The end of cache!

2/6/26

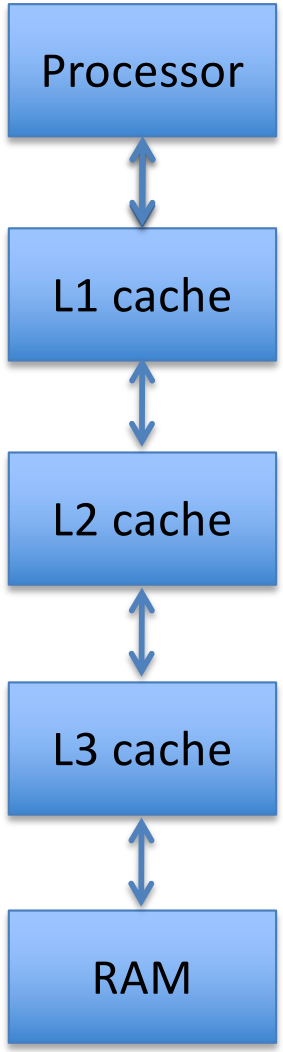
Administrivia

- HW 5 (MIPS simulator with caching) due Tuesday (2/10) night

- Extra credit: “Robot that prays”: Today (2/6) at 4pm in the Abolition Lab. Attend and email me a 1 page reaction (not summary)

Recall: Idea of caching

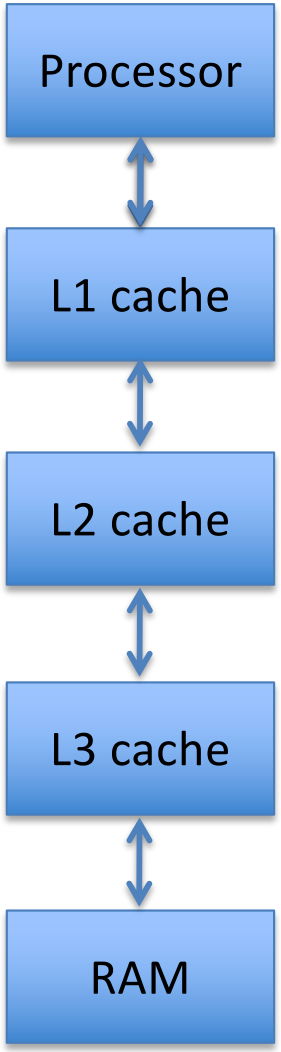
- Have small, fast memory near the processor and bigger, slower memory far way
- Locality:
  - temporal: Likely to reuse memory addresses soon
  - spatial: Likely to use memory addresses near those we use now



What about writes?

- Write-thru cache: Always send writes all the way down the memory hierarchy

- Write-back cache: Don't propagate changes down the hierarchy until data block is evicted
  - Add "dirty bit" to each line



Recall: Example of caching effects

```
int[][] array = new int[size][size];
for(int i=0; i < size; i++)
    for(int j=0; j < size; j++)
        /* DO SOMETHING */
```

Version 1: array[i][j] *= 2;

Version 2: array[j][i] *= 2;

Relevant detail: Java stores a 2D array as an array of array references

Recall: Example of caching effects

```
int[][] array = new int[size][size];
for(int i=0; i < size; i++)
    for(int j=0; j < size; j++)
        /* DO SOMETHING */
```

Version 1: array[i][j] *= 2;

Version 2: array[j][i] *= 2;

Which version should have better cache performance?

A) Version 1

B) Version 2

Relevant detail: Java stores a 2D array as an array of array references

Recall: Example of caching effects

```
int[][] array = new int[size][size];
for(int i=0; i < size; i++)
    for(int j=0; j < size; j++)
        /* DO SOMETHING */
```

Version 1: array[i][j] *= 2;

