CS 33

Exploiting Caches

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

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!)

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:

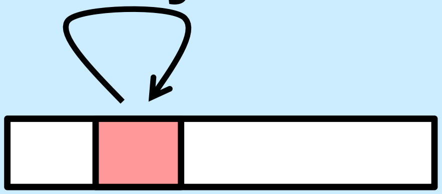
    97% hits: .97 * 1 cycle + 0.03 * 100 cycles ≈ 4 cycles

    99% hits: .99 * 1 cycle + 0.01 * 100 cycles ≈ 2 cycles

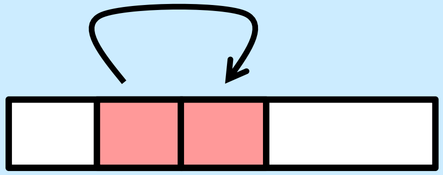
- **This is why "miss rate" is used instead of "hit rate"**

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

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

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;
}
```

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)

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

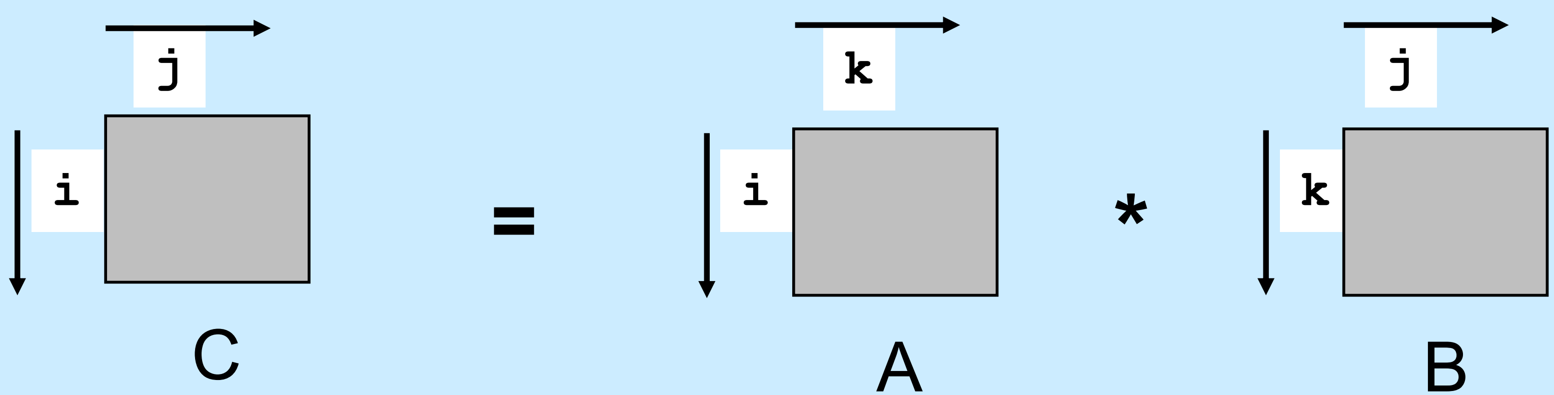
Variable sum held in register

```
/* 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];
  }
}
```

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

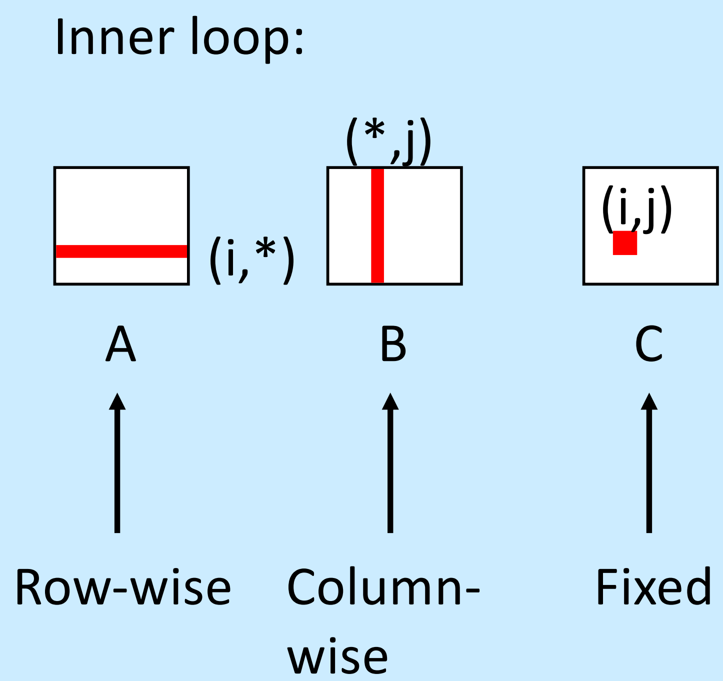


Layout of C Arrays in Memory (review)

- **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%)**

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;
  }
}
```

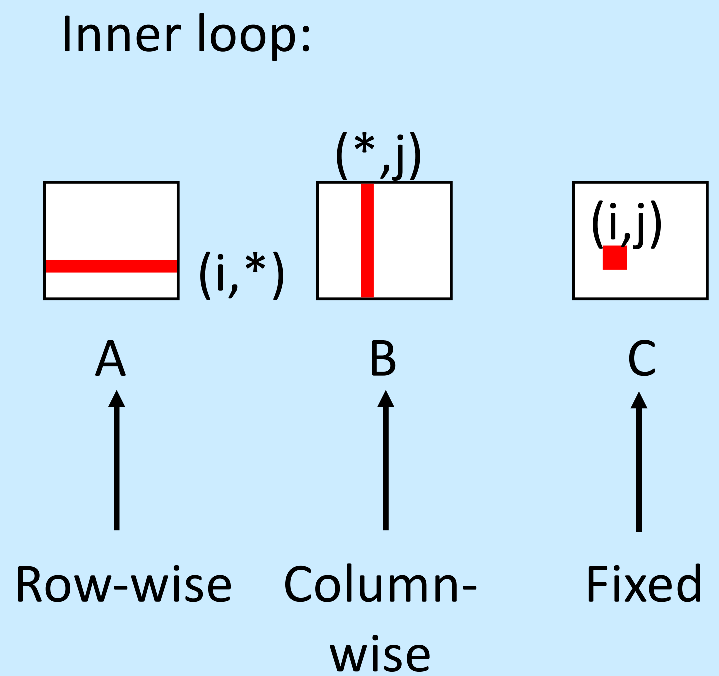


Misses per inner loop iteration:

