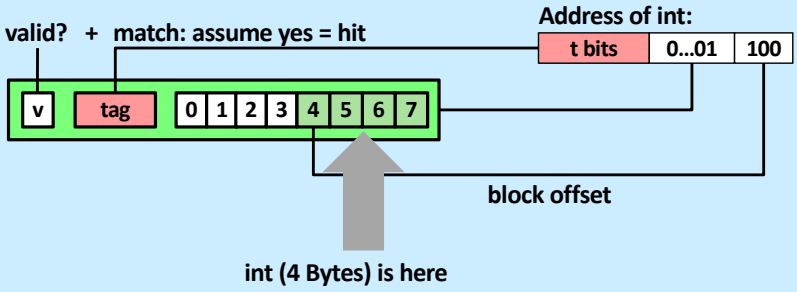
CS 33

Exploiting Caches

Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook "Computer Systems: A Programmer's Perspective," 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O'Hallaron in Fall 2010. These slides are indicated "Supplied by CMU" in the notes section of the slides.

Example: Direct Mapped Cache (E = 1)

Direct mapped: one line per set
Assume: cache block size 8 bytes



No match: old line is evicted and replaced

Supplied by CMU.

Conflict Misses: Aligned

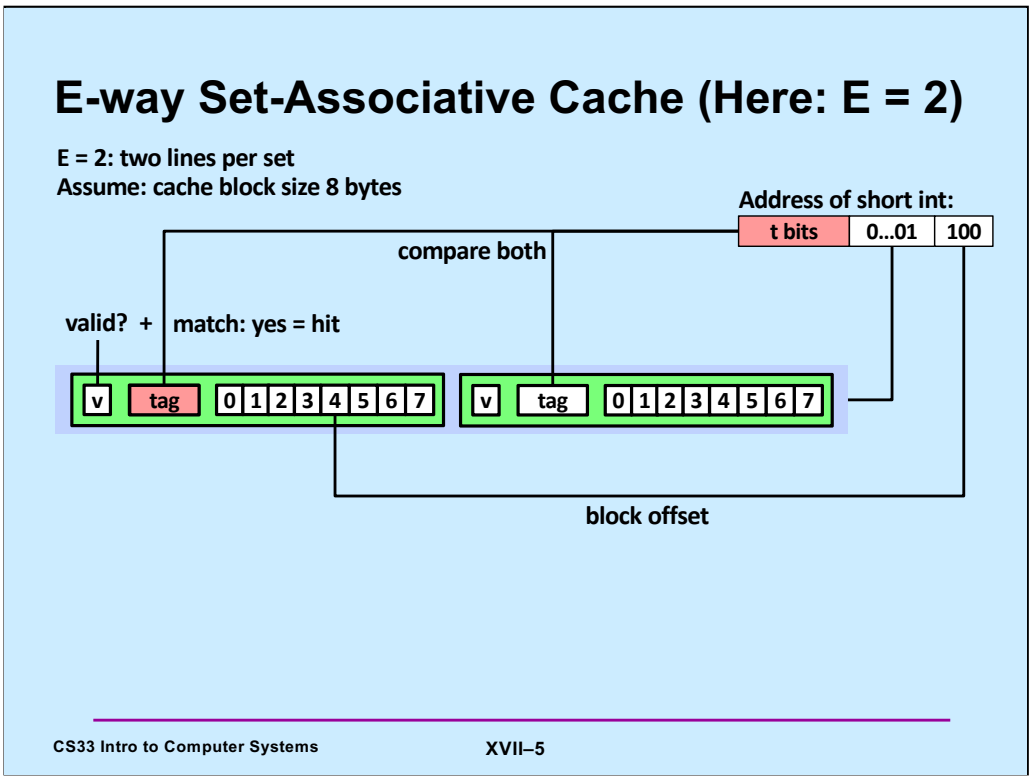
```
double dotprod(double x[8], double y[8]) {
  double sum = 0.0;
  int i;

  for (i=0; i<8; i++)
    sum += x[i] * y[i];

  return sum;
}
```

If arrays x and y have the same alignment, i.e., both start in the same cache set, then each access to an element of y replaces the cache line containing the corresponding element of x, and vice versa. The result is that the loop is executed very slowly — each access to either array results in a conflict miss.