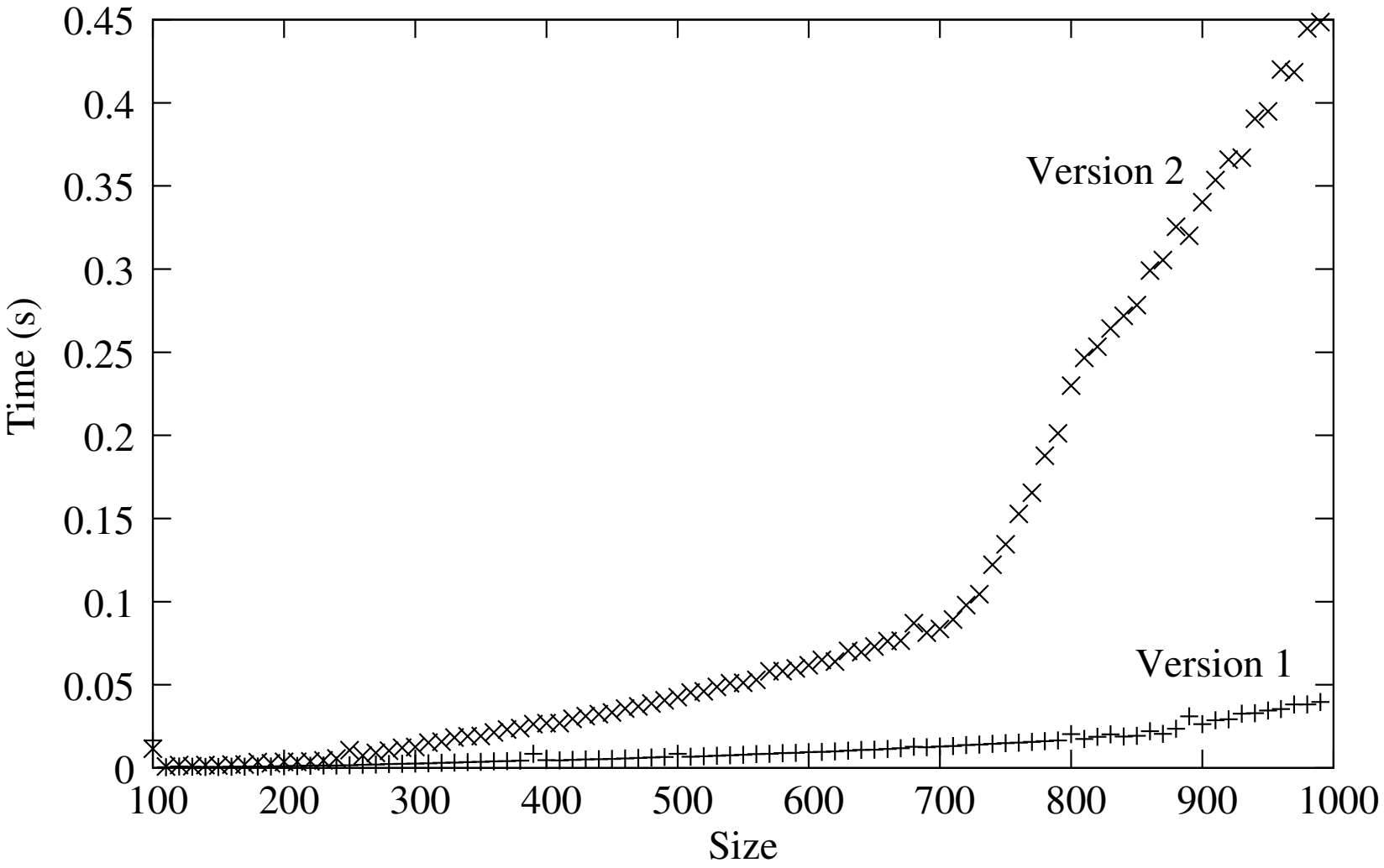
Version 2: array[j][i] *= 2;

Which version should have better cache performance?

A) Version 1

B) Version 2

Relevant detail: Java stores a 2D array as an array of array references


0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
Time (s)
100
200
300
400
500
600
700
800
900
1000
Size
Version 2
Version 1

