COMP 530: Operating Systems

Virtual Memory: Paging

Don Porter

Portions courtesy Emmett Witchel and Kevin Jeffay

COMP 530: Operating Systems

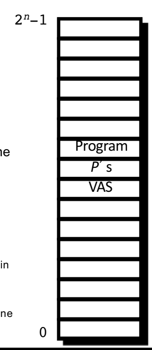
Review

- Program addresses are virtual addresses.
  - Relative offset of program regions can not change during program execution. E.g., heap can not move further from code.
  - (Virtual address == physical address) is inconvenient.
    - Program location is compiled into the program.
- Segmentation:
  - Simple: two registers (base, offset) sufficient
  - Limited: Virtual address space must be <= physical
  - Push complexity to space management:
    - Must allocate physically contiguous region for segments
    - Must deal with external fragmentation
    - Swapping only at segment granularity
- Key idea for today: Fixed size units (pages) for translation
    - More complex mapping structure
    - Less complex space management

COMP 530: Operating Systems

Virtual Memory

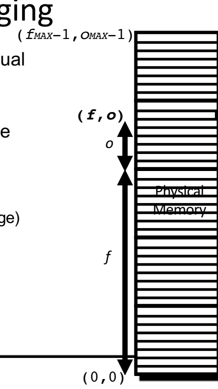
- Key problem: How can one support programs that require more memory than is physically available?
  - How can we support programs that do not use all of their memory at once?
- Hide physical size of memory from users
  - Memory is a "large" *virtual address space* of 2^n bytes
  - Only portions of VAS are in physical memory at any one time (increase memory utilization).
- Issues
  - Placement strategies
    - Where to place programs in physical memory
  - Replacement strategies
    - What to do when there exist more processes than can fit in memory
  - Load control strategies
    - Determining how many processes can be in memory at one time



COMP 530: Operating Systems

Solution: Paging

- Physical memory partitioned into equal sized *page frames*
  - Example page size: 4KB
- Memory only allocated in page frame sized increments
  - No external fragmentation
  - Can have internal fragmentation (rounding up smaller allocations to 1 page)
- Can map any page-aligned virtual address to a physical page frame

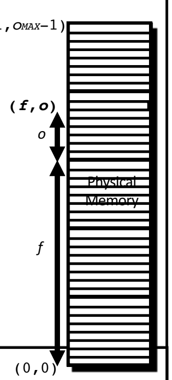


COMP 530: Operating Systems

Page Mapping

Abstraction: 1:1 mapping of page-aligned virtual addresses to physical frames

- Imagine a ***big ole' table*** **(BOT):**
  - The size of memory / the size of a page frame
- Address translation is a 2-step process
  1. Map virtual page onto physical frame (using BOT)
  2. Add offset within the page



COMP 530: Operating Systems

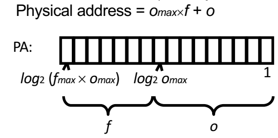
Physical Address Decomposition

A physical address can be split into a pair (f, o)

f — frame number (f_{max} frames)

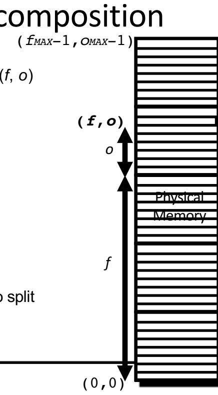
o — frame offset (o_{max} bytes/frames)

Physical address = $o_{max} \times f + o$



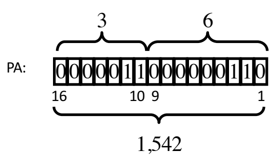
As long as a frame size is a power of 2, easy to split address using bitwise shift operations

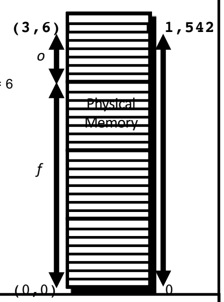
- Prepare for lots of power-of-2 arithmetic…



Physical Addressing Example

- Suppose a 16-bit address space with (o_{max} =) 512 byte page frames
  - Reminder: 512 == 2^9
  - Address 1,542 can be translated to:
    - Frame: 1,542 / 512 == 1,542 >> 9 = 3
    - Offset: 1,542 % 512 == 1,542 & (512-1) == 6
  - More simply: (3,6)

