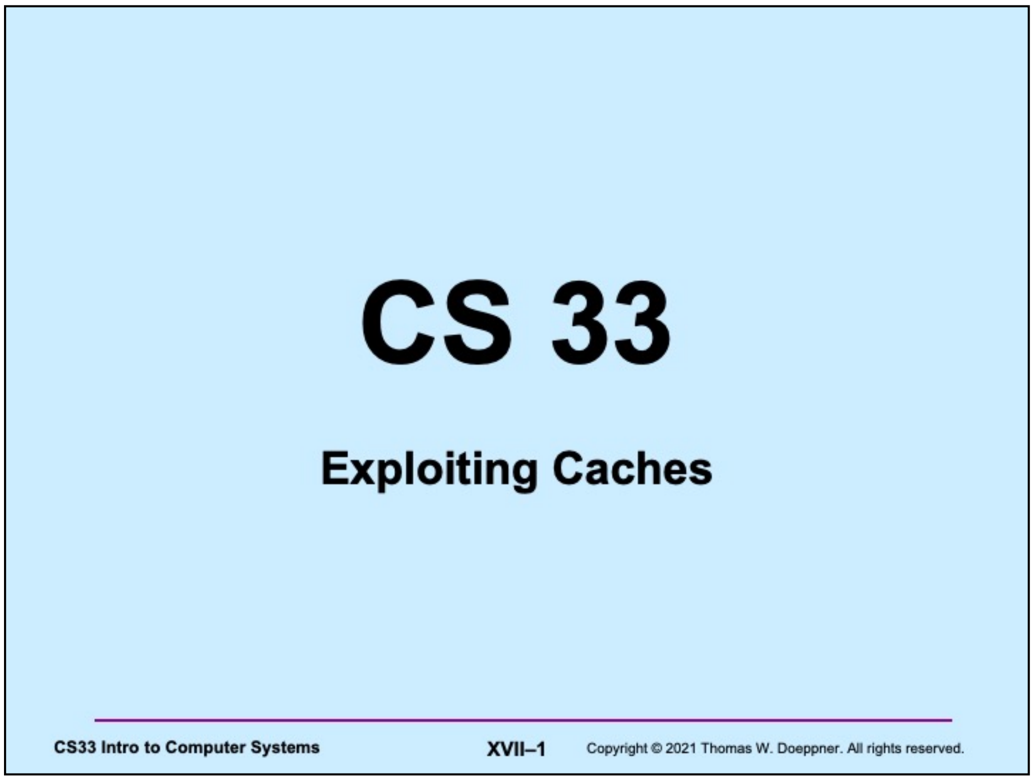
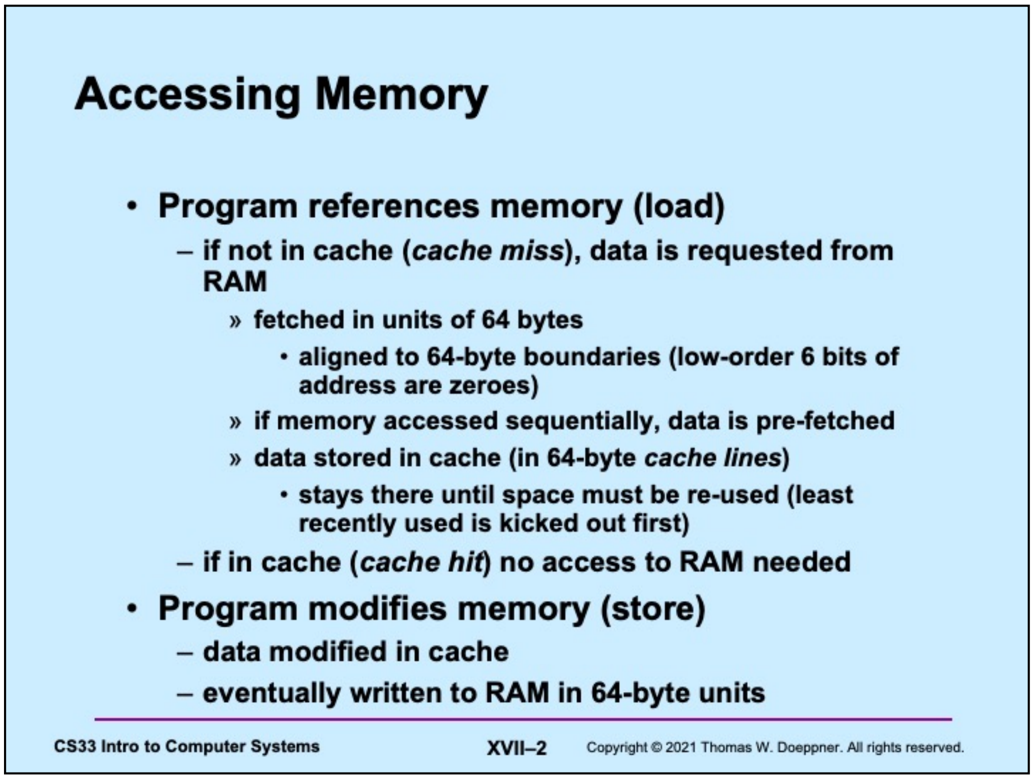


CS 33

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.

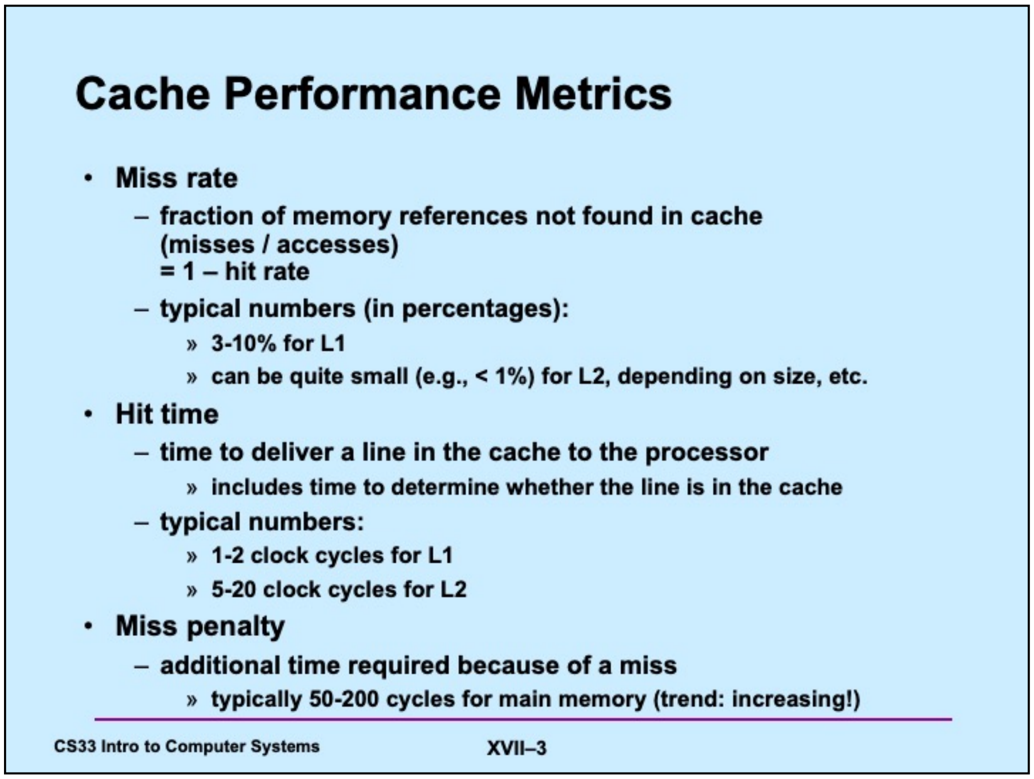


Accessing Memory

- **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.

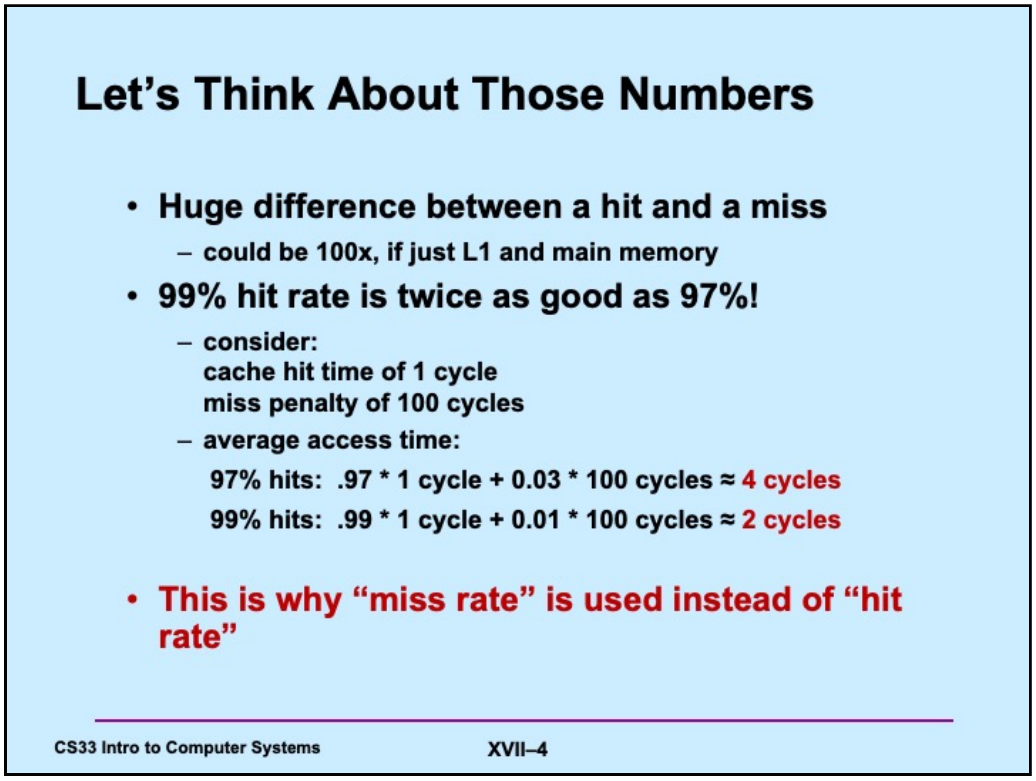
If the processor determines that a program is accessing memory sequentially (because the past few accesses have been sequential), then it begins the load of the next block from memory before it is requested. If this determination was correct, then the memory will be in the cache (or well on its way)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

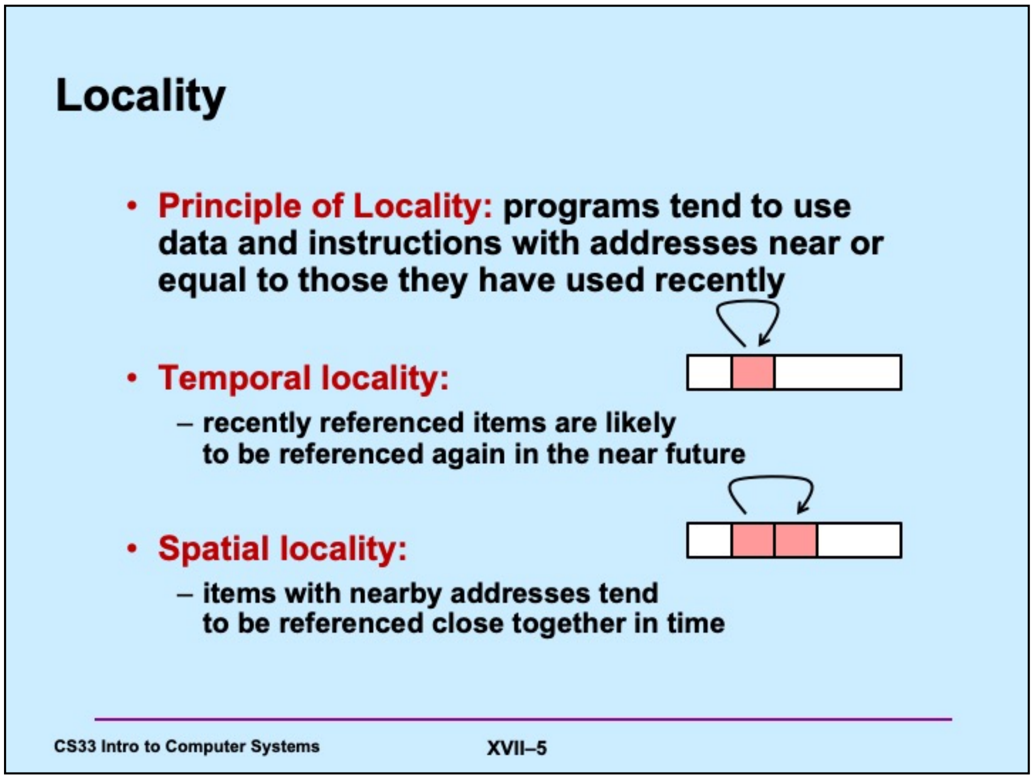
Supplied by CMU.