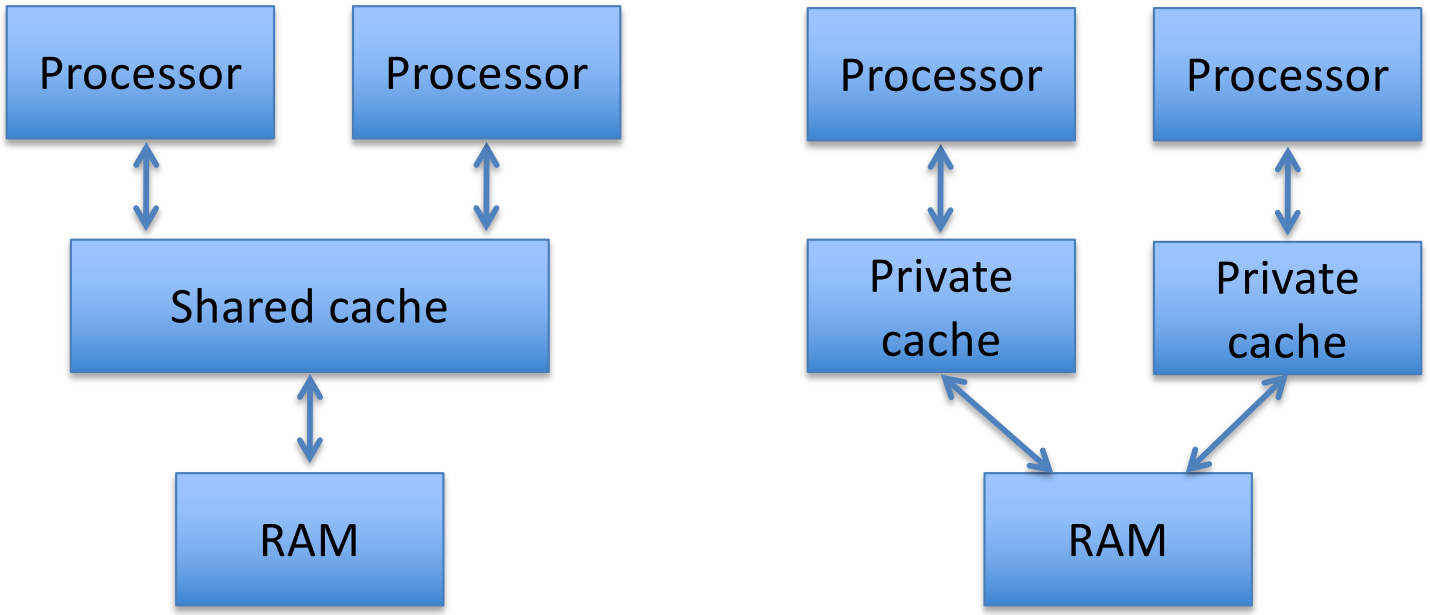
Caching: So cool it's not just for the processor

- Web browser caches pages that you've viewed
- Name servers cache translations they've recently done between names and IP addresses (e.g. cs.knox.edu becomes 72.26.72.37)

Multiprocessor caching

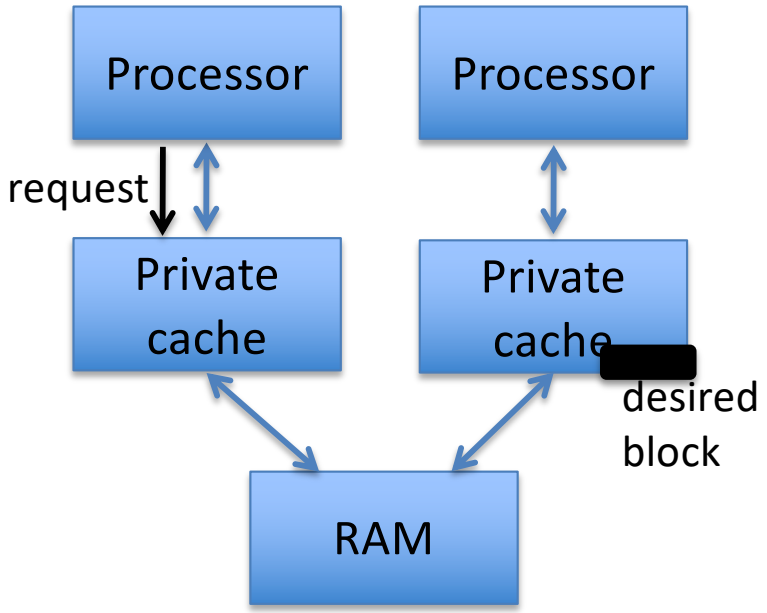
Multiprocessor caching

- How do you arrange caching for multiple processors/cores in the same address space?



