15-213

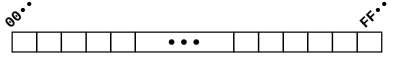
"The course that gives CMU its Zip!"

Virtual Memory
October 15, 2009

Topics
- Address spaces
- Motivations for virtual memory
- Address translation
- Accelerating translation with TLBs

`lecture-14.ppt`

Byte-Oriented Memory Organization



- **Programs Refer to Virtual Memory Addresses**
  - Conceptually very large array of bytes
  - Actually implemented with hierarchy of different memory types
  - System provides address space private to particular "process"
    - Program being executed
    - Program can clobber its own data, but not that of others
- **Compiler + Run-Time System Control Allocation**
  - Where different program objects should be stored
  - All allocation within single virtual address space

`From class02.ppt`

Simple Addressing Modes

- **Normal (R) Mem[Reg[R]]**
  - Register R specifies memory address

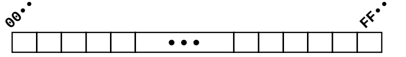
```
movl (%ecx),%eax
```

- **Displacement D(R) Mem[Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movl 8(%ebp),%edx
```

`From class04.ppt`

Lets think on this: physical memory?



How does everything fit?
- 32-bit addresses: ~4,000,000,000 (4 billion) bytes
- 64-bit addresses: ~16,000,000,000,000,000,000 (16 quintillion) bytes

How to decide which memory to use in your program?
- How about after a fork()?

What if another process stores data into your memory?
- How could you debug your program?

So, we add a level of indirection

One simple trick solves all three problems
- Each process gets its own private image of memory
  - appears to be a full-sized private memory range
- This fixes "how to choose" and "others shouldn't mess w/yours"
  - surprisingly, it also fixes "making everything fit"
- Implementation: translate addresses transparently
  - add a mapping function
    - to map private addresses to physical addresses
  - do the mapping on every load or store

This mapping trick is the heart of *virtual memory*

Address Spaces

A *linear address space* is an ordered set of contiguous nonnegative integer addresses:

{0, 1, 2, 3, … }

A *virtual address space* is a set of N = 2^n *virtual addresses*:

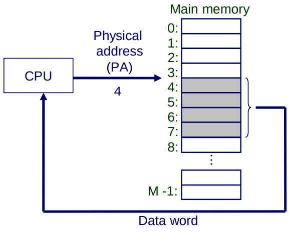
{0, 1, 2, …, N-1}

A *physical address space* is a set of M = 2^m (for convenience) *physical addresses*:

{0, 1, 2, …, M-1}

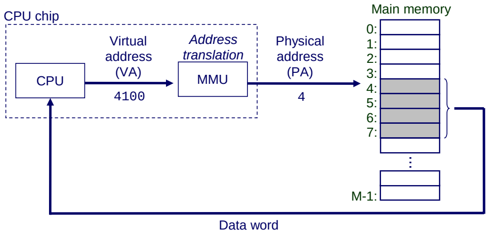
In a system based on virtual addressing, each byte of main memory has a physical address *and* a virtual address (or more)