Let's Think About Those Numbers

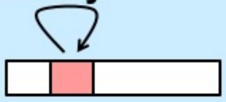
- Huge difference between a hit and a miss
  - could be 100x, if just L1 and main memory
- 99% hit rate is twice as good as 97%!
  - consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
  - average access time:
    97% hits: .97 * 1 cycle + 0.03 * 100 cycles ≈ 4 cycles
    99% hits: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

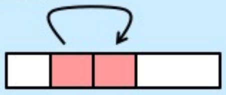
Supplied by CMU.



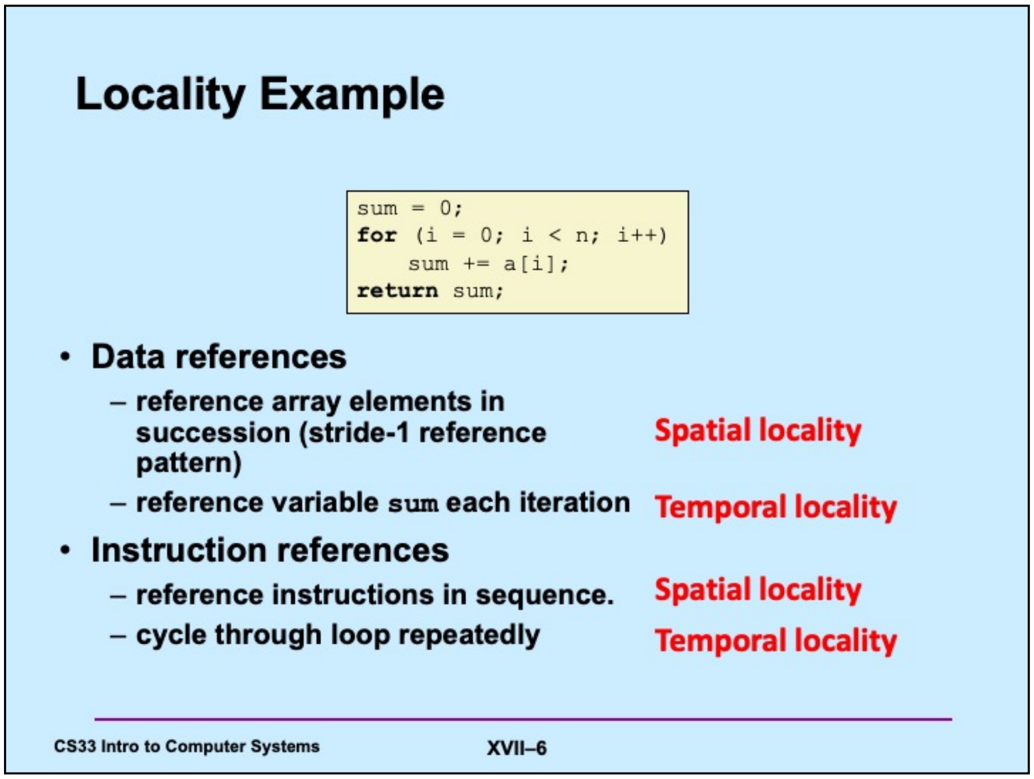
Locality

- **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.

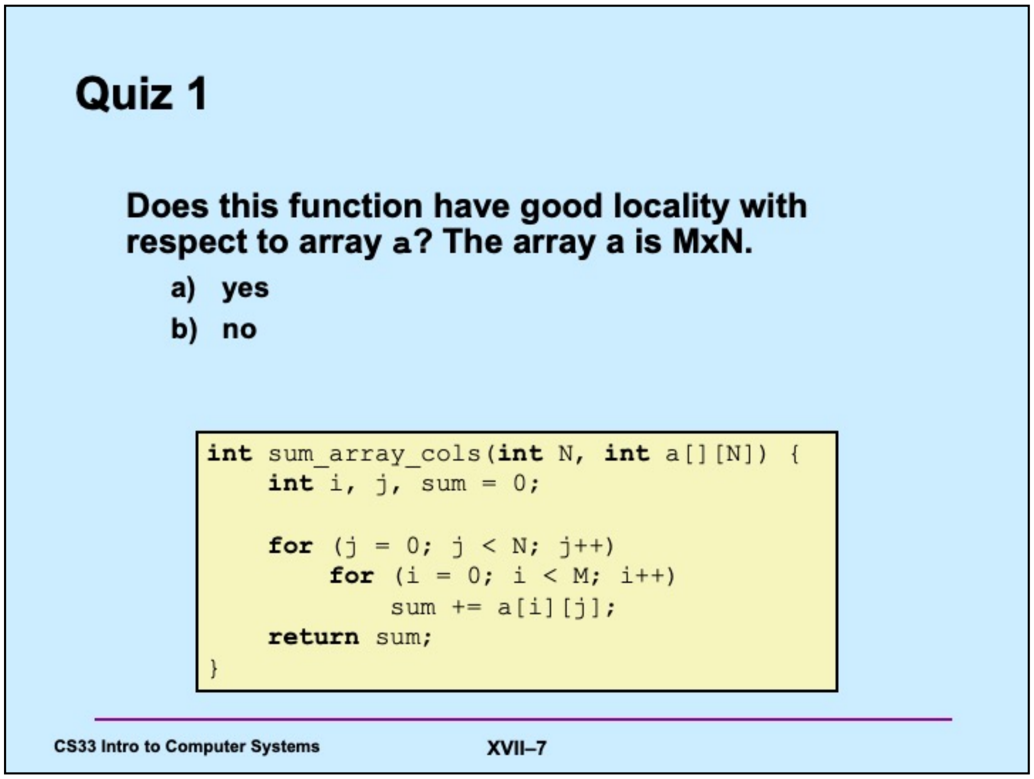


Locality Example

```
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 1

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

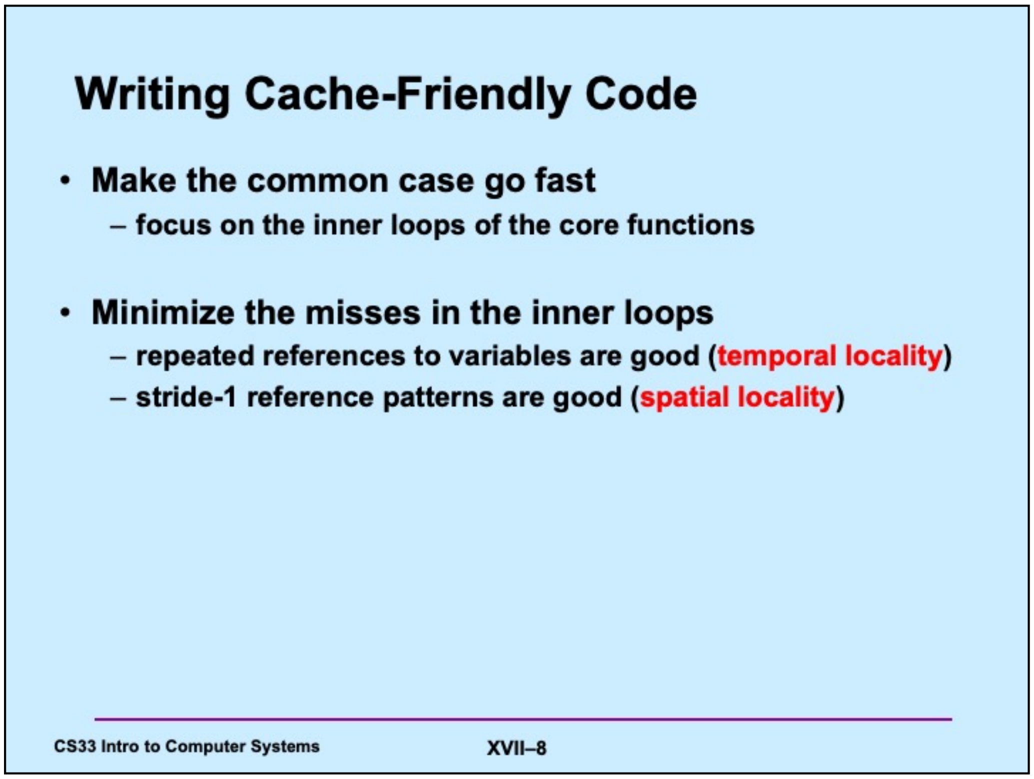
a) yes

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;
}
```

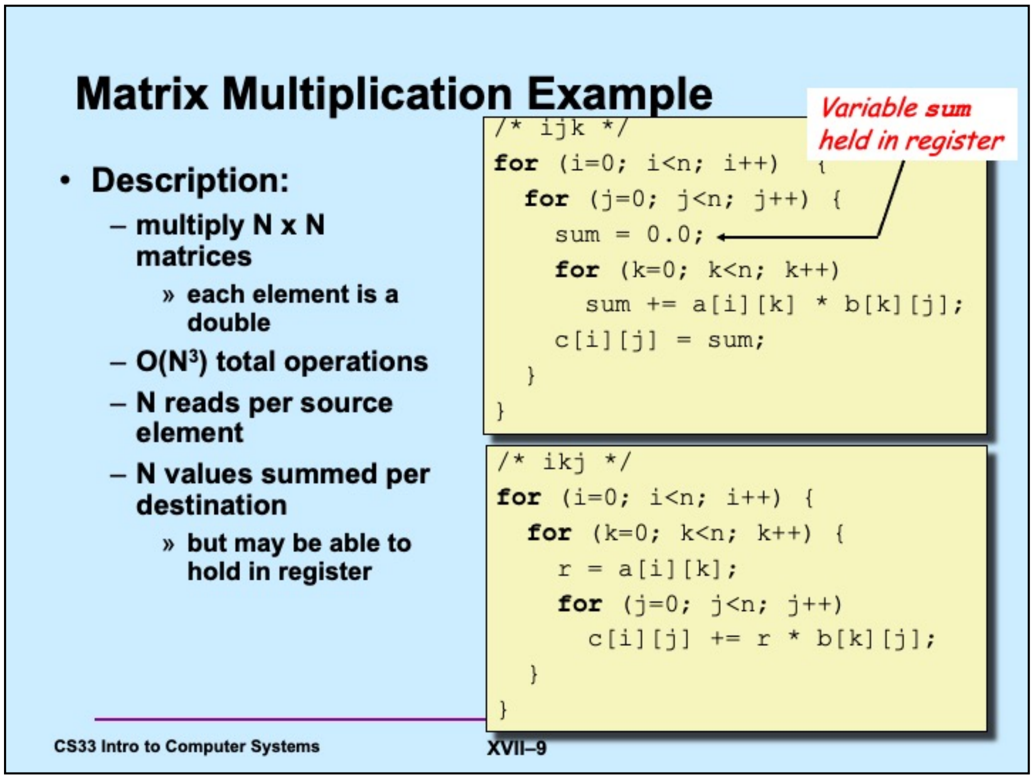
Supplied by CMU.



