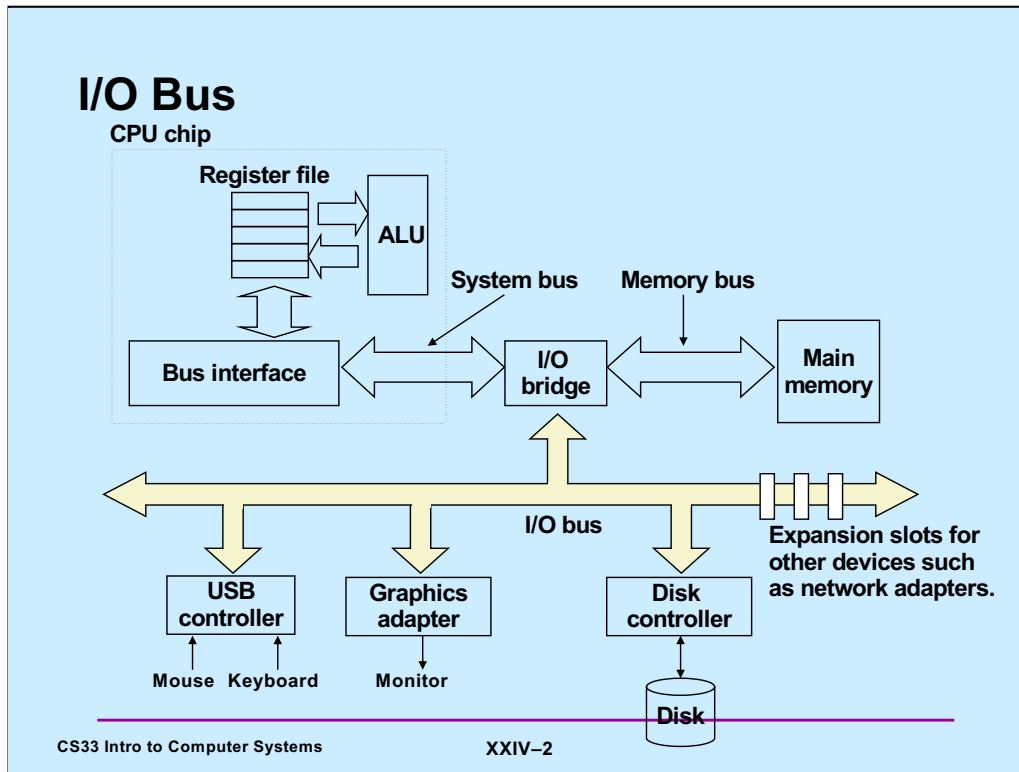


CS 33

Memory Hierarchy III

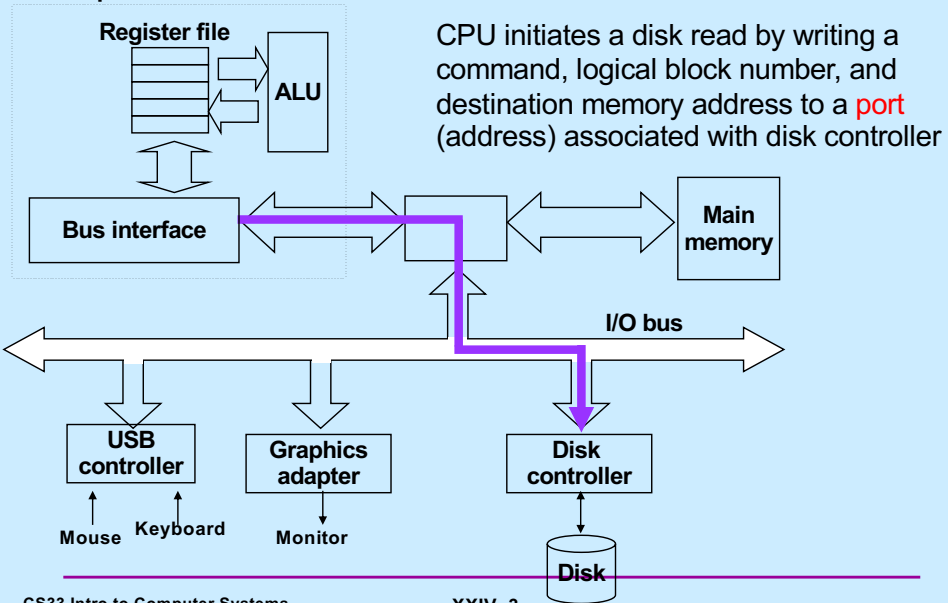
Most of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective,” 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in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



Supplied by CMU.