A System Using Physical Addressing



Used by many embedded microcontrollers in devices like cars, elevators, and digital picture frames

A System Using Virtual Addressing



One of the great ideas in computer science

- **used by all modern desktop and laptop microprocessors**

Why Virtual Memory?

(1) VM allows efficient use of limited main memory (RAM)

- **Use RAM as a cache for the parts of a virtual address space**
  - **some non-cached parts stored on disk**
  - **some (unallocated) non-cached parts stored nowhere**
- **Keep only active areas of virtual address space in memory**
  - **transfer data back and forth as needed**

(2) VM simplifies memory management for programmers

- **Each process gets a full, private linear address space**

(3) VM isolates address spaces

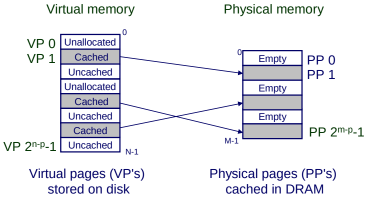
- **One process can't interfere with another's memory**
  - **because they operate in different address spaces**
- **User process cannot access privileged information**
  - **different sections of address spaces have different permissions**

(1) VM as a Tool for Caching

***Virtual memory* is an array of N contiguous bytes**

- **think of the array as being stored on disk**

The contents of the array on disk are cached in *physical memory (DRAM cache)*



DRAM Cache Organization

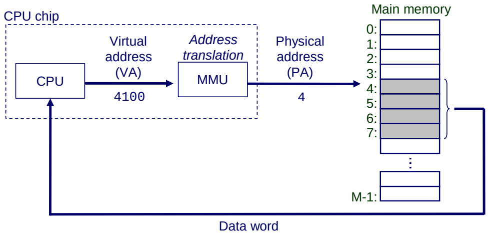
DRAM cache organization driven by the enormous miss penalty

- **DRAM is about 10x slower than SRAM**
- **Disk is about 100,000x slower than a DRAM**
  - **to get first byte, though fast for next byte**

DRAM cache properties

- **Large page (block) size (typically 4-8 KB)**
- **Fully associative**
  - **Any virtual page can be placed in any physical page**
  - **Requires a "large" mapping function – different from CPU caches**
- **Highly sophisticated replacement algorithms**
  - **Too complicated and open-ended to be implemented in hardware**
- **Write-back rather than write-through**

Reminder: using virtual addressing



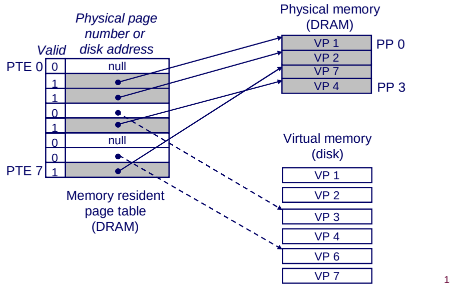
One of the great ideas in computer science

- **used by all modern desktop and laptop microprocessors**

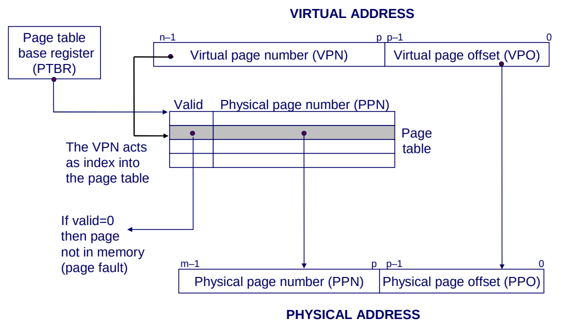
How? Page Tables

A *page table* is an array of page table entries (PTEs) that maps virtual pages to physical pages

- **Per-process kernel data structure in DRAM**

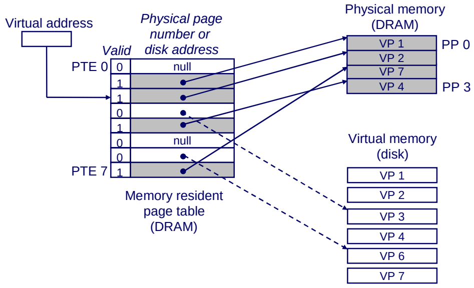


