Memory III

CSCI 2050U - Computer Architecture

Randy J. Fortier
@randy_fortier

Ontario Tech University

Outline

- Caching
- Virtual memory

Caching

CSCI 2050U - Computer Architecture

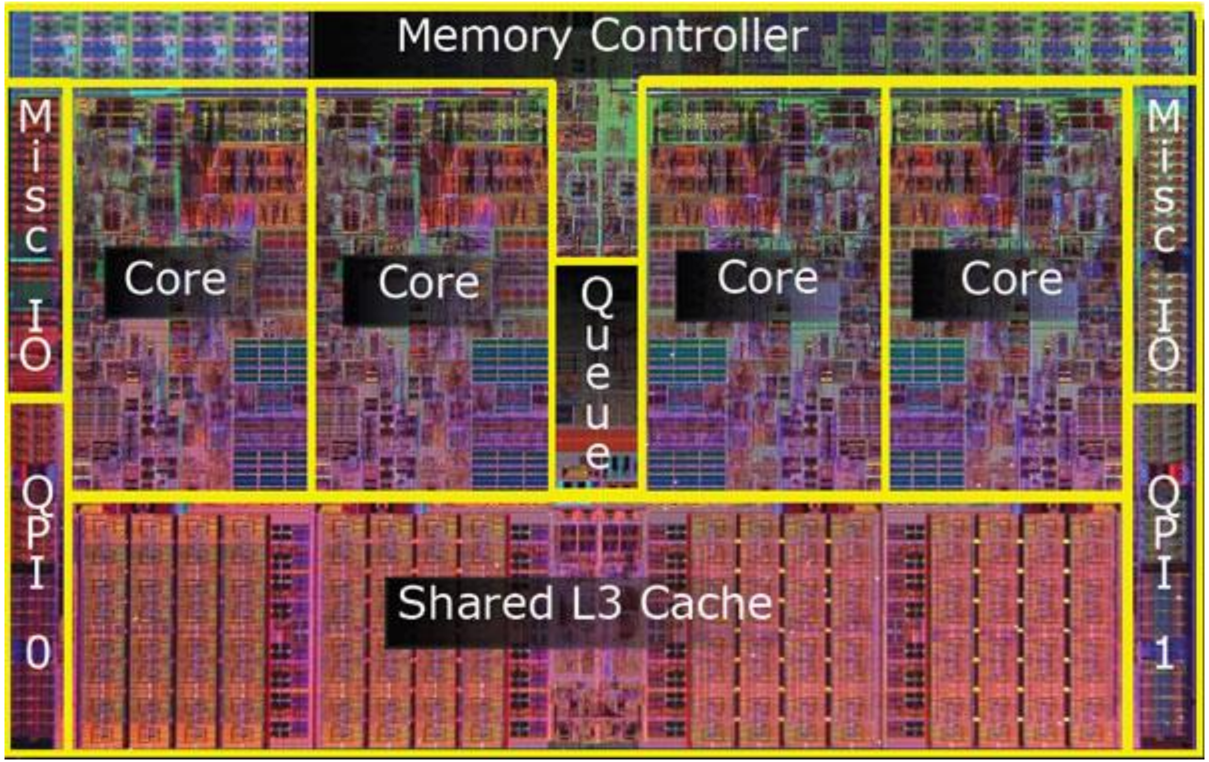
Locality

- Principle of locality: if we need a datum, other data nearby are likely to also be required
    - *Temporal locality*: Recently-used data may be needed again
    - *Spatial locality*: Accesses tend to be clustered in similar memory locations (addresses)
        - *Sequential locality*: Instructions and data are often accessed linearly
            - e.g. arrays, linear code blocks

Caching

- For simplicity, we often focus on registers and RAM
  - Frequently used data will be placed into registers
  - Less frequently used data will be placed into RAM
- Caching considers the possibility that there may be data in between
  - Data that is frequently transferred from RAM, but numerous enough to make registers impractical
- Caching types:
  - Explicit
  - Implicit

