CS61C Su18 Final Review

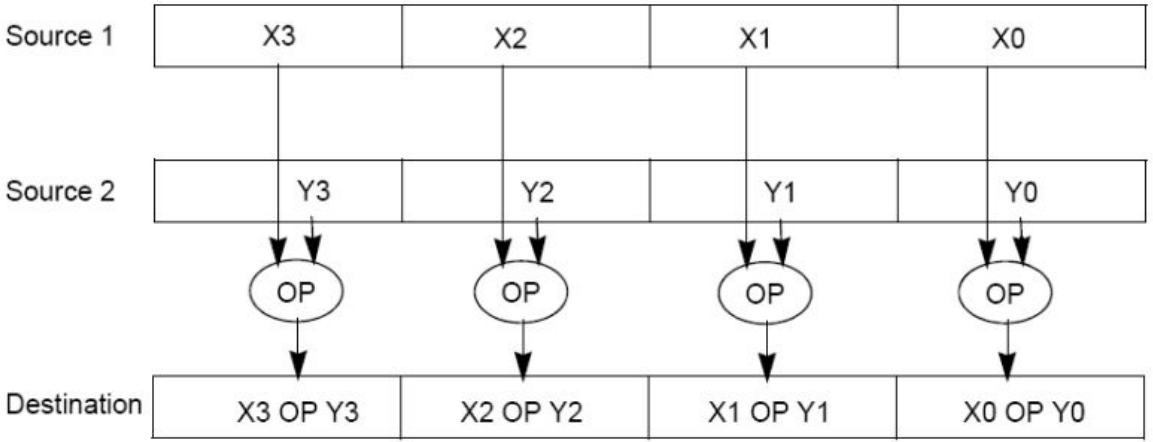
Quick Administrivia

- Another review session on Tuesday during lecture time
- Will cover MT1/MT2 material

Performance Programming: SIMD (DLP)

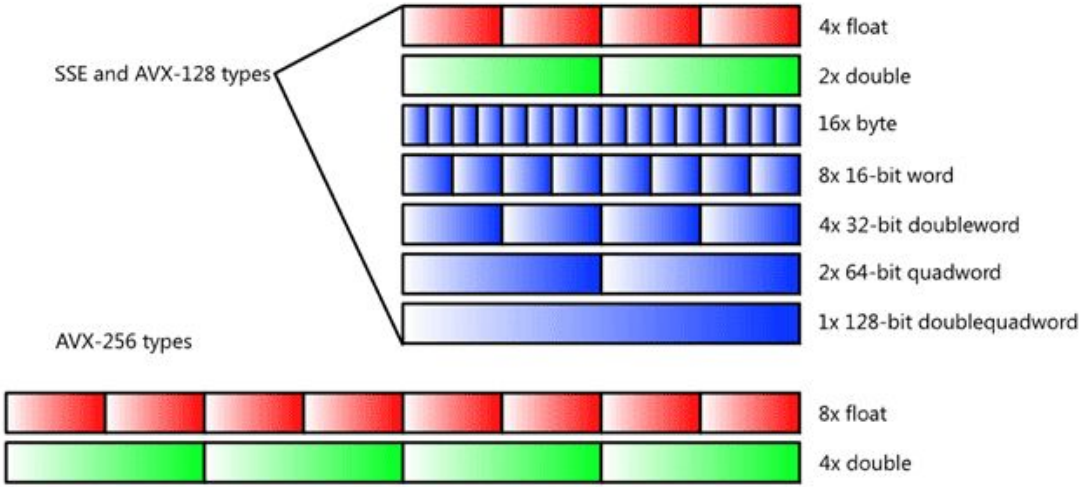
SIMD (Single Instruction Multiple Data)

- Execute same operation on multiple data streams
- Fetch one instruction, and do the work of multiple instructions



SIMD in 61C: Intel Intrinsics

- Some Intel processors have special, wider registers to do SIMD with; they can be split up multiple ways



Summer 2014 - Final

Lagrange interpolation is a useful technique for fitting a degree n polynomial to points $(x_0, y_0), \ldots, (x_n, y_n)$ by summing together $n+1$ different Lagrange polynomials, with each different Lagrange polynomial having the form

$$L_k(x) = y_k \frac{(x - x_0) \cdots (x - x_{k-1})(x - x_{k+1}) \cdots (x - x_n)}{(x_k - x_0) \cdots (x_k - x_{k-1})(x_k - x_{k+1}) \cdots (x_k - x_n)}$$

Starting from the following naive function:

```
/** Evaluates the Kth Lagrange polynomial generated by the N different
 *  inputs (X[0],Y[0]),(X[1],Y[1]),...,(X[N-1],Y[N-1]) at C. */
float eval_lagrange(float *X, float *Y, float c, size_t n, size_t k) {
    float retval = 1;
    for (size_t i = 0; i < n; i += 1) {
        if (i == k)
            continue;
        retval *= c - X[i];
        retval /= X[k] - X[i];
    }
    return retval * Y[k];
}
```

continue: skips remaining code in current iteration of loop & jumps to next iteration

Complete the following SIMD-ized version of the function, optimizing for performance, and assuming that n is a multiple of 4.

```
float eval_lagrange_fast(float *X, float *Y, float c, size_t n, size_t k) {
    float retval = 1, m[4];
    size_t i;
    __m128 ret_vec = _mm_set1_ps(1);

    for (i = 0; i < ________________; i += 1) {

        if (________________)
            continue;
        ret_vec = _mm_mul_ps(ret_vec, _mm_sub_ps(_mm_set1_ps(c),

                                                 _mm_loadu_ps(__________)));

        ret_vec = _mm_div_ps(ret_vec, _mm_sub_ps(_____________________,

                                                 _mm_loadu_ps(__________)));
    }

    for (________________; i < ________________; i += 1) {
        if (i == k)
            continue;
        retval *= c - X[i];
        retval /= X[k] - X[i];
    }

    ____________________________________________________________;

    return ____________________________________________________;
}
```

Hint: The instruction `_mm_load1_ps(addr)` loads the value at `addr` into each of the 4 entries of the resulting vector.

Original Code

```
float eval_lagrange(float *X, float *Y, float c, size_t n, size_t k) {
    float retval = 1;
    for (size_t i = 0; i < n; i += 1) {
        if (i == k)
            continue;
        retval *= c - X[i];
        retval /= X[k] - X[i];
    }
    return retval * Y[k];
}
```

Complete the following SIMD-ized version of the function, optimizing for performance, and assuming that n is a multiple of 4.

```
float eval_lagrange_fast(float *X, float *Y, float c, size_t n, size_t k) {
    float retval = 1, m[4];
    size_t i;
    __m128 ret_vec = _mm_set1_ps(1);

    for (i = 0; i < ________________; i += 1) {

        if (________________)
            continue;
        ret_vec = _mm_mul_ps(ret_vec, _mm_sub_ps(_mm_set1_ps(c),

                                                 _mm_loadu_ps(___________)));

        ret_vec = _mm_div_ps(ret_vec, _mm_sub_ps(_____________________________,

                                                 _mm_loadu_ps(___________)));
    }
```

Complete the following SIMD-ized version of the function, optimizing for performance, and assuming that n is a multiple of 4.

```
float eval_lagrange_fast(float *X, float *Y, float c, size_t n, size_t k) {
    float retval = 1, m[4];
    size_t i;
    __m128 ret_vec = _mm_set1_ps(1);
    for (i = 0; i < n/4; i += 1) {
        if (__________________)
            continue;
        ret_vec = _mm_mul_ps(ret_vec, _mm_sub_ps(_mm_set1_ps(c),
                                                 _mm_loadu_ps(___________)));
        ret_vec = _mm_div_ps(ret_vec, _mm_sub_ps(_____________________________,
                                                 _mm_loadu_ps(___________)));
    }
```

Complete the following SIMD-ized version of the function, optimizing for performance, and assuming that n is a multiple of 4.

```
float eval_lagrange_fast(float *X, float *Y, float c, size_t n, size_t k) {
    float retval = 1, m[4];
    size_t i;
    __m128 ret_vec = _mm_set1_ps(1);
    for (i = 0; i < n/4; i += 1) {
        if (i == k/4)
            continue;
        ret_vec = _mm_mul_ps(ret_vec, _mm_sub_ps(_mm_set1_ps(c),
                                                 _mm_loadu_ps(____________)));
        ret_vec = _mm_div_ps(ret_vec, _mm_sub_ps(_____________________________,
                                                 _mm_loadu_ps(____________)));
    }
```

The given formula omits $(x-x_k)$ in the numerator and (x_k-x_k) in the denominator

Complete the following SIMD-ized version of the function, optimizing for performance, and assuming that n is a multiple of 4.

```
float eval_lagrange(float *X, float *Y, float c, size_t n, size_t k) {
    float retval = 1;
    for (size_t i = 0; i < n; i += 1) {
        if (i == k)
            continue;
        retval *= c - X[i];
        retval /= X[k] - X[i];
    }
    return retval * Y[k];
}
```

```
float eval_lagrange_fast(float *X, float *Y, float c, size_t n, size_t k) {
    float retval = 1, m[4];
    size_t i;
    __m128 ret_vec = _mm_set1_ps(1);
    for (i = 0; i < n/4; i += 1) {
        if (i == k/4)
            continue;
        ret_vec = _mm_mul_ps(ret_vec, _mm_sub_ps(_mm_set1_ps(c),
                                                 _mm_loadu_ps(X + i*4)));
        ret_vec = _mm_div_ps(ret_vec, _mm_sub_ps(_mm_load1_ps(X + k),
                                                 _mm_loadu_ps(X + i*4)));
    }
```

Tail Case: for the area around i == k

```
for (__________________; i < __________________; i += 1) {
    if (i == k)
        continue;
    retval *= c - X[i];
    retval /= X[k] - X[i];
}

_____________________________________________________________;

return _____________________________________________________;
```

Tail Case: for the area around i == k

```
for (i = k/4*4; i < k/4*4 + 4; i += 1) {
    if (i == k)
        continue;
    retval *= c - X[i];
    retval /= X[k] - X[i];
}
_mm_storeu_ps(m, ret_vec);
return Y[k] * retval * m[0] * m[1] * m[2] * m[3];
```

Tail Case 2?

- We don’t need a second tail case, because…

Complete the following SIMD-ized version of the function, optimizing for performance, and assuming that n is a multiple of 4.

- But if this isn’t the case, be sure you don’t forget it!

```
for (int i = n/4 * 4; i < n; i++)
{
    if (i == k) {
        continue;
    }
    retval *= c - X[i];
    retval /= X[k] - X[i];
}
```

And Finally…

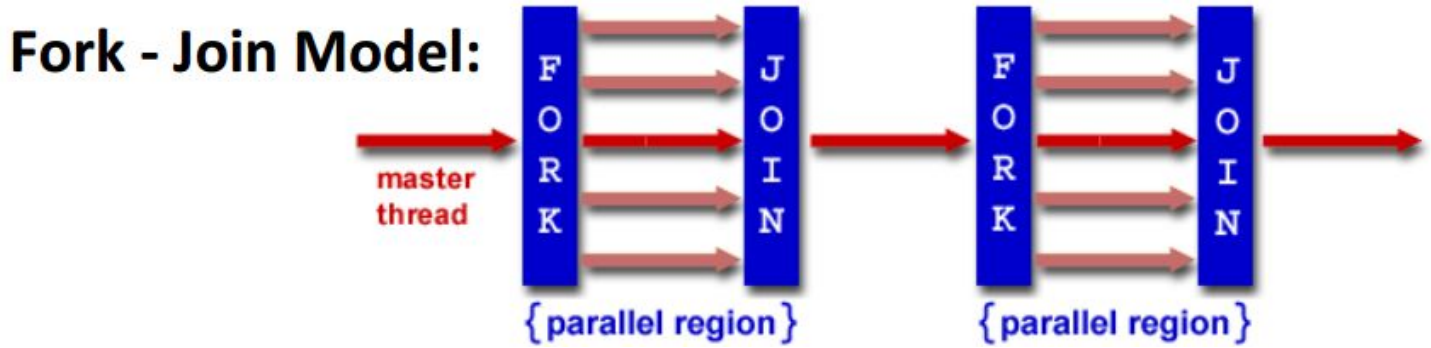
```
float eval_lagrange_fast(float *X, float *Y, float c, size_t n, size_t k) {
    float retval = 1, m[4];
    size_t i;
    __m128 ret_vec = _mm_set1_ps(1);
    for (i = 0; i < n/4; i += 1) {
        if (i == k/4)
            continue;
        ret_vec = _mm_mul_ps(ret_vec, _mm_sub_ps(_mm_set1_ps(c),
                                                 _mm_loadu_ps(X + i*4)));
        ret_vec = _mm_div_ps(ret_vec, _mm_sub_ps(_mm_loadl_ps(X + k),
                                                 _mm_loadu_ps(X + i*4)));
    }
    for (i = k/4*4; i < k/4*4 + 4; i += 1) {
        if (i == k)
            continue;
        retval *= c - X[i];
        retval /= X[k] - X[i];
    }
    _mm_storeu_ps(m, ret_vec);
    return Y[k] * retval * m[0] * m[1] * m[2] * m[3];
}
```