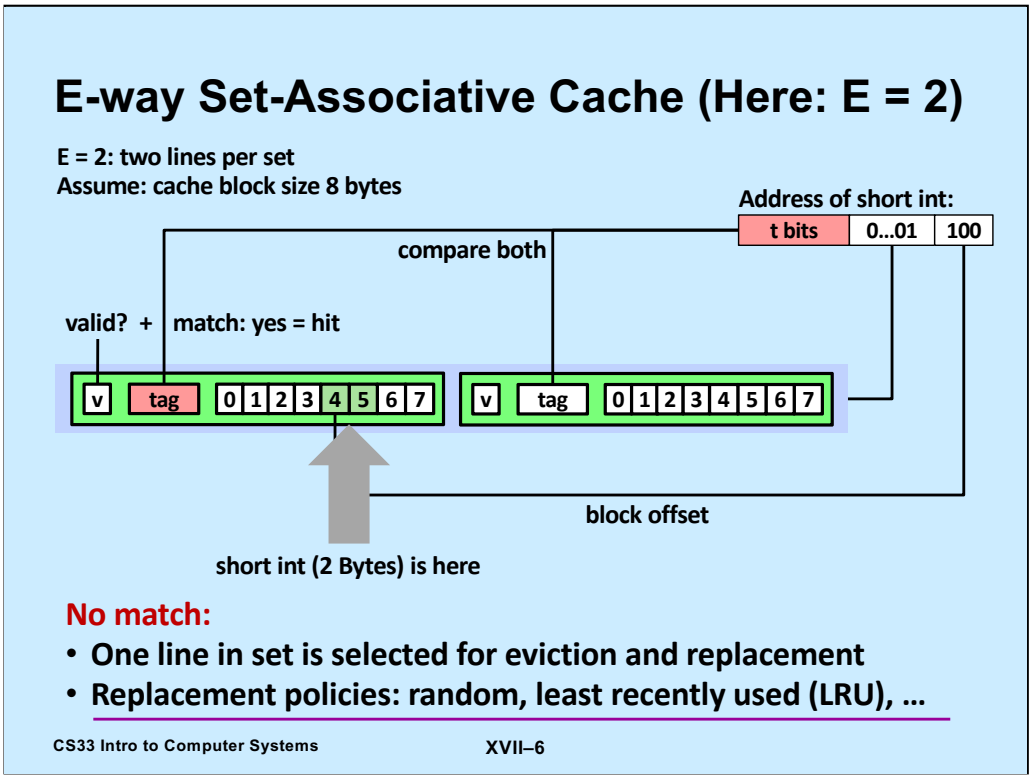
Different Alignments

```
double dotprod(double x[8], double y[8]) {
  double sum = 0.0;
  int i;

  for (i=0; i<8; i++)
    sum += x[i] * y[i];

  return sum;
}
```

However, if the two arrays start in different cache sets, then the loop executes quickly — there is a cache miss on just every fourth access to each array.



Supplied by CMU.



Supplied by CMU.

Quiz 1

Address of int:

100	01	100

0	v	tag=0	0	0	0	0	1	1	1	1	v	tag=2	2	2	2	2	3	3	3	3
1	v	tag=0	4	4	4	4	5	5	5	5	v	tag=4	6	6	6	6	7	7	7	7
2	v	tag=2	8	8	8	8	9	9	9	9	v	tag=3	a	a	a	a	b	b	b	b
3	v	tag=4	c	c	c	c	d	d	d	d	v	tag=a	e	e	e	e	f	f	f	f

Given the address above and the cache contents as shown, what is the value of the *int* at the given address?

a) 1111
b) 3333
c) 4444
d) 7777

2-Way Set-Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	**[0000$_2$],**	miss
1	**[0001$_2$],**	hit
7	**[0111$_2$],**	miss
8	**[1000$_2$],**	miss
0	**[0000$_2$]**	hit

	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

Supplied by CMU.