Caching

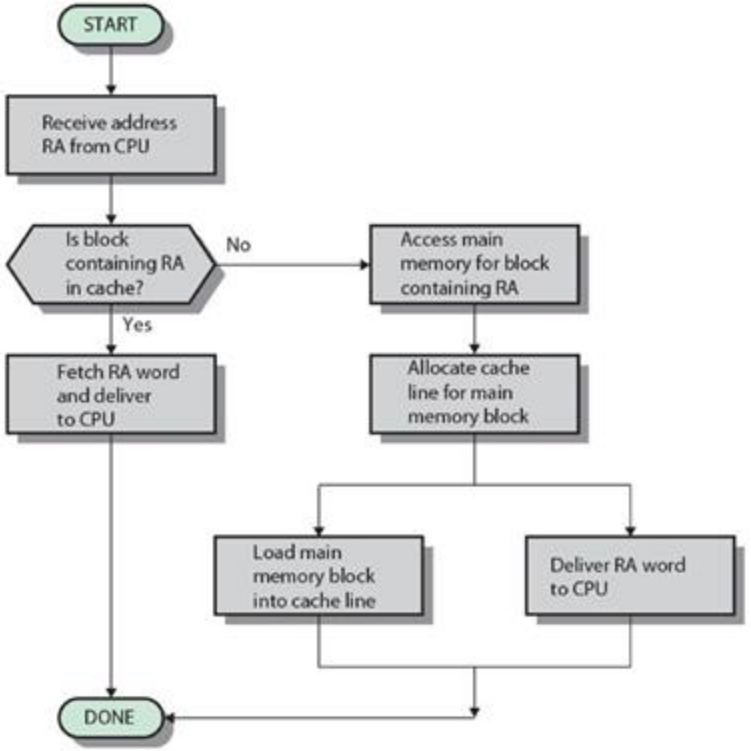


Memory Controller
Misc IO
QPI 0
Core
Core
Queue
Core
Core
Misc IO
QPI 1
Shared L3 Cache

Caching Basics

- A cache is not explicitly used by a programmer
  - A programmer requests data from RAM
  - If the data is in the cache (hit), the request is intercepted and the cached data is returned
  - If the data is not in the cache (miss), when the data (from RAM) makes its way back, it may be added to the cache
- To the programmer, these two situations are *functionally* identical
  - However, the performance is not identical

Caching Basics

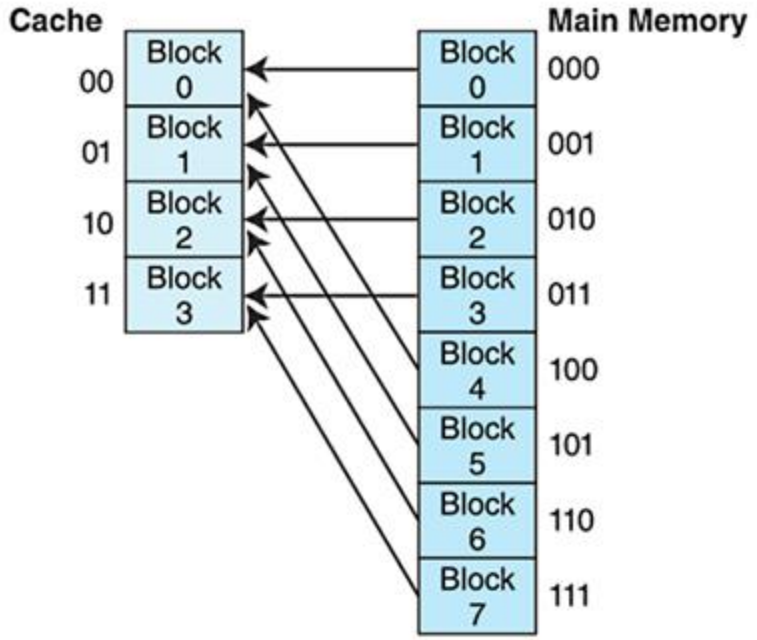

START
Receive address RA from CPU
Is block containing RA in cache?
No
Yes
Access main memory for block containing RA
Fetch RA word and deliver to CPU
Allocate cache line for main memory block
Load main memory block into cache line
Deliver RA word to CPU
DONE

Caching

- L1 cache
  - Located inside the processor cores
  - ~32Kb per core for data, ~32Kb per core for instructions
- L2 cache
  - Usually, located inside the processor cores
  - Larger than L1 (~1MB per core, data only), but slower/farther away
- L3 cache
  - Located on the SoC, but not inside the cores
  - Much larger than L2 (~15MB shared, data only)

Direct-Mapped Caches

- $A_{cache} = A_{main} \bmod N$



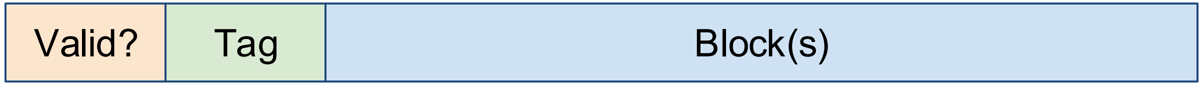
Direct-Mapped Caches

- The cache entries in a direct-mapped cache will contain the following:
  - Offset: The cache address, as computed by the modulo formula
  - Tag: The rest of the RAM address
  - Block: The actual data
  - Valid?: Does this block contain valid data?