Performance Programming: OpenMP (TLP)

Thread Level Parallelism (OMP)

- **Software thread:** a sequential flow of instructions that performs some task
- Each software thread has a PC + processor registers and access to the shared memory
- Each processor provides one (or more) **hardware threads** that actively execute instructions

Thread Level Parallelism (OMP)



- **Fork:** Master thread creates a team of parallel threads
- **Join:** When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread
- **Amdahl's Law:** 1/[(1-F) + F/S] (F = Fraction parallelizable, S = Amount of speedup)

Thread Level Parallelism (OMP)

- **OpenMP** is a language extension used for multi-threaded, shared-memory parallelism
- Uses various directives to create multi-threaded code segments
- #pragma omp parallel
  - Code inside block will be run on each thread
  - Helper Functions:
    - omp_get_num_threads() = Number of Threads running
    - omp_get_thread_num() = Current Thread ID

Credit to David for Slide Content

Thread Level Parallelism (OMP)

- Use **#pragma omp parallel for** to split for loop work over multiple threads

```
#pragma omp parallel for
{
    for(int i=0; i<ARRAY_SIZE; i++)
        z[i] = x[i] + y[i];
}
```

- Watch out for data races!
  - **Data race** when different threads try to access same memory location, and at least one is a write

```
#pragma omp parallel for
{
    for(int i=0; i<ARRAY_SIZE; i++)
        sum += x[i];
}
```

Data race for sum, end result will be inaccurate

Practice!

What's wrong with this code?

```
#pragma omp parallel
for (int i = 1; i < 1000; i++){
        A[i] = i*i;
}
```

Practice!

What's wrong with this code? Duplicated Work!

```
#pragma omp parallel
for (int i = 1; i < 1000; i++){
        A[i] = i*i;
}
```

Spring 2015 - Final

As the number of threads increases, will this program print the correct values for Even and Odd?

```
#include <stdio.h>
#include "omp.h"
void count_eo (int *A, int size, int threads) {
    int result[2] = {0, 0};
    int i,j;

    omp_set_num_threads(threads);

    #pragma omp parallel for
    for (j=0; j<size; j++)
        result[(A[j] % 2 == 0) ? 0 : 1] += 1;

    printf("Even: %d\n", result[0]);
    printf("Odd: %d\n", result[1]);

}
```

Spring 2015 - Final

As the number of threads increases, will this program print the correct values for Even and Odd? No, there may be a data race on `result`

```
#include <stdio.h>
#include "omp.h"
void count_eo (int *A, int size, int threads) {
    int result[2] = {0, 0};
    int i,j;

    omp_set_num_threads(threads);

    #pragma omp parallel for
    for (j=0; j<size; j++)
        result[(A[j] % 2 == 0) ? 0 : 1] += 1;

    printf("Even: %d\n", result[0]);
    printf("Odd: %d\n", result[1]);

}
```

In RISC-V, this line would involve a lw, addi, and a sw

Spring 2015 - Final

As the number of threads increases, will this program print the correct values for Even and Odd? No, there may be a data race on `result`

```
#include <stdio.h>
#include "omp.h"
void count_eo (int *A, int size, int threads) {
    int result[2] = {0, 0};
    int i,j;

    omp_set_num_threads(threads);

    #pragma omp parallel for
    for (j=0; j<size; j++)
        result[(A[j] % 2 == 0) ? 0 : 1] += 1;

    printf("Even: %d\n", result[0]);
    printf("Odd: %d\n", result[1]);

}
```

Let's say that we have 2 threads, and thread 0 is processing index 0 and thread 1 is processing index 2.

```
A = {2, 3, 4, 5}
result = {0, 0}
```

Spring 2015 - Final

As the number of threads increases, will this program print the correct values for Even and Odd? No, there may be a data race on `result`

```
#include <stdio.h>
#include "omp.h"
void count_eo (int *A, int size, int threads) {
    int result[2] = {0, 0};
    int i,j;

    omp_set_num_threads(threads);

    #pragma omp parallel for
    for (j=0; j<size; j++)
        result[(A[j] % 2 == 0) ? 0 : 1] += 1;

    printf("Even: %d\n", result[0]);
    printf("Odd: %d\n", result[1]);

}
```

Let's say that we have 2 threads, and thread 0 is processing index 0 and thread 1 is processing index 2. The order of execution is below.

`A = {2, 3, 4, 5}`

`result = {0, 0}`

Thread 0	Thread 1
1. lw	4. lw
2. addi	5. addi
3. sw	6. sw

`After: result = {2, 0} → Correct!`

Spring 2015 - Final

As the number of threads increases, will this program print the correct values for Even and Odd? No, there may be a data race on `result`

```
#include <stdio.h>
#include "omp.h"
void count_eo (int *A, int size, int threads) {
    int result[2] = {0, 0};
    int i,j;

    omp_set_num_threads(threads);

    #pragma omp parallel for
    for (j=0; j<size; j++)
        result[(A[j] % 2 == 0) ? 0 : 1] += 1;

    printf("Even: %d\n", result[0]);
    printf("Odd: %d\n", result[1]);

}
```

Let's say that we have 2 threads, and thread 0 is processing index 0 and thread 1 is processing index 2. The order of execution is below.

`A = {2, 3, 4, 5}`

`result = {0, 0}`

Thread 0	Thread 1
1. lw	2. lw
3. addi	5. addi
4. sw	6. sw

`After: result = {1, 0} → Incorrect!`

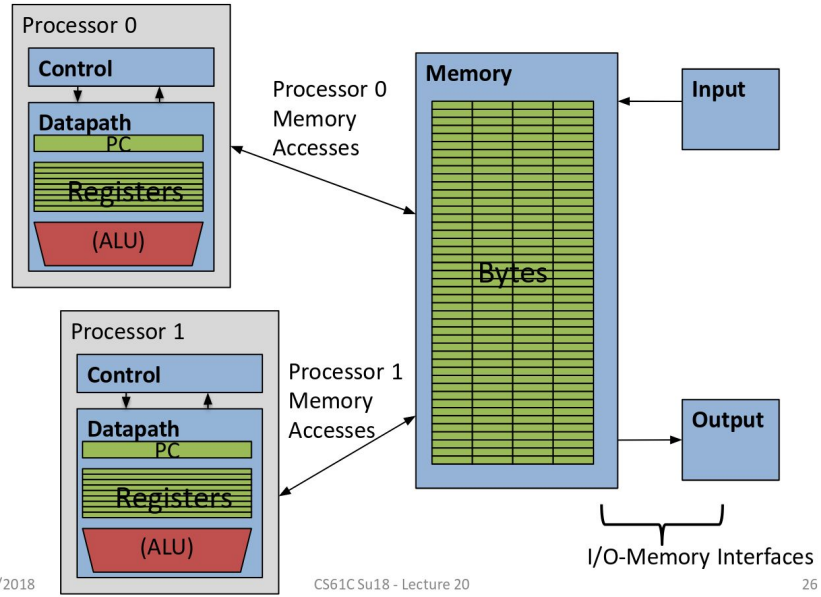
Spring 2015 - Final

How do we fix the data race? One solution: give each thread its own result

```
#include <stdio.h>
#include “omp.h”
void count_eo (int *A, int size, int threads) {
       omp_set_num_threads(threads);
       int even, odd;

       #pragma omp parallel
       {
       int result[2] = {0,0}
       int i, j;

       #pragma omp for
       for (j = 0; j < size; j++)
              result[ (A[j] % 2 == 0) ? 0 : 1] += 1;

       #pragma omp critical
       even += result[0];
       odd += result[1];
       }

       printf(“Even: %d\n”, result[0]);
       printf(“Odd: %d\n”, result[1]);
}
```

Performance Programming: Cache Coherence

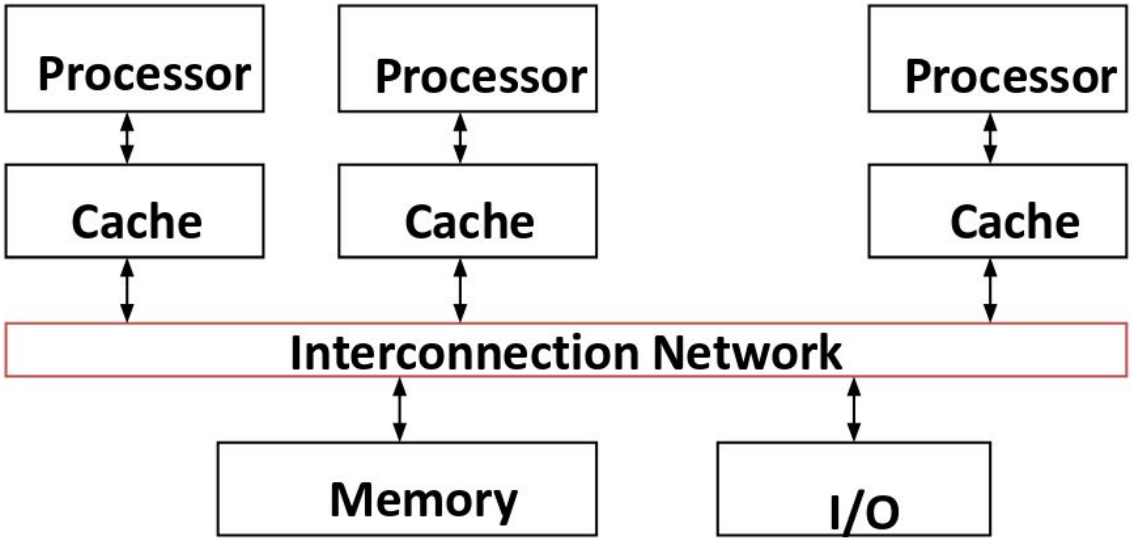
Why do we need cache coherence?



- In multicore architectures, multiple processors all share the same RAM

Why do we need cache coherence?

- Where are the caches?



Why do we need cache coherence?

- Each processor has its own private cache
- If other processors are changing memory, cache data needs to be kept **coherent** with memory
- How?

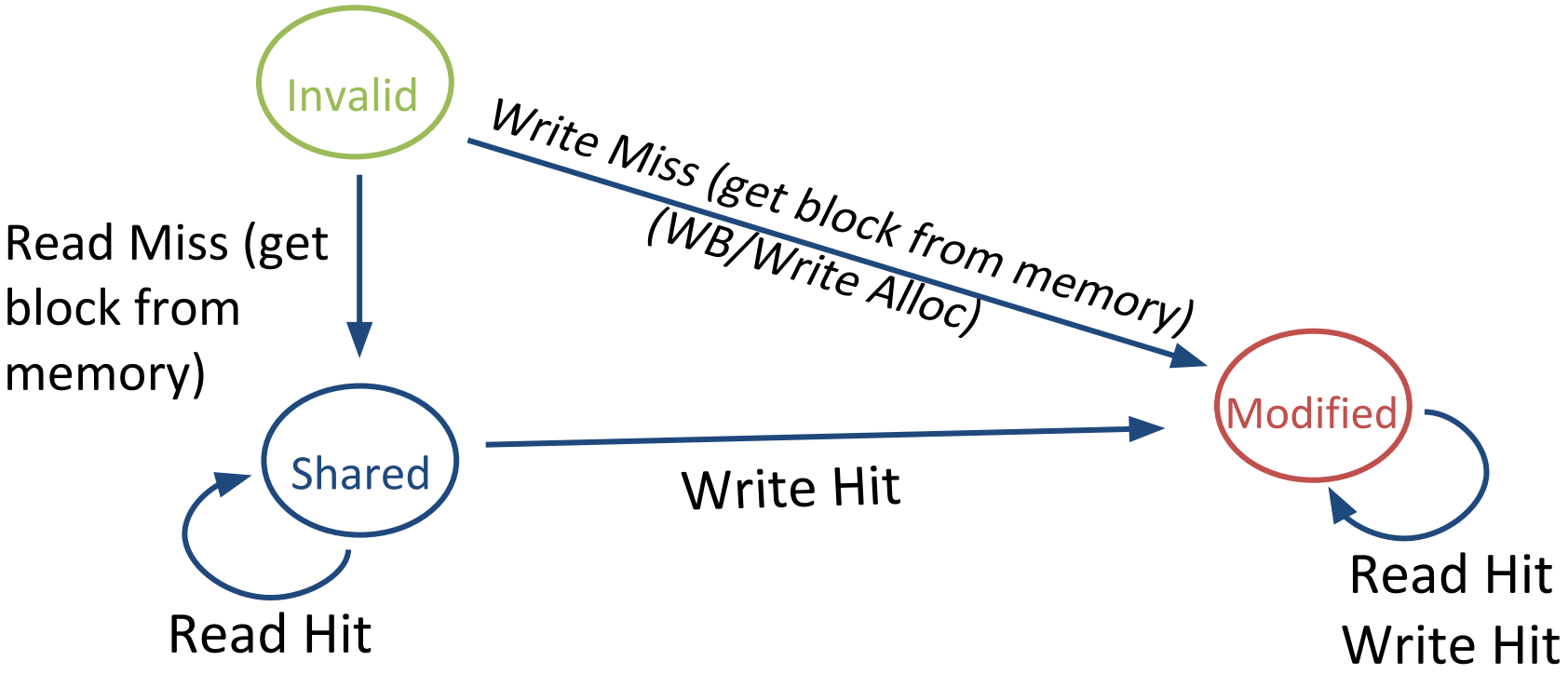
  (1) **Cache coherence protocol**: system for marking what state a block in the cache is in

