Lecture #12: Address translation

****

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	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

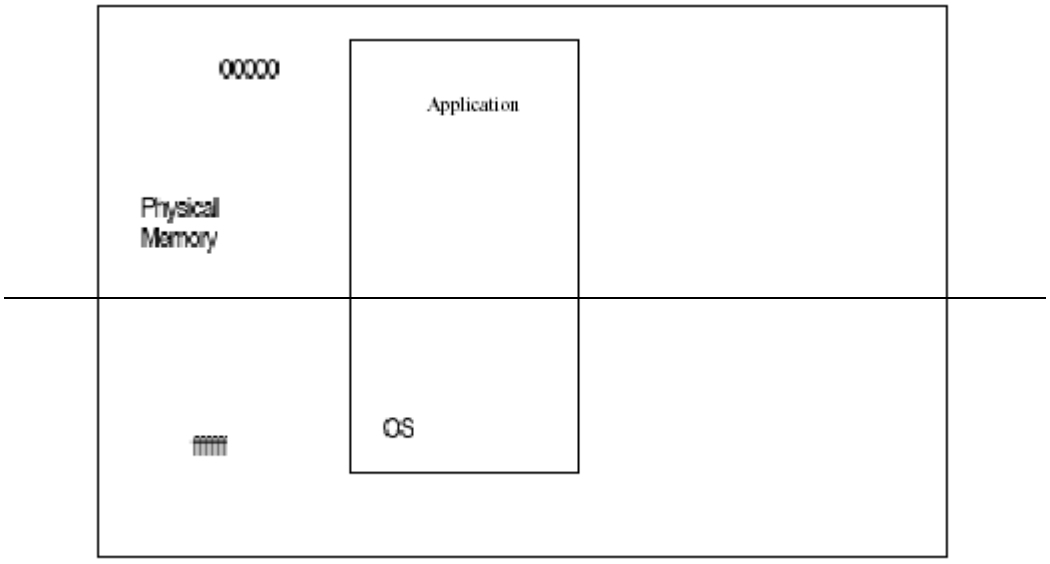
That all sounds great, but how do I do it?

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

~~2.1 Uniprogramming w/o protection~~

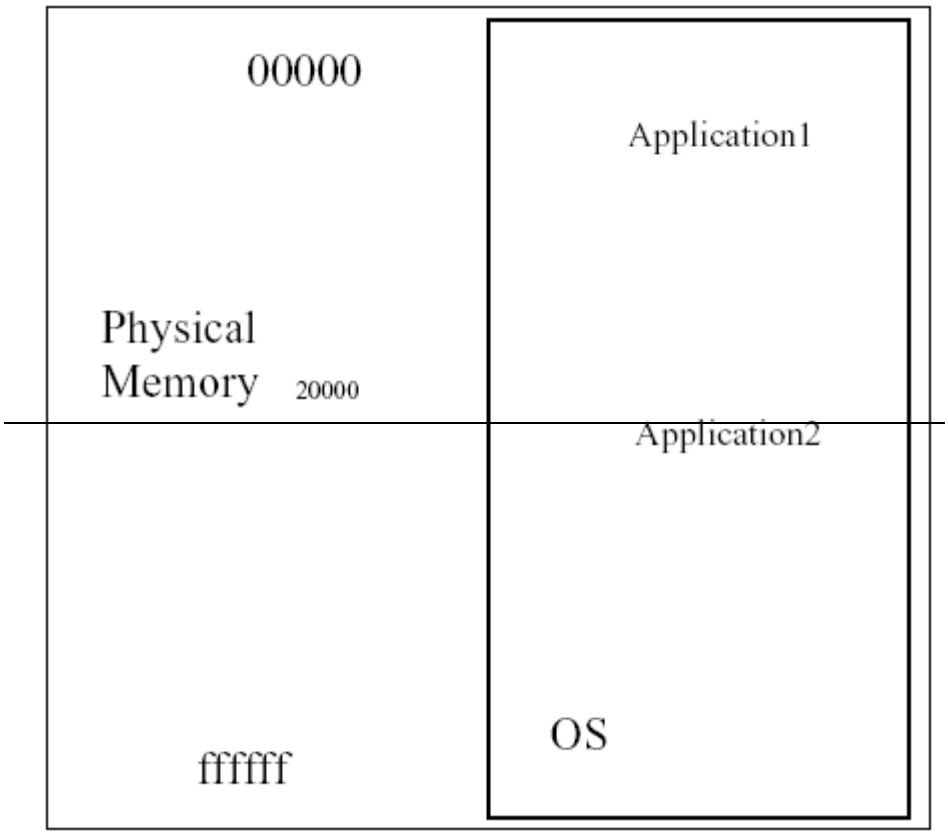
~~Personal computer OS's~~
~~Application always runs at the same place in physical memory~~
~~Each application runs one at a time~~
~~->give illusion of dedicated machine by giving reality of dedicated machine~~

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



~~2.2 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

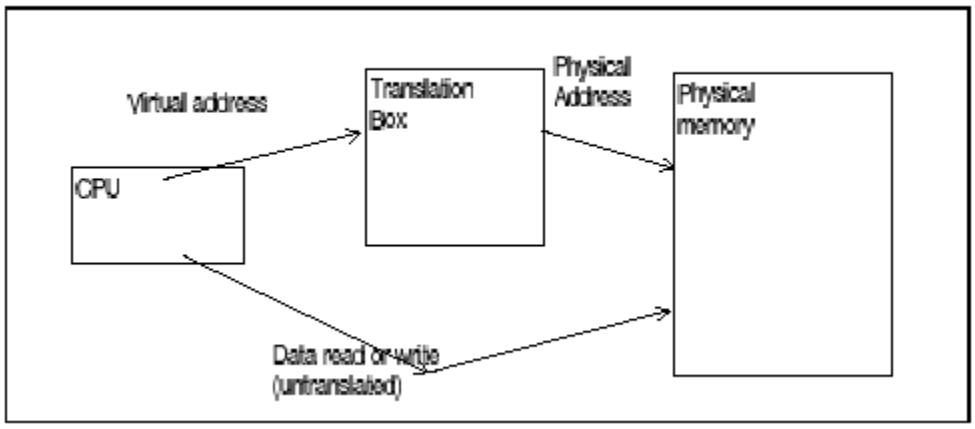
Goal of protection:

- 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:

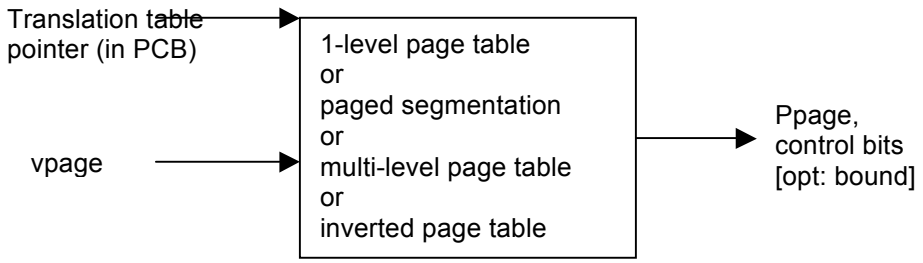
segment maps variable-sized range of contiguous virtual addresses to a range of contiguous physical addresses

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

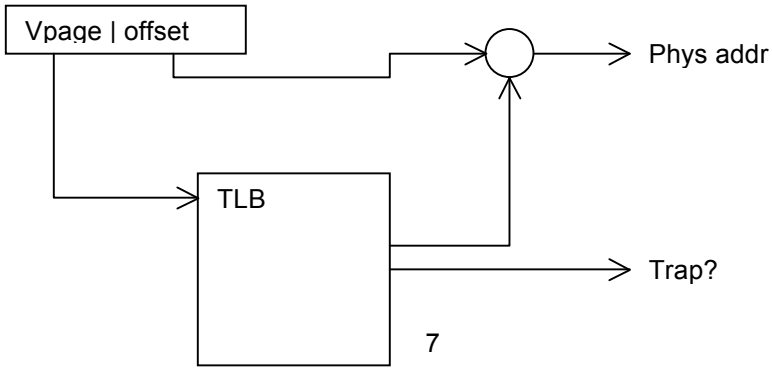
need data structures to lookup page/segment mapping given a virtual address
<segment #> → segment info {base, size}
<page #> → page info {base}