Address Translation with a Page Table



Page Hits

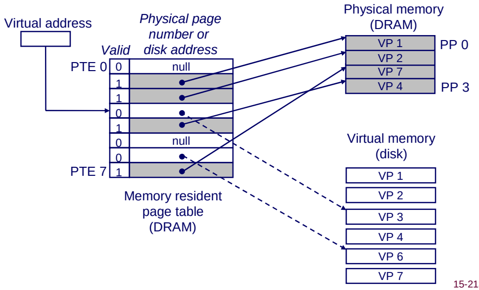
A *page hit* is a reference to a VM word that is in physical (main) memory



Page Faults

A *page fault* is caused by a reference to a VM word that is not in physical (main) memory

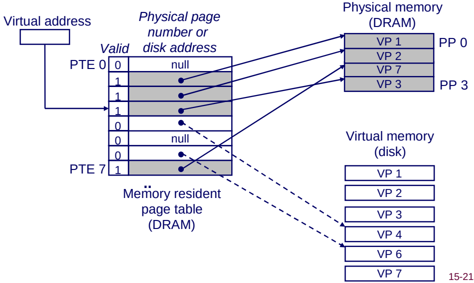
- **Example: An instruction references a word contained in VP 3, a miss that triggers a page fault exception**



Handling a Page Fault

The kernel's page fault handler selects VP 4 as the victim and replaces it with a copy of VP 3 from disk (*demand paging*)

- **When the offending instruction restarts, it executes normally, without generating an exception**



Why does it work? Locality

Virtual memory works because of locality

At any point in time, programs tend to access a set of active virtual pages called the *working set*

- **Programs with better temporal locality will have smaller working sets**

If (working set size < main memory size)

- **Good performance for one process after compulsory misses**

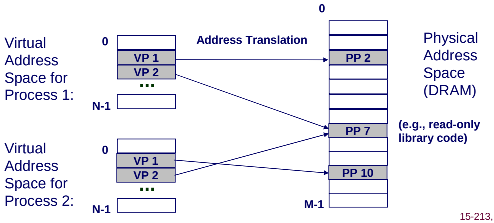
If (SUM(working set sizes) > main memory size)

- ***Thrashing:* Performance meltdown where pages are swapped (copied) in and out continuously**

(2) VM as a Tool for Memory Mgmt

Key idea: each process has its own virtual address space

- It can view memory as a simple linear array
- Mapping function scatters addresses through physical memory
  - Well chosen mappings simplify memory allocation and management



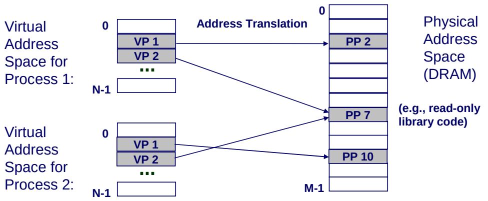
Simplifying Sharing and Allocation

Memory allocation

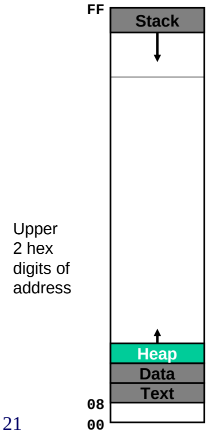
- Each virtual page can be mapped to any physical page
  - A virtual page can be stored in different physical pages at different times – the program never knows

Sharing code and data among processes

- Map virtual pages to the same physical page (PP 7)



IA32 Linux Memory Layout



- **Stack**
  - Runtime stack (8MB limit)
- **Heap**
  - Dynamically allocated storage
  - When call `malloc(), calloc(), new()`
- **Data**
  - Statically allocated data
  - E.g., arrays & strings declared in code
- **Text**
  - Executable machine instructions
  - Read-only

From `class08.ppt`

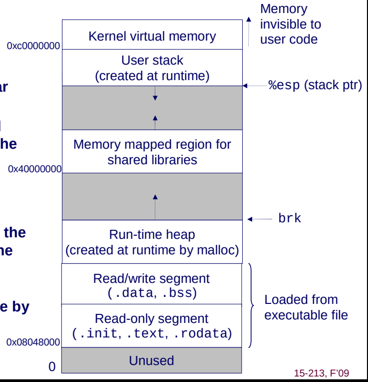
Simplifying Linking and Loading

Linking

- Each program has similar virtual address space
- Code, stack, and shared libraries always start at the same address

Loading