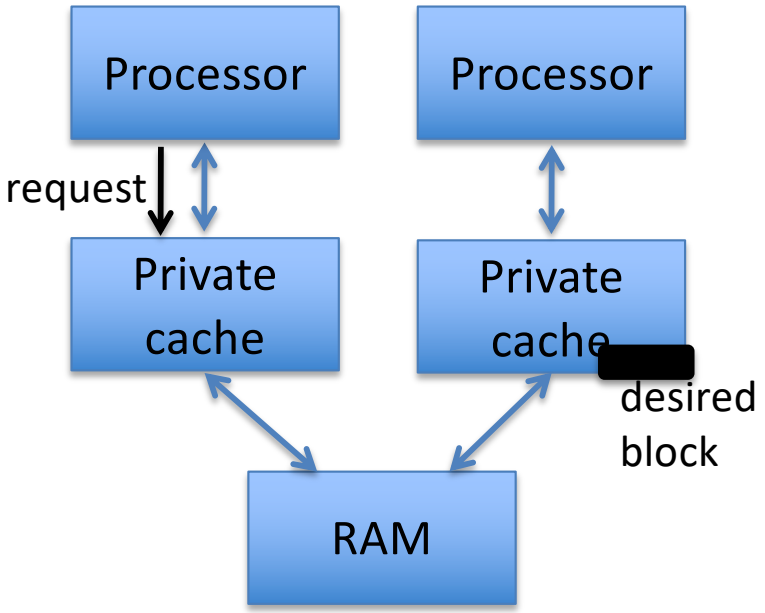
Cache coherence

- Caches should present unified view of memory
- Potentially an issue when a data block is in one cache and other PE requests it



Cache coherence

- Caches should present unified view of memory

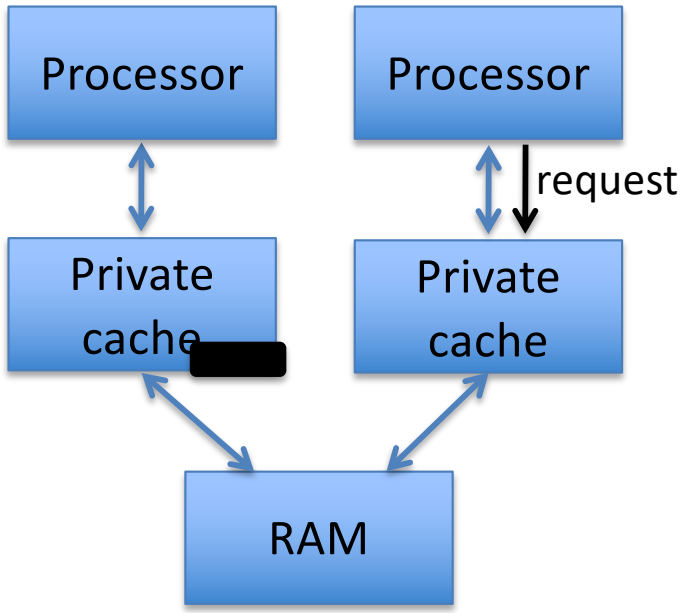


- Potentially an issue when a data block is in one cache and other PE requests it

- Left cache must “steal” the requested block from the right cache

Worst case

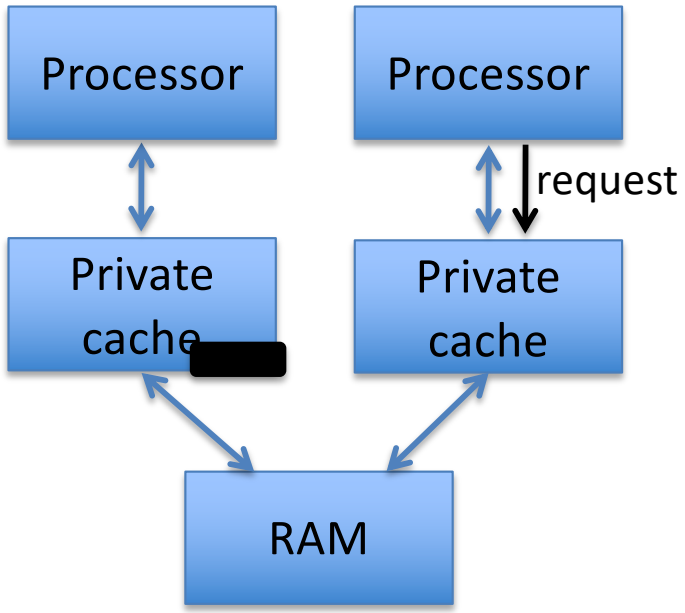
- Data block keeps getting stolen back and forth between the private caches



