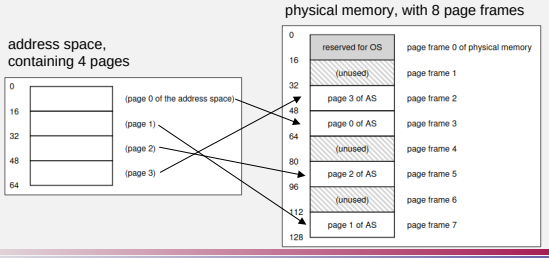
Paging

- dividing memory into variable-sized pieces leads to problems with fragmentation
- instead, divide memory into fixed-sized *pages*
  - view physical memory as an array of fixed-sized *page frames*
    - each page frame can contain one virtual page

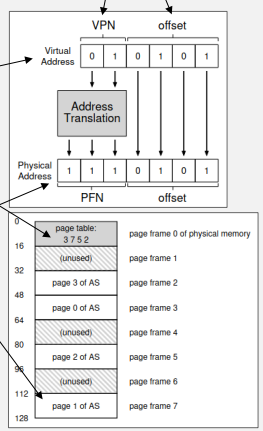


Address Translation Example

virtual address is split into VPN and offset

```
movl 21, %eax
```

- load from virtual address 21
  - 21 is 010101 in binary
    - top two bits are the VPN (virtual page number) – 01
  - look up entry 01 in the page table
    - PFN (physical frame number) is 7 (111)
  - replace VPN with PFN to get the translated address
    - translated address is 1110101 (117)
    - no change to offset because virtual pages and page frames are the same size



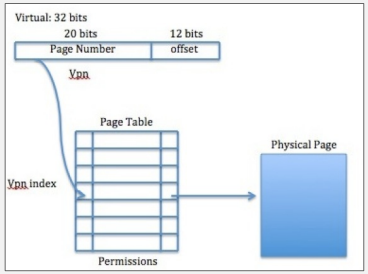
Page Table Organization

- 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 need to allocate 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 processes access?
  - A (*accessed* or *reference bit*), D (*dirty bit*)
    - whether page has been accessed (used in page replacement) or modified since it was brought into memory
  - G, PWT, PCD, PAT – related to hardware caching

Page Table Organization

- page table maps VPN to a *page table entry* (PTE)
- a *linear page table* is an array indexed by VPN

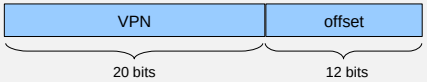


Paging

- advantages of paging
  - avoids external fragmentation
    - but can have internal fragmentation if page size doesn't match process needs
  - flexible – system can support address space abstraction independently of how that address space is used
    - e.g. no longer need direction of growth bits
  - simplifies free-space management
    - OS can use any *n* free pages for address space
    - no searching for suitable free space or need for coallescing freed chunks
- problems
  - page tables as described are too big
  - page tables as described are too slow

Too Big

- page tables are too large to be stored in registers
  - 32-bit address space (4GB memory) with 4KB pages means a 12-bit offset and a 20-bit VPN



    - 4KB page means 4096 addresses
      - 12 bits = 2^{12} = 4096
    - 20 bits = 2^{20} = 1,048,576 page table entries
      - 32 bit physical addresses means 4 bytes per entry = 4MB per process
- 4MB per process is too much memory
  - even worse with 64-bit address spaces

Too Slow

- details of address translation with paging
  - PTBR = *page table base register*
    - physical address of base of current process page table

```
1  // Extract the VPN from the virtual address
2  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3
4  // Form the address of the page-table entry (PTE)
5  PTEAddr = PTBR + (VPN * sizeof(PTE))
6
7  // Fetch the PTE
8  PTE = AccessMemory(PTEAddr)
9
10 // Check if process can access the page
11 if (PTE.Valid == False)
12     RaiseException(SEGMENTATION_FAULT)
13 else if (CanAccess(PTE.ProtectBits) == False)
14     RaiseException(PROTECTION_FAULT)
15 else
16     // Access OK: form physical address and fetch it
17     offset   = VirtualAddress & OFFSET_MASK
18     PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19     Register = AccessMemory(PhysAddr)
```

virtual address 21 = 010101_2
VPN_MASK 0x30 = 110000_2
SHIFT = 4
& result 010000_2
result of shift (VPN) 01_2

index into page table

OFFSET_MASK = 001111_2
offset = 000101_2

PTE.PFN = 111_2
result of shift 1110000_2
| result (PhysAddr) 1110101_2

  - this requires *two* memory accesses for each memory reference – a significant slowdown