A	B	C
0.125	1.0	0.0

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
  }
}
```



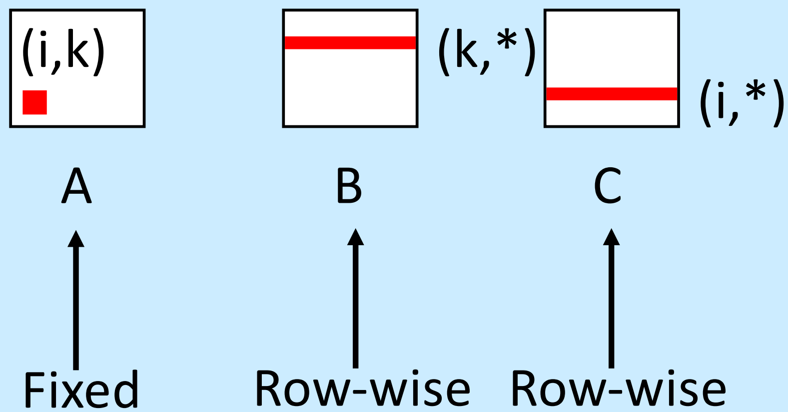
<u>Misses per inner loop iteration:</u>

<u>A</u>	<u>B</u>	<u>C</u>
0.125	1.0	0.0

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];
  }
}
```

Inner loop:

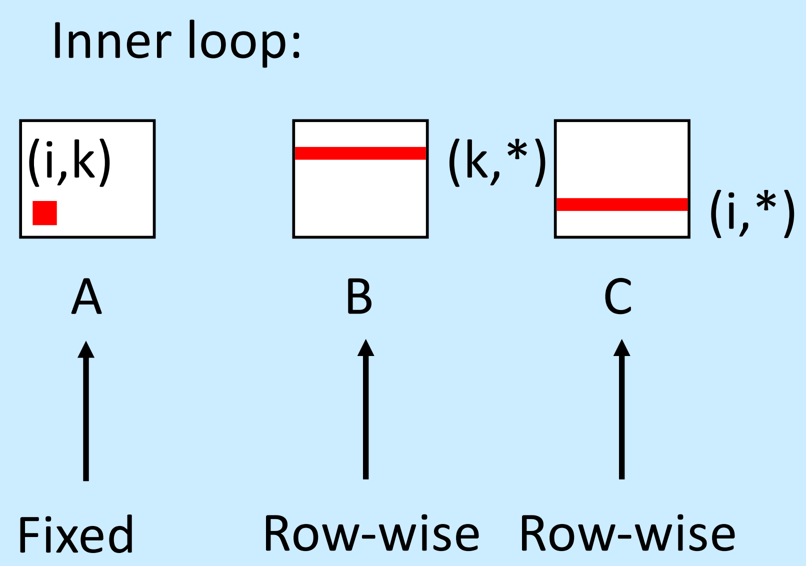


Misses per inner loop iteration:

A	B	C
0.0	0.125	0.125

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];
  }
}
```



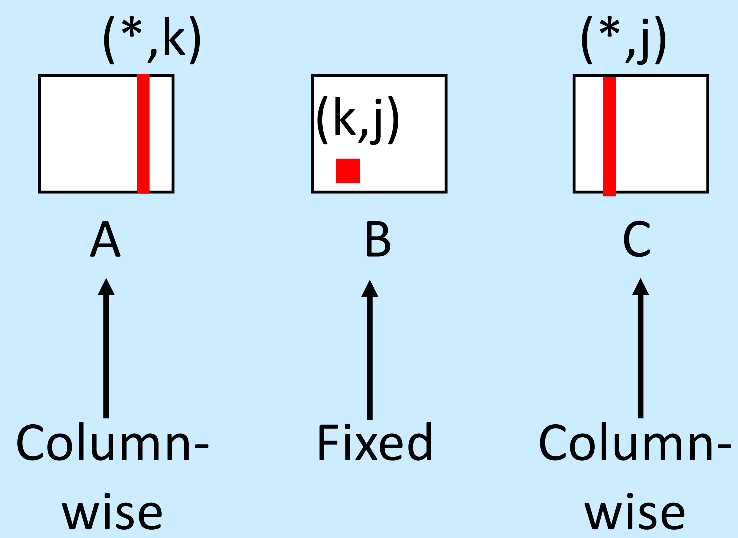
Misses per inner loop iteration:

A	B	C
0.0	0.125	0.125

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;
  }
}
```

Inner loop:



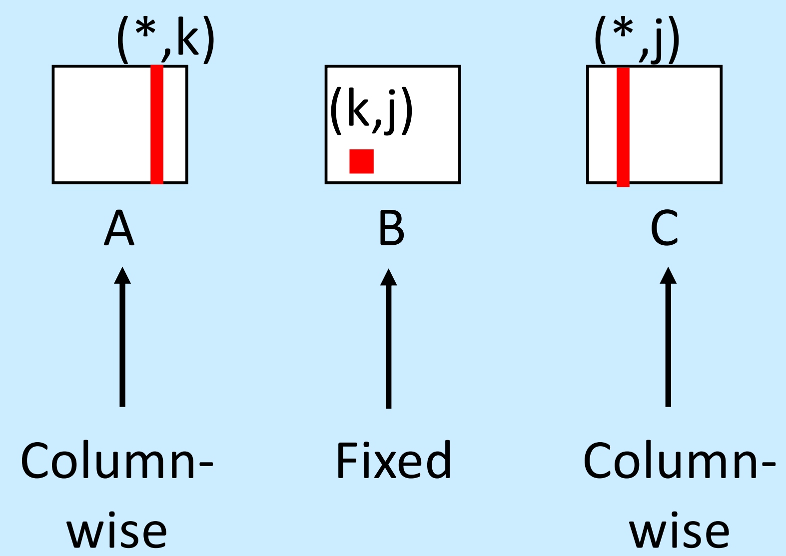
<u>Misses per inner loop iteration:</u>

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

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;
  }
}
```

Inner loop:



Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

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):

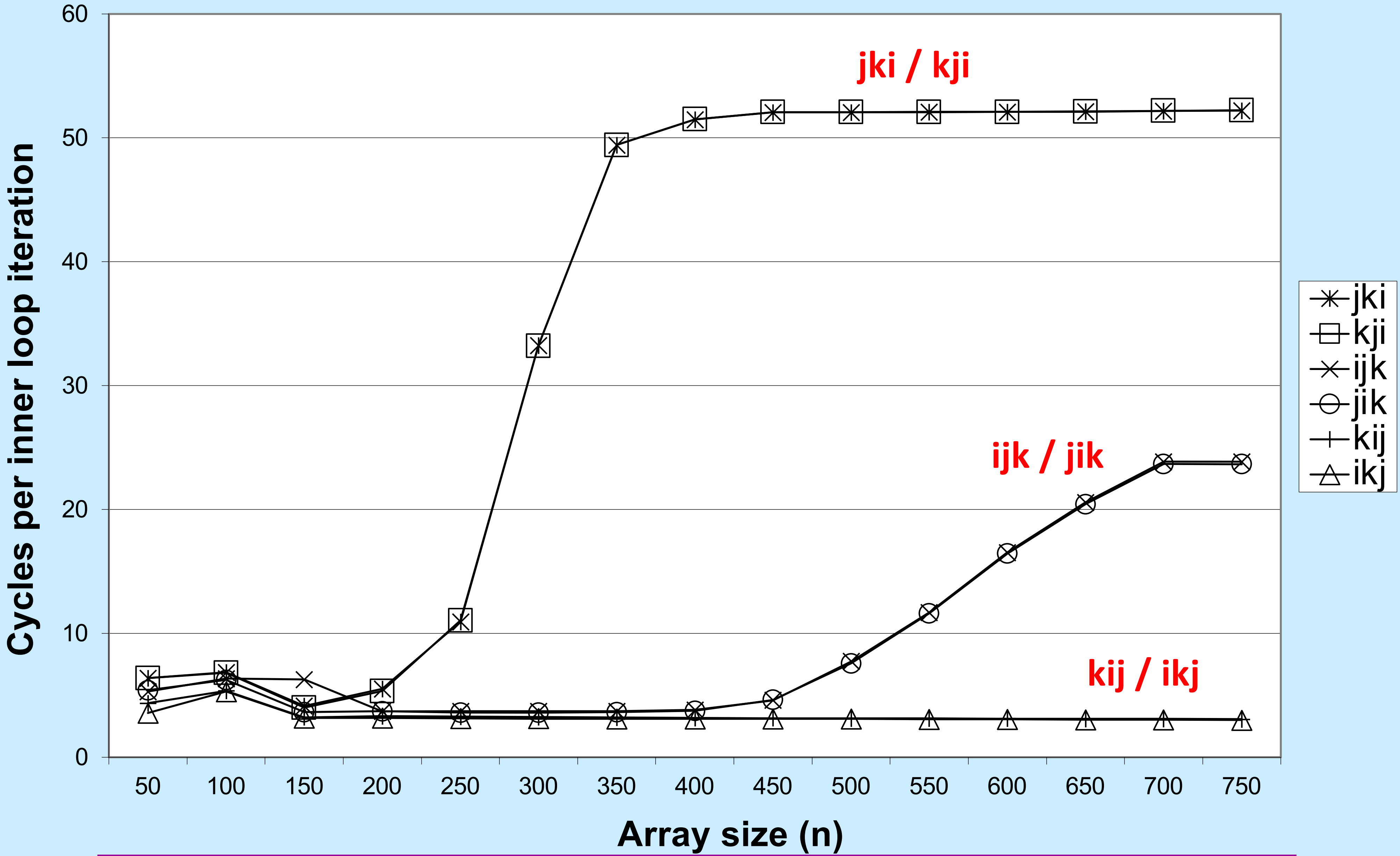
- 2 loads, 1 store
- misses/iter = **0.25**

```
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**

Core i7 Matrix Multiply Performance



In Real Life ...