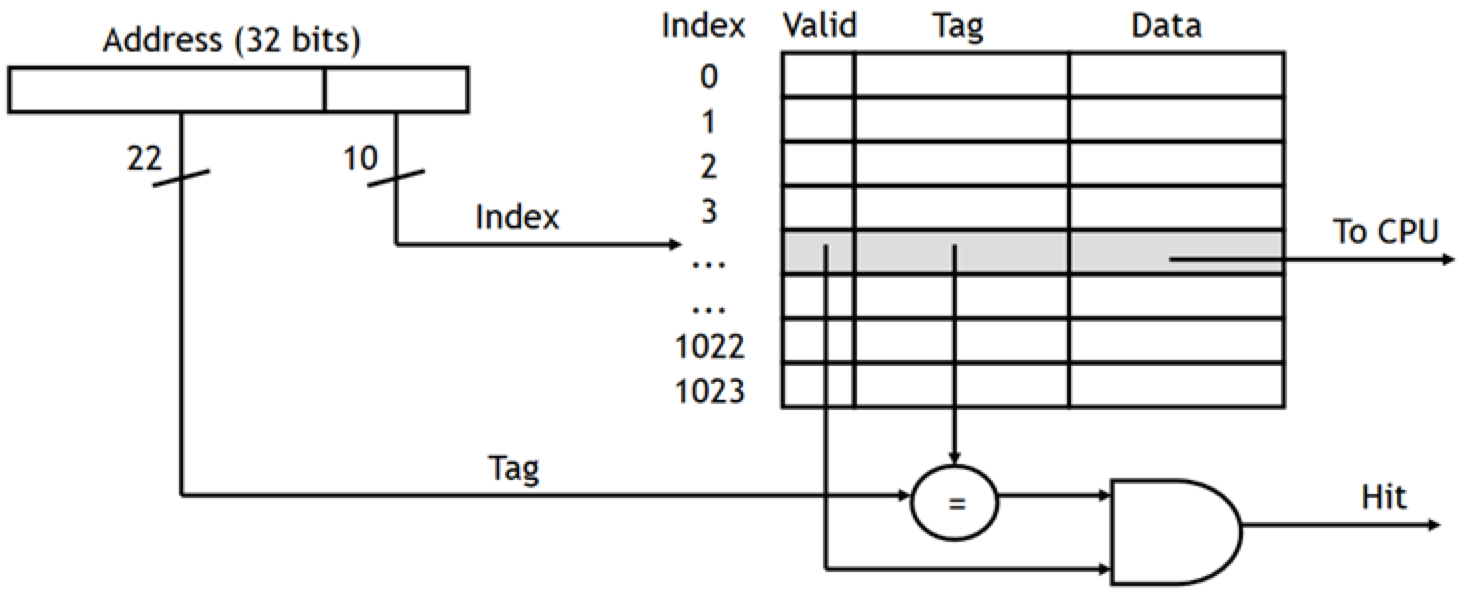
Valid?	Tag	Offset	Block(s)

Direct-Mapped Caches

- The cache entries in a direct-mapped cache will contain the following:
  - Offset: The cache address, as computed by the modulo formula
  - Tag: The rest of the RAM address
  - Block: The actual data
  - Valid?: Does this block contain valid data?



Direct-Mapped Caches



Thrashing

- Imagine writing a program to add the corresponding values from two separate arrays of integers:
  - `list1`: Located at address `000`
  - `list2`: Located at address `100`

RAM:

000	24
001	15
010	-7
011	31
100	-20
101	16
110	-4
111	0

Thrashing

```
n = 0
while n <= 3:
    x = list1[n]
    y = list2[n]
    n += 1
    print(x + y)
```

Cache:

00	0		
01	0		
10	0		
11	0		

RAM:

000	24
001	15
010	-7
011	31
100	-20
101	16
110	-4
111	0

Thrashing

```
n = 0
while n <= 3:
    x = list1[n]
    y = list2[n]
    n += 1
    print(x + y)
```

Cache:

00	0		
01	0		
10	0		
11	0		

RAM:

000	24
001	15
010	-7
011	31
100	-20
101	16
110	-4
111	0

Thrashing

```
n = 0
while n <= 3:
    x = list1[n]
    y = list2[n]
    n += 1
    print(x + y)
```

Cache:

00	0		
01	0		
10	0		
11	0		

RAM:

000	24
001	15
010	-7
011	31
100	-20
101	16
110	-4
111	0

Thrashing

```
n = 0
while n <= 3:
    x = list1[n]
    y = list2[n]
    n += 1
    print(x + y)
```

Cache:

