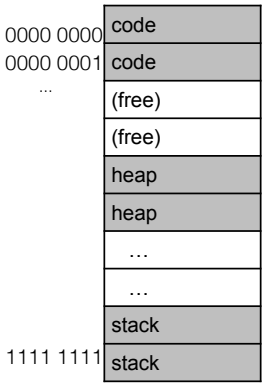
Virtual Memory

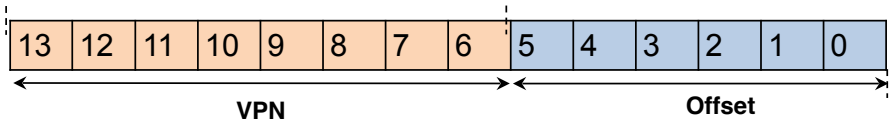
- 13 - Address Spaces
- 14 - Memory API
- 15 - Address Translation
- 16 - Segmentation
- 17 - Free Space Management
- 18 - Paging
- 19 - Translation Lookaside Buffers
- 20 - Advanced Paging
- 21 - Swapping
- 22 - Swapping Policy

Two-level example



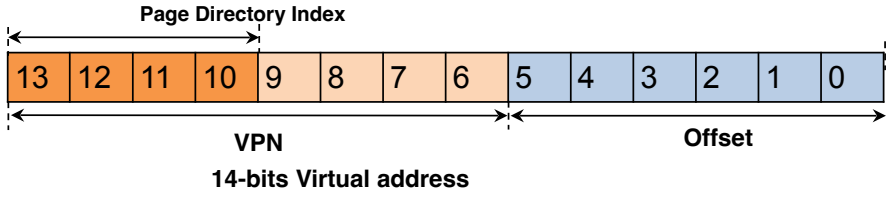
Flag	Detail
Address space	16 KB
Page size	64 byte
Virtual address	14 bit
Num pages	$2^{14}/2^6 = 2^8 = 256$ pages
VPN	8 bit
Offset	6 bit
Page table entry	4 bytes

A 16-KB Address Space With 64-byte Pages



Two-level example

- Page directory has one entry per page of the page table
  - 256 pages, 4 bytes for PTE, 64-byte pages
    - $\frac{256 \times 4}{64} = 16$ pages for single-level directory
    - each page can hold 64/4=16 entries:
      - —> 4 bits for PDI
      - —> 4 bits for PTI
- Accessing invalid page-directory entry raises exception



- Page-table index (PTI) is used to index into the page table page

Two-level Page Tables *with TLB*

```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)    // TLB Hit
4    if (CanAccess(TlbEntry.ProtectBits) == True)
5      Offset   = VirtualAddress & OFFSET_MASK
6      PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7      Register = AccessMemory(PhysAddr)
8    else
9      RaiseException(PROTECTION_FAULT)
10 else                    // TLB Miss
11   // first, get page directory entry
12   PDIndex = (VPN & PD_MASK) >> PD_SHIFT
13   PDEAddr = PDBR + (PDIndex * sizeof(PDE))
14   PDE     = AccessMemory(PDEAddr)
15   if (PDE.Valid == False)
16     RaiseException(SEGMENTATION_FAULT)
17   else
18     // PDE is valid: now fetch PTE from page table
19     PTIndex = (VPN & PT_MASK) >> PT_SHIFT
20     PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
21     PTE     = AccessMemory(PTEAddr)
22     if (PTE.Valid == False)
23       RaiseException(SEGMENTATION_FAULT)
24     else if (CanAccess(PTE.ProtectBits) == False)
25       RaiseException(PROTECTION_FAULT)
26     else
27       TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
28       RetryInstruction()
```