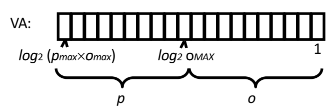


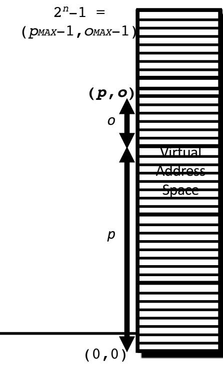


Virtual Page Addresses

- A process's virtual address space is partitioned into equal sized *pages*
  - |*page*| = |*page frame*|

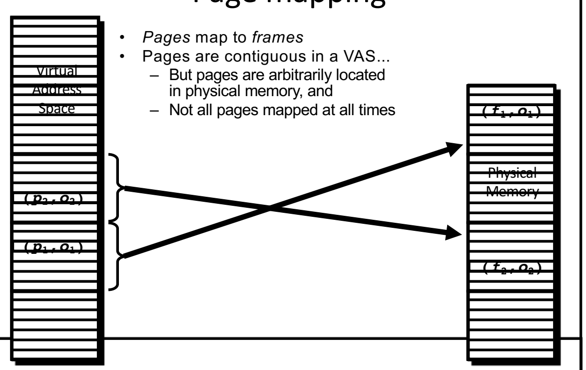
A virtual address is a pair (p, o)
- p — page number (p_{max} pages)
- o — page offset (o_{max} bytes/pages)
- Virtual address = $o_{max} \times p + o$





Page mapping

- *Pages* map to *frames*
- Pages are contiguous in a VAS...
  - But pages are arbitrarily located in physical memory, and
  - Not all pages mapped at all times

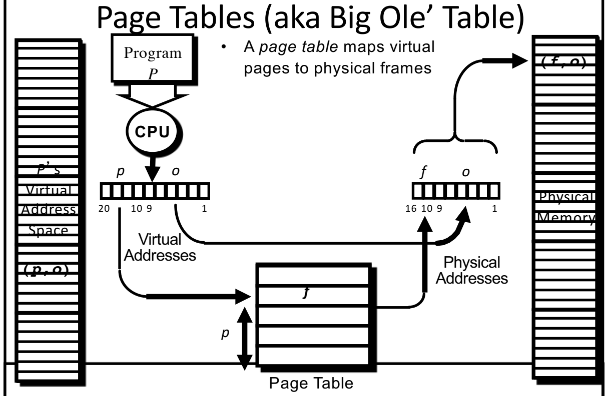


Questions

- The offset is the same in a virtual address and a physical address.
  - A. True
  - B. False

Page Tables (aka Big Ole' Table)

- A *page table* maps virtual pages to physical frames

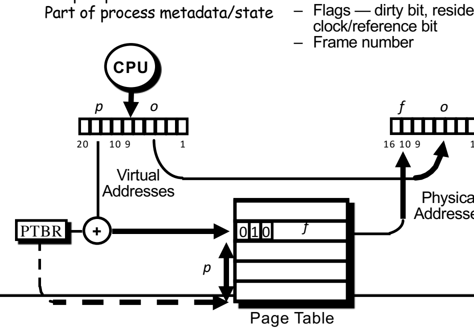


Page Table Details

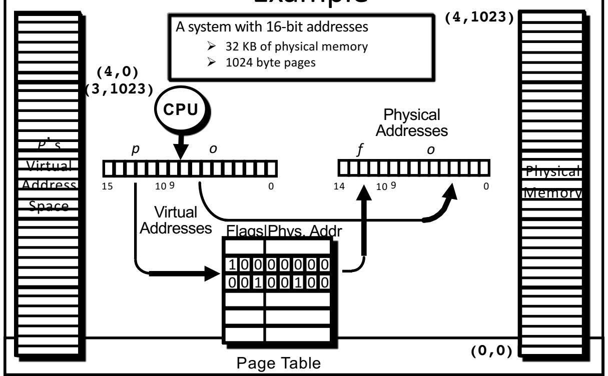
1 table per process

Part of process metadata/state

- Contents:
  - Flags — dirty bit, resident bit, clock/reference bit
  - Frame number



Example

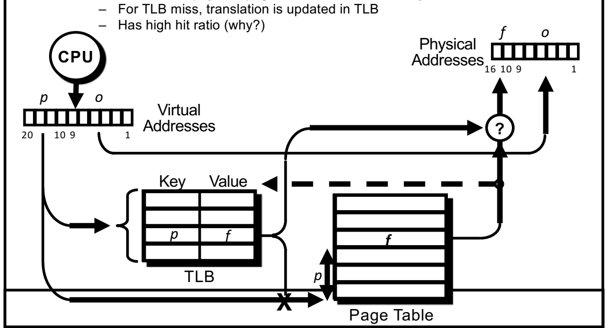


Performance Issues with Paging

- Problem — VM reference requires 2 memory references!
  - One access to get the page table entry
  - One access to get the data
- Page table can be very large; a part of the page table can be on disk.
  - For a machine with 64-bit addresses and 1024 byte pages, what is the size of a page table?
- What to do?
  - Most computing problems are solved by some form of…
    - Caching
    - Indirection