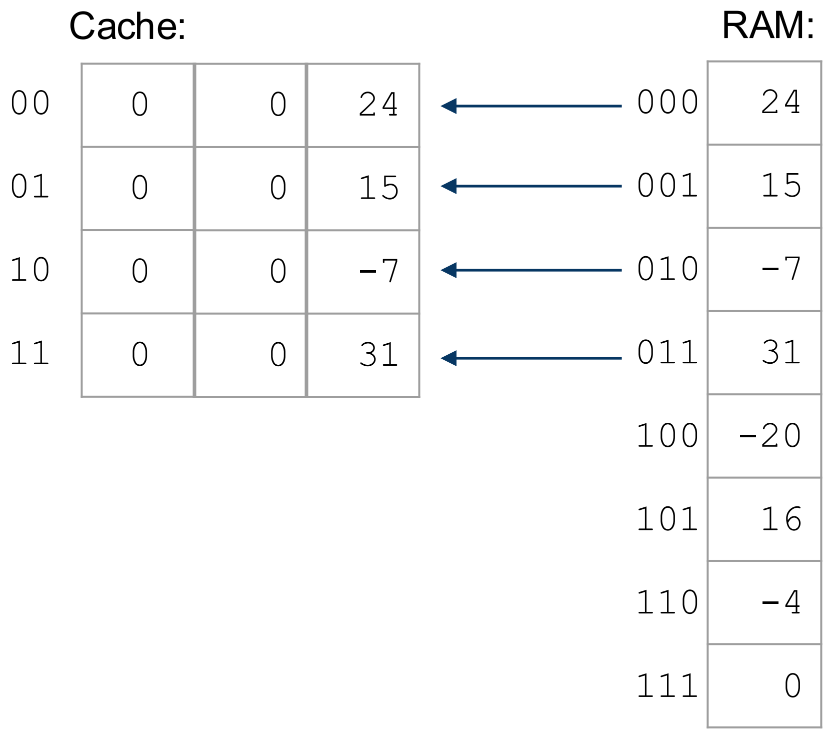
00	0		
01	0		
10	0		
11	0		

RAM:

000	24
001	15
010	-7
011	31
100	-20
101	16
110	-4
111	0

Thrashing

```
n = 0
while n <= 3:
    x = list1[n]
    y = list2[n]
    n += 1
    print(x + y)
```



Cache:

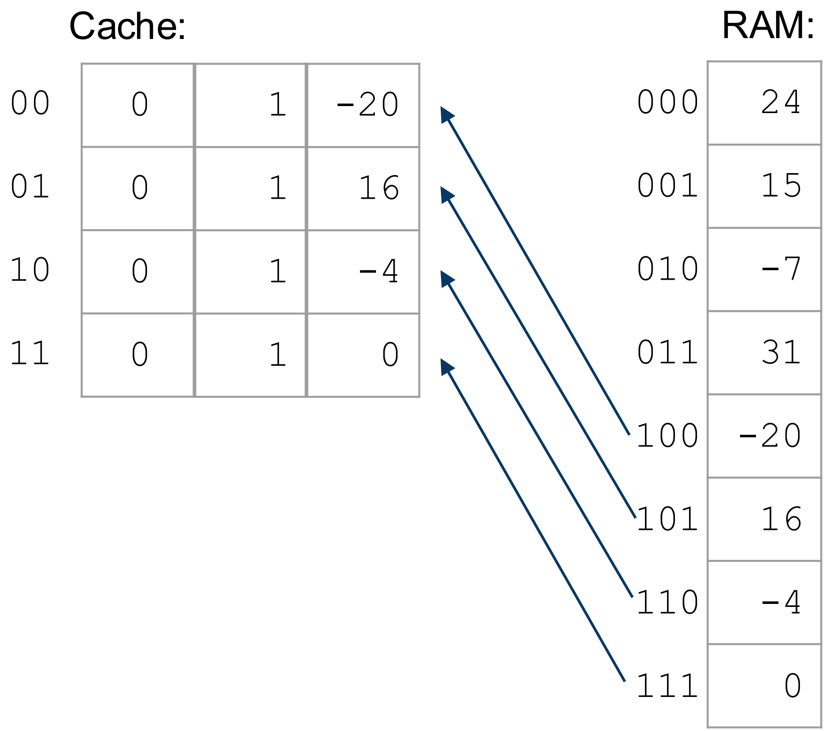
00	0	0	24
01	0	0	15
10	0	0	-7
11	0	0	31

RAM:

000	24
001	15
010	-7
011	31
100	-20
101	16
110	-4
111	0

Thrashing

```
n = 0
while n <= 3:
    x = list1[n]
    y = list2[n]
    n += 1
    print(x + y)
```



Cache:

00	0	1	-20
01	0	1	16
10	0	1	-4
11	0	1	0

RAM:

000	24
001	15
010	-7
011	31
100	-20
101	16
110	-4
111	0

Thrashing

```
n = 0
while n <= 3:
    x = list1[n]
    y = list2[n]
    n += 1
    print(x + y)
```

Cache:

00	0	1	-20
01	0	1	16
10	0	1	-4
11	0	1	0

RAM:

000	24
001	15
010	-7
011	31
100	-20
101	16
110	-4
111	0

Thrashing

```
n = 0
while n <= 3:
    x = list1[n]
    y = list2[n]
    n += 1
    print(x + y)
```

Cache:

00	0	1	-20
01	0	1	16
10	0	1	-4
11	0	1	0

RAM:

000	24
001	15
010	-7
011	31
100	-20
101	16
110	-4
111	0

Thrashing

```
n = 0
while n <= 3:
    x = list1[n]
    y = list2[n]
    n += 1
    print(x + y)
```

Cache:

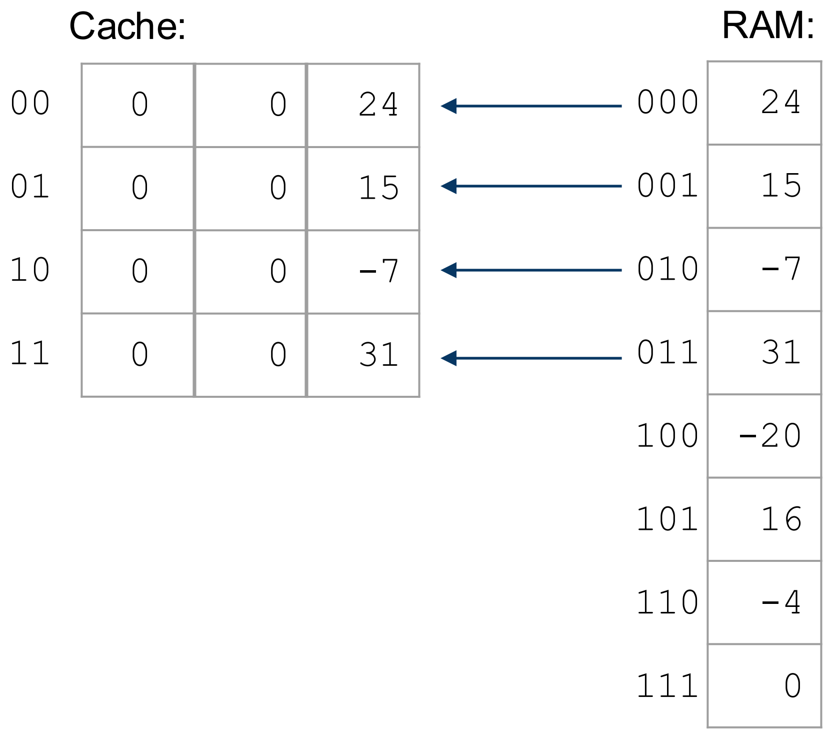
00	0	1	-20
01	0	1	16
10	0	1	-4
11	0	1	0

RAM:

000	24
001	15
010	-7
011	31
100	-20
101	16
110	-4
111	0

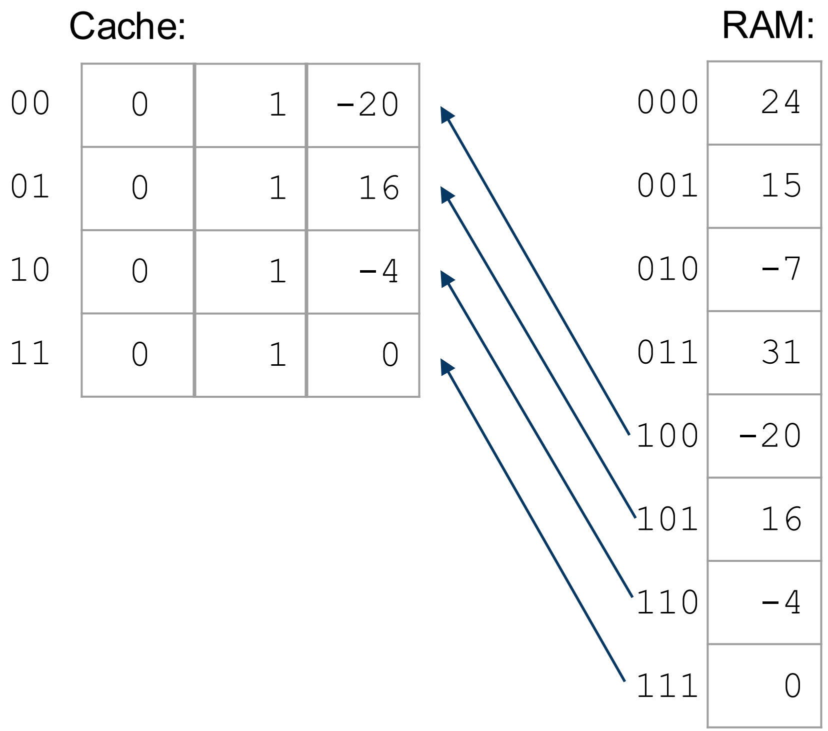
Thrashing

```
n = 0
while n <= 3:
    x = list1[n]
    y = list2[n]
    n += 1
    print(x + y)
```



