Lecture #12: Address translation

****

Review -- 1 min

****

****

Outline - 1 min

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
Use 2: paging to disk

****

Lecture - 35 min

****

1. Virtual memory abstraction

Reality v.	abstraction
Physical memory	Virtual memory
No protection address space	each program isolated
Limited size	infinite memory
sharing of physical frames	everyone thinks they are loaded at addr "0"
Easy to share data between	ability to share code, data programs

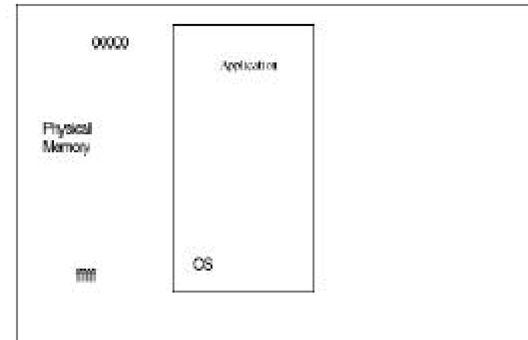
That all sounds great, but how do I do it?

2. Historical perspective: Operating system organizations

2.1 Uniprogramming w/o protection

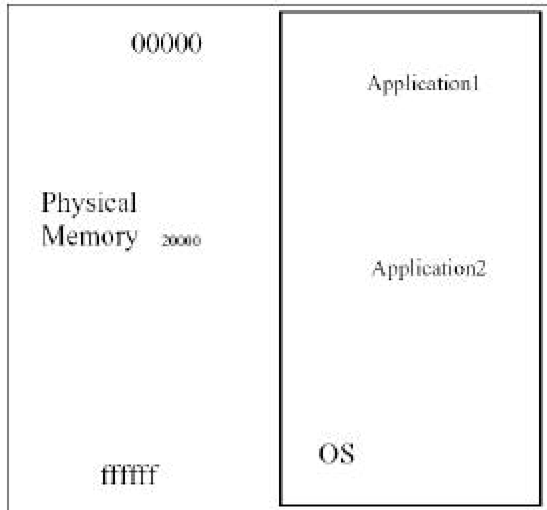
Personal computer OS's
Application always runs at the same place in physical memory
Each application runs one at a time
->give illusion of dedicated machine by giving reality of dedicated machine

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.

2.3 Multiprogrammed OS with protection

Goal of protection:

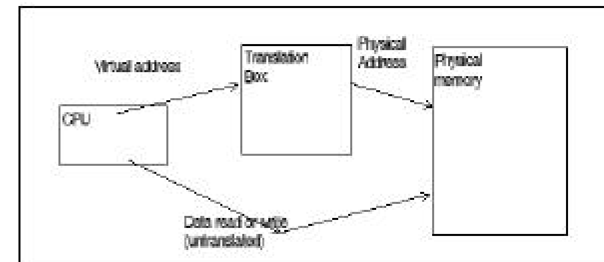
- keep user program from crashing OS

- 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



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
- 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)

5. Implementing protection, relocation

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

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

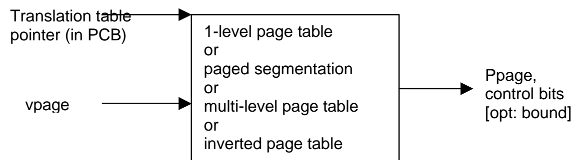
Basics:
segment maps variable-sized range of contiguous virtual addresses to a range of contiguous physical addresses

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

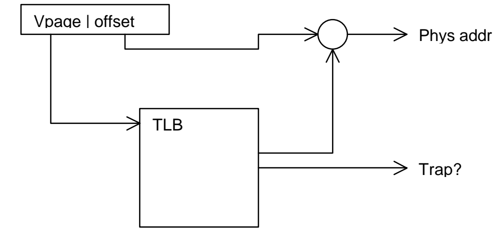
need data structures to lookup page/segment mapping given a virtual address
<segment #> → segment info {base, size}
<page #> → page info {base}

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
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.

Various kinds of translation schemes

-- start with simplest!