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:



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

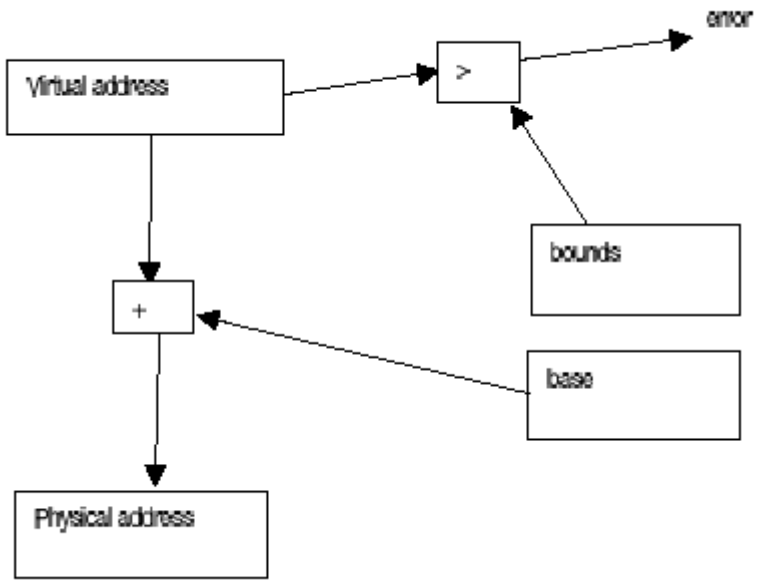
-- start with simplest!
****
Admin - 3 min

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

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

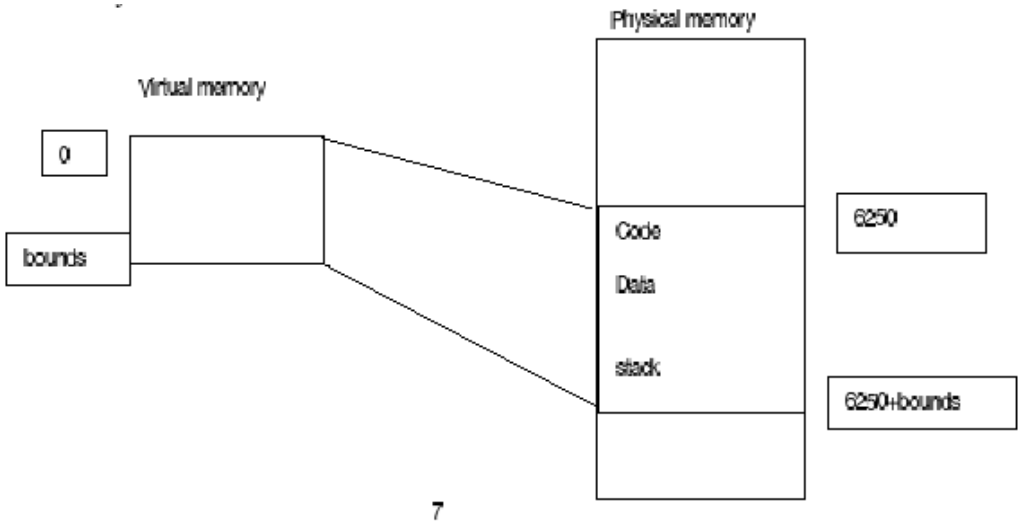
6. Base and bounds

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?

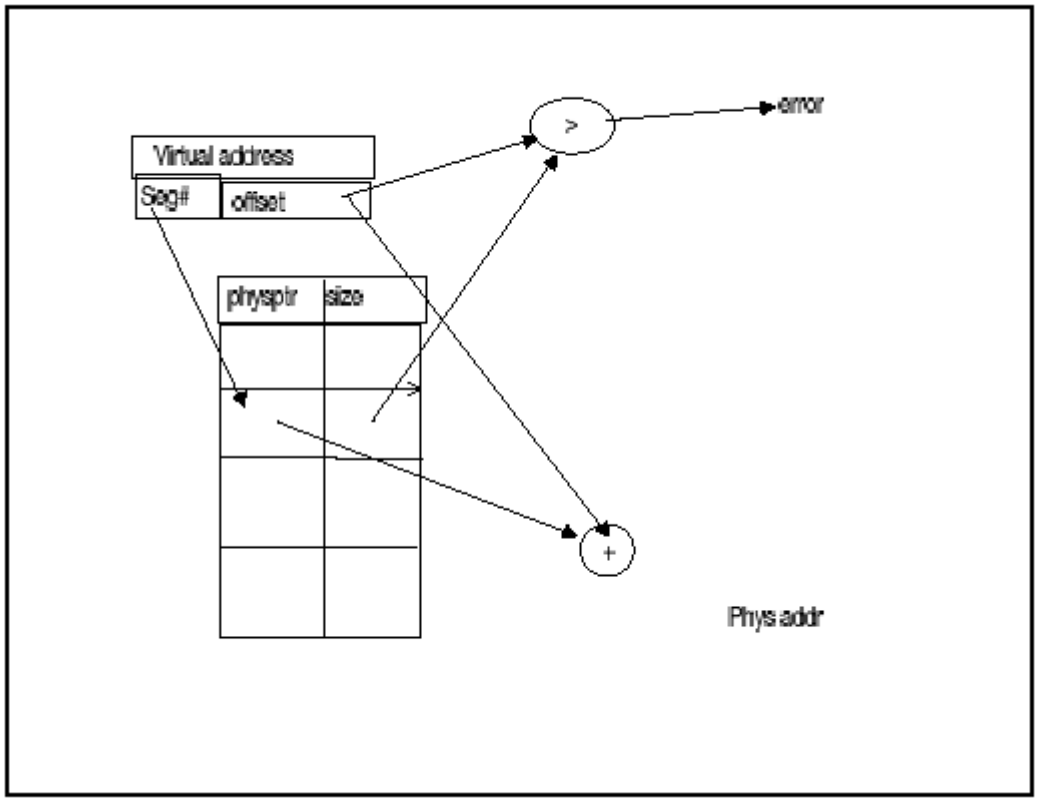
Evaluation
Base and bounds pros:
+ **protection**
+ simple, fast
Cons:
1. **sharing** -- Hard to share between programs
For example, suppose 2 copies of "vi"
Want to share code
Want data and stack to be different.
Cant do this with base and bounds.

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.