Thrashing

```
n = 0
while n <= 3:
    x = list1[n]
    y = list2[n]
    n += 1
    print(x + y)
```



Cache:

00	0	1	-20
01	0	1	16
10	0	1	-4
11	0	1	0

RAM:

000	24
001	15
010	-7
011	31
100	-20
101	16
110	-4
111	0

Associative-Mapped Caches

- Direct-mapped caches only let you store one chunk of memory in the cache
    - If you are accessing two arrays, in two different parts of memory, this will lead to very inefficient cache utilization
- An associative-mapped cache lets you store any block(s) from anywhere in RAM
    - As any block can be mapped to any line in the cache, we need to be able to identify the rest of the main memory address
        - e.g. store the entire address in the tag
        - e.g. use the LSBs to determine a cache line number, store the rest of the address
    - This is more overhead, but could lead to better cache utilization

Cache Replacement

- When the cache is full, unwanted values must be removed to make room for requested data
  - How do we determine what data is unwanted?
    - First in, first out (FIFO)
    - Random
    - Least recently used (LRU)

Cache-Aware Programming

- How can you write code that results in more efficient caching?
  - Focus on optimizing code executed more:
    - Inner loops are more important than outer loops
    - Common cases (likely code paths; e.g. if vs. else)
  - Commonly used variables should be local scope
    - The compiler will often put these into registers
  - Iterate over arrays using stride-1 reference patterns (i.e. one after the other)
    - Sequential access has maximum spatial locality

Virtual Memory

CSCI 2050U - Computer Architecture

Virtual Memory

- Virtual memory is a mapping between RAM and non-volatile storage (SSDs and HDDs)
    - A virtual memory address may reference data that is loaded into RAM (at some physical memory address)
    - A virtual memory address may also reference data that is not currently loaded into RAM, but is still on a disk

Virtual Memory

- You can think of virtual memory as a form of RAM-based caching for disk data
    - Cache blocks in virtual memory are called pages
    - Loading a disk data page to/from RAM is called *paging/swapping*
- Virtual memory can suffer from thrashing, just like SRAM caches
    - An application (or pair of applications) alternately request a page, which keeps getting swapped into the same memory location, swapping each other out
    - As this involves a disk load, this can have a big impact on performance

Virtual Addressing

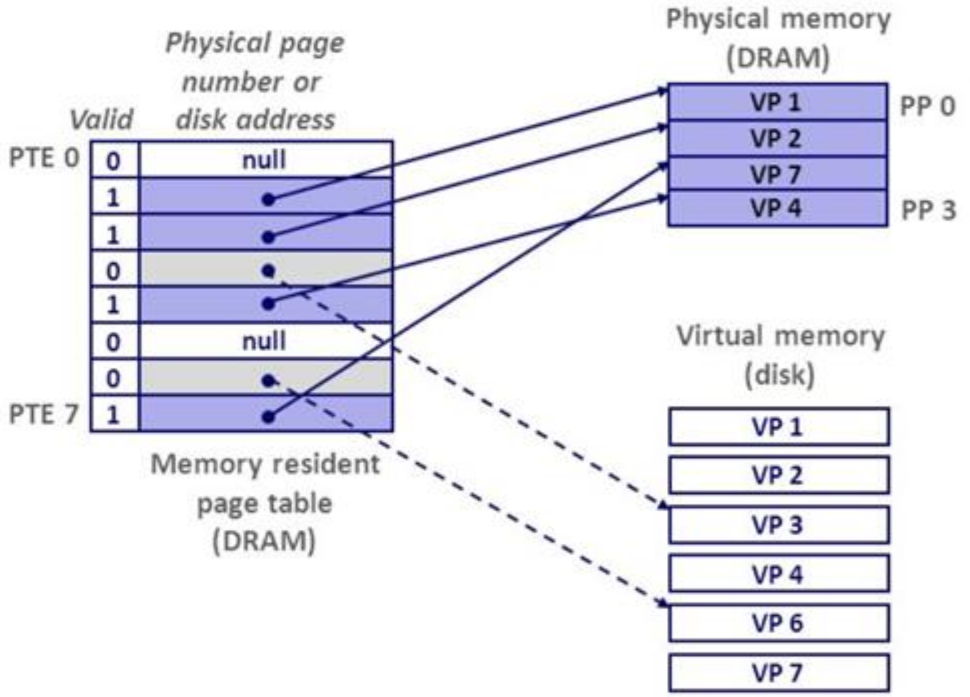
- Physical addresses (PA) are the actual addresses used by the hardware to request reads/writes from/to RAM
- Virtual addresses (VA) are the addresses used by programmers/compilers to reference memory locations
  - Pointers
  - References
- Translation between PAs and VAs is done by the memory management unit (MMU)