Writing Cache-Friendly Code

- Make the common case go 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.



Matrix Multiplication Example

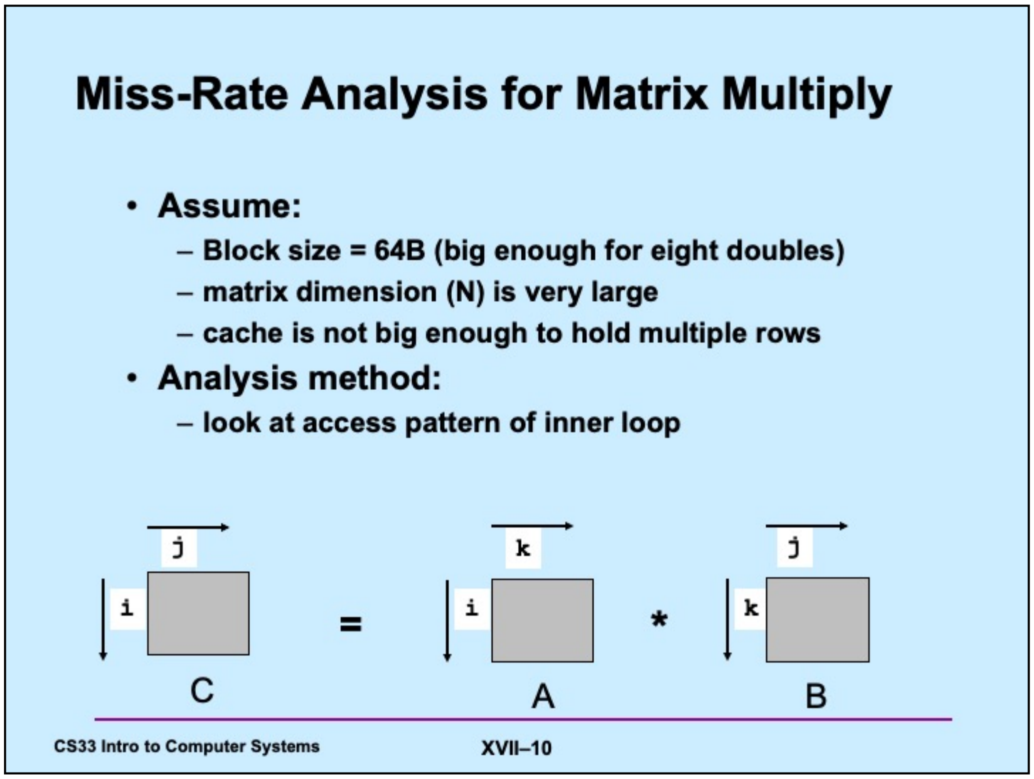
- **Description:**
  - **multiply N x N matrices**
    - » **each element is a double**
  - **O(N^3) total operations**
  - **N reads per source element**
  - **N values summed per destination**
    - » **but may be able to hold in register**

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];
  }
}
```

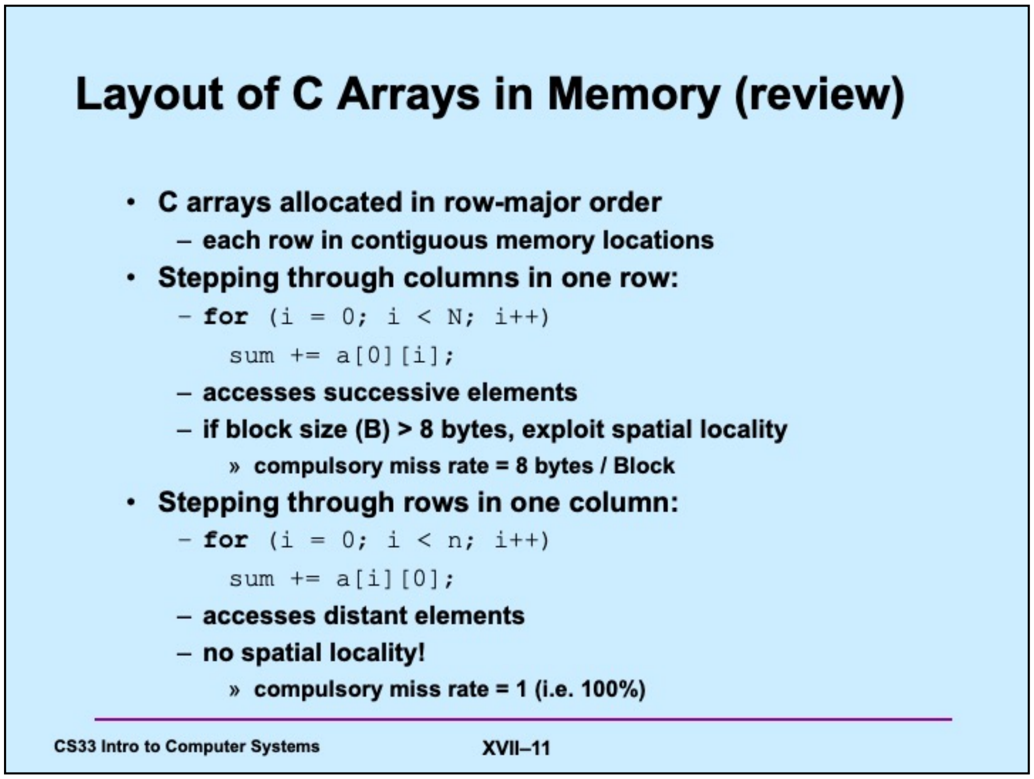
Based on slides supplied by CMU.



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.



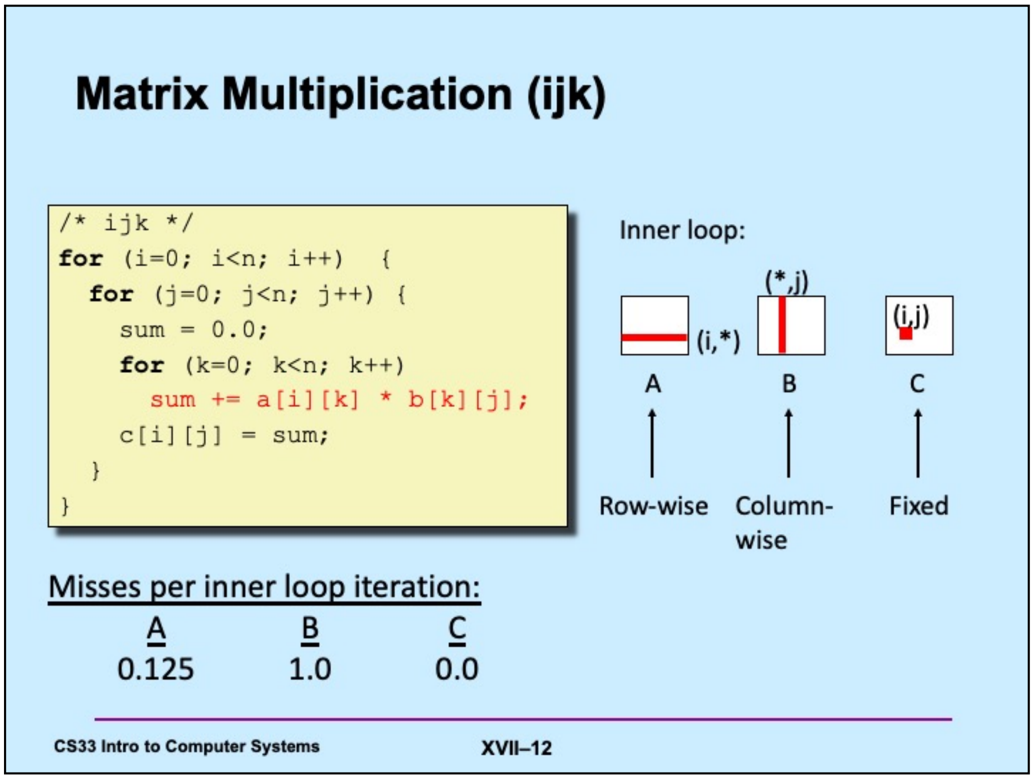
Layout of C Arrays in Memory (review)

- 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 distant elements
  - no spatial locality!
    - » compulsory miss rate = 1 (i.e. 100%)

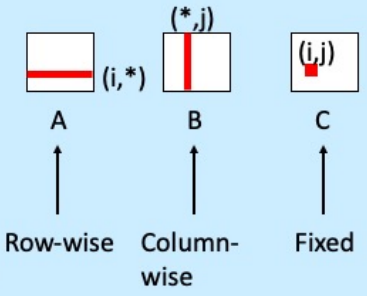
Supplied by CMU.



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:

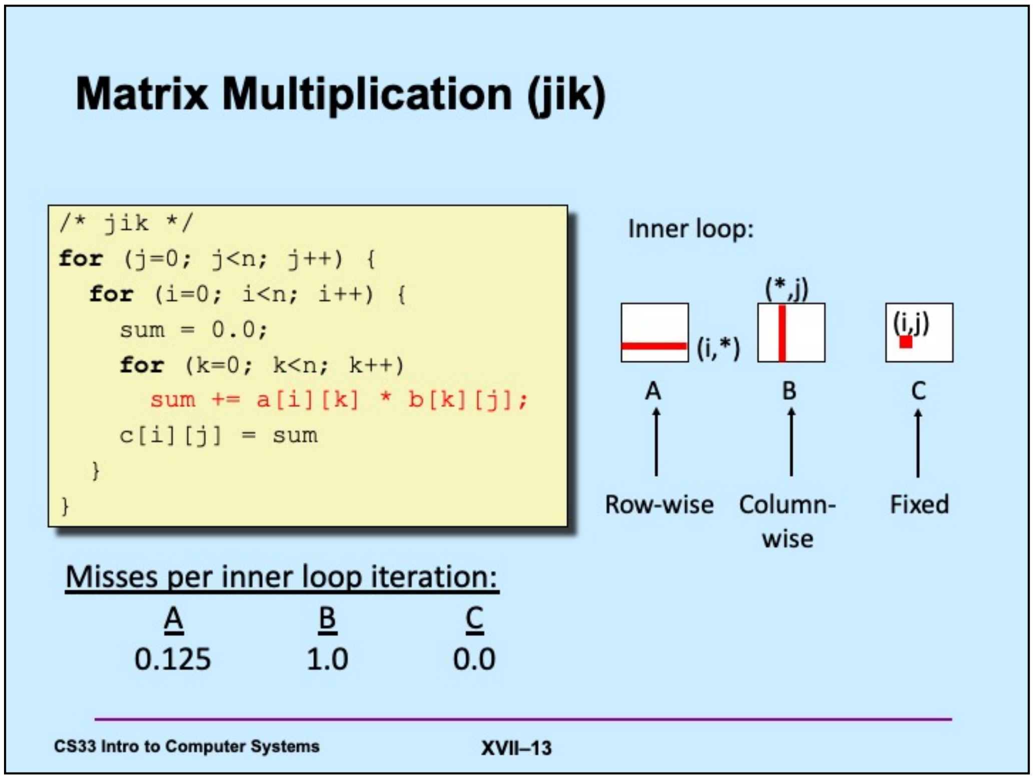


Misses per inner loop iteration:

A	B	C
0.125	1.0	0.0

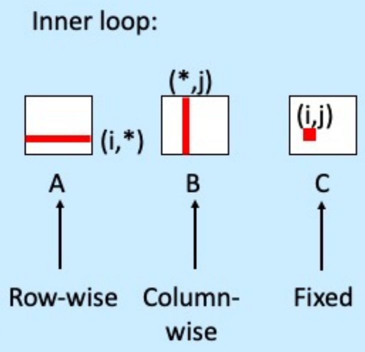
Supplied by CMU.

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.



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
  }
}
```