  (2) **Snooping**: processors broadcast their intentions over an interconnection network, so other processors know when to invalidate their data

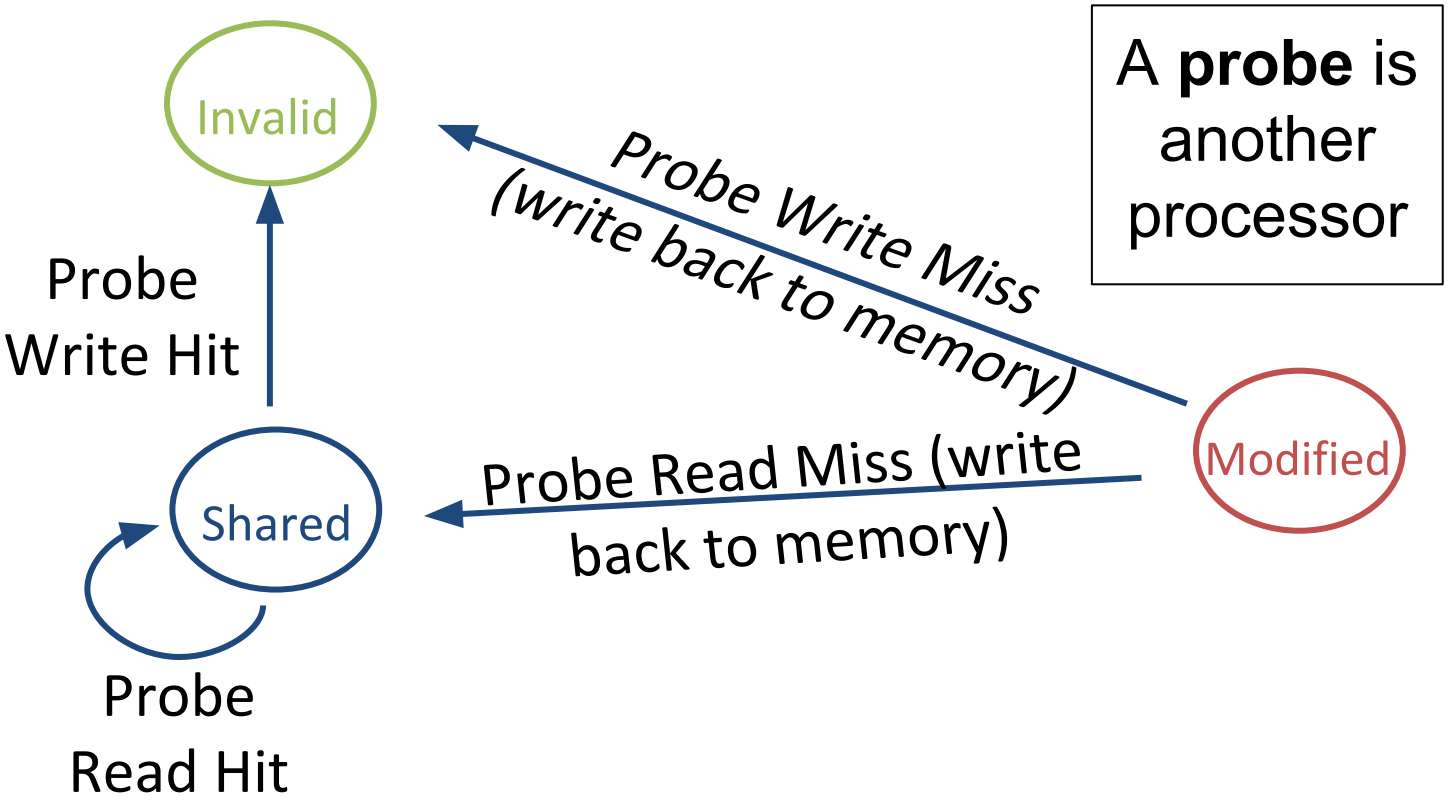
MSI

- Simplest cache coherence protocol
- Three states:
  - **Modified**: up-to-date, changed (dirty), OK to write
    - no other cache has a copy
    - copy in memory is out-of-date
    - must respond to read request by other processors by writing back to memory and moving to shared
  - **Shared**: up-to-date data, not allowed to write
    - other caches may have a copy
    - copy in memory is up-to-date
  - **Invalid**: data in this block is “garbage”

MSI Protocol: Current Processor



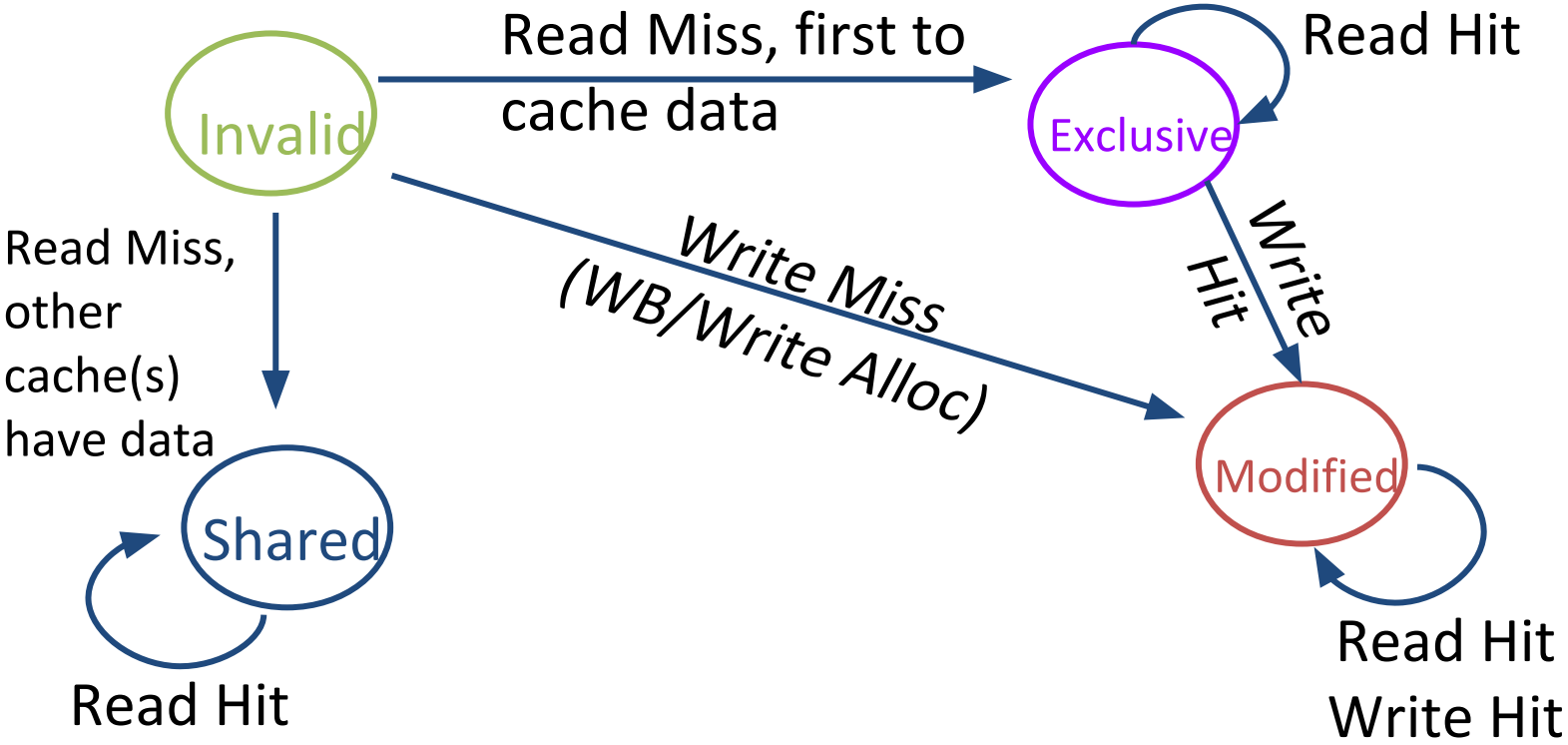
MSI Protocol: Response to Other Processors



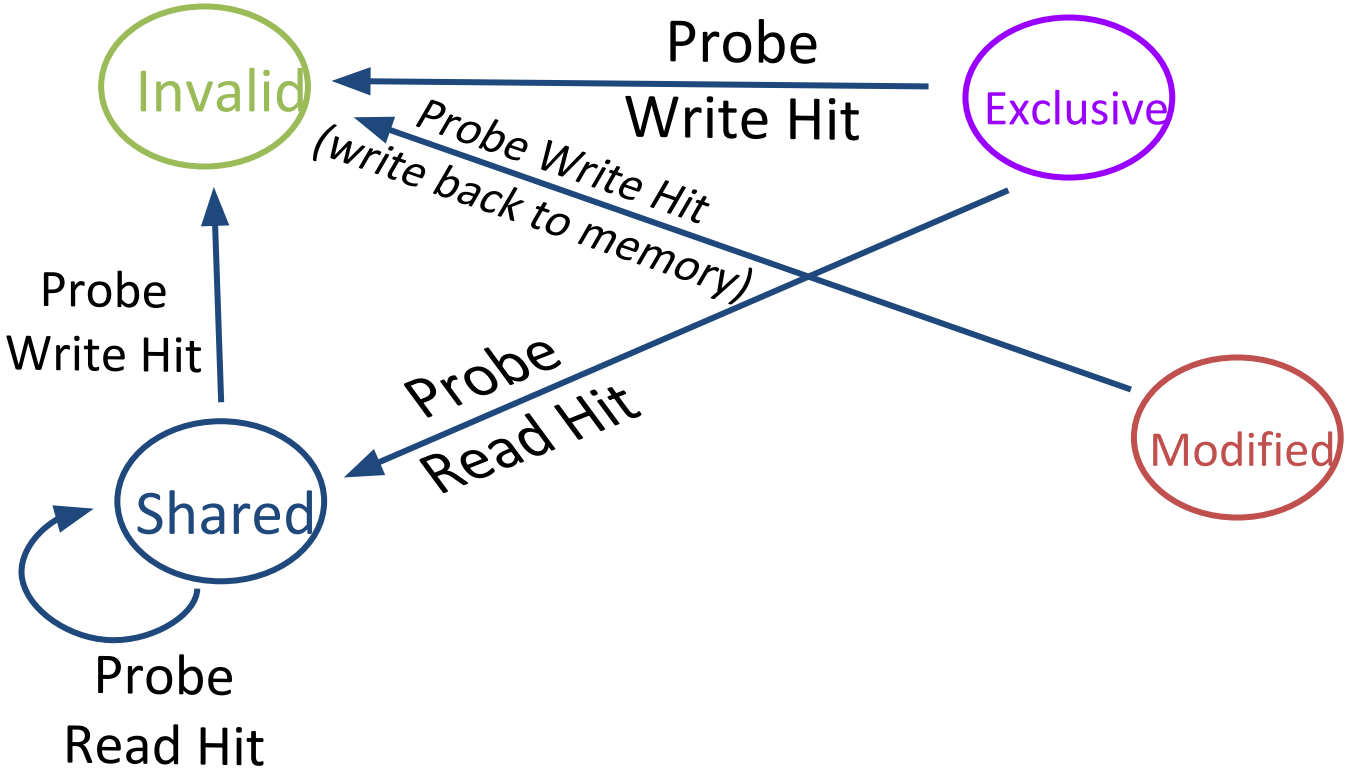
MESI

- **Problem:** in MSI, if we want to write to a block in Shared, we need to send an invalidation signal out on the interconnection network to every other processor
  - What if we have 100s of other processors, and this is the only cache with the data? We sent 100s of unnecessary invalidation signals, and this is expensive
- **Solution:** add the Exclusive state: up-to-date data, OK to write (after writing, change to modified)
  - no other cache has a copy
  - copy in memory up-to-date
  - no write to memory if block replaced
  - supplies data on read instead of going to memory
- Now, if block is in shared, at least 1 other cache must contain it:
  - **Shared**: up-to-date data, not allowed to write
    - other caches ~~may~~ definitely have a copy
    - copy in memory is up-to-date