- `execve()` maps PTEs to the appropriate location in the executable binary file
- The `.text` and `.data` sections are copied, page by page, on demand by the virtual memory system.



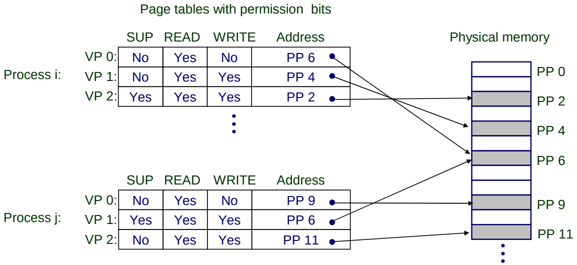
---

(3)VM as a Tool for Memory Protection

Extend PTEs with permission bits

Page fault handler checks these before remapping

- If violated, send process SIGSEGV (segmentation fault)



Page tables with permission bits

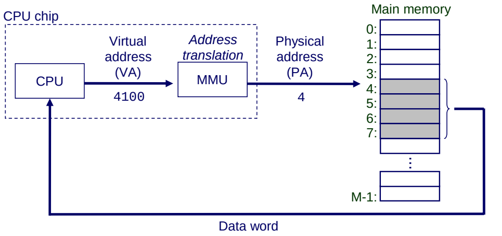
Process i:

	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

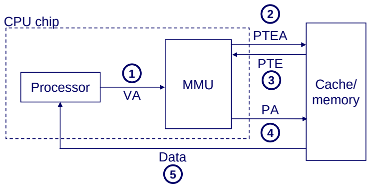
Reminder: using virtual addressing



One of the great ideas in computer science

- used by all modern desktop and laptop microprocessors

Address Translation: Page Hit



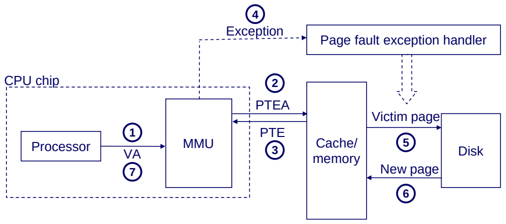
1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

4) MMU sends physical address to cache/memory

5) Cache/memory sends data word to processor

Address Translation: Page Fault



1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

4) Valid bit is zero, so MMU triggers page fault exception

5) Handler identifies victim (and, if dirty, pages it out to disk)

6) Handler pages in new page and updates PTE in memory

7) Handler returns to original process, restarting faulting instruction

---

Speeding up Translation with a TLB

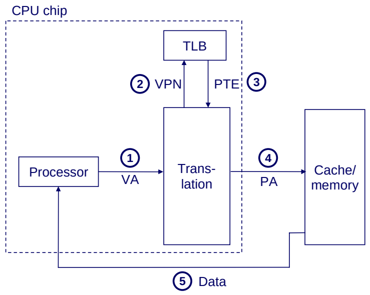
Page table entries (PTEs) are cached in L1 like any other memory word

- **PTEs may be evicted by other data references**
- **PTE hit still requires a 1-cycle delay**

Solution: *Translation Lookaside Buffer* (TLB)

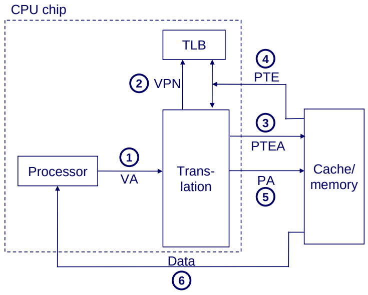
- **Small hardware cache in MMU**
- **Maps virtual page numbers to physical page numbers**
- **Contains complete page table entries for small number of pages**

TLB Hit



A TLB hit eliminates a memory access

TLB Miss



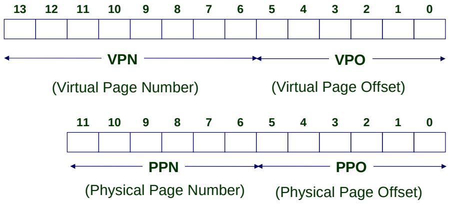
A TLB miss incurs an add'l memory access (the PTE)

- **Fortunately, TLB misses are rare**

Simple Memory System Example

Addressing

- **14-bit virtual addresses**
- **12-bit physical address**
- **Page size = 64 bytes**



Simple Memory System Page Table

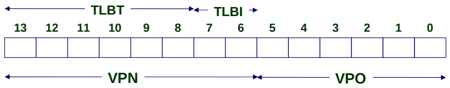
- Only show first 16 entries (out of 256)

VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	08	13	1
01	–	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	–	0
04	–	0	0C	–	0
05	16	1	0D	2D	1
06	–	0	0E	11	1
07	–	0	0F	0D	1