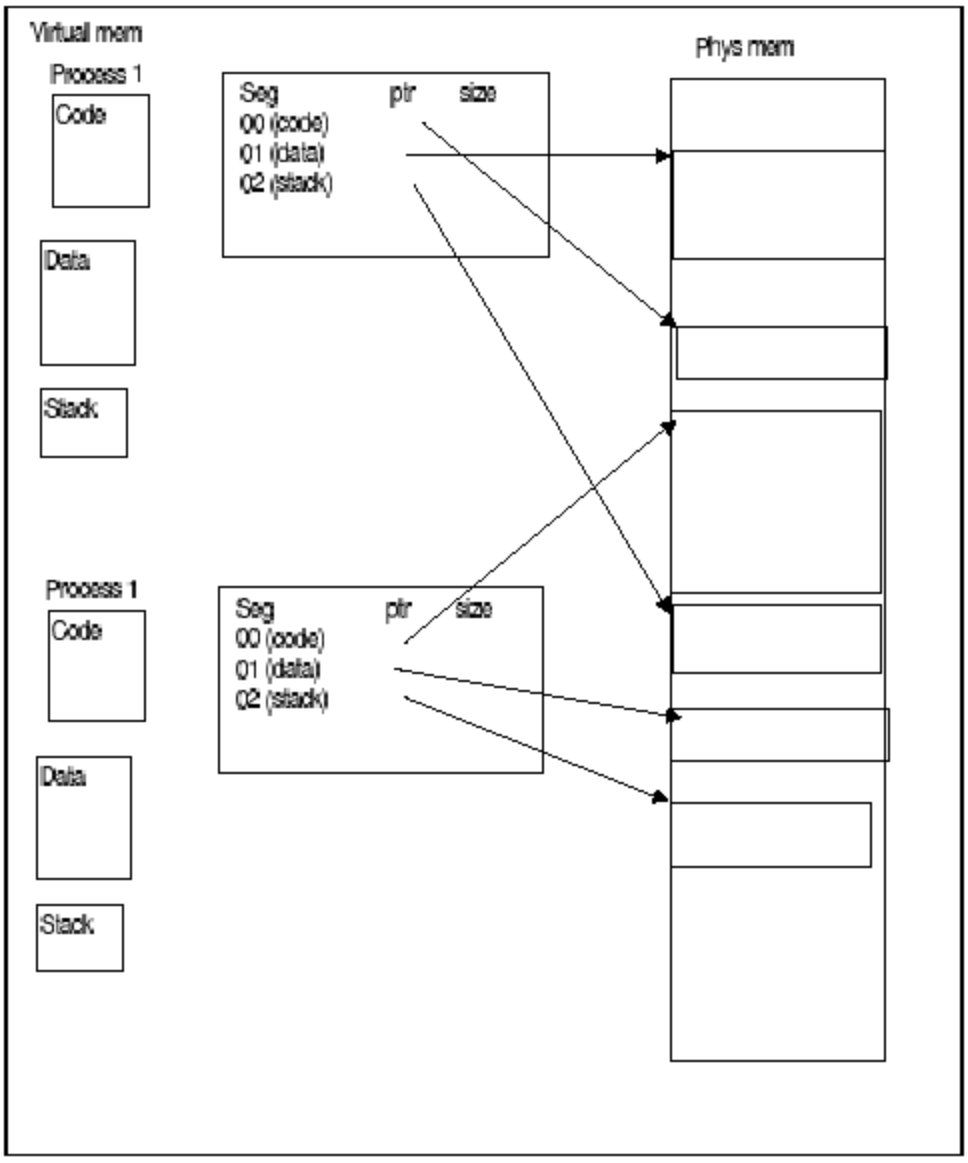
3. Complex memory allocation
see text: *First fit, best fit, buddy system*. Particularly bad if want address space
to grow dynamically (e.g the heap)
In worst case have to shuffle large chunks of memory to fit new Program

7. Segmentation

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

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

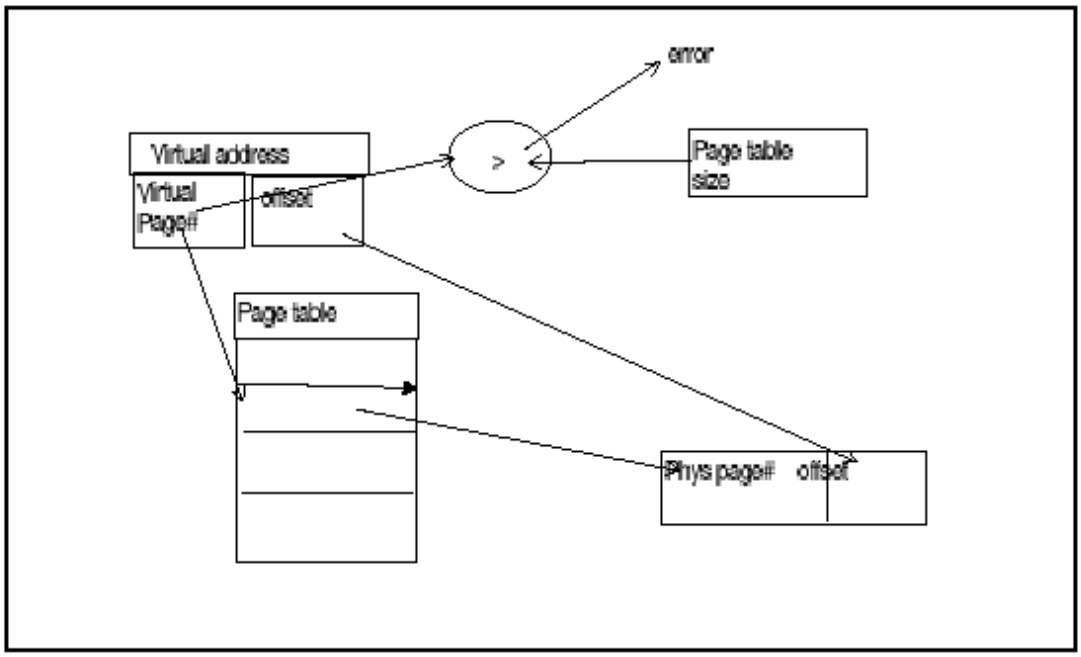
Each bit represents 1 page of physical memory – 1 means allocated 0 means free

Much simpler allocation than base&bounds or segmentation

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

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~~

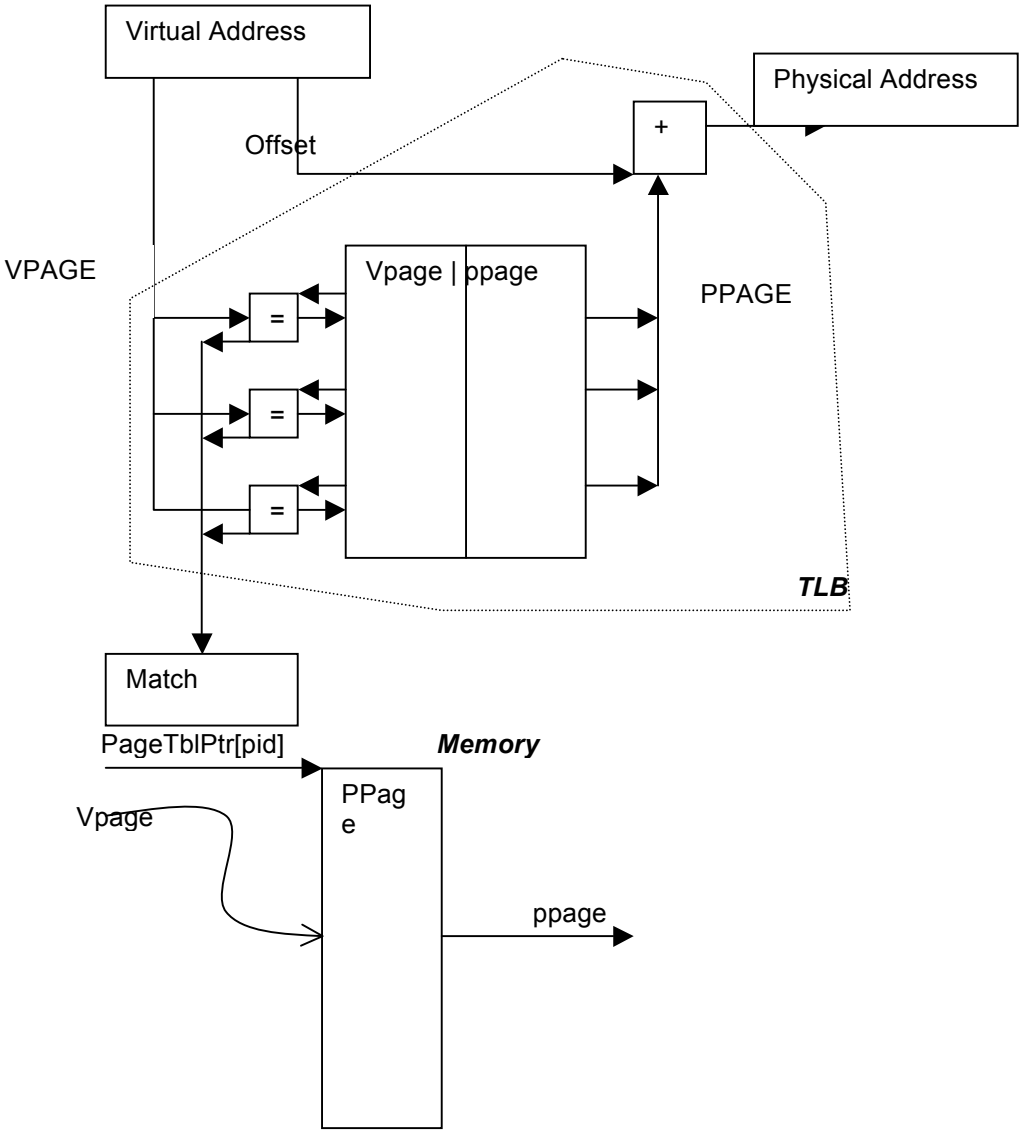
~~pageTablePtr is physical address, not virtual address~~

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

Problem -- page table could be huge

QUESTION: what if address space is sparse? For example 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!