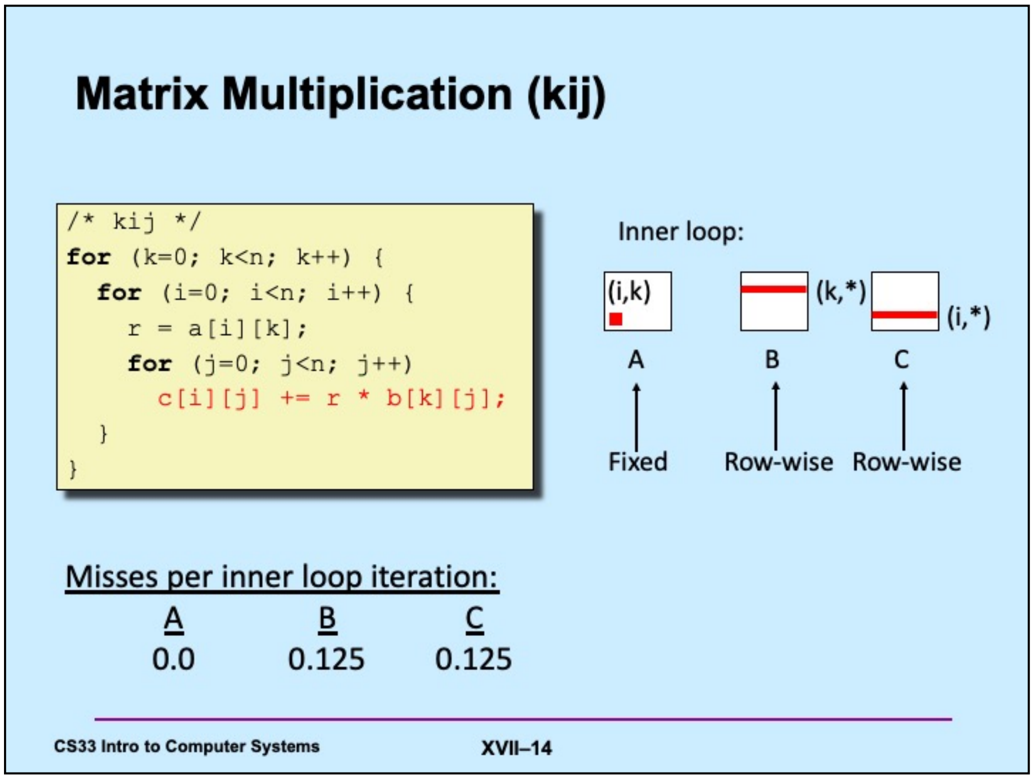


Misses per inner loop iteration:

A	B	C
0.125	1.0	0.0

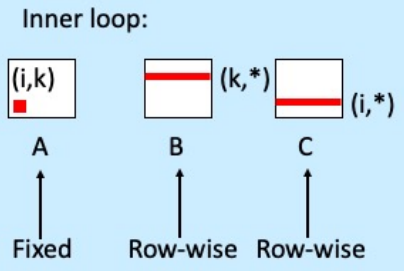
Supplied by CMU.

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];
  }
}
```

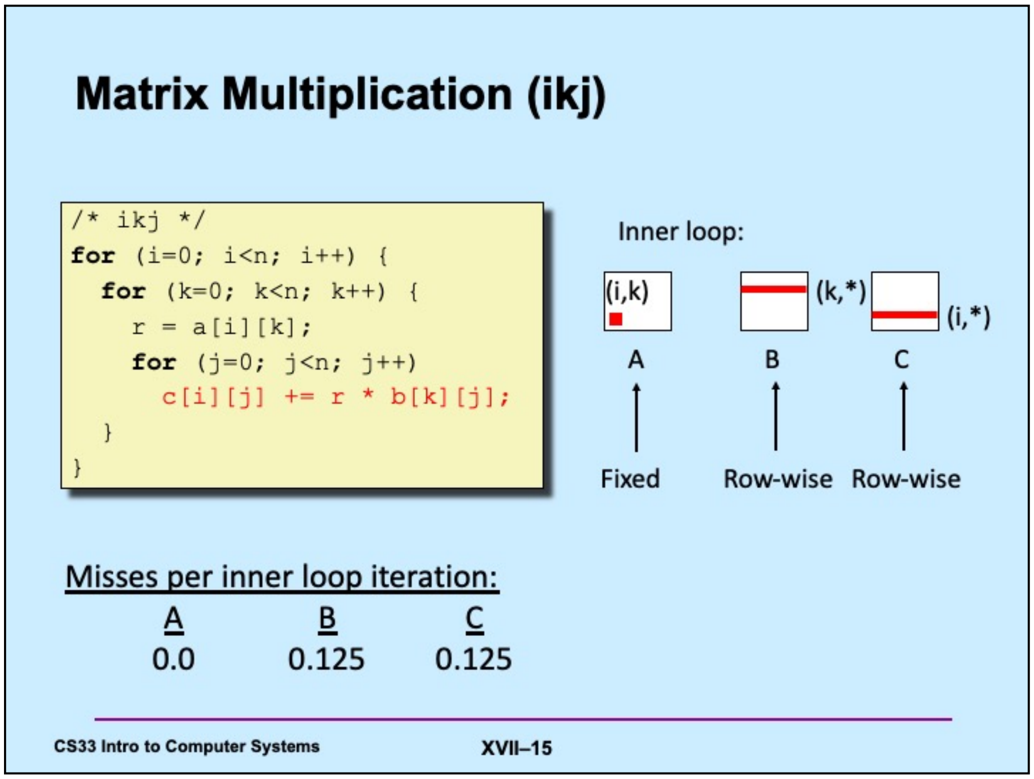


Misses per inner loop iteration:

A	B	C
0.0	0.125	0.125

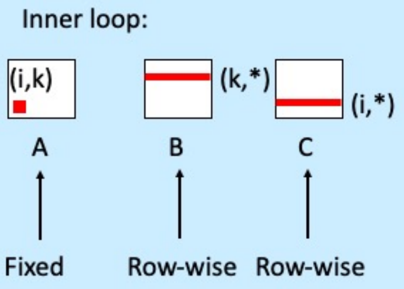
Supplied by CMU.

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];
  }
}
```

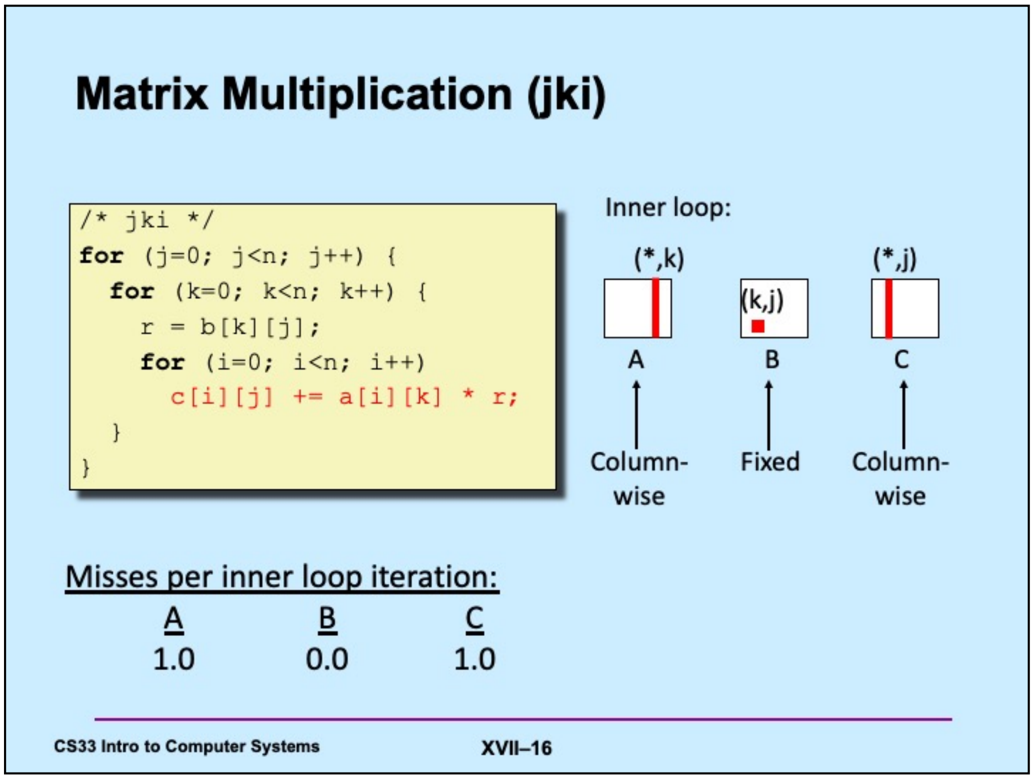


Misses per inner loop iteration:

A	B	C
0.0	0.125	0.125

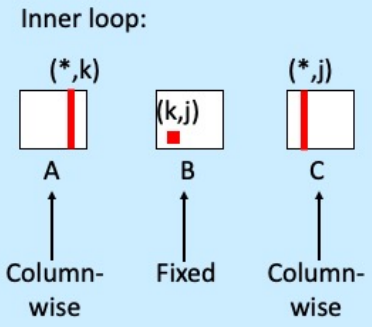
Supplied by CMU.

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;
  }
}
```

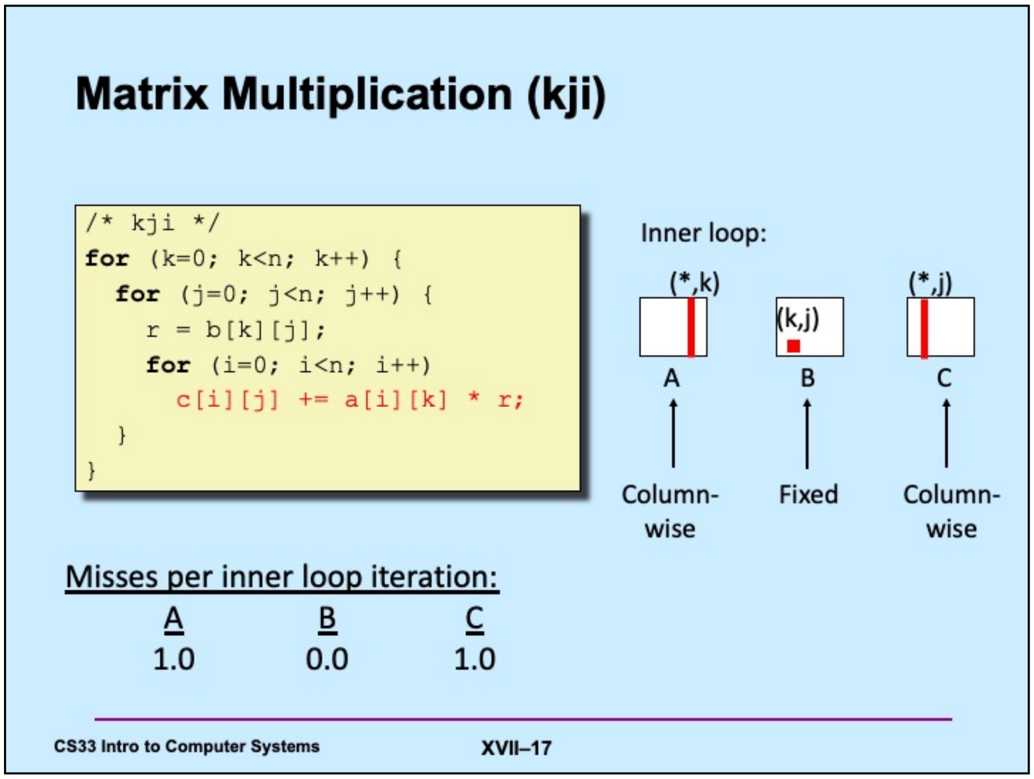


Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

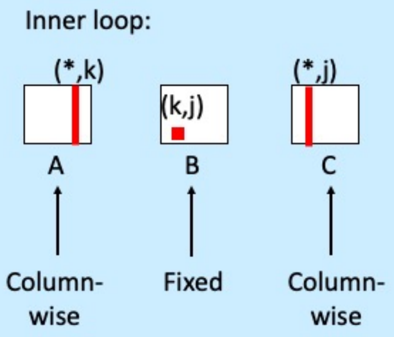
Supplied by CMU.

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;
  }
}
```