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

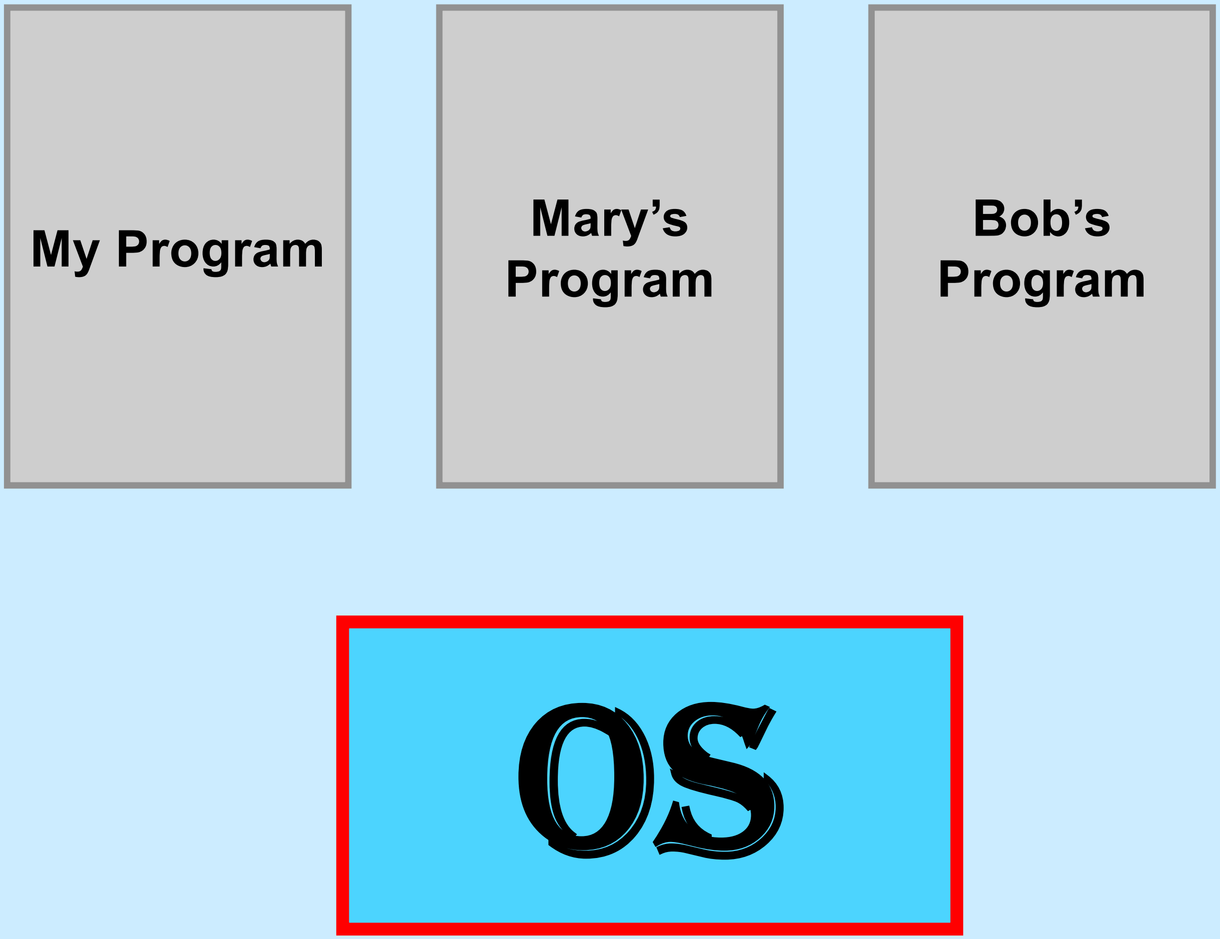
Concluding Observations

- **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)