MESI Protocol: Current Processor



MESI Protocol: Response to Other Processors

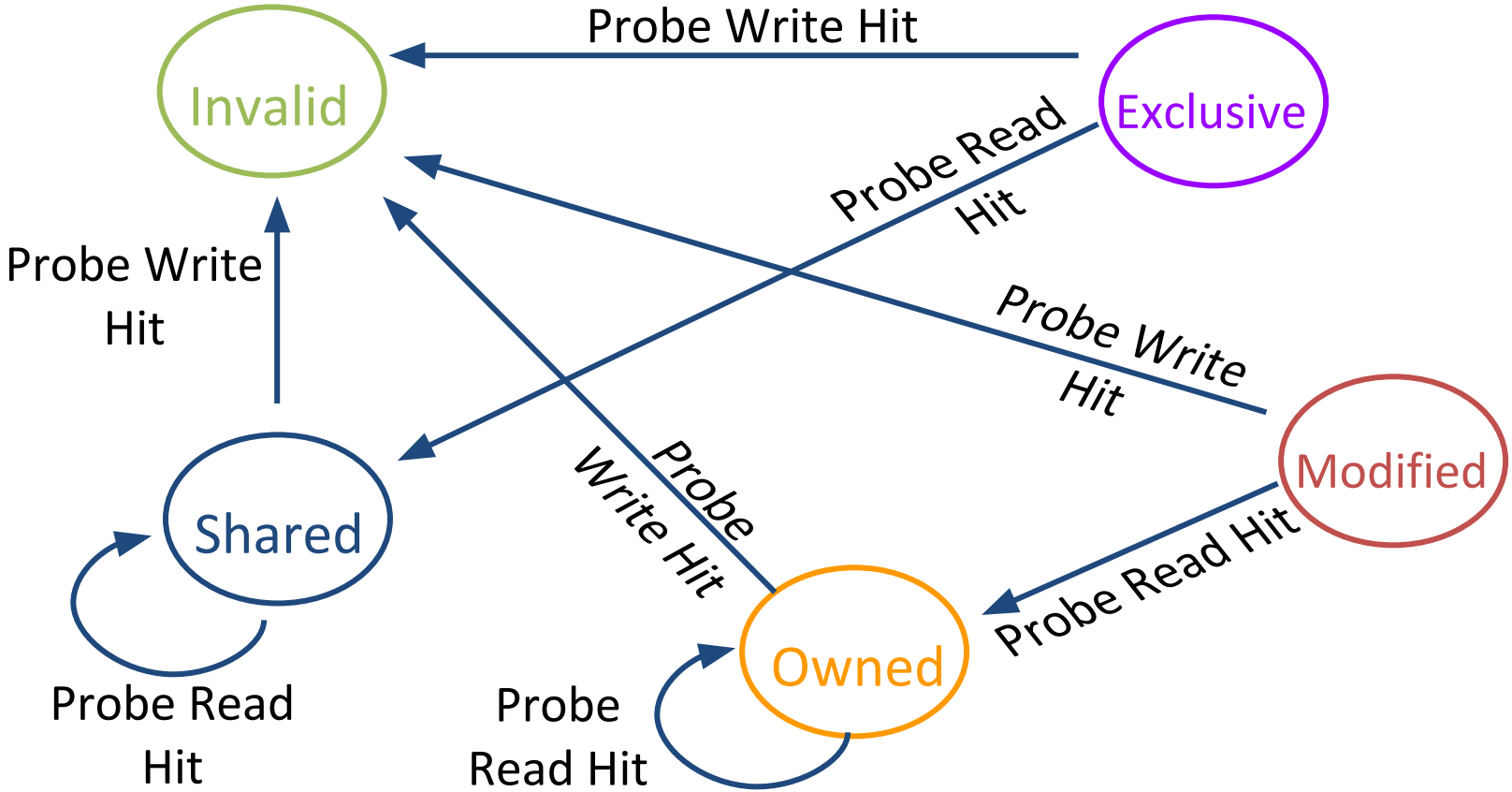


MOESI

- **Problem:** if a processor wants to write, it always needs to
- **Solution:** Owned state
  - Owner: up-to-date data, read-only (like shared, you can write if you invalidate shared copies first and your state changes to modified)
    - Other caches have a shared copy (Shared state)
    - Data in memory not up-to-date
    - Owner supplies data on probe read instead of going to memory
  - **Shared**: up-to-date data, not allowed to write
    - other caches definitely have a copy
    - copy in memory ~~is~~ *may* be up-to-date

MOESI Protocol: Current Processor

MOESI Protocol: Response to Other Processors

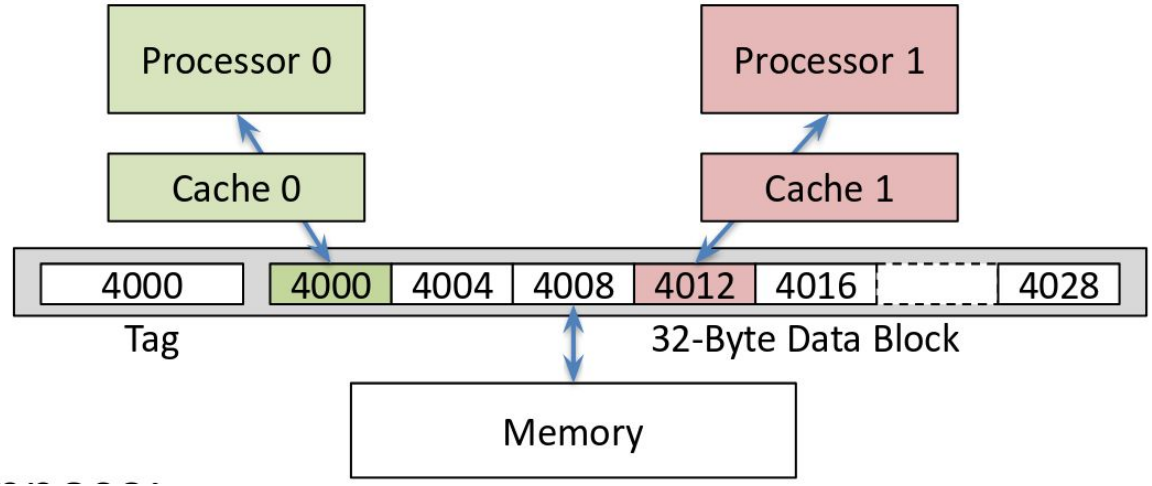


Writing to shared: clarification

- In MOESI, why is there no arrow from Shared corresponding to a write hit?
  - If a cache is holding a block in the Shared state, it does not have permission to write to it, so a cache holding a block in the Shared state *can't* have a write hit
  - The block must move to Invalid then to Modified, and it needs to send an invalidation signal to all other caches holding that block
  - But the Owned state is not useless! It is quite helpful when probes are only reading

False Sharing

- Ping-ponging caused by two different processors writing to **different parts** of the **same block**



False Sharing: Practice

```
#include <stdio.h>
#include "omp.h"
void count_eo (int *A, int size, int threads) {
    int result[2] = {0, 0};
    int i,j;

    omp_set_num_threads(threads);

    #pragma omp parallel for
    for (j=0; j<size; j++)
        result[(A[j] % 2 == 0) ? 0 : 1] += 1;

    printf("Even: %d\n", result[0]);
    printf("Odd: %d\n", result[1]);

}
```

Can there be false sharing with block size 8B?

False Sharing: Practice

```
#include <stdio.h>
#include "omp.h"
void count_eo (int *A, int size, int threads) {
    int result[2] = {0, 0};
    int i,j;

    omp_set_num_threads(threads);

    #pragma omp parallel for
    for (j=0; j<size; j++)
        result[(A[j] % 2 == 0) ? 0 : 1] += 1;

    printf("Even: %d\n", result[0]);
    printf("Odd: %d\n", result[1]);

}
```

Yes! If the pointer to result starts on a block boundary, and we have 2+ threads, if CPU1 and CPU2 are working on this code, there will be false sharing if CPU1 writes to result[0] at the same time CPU2 is trying to write to result[1]

Can there be false sharing with block size 8B?

False Sharing: Practice

```
#include <stdio.h>
#include "omp.h"
void count_eo (int *A, int size, int threads) {
    int result[2] = {0, 0};
    int i,j;

    omp_set_num_threads(threads);

    #pragma omp parallel for
    for (j=0; j<size; j++)
        result[(A[j] % 2 == 0) ? 0 : 1] += 1;

    printf("Even: %d\n", result[0]);
    printf("Odd: %d\n", result[1]);

}
```

Can there be false sharing with block size 4B?

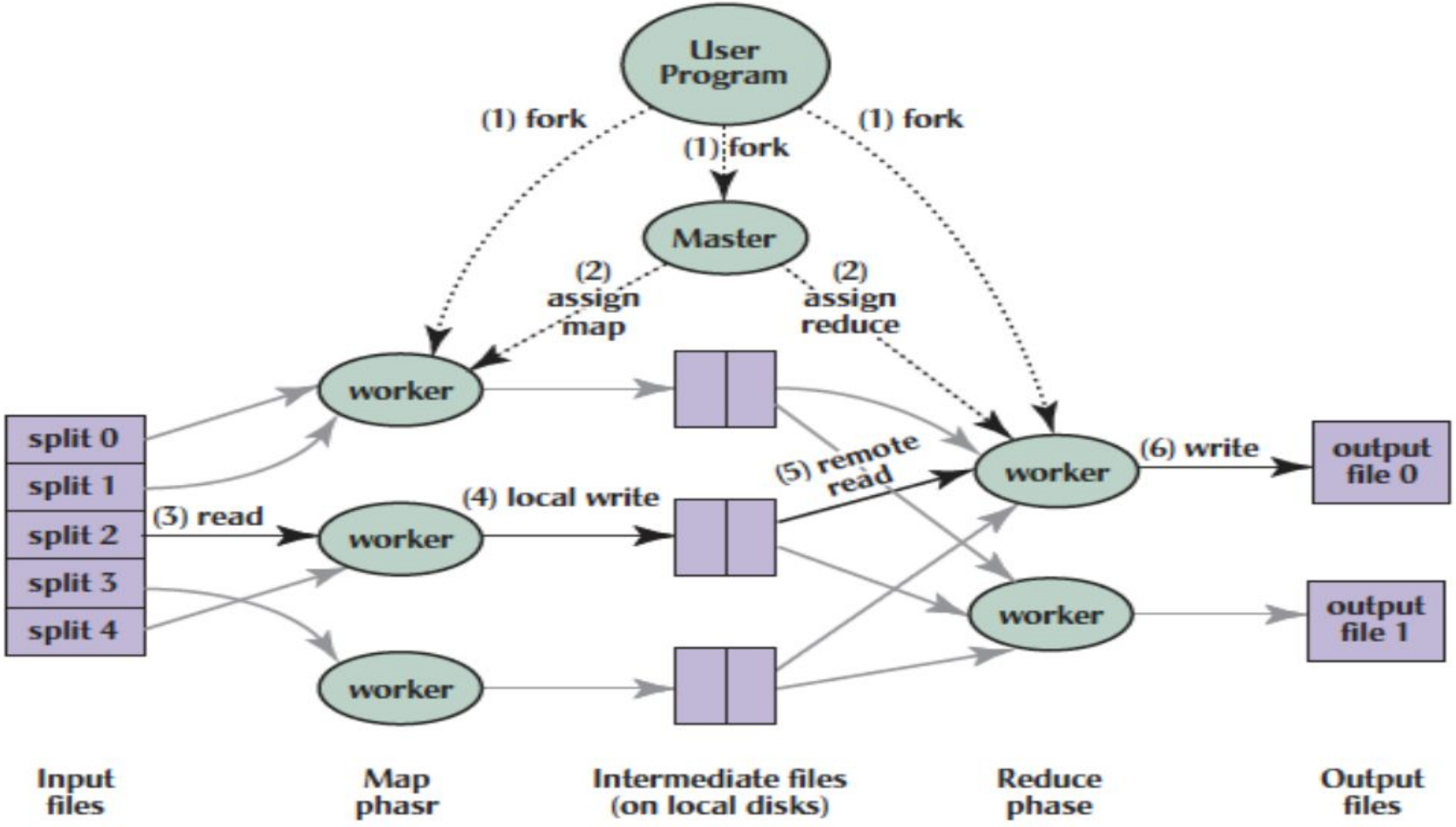
False Sharing: Practice

```
#include <stdio.h>
#include "omp.h"
void count_eo (int *A, int size, int threads) {
    int result[2] = {0, 0};
    int i,j;

    omp_set_num_threads(threads);

    #pragma omp parallel for
    for (j=0; j<size; j++)
        result[(A[j] % 2 == 0) ? 0 : 1] += 1;

    printf("Even: %d\n", result[0]);
    printf("Odd: %d\n", result[1]);

}
```

No! In this case, only one index of the array can fit into a block. If both processors are trying to write to the same index at the same time, this is **NOT** false sharing--it is **true sharing**.