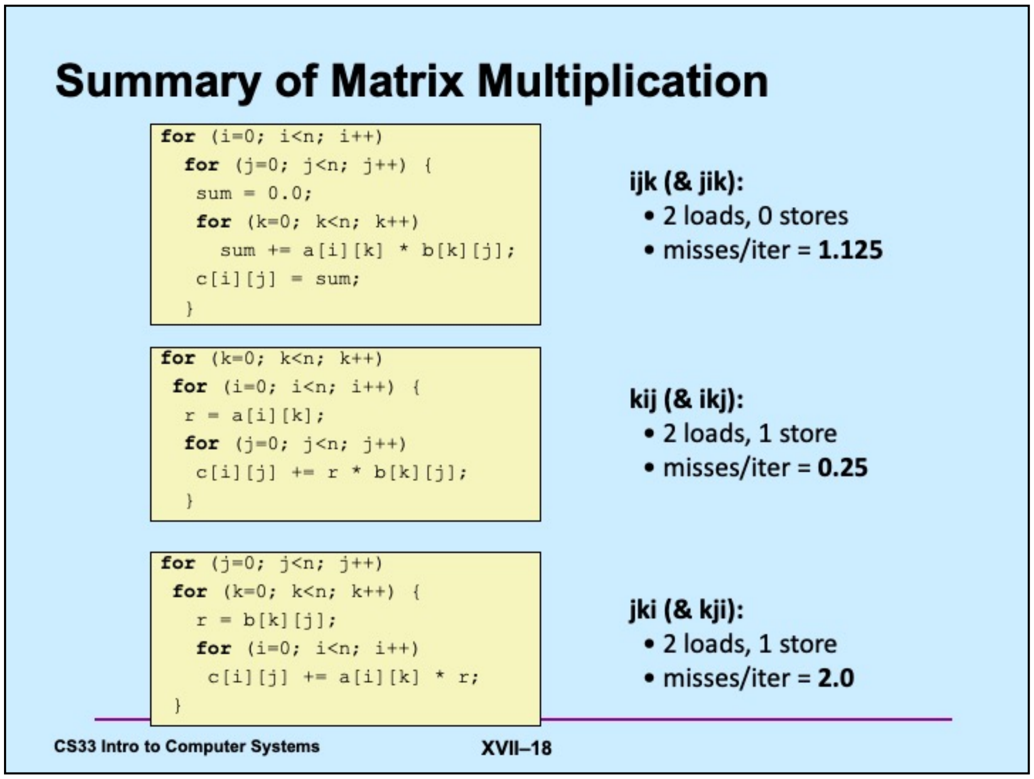


Misses per inner loop iteration:

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**

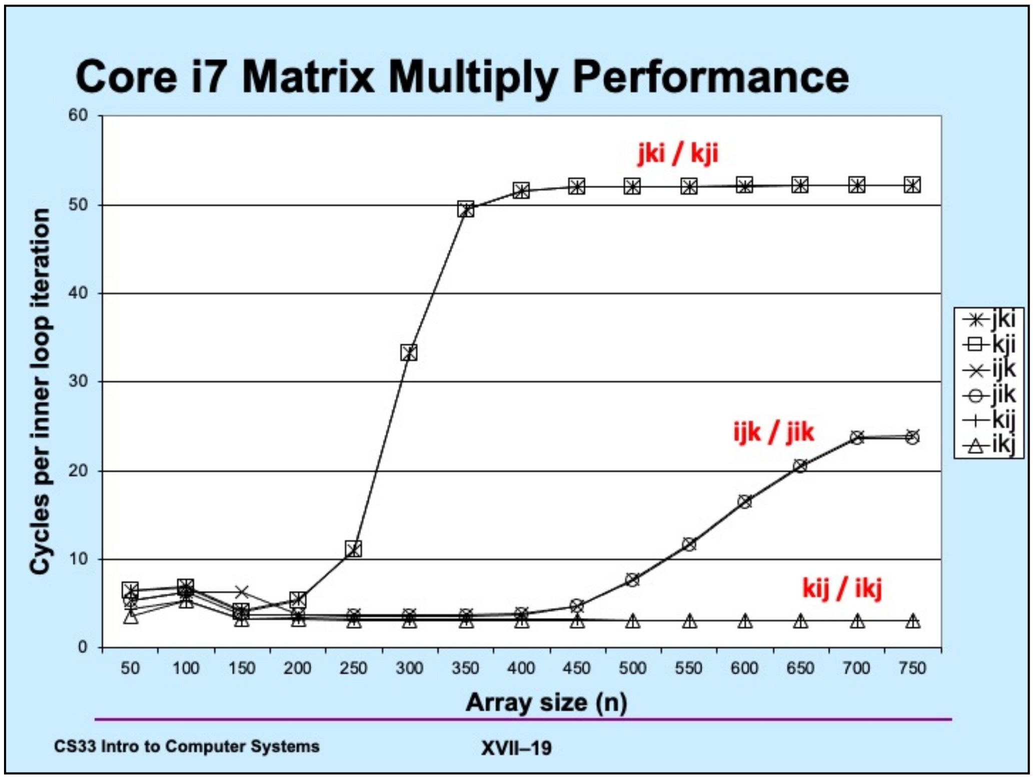
```
for (k=0; k<n; k++)
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
  }
```

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.

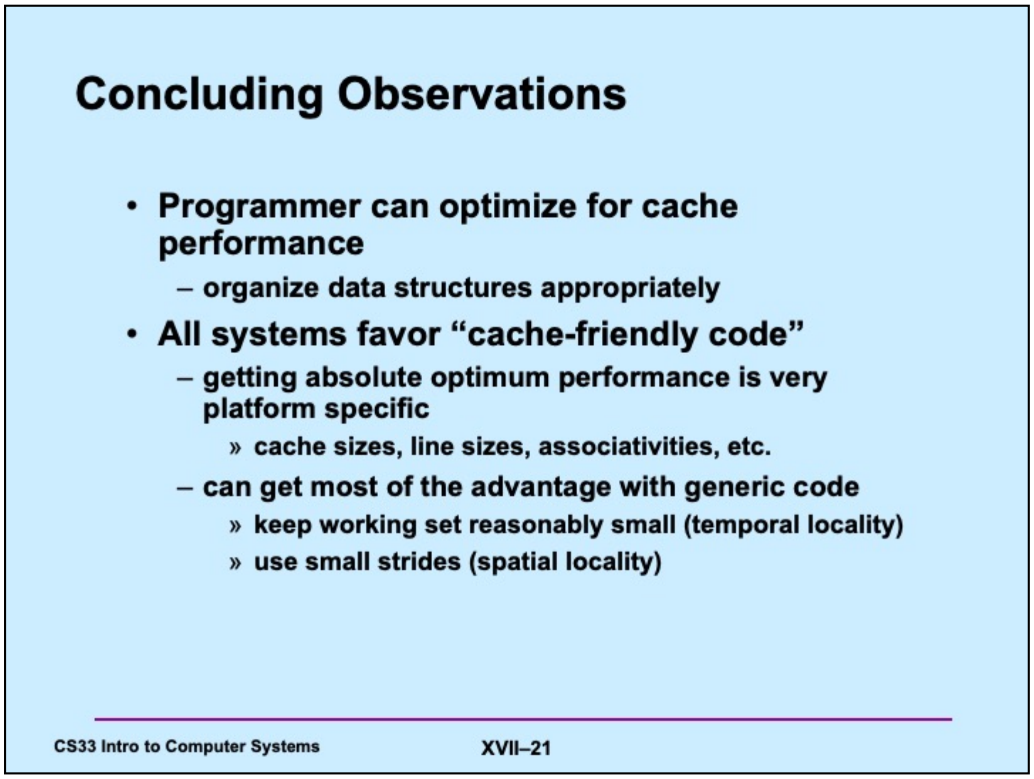


Core i7 Matrix Multiply Performance

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

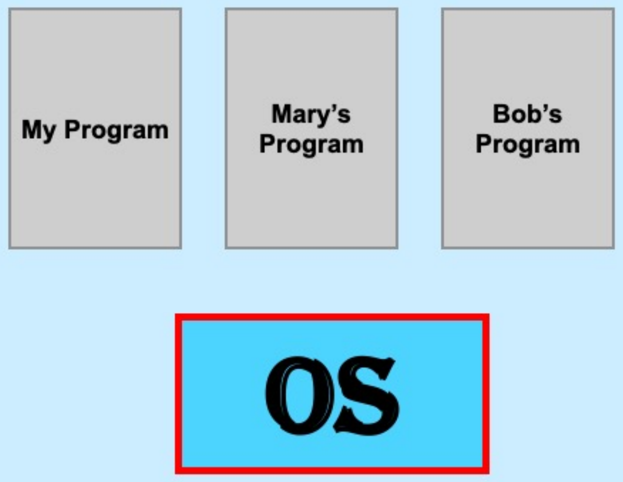
- Programmer can optimize for cache performance
  - organize data structures appropriately
- All systems favor "cache-friendly code"
  - getting absolute optimum performance is very platform specific
    - » cache sizes, line sizes, associativities, etc.
  - can get most of the advantage with generic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

Supplied by CMU.

