ECE 454
Computer Systems Programming

Virtual Memory and Prefetching

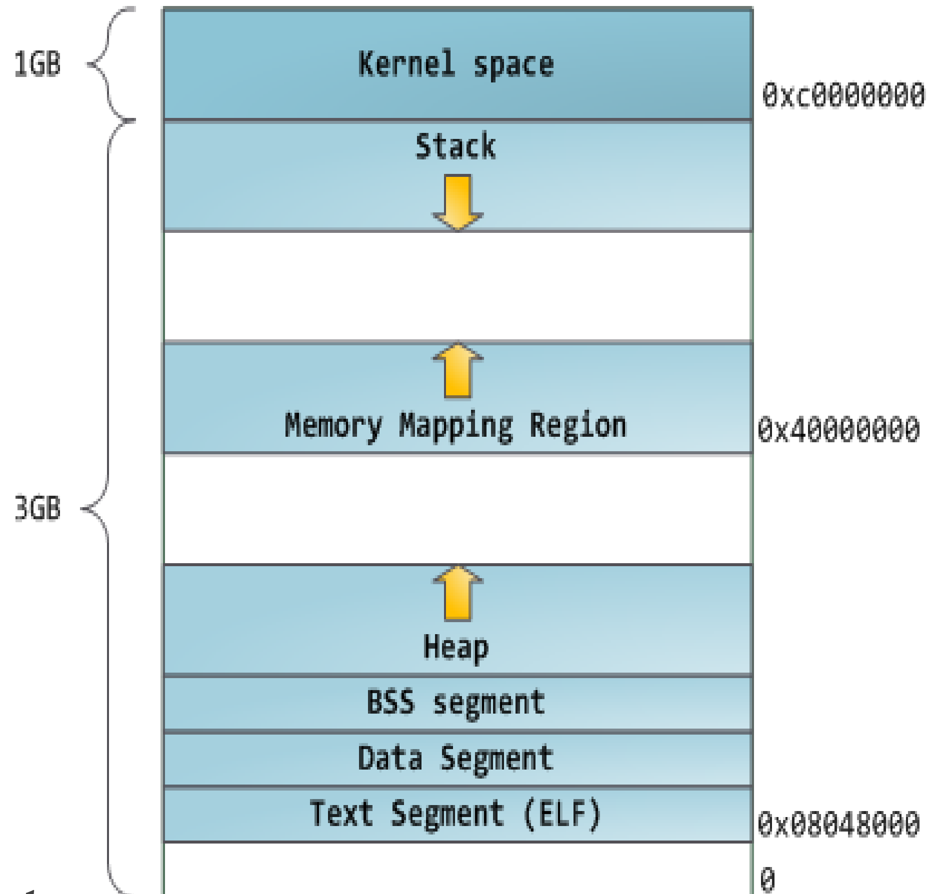
Ashvin Goel, Ding Yuan
ECE Dept, University of Toronto

Contents

- Virtual Memory (review hopefully)
- Prefetching

IA32 Linux Memory Layout

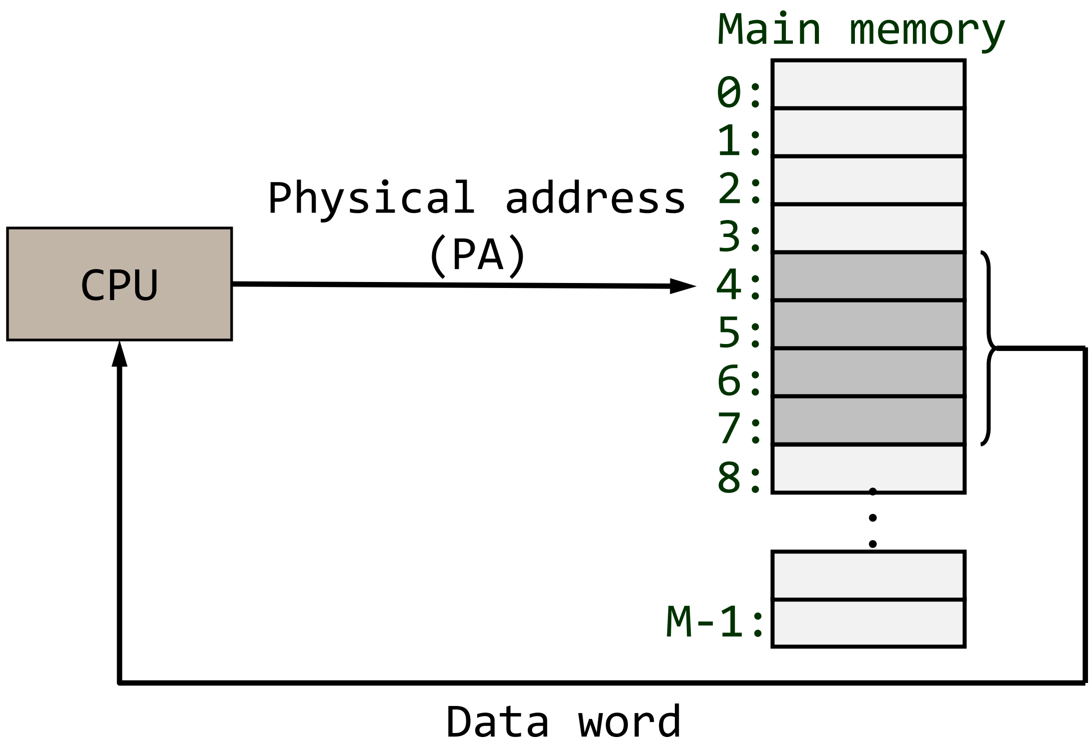
- Text
  - Executable instructions
  - Read-only
- Data
  - Statically allocated data
  - E.g., arrays & strings in code
- Heap
  - Dynamically allocated storage
  - `malloc(), calloc(), new()`
- Stack
  - Local variables, parameters, return values



Virtual Memory

- Programs access data and instructions using virtual memory addresses
  - Conceptually very large array of bytes
    - Each byte has its own address
    - 4GB for 32 bit architectures, 16EB (exabytes) for 64 bit architectures
  - System provides private address space to each process
- Memory allocation
  - Need to decide where different program objects should be stored
  - Performed by a combination of compiler and run-time system
- But why virtual memory? Why not physical memory?