Can there be false sharing with block size 4B?

Performance Programming: MapReduce & Spark


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phasr
Intermediate files (on local disks)
Reduce phase
Output files

MapReduce

- Map: (start_key, start_val) -> (key, val) or list(key,val)
  - Slice data for workers
  - Maps one key,value pair to 0 or more intermediate key value pairs

- Combine: list(key,val) -> (key, list(val))
  - Combines values with the same key
  - These tasks are distributed behind the scenes by the master node

- Reduce: (key, list(val)) -> (key, aggregated val)
  - Takes a key and a list of values that had the key as input
  - Combines the separate values to come to a final solution set

MapReduce

- The goal of MapReduce is to parallelize data processing on large datasets
- Sometimes a combination of many maps and reduces are necessary

Fall 15 Final

Imagine we're looking at Facebook's friendship graph, which we model as having a vertex for each user, and an undirected edge between friends. Facebook stores this graph as an adjacency list, with each vertex associated with the list of its neighbors, who are its friends. This representation can be viewed as a list of degree 1 friendships, since each user is associated with their direct friends. We're interested in finding the list of degree 2 friendships, that is, an association between each user and the friends of their direct friends.

You are given a list of associations of the form `(user_id, list(friend_id))`, where the `user_id` is 1st degree friends with all the users in the list.

Your output should be another list of associations of the same form, where the first item of the pair is a `user_id`, and the second item is a list of that user's 2nd degree friends. **Note:** a user is not their own 2nd degree friend, so the list of second degree friends must not include the user themselves.

Write pseudocode for the mapper and reducer to get the desired output from the input. Assume you have a set data structure, with `add(value)` and `remove(value)` methods, where `value` can be an item or a list of items. You can iterate through a list with the `for item in items` construct. You may not need all the lines provided.

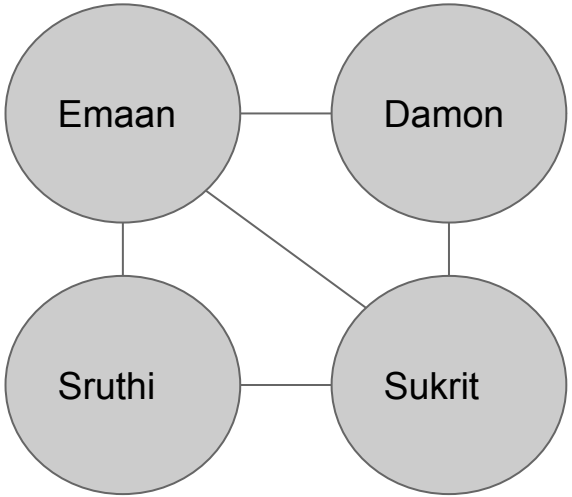
Fall 15 Final

Damon: [Emaan, Sukrit]

Sruthi: [Emaan, Sukrit]

Emaan: [Damon, Sruthi, Sukrit]

Sukrit: [Damon, Emaan, Sruthi]



Damon & Sruthi are 2nd degree friends

Want to emit: (Sruthi, [Damon, Sukrit, Emaan])

Fall 15 Final

```
map(user_id, friend_ids):
        for ____________________:
                emit(_______________, _________________)
reduce(key, values):
      second_degree_friends = set()
        ____________________________________
        ____________________________________
        ____________________________________
        ____________________________________
        ____________________________________
        emit(_______________, _________________)
```

Fall 15 Final

```
map(user_id, friend_ids):
        for friend_id in friend_ids:
            emit(friend_id, friend_ids)
reduce(key, values):
    second_degree_friends = set()
    for friend_ids in values:
        for friend_id in friend_ids:
            second_degree_friends.add(friend_id)
    second_degree_friends.remove(key)
    emit(key, list(second_degree_friends))
```

Spark

- Another implementation of the MapReduce programming model
- Start by creating a parallelizable collection from data using sc.parallelize(data); this creates an RDD (“resilient distributed dataset”)
  - Data should be an iterable or collection
- 2 types of operations on RDD:
  - Transforms: RDD → RDD
  - Actions: RDD → Value

Transforms & Actions

Transforms: (RDD → RDD)

- **map(func):** return new RDD by passing every element through func; each element returns exactly one element
- **flatMap(func):** return a new RDD by passing every element through func; each element returns 0 or more elements
- **reduceByKey(func):** aggregates elements by key, by passing two elements at a time through func (func must take in v,v and return v)
- **sortByKey():** sort the RDD by key

Actions: (RDD → value)

- **reduce(func):** aggregates elements regardless of key to produce a value

Second Degree Spark

```
def flatMapFunc(person, friendIDs):
      return ____________________________________________
def reduceFunc(__________________, __________________):
      return ____________________________________________
def mapFunc(__________________, __________________):
      ____________________________________________________
      return ____________________________________________

# persons = list((person, list(friendIDs))
secondDegree = sc.parallelize(persons)
                              .flatMap(lambda (k, v): flatMapFunc(k, v))
                              .reduceByKey(lambda (v, v): reduceFunc(v, v))
                              .map(lambda (k, v): mapFunc(k, v))

return secondDegree
```

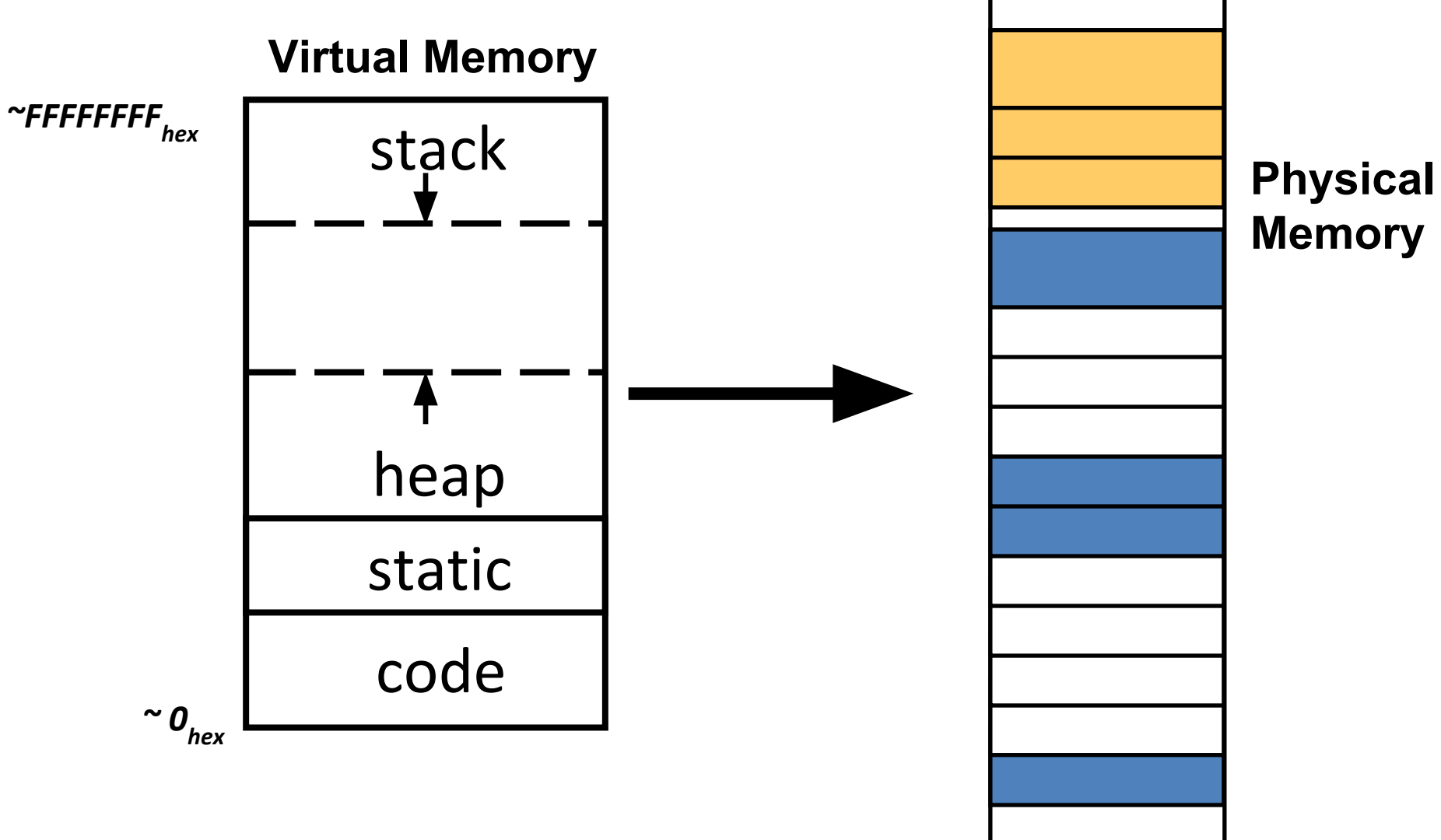
Second Degree Spark

```
def flatMapFunc(person, friendIDs):
      return [(friend, set(friendIDs)) for friend in friendIDs]
def reduceFunc( friendIDs_1, friendIDs_2 ):
      return friendIDs_1.union(friendIDs_2)
def mapFunc( person, friendSet ):
      friendSet = friendSet.remove(person)
      return (person, friendSet)

## persons = list((person, list(friendIDs))
secondDegree = sc.parallelize(persons)
                              .flatMap(lambda (k, v): flatMapFunc(k, v))
                              .reduceByKey(lambda (v, v): reduceFunc(v, v))
                              .map(lambda (k, v): mapFunc(k, v))

return secondDegree
```

Break Time!!!

Take a 5 minute break to relax, wiggle around a bit before we get back into the nitty gritty of things.

Virtual Memory


Virtual Memory
~FFFFFFFF hex
stack
heap
static
code
~ 0 hex
Physical Memory

Why Virtual Memory?

Recall from discussion:

- **Protection** between processes
- Abstract program address space (simulate **full address space**)
- Adds **disk** to the memory hierarchy

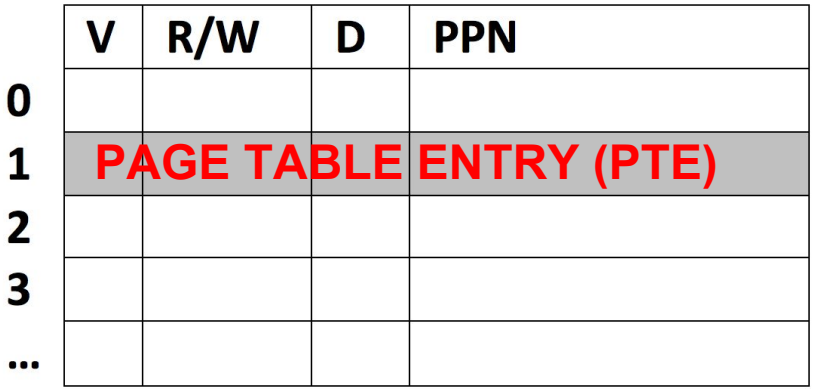
Page Table

- **Page:** contiguous block of memory
- Maps Virtual Page Numbers (VPN) to Physical Page Numbers (PPN)
- Direct-Mapped
- **Page Fault:** Miss in the page table, means the page is not in memory
- Page Tables are stored in memory
- Page tables only store mappings

	V	R/W	D	PPN
0				
1	PAGE TABLE ENTRY (PTE)			
2				
3				
...				

