Summary – Address Spaces and Paging

- each process has its own *virtual address space*
  - virtual addresses start at 0 for user code
  - OS code assigned high virtual addresses
- both virtual and physical address spaces are divided into fixed-size chunks
  - virtual address space – *pages*
  - physical address space – *page frames*, each able to hold a page
  - fixed-size chunks avoid external fragmentation
  - small page size reduces internal fragmentation (at the expense of page table size)
- *page table* maps virtual page numbers to physical page frames
  - one per process
  - used by the MMU when the CPU accesses memory

Beyond Physical Memory

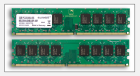
- so far we have assumed that all of the address spaces for all of the running processes fit into physical memory
  - not necessary – a process may not use all of its address space at once
  - not possible – with large address spaces
    - want to have large virtual address spaces so processes are not limited in how much memory they can use
- *demand paging* only brings pages into memory when they are needed by a process

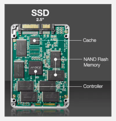
Beyond Physical Memory

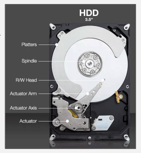
Where can pages be stored?

storage gets slower as it gets bigger – farther from the CPU, different technologies

- memory (storage) hierarchy
  - registers – essentially no latency
  - cache – latency 1-10 ns
    - TLB, instruction cache, data cache
  - RAM – latency 10-30 ns
  - non-volatile storage
    - SSD (solid state drive) – latency 25-100 µs
    - HDD (hard disk drive) – latency 10-15 ms





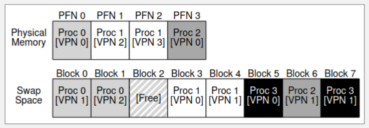


THE CRUX: HOW TO GO BEYOND PHYSICAL MEMORY
How can the OS make use of a larger, slower device to transparently provide the illusion of a large virtual address space?

https://pages.cs.wisc.edu/~remzi/OSTEP/vm-beyondphys.pdf
https://www.diskmfr.com/know-how-internal-structure-details-of-solid-state-drives/
https://en.wikipedia.org/wiki/Random-access_memory

Swap Space

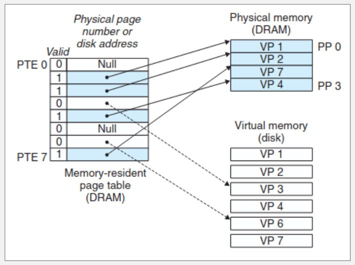
- *swap space* is the portion of disk space used for storing pages not in main memory
  - generally much larger than physical memory
  - can exist as a *swap partition* or a *swap file*
    - a dedicated partition allows the swap space to be a contiguous block (better performance)
    - swap files are regular files and can be more easily resized



even a single virtual address space no longer needs to fit entirely in main memory

Demand Paging

- OS must keep track of the *disk address* of each page
  - can use the PFN bits in the page table

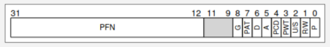


treating physical memory as a cache for the disk – pages are copied rather than moved into memory when they are swapped in – can improve performance when pages are evicted

Present Bit

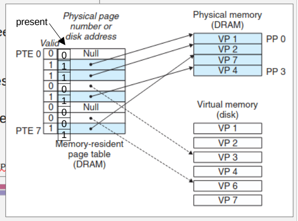
- include a *present bit* in the PTE to indicate if an allocated page is in memory

- page table maps VPN to a *page table entry* (PTE)



  - PFN – physical frame number
  - P – *present bit*
    - is page in physical memory or swapped out to disk
    - doubles as *valid bit* for the x86 – 1 means in physical memory and valid, 0 means swapped out (but valid) or not valid
      - unallocated space between heap and stack is invalid
      - valid bit supports sparse address spaces by removing the nee physical page frames for unused pages in address space
  - R/W (*read/write bit*), U/S (*user/supervisor bit*)
    - *protection bits* – are writes allowed? can user mode proces
  - A (*accessed* or *reference bit*), D (*dirty bit*)
    - whether page has been accessed (used in page replaceme since it was brought into memory
  - G, PWT, PCD, PAT – related to hardware caching

https://pages.cs.wisc.edu/~remzi/OSTEP



Page Fault Handler

- when a page fault occurs, control transfers to the OS *page fault handler*
  - regardless of whether the TLB is hardware- or software-managed, page faults are typically handled by software
    - disks are slow – software adds minimal overhead
    - handling page faults is complex – difficult to do in hardware

Handling Page Faults

- handling page faults
  - look up disk address in PTE
    - use the PFN bits to store the disk address for swapped-out pages
  - issue disk I/O request to fetch the page into memory