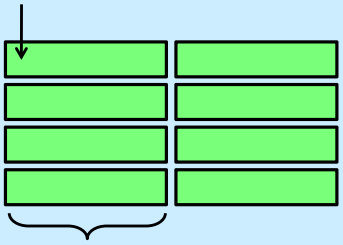
A Higher-Level Example

Ignore the variables sum, i, j

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; i++)
        for (i = 0; i < 16; j++)
            sum += a[i][j];
    return sum;
}
```

assume: cold (empty) cache,
a[0][0] goes here



32 B = 4 doubles

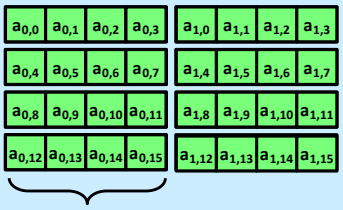
Supplied by CMU.

A Higher-Level Example

Ignore the variables sum, i, j

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```



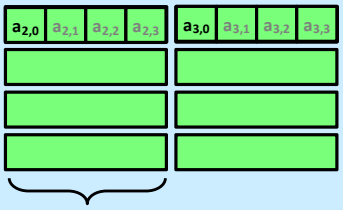
The cache still holds two rows of the matrix, but each row may go into one of two different cache lines. In the slide, the first row goes into the first lines of the cache sets, the second row goes into the second lines of the cache sets.

A Higher-Level Example

Ignore the variables sum, i, j

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

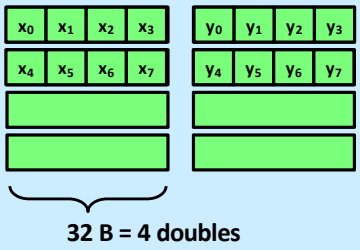


32 B = 4 doubles

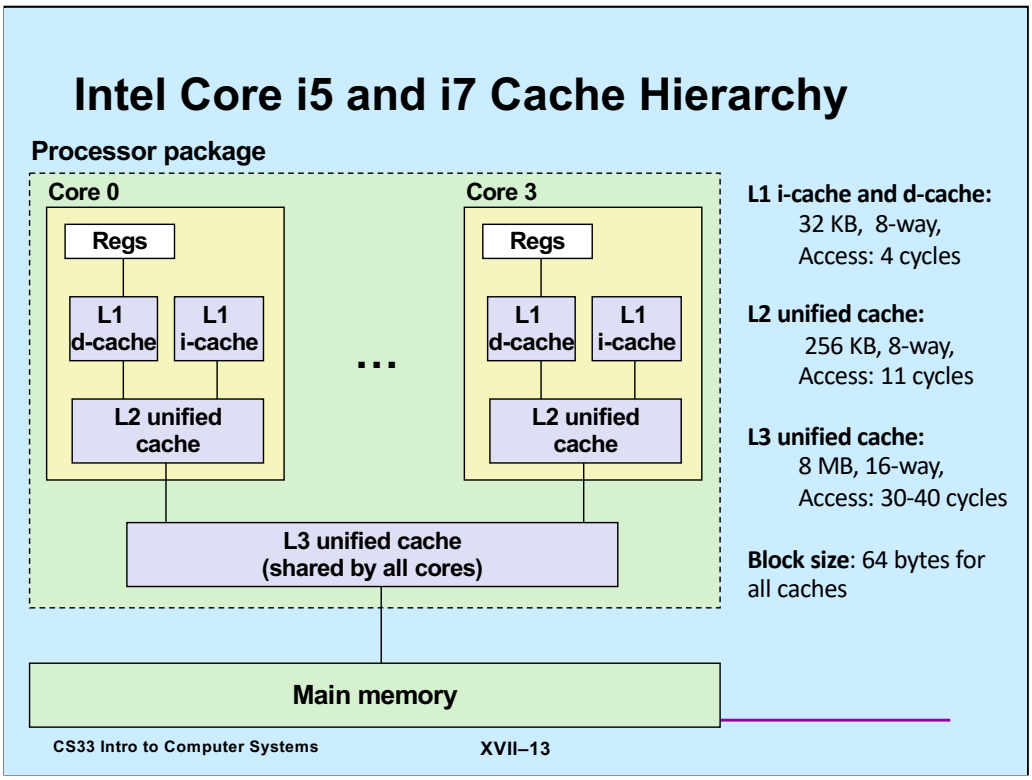
There is still a cache miss on each access.

Conflict Misses

```
double dotprod(double x[8], double y[8]) {
  double sum = 0.0;
  int i;

  for (i=0; i<8; i++)
    sum += x[i] * y[i];