no page table mem ever accessed

find and retrieve PDE, check for validity

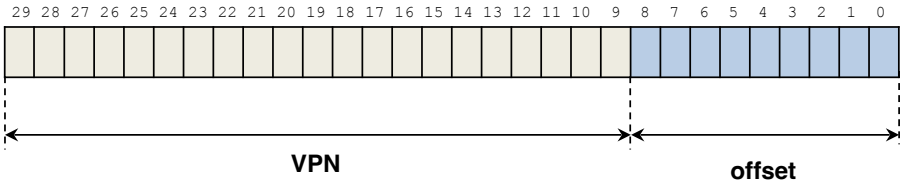
retrieve PTE, check for validity

insert PTE in TLB

Figure 20.6: **Multi-level Page Table Control Flow**

More than two levels

- In some cases, a deeper tree is possible.

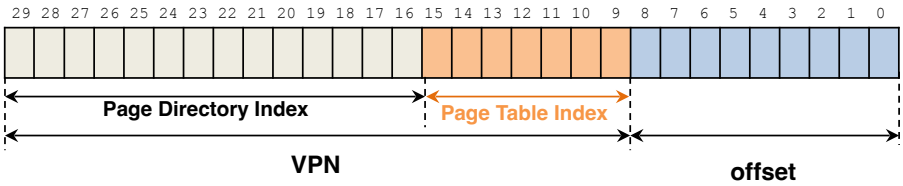


Flag	Detail
Virtual address	30 bits
Page size	512 bytes
VPN	21 bits
Offset	9 bits
PTE size	4 bytes

21-bit PDI means $2^{21} * 2^2 = 2^{23} =$ 8 MB for a single-level page table
- far too large

More than two levels

- Deeper trees are possible

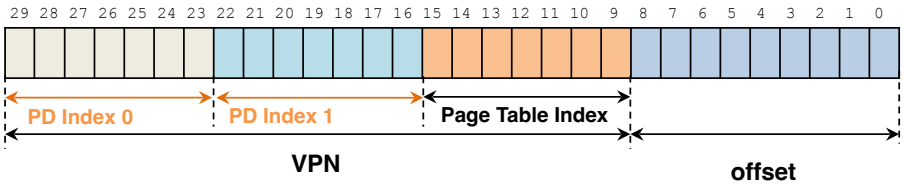


Flag	Detail
Virtual address	30 bit
Page size	512 byte
VPN	21 bit
Offset	9 bit
PTEs per page	128

14-bit PDI means $2^{14} = 16$K PTEs, each 4 bytes, so PD is $2^{14} * 2^2 = 2^{16}$ bytes
- 128 pages for the page directory.
- still too large

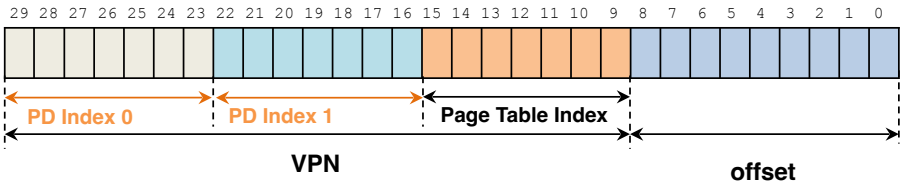
More than two levels

- If our page directory has 2^{14} entries, it spans 128 pages
- So….. we build a third level of the tree, by splitting the page directory itself into multiple pages of the page directory



- Each page has 128 entries (2^7):
  - Each page of page table needs 7 bits to index
  - Only one page for the page directory

More than two levels



Assume:	PTEs	PDE1s	PDE0s
• stack 768 bytes	2	1	1
• heap 1800 bytes	4	1	1
• code 1000 bytes	2	1	1
	probably three pages	*probably three pages*	*all same page*

