CS 33

Memory Hierarchy I

Most 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.

This is the first of two lectures on memory hierarchy. The second, covering secondary storage (disk, etc.) will be given in a few weeks.

Random-Access Memory (RAM)

- **Key features**
  - **RAM is traditionally packaged as a chip**
  - **basic storage unit is normally a cell (one bit per cell)**
  - **multiple RAM chips form a memory**
- **Static RAM (SRAM)**
  - **each cell stores a bit with a four- or six-transistor circuit**
  - **retains value indefinitely, as long as it is kept powered**
  - **relatively insensitive to electrical noise (EMI), radiation, etc.**
  - **faster and more expensive than DRAM**
- **Dynamic RAM (DRAM)**
  - **each cell stores bit with a capacitor; transistor is used for access**
  - **value must be refreshed every 10-100 ms**
  - **more sensitive to disturbances (EMI, radiation,…) than SRAM**
  - **slower and cheaper than SRAM**

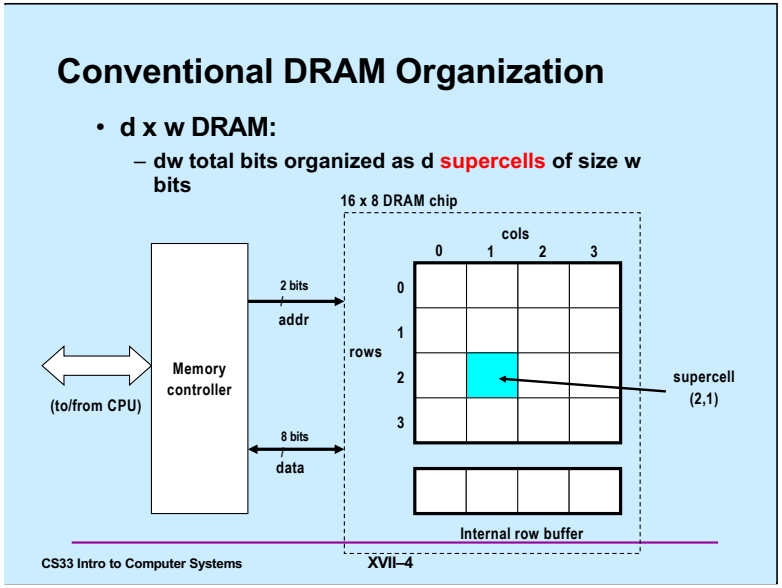
Supplied by CMU.

SRAM vs DRAM Summary

	Trans. per bit	Access time	Needs refresh?	Needs EDC?	Cost	Applications
SRAM	4 or 6	1X	No	Maybe	100x	Cache memories
DRAM	1	10X	Yes	Yes	1X	Main memories, frame buffers

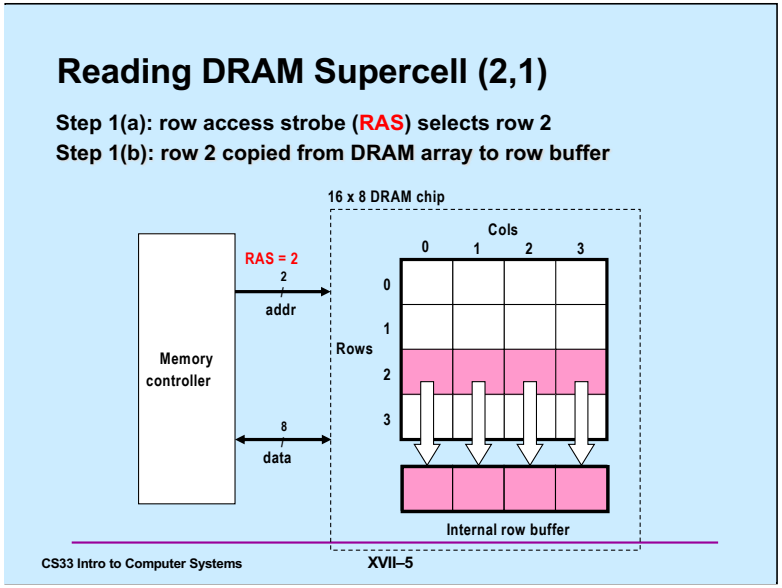
- **EDC = error detection and correction**
  - **to cope with noise, etc.**

Supplied by CMU.

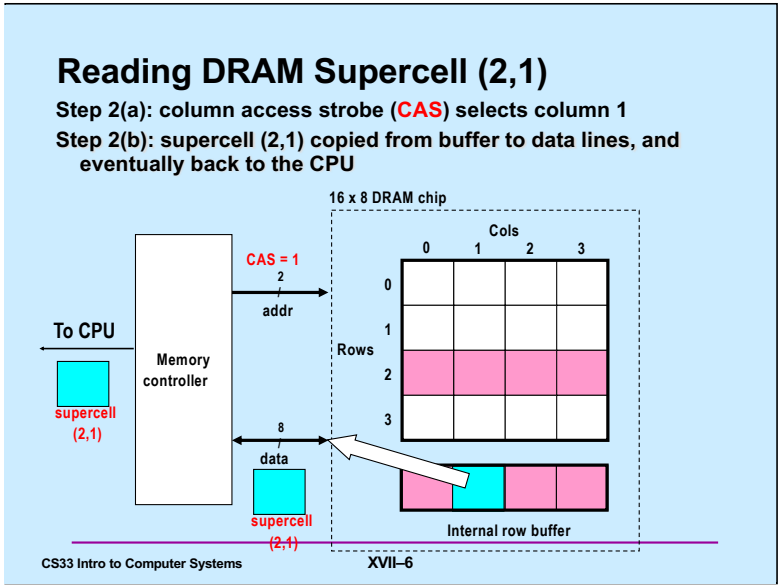


Supplied by CMU.

