

More Project Info

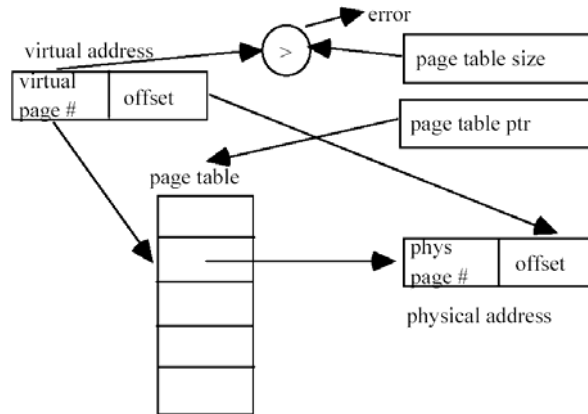
- **DESIGN DOC FOR PHASE II DUE TOMORROW.**
- Final Design docs for project 1 have been read and commented on, but still working out a consistent grading scheme.
- FAQs for Phase II:
  - LotteryScheduler should require minimal changes to existing code
  - System call is invoked when MIPS simulator does 'syscall' instruction
    - Causes instruction processing loop to throw 'MipsException'
    - When caught, calls MipsException.handle
      - handle() writes 'cause' into machine register
    - Causes 'Machine.processor().exceptionHandler' to be invoked
    - This is set by UserKernel constructor to be UserKernel.exceptionHandler
    - This gets current process (casts current KThread to UThread and extracts process field)
    - Calls process.handleException(cause)
  - Illegal operations do NOT include anything "illegal" done while calling a system call -> just return -1 from system call instead
  - write/read should return -1 if cannot read/write complete memory buffer provided by user, but number of bytes transferred if memory is OK but filesystem is not
  - Use Lib.random() to get an int for LotteryScheduler: Will be deterministic
  - Don't introduce overflow bugs in your code, but OK to assume that an 'int' is good enough to do the lottery with.

Questions from Lecture?

Today's Menu

- Simple Paging Review
- Multilevel Paging
- Paging with Segmenting
- TLBs
- Quiz

Simple Paging Review



- Segmenting with fixed size segments (plus more)
- Divide up physical and virtual memory into pages OF THE SAME SIZE.
  - Note: physical and virtual addresses do NOT have to have the same number of address bits.
    - Why?
- Doing actual address translation:
  - Each virtual address is divided up into page number and offset.
    - This is based on page size.
  - The page number is used to lookup into the page table.
  - The physical address of page is retrieved from the page table entry.
  - Physical address of page + offset = physical memory location.
- Load from memory can now proceed with this address . . . or can it?
  - What if the physical memory is smaller than the virtual memory? (yeah, it will be!)
    - Not all pages in memory.
  - What if page is not in memory?
    - Page Fault! (exception!)
  - How do we know if a page is memory?
    - Need extra bits in page table. . . Valid Bit.
  - What happens on a page fault?
    - Load a page if memory free
    - If not, kick someone out! (more on this later)
- Pros of paging:
  - Solves allocation—we do everything in fixed block
  - Easy sharing, with Protection Bits
- Cons
  - Complex
  - Slow—have to look up every address!
    - Can fix with buffering! More on this later.
  - Page tables have to be saved somewhere!
  - Adds to context switches
  - Internal fragmentation

- Pages fixed size. . .if page size is X bytes, and we need X+1, how much memory is wasted?
- Problem with paging: how big are single-level page tables?

**QUESTION:** Say we have 32-bit address with 4Kb pages. How many page table entries are there?

**ANSWER:** 20 bits of page address, so  $2^{20}$  or  $1024 * 1024$  entries

- How big is a page table entry?
- Minimum size is enough to span physical memory.

**QUESTION:** With 512 Mbytes in your machine and still assuming 4Kb pages, how many bits is the physical page frame number in the page table entry?