only 7 total pages in entire page table

Inverted Page Tables

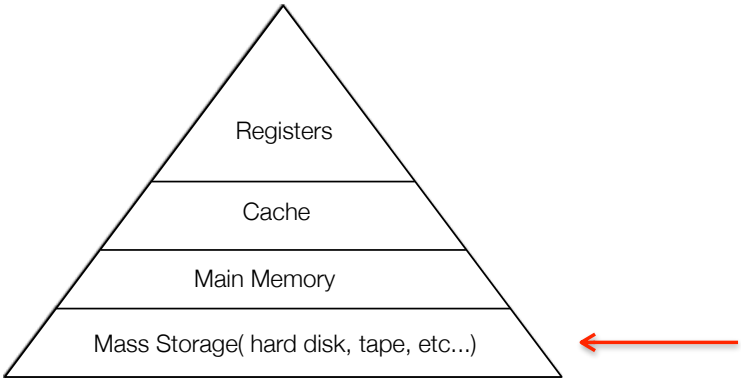
- Keeping only a single page table that has
  - an entry for each physical page of the system
- The entry tells us
  - which process is using this page, and
  - which virtual page that maps to this physical page
- Finding translating a virtual address now requires a search!
  - But can use a per-process hash (PowerPC)
  - Hash has entry for each *used* virtual/physical page, pointing to the single global page table

Virtual Memory

- 13 - Address Spaces
- 14 - Memory API
- 15 - Address Translation
- 16 - Segmentation
- 17 - Free Space Management
- 18 - Paging
- 19 - Translation Lookaside Buffers
- 20 - Advanced Paging
- 21 - Swapping and Demand Paging
- 22 - Swapping Policy

Beyond Physical Memory *mechanisms*

- Require an additional level in the memory hierarchy.
  - OS needs a place to stash portions of the address space not currently used
  - Usually served by *swapping to a hard drive*



Memory Hierarchy in modern systems

Single large address for a process

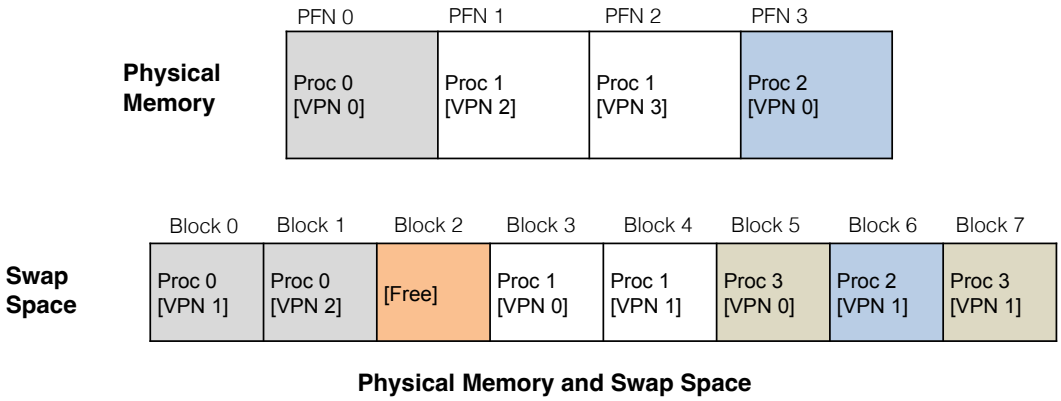
- Need to arrange for the code or data to be in memory before calling a function or accessing data.

- Beyond just a single process
  - The addition of *swapping* allows the OS to support the illusion of a large virtual memory for multiple concurrently-running processes

  - *even if total used virtual memory exceeds physical memory*

Swap Space

- Reserve some space on the disk for swapping pages



Physical Memory and Swap Space

- Processes 0, 1, 2 have page in physical memory
  - not all of them
  - process 3 is completely swapped out

Present Bit

- Need PTE bit to support swapping pages to disk.
  - When the hardware looks in the PTE, it may find that the page is not present in physical memory.

Value	Meaning
1	page is present in physical memory
0	The page is not in memory but rather on disk.