Reading a Disk Sector (1)

CPU chip



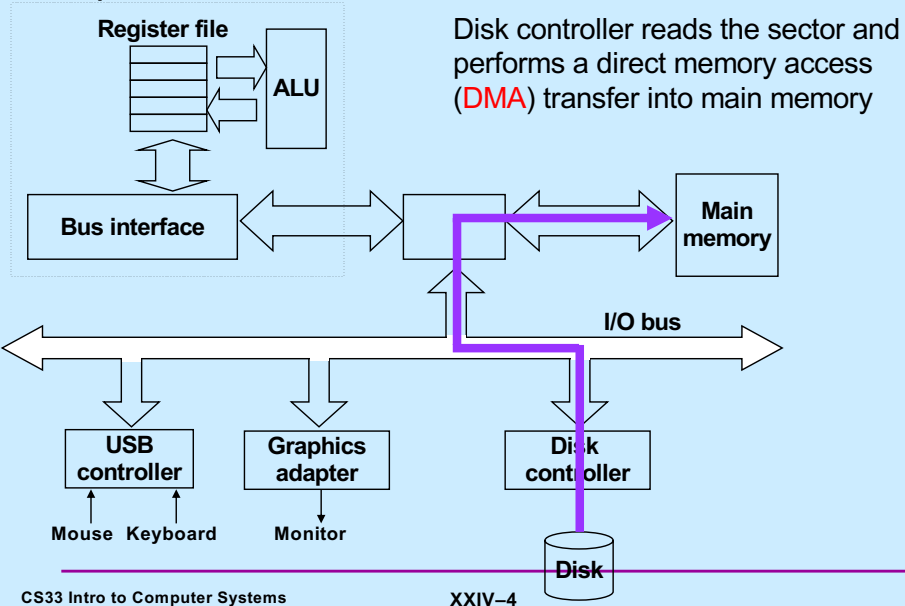
CS33 Intro to Computer Systems

XXIV-3

Supplied by CMU.

Reading a Disk Sector (2)

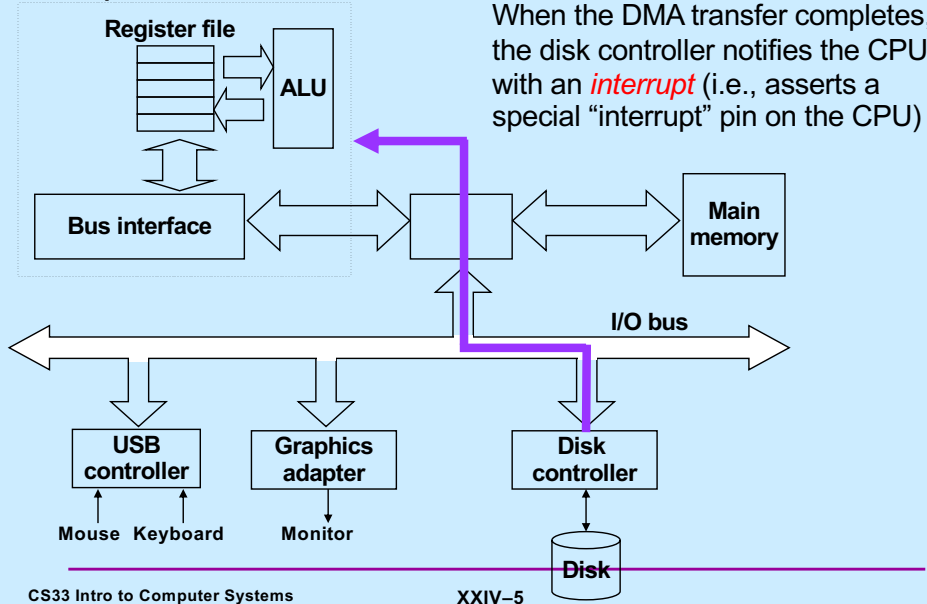
CPU chip



Supplied by CMU.

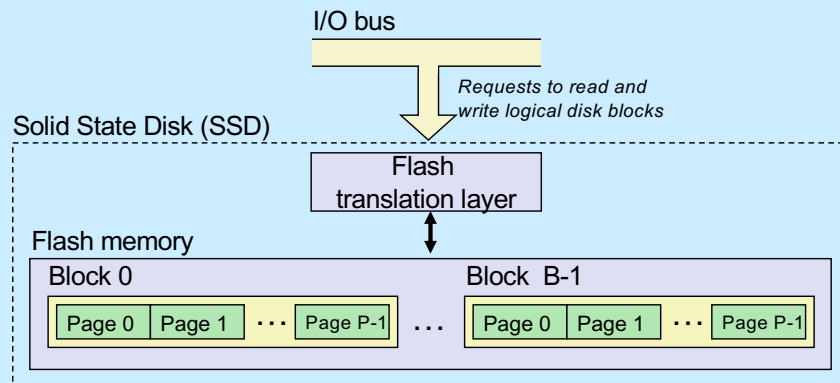
Reading a Disk Sector (3)

CPU chip



Supplied by CMU.