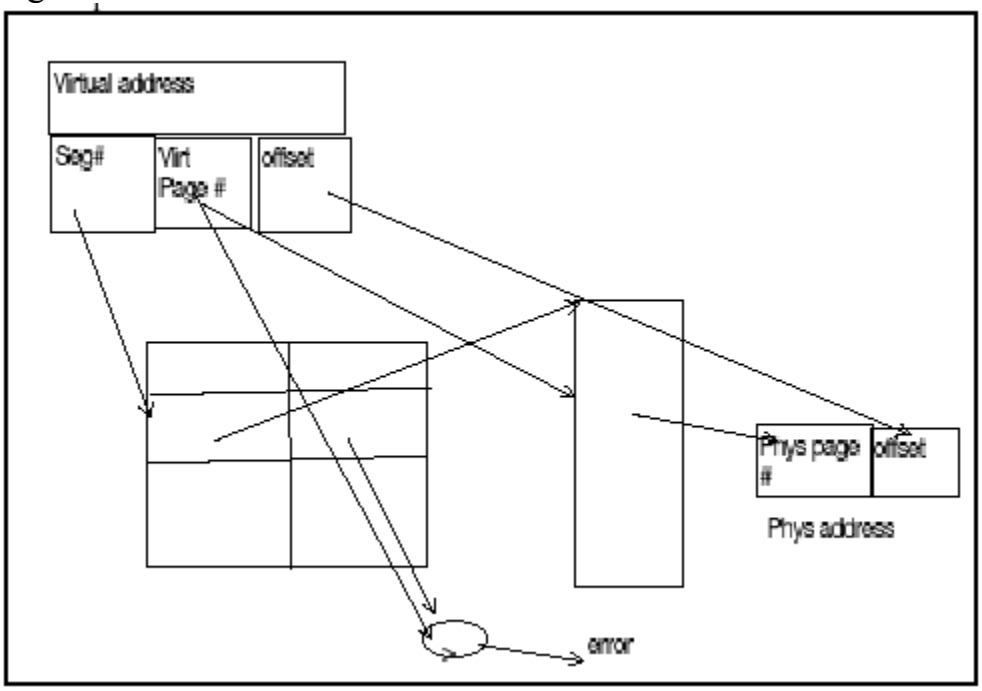
Use a **tree** of tables (but call it "multi-level page table" or "paged page table" or "segmented paging" or "paged segmenation" to sound impressive; we'll

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)

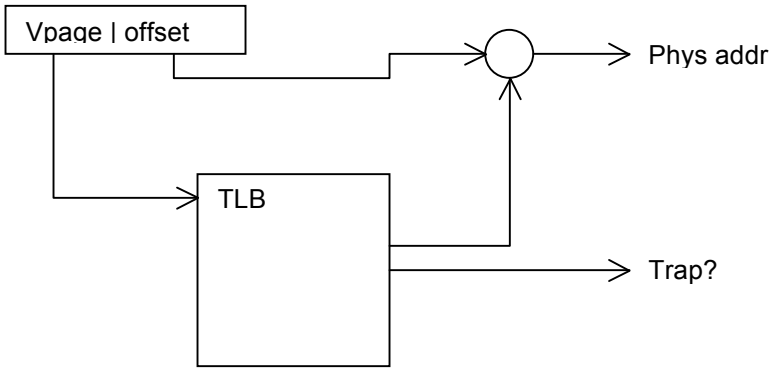
Question: Above shows logical picture. Add a TLB – does the TLB care about segments?

No – from TLB point of view,
address = <virtual page number, offset>

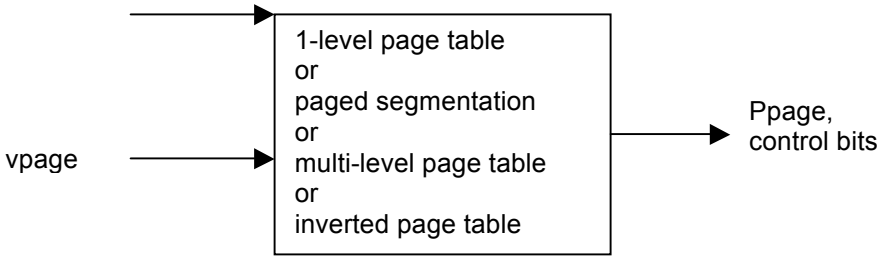
The virtual page number happens (in this case) to be organized as "seg, vpage" for when we look in memory, but TLB doesn't care

→ flexible software translation

Hardware is always:



Memory data structure is opaque object:

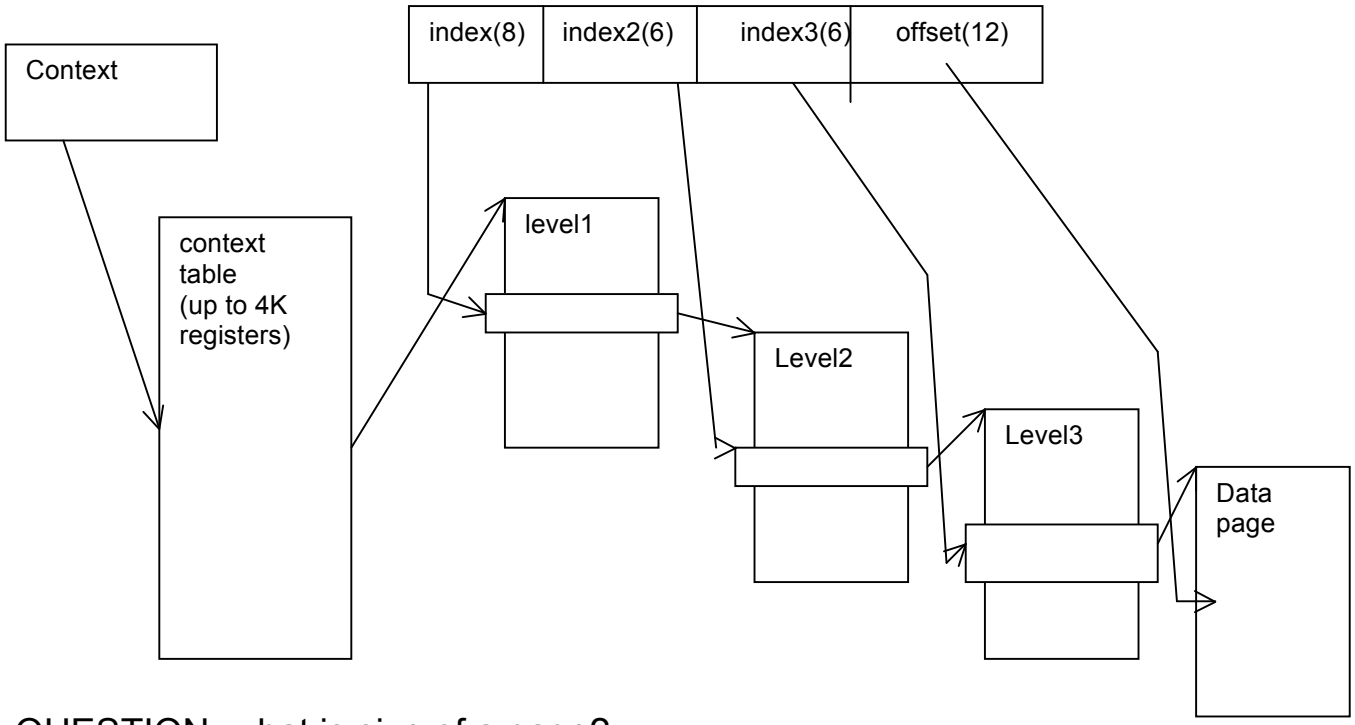


Evaluation:
The problem with page table was that it was inefficient (space) for sparse address spaces. How does paged segmentation do?

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!)