****

Admin - 3 min

Project 1 due W before class
- late policy

Advice for software engineering

****

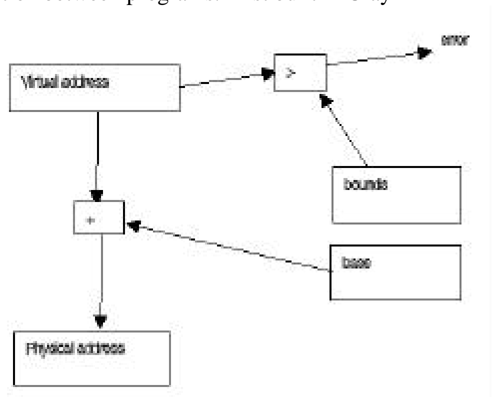
Lecture - 35 min

****

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

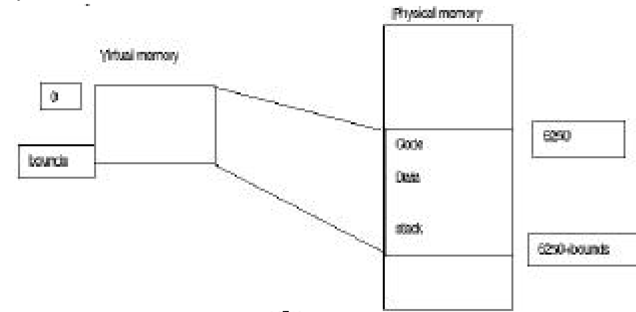
6. Base and bounds

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.

Simple translation hardware:

With base and bounds, what gets saved/restored on a context switch?

Only OS can change base and bounds.
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?

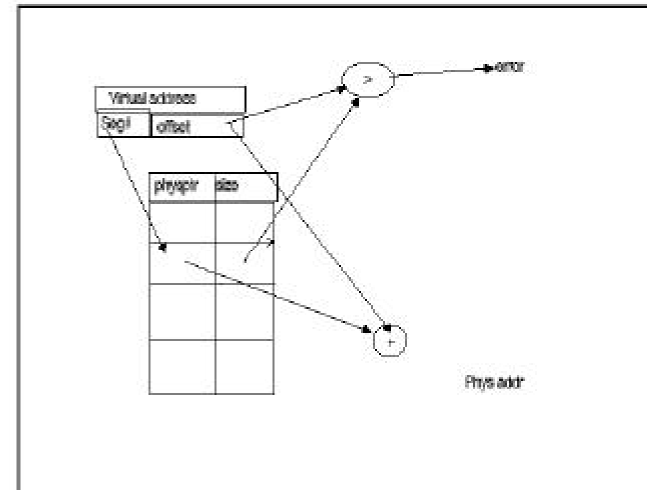
Evaluation
Base and bounds pros:
+ simple, fast
Cons:
1. Hard to share between programs
For example, suppose 2 copies of "vi"
Want to share code
Want data and stack to be different.
Cant do this with base and bounds.

2. 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

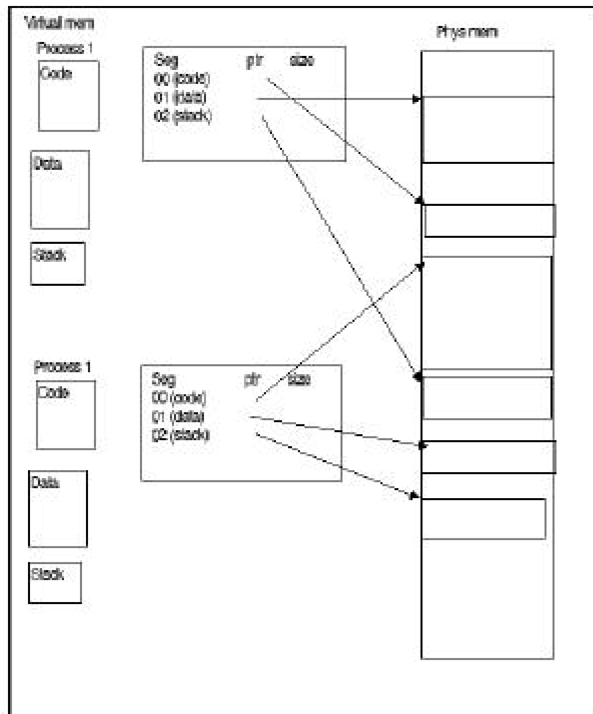
3. Doesn't allow heap, stack to grow dynamically – want to put these as far apart as possible in virtual memory so they can grow to whatever size is needed.

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.)

Simple TLB:
What must be saved/restored on context switch?
Typically, segment table stored in CPU not in memory because it's small.

QUESTION: How does protection work?, Sharing?

Segmentation pros and cons:
+ efficient for sparse addr spaces
+ 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?