  return sum;
}
```



With a 2-way set-associative cache, our dot-product example runs quickly even if the two arrays have the same alignment.

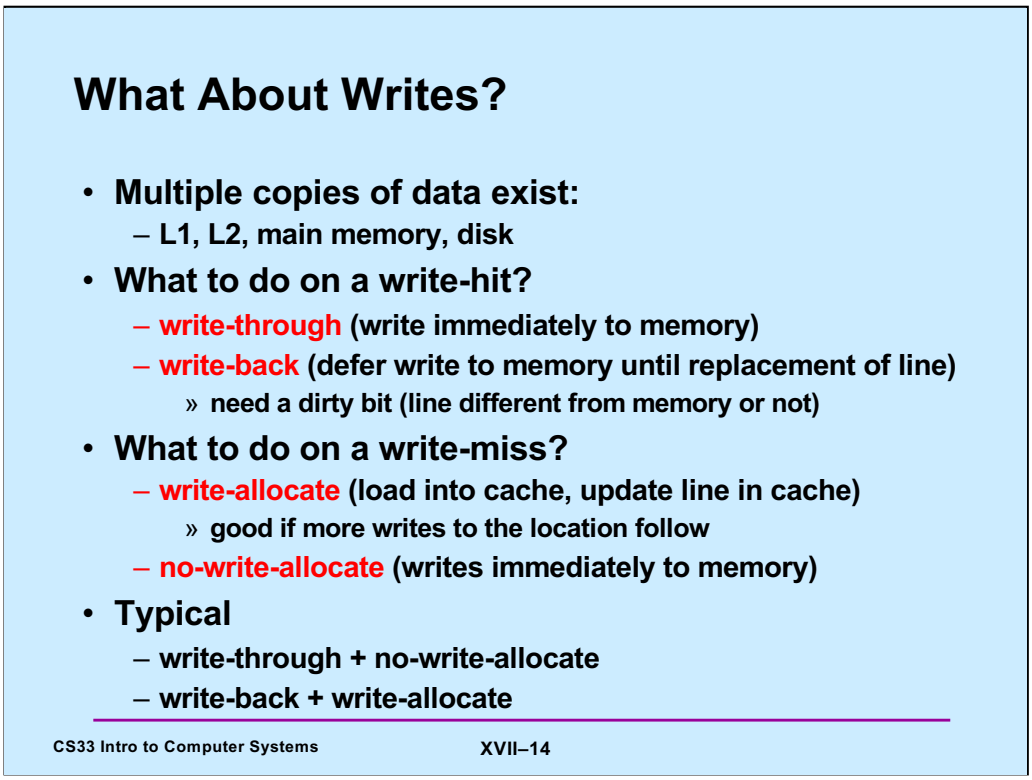


Supplied by CMU.

The L3 cache is known as the *last-level cache* (LLC) in the Intel documentation.

One concern is whether what's contained in, say, the L1 cache is also contained in the L2 cache. if so, caching is said to be **inclusive**. If what's contained in the L1 cache is definitely not contained in the L2 cache, caching is said to be **exclusive**. An advantage of exclusive caches is that the total cache capacity is the sum of the sizes of each of the levels, whereas for inclusive caches, the total capacity is just that of the largest. An advantage of inclusive caches is that what's been brought into the cache hierarchy by one core is available to the other cores.

AMD processors tend to have exclusive caches; Intel processors tend to have inclusive caches.



Supplied by CMU.

Most current processors use the write-back/write-allocate approach. This causes some (surmountable) difficulties for multi-core processors that have a separate cache for each core.

Accessing Memory

- Program references memory (load)
  - if not in cache (*cache miss*), data is requested from RAM
    - » fetched in units of 64 bytes
      - aligned to 64-byte boundaries (low-order 6 bits of address are zeroes)
    - » if memory accessed sequentially, data is pre-fetched
    - » data stored in cache (in 64-byte *cache lines*)
      - stays there until space must be re-used (least recently used is kicked out first)
  - if in cache (*cache hit*) no access to RAM needed
- Program modifies memory (store)
  - data modified in cache
  - eventually written to RAM in 64-byte units

This slide describes accessing memory on Intel Core I5 and I7 processors.

If the processor determines that a program is accessing memory sequentially (because the past few accesses have been sequential), then it begins the load of the next block from memory before it is requested. If this determination was correct, then the memory will be in the cache (or well on its way) before it's needed.

Cache Performance Metrics

- **Miss rate**
  - fraction of memory references not found in cache (misses / accesses) = 1 – hit rate
  - typical numbers (in percentages):
    - » 3-10% for L1
    - » can be quite small (e.g., < 1%) for L2, depending on size, etc.
- **Hit time**
  - time to deliver a line in the cache to the processor
    - » includes time to determine whether the line is in the cache
  - typical numbers:
    - » 1-2 clock cycles for L1
    - » 5-20 clock cycles for L2
- **Miss penalty**
  - additional time required because of a miss
    - » typically 50-200 cycles for main memory (trend: increasing!)

Supplied by CMU.

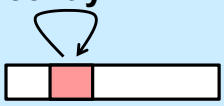
Hits vs. Misses

- Huge difference between hit and miss times
  - could be 100x, if just L1 and main memory
- 99% hit rate is twice as good as 97%!
  - consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
  - average access time:
    97% hits: .97 * 1 cycle + 0.03 * 100 cycles ≈ 4 cycles
    99% hits: .99 * 1 cycle + 0.01 * 100 cycles ≈ 2 cycles
- This is why "miss rate" is used instead of "hit rate"

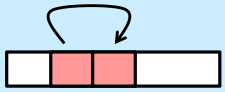
Supplied by CMU.

Locality

- **Principle of Locality:** programs tend to use data and instructions with addresses near or equal to those they have used recently



- **Temporal locality:**
  - recently referenced items are likely to be referenced again in the near future



- **Spatial locality:**
  - items with nearby addresses tend to be referenced close together in time

Supplied by CMU.

Locality Example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- **Data references**
  - reference array elements in succession (stride-1 reference pattern) — Spatial locality
  - reference variable `sum` each iteration — Temporal locality
- **Instruction references**
  - reference instructions in sequence. — Spatial locality
  - cycle through loop repeatedly — Temporal locality

Supplied by CMU.

Quiz 2

Does this function have good locality with respect to array `a`? The array a is MxN.

a) yes
b) no

```
int sum_array_cols(int N, int a[][N]) {
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```


Supplied by CMU.

Writing Cache-Friendly Code

- **Make the common case fast**
  - **focus on the inner loops of the core functions**
- **Minimize the misses in the inner loops**
  - **repeated references to variables are good (temporal locality)**
  - **stride-1 reference patterns are good (spatial locality)**

Supplied by CMU.