Terminology

- *page hit*: When a program tries to access a virtual address within a page that is currently located in memory
- *page fault*: When a program tries to access a virtual address within a page that is not currently located in memory
- *swapped out*: When a page in memory is moved to disk
- *swapped in*: When a page on disk is moved to memory
- *page*: A unit of memory (equivalent to a cache's block)

Page Tables

- A page table is a data structure used to store the location of a page



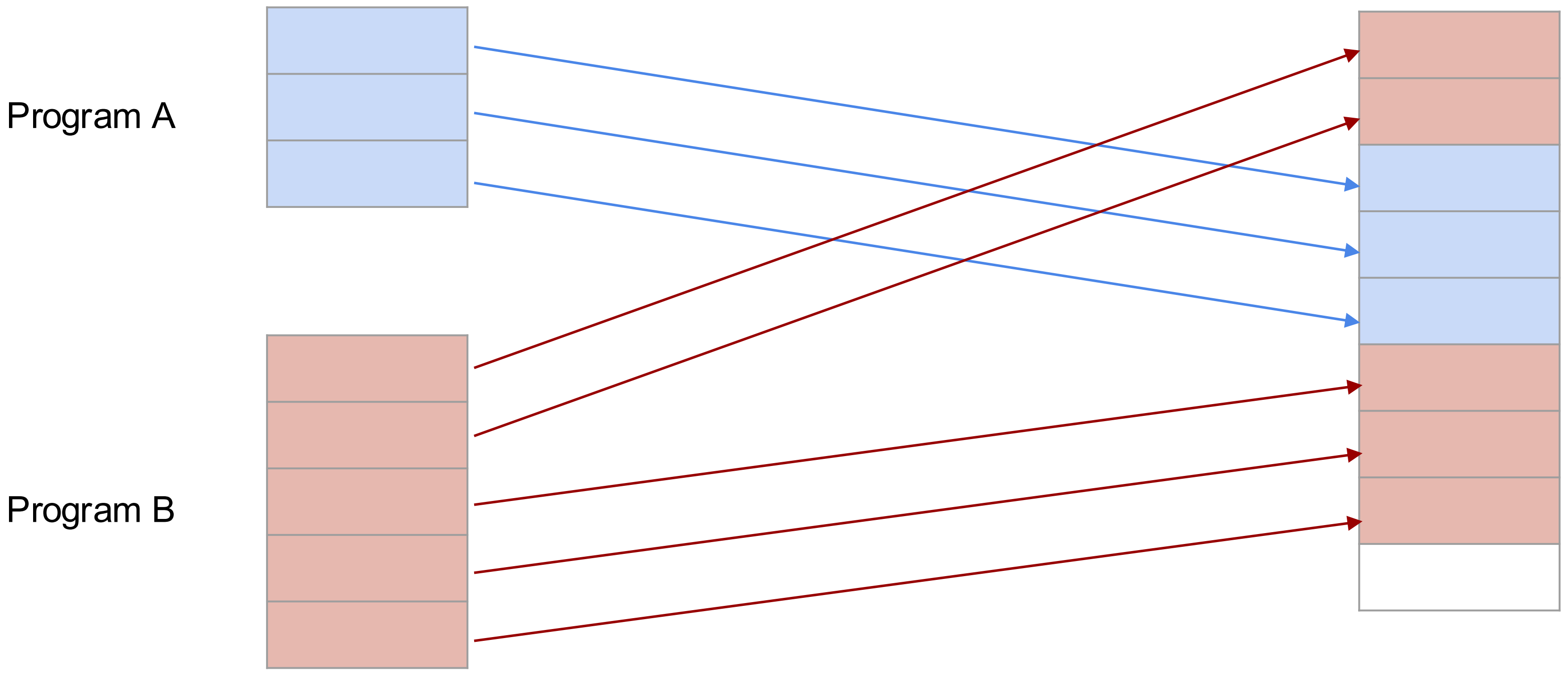
Virtual Address Translation

- Each page table does not map a single cell, as discussed previously
- Each page is a sequence of memory cells
- The page table will use a portion of the virtual address as the offset within the page
    - e.g. Page size is 4k, then the offset would be 12 bits (2^{12} = 4096 = 4k)
- Each page table entry will contain the base address for the page in memory
    - The resulting physical address would then be: `base_address + offset`

