11/09 Recitation Notes - Realities of Virtual Memory

brief re-motivation

```
sparse address space
contiguous virtual addresses not necessarily contiguous in memory
protection (OS from processes, processes from each other,
            red zones, per-process PTs)
bigger than physical address space
  useful when RAM was expensive, less useful now -- paging is *slow*
    why is it slow?  Every time we page the processor has to wait
      about 12ms == 6000 cycles on a 500MHz machine.
Cray supercomputers do not have VM:
  only one program runs at a time
  performance is critical -- paging is too slow
  MMU can slow down processor, so leave it out of the critical path
```

what does it mean?

```
Applications (and kernel, to a degree) deal in “virtual addresses”
  that correspond to “physical addresses”
somehow the references to VA's must be translated into PA's
```

how do we do VM?

```
in software:
  Mac has indirect pointers, allowing the OS to move memory around.
  protection is hard to do in software -- would have to insert checks
    around each load and store.
    -> slow
    -> insecure: a program can skip these checks
in hardware:
  complex, more expensive
  ...but *fast*, and *secure*
```

hardware support -- MMU

```
part of processor (used to be separate)
has enable switch
increases memory overhead -- what is now involved in a single load?
  must find correct address somehow (page table)
  must do translation
  can then do load (access cache, memory)
granularity
  one-to-one VA<->PA? not feasible
  need to choose page size
    too small - bad overhead
    too large - fragmentation (unit of protection, sharing, allocation)
page tables: big mapping of virtual pages to physical pages
address translation is slow -- how do we speed it up?
  low spatial locality -- pages are too big to decide that if you touch
    one you will touch its neighbours (in contrast to memory caches)
  high temporal locality -- we already know that memory accesses have
    locality because memory caches work
  so if we access an address, we will likely access another address on
    the same page soon -- so create a cache which holds its translation: TLB
TLB built-in
```

the page table

```
organized in levels (1 and up):
  single-level page tables are LARGE
     have to allocate entries for all of physical memory
  example:
       32 bit addr space, 12 bits page offset (4K page), 4B entries
       -> page table size = 2^20*4 = 4MB
       this was ridiculous when memory was $50/MB, isn't it more
         reasonable now?
       -> Each process has its own page table.
       -> Do you want every process to chew up 4MB of physical RAM?
       -> My desktop currently is running 77 processes...
  multilevel page tables:
    let you leave out parts of the table
    also lets you *page the page table*, meaning that only the first
      level has to be in physical RAM
  example:
    two-level table for 32-bit address space, 4K pages, 4B entries
    bit breakdown: 10/10/12 = index/index/offset
    10 index bits -> 1K entries
    4B entries -> 1 page/table
    location of top-level "directory" stored in special register
    each entry in top level points to second-level page table page
    second-level page has 1K entries; each maps a 4KB page
    each second-level page can map 4MB
    top-level page maps 4GB (32bits)
    must allocate top-level page, second-level for each top-level slot used
protection of lowest page for *(0) accesses
what does an entry look like?
  top-level has 20 bits to locate start of second-level page table page
  second-level has 20 page-frame bits
  each has 12bits for flags (read,write,valid,present,dirty,accessed,shared)
  bits defined by hardware
    MMU kicks processor for faults, violations based on flag bits
more levels are possible
complete page translations are the things that are cached by the TLB
```

how OS's use VM

```
startup:
  MMU disabled, deal with physical addresses
  construct page tables, map kernel virtual=physical
    have been referring to kernel code/data with PA's; need VA's to be same
  set register to point to top-level page table, enable MMU
  know # of first free physical page "highwatermark"
run-time management:
  allocation – increase highwatermark until memory full, maintain freelist
  replacement – when physmem full, choose page to replace (ie put on disk)
  swapping – write all pages of inactive processes to disk
  working set – set of all pages being actively used
  paging – some memory pages on disk; "page fault" when a page table entry
    is accessed that says "valid addr, but not present in memory";
    OS services page fault as necessary: read page from disk, or
       load program code page or init new page)
  demand image loading – load program-code pages only as needed
  lazy committing – on malloc, allocate but don't commit (shrink workingset)
  thrashing – working set is larger than physical memory; disk is busy!
```