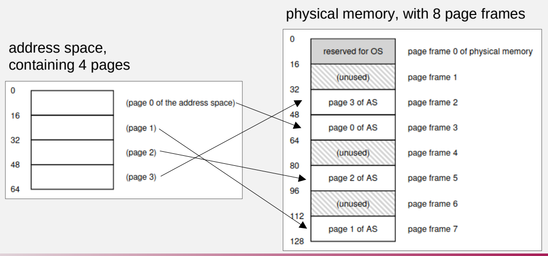
Paging

- 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

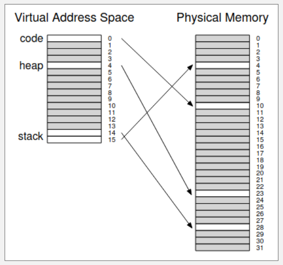


Too Big

Strategies for dealing with linear page tables being too big –

- bigger pages
  - suffers from internal fragmentation
    - default page size is typically small (4KB, 8KB)
  - supporting multiple page sizes can improve TLB performance for applications with high memory needs
    - tradeoff is more complex memory management due to variable-sized chunks allocated
- combine paging and segments
- multi-level page tables
- inverted page tables

Combining Paging and Segments

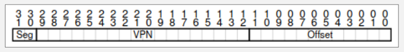


PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
23	1	rw-	1	1
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
28	1	rw-	1	1
4	1	rw-	1	1

- a sparse address space means a page table mostly full of invalid entries
  - most pages are not mapped to page frames

Combining Paging and Segments

- hybrid approach: one page table per segment instead of one page table per process
- implementation
  - base and bounds registers refer to the page table for the segment rather than the segment itself
    - base register points to the physical address of the page table
    - bounds register contains the number of pages in the page table



32-bit virtual address space, 4KB pages
segments – code 01, heap 10, stack 11

three segments means three page tables and three sets of base/bounds registers

```
SN           = (VirtualAddress & SEG_MASK) >> SN_SHIFT
VPN          = (VirtualAddress & VPN_MASK) >> VPN_SHIFT
AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))
```

for a hardware-managed TLB, the segment bits (SN) are used to determine which set of base/bounds registers to use

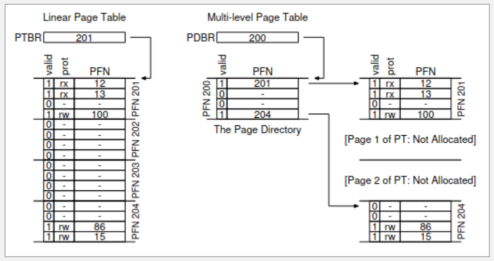
linear page table (array) – VPN is index where PTE is stored

Combining Paging and Segments

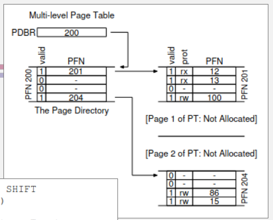
- reduces page table size
  - unallocated sections of the virtual address space aren't part of a segment and don't take up space in the page table
    - page table is only as big as needed for each segment
- still has the drawbacks of segmentation
  - assumes a certain pattern of usage of the address space
    - e.g. code, heap, stack segments; heap grows positively and stack grows negatively
  - a large but sparsely-used segment (e.g. heap) takes up allocated memory and thus page table space even though the space is unused
- external fragmentation can occur
  - segments are variable-sized, so page tables are also variable-sized (different numbers of PTEs)
    - need to find and manage multi-page chunks

Two-level Page Tables

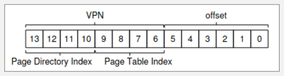
- split the page table into pages
- add a top level *page directory* containing the PFNs for pages of the page table
  - if a whole page of PTEs is invalid, no need to allocate that page
    - mark that page directory entry (PDE) as invalid



