Previously discussed

- Segmentation - the process address space is divided into logical pieces called segments. The following are the example of types of segments
  - code
  - bss (statically allocated data)
  - heap
  - stack
- process may have several segments of the same type!
- segments are used for different purposes, some of them may grow, OS can distinguish types of segments and treat them differently, example:
  - allowing code segments to be shared between processes
  - prohibiting writing into code segments

Lecture 14: paging and virtual memory

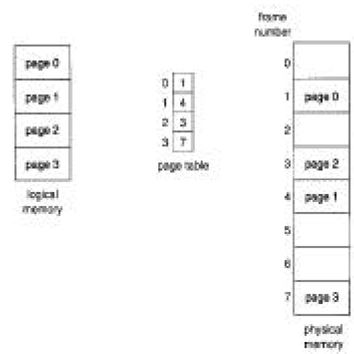
- paging
  - definition of paging
  - implementation
  - sharing
  - protection
  - speeding up - translation look-aside buffers
- virtual memory
  - definition
  - page replacement strategies
    - FIFO
    - optimal
    - Least Recently Used
  - Implementing LRU
    - recently used bit
    - second chance algorithm
  - Frame allocation
  - Thrashing

What is paging

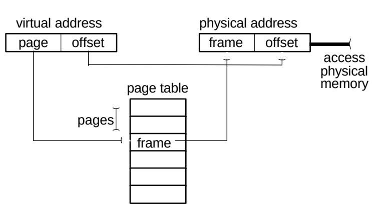
- Each process is divided into a number of small, fixed-size partitions called *pages*
  - Physical memory is divided into a large number of small, fixed-size partitions called *frames*
  - Pages have nothing to do with segments
  - Page size = frame size
    - Usually 512 bytes to 16K bytes
  - The whole process is still loaded into memory, but the pages of a process do ***not*** have to be loaded into a contiguous set of frames
  - Virtual address consists of page number and offset from beginning of that page
- Compared to segmentation, paging:
  - Makes allocation and swapping easier
  - No external fragmentation
  - but there may be internal fragmentation. Why?

Page table

- A *page table* keeps track of every page in a particular process
- each entry contains the corresponding frame in main (physical) memory
- note that there is a separate page table for every process
- *address space* - the set of all addresses
- user view of address space - continuous
- physical address space - fragmented



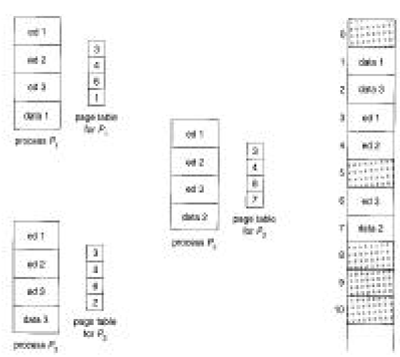
Address lookup in paging



- *Virtual address* (or *logical address*) - used by user program consists of two parts - page number and *offset* (or displacement) within the page.
- When the address needs to be accessed page number is translated into frame number in page table.
- *Physical address* contains frame number and frame offset

Sharing pages

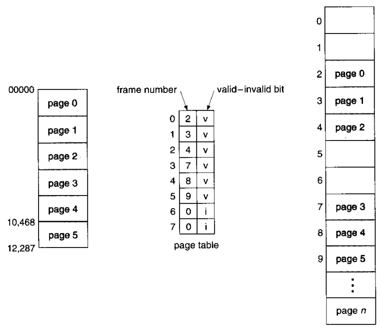
- Paging provides easy way of reusing the code
- if multiple processes are executing the same code (say, text editor) the pages containing the code for the editor can be shared



- the code has to be *reentrant* (not self-modifying) and write protected - usually every page has a read-only bit
- how can shared memory be implemented using the same technique?