A System Using Physical Addressing

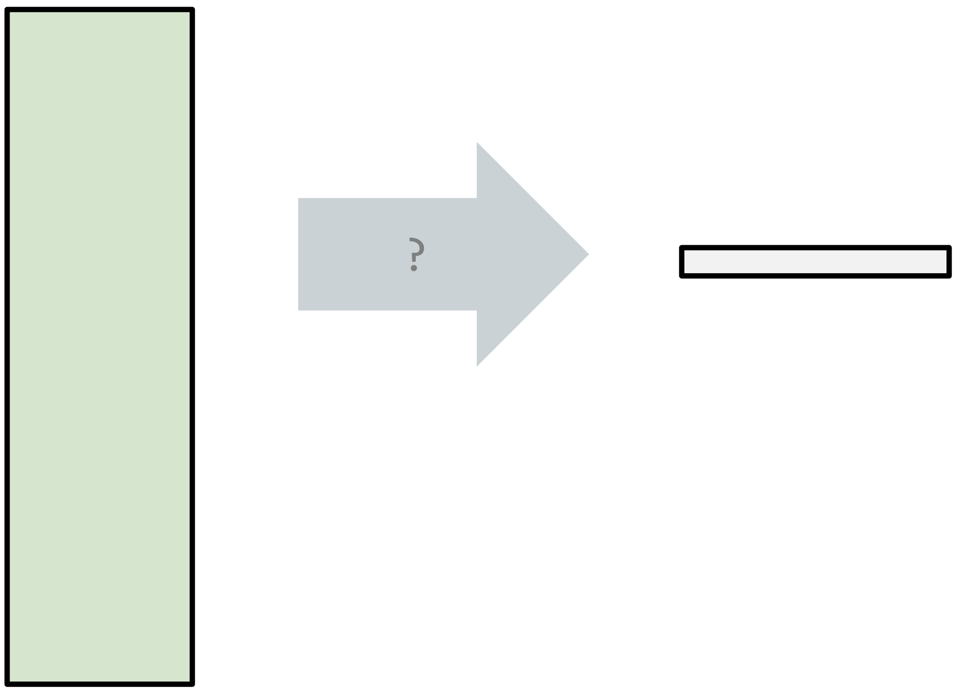


- Used in “simple” embedded microcontrollers
- Introduces several problems for larger, multi-process systems

Problem 1: How Does Everything Fit?

64-bit addresses:
16 Exabytes (16M TB)

Physical main memory
in BIG server: 1TB



And there are many processes ….