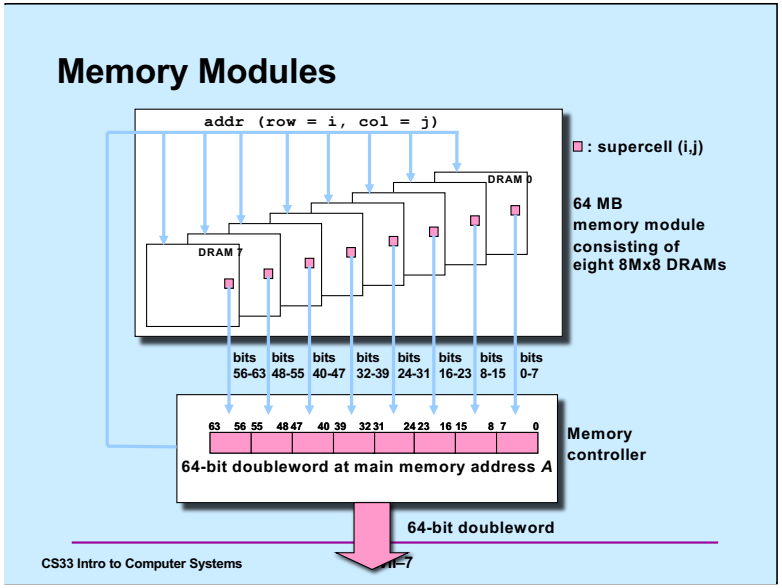
Note that the chip in the slide contains 16 supercells of 8 bits each. The supercells are organized as a 4x4 array.



Supplied by CMU.



Supplied by CMU.



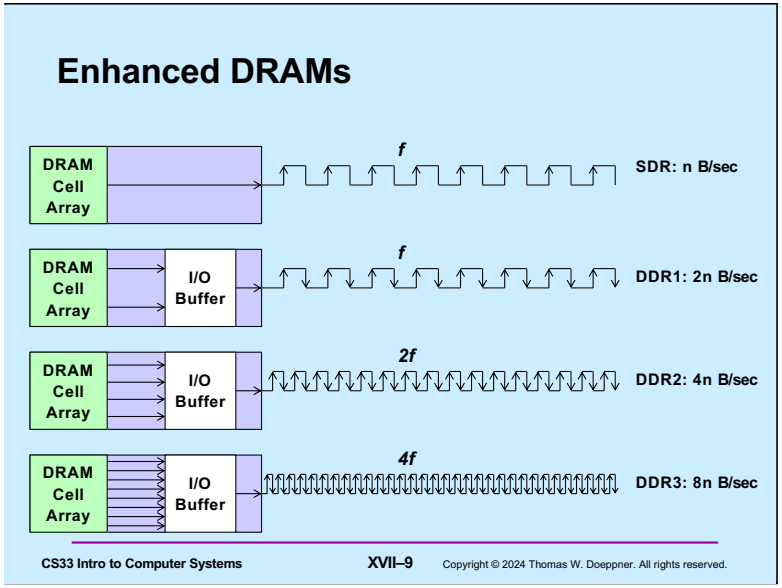
Supplied by CMU.

The memory controller pulls in eight supercells from eight DRAM modules and transfers them to the processor over the memory bus.

Enhanced DRAMs

- Basic DRAM cell has not changed since its invention in 1966
  - commercialized by Intel in 1970
- DRAMs with better interface logic and faster I/O:
  - synchronous DRAM (SDRAM or SDR)
    - » uses a conventional clock signal instead of asynchronous control
    - » allows reuse of the row addresses (e.g., RAS, CAS, CAS, CAS)
  - double data-rate synchronous DRAM (DDR SDRAM)
    - » DDR1
      - twice as fast: 16 consecutive bytes xfr'd as fast as 8 in SDR
    - » DDR2
      - 4 times as fast: 32 consecutive bytes xfr'd as fast as 8 in SDR
    - » DDR3
      - 8 times as fast: 64 consecutive bytes xfr'd as fast as 8 in SDR

Adapted from a slide supplied by CMU.



This slide is based on figures from **What Every Programmer Should Know About Memory** (http://www.akkadia.org/drepper/cpumemory.pdf), by Ulrich Drepper. It's an excellent article on memory and caching.

