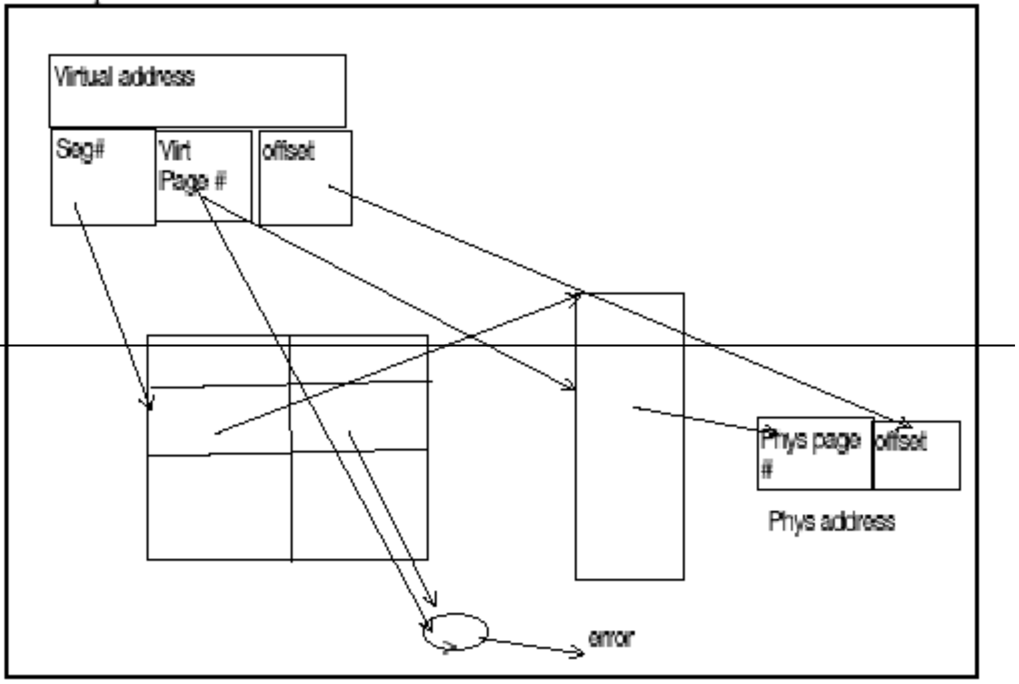
Problem -- address spaces are mostly sparse. Array is a stupid data structure for sparse dictionary!

Use a **tree** of tables (but call it "multi-level page table" or "paged page table" or "segmented paging" or "paged segmentation" to sound impressive; we'll focus on multi-level page tables; others vary in details, but same basic idea **tree)**

~~Lowest level is page table so that physical memory can be allocated via bitmap~~
~~Higher levels segmented or paged (what is the difference? Base v. base + bounds)~~

~~*Multi-level page table* – top levels are page table~~
~~*Paged segmentation* – top level is segmentation, bottom level is paging~~

~~example: 2-level paged segmentation translation~~
~~Logical view:~~



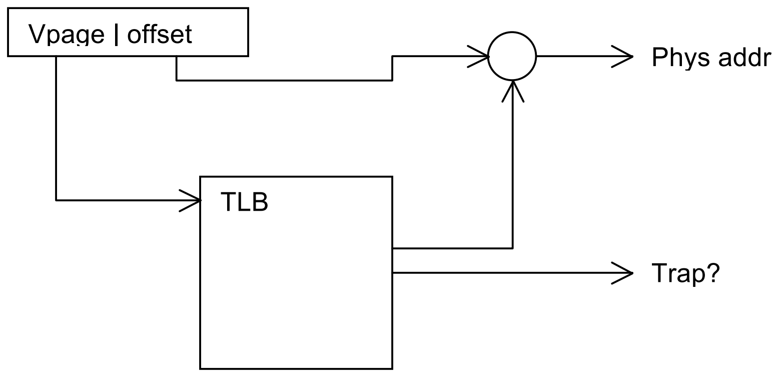
~~Just like recursion, can have any number of levels in tree~~
~~Question: what must be saved/restored on context switch?~~
~~Question: How do we share memory?~~
~~(Can share entire segment or single page)~~

~~Question: Above shows logical picture. Add a TLB – does the TLB care about segments?~~

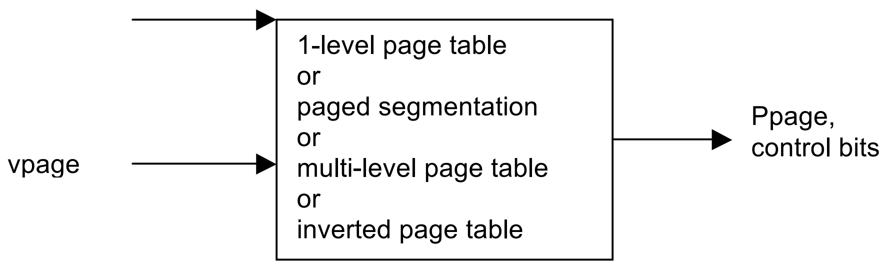
~~No – from TLB point of view,~~
~~address = <virtual page number, offset>~~
~~The virtual page number happens (in this case) to be organized as "seg, vpage" for when we look in memory, but TLB doesn't care~~