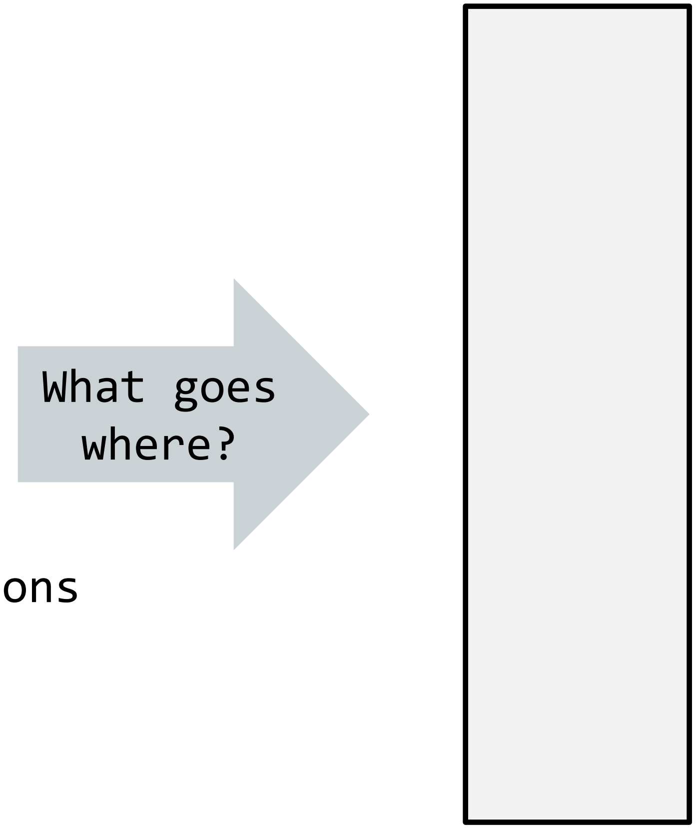
Problem 2: Memory Management

Physical main memory

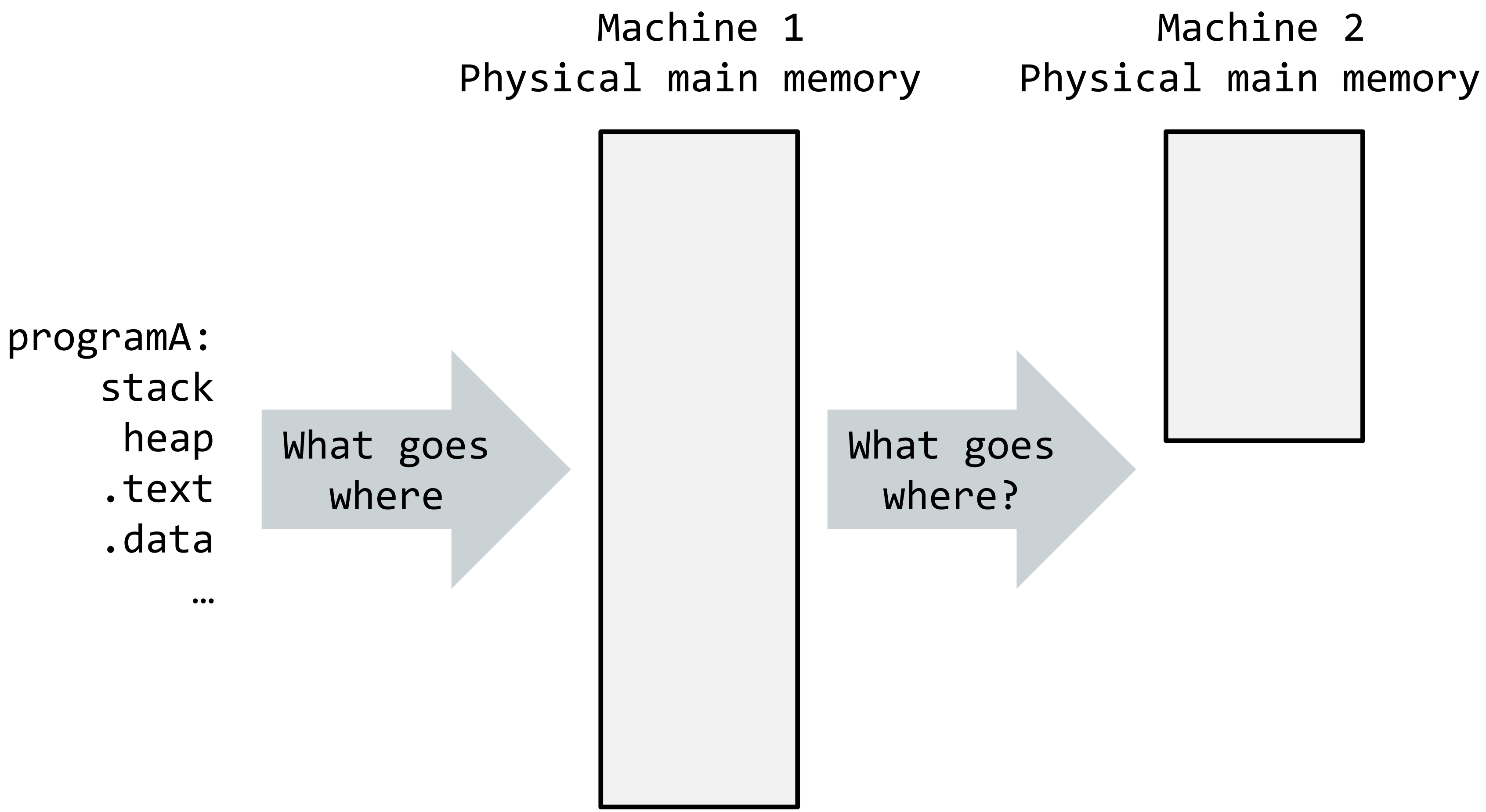
Process 1
Process 2
Process 3
…
Process n