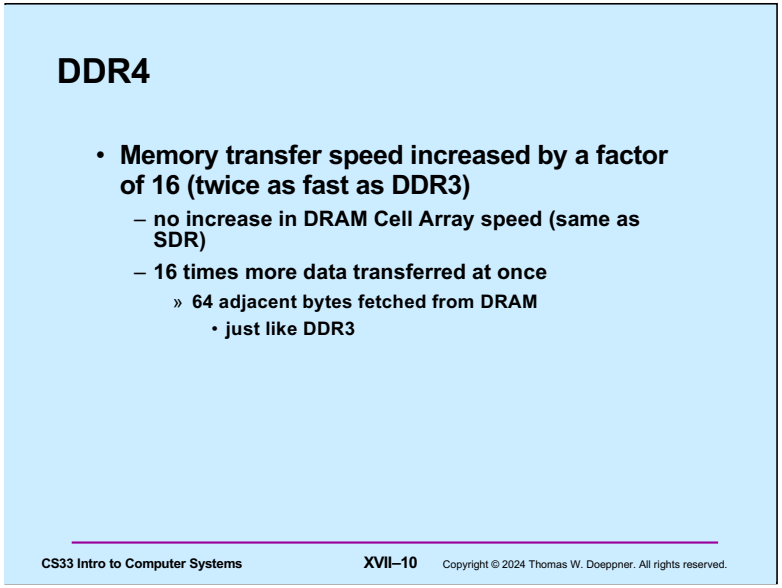
It is costly to make DRAM cell arrays run at a faster rate. Thus, rather than speed up the operation of the individual modules, they are organized to transfer in parallel. Thus, all that needs to be sped up is the bus that carries the data (something that is relatively inexpensive to do).

With SDR (Single Data-Rate DRAM), the DRAM cell array produces data at the same frequency as the memory bus, sending data on the rising edge of the signal.

With DDR1 (double data-rate), data is sent twice as fast by "double-pumping" the bus: sending data on both the rising and falling edges of the signal. To get data out of the cell array at this speed, data from two adjacent supercells are produced at once. These are buffered so that one doubleword at a time can be transmitted over the bus.

With DDR2, the frequency of the memory bus is doubled, and four supercells are produced at once. DDR3 takes this one step further, with eight supercells being produced at once. DDR4 takes this a step further and delivers 16 supercells at once.

Note that the processor fetches and stores 64 bytes of data at a time (for reasons having to do with caching, which we cover later in this lecture).



DDR4 memory became available in 2015. It's 16 times as fast as SDRAM, but transfers 64 consecutive bytes at a time, the same as DDR3. DDR5 is currently being discussed.

Quiz 2

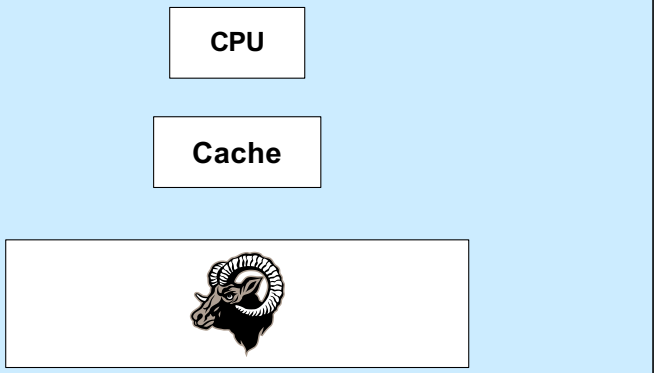
A program is loading randomly selected bytes from memory. These bytes will be delivered to the processor on a DDR4 system at a speed that's n times that of an SDR system, where n is:

a) 8
b) 4
c) 2
d) 1

A Mismatch

- A processor clock cycle is ~0.3 nsecs
  - Older SunLab machines (Intel Core i5-4690) run at 3.5 GHz
- Basic operations take 1 – 10 clock cycles
  - .3 – 3 nsecs
- Accessing memory takes 70-100 nsecs
- How is this made to work?

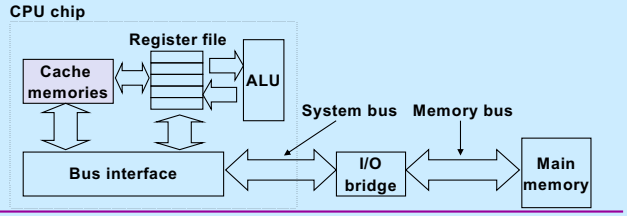
Caching to the Rescue



Sitting between the processor and RAM are one or more caches. (They actually are on the chip along with the processor.) Recently accessed items by the processor reside in the cache, where they are much more quickly accessed than directly from memory. The processor does a certain amount of pre-fetching to get things from RAM before they are needed. This involves a certain amount of guesswork, but works reasonably well, given well behaved programs.

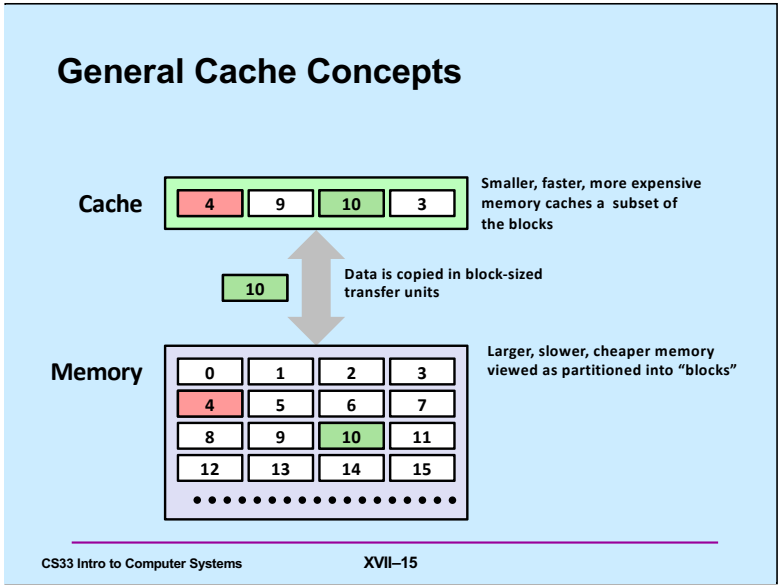
Cache Memories

- **Cache memories are small, fast SRAM-based memories managed automatically in hardware**
  - **hold frequently accessed blocks of main memory**
- **CPU looks first for data in caches (e.g., L1, L2, and L3), then in main memory**
- **Typical system structure:**