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

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)

Example: x86 paging (basic 32-bit)
(Actually combine segments and paging, but we'll just look at paging part for now…see whole story in a page or so…)

32-bit virtual address ("linear address")
32-bit physical address
Linear address has 3-part structure as follows

```
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table    | Offset within Page  |
// |      Index     |      Index      |                     |
// +----------------+-----------------+---------------------+
```

How big is a page? 4KB

2 level page table [[PICTURE]]

- Top level: "Page directory" – how big is a page directory?
- Second level: "Page table" – how big is a page table?

A page table or directory entry is 4 bytes
How should we format a page table entry?
X bits of ppage
Y bits of control (valid, read, write, …)

Example: x86 protected mode – the whole story

virtual addr *– seg table -->****linear address*** *– page table -->****phys addr***

seg#, 32-bit offset 32 bit 32 bit

Why? Mostly historical. Both Linux and JOS install IDENTITY mapping to seg table so that virtual address == linear address

x86 Segment translation:

- GDTR register points to GDT table
  - Base address and limit (# of entries in GDT table)
- Global Descriptor Table (GDT) has array of [base, bounds, permissions]
- Instructions specify segment # → index into GDT
  - Segment number often implicit – esp uses SS, EIP uses CS, othes mostly use DS
  - `ljmp selector, offset` changes EIP to explicitly specified segment # and offset within that segment
- la = offset + GDT[seg#].base; assert offset < GDT[seg#].limit

x86 paging translation

- CR0 register points to page table (physical address)
- TLB miss: lookup la in 2-level page table to get PA

For project, once we get paging set up, we will "turn off" segmentation by setting base = 0, limit = 2^32-1 for all segments.

Example: 64-bit address space (naive)

Assume 64-bit PA, 64-bit VA, 8 control bits, 8KB pages.

Q: How many levels of multi-level table?

A: 8 bytes per entry (54 ppage + 8 control + round up to word)

2^10 entries per page (2^13bytes/page/2^3bytes/entry)

--> 54 vpage / 9 = 6 levels (top one one full)

Suppose I have 3 regions -- 0..54MB, 256-263MB, 1GB-1GB+1KB

How much space overhead?

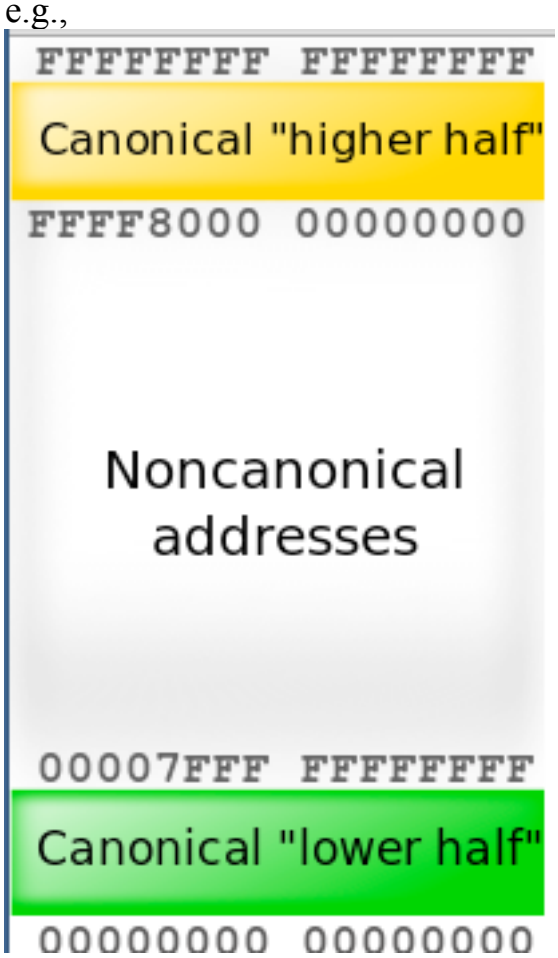
-- data structures

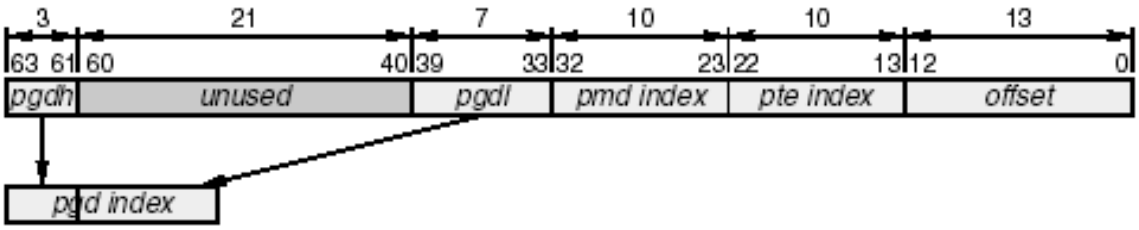
-- external fragmentation

-- internal fragmentation
-- percentage of total space?

Example: x86-64 long mode
64-bit registers
43-bit VA (8TB), 40-bit PA (1TB), 8KB page
--> reduce memory overhead, cache/TLB tags
--> Addresses are still 64 bits, so easy to expand later...
 -- kernel knows what regions are "legal
 -- user processes never see PA
 -- user processes ask for "X bytes of VA" and kernel picks where

Useful to let kernel be at "high" addresses and user at "low" addresses
e.g.,



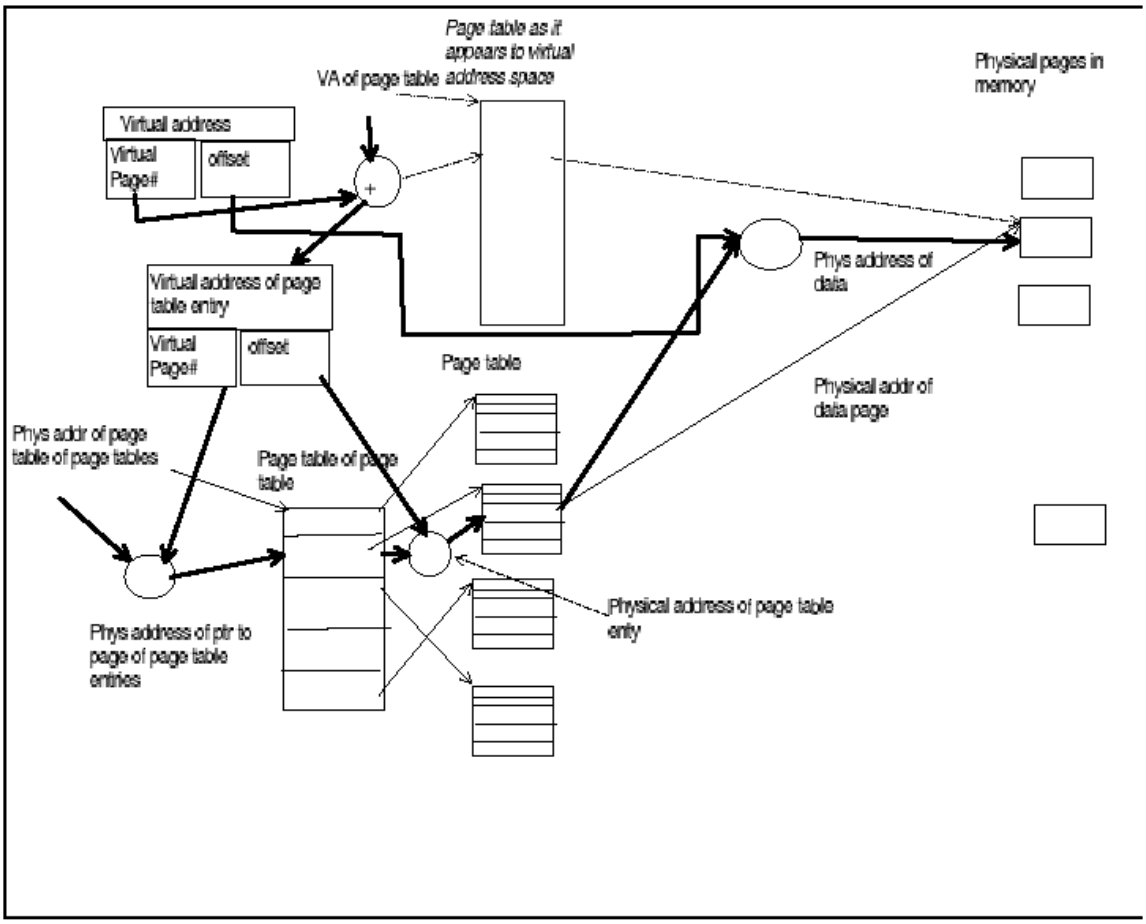


How many levels?
-- still have

10. Paged page tables

Another way to solve sparse address spaces is to allow page tables to be paged; only allocate physical memory for page table entries you actually use

Top level page table is in physical memory (stored in contiguous phys mem)
all lower levels are in virtual memory (stored in contiguous virt mem but in fixed-sized frames in physical memory)



notice – although page table stored in virtual memory, it is stored in kernel virtual address space, not process virtual address space (otherwise process could modify it)

Problem: every memory reference takes 3 memory references (one for system page table, one for user page table, one for real data)

How do we reduce the overhead of translation? Caching in **Translation Lookaside Buffer** (TLB)

Relative to multilevel translation with segments, paged page tables are more efficient if using a TLB: if VA of page table entry is in TLB can skip one or more levels of translation

11. Inverted page table

What is an efficient data structure for doing lookup?
A: Hash table

Why not use a hash table to translate from virtual address to physical address?