x

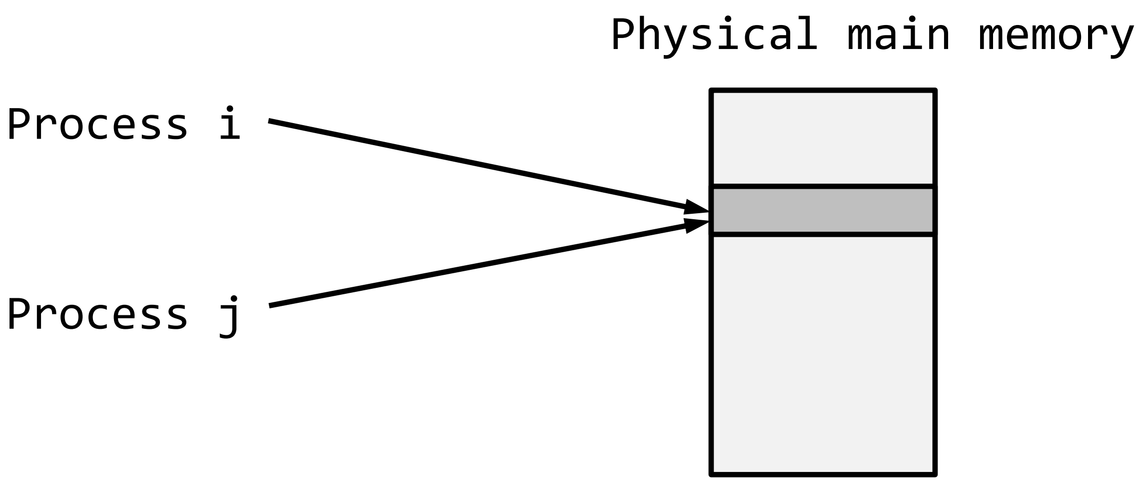
text
data
heap
stack
…
mapped regions



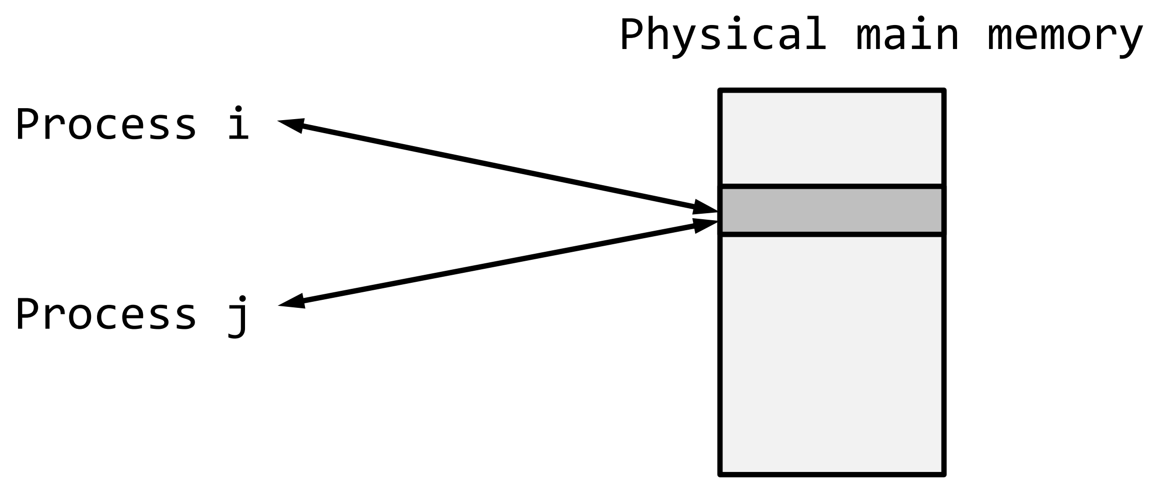
Problem 3: Portability



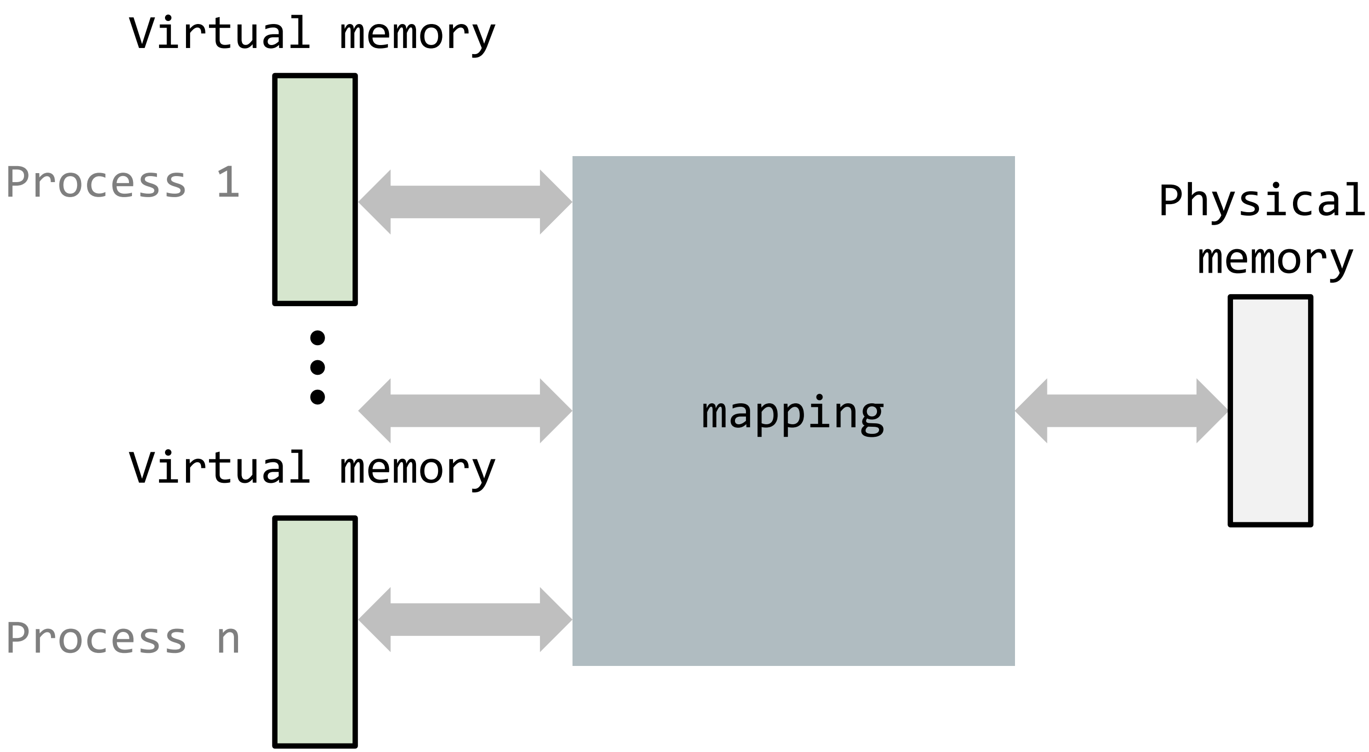
Problem 4: Protection



Problem 4: Sharing

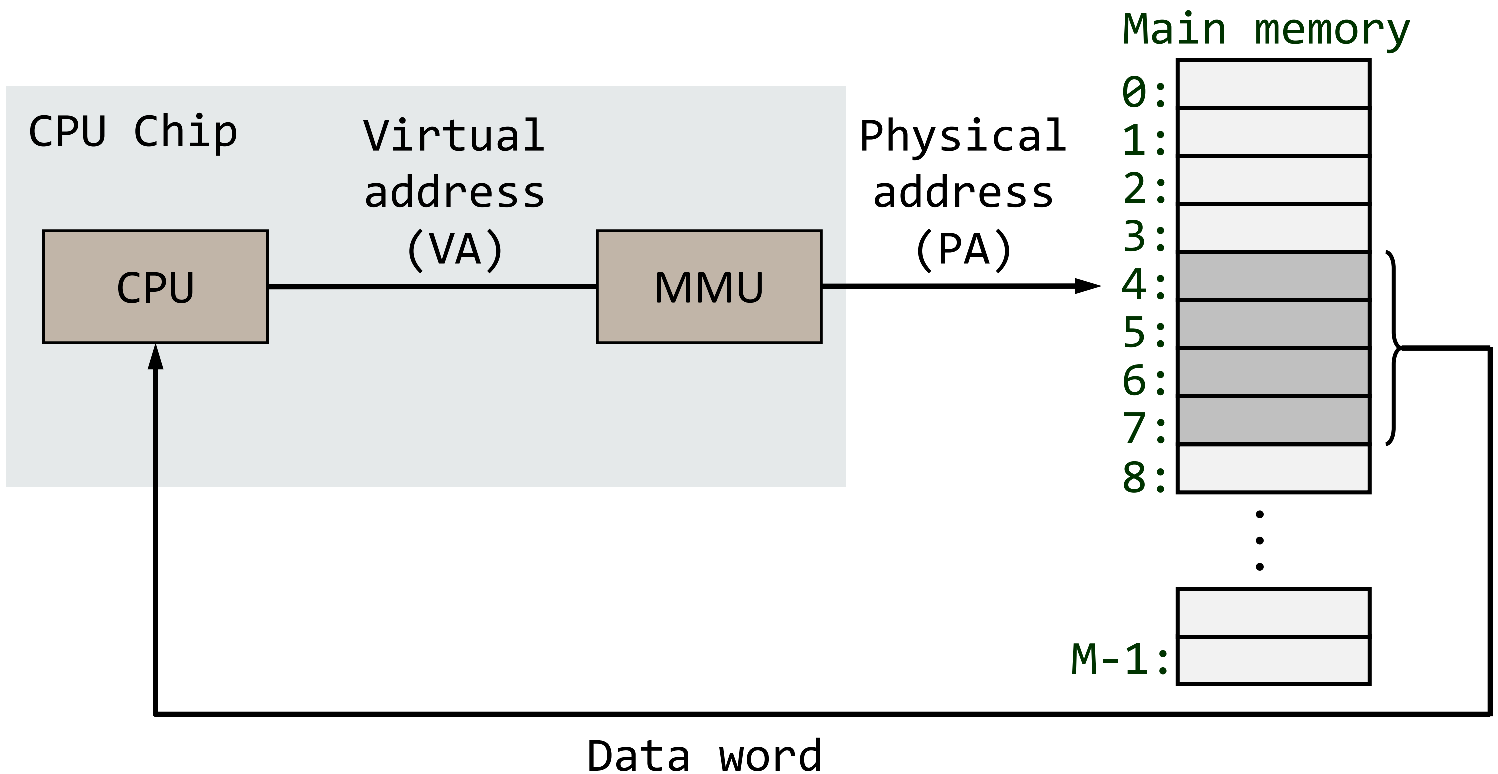


Solution: Add a Level Of Indirection



- Each process gets its own private memory space
- Solves all the previous problems