**ANSWER:** 29 bits of physical address space (512 MBytes), divided into 4 KB pages -> 12 bits of offset, 17 bits of page address. So, need 17 bits to store physical page frame number

- Problem is that we don't want to hard-wire the amount of physical memory the machine might have!
- In general, on a 32-bit machine can have 32-bit physical addresses. So, with 4KB pages need 12 bits for offset and 20 bits for page address, just like virtual addresses

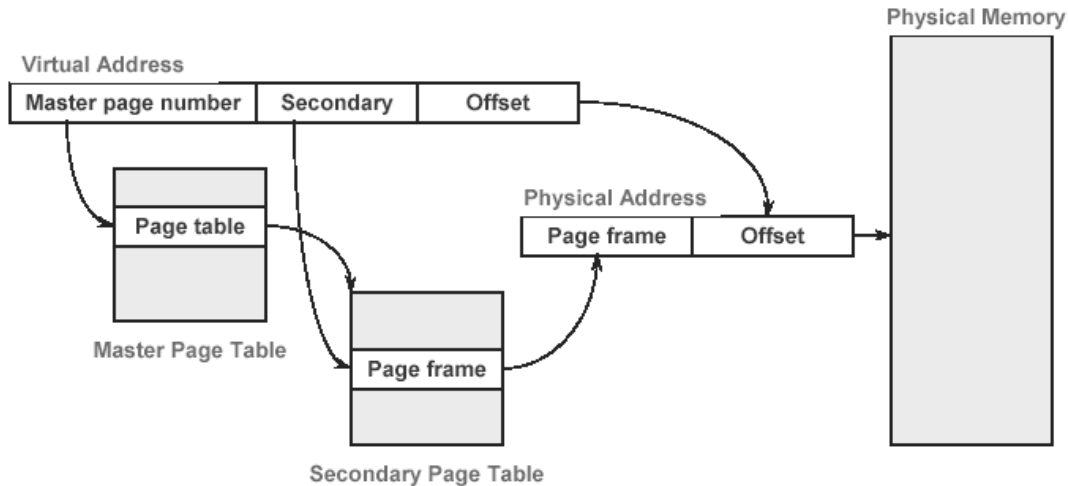
**QUESTION:** So how big is the page table then?

**ANSWER:**  $1024 * 1024 * 20$  bits = 2.5 MBytes, all in physical memory!! AND THAT'S FOR EACH PROCESS!

- Note also that each page table entry also contains some accounting and status bits, for example, indicating whether a given page is read-only, whether it has been modified, etc.
- So on a 32-bit machine a page table entry tends to be 32 bits wide, which means  $1024 * 1024 * 32 = 4$  MBytes.
- Solution: page the page tables!

### Multilevel Paging

- Solves the problem by using a hierarchy of page tables



- Key observation: since at any given time a process is only using a small portion of its address space, only need page table entries corresponding to what's actually in use
  - Note that later on we will talk about DEMAND PAGING: Moving memory to and from disk
  - This is going to allow us to allocate more virtual memory than we actually have physical RAM
  - With multilevel page tables, the PAGE TABLES can be PAGED—meaning that if the process has a portion of its address space which hasn't been used in a long time, those page tables (and the associated pages) go out to disk
    - This is project phase 3!
- Multilevel paging on the x86
  - Uses two-level page tables
  - Virtual address is divided into 3 components:

10 bits (31 .. 22): Page directory index  
 10 bits (21 .. 12): Page table index  
 12 bits (11 .. 0): Offset

- Example: Virtual address

0x13a49F01 =	0001001110	1001001001	11110000001
	0x04e	0x249	0xF01
	pgdirind.	pgtblind.	offset

- First, use page dir index as index into "page directory."
  - Physical base address of the page directory is stored in a special register (called "CR3") on the x86. It is always page-aligned, so the base address might be 0x0001c000.