Solid-State Disks (SSDs)



- **Pages: 512KB to 4KB; blocks: 32 to 128 pages**
- **Data read/written in units of pages**
- **Page can be written only after its block has been erased**
- **A block wears out after 100,000 repeated writes**

Supplied by CMU.

SSD Performance Characteristics

Sequential read tput	250 MB/s	Sequential write tput	170 MB/s
Random read tput	140 MB/s	Random write tput	14 MB/s
Random read access	30 us	Random write access	300 us

- **Why are random writes so slow?**
 - erasing a block is slow (around 1 ms)
 - modifying a page triggers a copy of all useful pages in the block
 - » find a used block (new block) and erase it
 - » write the page into the new block
 - » copy other pages from old block to the new block

Supplied by CMU.

SSD Tradeoffs vs Rotating Disks

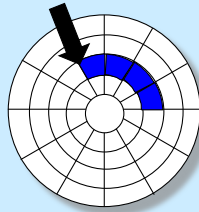
- **Advantages**
 - no moving parts → faster, less power, more rugged
- **Disadvantages**
 - have the potential to wear out
 - » mitigated by “wear-leveling logic” in flash translation layer
 - » e.g. Intel X25 guarantees 1 petabyte (10^{15} bytes) of random writes before they wear out
 - in 2010, about 100 times more expensive per byte
 - in 2017, about 6 times more expensive per byte
 - in 2023, about 1-1.5 times more expensive per byte
- **Applications**
 - smart phones, laptops, desktops

Adapted from a slide supplied by CMU.

SSDs are on their way to supplanting disks.

Reading a File on a Rotating Disk

- **Suppose the data of a file are stored on consecutive disk sectors on one track**
 - **this is the best possible scenario for reading data quickly**
 - » **single seek required**
 - » **single rotational delay**
 - » **all sectors read in a single scan**

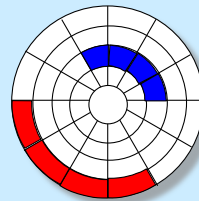


Quiz 1

We have two files on the same (rotating) disk. The first file's data resides in consecutive sectors on one track, the second in consecutive sectors on another track. It takes a total of t seconds to read all of the first file then all of the second file.

Now suppose the files are read concurrently, perhaps a sector of the first, then a sector of the second, then the first, then the second, etc. Compared to reading them sequentially, this will take

- a) less time
- b) about the same amount of time (within a factor of 2)
- c) much more time



Quiz 2

We have two files on the same solid-state disk. Each file's data resides in consecutive blocks. It takes a total of t seconds to read all of the first file then all of the second file.

Now suppose the files are read concurrently, perhaps a block of the first, then a block of the second, then the first, then the second, etc. Compared to reading them sequentially, this will take

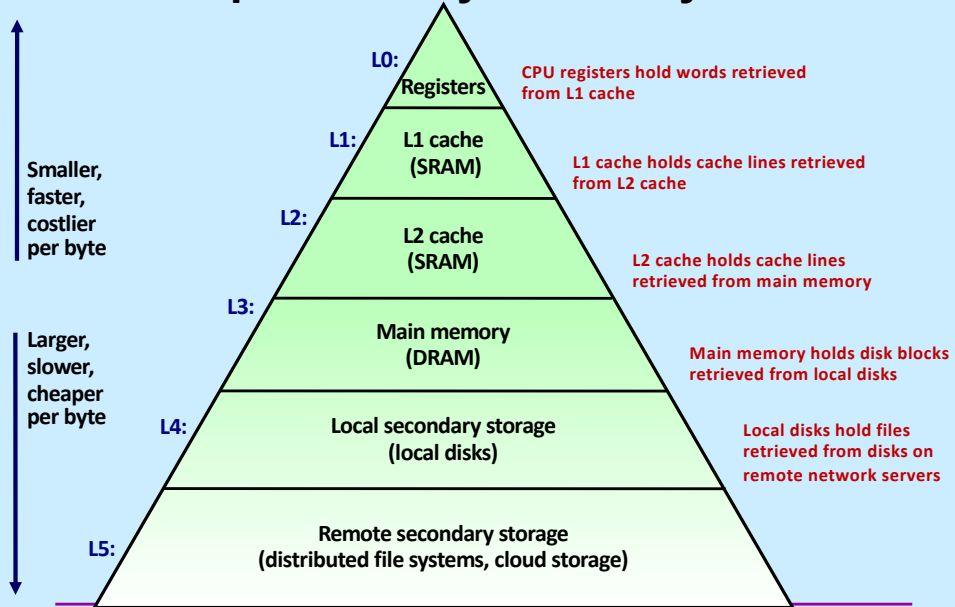
- a) less time
- b) about the same amount of time (within a factor of 2)
- c) much more time