Simple Memory System TLB

TLB

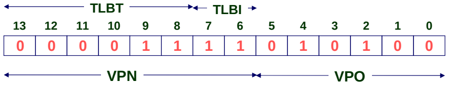
- 16 entries
- 4-way associative



Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Address Translation Example #1

Virtual Address `0x03D4`



13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0

VPN 0x0F TLBI 3 TLBT 0x03 TLB Hit? Y Page Fault? NO PPN: 0x0D

Summary

Programmer's View of Virtual Memory

- Each process has its own private linear address space
- Cannot be corrupted by other processes

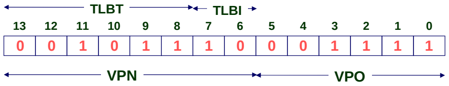
System View of Virtual Memory

- Uses memory efficiently by caching virtual memory pages
  - Efficient only because of locality
- Simplifies memory management and programming
- Simplifies protection by providing a convenient interpositioning point to check permissions

---

Address Translation Example #2

Virtual Address `0x0B8F`

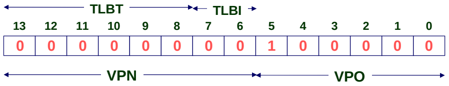


13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	0	0	0	1	1	1	1

VPN 0x2E TLBI 2 TLBT 0x0B TLB Hit? NO Page Fault? YES PPN: TBD

Address Translation Example #3

Virtual Address `0x0020`



13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0

VPN 0x00 TLBI 0 TLBT 0x00 TLB Hit? NO Page Fault? NO PPN: 0x28

Summary

Programmer's View of Virtual Memory

- Each process has its own private linear address space
- Cannot be corrupted by other processes

System View of Virtual Memory

- Uses memory efficiently by caching virtual memory pages
  - Efficient only because of locality
- Simplifies memory management and programming
- Simplifies protection by providing a convenient interpositioning point to check permissions

---

Simple Memory System Cache

Cache

- 16 lines
- 4-byte line size
- Direct mapped

CT ← bits 11–6 →, CI ← bits 5–2 →, CO ← bits 1–0 →

11	10	9	8	7	6	5	4	3	2	1	0

PPN ← bits 11–6 →, PPO ← bits 5–0 →

Idx	Tag	Valid	B0	B1	B2	B3	Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11	8	24	1	3A	00	51	89
1	15	0	–	–	–	–	9	2D	0	–	–	–	–
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	–	–	–	–	B	0B	0	–	–	–	–
4	32	1	43	6D	8F	09	C	12	0	–	–	–	–
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	–	–	–	–	E	13	1	83	77	1B	D3
7	16	1	11	C2	DF	03	F	14	0	–	–	–	–

Address Translation Example #1

Virtual Address `0x03D4`

TLBT ← bits 13–8 →, TLBI ← bits 7–6 →

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0

VPN ← bits 13–6 →, VPO ← bits 5–0 →

VPN 0x0F TLBI 3 TLBT 0x03 TLB Hit? Y Page Fault? NO PPN: 0x0D

