Large address spaces = Large tables

- On 64-bit systems, virtual address spaces are up to $2^{48}$ bytes in size
- Given 4KB page size and 8 byte page table entries
  - Page table size = $(2^{48} \div 2^{12} \text{ pages}) \times 2^3 \text{ bytes/page}$
    $= 2^{39} \text{ bytes} = 512\text{GB}$
- Most of the address space will be unmapped — i.e., the page table is a very sparse data structure
- How to reduce the size of page tables (without increasing page size)?
Reducing page table size

- Option 1: constrain the scope of page tables with segments
  - Each segment describes a relatively small linear address space
  - Each linear address space is mapped using a separate page table

- Option 2: multi-level page tables
  - Break up a monolithic page table into a tree structure
  - Page table walk searches for leaf node containing PPN
Segmentation + Paging

- Each segment is associated with a page table (located via base address)
- Kernel maintains PTs and updates base/limit registers on context switches
Multi-level page tables

- Split virtual page number into multiple fragments — each acting as an index into a separate level of paging structures
- Unused tables (i.e., without any valid entries) don’t need to be allocated
E.g., single level (linear) page table

- 8-bit addresses
  - 32-byte pages

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

VPN — offset

Page Table

![Diagram showing the mapping of VPN and offset to page table entries.](image-url)
E.g., multi-level page table

- 8-bit addresses
- 32-byte pages

page "directory"

all unmapped; don’t need in memory!
E.g., multi-level page table

- 8-bit addresses
- 32-byte pages
Multiple (fixed) page sizes

- Multi-level paging makes it possible to accommodate multiple page sizes

- Each level of paging partitions physical memory into smaller pieces (intuitively, fewer bits lefts over for offset field)

  - “Skipping” one or more levels results in mapping larger pages from virtual to physical space

- Mapping large pages may greatly improve TLB effectiveness
E.g., 4KB and 4MB pages

VA₁:

VA₂:
Multi-level page table pros/cons

- **Pros:**
  - May reduce page table footprint
  - Allocate levels as needed
  - Multiple page sizes may coexist
  - Large pages help TLB while reducing PT size

- **Cons:**
  - Page table walk is expensive!
  - Requires multiple memory accesses for translation
  - More complex to access/manage
  - Kernel must maintain PT data structures for each process
Physical memory limits

- Even with all VM memory techniques covered so far, aggregate process memory requirements may exceed available physical memory

- What to do?
  - Offload memory burden to disk
    - “Swap space” set aside to hold non-resident pages
Swapping
Memory hierarchy

- Goal: prioritize using the *fast but scarce* types of memory
- Fall back on the slower but more plentiful types as needed
- Compiler maps variables to registers
- Hardware maps memory accesses to cache lines
- Who should map memory to disk?
Manual vs. Automatic swapping

- Option 1: user decides what to keep in memory and what resides on disk
  - Swapping is a manual task
  - Most control, but painful for any non-trivial application!
- Option 2: kernel automatically swaps data into and out of memory as needed by processes
  - Users can ignore physical memory constraints (to an extent)
  - Common approach: use pages as the unit of swapping
Page status

- Need to distinguish between the questions of whether access to a virtual page is *legal* and whether the page currently *resides in physical memory*.

- Expand page table entry metadata to include both:
  - *Valid flag*: is the request for a legal page?
  - *Present flag*: is the corresponding data loaded in physical memory?
    - If not, data is in swap space; PTE contains disk address.
Valid & Present flags

VA₁:

<table>
<thead>
<tr>
<th>VPN</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

VA₂:

<table>
<thead>
<tr>
<th>VPN</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PA₁:

<table>
<thead>
<tr>
<th>PPN</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Disk (Swap space)

Page fault (no PA)
Page fault

- A page fault can be generated by the MMU when:
  - An invalid page is accessed
  - Access control assertions fail (e.g., insufficient privilege)
  - A page is not currently present in physical memory
  - Kernel is responsible for swapping data in from disk and updating the page table(s)
Address translation: page present

1. VA: N
2. Page table walk
3. PA: N'
4. Data

CPU → Address Translator (part of MMU) → Page Table → Physical Page Table
Address translation: page fault

1. VA: N (retry)
2. page fault
3. transfer control to kernel
4. PTE update
5. data transfer
6. data
7. VA: N (retry)
8. page table walk
9. PA: N'
10. Address Translator (part of MMU)
11. Physical Page Table
12. kernel
13. Swap space
14. CPU
Medium term scheduler

- Ready
- Running
- Blocked

Swap in/out

Diagram showing the flow between states: Ready → Running → Blocked → Ready → Blocked → Ready.
Medium term scheduler

- Kernel module responsible for swapping processes & pages
- If memory is low, may need to evict in-memory pages to make room
  - Common page-replacement policy: least-recently used (LRU)
- Swap-outs are driven by memory usage threshold — kernel will evict pages proactively to ensure minimum memory availability
- Pages may be swapped in *on demand* or by *prefetching* (e.g., based on spatial locality)
If all else fails ... 

- Worst case scenario: total activate process memory footprints is too large for physical memory — constantly swapping pages in/out
  
  - Situation known as **thrashing**

- What to do? 
  
  - Suspend execution of some subset of processes
  
  - Terminate memory-intensive processes (ideally, restartable ones)