CS 33

Architecture and the OS

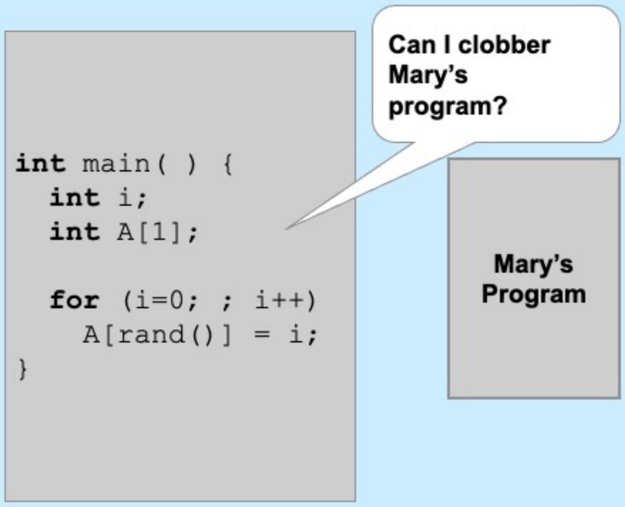
The Operating System



Processes

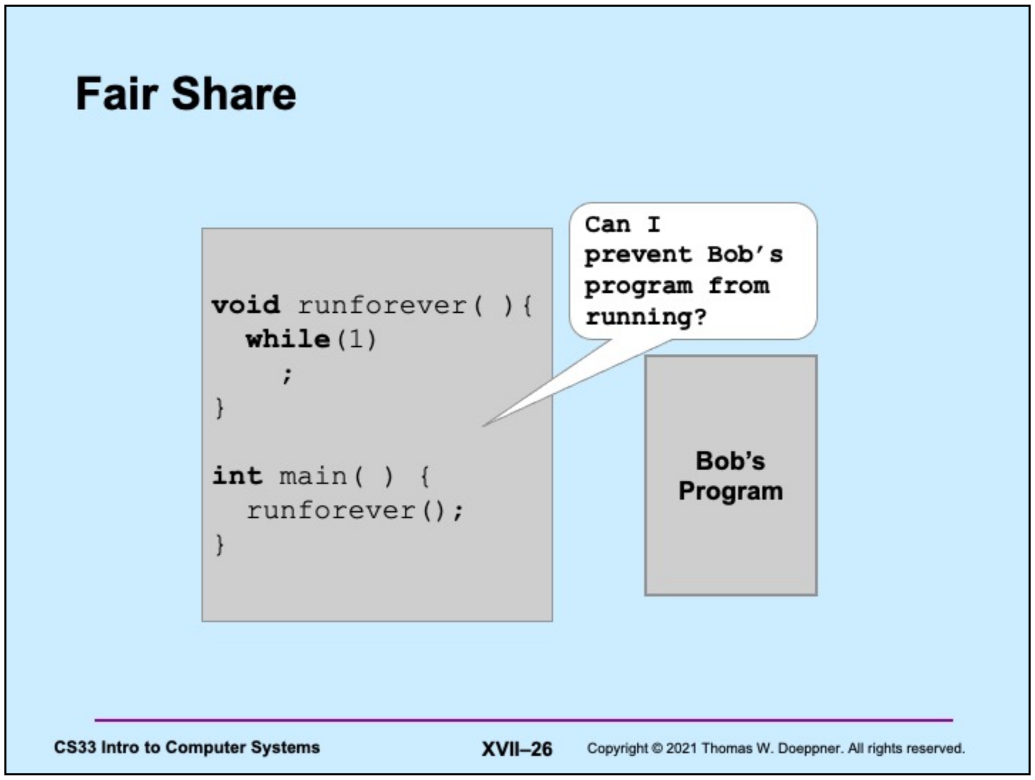
- Containers for programs
  - virtual memory
    - » address space
  - scheduling
    - » one or more threads of control
  - file references
    - » open files
  - and lots more!

Idiot Proof …



```
int main( ) {
  int i;
  int A[1];

  for (i=0; ; i++)
    A[rand()] = i;
}
```



Fair Share

```
void runforever( ){
  while(1)
    ;
}

int main( ) {
  runforever();
}
```

Can I prevent Bob's program from running?
Bob's Program

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?

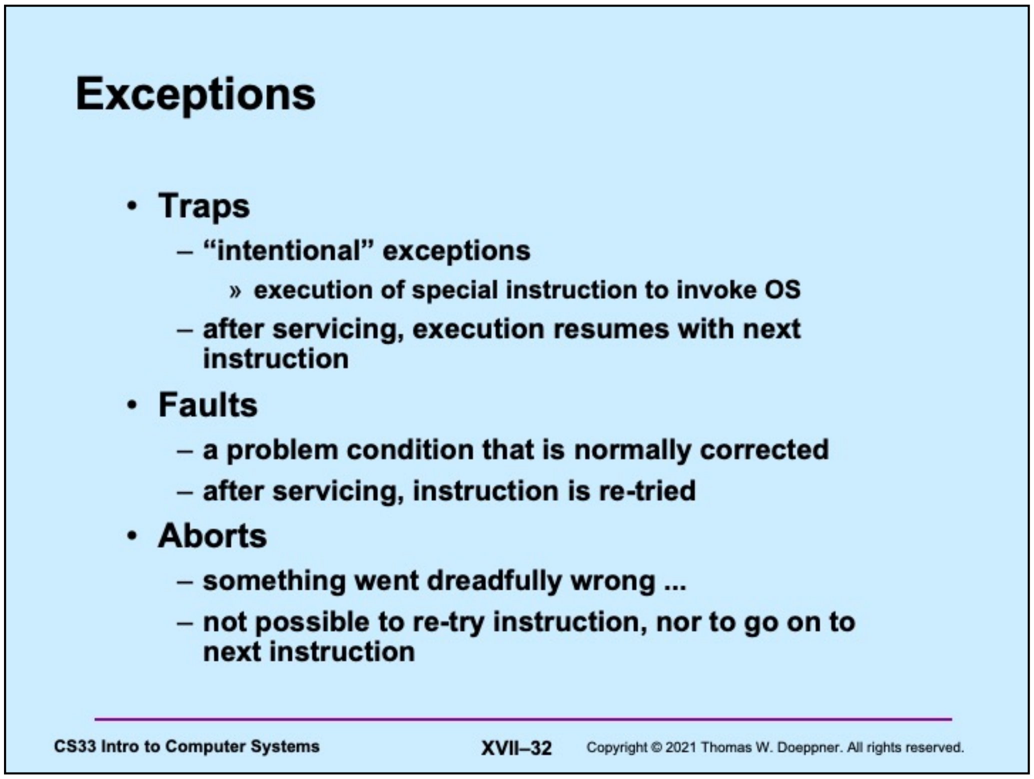
- No one
  - user code always runs in user mode
- The operating-system kernel runs in privileged mode
  - nothing else does
  - not even super user on Unix or administrator on Windows

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

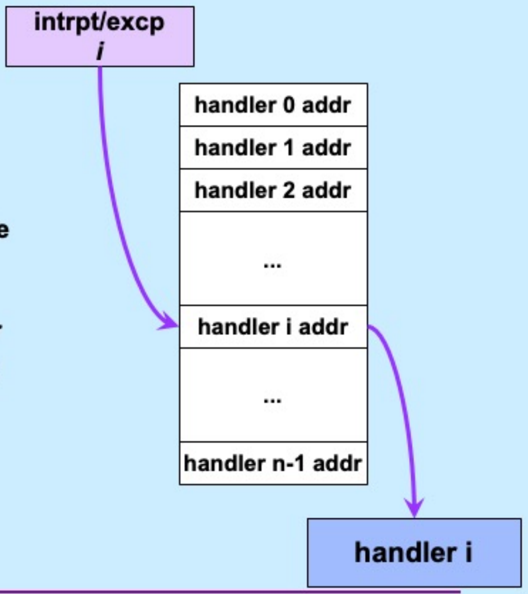
These definitions follow those given in "Intel® 64 and IA-32 Architectures Software Developer's Manual" and are generally accepted even outside of Intel.

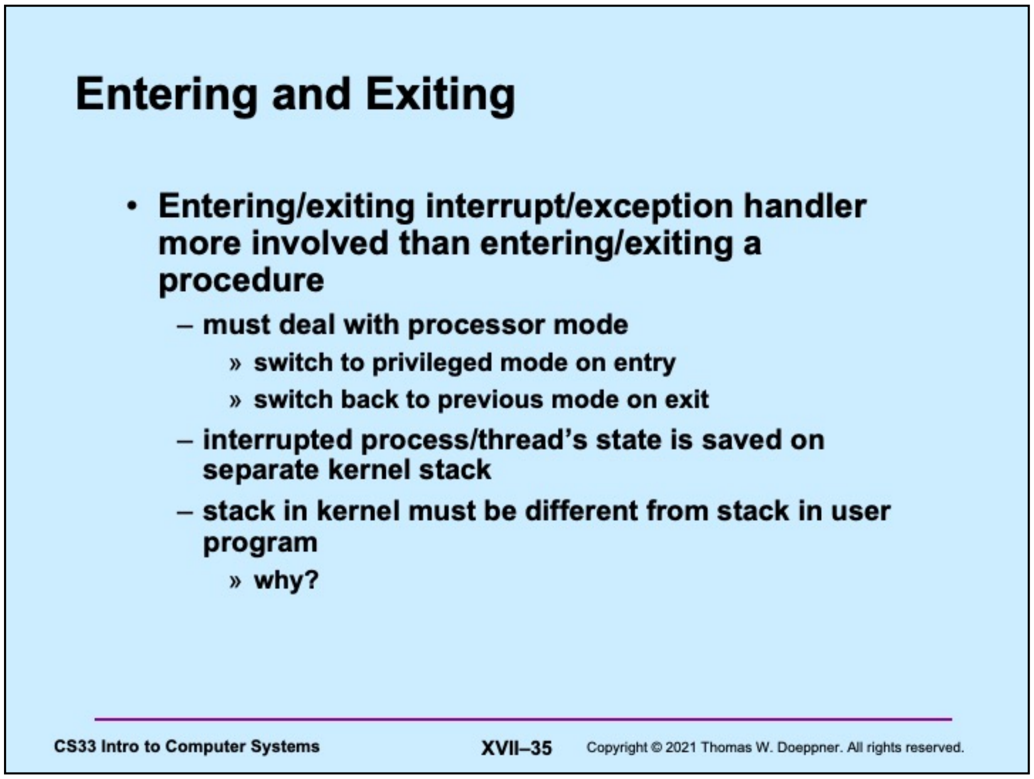
Actions for Interrupts and Exceptions

- 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

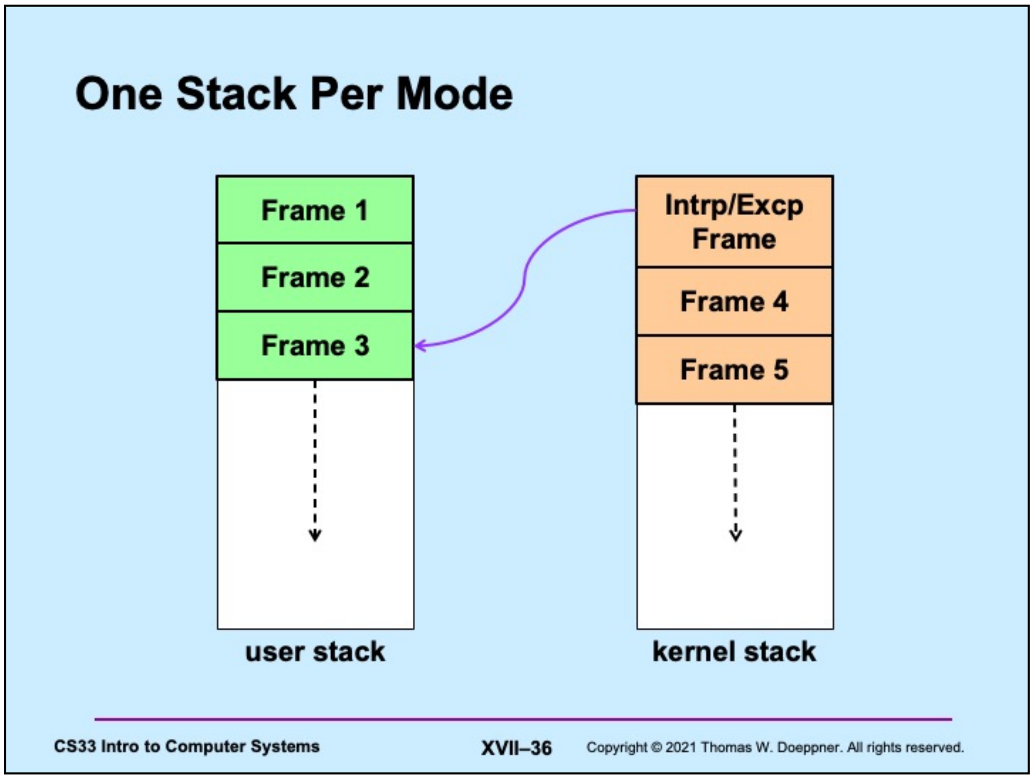




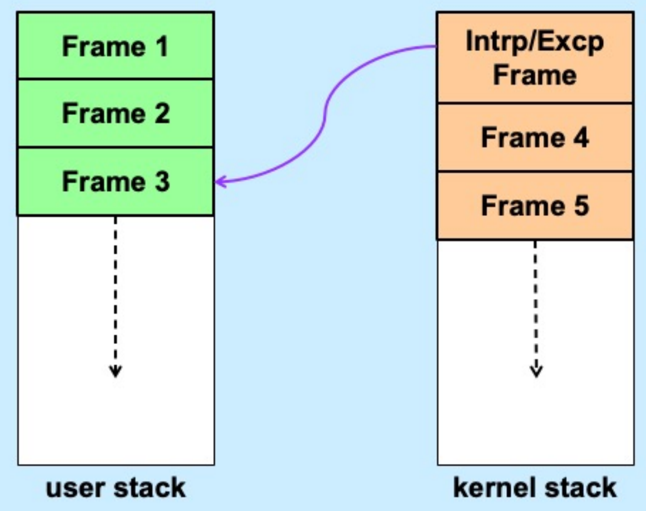
Entering and Exiting

- 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.