Physical Address

CT ← bits 11–6 →, CI ← bits 5–2 →, CO ← bits 1–0 →

11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	1	0	1	0	0

PPN ← bits 11–6 →, PPO ← bits 5–0 →

Offset 0 CI 0x5 CT 0x0D Hit? Y Byte: 0x36

Address Translation Example #2

Virtual Address `0x0B8F`

TLBT ← bits 13–8 →, TLBI ← bits 7–6 →

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	0	0	0	1	1	1	1

VPN ← bits 13–6 →, VPO ← bits 5–0 →

VPN 0x2E TLBI 2 TLBT 0x0B TLB Hit? NO Page Fault? YES PPN: TBD

Physical Address

CT ← bits 11–6 →, CI ← bits 5–2 →, CO ← bits 1–0 →

11	10	9	8	7	6	5	4	3	2	1	0

PPN ← bits 11–6 →, PPO ← bits 5–0 →

Offset ___ CI___ CT ____ Hit? __ Byte: ____

Address Translation Example #3

Virtual Address `0x0020`

TLBT ← bits 13–8 →, TLBI ← bits 7–6 →

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0

VPN ← bits 13–6 →, VPO ← bits 5–0 →

VPN 0x00 TLBI 0 TLBT 0x00 TLB Hit? NO Page Fault? NO PPN: 0x28

Physical Address

CT ← bits 11–6 →, CI ← bits 5–2 →, CO ← bits 1–0 →

11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	1	0	0	0	0	0

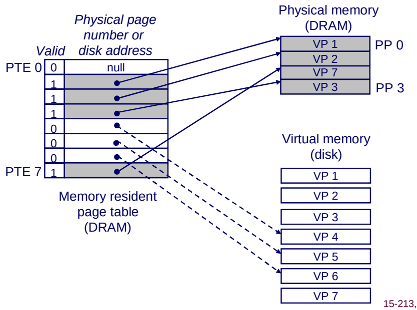
PPN ← bits 11–6 →, PPO ← bits 5–0 →

Offset 0 CI 0x8 CT 0x28 Hit? NO Byte: MEM

Allocating Virtual Pages

Example: Allocating new virtual page VP5

- Kernel allocates VP 5 on disk and points PTE 5 to it



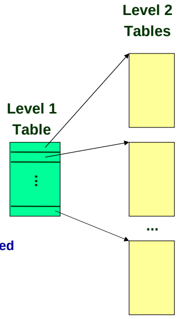
Multi-Level Page Tables

Given:
- 4KB (2^{12}) page size
- 48-bit address space
- 4-byte PTE

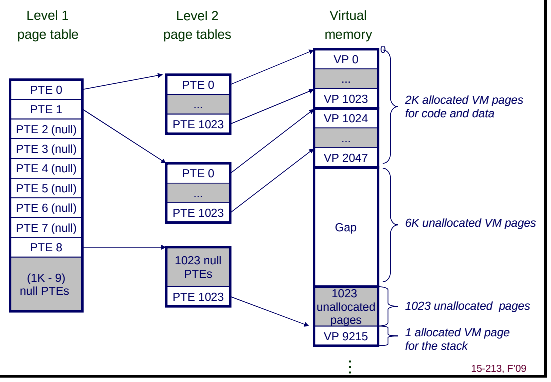
Problem:
- Would need a 256 GB page table!
  - $2^{48} * 2^{-12} * 2^{2} = 2^{38}$ bytes

Common solution
- Multi-level page tables
- Example: 2-level page table
  - Level 1 table: each PTE points to a page table (memory resident)
  - Level 2 table: Each PTE points to a page (paged in and out like other data)

- Level 1 table stays in memory
- Level 2 tables paged in and out



A Two-Level Page Table Hierarchy



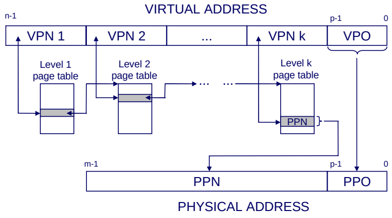
2K allocated VM pages for code and data

6K unallocated VM pages

1023 unallocated pages

1 allocated VM page for the stack

Translating with a k-level Page Table



Servicing a Page Fault

(1) Processor signals disk controller
- Read block of length P starting at disk address X and store starting at memory address Y

(2) Read occurs
- Direct Memory Access (DMA)
- Under control of I/O controller

(3) Controller signals completion
- Interrupts processor
- OS resumes suspended process