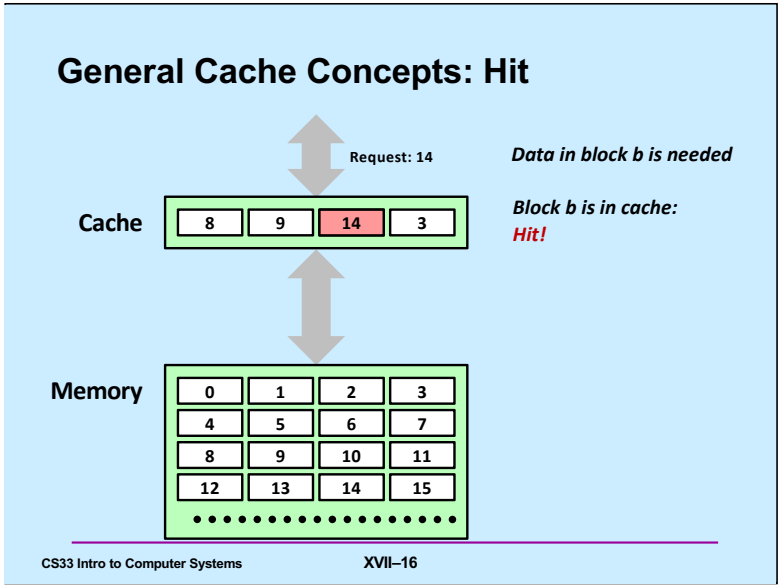


Supplied by CMU.

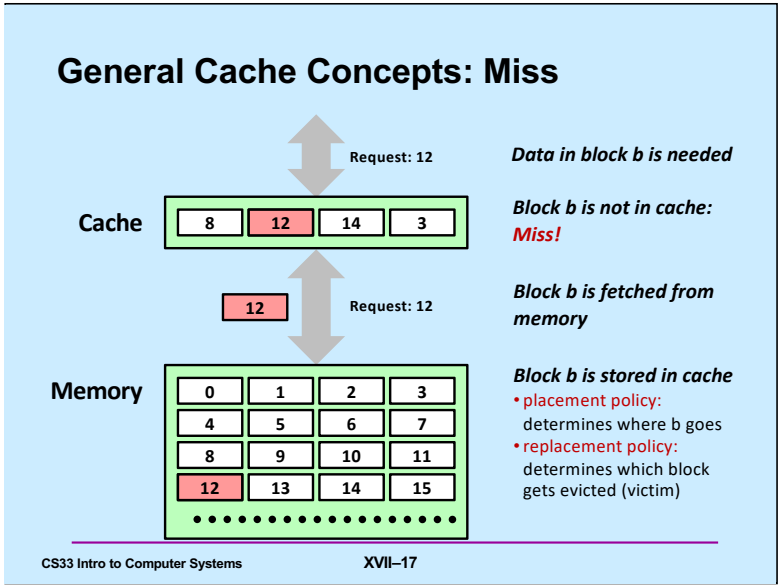
"ALU" (arithmetic and logic unit) is a traditional term for the instruction and execution units of a processor.



Supplied by CMU.



Supplied by CMU.

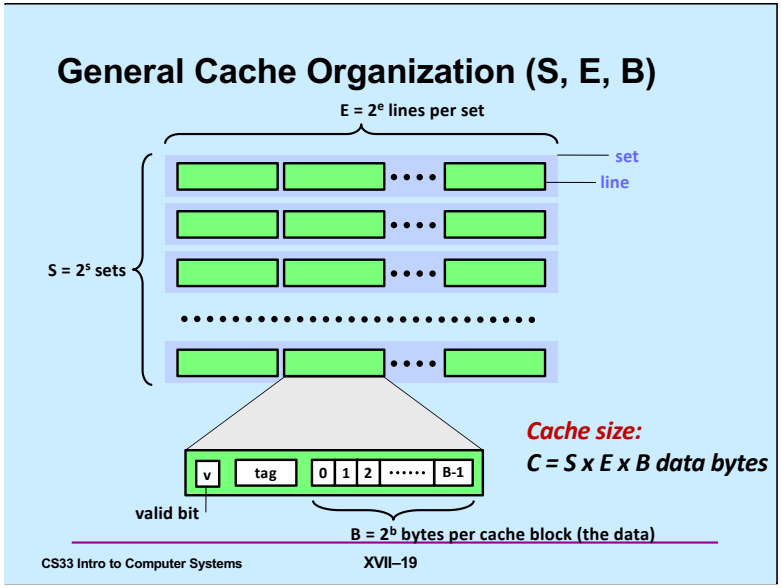


Supplied by CMU.