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;
}
```

Can I clobber Mary's program?

Mary's Program

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

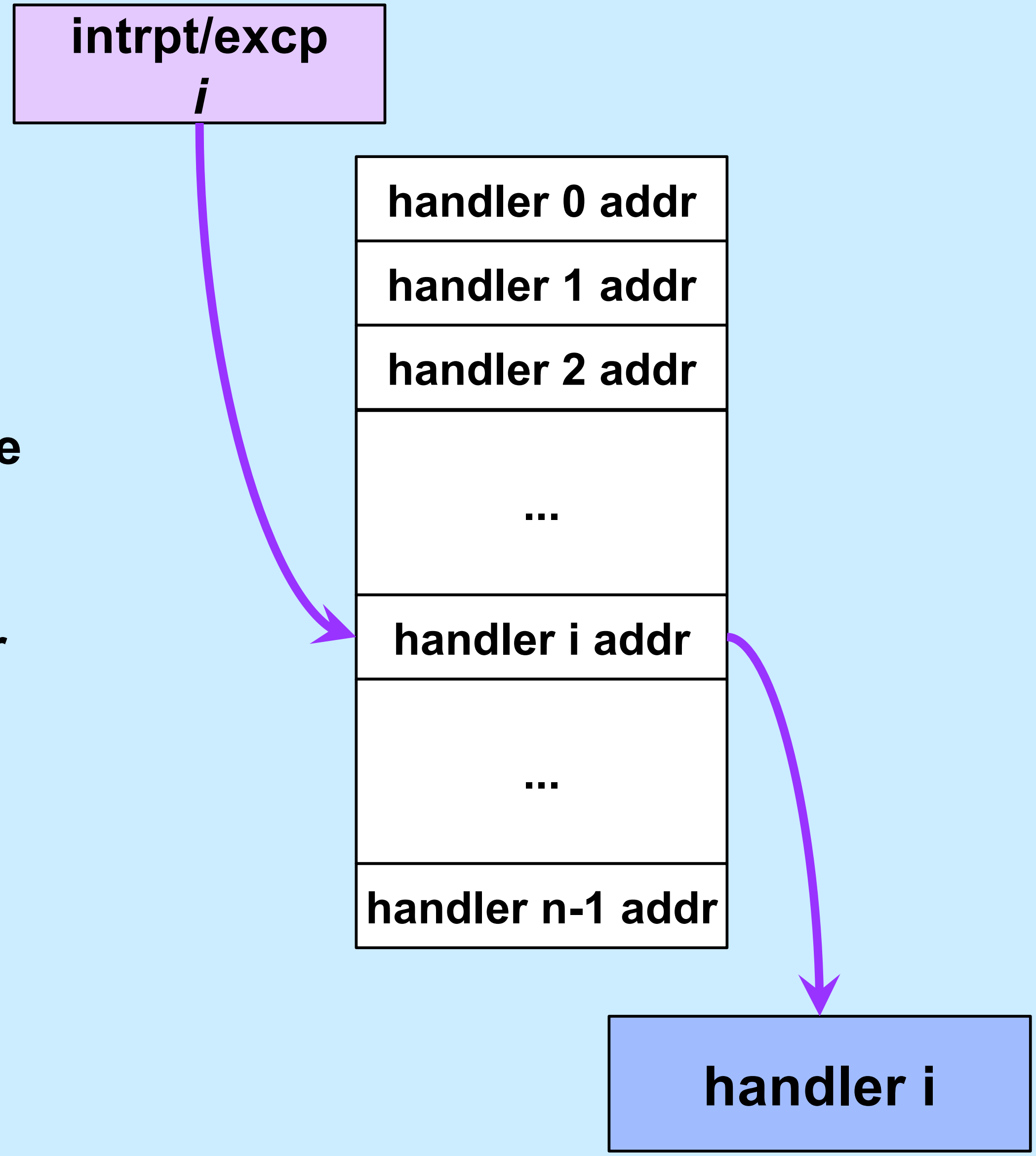
- **Traps**
  - "intentional" exceptions
    - » execution of special instruction to invoke OS
  - after servicing, execution resumes with next instruction
- **Faults**
  - a problem condition that is normally corrected
  - after servicing, instruction is re-tried
- **Aborts**
  - something went dreadfully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