Worst case

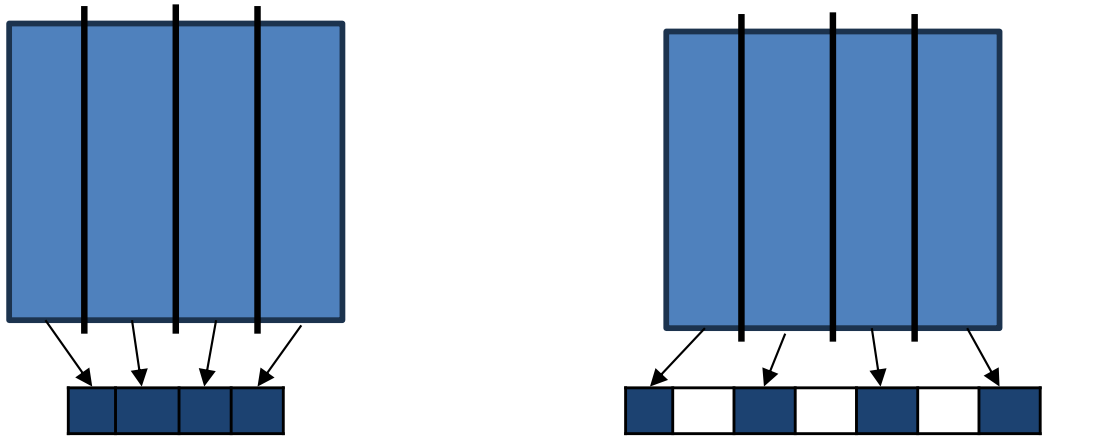
- Data block keeps getting stolen back and forth between the private caches

- Even worse: There isn't any data being shared (called *false sharing*)