This is called an inverted page table for historical reasons

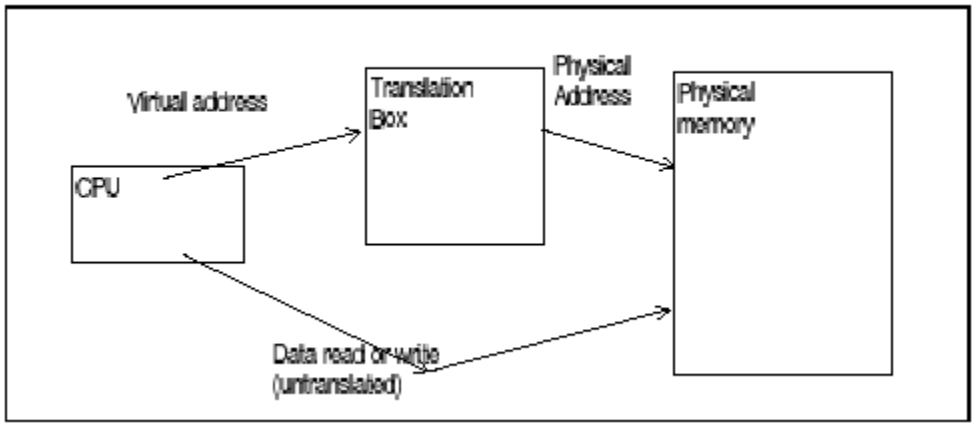
Take <pid, virtual page #>, run hash function on it, index into hash table to find page table entry with <tag, physical frame #>

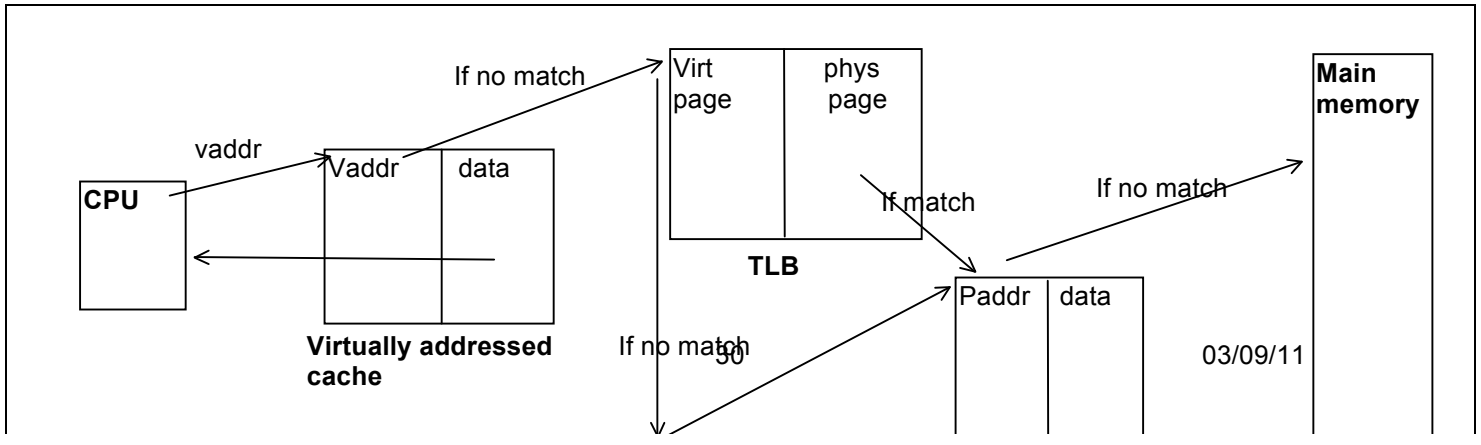
QUESTION: why do you need PID?
QUESTION: what needs to be in tag?

Advantages
• O(1) lookup to do translation
• Requires page table space proportional to size of how many physical pages actually being used, not proportional to size of address space – with 64-bit address spaces, a big win
• DA: overhead of managing hash chains etc

2. The big picture:

Abstraction:





3. Time overhead

AMAT (average memory access time)
AMAT = T_L1 + P_L1miss * T_L1miss
T_L1Miss = T_TLB + P_TLB_miss * T_TLB_miss
+ T_L2 + P_L2miss * T_mem
T_TLB_miss = # references * T_L2 + P_L2miss * T_mem

Are TLBs a significant source of overhead?
What are reasonable estimates for T_TLB_miss?
What hit rates do you need to keep TLB overhead manageable?

Notice: TLB performance matters a lot
TLB miss --> 1 memory reference turns into 2, 3, 4 (to do lookup + the reference itself)

Old study (mid/late 90s ... have not seen newer one...I'm sure they exist...probably not a huge change...maybe a bit worse....) -- easy to find applications that spend 5-10% of their time on TLB fills

Optimization: Superpage

Some large contiguous regions of memory get repeatedly accessed as a unit

e.g., bitmap display -- [simplified from realty] -- suppose you have a display with 1024 x 1024 pixels and 4-bytes per pixel, and suppose that your CPU can write values to a specified 4MB region of memory and your graphics card reads that region of memory and renders it on the screen; suppose you are doing some full-screen application, so that your CPU repeatedly re-writes the entire 4MB region from start to end. Assume 4KB pages.

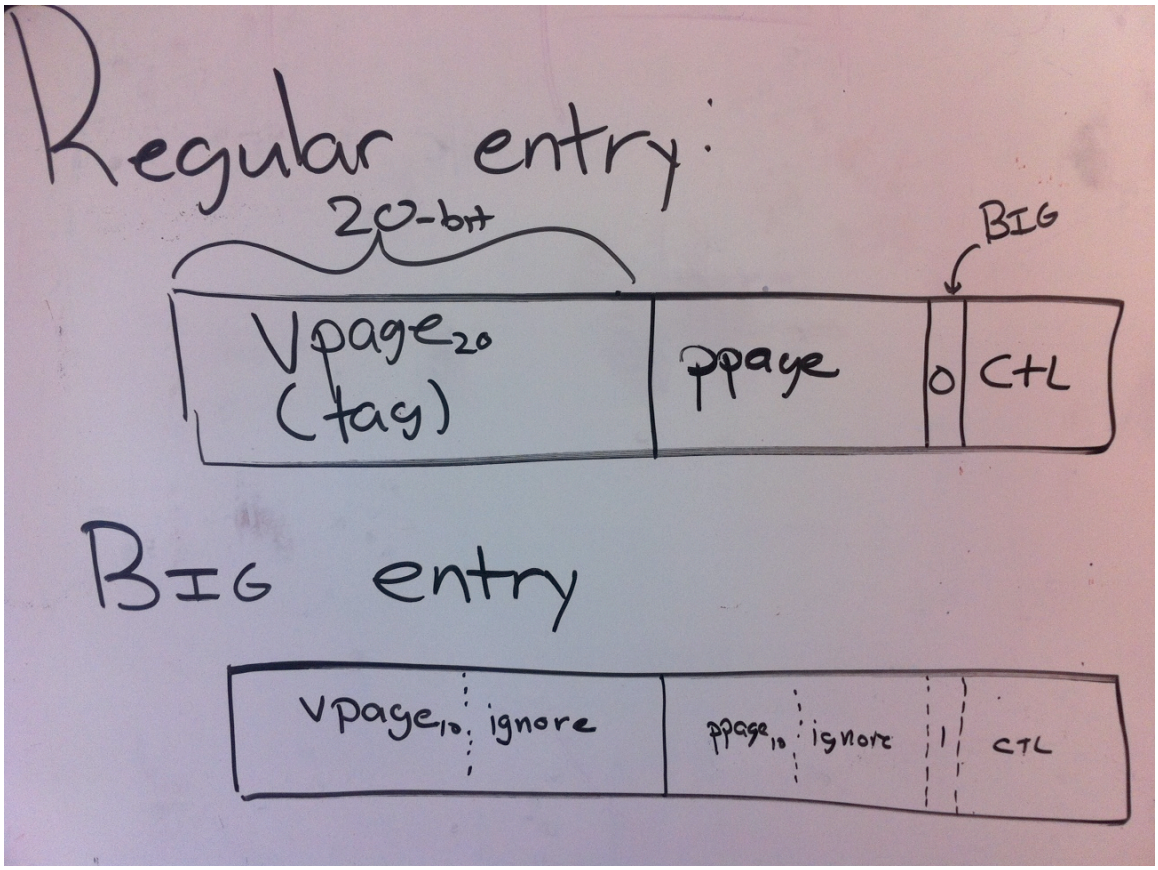
--> 4MB / 4KB/page --> 1024 pages
Typical TLB 32-64 entries
--> every time you refresh the screen you flush the TLB