One Stack Per Mode



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.

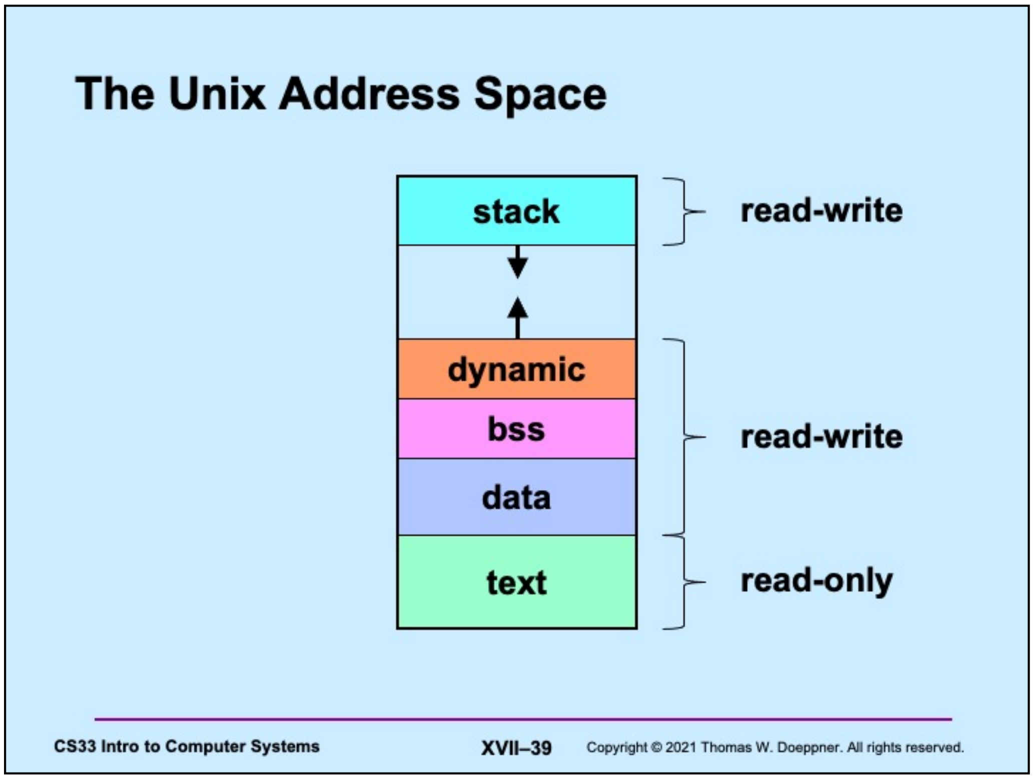
Quiz 2

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

Back to the x86 ...

- It's complicated
  - more than it should be, but for historical reasons ...
- Not just privileged and non-privileged modes, but four "privilege levels"
  - level 0
    - » most privileged, used by OS kernel
  - level 1
    - » not normally used
  - level 2
    - » not normally used
  - level 3
    - » least privileged, used by application code



The Unix Address Space

A Unix process's address space appears to be three regions of memory: a read-only *text* region (containing executable code); a read-write region consisting of initialized *data* (simply called data), uninitialized data (*BSS* — a directive from an ancient assembler (for the IBM 704 series of computers) standing for Block Started by Symbol and used to reserve space for uninitialized storage), and a *dynamic area*; and a second read-write region containing the process's user *stack* (a standard Unix process contains only one thread of control).

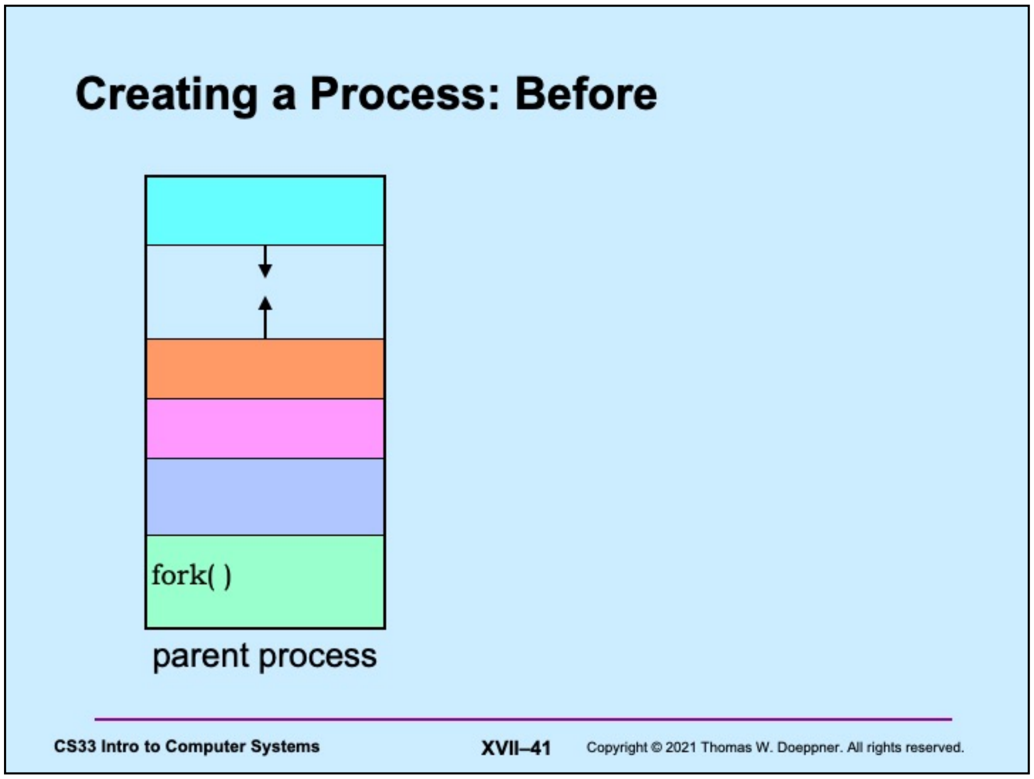
The first area of read-write storage is often collectively called the data region. Its dynamic portion grows in response to **sbrk** system calls. Most programmers do not use this system call directly, but instead use the **malloc** and **free** library routines, which manage the dynamic area and allocate memory when needed by in turn executing **sbrk** system calls.

The stack region grows implicitly: whenever an attempt is made to reference beyond the current end of stack, the stack is implicitly grown to the new reference. (There are system-wide and per-process limits on the maximum data and stack sizes of processes.)

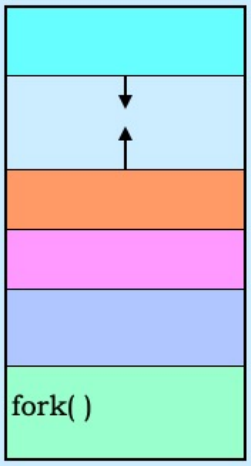
Note that here we are discussing strictly single-threaded processes. Later we will discuss multi-threaded processes, whose address spaces contain multiple stacks.

Creating Your Own Processes

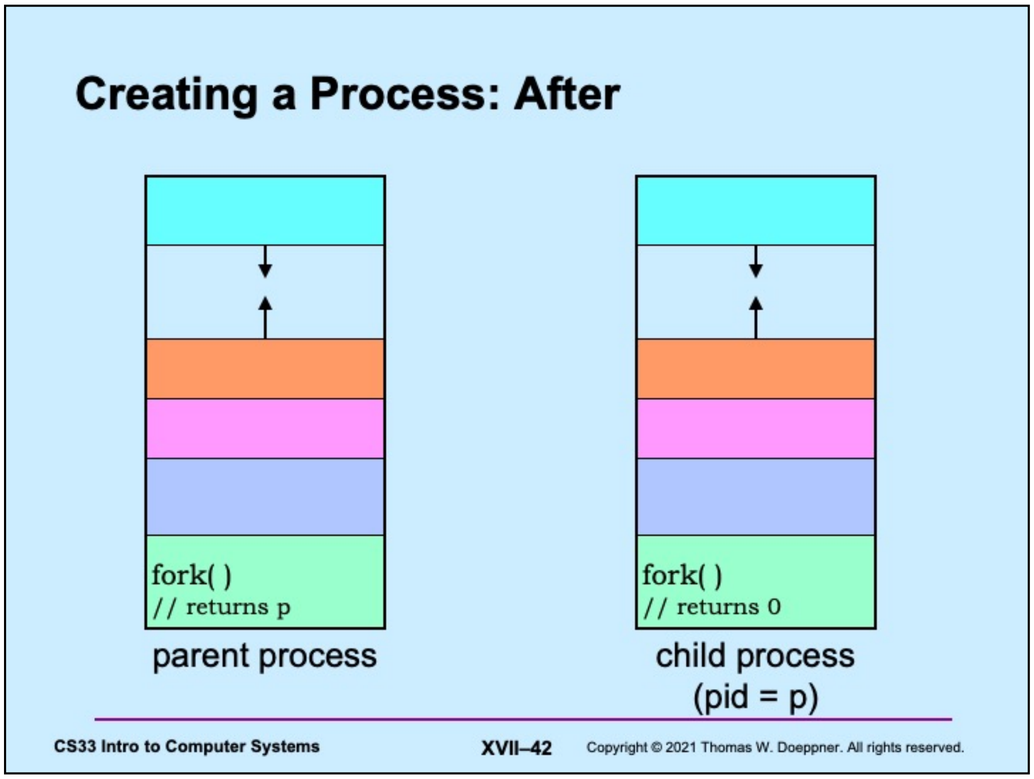
```
#include <unistd.h>
int main( ) {
  pid_t pid;
  if ((pid = fork()) == 0) {
       /* new process starts
          running here */
  }
  /* old process continues
     here */
}
```



Creating a Process: Before



The only way to create a new process is to use the **fork** system call.



Creating a Process: After

By executing **fork** the parent process creates an almost exact clone of itself that we call the child process. This new process executes the same text as its parent, but contains a copy of the data and a copy of the stack. This copying of the parent to create the child can be very time-consuming if done naively. Some tricks are employed to make it much less so.

Fork is a very unusual system call: one thread of control flows into it but two threads of control flow out of it, each in a separate address space. From the parent’s point of view, fork does very little: nothing happens to the parent except that fork returns the process ID (PID — an integer) of the new process. The new process starts off life by returning from fork, which it sees as returning a zero.