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

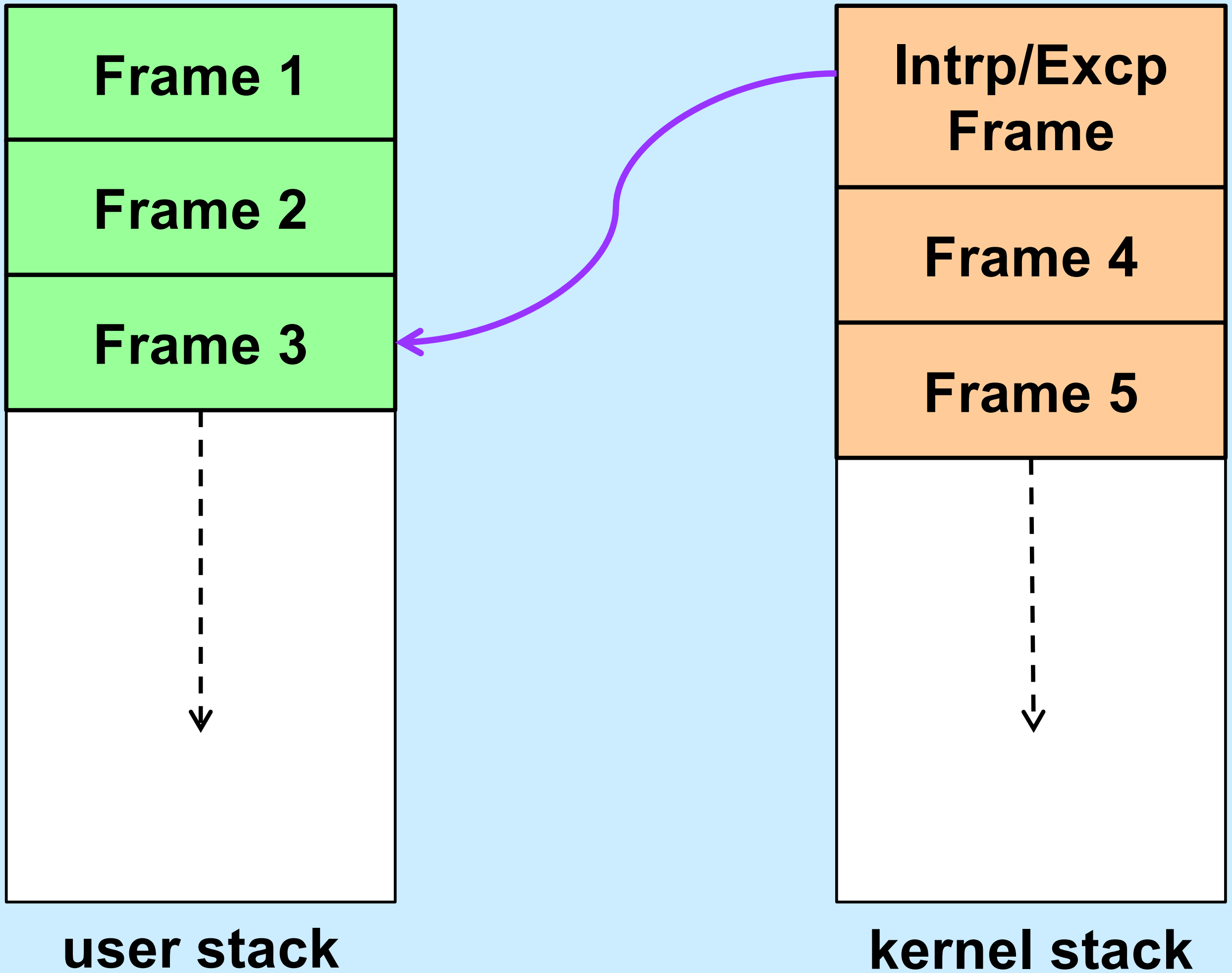
- Interrupt or exception invokes handler (in OS)
  - via interrupt and exception vector
    - » one entry for each possible interrupt/exception
      - contains
        - address of handler
  - code executed in privileged mode
    - » but code is part of the OS



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?

One Stack Per Mode



Quiz 2

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

a) all

b) none

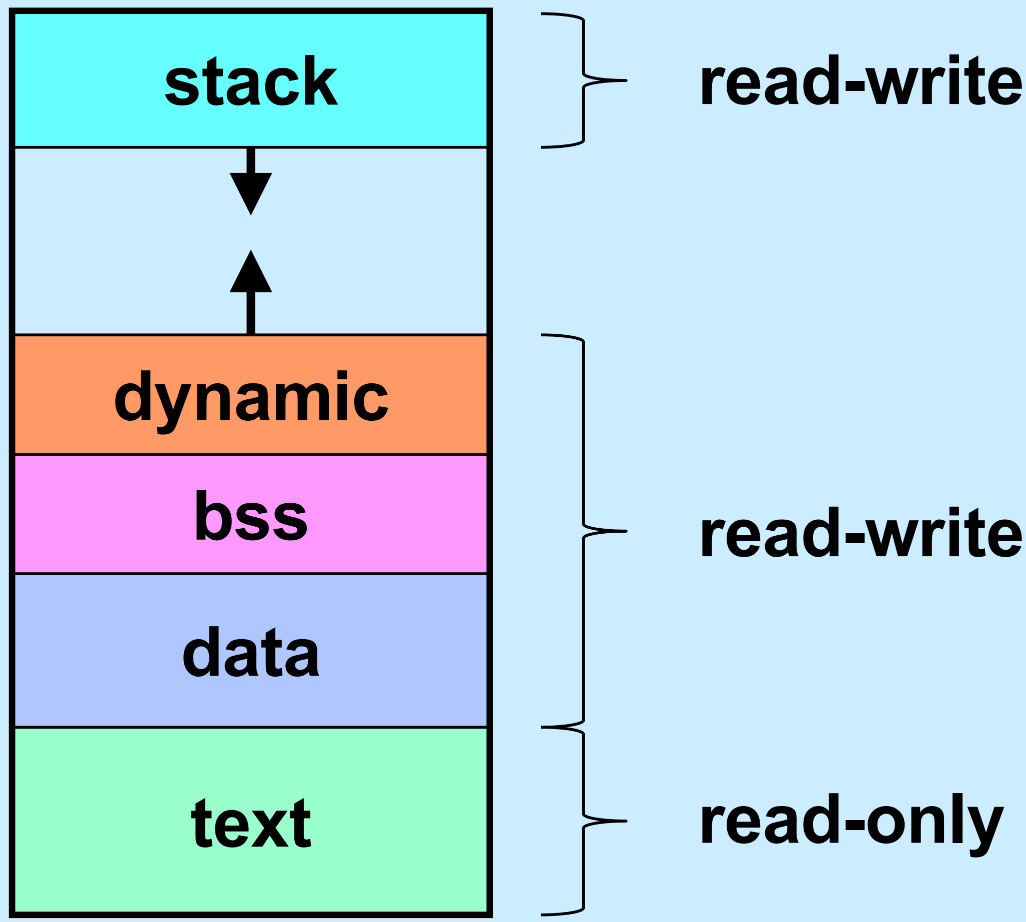
c) callee-save registers