Two-Level Page Tables



split VPN into *page directory index* and *page table index*



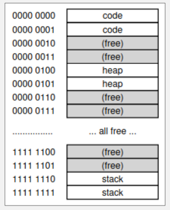
extract the VPN and look it up in the TLB

TLB hit – construct physical address from TLB entry

TLB miss –
– use page directory index to look up PDE
– if PDE is valid, use page table index to look up PTE
– if PTE is valid, insert PTE in TLB and retry instruction

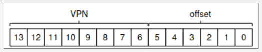
```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)   // TLB Hit
  if (CanAccess(TlbEntry.ProtectBits) == True)
    Offset   = VirtualAddress & OFFSET_MASK
    PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
    Register = AccessMemory(PhysAddr)
  else
    RaiseException(PROTECTION_FAULT)
else                  // TLB Miss
  // first, get page directory entry
  PDIndex = (VPN & PD_MASK) >> PD_SHIFT
  PDEAddr = PDBR + (PDIndex * sizeof(PDE))
  PDE     = AccessMemory(PDEAddr)
  if (PDE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
  else
    // PDE is valid: now fetch PTE from page table
    PTIndex = (VPN & PT_MASK) >> PT_SHIFT
    PTEAddr = (PDE.PFN<<SHIFT) + (PTIndex*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()
```

Example



0000 0000	code
0000 0001	code
0000 0010	(free)
0000 0011	(free)
0000 0100	heap
0000 0101	heap
0000 0110	(free)
0000 0111	(free)
....	... all free ...
1111 1100	(free)
1111 1101	(free)
1111 1110	stack
1111 1111	stack

16KB address space, 64-byte pages → 14-bit virtual address (2^{14} = 16KB) with 8 bit VPN, 6 bit offset (2^6 = 64)

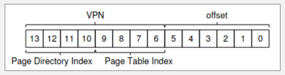


a linear page table with an 8 bit VPN requires 2^8 = 256 entries

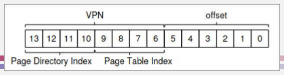
with a 4 byte PTE, 256 x 4 bytes = 1KB is needed to store the page table

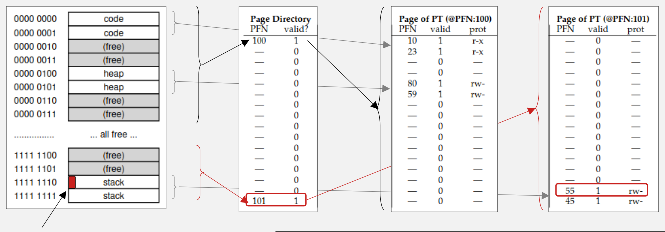
with 64 byte pages, a 1KB page table requires 1KB/64 = 16 pages with 4 byte PTEs, each page table page holds 64/4 = 16 PTEs

→ 16 pages means 4 bits for the *page directory index*
→ 16 PTEs per page means 4 bits for the *page table index*



Example





0000 0000	code
0000 0001	code
0000 0010	(free)
0000 0011	(free)
0000 0100	heap
0000 0101	heap
0000 0110	(free)
0000 0111	(free)
....	... all free ...
1111 1100	(free)
1111 1101	(free)
1111 1110	stack
1111 1111	stack

Page Directory

PFN	valid?
100	1
—	0
—	0
—	0
—	0
—	0
—	0
—	0
—	0
—	0
—	0
—	0
—	0
—	0
—	0
101	1

Page of PT (@PFN:100)

PFN	valid	prot
10	1	r-x
23	1	r-x
—	0	—
—	0	—
80	1	rw-
59	1	rw-
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—

Page of PT (@PFN:101)

PFN	valid	prot
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
—	0	—
55	1	rw-
45	1	rw-

$$\texttt{0x3F80} = \underbrace{\underbrace{\texttt{1111}}_{\text{PDIndex}}\ \underbrace{\texttt{1110}}_{\text{PTIndex}}}_{\text{VPN}}\ \underbrace{\texttt{000000}}_{\text{offset}}$$