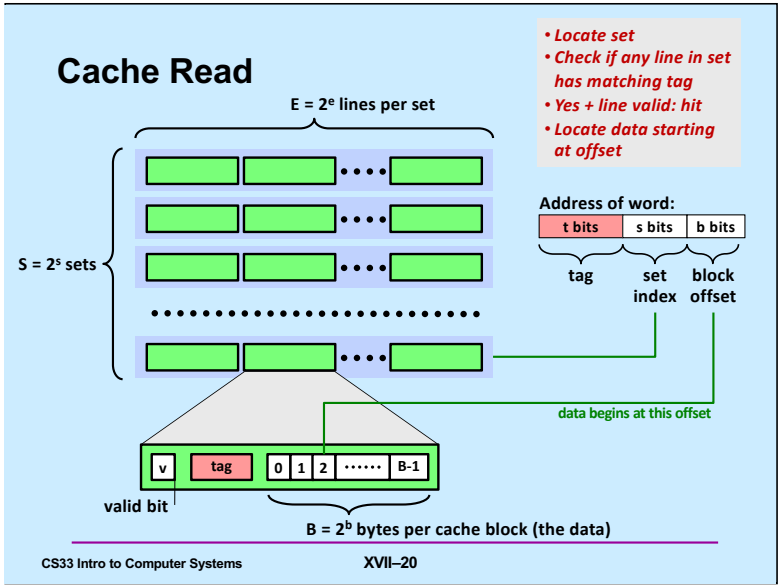
General Caching Concepts: Types of Cache Misses

- **Cold (compulsory) miss**
  - cold misses occur because the cache is empty
- **Conflict miss**
  - most caches limit blocks to a small subset (sometimes a singleton) of the block positions in RAM
    - » e.g., block i in RAM must be placed in block (i mod 4) in the cache
  - conflict misses occur when the cache is large enough, but multiple data objects all map to the same cache block
    - » e.g., referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time
- **Capacity miss**
  - occurs when the set of active cache blocks (working set) is larger than the cache

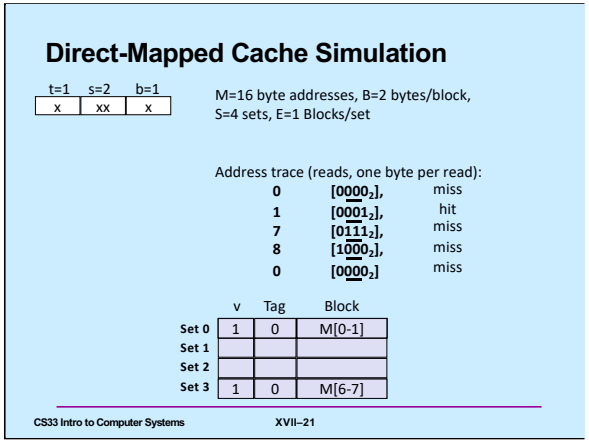
Supplied by CMU.



Supplied by CMU.



Supplied by CMU.



Supplied by CMU.

A Higher-Level Example

```
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;
}
```

Ignore the variables sum, i, j

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

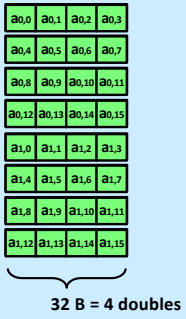
32 B = 4 doubles

Supplied by CMU.

A Higher-Level Example

```
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

Note that the cache holds two rows of the matrix; each cache block holds four doubles. When a[0][0] is read, so are a[0][1] through a[0][3]. Thus, after one cache miss, we get three hits.

A Higher-Level Example

```
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;
}
```

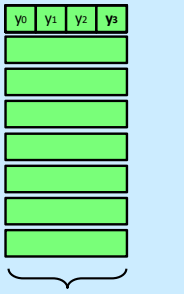
```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

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

32 B = 4 doubles

For each reference to an element of the matrix, its entire row is brought into the cache, even though the rest of the row is not immediately used.

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;
}
```

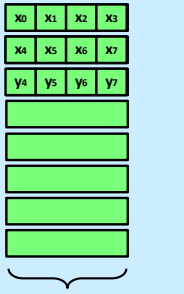


32 B = 4 doubles

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;
}
```



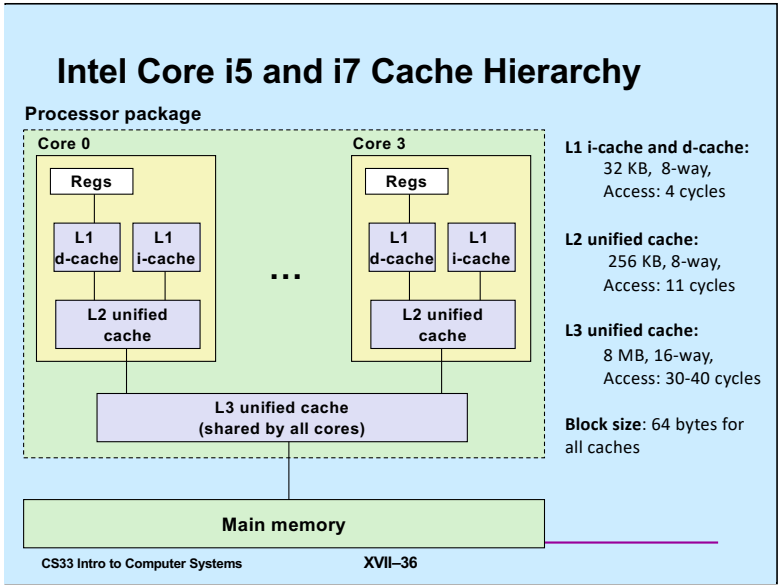
32 B = 4 doubles

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.