Memory Hierarchies

- **Some fundamental and enduring properties of hardware and software:**
 - fast storage technologies cost more per byte, have less capacity, and require more power (heat!)
 - the gap between CPU and main memory speed is widening
 - well written programs tend to exhibit good locality
- **These fundamental properties complement each other beautifully**
- **They suggest an approach for organizing memory and storage systems known as a **memory hierarchy****

Supplied by CMU.

An Example Memory Hierarchy



Supplied by CMU.

Putting Things Into Perspective ...

- **Reading from:**
 - ... the L1 cache is like grabbing a piece of paper from your desk (3 seconds)
 - ... the L2 cache is picking up a book from a nearby shelf (14 seconds)
 - ... main system memory (DRAM) is taking a 4-minute walk down the hall to talk to a friend
 - ... a hard drive is like leaving the building to roam the earth for one year and three months

This analogy is from <http://duartes.org/gustavo/blog/post/what-your-computer-does-while-you-wait> (definitely worth reading!).

Disks Are Still Important

- **Cheap**
 - cost/byte less than SSDs
- **(fairly) Reliable**
 - data written to a disk is likely to be there next year
- **Sometimes fast**
 - data in consecutive sectors on a track can be read quickly
- **Sometimes slow**
 - data in randomly scattered sectors takes a long time to read

Abstraction to the Rescue

- Programs don't deal with sectors, tracks, and cylinders
- Programs deal with *files*
 - maze.c rather than an ordered collection of sectors
 - OS provides the implementation

Implementation Problems

- **Speed**
 - **use the hierarchy**
 - » **copy files into RAM, copy back when done**
 - **optimize layout**
 - » **put sectors of a file in consecutive locations**
 - **use parallelism**
 - » **spread file over multiple disks**
 - » **read multiple sectors at once**

Implementation Problems

- **Reliability**
 - **computer crashes**
 - » what you thought was safely written to the file never made it to the disk — it's still in RAM, which is lost
 - » worse yet, some parts made it back to disk, some didn't
 - you don't know which is which
 - on-disk data structures might be totally trashed
 - **disk crashes**
 - » you had backed it up ... yesterday
 - **you screw up**
 - » you accidentally delete the entire directory containing your shell 1 implementation

Implementation Problems

- **Reliability solutions**
 - **computer crashes**
 - » transaction-oriented file systems
 - » on-disk data structures always in well defined states
 - **disk crashes**
 - » files stored redundantly on multiple disks
 - **you screw up**
 - » file system automatically keeps "snapshots" of previous versions of files

CS 33

Linkers

gcc Steps

1) Compile

- to start here, supply .c file
- to stop here: `gcc -S` (produces .s file)
- if not stopping here, gcc compiles directly into a .o file, bypassing the assembler

2) Assemble

- to start here, supply .s file
- to stop here: `gcc -c` (produces .o file)

3) Link

- to start here, supply .o file

The Linker

- **An executable program is one that is ready to be loaded into memory**
- **The linker (known as ld: /usr/bin/ld) creates such executables from:**
 - object files produced by the compiler/assembler
 - collections of object files (known as libraries or archives)
 - and more we'll get to soon ...

The technology described here is current as of around 1990 and is known as static linking. We discuss static linking first, then move on to dynamic linking (in a few weeks), which is commonplace today.

Linker's Job

- **Piece together components of program**
 - arrange within address space
 - » code (and read-only data) goes into text region
 - » initialized data goes into data region
 - » uninitialized data goes into bss region
- **Modify address references, as necessary**

A Program

```
int nprimes = 100;
int *prime, *prime2;
int main() {
    int i, j, current = 1;
    prime = (int *)malloc(nprimes*sizeof(*prime));
    prime2 = (int *)malloc(nprimes*sizeof(*prime2));
    prime[0] = 2; prime2[0] = 2*2;
    for (i=1; i<nprimes; i++) {
        NewCandidate:
        current += 2;
        for (j=0; prime2[j] <= current; j++) {
            if (current % prime[j] == 0)
                goto NewCandidate;
        }
        prime[i] = current; prime2[i] = current*current;
    }
    return 0;
}
```

Annotations:

- data**: points to `int nprimes = 100;`
- bss**: points to `int *prime, *prime2;`
- dynamic**: points to the `malloc` calls.
- text**: points to the `main` function body.