Example

- Take an image, split into regions, and count black pixels in each region



- With 16 threads, version on right took about half the time

Virtual memory

Awkward facts

- Modern processors use 64-bit addresses

  $\Longrightarrow$ can access 2^{64} addresses

Awkward facts

- Modern processors use 64-bit addresses

  $\Longrightarrow$ can access 2^{64} addresses

  $2^{10} = 1024 \approx 1000$

  $2^{64} = 2^4(2^{10})^6 \approx 16(1000)^6$

  $\approx 16 \cdot 1{,}000{,}000{,}000{,}000{,}000$

  aka 16 exabytes

Awkward facts

- Modern processors use 64-bit addresses

  $\Longrightarrow$ can access 2^{64} addresses

  $2^{10} = 1024 \approx 1000$

  $2^{64} = 2^4(2^{10})^6 \approx 16(1000)^6$

  $\approx 16 \cdot 1{,}000{,}000{,}000{,}000{,}000$

  aka 16 exabytes

- Every program gets its own address space

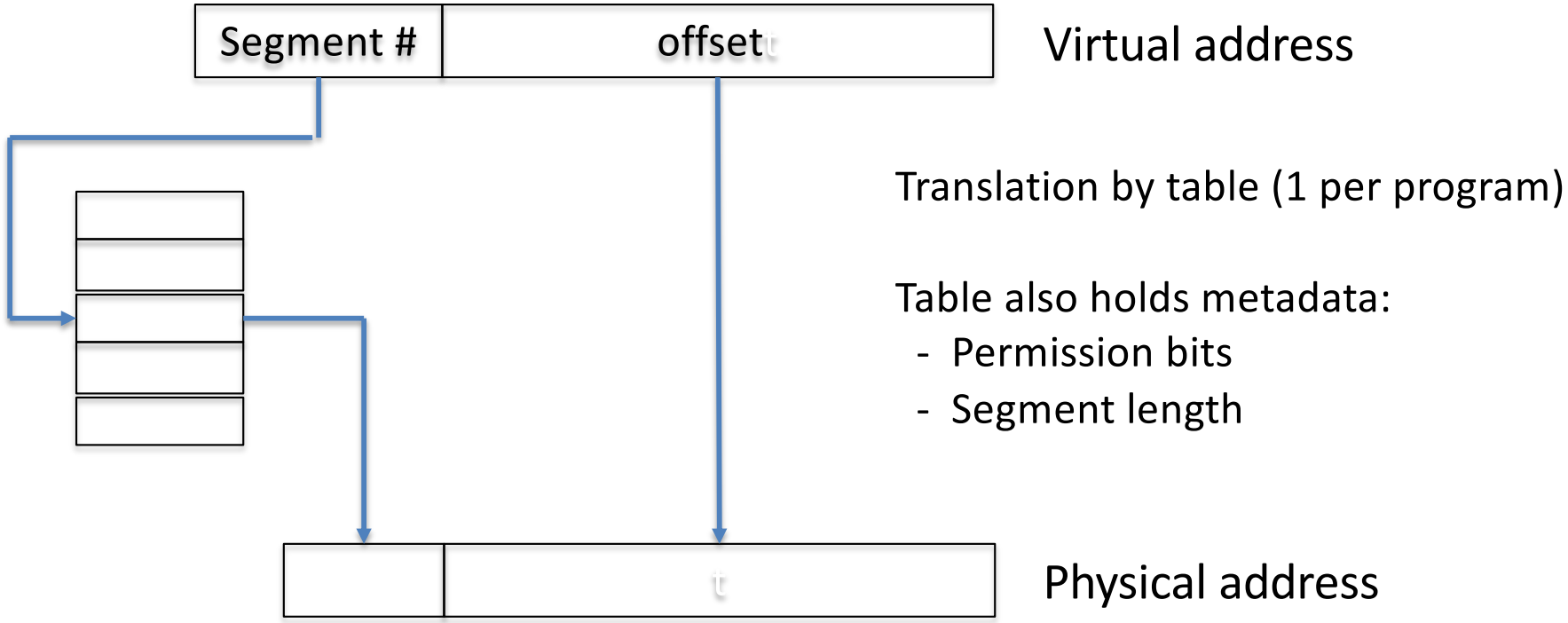
Virtual memory

- Programs use virtual addresses that map to physical addresses

- Give each program its own address space
  - Simplifies programming:
    - Programs don't have to manage memory
  - Simplifies multitasking
    - Programs use any addresses they want
  - Isolates programs from each other and the OS