~~→ flexible software translation~~

~~Hardware is always:~~



~~Memory data structure is opaque object:~~

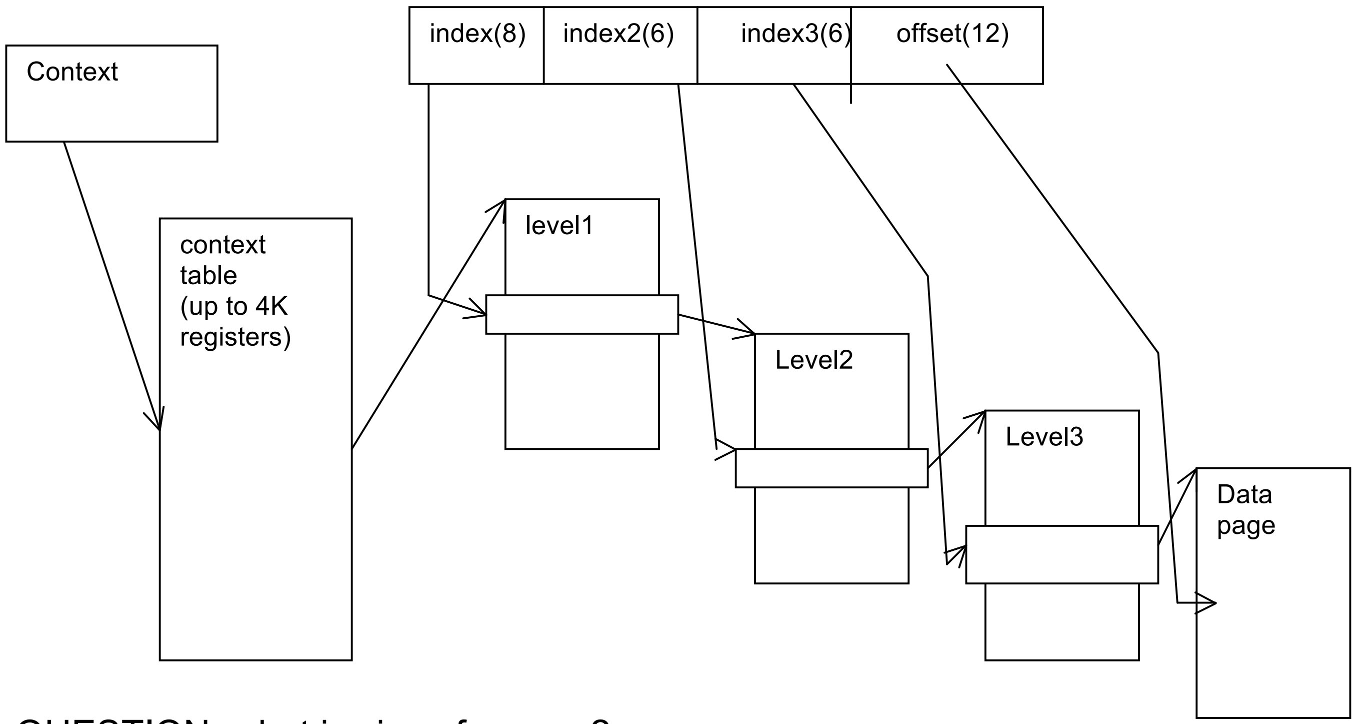


~~Evaluation:~~
~~The problem with page table was that it was inefficient (space) for sparse address spaces. How does paged segmentation do?~~

~~Multilevel translation pros & cons~~
~~+ **protection, page-level sharing, relocation and simple/cheap implementation**~~
~~+ only need to allocate as many page table entries as we need~~
~~+ easy memory allocation~~
~~+ share at segment or page level~~

~~- pointer per page (typically 4KB - 16 KB pages today)~~
~~- two (or more) hops per memory reference (TLB had better work!)~~

~~Multilevel page table~~
~~Example: SPARC (*slide*)~~



~~QUESTION: what is size of a page?~~
~~QUESTION: what is size of virtual address space?~~
~~Assume 36-bit physical address space (64 GB) and 4 protection bits~~
~~QUESTION: What is size of top-level page table?~~
~~What is size of 2nd level page table?~~
~~What is size of bottom level page table?~~
~~QUESTION: for a Unix process with 3 "segments" – data 64KB, stack 13KB, code 230KB, what is space overhead?~~
~~QUESTION: what is largest contiguous region needed for a level of the page table?~~

~~Note:~~
- ~~Only level 1 need be there entirely~~
- ~~second and third levels of table only there if necessary~~
- ~~three levels is "natural" b/c never need to allocate more than one contiguous page in physical memory~~

~~QUESTION: what needs to change on context switch?~~

~~Evaluation: multi-level page table~~

- ~~good protection, sharing~~
- ~~reasonable space overhead~~
- ~~simple allocation~~
- ~~DA: several memory reads needed per memory access (hope TLB solves)~~

NEXT TIME -- details
(1) Quantifying overheads
(2) Case studies -- x86
(3) Other approaches

****

Summary - 1 min

****

Goals of virtual memory:

- protection
- relocation
- sharing
- illusion of infinite memory
- minimal overhead
  - space
  - time