A System Using Virtual Addressing



- MMU = Memory Management Unit

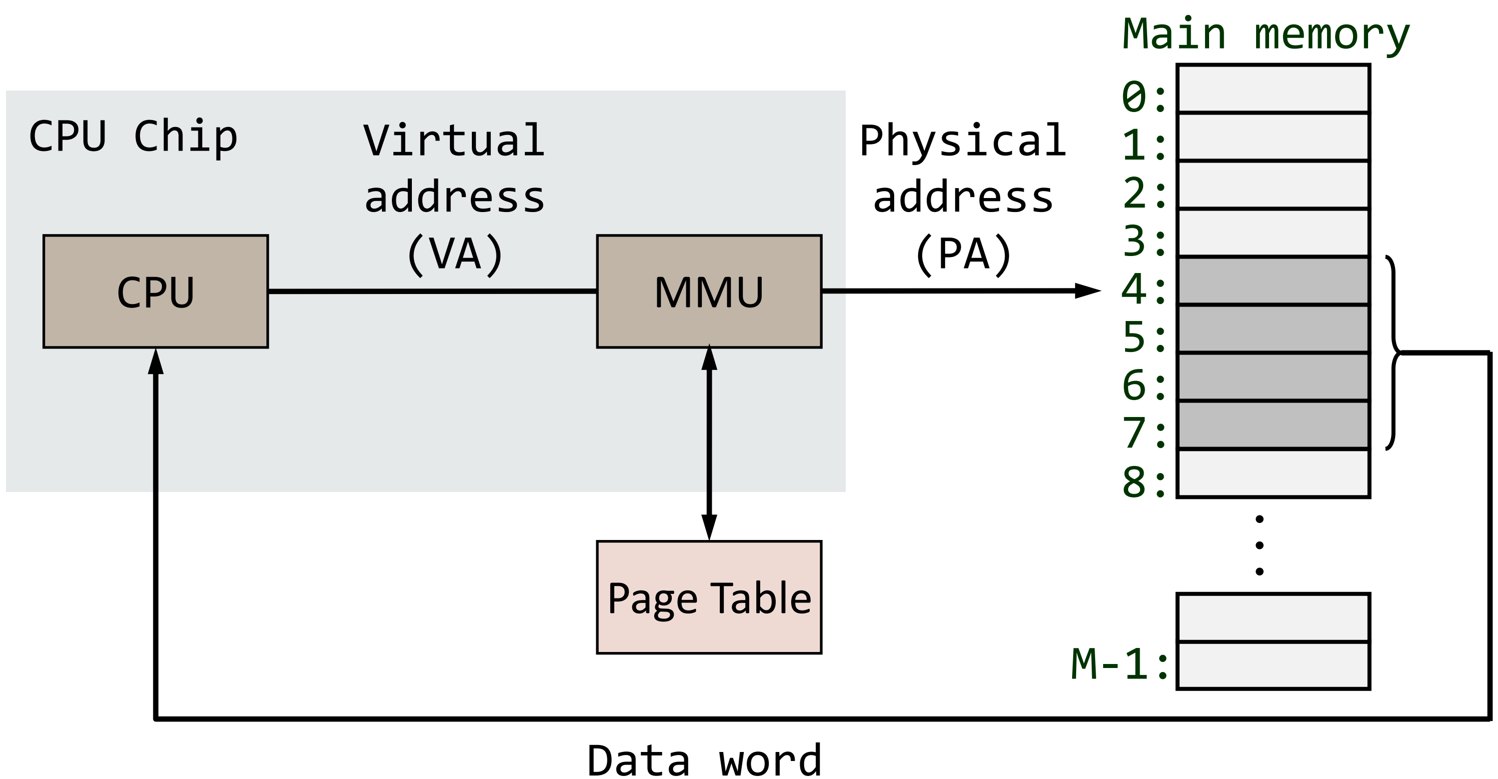
Virtual Memory Turns Main Memory into a Cache

- However, miss penalty is large:
  - DRAM latency: ~100ns
  - Disk latency: 10ms, 100,000x slower than DRAM
  - SATA SSD latency: 70us, 700x slower than DRAM
  - Recent Intel Optane NVMe SSD latency: 2.8us, 28x slower
    - For 4KB read