Linear Page Tables

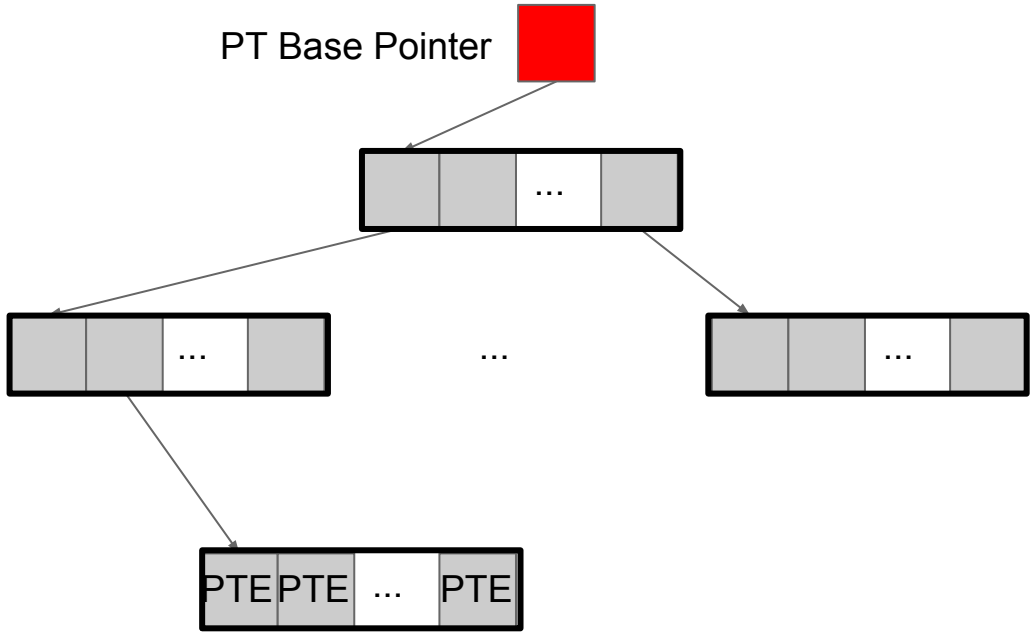
- Stores all PTEs in one big array
- Needs 1 entry for every virtual page



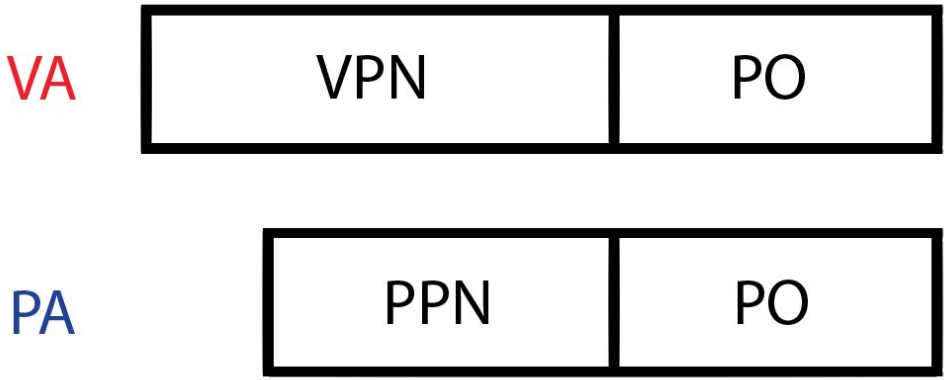
Hierarchical PTs

- Most PTEs are invalid (processes rarely use all of virtual memory)
- What is a good, searchable data structure for sparse values?
  - Tree!

- Hierarchical PT: B-Tree
  - Each node has N children
  - Leaf entries are PTEs



Mapping Virtual To Physical

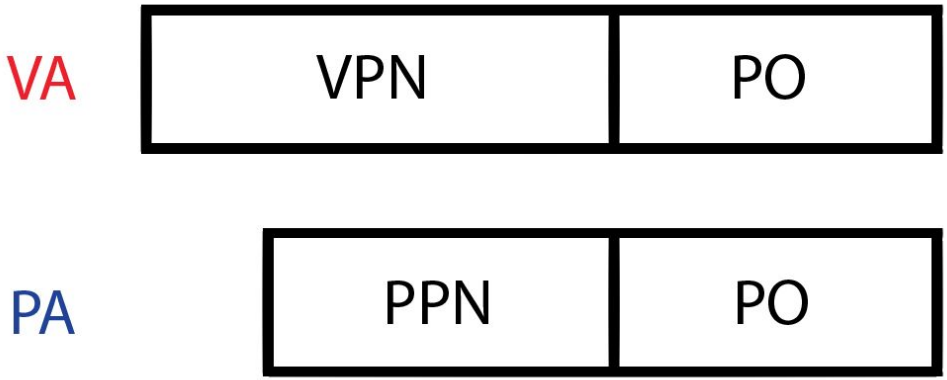


P.O. (Page Offset) = $\log_2$(page size)

V.A. (Virtual Address) bits = $\log_2$(VA space)

P.A. (Physical Address) bits = $\log_2$(PA space)

Mapping Virtual To Physical



Does VA > PA, VA == PA, VA < PA, etc.?

No. It really depends. There are different reasonings for why there are different variations in how large of a gap there is between VA and PA is in respect to another.

Virtual & Physical Addresses

Page size = 4 KiB

Virtual address space = 4 GiB

Physical address space = 512 MiB

PO bits?

VPN bits?

PPN bits?

Virtual & Physical Addresses

Page size = 4 KiB

Virtual address space = 4 GiB

Physical address space = 512 MiB

PO bits? $\log_2(4Ki) = 12$

VPN bits? $\log_2(4Gi) - 12 = 32 - 12 = 20$

PPN bits? $\log_2(512Mi) - 12 = 29 - 12 = 17$

Page Table

Page size = 4 KiB

Virtual address space = 4 GiB

Physical address space = 512 MiB

PO bits = 12

VPN bits = 20

PPN bits = 17

How big would a linear page table be?

(Assume there are valid, dirty, read, write bits)

How many PTEs can be valid at a time?

Page Table

Page size = 4 KiB

Virtual address space = 4 GiB

Physical address space = 512 MiB

PO bits = 12; VPN bits = 20; PPN bits = 17

How big is the page table?

PTE bits =1(V) + 2(R/W) + 1(D) + 17(PPN) =21

PT size = 2^{20} * 21 bits

How many PTEs can be valid at a time?2^{17}

TLB

- “Cache for Page Table”
- Fully associative, LRU Replacement
- Miss: TLB miss - go to PT

VPN	V	R/W	D	PPN

TLB

Page size = 4 KiB

Virtual address space = 4 GiB

Physical address space = 512 MiB

PO bits = 12; VPN bits = 20; PPN bits = 17

32 entry TLB

How big is the TLB? (Assume there are valid, dirty, read, write bits)

How many addresses can we hold?

TLB

Page size = 4 KiB

Virtual address space = 4 GiB

Physical address space = 512 MiB

PO bits = 12; VPN bits = 20; PPN bits = 17

32 entry TLB

How big is the TLB?

LRU bits = $\log_2(32) = 5$

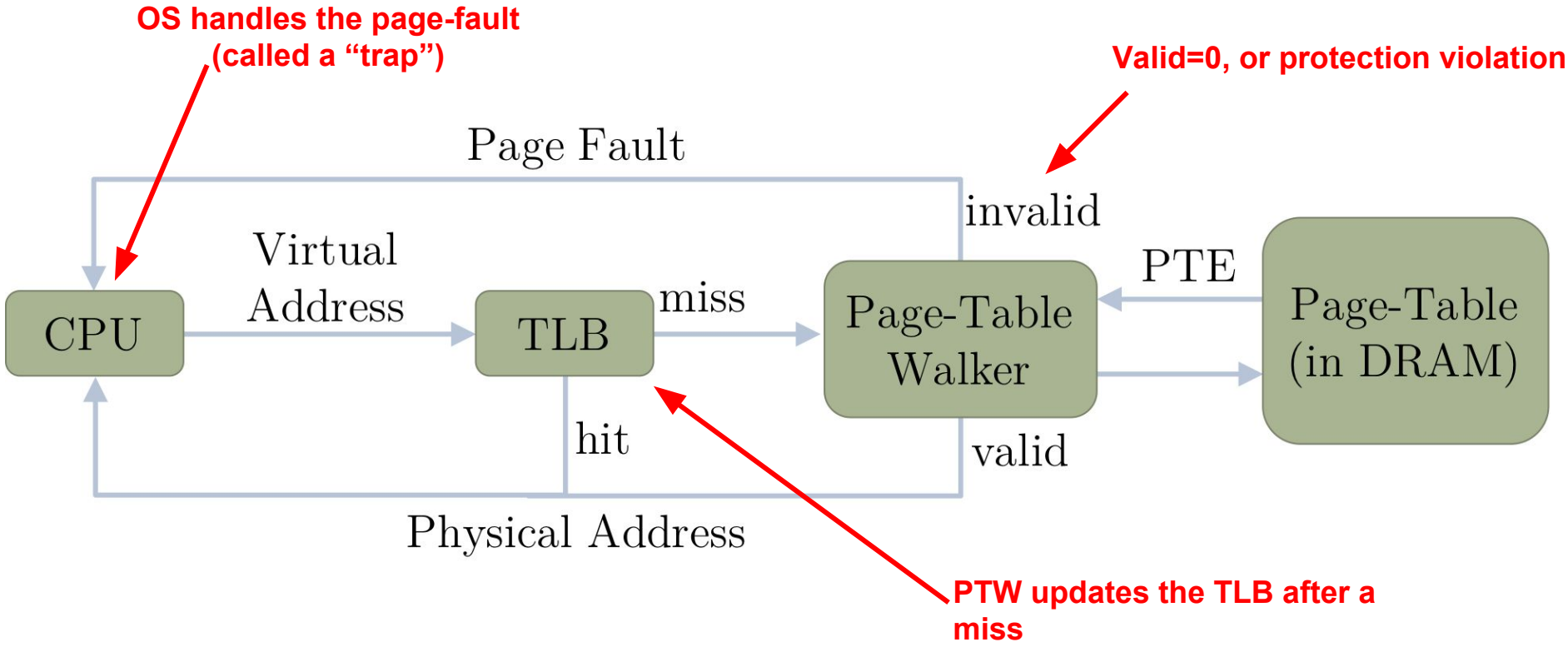
TLBE bits = 20 + 4 + 17 + 5 = 46

TLB size = 32 * 46 bits

How many addresses can we hold?

32 * 4 KiB = 128 KiB (32 pages of 4KiB each)

Anatomy of Address Translation

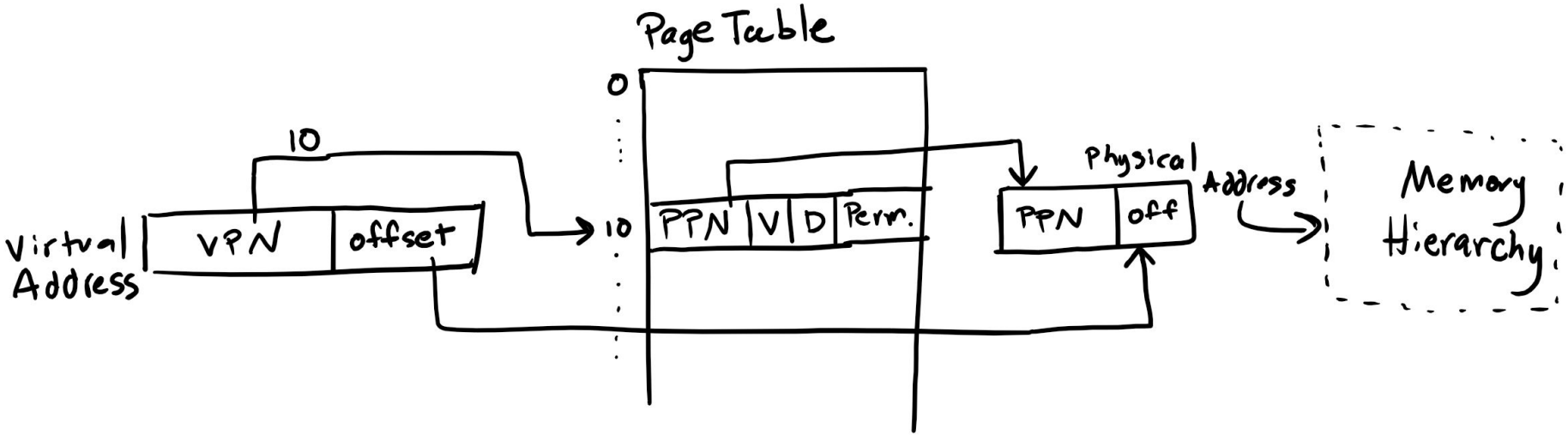


What happens in a Page Fault?