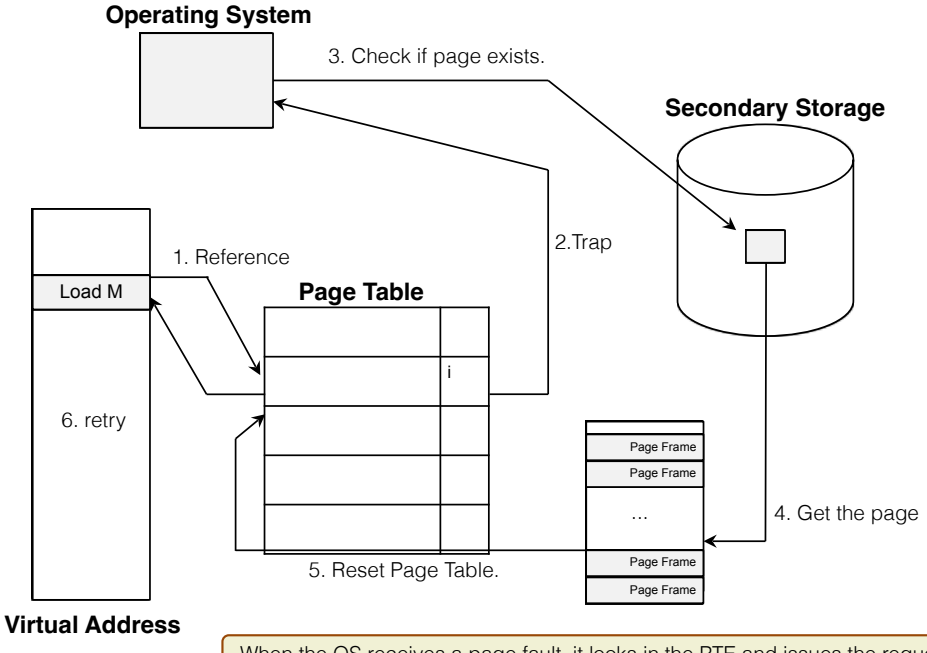
- OS often needs to make room for the new pages
  - Picking a page to replace is the *page-replacement* or *victim-selection policy*

The Page Fault

- Accessing page that is not in physical memory.
  - A page with *false* present bit has either:
    - never been in-core (lazily loaded), or
    - has been swapped out to disk

Page Fault Control Flow

- PTE used for data such as the PFN of the page for a disk address.



When the OS receives a page fault, it looks in the PTE and issues the request to disk.

Single-Level Page Fault *hardware*

```
1:        VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2:        (Success, TlbEntry) = TLB_Lookup(VPN)
3:        if (Success == True) // TLB Hit
4:            if (CanAccess(TlbEntry.ProtectBits) == True)
5:                  Offset = VirtualAddress & OFFSET_MASK
6:                  PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7:                  Register = AccessMemory(PhysAddr)
8:            else
9:                  RaiseException(PROTECTION_FAULT)
10:       else // TLB Miss
11:           PTEAddr = PTBR + (VPN * sizeof(PTE))
12:           PTE = AccessMemory(PTEAddr)
13:           if (PTE.Valid == False)
14:                 RaiseException(SEGMENTATION_FAULT)
15:           else
16:               if (CanAccess(PTE.ProtectBits) == False)
17:                     RaiseException(PROTECTION_FAULT)
18:               else if (PTE.Present == True)
19:                   // assuming hardware-managed TLB
20:                   TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
21:                   RetryInstruction()
22:               else if (PTE.Present == False)
23:                     RaiseException(PAGE_FAULT)
```

no page table mem ever accessed

crash?

all good

page fault

Single-Level Page Fault *software*

```
1:        PFN = FindFreePhysicalPage()
2:        if (PFN == -1) // no free page found
3:                PFN = EvictPage() // run replacement algorithm
4:                DiskRead(PTE.DiskAddr, pfn) // sleep (waiting for I/O)
5:                PTE.present = True // update page table with present
6:                PTE.PFN = PFN // bit and translation (PFN)
7:                RetryInstruction() // retry instruction
```