DRAM-Cache Design

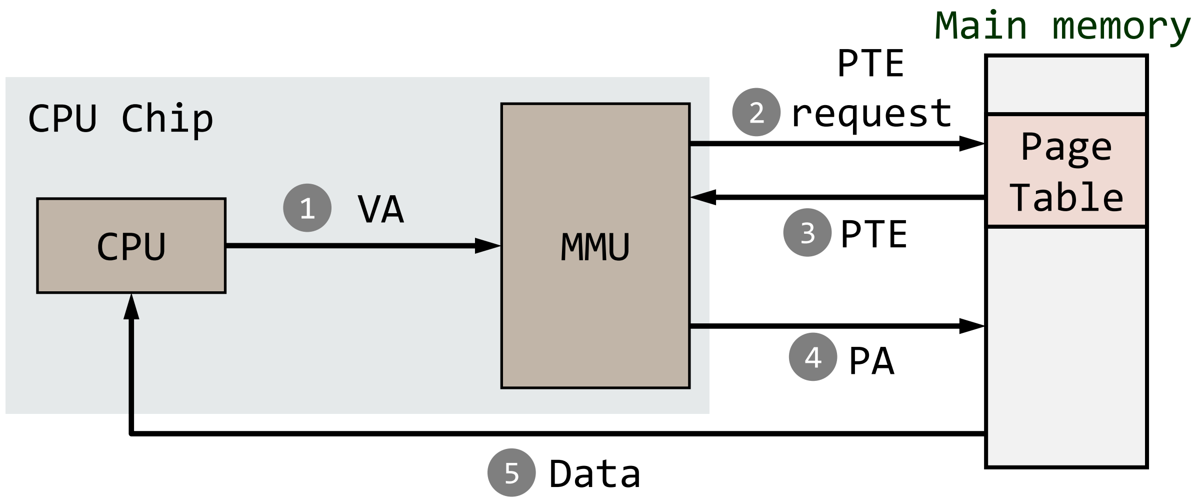
- Use paging
  - Map virtual to physical address at fixed-size page granularity
  - Use large page size to reduce mapping information
  - Typically, 4KB page size, matches disk block access granularity
- Use fully associative cache
  - Any virtual page can be mapped to any physical frame
  - Needs sophisticated mapping function
    - E.g., multi-level page table for the mapping information
  - Needs sophisticated replacement algorithms
    - E.g., LRU, clock/second chance
- Use write-back rather than write-through

MMU Needs A Large Table of Translations



- MMU keeps mapping of VAs -> PAs in page tables

Page Table is Stored in Memory



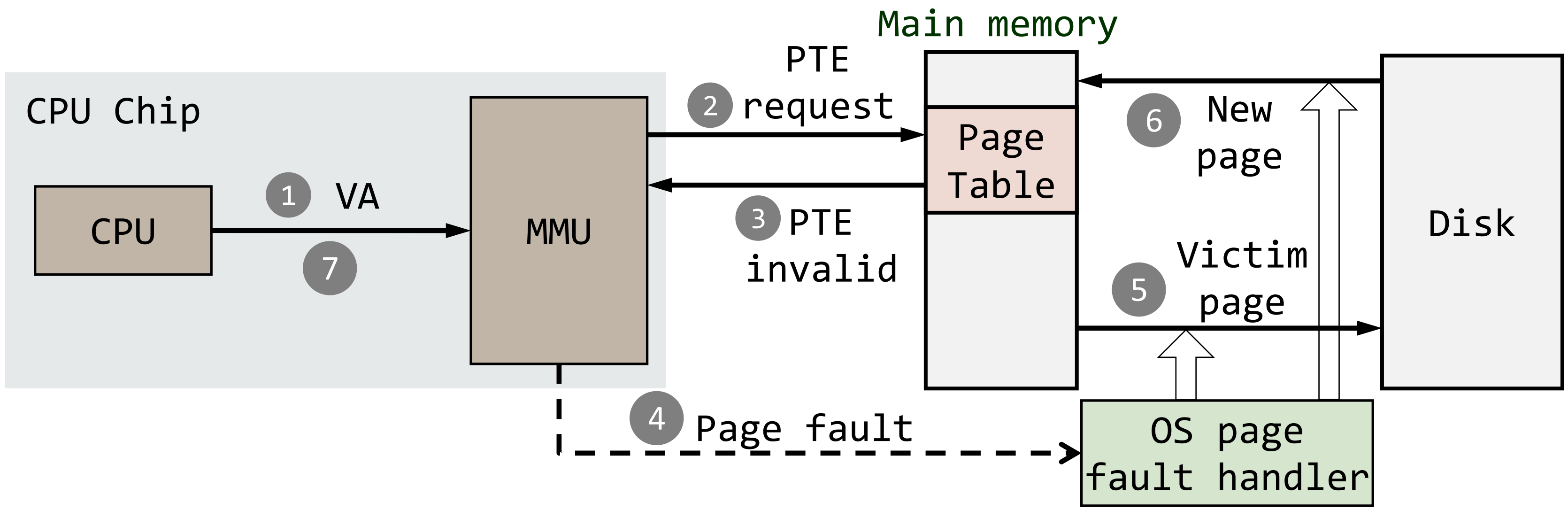
1) Processor sends virtual address (VA) to MMU

2-3) MMU requests page table entry (PTE) from page table

4) MMU sends physical address (PA) to cache/memory

5) Cache/memory sends data word to processor

Page Not Mapped in Physical Memory



1) Processor sends virtual address (VA) to MMU

2-3) MMU requests PTE from page table, but PTE is invalid

4) PTE absent so MMU triggers page fault exception

5) Handler chooses victim page (and, if page is dirty, pages it out to disk)

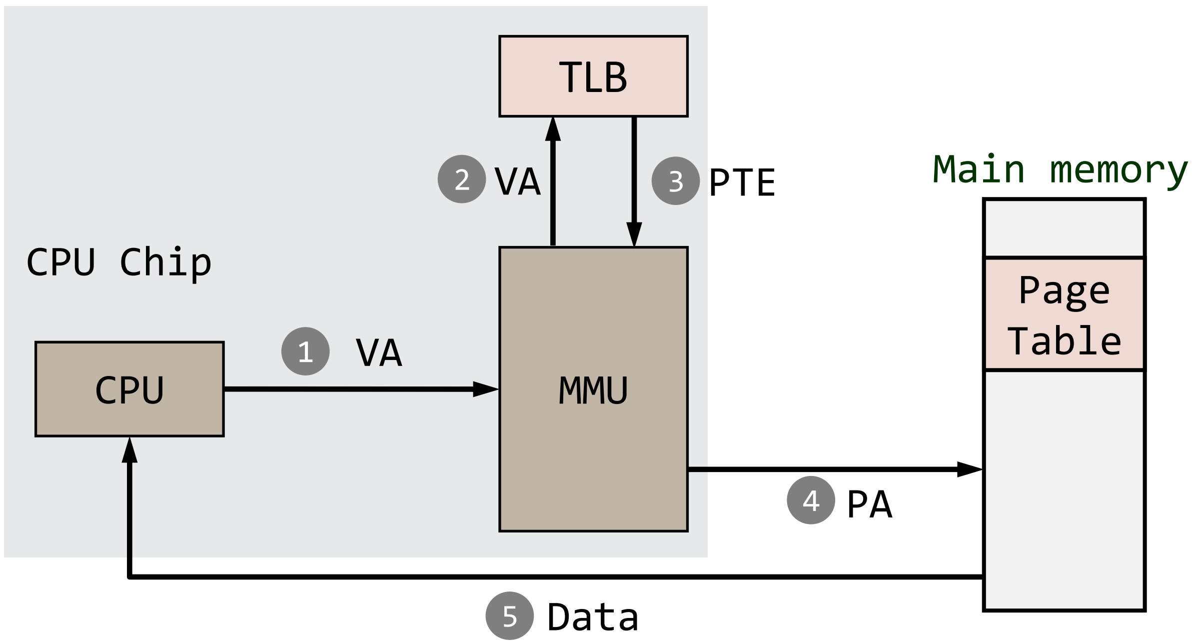
6) Handler pages in new page and updates PTE in memory