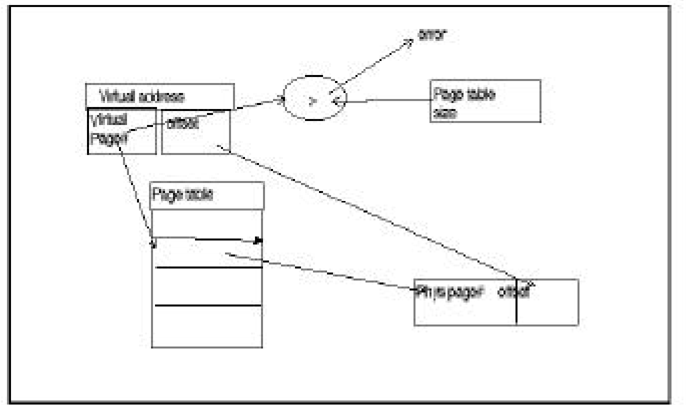
8. Paging

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.

Logical view:



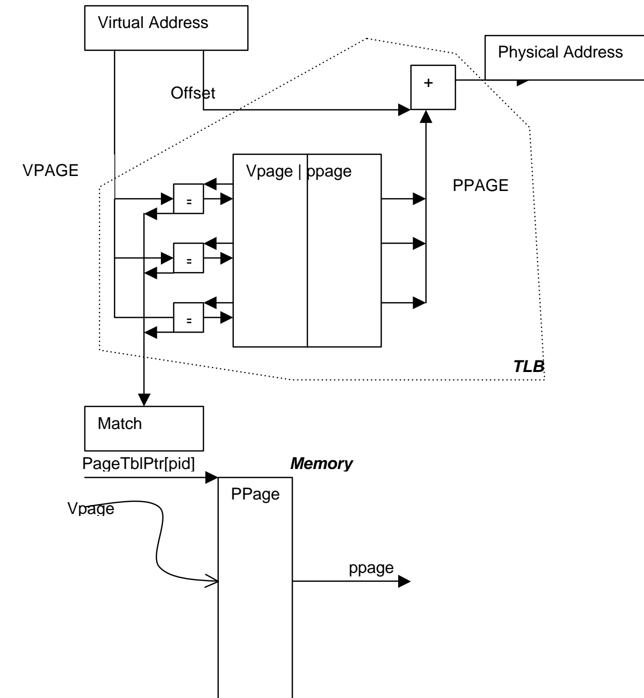
each address space has its own page table stored in physical memory
→need pageTablePtr

pageTablePtr is physical address, not virtual address

DA: More complex TLB

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



1) each address space has its own page table stored in 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 (most current OS's)
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

Other option: Have HW read page tables/segment tables directly

- 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?

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
*= # 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. should also include control bits (valid, read-only, …)
b. 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: what if address space is sparse? For example UNIX – code starts at 0, stack starts at 2^31 - 1
1KB pages means 2million page table entries!
~8MB overhead per process
worse: this 8MB must be contiguous in physical memory!
Solution: (next section)

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

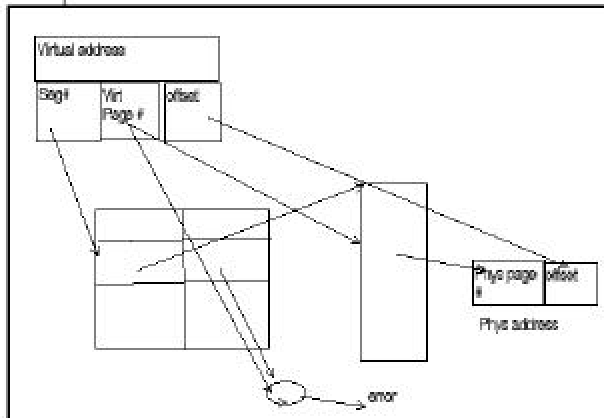
Is there a solution that allows simple memory allocation, easy to share memory, **and** efficient for sparse addr spaces?
How about combining segments and paging?

9. Multi-level translation

Use a tree of tables
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

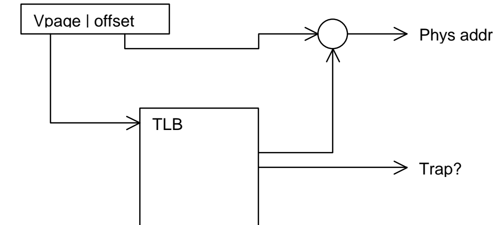
Question: what must be saved/restored on context switch?
Question: How do we share memory?
(Can share entire segment or single page)

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

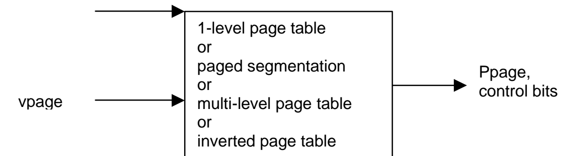
No – from TLB point of view,
address = <virtual page number, offset>
The virtual page number happens (in this case) to be organized as "seg, vpage" for when we look in memory, but TLB doesn't care

→ flexible software translation

Hardware is always:



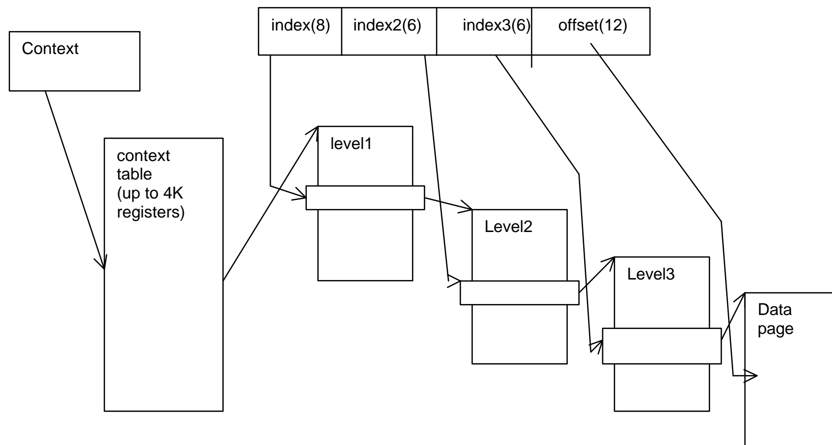
Memory data structure is opaque object:



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
+ 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)
- page tables need to be physically contigous
    128 MB executable w 4KB pages
    32K entries at least 4 bytes / entry
    128KB contigous
    needed
- two (or more) hops per memory reference
2 problems with segmented paging
1) need large contiguous memory →page page tables
2) multiple memory references to do lookup → inverted page table

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

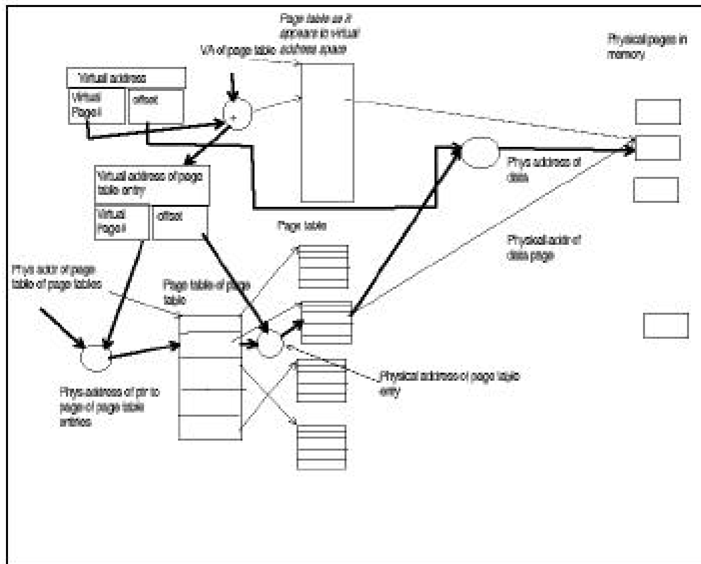
QUESTION: what needs to change on context switch?

Evaluation: multi-level page table
- good protection, sharing
- reasonable space overhead
- simple allocation
- DA: several memory reads needed per memory access (hope TLB solves)

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?

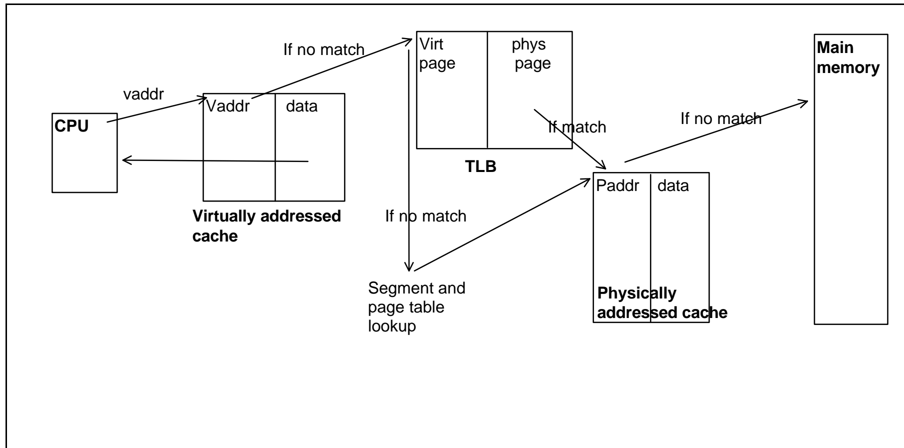
This is called an inverted page table for historical reasons

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

1. The big picture:



2. 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

****

Summary - 1 min

****

Goals of virtual memory:

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