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

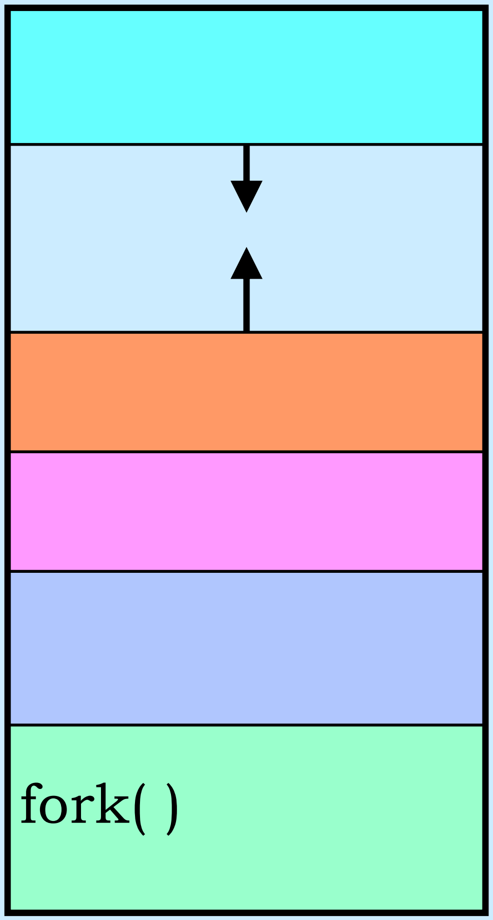
The Unix Address Space



Creating Your Own Processes

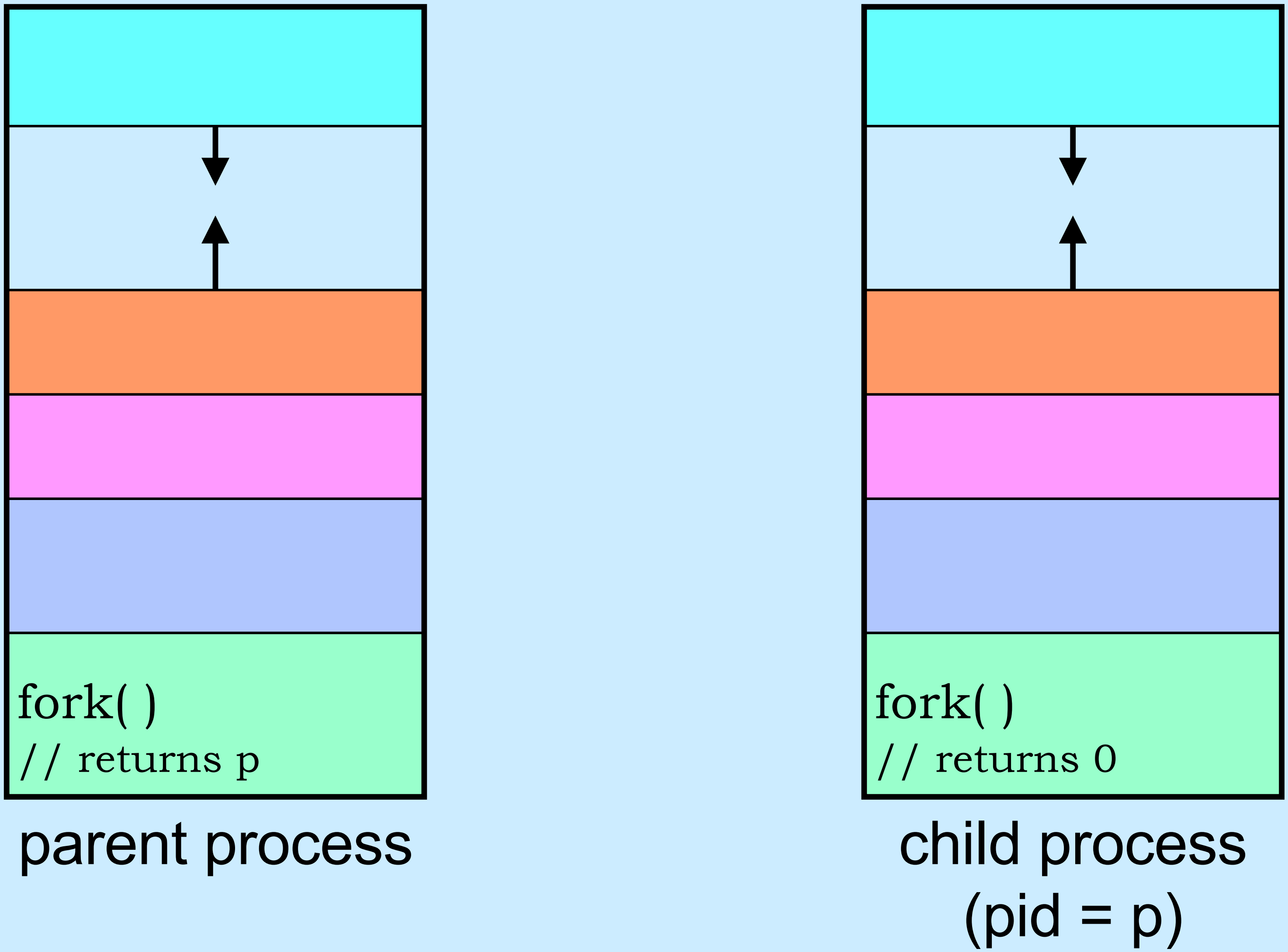
```
#include <unistd.h>
int main( ) {
  pid_t pid;
  if ((pid = fork()) == 0) {
      /* new process starts
         running here */
  }
  /* old process continues
     here */
}
```


Creating a Process: Before



parent process

Creating a Process: After



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

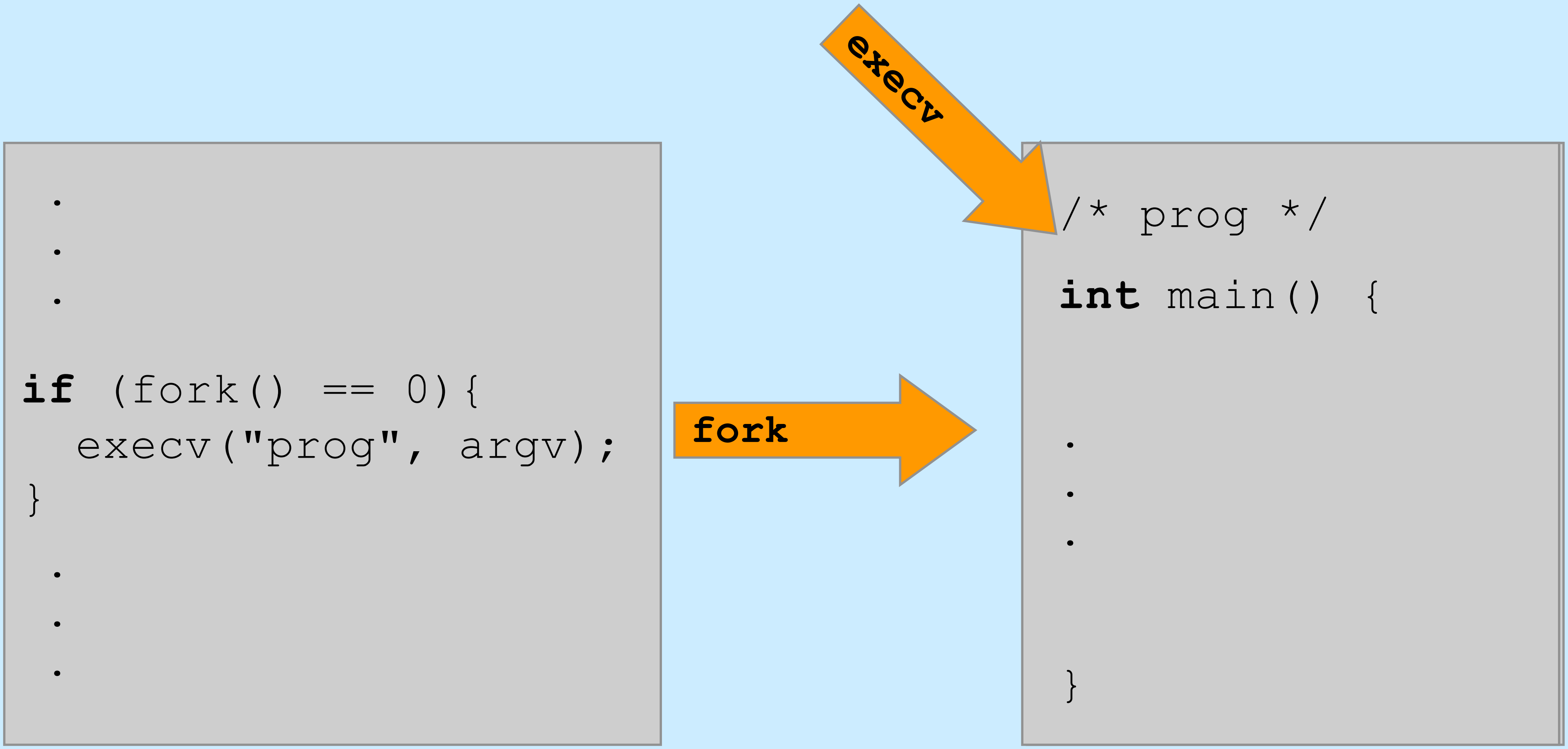
```
int main( ) {
  pid_t pid;
  pid_t ParentPid = getpid();