PDIndex = `1111` → 15
  PDE @15 = (101,1)
  PDE.PFN = 101
PTIndex = `1110` → 14
  PTE @14 = (55,1,rw-)
  PTE.PFN = 55 = `00110111`
physical address = `00110111 000000`
  = `0x0DC0`

– extract the VPN from the virtual address
– extract the page directory index (`PDIndex`) from the VPN
– compute address of page directory entry (PDE)

```
PDEAddr = PDBR + (PDIndex * sizeof(PDE))
```

– if `PDE.Valid` is false (invalid), raise an exception
– extract the page table index (`PTIndex`) from the VPN
– compute address of page table entry (PTE)

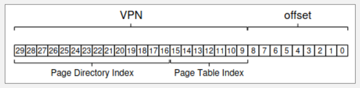
```
PTEAddr = (PDE.PFN<<SHIFT) + (PTIndex*sizeof(PTE))
```

– if `PTE.Valid` is false (invalid), raise an exception
– store in TLB and retry operation

```
PhysAddr = (PTE.PFN << SHIFT) + offset
```

More Levels

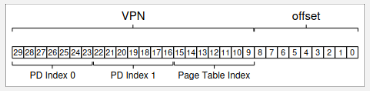
- consider a 30-bit virtual address space with a 512 byte page
  - 512 byte page → 9 bit offset
  - 512 byte page, 4 bytes per PTE → 128 PTEs per page
    → 7 bit page table index
  - the remaining 30-9-7 = 14 bits are the page directory index
    → 2^{14} 4 byte PDEs requires 128 pages to store



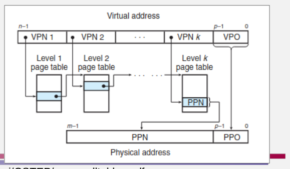
- in a two-level table the page directory is still treated as a linear page table `PDEAddr = PageDirBase + (PDIndex*sizeof(PDE))`
  - requires a contiguous chunk of 128 pages

More Levels

- create a directory for the directory



- on a TLB miss –
  - use PD Index 0 to fetch PDE from top-level directory
  - if valid, use top-level PFN combined with PD Index 1 to fetch PDE from second-level directory
  - if valid, use second-level PFN combined with page table index to fetch PTE
  - if valid, physical address is the PTE PFN combined with offset
- *k*-level page tables are possible
  - the top-level directory should be a single page



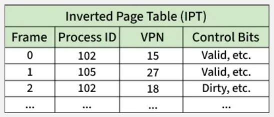
Multi-level Page Tables

- advantages
  - only need to allocate space for the page table for portions of the address space actually in use
    - efficiently supports sparse address spaces
  - no longer requires a contiguous chunk of physical memory for the whole page table (or any part of it)
    - page table pages can be allocated where there's room
  - page table sizes are in multiples of pages rather than PTEs
    - avoid external fragmentation due to variable-sized allocations
    - simplifies management
- costs
  - TLB misses are more expensive
    - *k*-level page table requires *k* memory accesses to find the PTE rather than one
  - more complicated to implement multi-level lookup vs linear page table

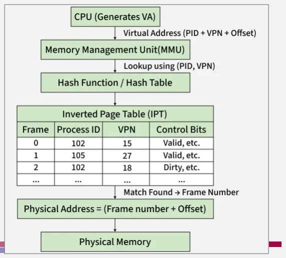
Inverted Page Tables

- one page table for the whole system
  - indexed by PFN
  - PTE stores which process is using each page frame and which VPN maps to it

- VPN→PFN requires searching the page table for the VPN
  - use data structures (e.g. hashtable) to speed up lookup



Inverted Page Table (IPT)			
Frame	Process ID	VPN	Control Bits
0	102	15	Valid, etc.
1	105	27	Valid, etc.
2	102	18	Dirty, etc.
...	...	...	...