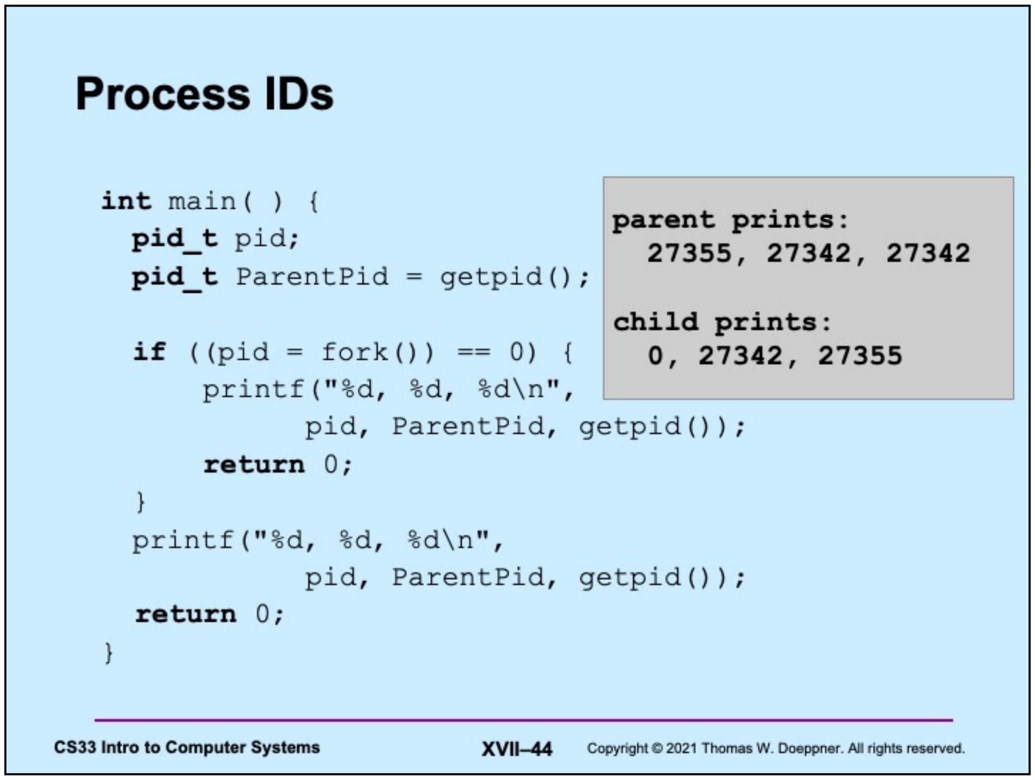
Quiz 3

The following program

a) runs forever

b) terminates quickly

```
int flag;
int main() {
  while (flag == 0) {
    if (fork() == 0) {
      // in child process
      flag = 1;
      exit(0);  // causes process to terminate
    }
  }
}
```



Process IDs

```
int main( ) {
  pid_t pid;
  pid_t ParentPid = getpid();

  if ((pid = fork()) == 0) {
      printf("%d, %d, %d\n",
             pid, ParentPid, getpid());
      return 0;
  }
  printf("%d, %d, %d\n",
             pid, ParentPid, getpid());
  return 0;
}
```

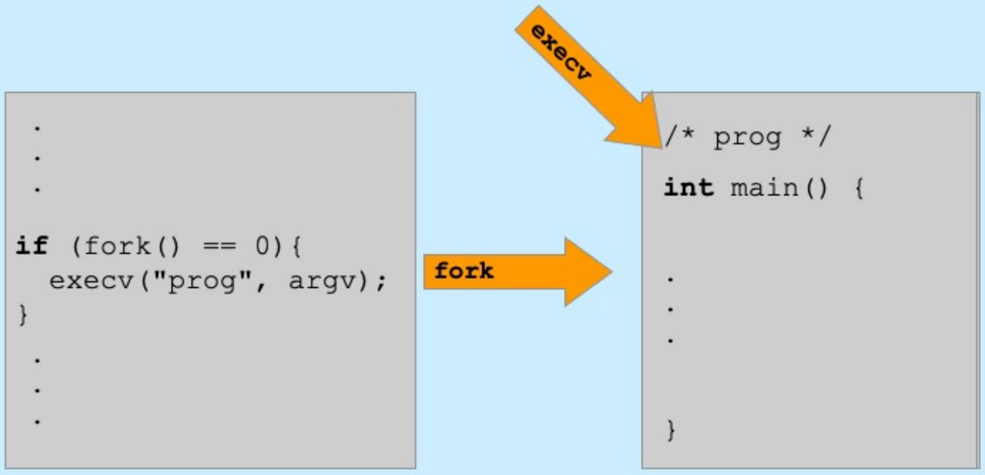
```
parent prints:
  27355, 27342, 27342

child prints:
  0, 27342, 27355
```

The **getpid** function returns the caller's process ID.

The parent process executes the second printf; the child process executes the first printf.

Putting Programs into Processes

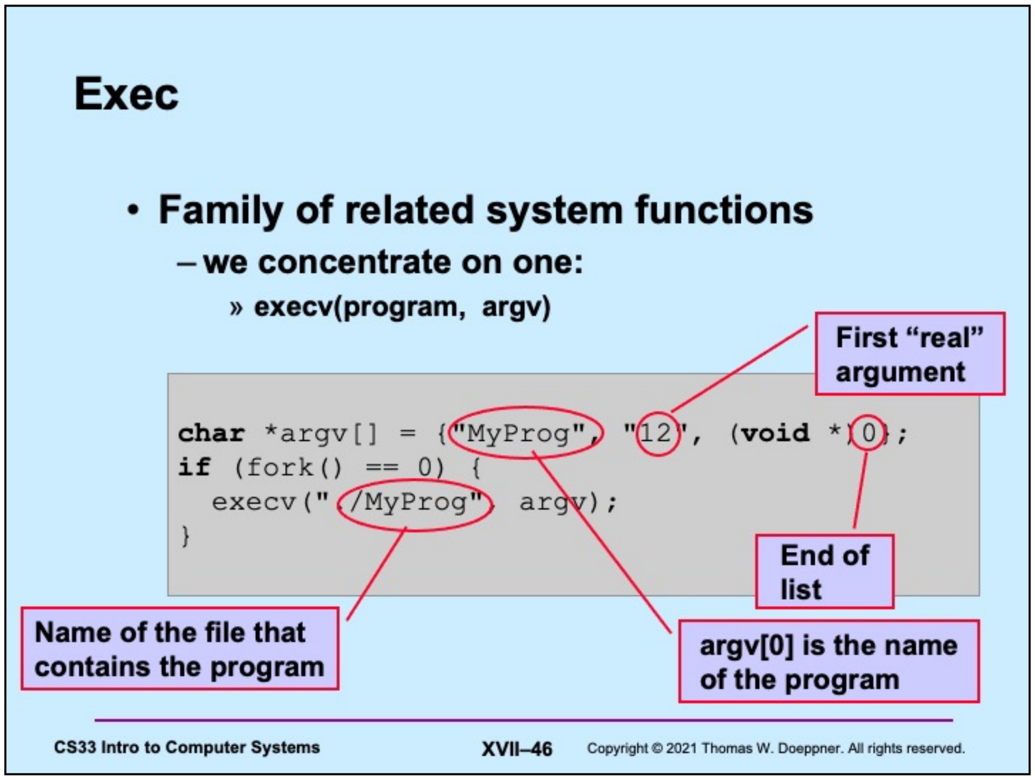


```
 .
 .
 .
if (fork() == 0){
  execv("prog", argv);
}
 .
 .
 .
```

```
/* prog */
int main() {

 .
 .
 .

}
```



Exec

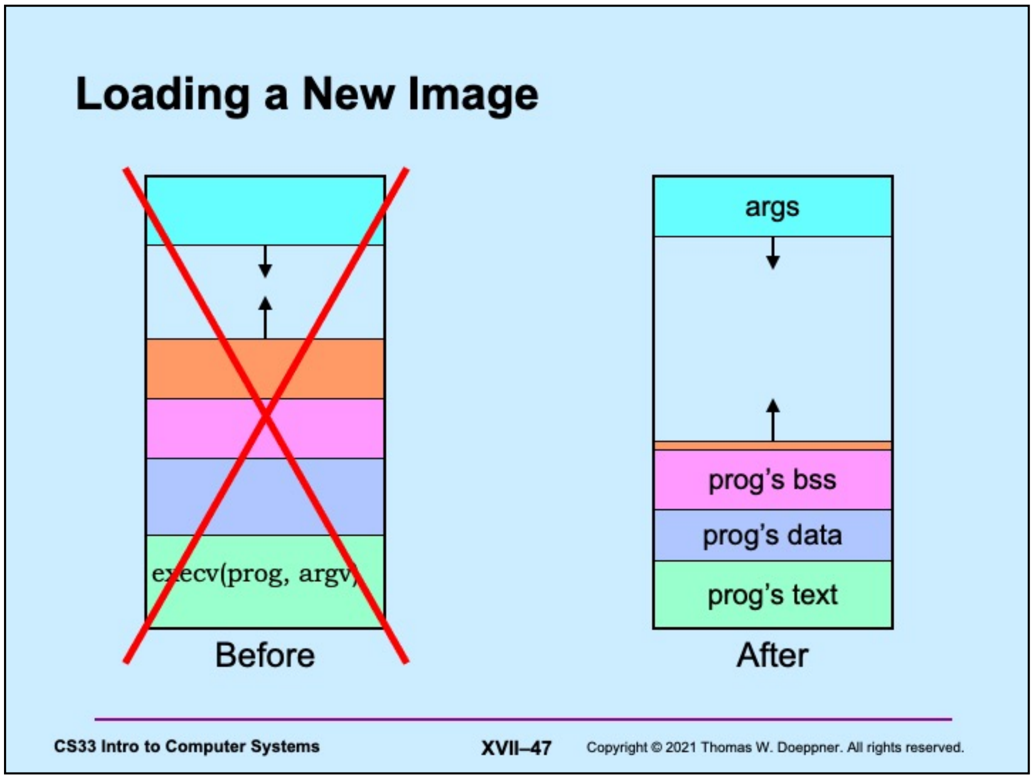
- Family of related system functions
  - we concentrate on one:
    - » execv(program, argv)

```
char *argv[] = {"MyProg", "12", (void *)0};
if (fork() == 0) {
  execv("./MyProg", argv);
}
```

First “real” argument
End of list
Name of the file that contains the program
argv[0] is the name of the program

Note that the name of the program might be different from the name of the file that contains the program, but usually the name of the program is the last component of the file’s pathname.

Note that a null pointer, termed a *sentinel*, is used to indicate the end of the list of arguments.



Loading a New Image

Most of the time the purpose of creating a new process is to run a new (i.e., different) program. Once a new process has been created, it can use one of the *exec* system calls to load a new program image into itself, replacing the prior contents of the process's address space. Exec is passed the name of a file containing an executable program image. The previous text region of the process is replaced with the text of the program image. The data, BSS and dynamic areas of the process are "thrown away" and replaced with the data and BSS of the program image. The contents of the process's stack are replaced with the arguments that are passed to the main procedure of the program.

A Random Program …

```
int main(int argc, char *argv[]) {
 if (argc != 2) {
    fprintf(stderr, "Usage: random count\n");
    exit(1);
  }
  int stop = atoi(argv[1]);
  for (int i = 0; i < stop; i++)
    printf("%d\n", rand());
  return 0;
}
```

Passing It Arguments

- **From the shell**

```
$ random 12
```

- **From a C program**

```
if (fork() == 0) {
  char *argv[] = {"random", "12", (void *)0};
  execv("./random", argv);
}
```

Quiz 4

```
if (fork() == 0) {
  char *argv[] = {"random", "12", (void *)0};
  execv("./random", argv);
  printf("random done\n");
}
```

The *printf* statement will be executed

- **a) only if execv fails**
- **b) only if execv succeeds**
- **c) always**