Solution: superpage
Allow a TLB entry to have its "BIG" bit set (could be several bits for several sizes)

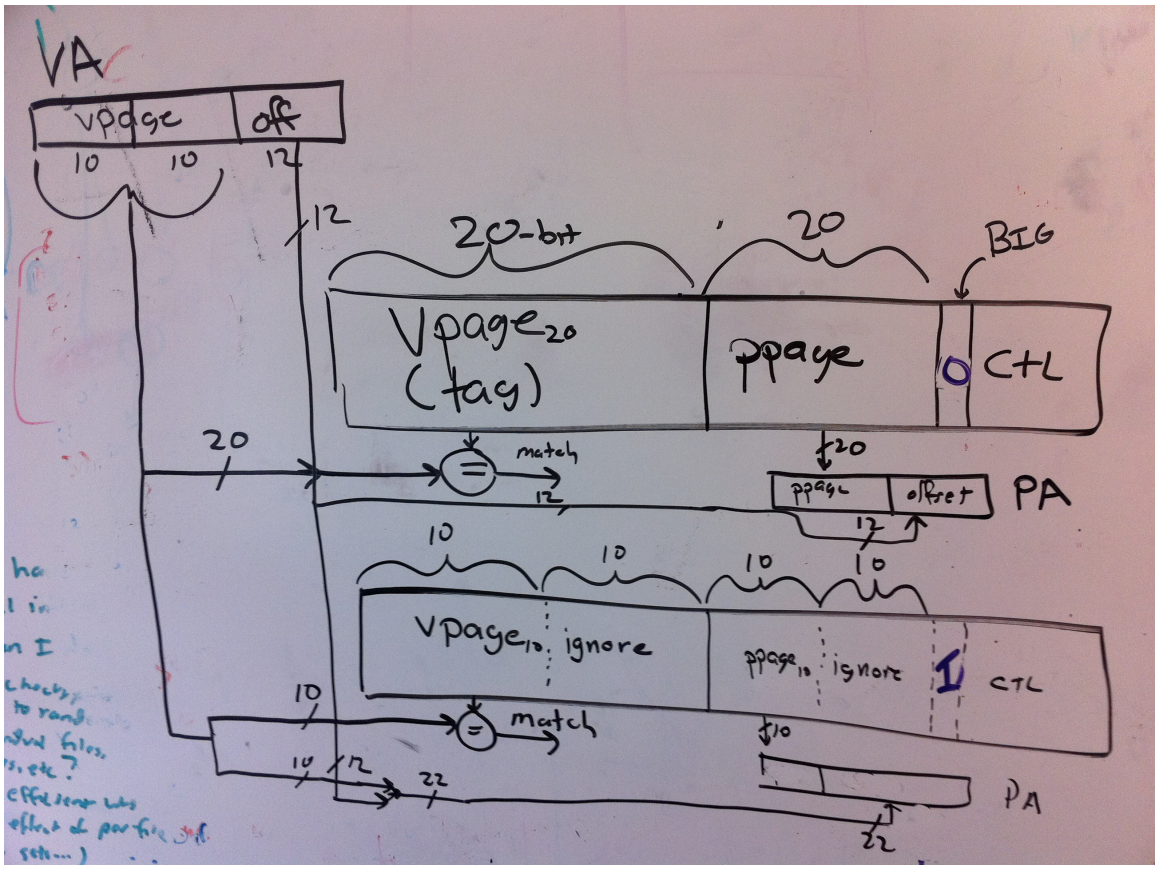
Now one entry in TLB can cover a lot more ground
(Note that both VA and PA range must be contiguous)

Example.
32-bit VA space, 32-bit PA space, 4KB "normal" pages (2^12 bytes), 4KB "superpages" (2^22 bytes)

TLB entry:



TLB: VA->PA with superpages



What do you need in in-memory page table to support this TLB hardware extension?

Simple answer: Just a "BIG" bit in bottom level entry of page table.

-- Allocate a large, aligned, contiguous region of (virtual and physical) pages
(e.g., 1024 contiguous 4KB pages aligned on 4MB boundary)
-- Load all normal page table entries, but set BIG bit
(e.g., 1024 entries; each contains full 20-bit ppage number as normal, so you could treat it as a "normal" entry)
-- On TLB miss to *any* page in the region, the "normal" entry is found and loaded to TLB, but since BIG bit is set, the TLB ignores the bottom 10 bits of vpage and ppage --> any entry from "BIG" region will match any VA in "BIG" region and calculate proper PA
(neat, huh?)

[[PICTURE]]

Other answers
-- In this case, we happen to have the superpage size exactly match the size indexed by a top-level page table entry (4MB)
-- Could set an "I am a superpage" bit in top level page table, and load it directly to TLB instead

NOTE:
Above is simple example
-- Can have larger or smaller superpage size
(Superpage size matched top-level entry in above example; cooincidence; not required)
-- Can have multiple superpage sizes (e.g., not just 2 sizes -- BIG/NORMAL but 4 sizes -- NORMAL, MED, BIG, VERYBIG)