CPU (Generates VA)
Virtual Address (PID + VPN + Offset)
Memory Management Unit(MMU)
Lookup using (PID, VPN)
Hash Function / Hash Table
Match Found → Frame Number
Physical Address = (Frame number + Offset)
Physical Memory

Inverted Page Table (IPT)			
Frame	Process ID	VPN	Control Bits
0	102	15	Valid, etc.
1	105	27	Valid, etc.
2	102	18	Dirty, etc.
...	...	...	...

Paging Summary

- a variety of strategies can be employed to reduce the size of page tables
  - bigger page sizes
    - not practical by itself due to increased internal fragmentation
    - supporting multiple page sizes can help with TLB performance
  - a page table for each segment
    - reduces the address space that each page table needs to cover
    - poor for sparse segments
    - more complex to manage and external fragmentation issues arise due to variable-sized page tables
  - multi-level page tables
    - handles sparse address spaces by only allocating space for the page table covering the parts of the address space actually in use
    - TLB misses are more expensive – one additional memory access per level of the page table
    - more complex to implement lookup
  - inverted page tables
    - store a single page table indexed by PFN rather than VPN
    - smaller overall size
    - requires efficient data structures (e.g. hashtable) to avoid expensive searching on lookup

Paging Summary

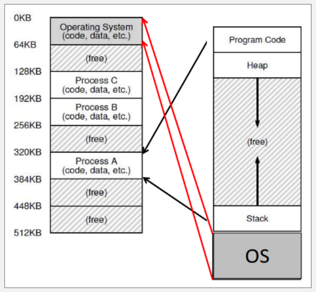
- the right choice of structure depends on the environment
  - time-space tradeoff
  - for software-managed TLBs, OS can use any data structure
    - e.g. inverted page tables with hashing
  - for hardware-managed TLBs, page table is determined by the hardware
    - e.g. x86 uses multi-level page tables
      - each page table fits within a page

Handling OS Memory

- the OS is not a separate process with its own address space
- in general, the MMU only allows access to memory via virtual addresses
  - can't bypass address translation

Handling OS Memory

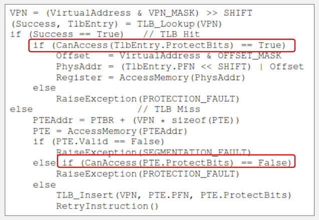
- OS code is part of the address space of every process
  - occupies high end of the process address space
  - compiler does not assign those addresses to user code
  - makes it easy to jump to OS code during a trap – trap handler is in the current address space
- there is only one copy of the OS code/data in physical memory
  - loaded into low addresses during system boot
  - page tables of all processes map their high virtual addresses to the same physical addresses



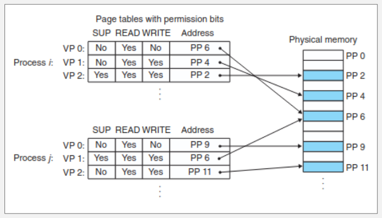
0KB
64KB
128KB
192KB
256KB
320KB
384KB
448KB
512KB
Operating System (code, data, etc.)
(free)
Process C (code, data, etc.)
Process B (code, data, etc.)
(free)
Process A (code, data, etc.)
(free)
(free)
Program Code
Heap
(free)
Stack
OS

Handling OS Memory

- to protect OS code/data –
  - PTEs include permission bits for page-level access control
    - read/write or read only (code pages are read only)
    - accessible in user or supervisor (kernel) mode
  - during address translation, permission bits are checked
    - CPU in user mode cannot access high virtual addresses
    - MMU traps to OS if an illegal access is attempted



```
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()
```



Process i:

	SUP	READ	WRITE	Address
VP 0:	No	Yes	No	PP 6
VP 1:	No	Yes	Yes	PP 4
VP 2:	Yes	Yes	Yes	PP 2
⋮				

Process j:

	SUP	READ	WRITE	Address
VP 0:	No	Yes	No	PP 9
VP 1:	Yes	Yes	Yes	PP 6
VP 2:	No	Yes	Yes	PP 11
⋮				