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.



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.

What About Writes?

- **Multiple copies of data exist:**
  - L1, L2, main memory, disk
- **What to do on a write-hit?**
  - write-through (write immediately to memory)
  - write-back (defer write to memory until replacement of line)
    - » need a dirty bit (line different from memory or not)
- **What to do on a write-miss?**
  - write-allocate (load into cache, update line in cache)
    - » good if more writes to the location follow
  - no-write-allocate (writes immediately to memory)
- **Typical**
  - write-through + no-write-allocate
  - write-back + write-allocate

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.

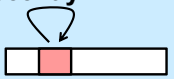
Hits vs. Misses

- 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"

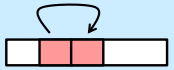
Supplied by CMU.

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.

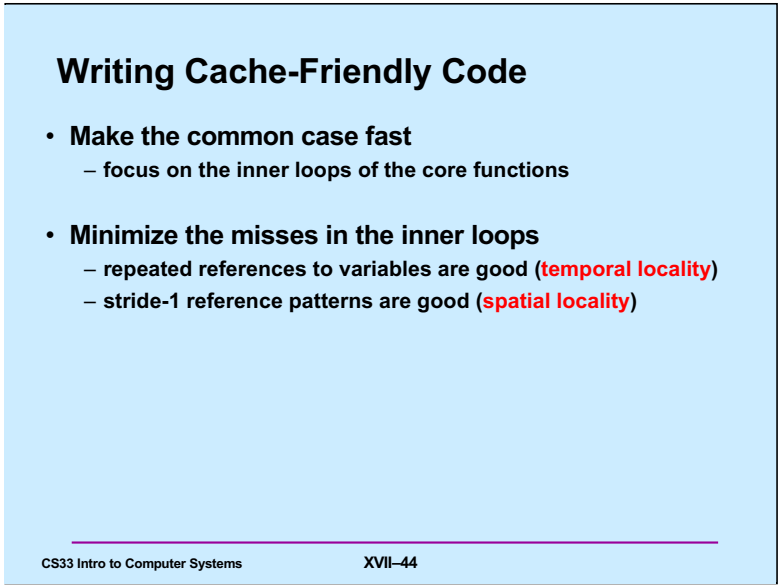
Quiz 2

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

a) yes
b) no

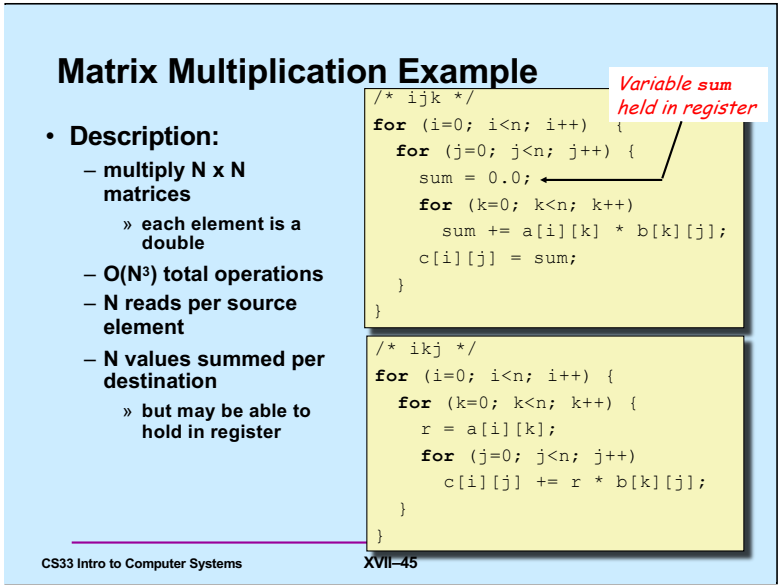
```
int sum_array_cols(int N, int a[][N]) {
    int i, j, sum = 0;

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


Supplied by CMU.



Supplied by CMU.

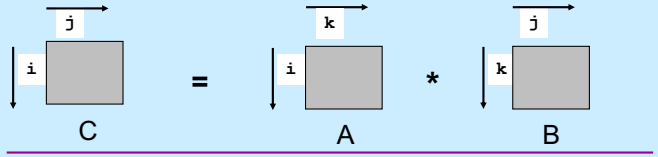
"Stride n" reference patterns are sequences of memory accesses in which every nth element is accessed in memory order. Thus stride 1 means that every element is accessed, starting at the beginning of a memory area, continuing to its end.



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)

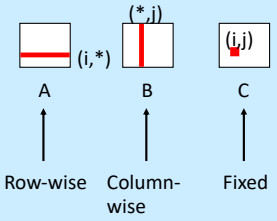
- C arrays allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row:
  - `for (i = 0; i < N; i++)`
    `sum += a[0][i];`
  - accesses successive elements
  - if block size (B) > 8 bytes, exploit spatial locality
    - » compulsory miss rate = 8 bytes / 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;
  }
}
```