"Stride n" reference patterns are sequences of memory accesses in which every nth element is accessed in memory order. Thus stride 1 means that every element is accessed, starting at the beginning of a memory area, continuing to its end.

Matrix Multiplication Example

- **Description:**
  - **multiply N x N matrices**
    - » **each element is a double**
  - **O(N^3) total operations**
  - **N reads per source element**
  - **N values summed per destination**
    - » **but may be able to hold in register**

Variable **sum** *held in register*

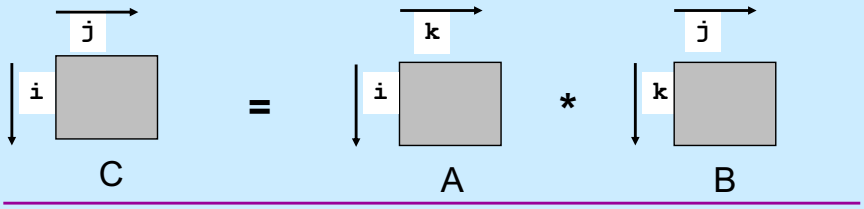
```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```


Based on slides supplied by CMU.

Miss-Rate Analysis for Matrix Multiply

- **Assume:**
  - Block size = 64B (big enough for eight doubles)
  - matrix dimension (N) is very large
  - cache is not big enough to hold multiple rows
- **Analysis method:**
  - look at access pattern of inner loop



Adapted form a slide by CMU.

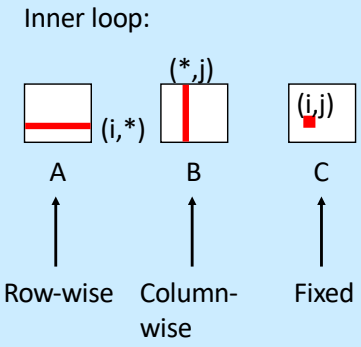
Layout of C Arrays in Memory (review)

- C arrays allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row:
  - `for (i = 0; i < N; i++)`
    `sum += a[0][i];`
  - accesses successive elements
  - if block size (B) > 8 bytes, exploit spatial locality
    - » compulsory miss rate = 8 bytes / Block
- Stepping through rows in one column:
  - `for (i = 0; i < n; i++)`
    `sum += a[i][0];`
  - accesses widely separated elements
  - no spatial locality!
    - » compulsory miss rate = 1 (i.e. 100%)

Supplied by CMU.

Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```



Misses per inner loop iteration:

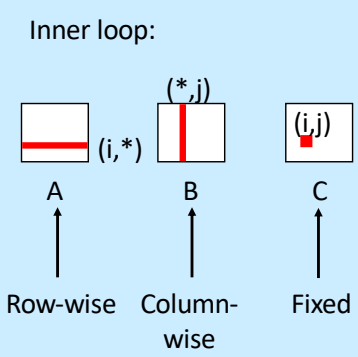
A	B	C
0.125	1.0	0.0

Supplied by CMU.

Assume we are multiplying arrays of doubles, thus each element is eight bytes long, and thus a cache line holds eight matrix elements. The slide shows a straightforward implementation of multiplying A and B to produce C.

Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
```



Misses per inner loop iteration:

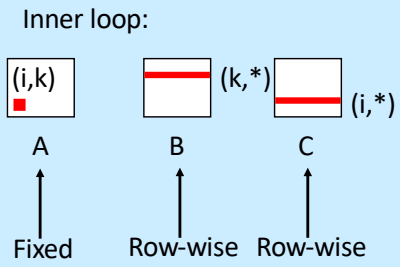
A	B	C
0.125	1.0	0.0

Supplied by CMU.

If we reverse the order of the two outer loops, there's no change in results or performance.

Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```



Misses per inner loop iteration:

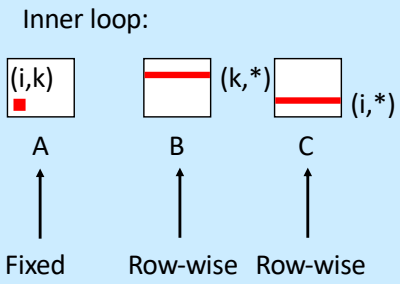
A	B	C
0.0	0.125	0.125

Supplied by CMU.

Moving the loop on k to be the outer loop does not affect the result, but it improves performance.

Matrix Multiplication (ikj)

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```



Misses per inner loop iteration:

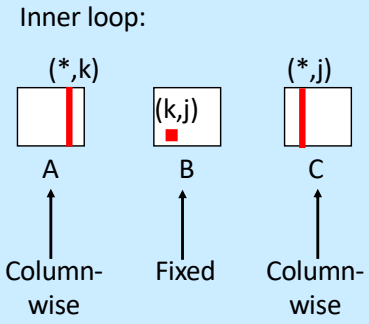
A	B	C
0.0	0.125	0.125

Supplied by CMU.

Switching the two outer loops affects neither results nor performance.

Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```



Misses per inner loop iteration:

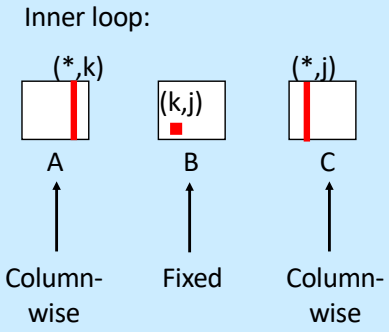
A	B	C
1.0	0.0	1.0

Supplied by CMU.

Moving the loop on i to be the inner loop makes performance considerably worse.

Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```



Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Supplied by CMU.

The poor performance is not improved by reversing the outer loops.

Summary of Matrix Multiplication

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++) {
   sum = 0.0;
   for (k=0; k<n; k++)
     sum += a[i][k] * b[k][j];
   c[i][j] = sum;
  }
```

ijk (& jik):
- 2 loads, 0 stores
- misses/iter = **1.125**

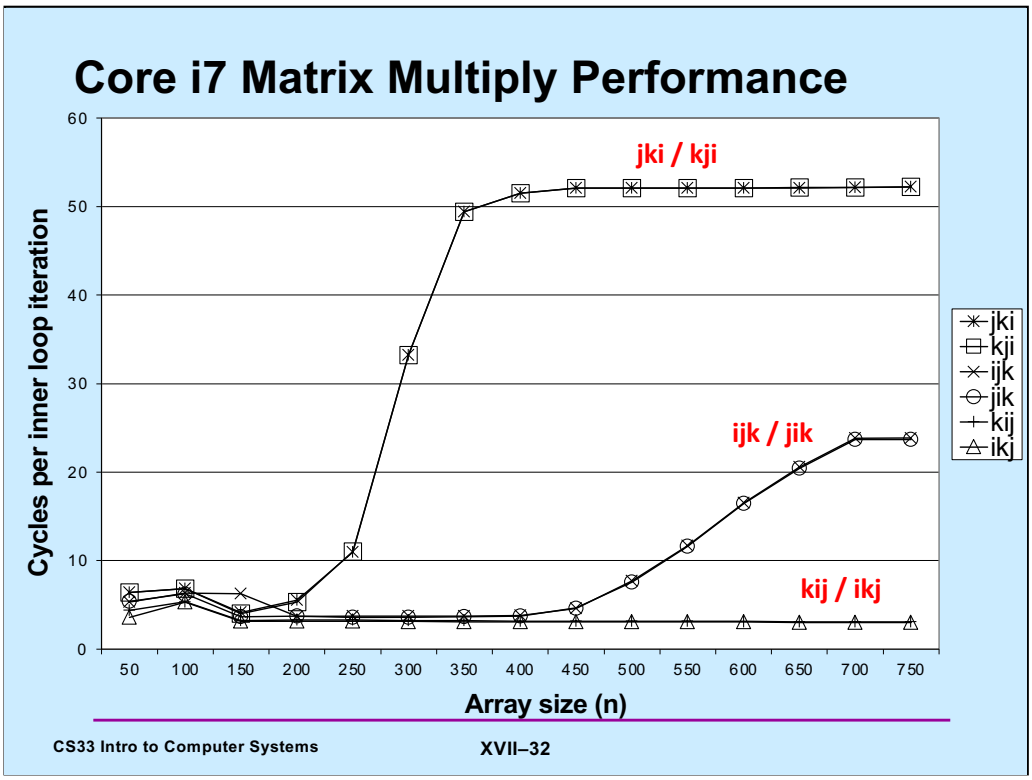
```
for (k=0; k<n; k++)
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
  }
```

kij (& ikj):
- 2 loads, 1 store
- misses/iter = **0.25**

