Virtual memory is complicated, but not deep
- Again: Despite the obscure terminology, all virtual memory is trying to do is map ints to ints:
  - $VA : int \rightarrow PA : int$
- Most complexity from fact that CS has no good way to construct a general integer function.
  - Mapping between two *arbitrary* sets of integers, fundamentally requires a table of some sort.
  - Page table = a lookup table to manually build function
    - The usual variations: an array ("direct" page table), hash tables, weird trees of arrays.
    - The usual complications: speed, space.

Paging and segmentation not so different
- Main difference is the way they map ints.
  - Segmentation: represents $VA$'s as byte ranges ([va, nbytes]).
    - Restricts outputs so it can map them using a single table entry
    - Result: can protect & alloc variables sized units, but force mapped range to be contiguous, hard codes index for speed.
  - Paging: represents $VA$'s as pages, maps them using a tuple ([vpn, ppn]) for each page sized unit (forces base to be page aligned)
    - Result: can only protect and allocate fixed-size units, but can map any page to any other.
    - Hardware caches and lookups differ, but only because of different table layouts and logic, nothing fundamental...

Mapping functions: a perennial OS/CS theme
- OSes are constantly in the business of constructing a mapping function and then trying to make it fast
  - Example: File systems:
    - directory: map file name to inode
    - inodes: map file offset to disk block number
  - others: DNS names to IP addr, IP addr to eth, to routes,

Problem: large range = large page tables
- Same problem as memory: Don’t want to have to allocate page tables contiguously
  - So use same solution: map page tables using another page table
  - To stop recursion: the page-table page table ("system PT") resides in contiguous memory, usually at a known fixed address
    - Win: Page tables can be pieced together from scattered pages
    - Win: invalid mappings can be represented with invalid addresses rather than requiring space in a table.
    - Lose: Worst case lookup?

Extending idea: hierarchical page tables
- Large address range = lots of unused space = unwieldy PTs
  - Conceptually: map small regions with direct page table, then these with page tables, ...
    - space savings with $2^{24}$ (16M) regions?

Problem: mapping = slow
- If each memory reference requires 1 or more page table translations, speed will suck
  - The obvious idea: caching.
    - What to cache? VA-to-PA translations
      - Why? program doesn’t wildly access entire address space
        - usually references (relatively) small, slowly-changing subsets
        - So, just cache the translations for these regions!
Translation look-aside buffer (TLB)

- TLB is just a very fast memory with some comparators
- Each TLB entry maps a VPN to PPN + protection information
- On each memory reference: check TLB, if there, fast. If not insert in TLB for next time. (Must remove some entry)
- Typical: 64-2K entries, 95% hit rate

Virtual addr

$$3\quad 256\ (12\text{bits})$$

VPN page offset

mem

$$0x10ff\quad (4<12)|0xff$$

T

TLB

Prot VPN PPN

r 3 1

Invalid access

TLB details

- TLB operates at CPU pipeline speed
- Small, high speed
- Complication: what to do when switch address space?
  - Flush TLB on context switch (e.g., x86)
  - Tag each entry with associated process's ID (e.g., MIPS)
- In general, OS must manually keep TLB in valid state.
  - Change a page table entry
  - Complexity on a multiprocessor (TLB shootdown)
- TLB reload can be done to hardware or software
  - Hardware – x86 walks page tables on miss.
  - Software – MIPS – trap to OS to reload.
  - OS get to decide "page table" format.

Example: MIPS R2000/R3000 TLB

- Used in DecStations and SGI machines
- 64 entries, fully associative
- TLB entry format (64 bits per entry)

<table>
<thead>
<tr>
<th>VPN</th>
<th>PID</th>
<th>Undef</th>
<th>PFN</th>
<th>N</th>
<th>D</th>
<th>V</th>
<th>G</th>
<th>Undef</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>6</td>
<td>6</td>
<td>20</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

G - Global, valid for any PID (why?)
V - entry is valid (why have this?)
D - dirty bit, page has been modified
N - don't cache memory addresses (why?)
PFN - physical address of the page
PID - process id for the page (how many? How to reuse?)
VPN - virtual page number
Undef - undefined bits, set by OS (why?)

Where does OS live?

- Different address space?
  - "Microkernels"
  - Nice: "catch" wild pointer writes
  - Bad: Accessing user stuff clumsy. Having OS handle TLB faults on its own data can be a (real) pain.
- Common: user addresses co-exist with OS's:
  - OS aliased into every application address space at the same place (allows user addresses to co-exist with OS's)
    - e.g., MIPS: the upper 2 gig of address space reserved for OS.
    - As a refinement: OS runs "unmapped"
      - special address range in "privileged" mode that "mapped" to physical memory by subtracting constant
        - e.g., MIPS: addresses >= 0x80000000 have this subtracted
  - What happens when an application tries to read/write OS data?

Problem: accessing user data

- OS needs to access user "stuff"
  - I/O routines read/write user buffers. BUT:
    - User pointers cannot be trusted
      - obvious: user passes in null pointer or invalid address "0xfffee000"
      - less obvious: user passes in valid "kernel" address
    - User memory might bepaged out.
      - OS will get a page fault in middle of system call.
      - If it switches to new process then:
        - Prob 1: if kernel held single OS lock, system will deadlock.
        - Prob 2: if system call halfway through, and depends on current state, when restarted will have invalid view of world.
        - Prob 3: if system call partially altered OS data structures...
    - User pointers only valid in user address space
      - What happens if OS stores them away?

Summary: Virtual memory mapping

- What?
  - Give programmer logical (virtual) set of addresses rather than actual physical set
- How?
  - translate every memory operation using table (page table, segment table).
  - Speed: cache frequently used translations
- Result?
  - each user has a private address space
  - programs run independently of actual physical memory addresses used, and actual memory size
  - protection: check that they only access their own memory
- Next: partial residency or: paging