Inner loop:

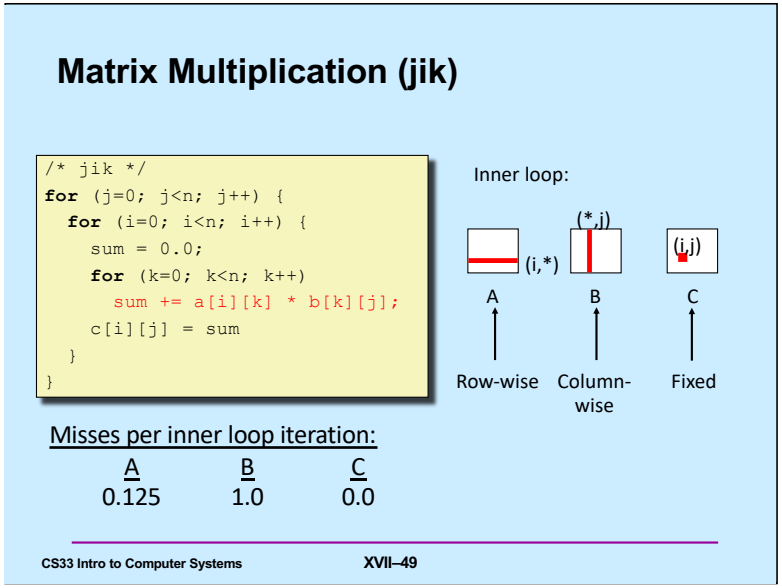


Misses per inner loop iteration:

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.

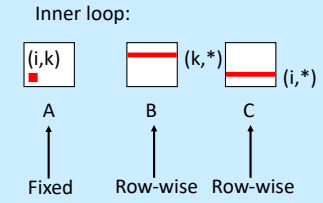


Supplied by CMU.

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:

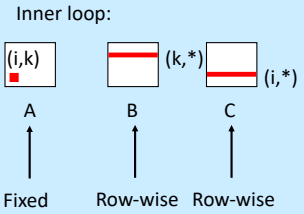
A	B	C
0.0	0.125	0.125

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:

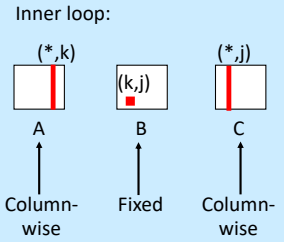
A	B	C
0.0	0.125	0.125

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:

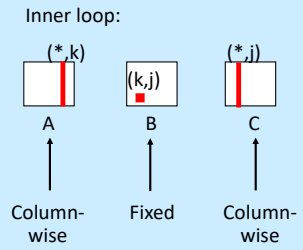
A	B	C
1.0	0.0	1.0

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;
  }
}
```

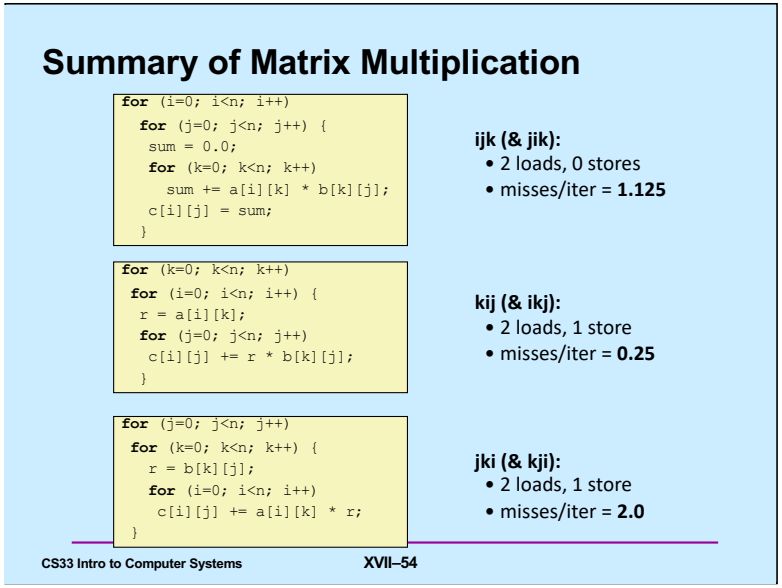


Misses per inner loop iteration:

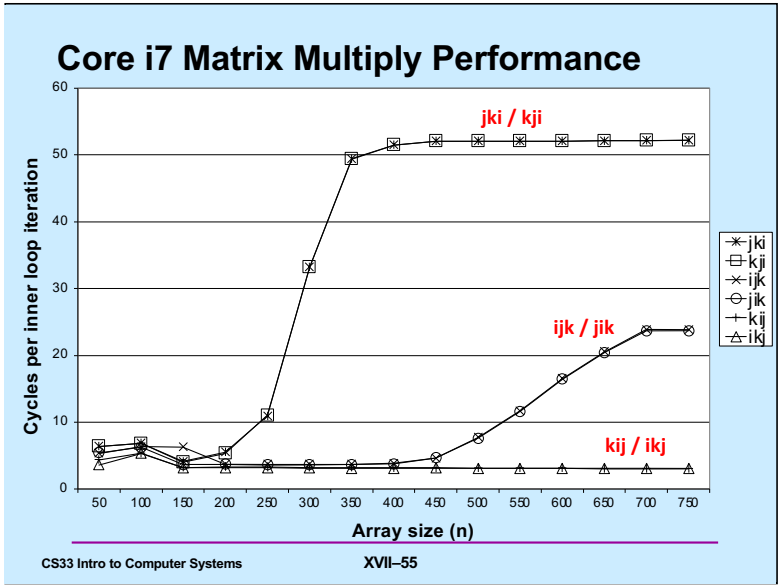
A	B	C
1.0	0.0	1.0

Supplied by CMU.

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



Supplied by CMU.



Supplied by CMU.

In Real Life ...

- Multiply two 1024x1024 matrices of doubles on core i5 machines (formerly in the Sunlab)
  - ijk
    - » 4.185 seconds
  - kij
    - » 0.798 seconds
  - jki
    - » 11.488 seconds

Concluding Observations

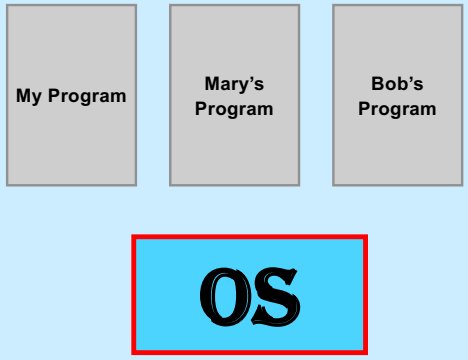
- **Programmer can optimize for cache performance**
  - **organize data structures appropriately**
- **All systems favor "cache-friendly code"**
  - **getting absolute optimum performance is very platform specific**
    - » **cache sizes, line sizes, associativities, etc.**
  - **can get most of the 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

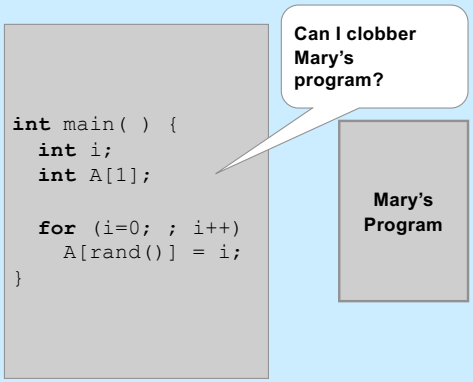
The Operating System



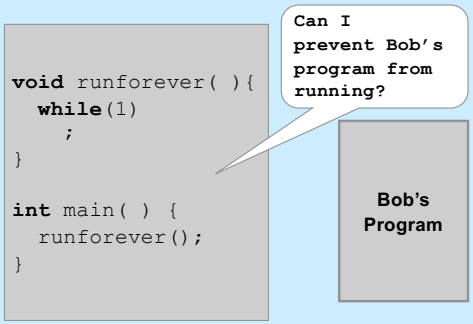
Processes

- 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 from 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

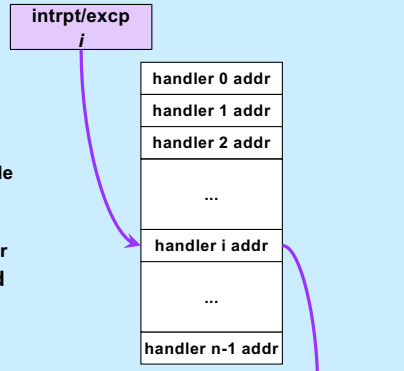
These definitions follow those given in "Intel® 64 and IA-32 Architectures Software Developer's Manual" and are generally accepted even outside of Intel.

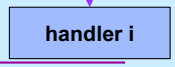
Actions for Interrupts and Exceptions

- 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

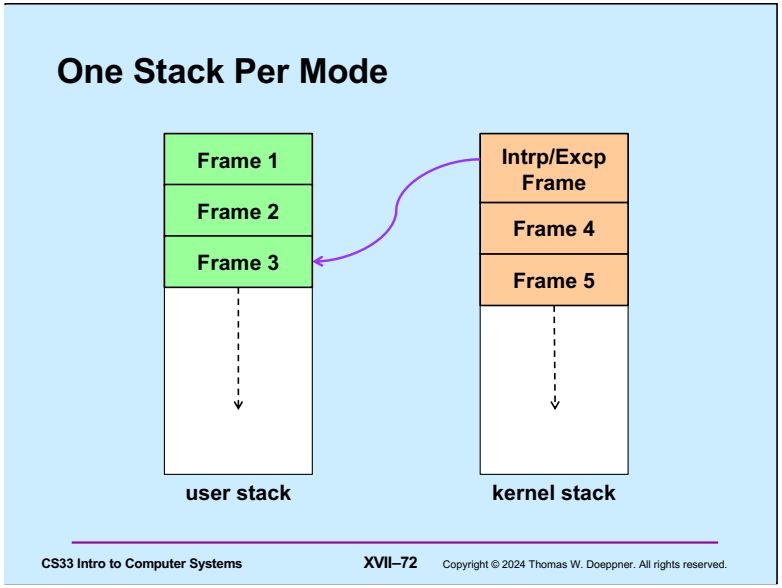




Entering and Exiting

- Entering/exiting interrupt/exception handler more involved than entering/exiting a function
  - 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, and 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