```
for (j=0; j<n; j++)
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
```

jki (& kji):
- 2 loads, 1 store
- misses/iter = **2.0**

Supplied by CMU.



Supplied by CMU.

In Real Life ...

- Multiply two 1024x1024 matrices of doubles on sunlab machines
  - ijk
    - » 4.185 seconds
  - kij
    - » 0.798 seconds
  - jki
    - » 11.488 seconds

Concluding Observations

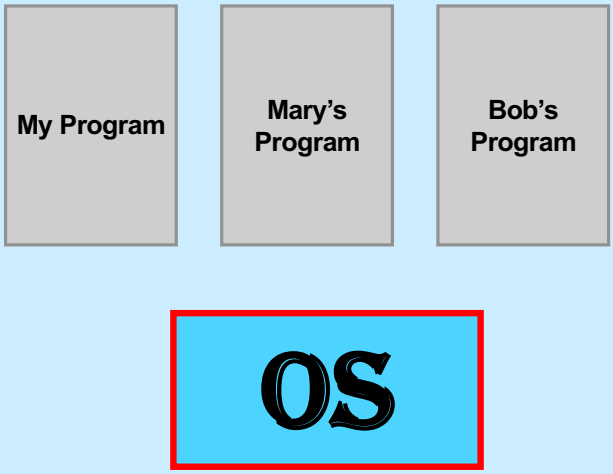
- Programmer can optimize for cache performance
  - organize data structures appropriately
- All systems favor "cache-friendly code"
  - getting absolute optimum performance is very platform specific
    - » cache sizes, line sizes, associativities, etc.
  - can get most of the advantage with generic code
    - » keep working set reasonably small (temporal locality)
    - » use small strides (spatial locality)

Supplied by CMU.

CS 33

Architecture and the OS

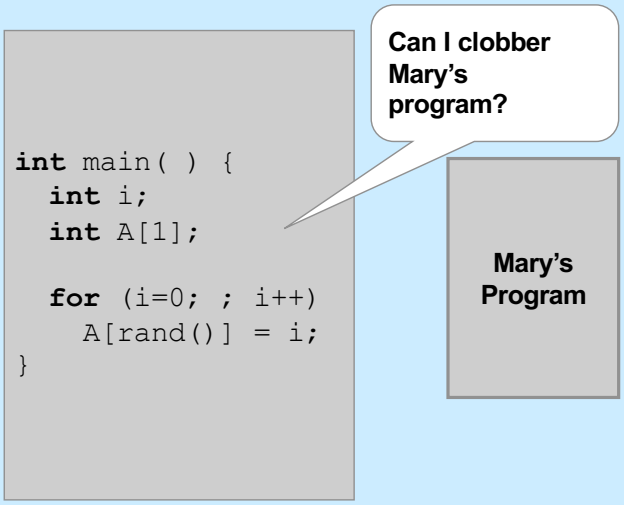
The Operating System



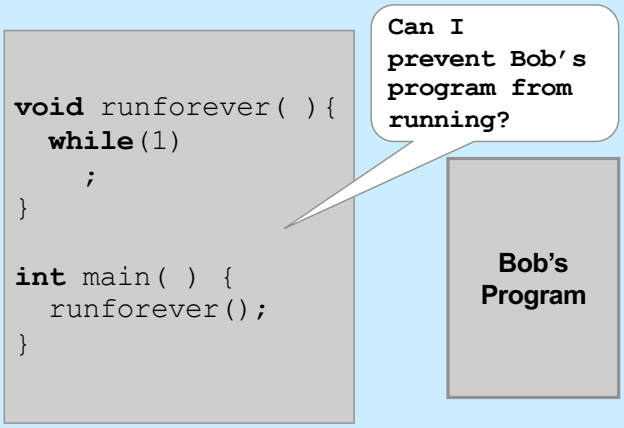
Processes

- Containers for programs
  - virtual memory
    - » address space
  - scheduling
    - » one or more threads of control
  - file references
    - » open files
  - and lots more!

Idiot Proof …



Fair Share



Architectural Support for the OS

- Not all instructions are created equal ...
  - non-privileged instructions
    - » can affect only current program
  - privileged instructions
    - » may affect entire system
- Processor mode
  - user mode
    - » can execute only non-privileged instructions
  - privileged mode
    - » can execute all instructions

Which Instructions Should Be Privileged?

- I/O instructions
- Those that affect how memory is mapped
- Halt instruction
- Some others ...

Who Is Privileged?

- No one
  - user code always runs in user mode
- The operating-system kernel runs in privileged mode
  - nothing else does
  - not even super user on Unix or administrator on Windows

Entering Privileged Mode

- How is OS invoked?
  - very carefully ...
  - strictly in response to interrupts and exceptions
  - (booting is a special case)

Interrupts and Exceptions

- Things don't always go smoothly ...
  - I/O devices demand attention
  - timers expire
  - programs demand OS services
  - programs demand storage be made accessible
  - programs have problems
- Interrupts
  - demand for attention by external sources
- Exceptions
  - executing program requires attention

Exceptions

- Traps
  - "intentional" exceptions
    - » execution of special instruction to invoke OS
  - after servicing, execution resumes with next instruction
- Faults