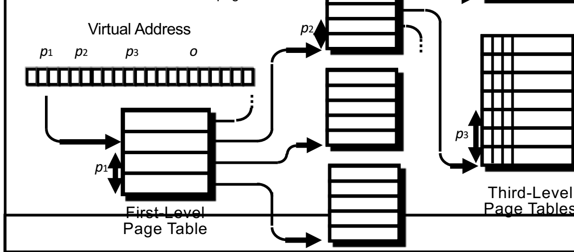
Using a TLB to Cache Translations

- Cache recently accessed page-to-frame translations in a TLB
  - For TLB hit, physical page number obtained in 1 cycle
  - For TLB miss, translation is updated in TLB
  - Has high hit ratio (why?)



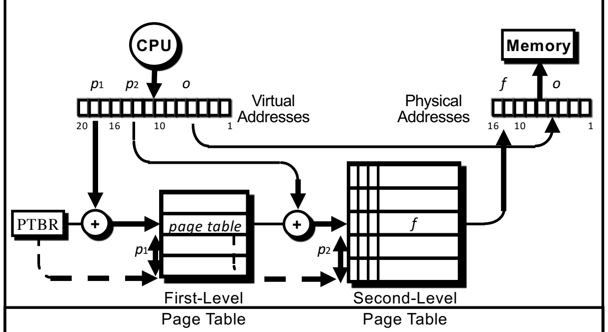
Dealing with Large Tables

- Add additional levels of indirection to the page table by sub-dividing page number into *k* parts
  - Create a "tree" of page tables
  - TLB still used, just not shown
  - The architecture determines the number of levels of page table



Dealing with Large Tables

- Example: Two-level paging



Large Virtual Address Spaces

- With large address spaces (64-bits) forward mapped page tables become cumbersome.
  - E.g. 5 levels of tables.
- Instead of making tables proportional to size of virtual address space, make them proportional to the size of physical address space.
  - Virtual address space is growing faster than physical.
- Use one entry for each physical page with a hash table
  - Translation table occupies a very small fraction of physical memory
  - Size of translation table is independent of VM size
- Page table has 1 entry per virtual page
- Hashed/Inverted page table has 1 entry per physical frame

Frames and pages

- Only mapping virtual pages that are in use does what?
  - A. Increases memory utilization.
  - B. Increases performance for user applications.
  - C. Allows an OS to run more programs concurrently.
  - D. Gives the OS freedom to move virtual pages in the virtual address space.
- Address translation and changing address mappings are
  - A. Frequent and frequent
  - B. Frequent and infrequent
  - C. Infrequent and frequent
  - D. Infrequent and infrequent

Hashed/Inverted Page Tables

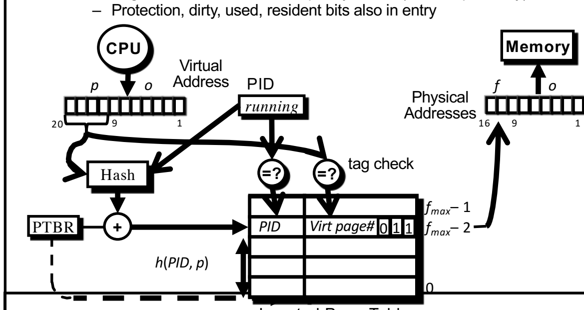
- Each frame is associated with a register containing
  - Residence bit: whether or not the frame is occupied
  - Occupier: page number of the page occupying frame
  - Protection bits
- Page registers: an example
  - Physical memory size: 16 MB
  - Page size: 4096 bytes
  - Number of frames: 4096
  - Space used for page registers (assuming 8 bytes/register): 32 Kbytes
  - Percentage overhead introduced by page registers: 0.2%
  - Size of virtual memory: irrelevant

Inverted Page Table Lookup

- CPU generates virtual addresses, where is the physical page?
  - Hash the virtual address
  - Must deal with conflicts
- TLB caches recent translations, so page lookup can take several steps
  - Hash the address
  - Check the tag of the entry
  - Possibly rehash/traverse list of conflicting entries
- TLB is limited in size
  - Difficult to make large and accessible in a single cycle.
  - They consume a lot of power (27% of on-chip for StrongARM)