7) Handler returns to original process, restarting faulting instruction

Speeding up Translation with a TLB

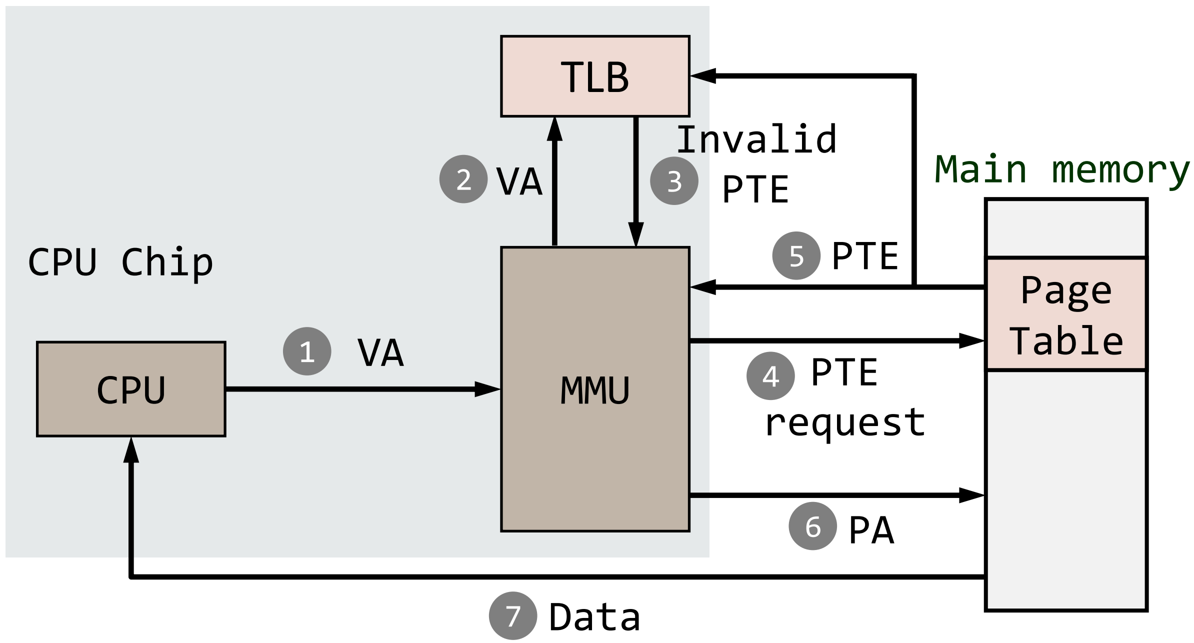
- Page table entries (PTEs) are cached in L1 (like any other memory word)
  - But PTEs may be evicted by other data references
  - Even on a cache hit, PTE access requires a 1-cycle delay
  - Doubles the cost of accessing physical addresses from cache
- Solution: Translation Lookaside Buffer (TLB)
  - Small hardware cache in MMU
  - Caches PTEs for a small number of pages (e.g., 256 entries)

TLB Hit



- A TLB hit avoids access to page table in memory

TLB Miss



- A TLB miss incurs additional memory access to read PTE

How to Program for Virtual Memory

- Programs tend to access a set of active virtual pages at any point in time called the working set
- Programs with better locality will have smaller working sets
- If (working set size) > main mem size:
  - Pages are swapped (copied) in and out continuously
    - Called thrashing, leads to performance meltdown
- If (# working set pages) > # TLB entries:
  - TLB misses occur
  - Not as bad as page thrashing, but still worth avoiding

More on TLBs

- Assume a 256-entry TLB, 4kB pages
  - 256*4kB = 1MB: can only have TLB hits for 1MB of data
  - This is called the TLB reach, i.e., amount of memory TLB covers
- Typical L2 cache is 8MB
  - Hence can't have TLB hits for all L2
  - Possibly consider TLB-size before L2 size
- Real CPUs have second-level TLBs
  - This is getting complicated to reason about!
  - Need to experiment with varying sizes, e.g., find best tile size

Prefetching

Prefetching

ORIGINAL CODE:

```
inst1
inst2
inst3
inst4
load X (misses cache)
   |
   |  Cache miss latency
   v
inst5 (must wait for load value)
inst6
```

CODE WITH PREFETCHING:

```
inst1
prefetch X
inst2  |
inst3  |  Cache miss latency
inst4  v
load  X (hits cache)
inst5 (load value is ready)
inst6
```

- Basic idea:
  - Predict data that might be needed soon (might be wrong)
  - Initiate an early request for that data (a load-to-cache)
  - If effective, helps tolerate latency to memory