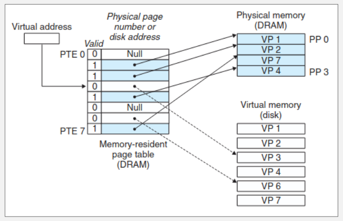
    [process blocks until I/O completes – other processes can run in the meantime]
  - update the page table
    - mark page as present
    - update PFN to reflect location of page in memory
  - retry the instruction that generated the page fault
    - if the TLB is updated by the page fault handler, the retry will succeed (TLB hit)
    - otherwise the retry will cause a TLB miss and a third retry before the original memory access succeeds

Page Replacement

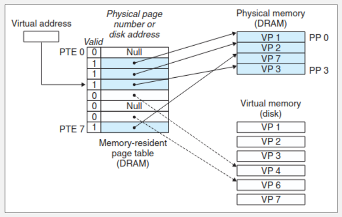
- if there is no free memory when a page fault occurs, *page replacement* is needed
  - identify a *victim page* to remove
    - does not need to be from the same process (though it could be)
  - copy the contents of the victim page to the swap space
  - copy the contents of the page on disk into the physical frame
  - (update page table entries and) retry access

Handling Page Faults

VM page fault (before). The reference to a word in VP 3 is a miss and triggers a page fault.



VM page fault (after). The page fault handler selects VP 4 as the victim and replaces it with a copy of VP 3 from disk. After the page fault handler restarts the faulting instruction, it will read the word from memory normally, without generating an exception.



Putting It All Together

address translation with a hardware-managed TLB and swapping

first check the TLB

if TLB hit, do the address translation and access memory

if TLB miss, load PTE from page table

if page is present, load the PTE into the TLB and retry

if the page is not present, raise a *page fault*

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)   // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else                  // TLB Miss
    PTEAddr = PTBR + (VPN * sizeof(PTE))
    PTE = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
    else
        if (CanAccess(PTE.ProtectBits) == False)
            RaiseException(PROTECTION_FAULT)
        else if (PTE.Present == True)
            // assuming hardware-managed TLB
            TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
            RetryInstruction()
        else if (PTE.Present == False)
            RaiseException(PAGE_FAULT)
```

page fault handler

if there is no free memory, *page replacement* is needed

```
PFN = FindFreePhysicalPage()
if (PFN == -1)              // no free page found
    PFN = EvictPage()       // replacement algorithm
DiskRead(PTE.DiskAddr, PFN) // sleep (wait for I/O)
PTE.present = True          // update page table:
PTE.PFN     = PFN           // (present/translation)
RetryInstruction()          // retry instruction
```

Page Replacement

- performance can be increased by not waiting until memory is full to evict pages
  - define two thresholds: a *high watermark* and a *low watermark*
  - a background *swap daemon* (or *page daemon*) thread handles swapping pages out
  - if there are fewer than LW free pages, the swap daemon evicts pages until there are at least HW pages free

with a swap daemon, page fault handler notifies it to free some pages, then blocks until a page is available

```
PFN = FindFreePhysicalPage()
if (PFN == -1)              // no free page found
    PFN = EvictPage()       // replacement algorithm
DiskRead(PTE.DiskAddr, PFN) // sleep (wait for I/O)
PTE.present = True          // update page table:
PTE.PFN     = PFN           // (present/translation)
RetryInstruction()          // retry instruction
```

  - advantages
    - increases the likelihood of the necessary pages being free when the page fault handler needs them
    - clustering page replacements allows for increased efficiency with disk I/O

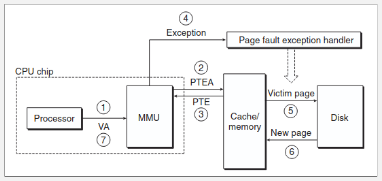
Page Replacement

- *file-backed pages* contain data from files
  - e.g. program code
- *anonymous pages* do not
  - e.g. stack, heap
- *dirty pages* are those whose content has changed since being loaded into memory
- observations
  - file-backed pages don't need to be copied into swap space
  - non-dirty victim pages don't need to be copied out when removed
- add info to PTE to track
  - e.g. *dirty bit*

Summary – Mechanisms

To support swapping –

- add a *present bit* to the page table entry
  - to keep track of whether the page is in memory or on disk
- add a *dirty bit* to the page table entry
  - to keep track of whether the page has been modified
- have a *page-fault handler* to swap pages in



1) CPU accesses the virtual address (VA)
2) TLB miss – MMU up the page table in memory to find the PTE
3) get PTE, present bit not set
4) MMU traps to OS (page fault)
5) OS swaps out victim page (if needed)
6) OS swaps in new page (if needed), updates PTE
7) retry original access