Protection in page tables

- The process may not use the whole page table
- unused portions of the page table are protected by *valid/invalid bit*
- the address space for the process is 0 through 12,287 (6 - 2K pages)
- even though the page table contains additional unused page references they are marked as invalid
- attempt to address these pages will result in a trap with "memory protection violation"
- similarly read/write protection can be organized

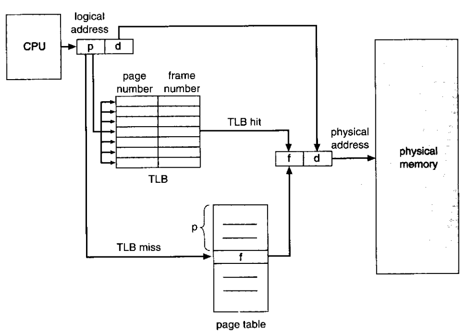


TLBs

- A modern microprocessor has, say, a 32 bit virtual address space (2^{32} = 4 GB)
- If page size is 1k (2^{10}), that means all the page tables combined could have over 2^{22} (approximately 4 million) page entries, each at least a couple of bytes long
- Problem: if the main memory is only, say, 16 Mbytes, storing these page table there presents a problem!
  - Solution: store page tables in virtual memory (discussed later), bring in pieces as necessary
- New problem: memory access time may be doubled since the page tables are now subject to paging
  - (one access to get info from page table, plus one access to get data from memory)
  - New solution: use a special cache (called a Translation Lookaside Buffer (TLB)) to cache page table entries
- a TLB consists of two parts key (page number) and value (frame number)
- page lookup in all TLBs can be done in one step

8

Using TLBs



- Page # is first looked in TLBs if found (cache hit) we can go straight to the frame
- if not found (cache miss) we have to look up the frame in the page table
- program with good code *locality of reference* benefits from TLBs
- TLBs have to be *flushed* with each context switch - new page table is loaded

9

Virtual memory

- *Virtual memory* is the technique that allows to execute processes that may not be completely in physical memory
- can be implemented by:
  - demand paging (only the necessary pages are brought in)
  - segmentation (only the segments that are currently in use are brought in)
- demand paging
  - while not in use the pages are stored on a disk - *backing store*
  - the page table indicates whether the page is in memory or in backing store
  - if a process requests a page that is not in memory
    - a *page fault* trap is generated and control is passed to OS
    - the faulted process is suspended (another process may be started while it waits) and a request to fetch the page is generated
    - when page is in memory the page table is updated and the instruction that caused page fault is re-executed
- virtual memory is *transparent* to user processes

10

Page faults in detail

- an attempts to access a page that's not in physical memory causes a *page fault*
  - page table must include a *present* bit (sometimes called *valid* bit) for each page
  - an attempt to access a page without the present bit set results in a *page fault*, an *exception* which causes a *trap* to the OS
  - when a page fault occurs:
    - OS must *page in* the page — bring it from disk into a free frame in physical memory
    - OS must update page table & present bit
    - faulting process continues execution
- unlike interrupts, a page fault can occur any time there's a memory reference
  - even in the middle of an instruction!
(how? and why not with interrupts??)
  - however, handling the page fault must be invisible to the process that caused it

11

Handling Page Faults

- the page fault handler must be able to recover enough of the machine state (at the time of the fault) to continue executing the program
- the PC is usually incremented at the beginning of the instruction cycle
  - if OS / hardware doesn't do anything special, faulting process will execute the next instruction (skipping faulting one)
- with hardware support:
  - test for faults before executing instruction (IBM 370)
  - instruction completion — continue where you left off (Intel 386…)
  - restart instruction, undoing (if necessary) whatever the instruction has already done (PDP-11, MIPS R3000, DEC Alpha, most modern architectures)

12

Starting a new process

- processes are started with 0 or more of their virtual pages in physical memory, and the rest on the disk
- *page selection* — ***when*** are new pages brought into physical memory?
  - prepaging — pre-load enough to get started: code, static data, one stack page (DEC ULTRIX)
  - demand paging — start with 0 pages, load each page on demand (when a page fault occurs) (most common approach)
    - disadvantage: many (slow) page faults when program starts running
- demand paging works due to the principle of *locality of reference*
  - Knuth estimated that 90% of a program's time is spent in 10% of the code

Page replacement

- To improve I/O utilization the OS *over-allocates* the main memory - if sum of address space of all executing processes is greater than the physical memory
- What happens if a process requests a page and there are no free frames?
- to (partially) remedy the situation a *clean/dirty bit* is associated with every in-memory page
  - if the page has been modified - it's dirty and has to be written to disk
  - if the page has not been modified - (it is the same as it's copy in the backing store) - it can be just discarded and replaced
- What if we still need to select a page to replace? The OS has to *evict* (remove) a page from memory to backing store
- Replacement strategies
  - FIFO - simplest to implement, performance is not always good
  - Optimal - replace the page that will not be used for the longest period of time - cannot be implemented (requires future knowledge)
  - LRU - replace the least recently used page