  - a problem condition that is normally corrected
  - after servicing, instruction is re-tried
- Aborts
  - something went dreadfully wrong ...
  - not possible to re-try instruction, nor to go on to next instruction

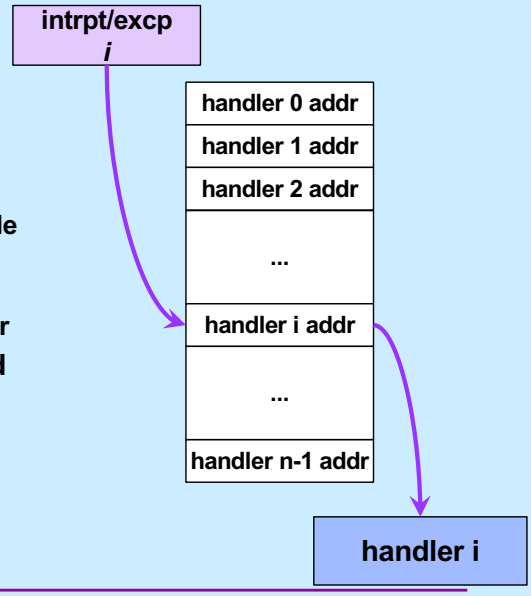
These definitions follow those given in "Intel® 64 and IA-32 Architectures Software Developer's Manual" and are generally accepted even outside of Intel.

Actions for Interrupts and Exceptions

- When interrupt or exception occurs
  - processor saves state of current thread/process on stack
  - processor switches to privileged mode (if not already there)
  - invokes handler for interrupt/exception
  - if thread/process is to be resumed (typical action after interrupt)
    - » thread/process state is restored from stack
  - if thread/process is to re-execute current instruction
    - » thread/process state is restored, after backing up instruction pointer
  - if thread/process is to terminate
    - » it's terminated

Interrupt and Exception Handlers

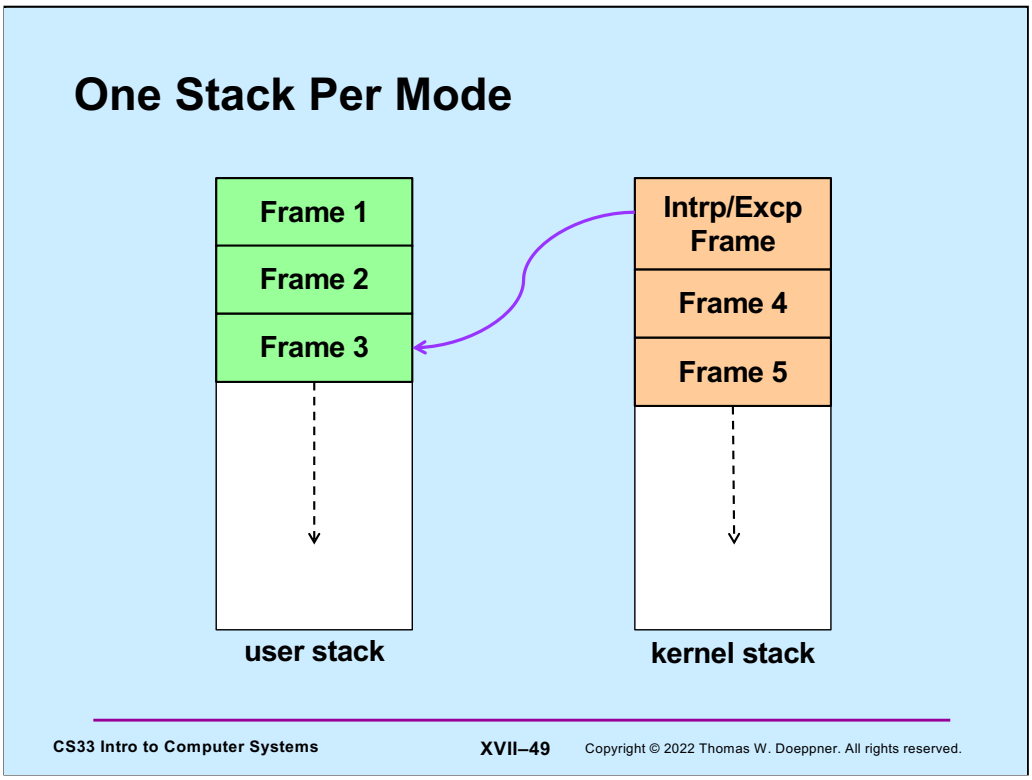
- Interrupt or exception invokes handler (in OS)
  - via interrupt and exception vector
    - » one entry for each possible interrupt/exception
      - contains
        - address of handler
  - code executed in privileged mode
    - » but code is part of the OS



Entering and Exiting

- **Entering/exiting interrupt/exception handler more involved than entering/exiting a procedure**
  - **must deal with processor mode**
    - » **switch to privileged mode on entry**
    - » **switch back to previous mode on exit**
  - **interrupted process/thread's state is saved on separate kernel stack**
  - **stack in kernel must be different from stack in user program**
    - » **why?**

The reason why there must be a separate stack in privileged mode is that the OS must be guaranteed that when it is executing, it has a valid stack, that the stack pointer must be pointing to a region of memory that can be used as a stack by the OS. Since while the program was running in user mode any value could have been put into the stack-pointer register, when the OS is invoked, it switches to a pre-allocated stack set up just for it.



When a trap or interrupt occurs, the current processor state (registers, including RIP, condition codes, etc.) are saved on the kernel stack. When the system returns back to the interrupted program, this state is restored.

Quiz 3

If an interrupt occurs, which general-purpose registers must be pushed onto the kernel stack?

a) all
b) none
c) callee-save registers
d) caller-save registers