Virtual vs Physical Addresses

Physical addresses refer to hardware addresses of physical memory.

Virtual addresses refer to the *virtual store* viewed by the process.

- virtual addresses might be the same as physical addresses
- might be different, in which case virtual addresses must be *mapped* into physical addresses. Look at Fig 4-9. Mapping is done by Memory Management Unit (MMU).
- virtual space is limited by size of virtual addresses (not physical addresses)
- virtual space and physical memory space are independent

How can it work if a 32-bit virtual address allows 4GB of space to be addressible for a single process and multiple processes run on machines with less than this amount of physical memory?

1. processes don't use all of the 4GB of space.
2. only a portion of the address space that processes do use is loaded into physical memory at a time.

Virtual Memory with Paging

Paging is the most common memory management technique:

- virtual space of process divided into fixed-size *pages*
- virtual address composed of *page number* and *page offset*
- physical memory divided into fixed-size *frames*
- page in virtual space fits into frame in physical memory

Mapping of VA to PA is done with a page table. Fig 4-10, 4-11. *Each* process has an associated page table.

Page tables can be quite large. Thus have multiple levels of page tables. Many examples given in Tanenbaum. Just using additional bits of the virtual address and actually write out page tables to disk.

Present an overall picture of paging.

Draw:

- Disk
- Page table: Frame number, permissions, present/absent, modified (dirty), (add referenced bit later).
- Two processes, each with own page table.
- physical memory (1K frames).
- Frame table (supported by Operating System). Fields: status (free/used add, copied in or out later), pid, page, free frame index.
- Frame list. Queue of free frames.

Address Translation

Start execution, causing address translation to be used.

1. extract page number from virtual address (show page|offset). If we assume 1K page then 1024 bytes then offset is 10 bits.
2. if page number is greater than number of pages, generate an "illegal page" trap
3. access appropriate page table entry
4. if page is not in memory, trap "missing page"
5. if access violation, trap "protection violation" (segmentation violation in Unix)
6. finally, return physical address (frame number * page size + page offset)

Choosing a page size:

- overhead space — small page size means larger page tables
- internal fragmentation — large pages result in more internal fragmentation
- larger pages generally better for transferring to/from backing store
- typical sizes run from .5K to 4K bytes to 8K bytes

What happens on a page fault

Trap is handled by a *Page Fault Handler()*. This routine is responsible for swapping a page from disk into memory.

1. GetFreeFrame
2. Determine location of page. If on disk (such as text, initialized data or previously saved page) then get disk location:
   (a) Disk mapping table (pid, page, disk addr)
   (b) Routine *bsloc(pid, page)* returns disk addr.
3. Schedule reading in page from disk into physical memory (introduce copyin status here). Is the process ready for execution? Don't want to wait. This is a "major" fault because it requires disk access to satisfy.
4. Stack and uninitialized data page would not need to be initially read from disk, but would be zero-filled and used (why zero-fill?). A "minor fault."

What to do when the read is complete. Disk driver will generate an interrupt, what will the memory manager do in response?

1. mark frame entry as copied in.
2. make process needing the frame eligible for execution

What happens when there are no free frames?

Must remove one or more entries from physical memory. Which ones? Page replacement policy is used—talk about in a bit.

Given that a page has been selected for replacement what to do?

- mark the page as no longer present in page table
- if not modified then can simply put free frame back in the pool
- if modified then must schedule page for writing. Again we must consider what to do when the page is done being written.
- Modified page is written to "swap space" or a special "swap file" on disk—these include modified data, stack and heap pages.
- what to do if the frame that is being written is accessed again (would like to reclaim it—another type of "minor" fault)

Translation Lookaside Buffer

To speed translations, hardware maintains a cache called the *translation lookaside buffer (TLB)*. Part of the MMU. Look at Fig 4-14. Thus, virtual memory accesses:

1. check TLB for desired mapping; if present, we are done
2. if not present in TLB, consult mapping tables in (slower) memory
3. place virtual/physical address pair in cache

Note: there is an additional hidden cost to context switching. The cache must be flushed at every context switch! Once the cache has been filled, typical hit rates approach 95%.