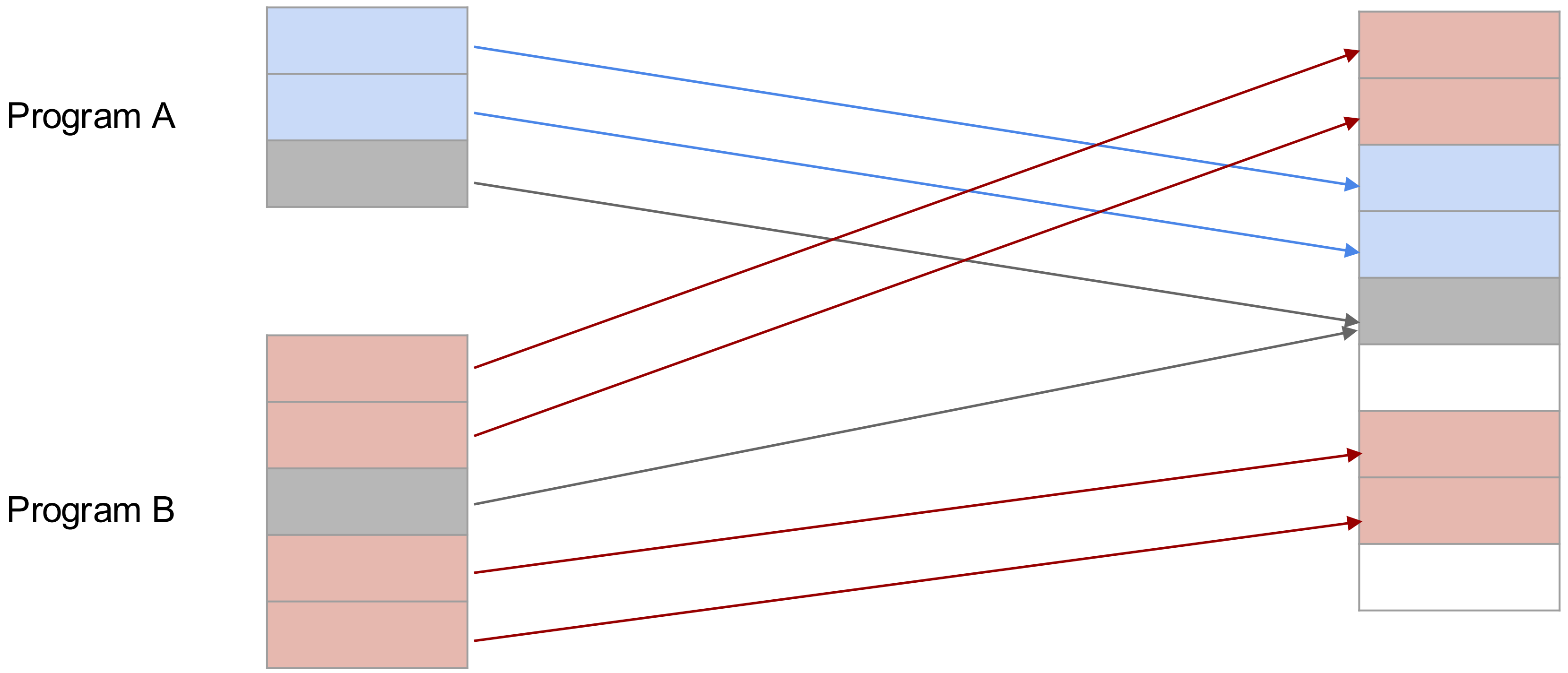
Advantages of Virtual Memory

1. Memory is separated for each process
   - Virtual address mapping is unique to each process
   - Other processes cannot access the same physical addresses, since there is no mapping present
2. Simplified linking/loading
   - Some operating systems load programs into the same virtual address
   - The memory structure of every running program is identical
3. Simplified memory allocation
   - Pages do not need to be physically contiguous to be part of a contiguous virtual memory structure
4. Better address utilization

Virtual Memory Allocation



Shared Data



Page Faults

- When a page is requested, but the page table entry shows that the page is located on disk, a *page fault* occurs
    - The CPU will generate a page fault exception (also called an *interrupt*)
    - The operating system will have a handler for this exception that will load the page into memory
    - An old page will be chosen to be removed from memory
- What if the data in the old page has been changed?
    - This is called a *dirty* page
    - Before the new page can be loaded, the existing page must be written to disk

Wrap-up

- Caching
- Virtual memory

What is next?

- Instruction cycle
- Additional digital circuit components
- Data path/bus
- Fetch