  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
```


Putting Programs into Processes



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);
}
```

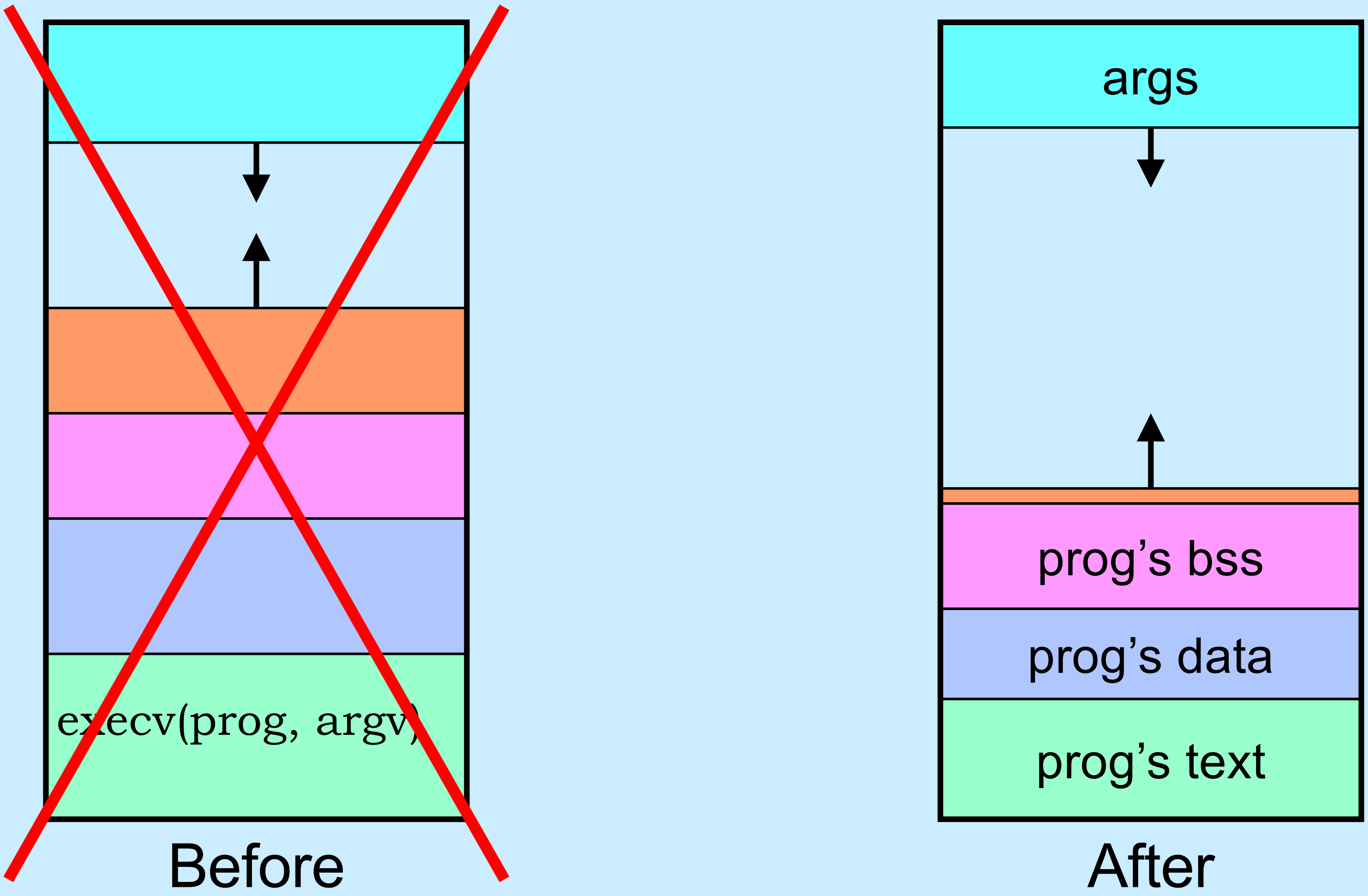
First "real" argument

End of list

Name of the file that contains the program

argv[0] is the name of the program

Loading a New Image



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