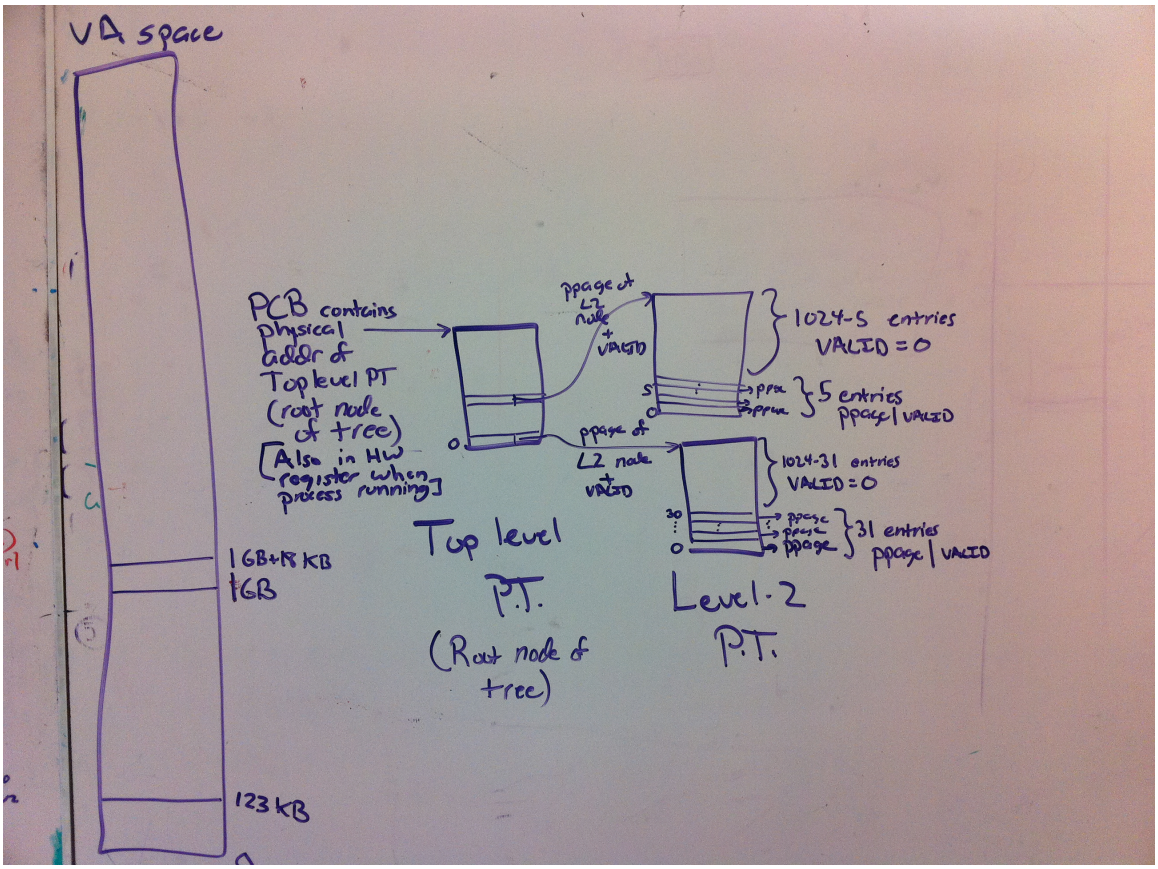
4. Space overhead

What are sources of space overhead?
Internal fragmentation
External fragmentation
Data structures (page tables, etc.)

4.1 All parameters give --> just math

overhead = data structure overhead + fragmentation overhead (internal + external)

Example.
Suppose you are using 2-level page table in 32-bit x86. Pages are 4KB (2^12 bytes). Top level and second-level nodes of page table are 1024 entries. Each entry is 4 bytes. A process has 2 logical segments. Segment 1 starts at address 0 and is of size 123 KB, so it consumes 1KB less than 31 pages. Segment 2 starts at address 1GB and is of size 18KB so it consumes 2KB less than 5 pages. To accommodate these two segments, the 2-level page table has 1 top-level node and two second-level nodes.



How much memory overhead is there?

internal fragmentation = 1KB + 2KB = 3KB
external fragmentation = 0
data structure: 3 pages * 4KB/page = 12KB
total overhead = 15KB

What is the overhead as a percentage of the process's virtual memory consumption?

virtual memory consumption = 123KB + 18KB = 141KB
percentage overhead = 15 KB/ 141 KB * 100 = 10.64%

4.2 Some parameters missing / open design problem --> infer from what is given --> may require design/engineering judgment

internal fragmentation:

paging: 1/2 page per contiguous region of memory

(Reasonable estimate of wasted space because of fixed allocation size (page) v. variable allocation desire (region of memory/"logical segment")

"Contiguous region of memory" -- e.g., stack, heap, code, memory mapped files, shared memory region.

> Probably number of regions will be given (or measured in a workload)
>
> If not, make a reasonable estimate. (How many barbers in austin?) 3-10 regions seems plausible...

segmentation: 0*

Simple answer: Variable allocation size matches variable allocation desire

*You might allocate extra space to accommodate growth. See, for example "base and bounds" example where stack and heap grow towards each other in same allocation unit --> need to pre-reserve extra space. Another example: Since it is inconvenient to keep growing a segment, you might allocate your stack segment larger than you initially need to avoid the need to grow it (maybe move it) in the future.

Notice how paging based schemes avoid this cause of internal fragmentation.

external fragmentation

paging: 0

All allocation units interchangable, so all memory chunks outside of currently allocated memory is usable

segmentation: >0

I don't know of a good way to estimate this a priori.

- Depends on allocation strategy (first-fit, best-fit, buddy, ... See silbershatz).
- Depends on history of past requests

In reality, you would probably have to run a bunch of simulations under various workloads to estimate this

data structure overhead

Nentries * size of entry

This is the semi-tricky one -- you need to understand the data structures.

(1) Size of entry – depends on

- (a) size of PA space (enough to address all physical pages)
- (b) number of control bits
- (c) size of machine word (usually word or sometimes byte aligned – round up)

(a) Size of PA space

if 2^PA bytes of PA space and page size is P, then 2^(PA-P) physical pages ("frames" or "page frames" or "physical frames") --> need PP = PA-P bits to identify physical page

Example: 32-bit PA and 4KB (=2^12 bytes) pages, 2^20 physical pages; ppage number is 20-bit number

(b) Number of control bits

Examples:

- valid
- writable (v. read-only)
- user-may-access (v. kernel-only)
- executable
- copy-on-write
- dirty
- accessed
- reserved for future use
- ...

Some subset of these on any given architecture.

(c) Round up to byte or word

Manage complexity; optimize performance...

(2) Nentries – depends on

- data structure (array v. tree v. hash v. ...),
- size of VA space (need to be able to map all VAs...)
- population of VA space (tree + hash support sparse addr space)
- [exception: inverted page table depends on size of PA size]

--> Details depend un understanding allocation data structure (page table v. multi-level v. inverted page table v. ...)

e.g., Multi-level page table

If you are given number of levels, figure out what internal nodes are populated in order to reach in-use/allocated areas of VA space.

Example: (As above.) Suppose you are using 2-level page table in 32-bit x86. Pages are 4KB (2^12 bytes). Top level and second-level nodes of page table are 1024 entries. Each entry is 4 bytes. A process has 2 logical segments. Segment 1 starts at address 0 and is of size 123 KB. Segment 2 starts at address 1GB and is of size 18KB.

Answer: This is exactly as above with some details missing, but what is here is sufficient to exactly calculate the above...

Each top-level PT has 1024 pointers. Each pointer points to a second level PT that has 1024 pointers. Each second level PT entry points to a physical page. --> Each top level PT indexes 1024 * 1024 * 4096 = 4MB of virtual memory space.

Check: 1024 entries * 4MB = 4GB = 2^32-bit VA space

So...(after verifying size and alignment) segment 1 uses entry 0 (only) of top-level PT and requires 1 (only) level 2 node.

So...(after verifying size and alignment) segment 2 uses entry 256 (only) of top-level PT and requires 1 (only) level 2 node.

--> nentries = 3 * 1024 (1 top level and 2 second-level nodes, each of which has 1024 entries)

If not given number of levels...
What are reasonable assumptions?
Design question -- how to design number of levels?

Need to index full VA space. If 2^X-bit VA space and 2^Y-bit page, then need V = X-Y-bit vpage number --> V-bit index

V-bit index can be divided into
- 1-level page table with 2^V entries
- 2-level PT with 2^P1 entries in top level and 2^P2 entries in L2 (where P1 + P2 = V)
- 3-level PT with 2^P1, 2^P2, 2^P3 entries in top, L2, L3 (where P1 + P2 + P3 = V)
- 4-level PT with ...
- ...

How many levels?
How to divide V-bits among P1, P2, ...

Rules of thumb:
- Usually want each node to fit on a page
  (Awkward to allocate large, contiguous regions of physical/kernel memory, so keep each node less than 1 page)
- Often have same node sizes
  (Simpler to allocate a bunch of things of same size; simpler to define data structures; etc.)
  If not *all* the same size, often top level is small, and remaining levels are the same.

Example: x86 32-bit 2-level PT. Page size is 2^12 bytes so need V = 20 bits to index 2^20 virtual pages.
They use a 2-level page table, dividing the 20 vpage index bits into 10 and 10. Each node then has 2^10 entries. Each entry is 2^2 bytes, so each node is 2^12 bytes = 1 page,
--> Meet both "fit on page" and "evenly divide" rules of thumb.

Example: Sun solaris 32-bit (above)
Example: 64-bit x86 (above)

****

Summary - 1 min

****

Goals of virtual memory:

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