Performance of demand paging

- effective access time for demand-paged memory can be computed as:

  eacc = (1–p)(macc) + (p)(pfault)

  where:

  p = probability that page fault will occur

  macc = memory access time,

  pfault = time needed to service page fault
- with typical numbers:

  eacc = (1–p)(100) + (p)(25,000,000) = 100 + (p)(24,999,900)
  - If p is 1 in 1000,
    eacc = 25,099.9 ns (250 times slower!)
  - To keep overhead under 10%,
    110 > 100 + (p)(24,999,900)
    - p must be less than 0.0000004
    - less than 1 in 2,500,000 memory accesses must page fault!

Page replacement strategies

FIFO	A	B	C	A	B	D	A	D	B	C	B
frame 1											
frame 2											
frame 3											

Optimal	A	B	C	A	B	D	A	D	B	C	B
frame 1											
frame 2											
frame 3											

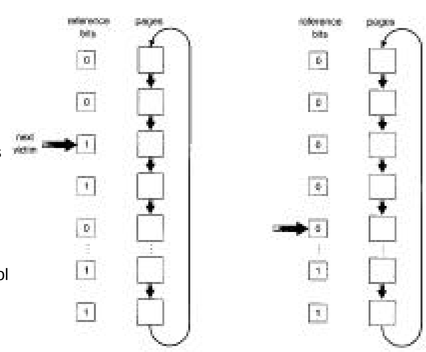
LRU	A	B	C	A	B	D	A	D	B	C	B
frame 1											
frame 2											
frame 3											

- Assumptions: 4 pages, 3 frames
- Page references: ABCABDADBCA

Implementing LRU

- A perfect implementation would be something like this:
  - Associate a clock register with every page in physical memory
  - Update the clock value at every access
  - During replacement, scan through all the pages and find the one with the lowest value in its clock register
  - What's wrong with all this implementation?
- Simple approximations:
  - Not-recently-used (NRU)
    - Use an R (reference) bit, and set it whenever a page is referenced
    - Clear the R bit periodically, such as every clock interrupt
    - Choose any page with a clear R bit to evict
    - there is overhead on clearing the bits and the picture may not be good enough - additional bits may be needed

Second-chance replacement (clock)



- Use an R (reference) bit as before
- On a page fault, circle around the "clock" of all pages in the user memory pool
- Start after the page examined last time
- If the R bit for the page is set, clear it
- If the R bit for the page is clear, replace that page and set the bit
- Can it loop forever? What does it mean if the "hand" is moving slowly? …if the hand is moving quickly

Frame Allocation

- How many frames does each process get (M frames, N processes)?
  - At least 2 frames (one for instruction, one for data), maybe more…
  - Maximum is number in physical memory
- Allocation algorithms:
  - Equal allocation - each gets M / N frames
  - Proportional allocation - number depends on size and priority
- Which pool of frames is used for replacement?
  - Local replacement - process can only reuse its own frames
    - predictable
  - Global replacement - process can reuse any frame (even if used by another process)
    - processes may be able to grow dynamically

Thrashing

- Consider what happens when memory gets overcommitted:
  - After each process runs, before it gets a chance to run again, all of its pages may get paged out
  - The next time that process runs, the OS will spend a ***lot*** of time page faulting, and bringing the pages back in
    - All the time it's spending on paging is time that it's not getting useful work done
    - With demand paging, we wanted a very large virtual memory that would be as fast as physical memory, but instead we're getting one that's as slow as the disk!
- This wasted activity due to frequent paging is called *thrashing*
  - Analogy — student taking too many courses, with too much work due

Working Sets

- Thrashing occurs when the sum of all processes' requirement is greater than physical memory
    - Solution — use local page frame replacement, don't let processes compete
      - Doesn't help, as an individual process can still thrash
    - Solution — only give a process the number of frames that it "needs"
      - Change number of frames allocated to each process over time
      - If total need is too high, pick a process and suspend it
- *Working set* (Denning, 1968) — the collection of pages that a process is working with, and which must be resident in main memory, to avoid thrashing
  - Always keep working set in memory
  - Other pages can be discarded as necessary
  - implementation - choose time T, pages that were accessed during time T constitute a working set, the rest can be discarded, scan periodically to update working set
    - Unix: T - about one second; scans - every several milliseconds