Reasons for a Miss:

- Page never existed
  - OS just gives a new physical frame
- Page is on disk (it's been paged out)
  - OS must read page from disk into a new physical frame
- Protection Violation
  - OS fixes problem, or kills program
  - E.G. SegFault!

Turning a PTE into a Physical Address



Multiprogramming

- Each process has its own **Page Table**
- **Page Table Base Register** points to the Page Table of the process currently running
- TLB invalidated when swapping processes
  - Usually, but not on all architectures

Virtual Memory (Fall 2006 F2)

The specs for a 32-bit RISC-V (RV32) machine's memory system that has one level of cache and virtual memory are:

- 1MiB of Physical Address Space
- 4GiB of Virtual Address Space
- 4KiB page size
- 16KiB 8-way set-associative write-through cache, LRU replacement
- 1KiB Cache Block Size
- 2-entry TLB, LRU replacement

Question

- How many bits for Tag, index, and offset in the data cache?

- How many bits for VPN and page offset?

Answer

- Block offset: log(1 KiB) = 10 bits
- Index: 16 KiB/(8 way x 1 KiB) = 2 rows, Index = 1 bit
- Tag: log(1 MiB)-0-1 = 20-10-1 = 9 bits

- Page Offset: log(4 KiB) = 12 bits
- VPN: log(4 GiB) - Page Offset = 20 bits

Now pretend this code gets run

```
#define NUM_INTS 8192 // 2^13
int *A = (int *) malloc(NUM_INTS * sizeof(int));
int i, total = 0;
for(i = 0; i < NUM_INTS; i += 128) A[i] = i;
for(i = 0; i < NUM_INTS; i += 128) total += A[i]; //Special
```

Calculate the Hit Rate for the cache and TLB only for the line //special

Calculate the Hit Rate for the cache and TLB for both loops

Cache hit Rate?

- Block size of 1 KiB
- Each block is valid for 2 iterations because 128*4 = 512, so we jump 512 bytes every iteration
- Every other iteration is a miss
- Array Size is 2^{13} x 2^{2} = 2^{15} bytes, which is double the size of the cache, so no useful residue from first loop
- Hit Rate: 50%

TLB Hit Rate?

- Page size of 4 KiB
- TLB holds 8 KiB of addresses, which is much smaller than the array size
- Each page loaded in is valid for 8 iterations because 4 KiB/(128*4 B) = 8
- TLB miss once every 8 iterations
- Hit rate: 7/8

Page Table Hit Rate?

First Loop only?

Second Loop?

Page Table Hit Rate?

First Loop only?

Depends on the initial state…

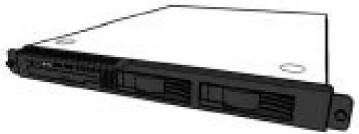
Second Loop?

100%

By the time the first loop finishes, all the pages needed for the second loop are already loaded into the page table!

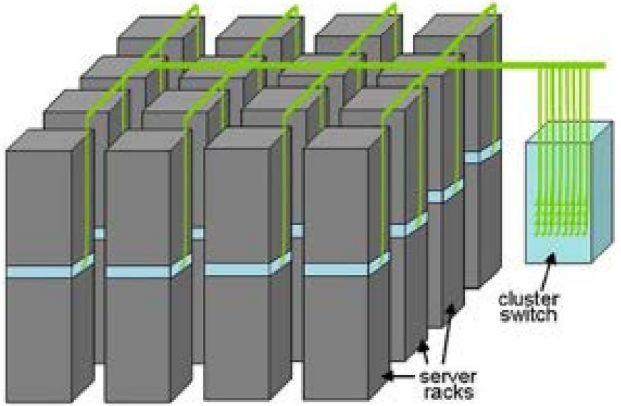
WSC, I/O, RAID, ECC

Warehouse Scale Computers



1U Server:
8 cores,
16 GB DRAM,
4x1 TB disk

Rack:
40-80 severs,
Local Ethernet (1-10Gbps) switch
(30$/1Gbps/server)



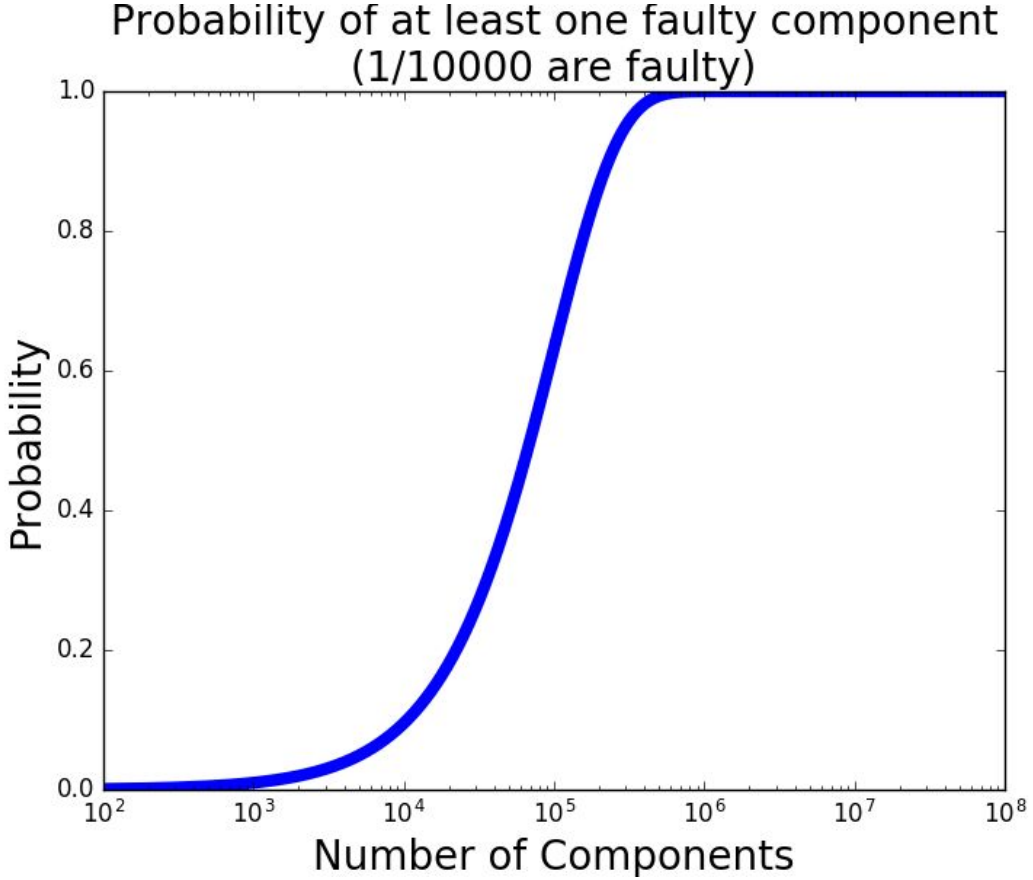
Array (aka cluster):
16-32 racks
Expensive switch
(10X bandwidth → 100x cost)

PUE = $\frac{\text{Total Power}}{\text{IT Power}}$

Dependability Measures

- Mean Time To Failure (MTTF)
- Mean Time To Repair (MTTR)
- Mean time between failures (MTBF): MTTF + MTTR
- Availability = MTTF / (MTTF + MTTR) x 100%
  - Usually only concerned with “number of 9’s”
  - 90% = 1 nine, 99% = 2 nines, 99.9% = 3 nines, etc.
- Annualized Failure Rate: Avg. # of failures per year

Dependability and Parallelism



I/O Strategies

- Memory Mapped I/O
  - Certain ranges of memory don't actually address main memory, instead they point to I/O devices
- Polling
  - Keep checking if data is available, if so do read
  - bad if time spent polling bogs down CPU performance
- Interrupts
  - Register a function to be called when data is available
  - high overhead

Questions

- List 1 “pro” and 1 “con” for:

- Polling
  - Pro:
  - Con:

- Interrupts
  - Pro:
  - Con:

Questions

- List 1 “pro” and 1 “con” for:

- Polling
  - Pro: Low latency
  - Con: Can’t do other work while polling

- Interrupts
  - Pro: Doesn’t require constant CPU work
  - Con: Takes longer to handle event
    - Also, adds overhead to each event

Operation of a DMA Transfer

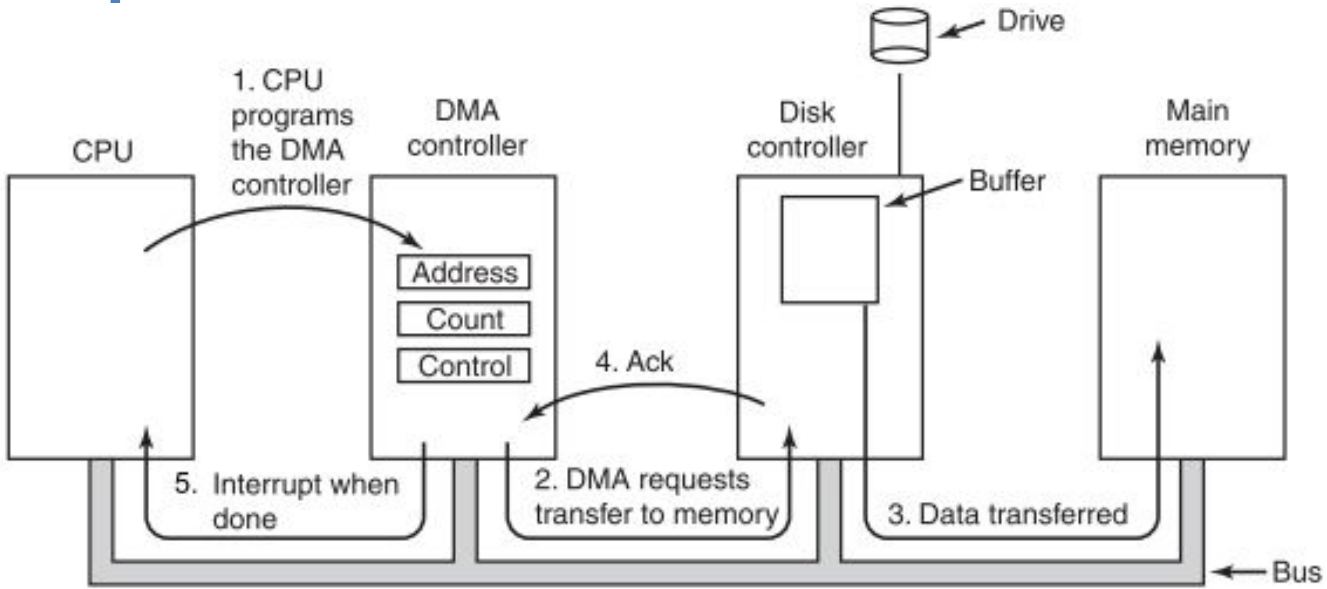


Figure 5-4. Operation of a DMA transfer.

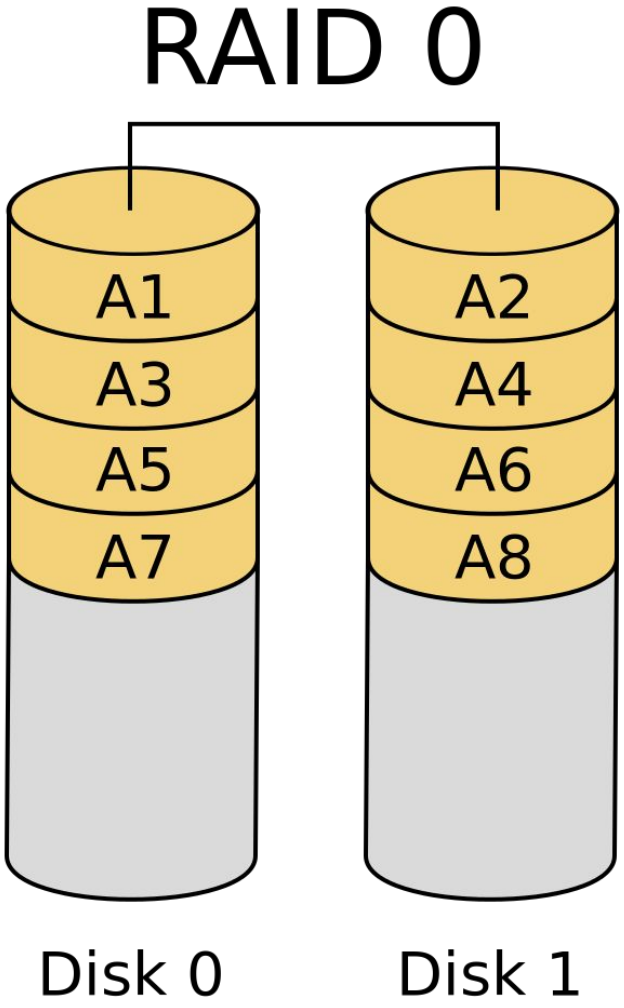
[From Section 5.1.4 Direct Memory Access in *Modern Operating Systems* by Andrew S. Tanenbaum, Herbert Bos, 2014]