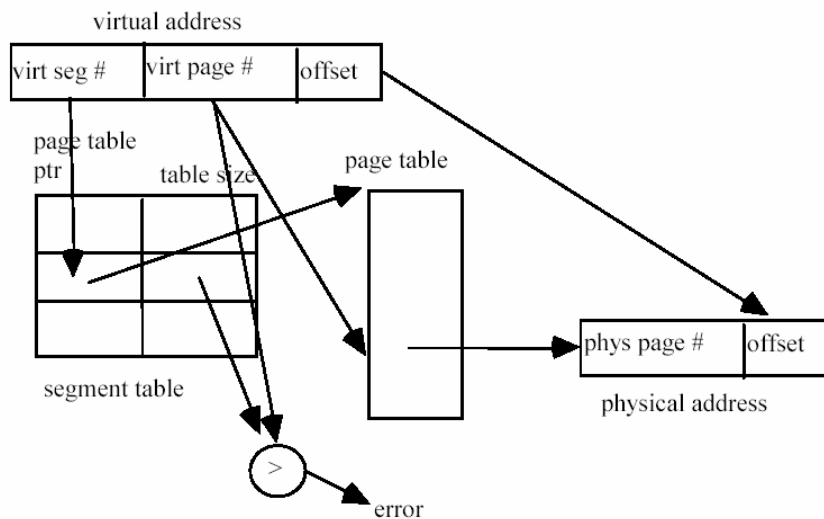
**QUESTION:** How many entries in the page directory if each entry is 4 bytes (why 4 bytes)?

**ANSWER:** Well 10 bits of index, so  $2^{10}$  entries \* 4 bytes each entry = 4Kb (One page!)

- (use picture on board) At offset 0x04e in the page directory we find the physical address of page table, e.g. 0x000a2000. (Note that the page table is also page-aligned and holds just one page.)
  - We use the "page table index" as an index into this second-level table.
  - At index 0x249 into the page table at 0xa2000 we find the physical page number of the address we are looking for, e.g., 0x0341b000.
  - We add to this the offset (0xf01) to get the final physical address: 0x0341bf01.
- Generally the page directory entries contain PHYSICAL addresses for page tables.
  - However, if page not in memory, then the entry would contain an OS-specific pointer (such as a disk block address) used to record where the page table is stored on disk.
  - In this way, page directory entries always use PHYSICAL addresses, but the page tables can still be swapped out.
- Note that the page directory is always resident in physical memory, meaning that each process has 1 page always "pinned"

### Paging and Segmentation

- Can use the same logic in multilevel paging to combine multilevel segmentation.



- Now, instead of dividing address up into two levels of page tables, instead divide up into segments
  - Page each segment.
- Basically just another case of what we just went through: see lecture notes for more details.

## Caching and TLBs

**QUESTION:** Multilevel paging makes storage efficient, but what's the drawback? What's the solution?

**ANSWER:** Waaay too many memory accesses to do this for every instruction. Things would crawl! Solution: why not cache translations to make this fast? Use TLB, Translation Lookaside Buffer: caching applied to address translation.

- Types of cache as applied to TLB:
  - Direct Mapped - restrict each virtual page to use specific TLB entry
    - Kinda like hashing—1 to 1 mapping.
    - Fast and simple, but have to deal with collisions.
  - Set Associative - partition TLB into N separate banks. Select entry with low order virtual page number bits. Compare all N entries in parallel.
    - Solves collisions partially—each entry has a specific bank it can be in.
    - However, more complex, still have collisions.
    - Slow unless done in parallel.
  - Fully Associative - Compare entire TLB in parallel. Only one entry per bank.
    - Entry can be anywhere in cache.
      - No collisions—easy to place in cache.
    - However, must search in parallel to make fast enough
      - HARD to implement in hardware.
- Effective access time in a TLB:
  - $P(\text{hit}) * \text{cost of hit} + P(\text{miss}) * \text{cost of miss}$

**QUESTION:** Given the following TLB characteristics:

- 20 ns ==> TLB access
- 100 ns ==> Memory access
- $P(\text{hit}) = .80$

What would the effective access time be?

**ANSWER:**  $EAT = .80(120) + .20(220) = 140 \text{ ns}$

- Is this good?
  - Hierarchy of access times for all things memory:

	Latency	Size	Cost
Registers	2.5ns	32-128 bytes	on chip
On-chip cache	5ns	32KB	on chip
Off-chip cache	20ns	512KB	\$2000/MB
Main memory	40ns	512MB	\$4/MB
Disk	10ms (10M ns)	100 GB	\$0.001MB
Robotic tape	10s (10B ns)	100 TB	Factor of 3-5 less than disk.

## Quiz