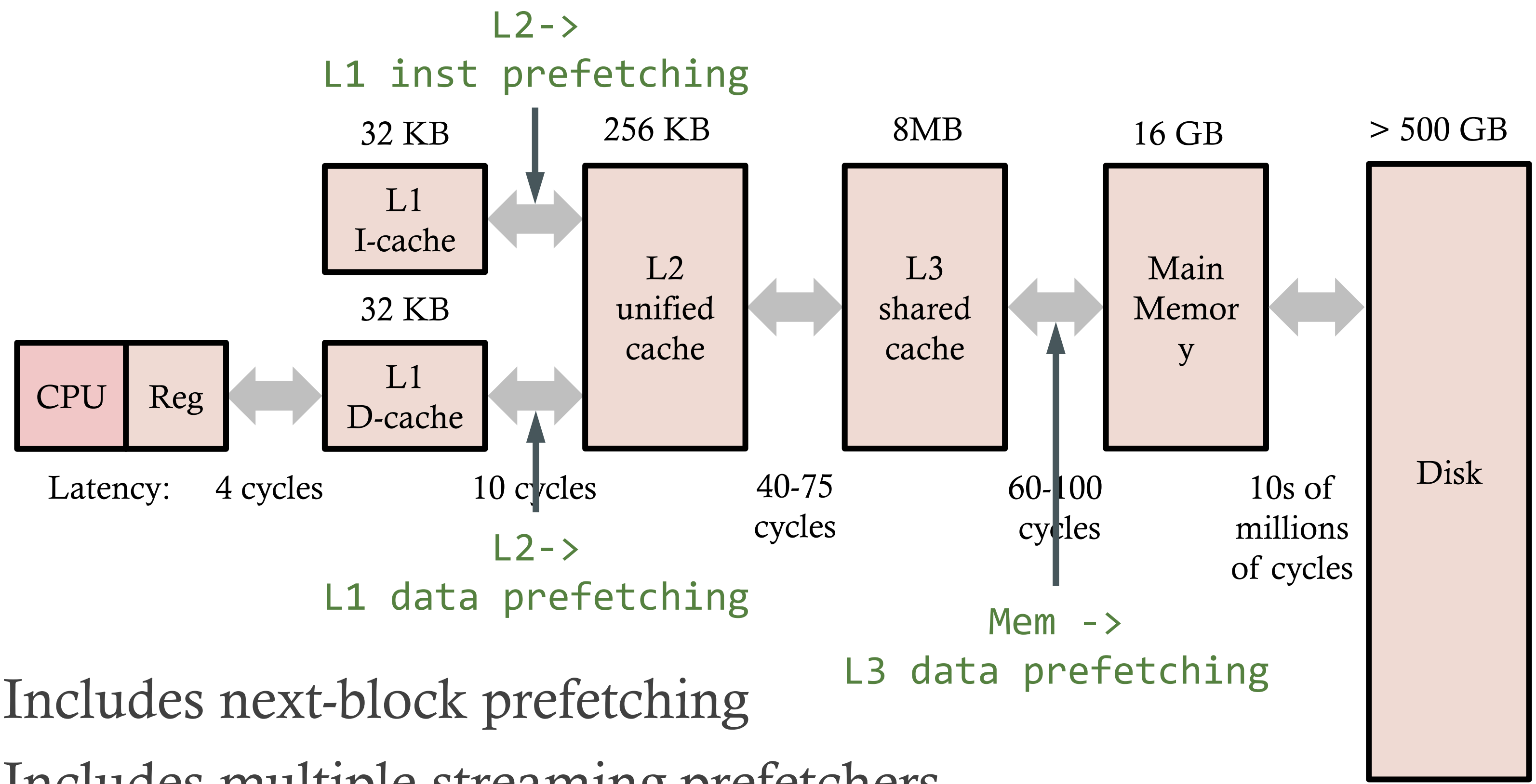
Prefetching is Difficult

- Prefetching is effective only if all of these are true:
  - There is spare memory bandwidth
    - Otherwise prefetching can cause bandwidth bottleneck
  - Prefetching is accurate
    - Only useful if the prefetched data will be used soon
  - Prefetching is timely
    - I.e., prefetch the right data, but not enough in advance
  - Prefetched data doesn't displace other in-use data
    - E.g., prefetched data should not replace a cache block about to be used
  - Latency hidden by prefetching outweighs its cost
    - Cost of lots of useless prefetched data can be significant
- Ineffective prefetching can hurt performance!

Hardware Prefetching

- A simple hardware prefetcher:
  - When one cache block is accessed, prefetch the adjacent block
  - I.e., behaves like cache blocks are twice as big
  - Helps with unaligned instructions (even when data is aligned)
- A more complex hardware prefetcher:
  - Can recognize a “stream”: addresses separated by a “stride”
  - Eg1: 0x1, 0x2, 0x3, 0x4, 0x5, 0x6... (stride = 0x1)
  - Eg2: 0x100, 0x300, 0x500, 0x700, 0x900… (stride = 0x200)
  - Prefetch predicted future addresses
    - Eg., cur_addr + stride, cur_addr + 2*stride, cur_addr + 3*stride, …

Core 7 Hardware Prefetching



- Includes next-block prefetching
- Includes multiple streaming prefetchers
- Prefetching performed within a page boundary
- Details are kept vague/secret

Software Prefetching

- Hardware provides special prefetch instructions:
  - Eg., intel's prefetchnta instruction, __mm_prefetch() intrinsic
- Compiler or programmer can insert them in code
  - Can PF (non-strided) patterns that hardware wouldn't recognize

```
void
process_list(list_t *head){
   list_t *p = head;
   while (p){
     process(p);
     p = p->next;
   }
}
```

Assume `process()` runs long enough to hide prefetch latency

```
void
process_list_PF(list_t *head){
   list_t *p = head;
   list_t *q;

   while (p){
     q = p->next;
     prefetch(q);
     process(p);
     p = q;
   }
}
```

Summary: Optimizing for a Modern Memory Hierarchy

Memory Optimization: Summary

- Caches
  - Conflict misses: less of a concern due to high-associativity
    - Modern CPUs have 8-way L1/L2, 16-way L3
  - Cache capacity: keep working set within on-chip cache capacity
    - Focus on either L1 or L2 depending on required working-set size
- Virtual memory
  - Page Misses: keep working set within main memory capacity
  - TLB Misses: keep working set #pages < TLB #entries
- Prefetching
  - Arrange data structures so access patterns are sequential/strided
  - Use compiler or manually-inserted prefetch instructions