Inverted Page Table Lookup

- Hash page numbers to find corresponding frame number
  - Page frame number is not explicitly stored (1 frame per entry)
  - Protection, dirty, used, resident bits also in entry



Searching Inverted Page Tables

- Page registers are placed in an array
- Page *i* is placed in slot *f(i) where f* is an agreed-upon hash function
- To lookup page *i*, perform the following*:*
  - Compute *f(i)* and use it as an index into the table of page registers
  - Extract the corresponding page register
  - Check if the register tag contains *i,* if so, we have a hit
  - Otherwise, we have a miss

Searching Inverted Page Tables

- Minor complication
  - Since the number of pages is usually larger than the number of slots in a hash table, two or more items *may* hash to the same location
- Two different entries that map to same location are said to collide
- Many standard techniques for dealing with collisions
  - Use a linked list of items that hash to a particular table entry
  - Rehash index until the key is found or an empty table entry is reached (open hashing)

Observation

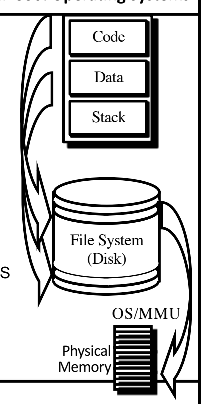
- One cool feature of inverted page tables is that you only need one for the entire OS
  - Recall: each entry stores PID and virtual address
  - Multiple processes can share one inverted table
- Forward mapped tables have one table per process

Questions

- Why use hashed/inverted page tables?
  - A. Forward mapped page tables are too slow.
  - B. Forward mapped page tables don't scale to larger virtual address spaces.
  - C. Inverted pages tables have a simpler lookup algorithm, so the hardware that implements them is simpler.
  - D. Inverted page tables allow a virtual page to be anywhere in physical memory.

Swapping

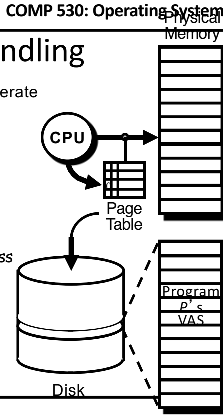
- A process's VAS is its context
  - Contains its code, data, and stack
- Code pages are stored in a user's file on disk
  - Some are currently residing in memory; most are not
- Data and stack pages are also stored in a file
  - Although this file is typically not visible to users
  - File only exists while a program is executing
- OS determines which portions of a process's VAS are mapped in memory at any one time



Page Fault Handling

- References to non-mapped pages generate a *page fault*
  - *Remember Interrupts?*

Page fault handling steps:
- Processor runs the interrupt handler
- OS blocks the running process
- OS starts read of the unmapped page
- OS resumes/initiates some other process
- Read of page completes
- OS maps the missing page into memory
- OS restart the faulting process



Performance Analysis

- To understand the overhead of paging, compute the *effective memory access time* (*EAT*)
  - *EAT* = *memory access time* × *probability of a page hit* + *page fault service time* × *probability of a page fault*
- Example:
  - Memory access time: 60 *ns*
  - Disk access time: 25 *ms*
  - Let p = the probability of a page fault
  - $EAT = 60(1-p) + 25{,}000{,}000\boldsymbol{p}$
- To realize an *EAT* within 5% of minimum, what is the largest value of p we can tolerate?

Segmentation vs. Paging

- Segmentation has what advantages over paging?
  - A. Fine-grained protection.
  - B. Easier to manage transfer of segments to/from the disk.
  - C. Requires less hardware support
  - D. No external fragmentation
- Paging has what advantages over segmentation?
  - A. Fine-grained protection.
  - B. Easier to manage transfer of pages to/from the disk.
  - C. Requires less hardware support.
  - D. No external fragmentation.

Meta-Commentary

- Paging is really efficient when memory is relatively scarce
  - But comes with higher latency, higher management costs in hardware and software
- But DRAM is getting more abundant!
  - Push for larger page granularity (fewer levels of page tables)
  - Or just go back to segmentation??
    - If everything fits into memory with space to spare, why not?

Summary

- Physical and virtual memory partitioned into equal size units
- Size of VAS unrelated to size of physical memory
- Virtual *pages* are mapped to physical *frames*
- Simple placement strategy
- There is no external fragmentation
- Key to good performance is minimizing page faults