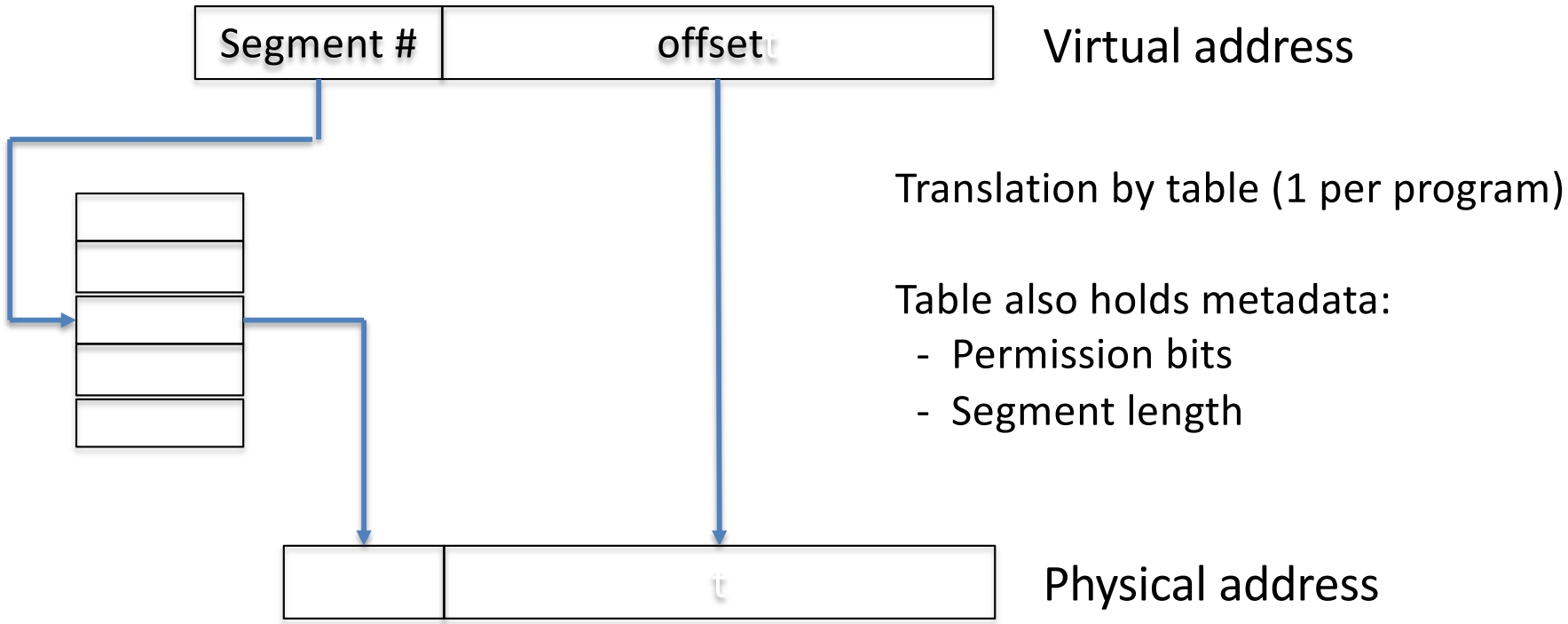
Version 1: Segmentation

- Memory divided by logical function (OS, program code, library, data, heap, …) into segments



Version 1: Segmentation

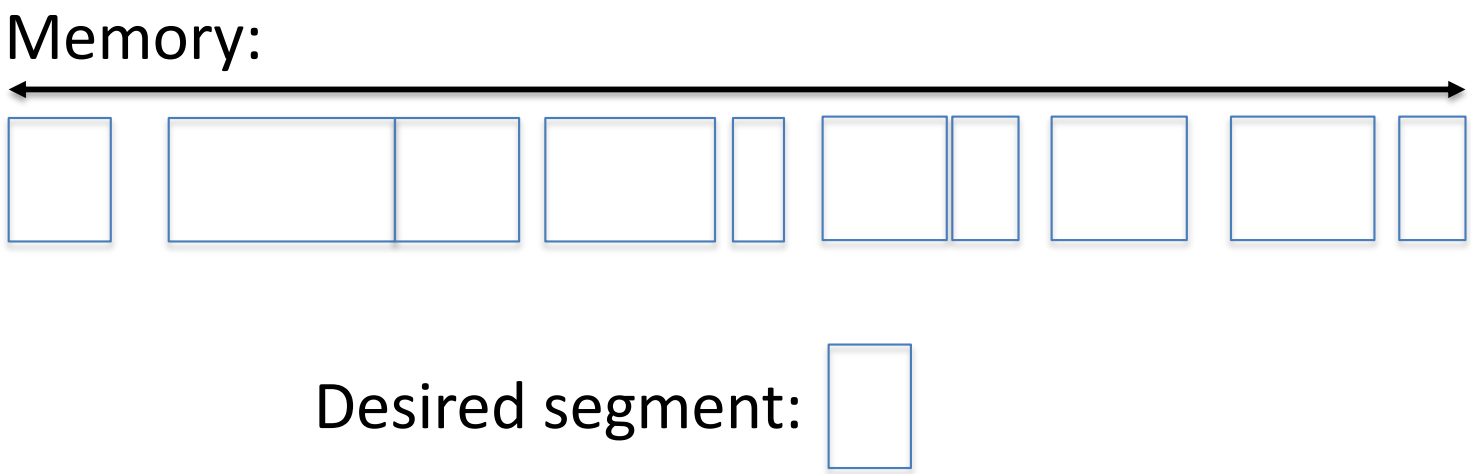
- Memory divided by logical function (OS, program code, library, data, heap, …) into segments



What if segment length exceeds maximum possible?

External fragmentation

- Occurs when there is enough free space for a desired segment, but it's not all together



Internal fragmentation

- Occurs when space inside the segments is wasted

Version 2: Paging

- Memory organized into small (4-16KB), fixed-size “pages” which are allocated as needed
- Translation by “page table”
- Accessing unmapped memory is “page fault”

Real systems use a combination

- Memory organized into small pages, each containing one type of data

- Terminology from both: “segmentation fault”, “page fault”, “page table”