- The OS must find a physical frame for the soon-be-faulted-in page
- If no such page, run replacement algorithm (often asynchronous)

When Replacements Really Occur

- Wait until memory entirely full?
  - No, proactively try to keep small portion of memory free
- Swap or Page Daemon
  - Frees/evicts page frames if fewer than a low-water threshold available
  - …until a high-water threshold pages available

Virtual Memory

- 13 - Address Spaces
- 14 - Memory API
- 15 - Address Translation
- 16 - Segmentation
- 17 - Free Space Management
- 18 - Paging
- 19 - Translation Lookaside Buffers
- 20 - Advanced Paging
- 21 - Swapping
- 22 - Swapping Policy

Beyond Physical Memory: Policies

- Memory pressure forces the OS to start paging out pages to make room for actively-used pages.
- Deciding which page to evict is encapsulated within the replacement policy of the OS.

Swap Management

- Goal in picking a replacement policy for this *cache* is to minimize the number of cache misses.
- The number of cache hits and misses let us calculate the *average memory access time (AMAT)*.

$$\text{AMAT} = T_M + P_{miss} * T_D$$

Argument	Meaning
T_M	The cost of accessing memory
T_D	The cost of accessing disk
P_{hit}	The probability of finding the data item in the cache(a hit)
P_{miss}	The probability of not finding the data in the cache(a miss)

The Optimal Replacement Policy *OPT*

- Leads to the fewest number of misses overall
  - Replaces the page that will be accessed furthest in the future
  - Resulting in the fewest-possible cache misses
- Not achievable

Tracing the Optimal Policy

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	2	0,1,3
0	Hit		0,1,3
3	Hit		0,1,3
1	Hit		0,1,3
2	Miss	3	0,1,2
1	Hit		0,1,2

replace the page first subsequently referenced furthest in the future

6 hits
5 misses

Reference Row

0 1 2 0 1 3 0 3 1 2 1

A Simple Policy: Random

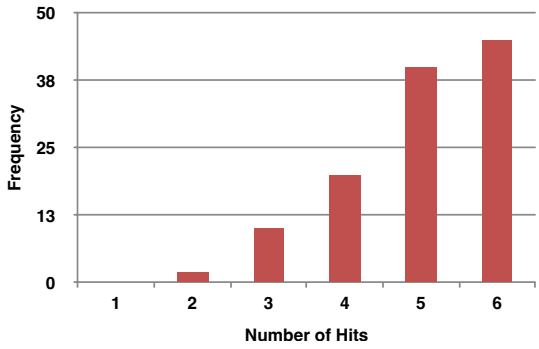
- Pick a random page to replace under memory pressure:
  - No attempt to do anything fancy
  - Performance depends entirely on random chance

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	0	1,2,3
0	Miss	1	2,3,0
3	Hit		2,3,0
1	Miss	3	2,0,1
2	Hit		2,0,1
1	Hit		2,0,1

5 hits
6 misses

Random Performance

- Sometimes, Random is as good as optimal, achieving 6 hits on the example trace.



Random Performance over 10,000 Trials

The Exam (all point totals approximate)

- GeekOS and general kernel structure:
- Queueing:
  - Characteristics
  - Deriving queue lengths
  - turnaround time
  - etc.…
- Paging and memory systems:
  - segmentation
  - paging
  - multi-level page tables
- Paging and swap mechanisms systems:
  - victim-replacement policies: LRU, FIFO, OPT

The Exam (all point totals approximate)

- GeekOS and general kernel structure: 10 pts
- Queueing: 25 pts
  - Characteristics
  - Deriving queue lengths
  - turnaround time
  - etc.
- Paging and memory systems: 25 pts
  - segmentation
  - paging
  - multi-level page tables
- Paging and swap mechanisms systems: 25 pts
  - victim-replacement policies: LRU, FIFO, OPT