Memory Trace

assume `%edi` holds the base address of the array, `%eax` holds the current array index `i`

```
int array[1000];
...
for (i = 0; i < 1000; i++)
    array[i] = 0;
```

```
1024 movl $0x0,(%edi,%eax,4)
1028 incl %eax
1032 cmpl $0x03e8,%eax
1036 jne  1024
```

move the value 0 to the virtual memory address of the location of the array (multiply by 4 because integers are 4 bytes each)

increment the array index (in `%eax`)

compare the contents of `%eax` to 0x03e8 = 1000

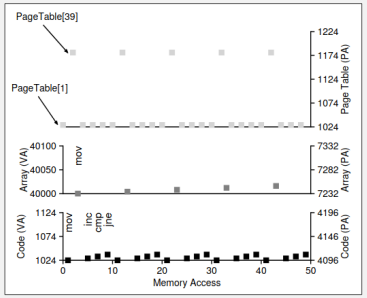
if the values are not equal, jump to instruction 1024 (the beginning of the loop) – `jne`

Memory Trace

```
int array[1000];
...
for (i = 0; i < 1000; i++)
    array[i] = 0;
```

```
1024 movl $0x0,(%edi,%eax,4)
1028 incl %eax
1032 cmpl $0x03e8,%eax
1036 jne  1024
```

- assumptions
  - 64KB virtual address space with page size 1KB
  - linear page table at physical memory address 1KB (1024)
- pages needed
  - code – virtual address 1024 is VPN 1
  - array – 1000 integers = 4000 bytes
    - assume at virtual addresses 40000-44000 (VPN 39-42)



memory accesses for i = 0..4 (5 iterations)

virtual addresses (VA) on left; corresponding physical addresses (PA) on right

instruction memory accesses (code) in black
array accesses in gray
page table memory accesses in light gray
→ 10 per iteration

Too Slow

So far –

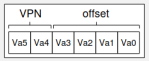
- hardware has circuitry for address translation
- page table is in memory
- *page table base register* (PTBR) stores the address of the page table
- each memory reference requires two memory accesses
  - one for the page table, one for the actual memory address

Translation-Lookaside Buffer (TLB)

- two tactics for speeding things up
  - hardware
  - caching
- *translation-lookaside buffer* (TLB) is a hardware cache in the MMU
  - stores (some) virtual-to-physical address translations (VPN→PFN)
    - only use the page table if the desired translation isn't in the TLB
  - TLB is close to the CPU so very fast to access
    - use of TLB makes address translation fast enough

Address Translation with a TLB

- assumptions
  - linear page table (an array)
  - hardware-managed TLB




```
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
        TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
        RetryInstruction()
```

extract the VPN from the virtual address

check if TLB contains an entry for the VPN

for a hit –
check the protection bits to make sure this is a legal access

extract PFN from the TLB entry and concatenate with offset from the virtual address to get the physical address

access memory

for a miss –
compute address of PTE and load

check that the PTE is valid (0 can mean not allocated or just swapped out)

check the protection bits to make sure this is a legal access

add the translation to the TLB and retry the access (which will now result in a TLB hit)

About Caching

- a *hardware cache* keeps data in fast memory close to the CPU
  - processor checks the cache first (fast) and only goes to memory (slow) if the desired value isn't in the cache
  - adding the cache check makes retrieval slightly slower for things not in the cache, but much faster for things that are
- caches exploit *locality* in instruction and data references
  - *temporal locality* – a memory location that has been accessed is likely to be accessed again in the near future
  - *spatial locality* – nearby memory locations are likely to be accessed in the near future
- fast caches are necessarily small
  - physics matters – electricity is fast, but it isn't instantaneous

Handling TLB Misses

```
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
        TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
        RetryInstruction()
```

- hardware-managed TLB (CISC)
  - TLB miss is handled entirely by the hardware
  - PTBR stores the location of the page table in memory
  - hardware knows layout of page table to be able to look up the entry for a particular VPN
    - e.g. linear page table – index into an array

```
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
    RaiseException(TLB_MISS)
```

- software-managed TLB (RISC)
  - hardware raises an exception on a TLB miss
  - control transfers to a trap handler
    - trap handler looks up the VPN in the page table and updates the TLB (via a privileged instruction)
  - on return from trap, the hardware retries the instruction

Software-Managed TLB

- handling return-from-trap
  - normally execution resumes with the instruction *after* the trap into the OS
  - for a TLB miss, should retry the instruction that *caused* the trap
    - hardware must save a different PC when TLB miss causes the trap
- must prevent infinite chains of TLB misses
  - executing the trap handler involves loading instructions from memory...which could themselves cause TLB misses
  - e.g.
    - store TLB miss handlers in reserved location in physical memory so address translation is not needed
    - reserve some TLB entries for *wired translations* that are always valid