DMA: Incoming Data

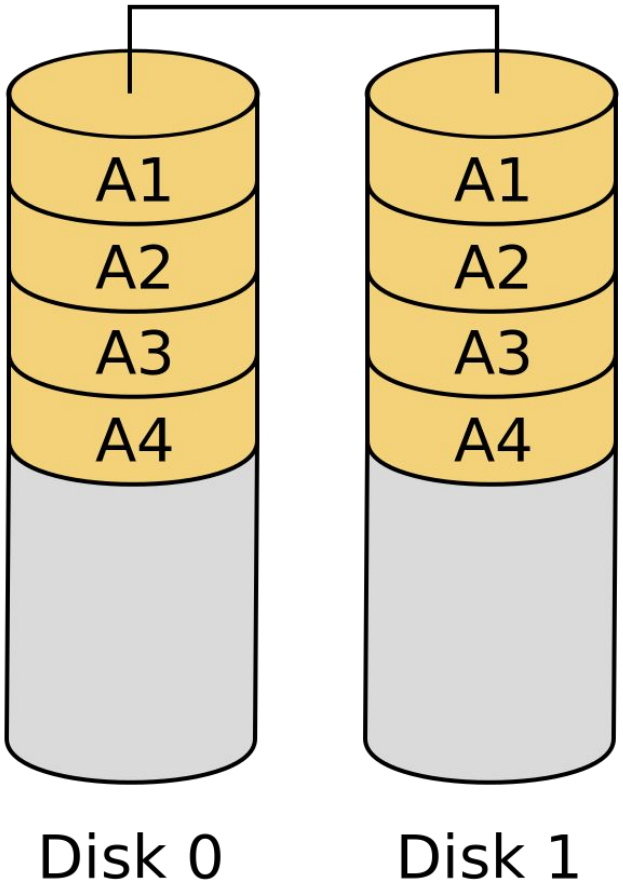
1. Receive interrupt from device
2. CPU takes interrupt, begins transfer
   - Instructs DMA engine/device to place data @ certain address
3. Device/DMA engine handle the transfer
   - CPU is free to execute other things
4. Upon completion, Device/DMA engine interrupt the CPU again

DMA: Outgoing Data

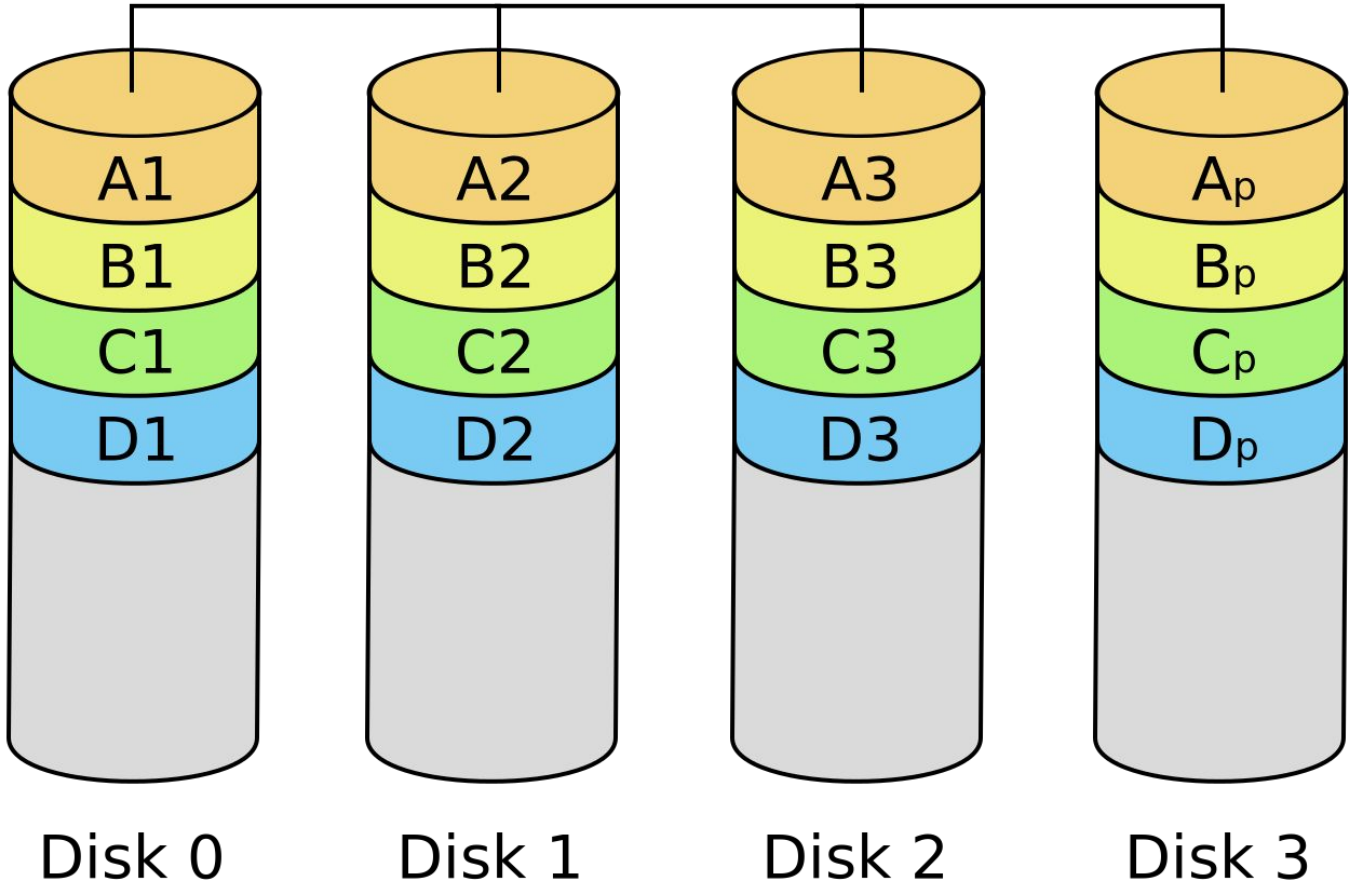
1. CPU decides to initiate transfer, confirms that external device is ready
2. CPU begins transfer
   - Instructs DMA engine/device that data is available @ certain address
3. Device/DMA engine handle the transfer
   - CPU is free to execute other things
4. Device/DMA engine interrupt the CPU again to signal completion

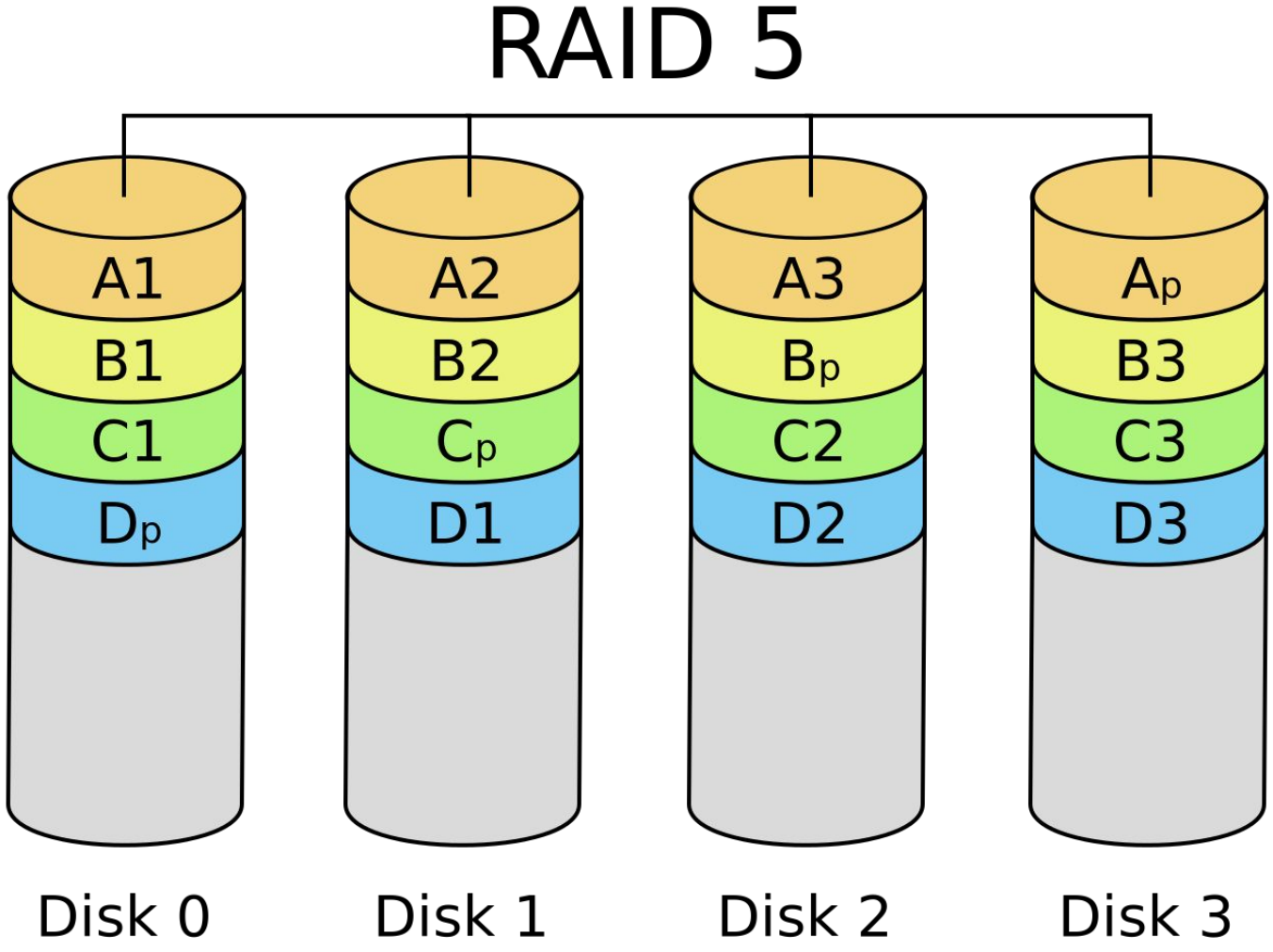

RAID 0
A1
A3
A5
A7
A2
A4
A6
A8
Disk 0
Disk 1

RAID 1



RAID 4


A1
B1
C1
D1
A2
B2
C2
D2
A3
B3
C3
D3
Ap
Bp
Cp
Dp
Disk 0
Disk 1
Disk 2
Disk 3


RAID 5
A1
B1
C1
Dp
A2
B2
Cp
D1
A3
Bp
C2
D2
Ap
B3
C3
D3
Disk 0
Disk 1
Disk 2
Disk 3

True/False

Fall 2015 Final

RAID 4 is fast for concurrent writes

RAID 5 is fast for concurrent writes

RAID 4 is fast for concurrent reads

RAID 0 is the most expensive RAID

True/False

Fall 2015 Final

RAID 4 is fast for concurrent writes False

RAID 5 is fast for concurrent writes True

RAID 4 is fast for concurrent reads True

RAID 0 is the most expensive RAID False

Error Detection and Correction

- Parity: Even/Odd numbers of 1's in the number
- 1 Parity Bit = 1-bit Error Detection
- Hamming ECC: Error Detection AND Correction

Bit	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Data	P1	P2	D1	P4	D2	D3	D4	P8	D5	D6	D7	D8	D9	D10	D11
P1	X		X		X		X		X		X		X		X
P2		X	X			X	X			X	X			X	X
P4				X	X	X	X					X	X	X	X
P8								X	X	X	X	X	X	X	X

ECC Question (sp16 Final)

ii. State the Hamming distance between the received message and the correct message, and give the corrected message if necessary and possible.

	p1	**p2**	d1	**p4**	d2	d3	d4	**p8**
	1	1	1	1	1	0	1	0
p1	X		X		X		X	
p2		X	X			X	X	
p4				X	X	X	X	

Hamming distance: Corrected message: or **Impossible**

ECC Question (sp16 Final)

ii. State the Hamming distance between the received message and the correct message, and give the corrected message if necessary and possible.

	p1	**p2**	d1	**p4**	d2	d3	d4	**p8**
	1	1	1	1	1	0	1	0
p1	X		X		X		X	
p2		X	X			X	X	
p4				X	X	X	X	

Hamming distance: `1`

Corrected message: `0b1111` or **Impossible**

Questions?

- MT1?
- MT2?
- Other things?

Good Luck!!