The code is an implementation of the “sieve of Eratosthenes”, an early (~200 BCE) algorithm for enumerating prime numbers. The idea is to iterate through the positive integers. 2 is the first prime number. 3 is prime, since it’s not divisible by 2. 4 is not prime, since it is divisible by 2. 5 is not prime, since it’s not divisible by any of the primes discovered so far (5 is less than the largest’s square). This continues ad infinitum.

The **malloc** function allocates storage within the dynamic region. We discuss it in detail in an upcoming lecture.

... with Output

```
int nprimes = 100;
int *prime, *prime2;
int main() {
    ...
    printcol(5);
    return 0;
}

void printcol(int ncols) {
    int i, j;
    int nrows = (nprimes+ncols-1)/ncols;
    for (i = 0; i<nrows; i++) {
        for (j=0; (j<ncols) && (i+nrows*j < nvals); j++) {
            printf("%6d", prime[i + nrows*j]);
        }
        printf("\n");
    }
}
```

What this program actually does isn't all that important for our discussion. However, it prints out the vector of prime numbers in multiple columns.

... Compiled Separately

should refer to same thing

```
int nprimes = 100;
int *prime, *prime2;
int main() {
    ...
    printcol(5);
    return 0;
}
```

primes.c

ditto

```
extern int nprimes;
int *prime;
void printcol(int ncols) {
    int i, j;
    int nrows = (nprimes+ncols-1)/ncols;
    for (i = 0; i < nrows; i++) {
        for (j=0; (j < ncols)
            && (i+nrows*j < nvals); j++) {
            printf("%6d", prime[i + nrows*j]);
        }
        printf("\n");
    }
}
```

printcol.c

gcc -c primes.c

gcc -c printcol.c

gcc -o primes primes.o printcol.o

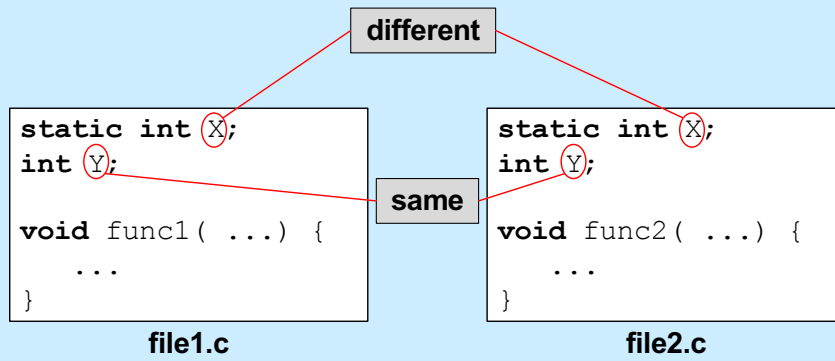
In the first two invocations of gcc, the “-c” flag tells it to compile the C code and produce an object (“.o”) file, but not to go any further (and thus not to produce an executable program). In the third invocation, gcc invokes the ld (linker) program to combine the two object files into an executable program. As we discuss soon, it will also bring in code (such as printf) from libraries.

Global Variables

- **Initialized vs. uninitialized**
 - initialized allocated in *data* section
 - uninitialized allocated in *bss* section
 - » implicitly initialized to zero
- **File scope vs. program scope**
 - *static* global variables known only within file that declares them
 - » two of same name in different files are different
 - » e.g., `static int X;`
 - non-static global variables potentially shared across all files
 - » two of same name in different files are same
 - » e.g., `int X;`

BSS is a mnemonic from an ancient assembler (not as ancient as Eratosthenes) and stands for “block started by symbol”, a rather meaningless phrase. The BSS section of the address space is where all uninitialized global and static local variables are placed. When the program starts up, this entire section is filled with zeroes.

Scope



Static Local Variables

```
int *sub1() {
    int var = 1;
    ...
    return &var;
    /* amazingly illegal */
}

int *sub2() {
    static int var = 1;
    ...
    return &var;
    /* (amazingly) legal */
}
```

Static local variables have the same scope as other local variables, but their values are retained across calls to the procedures they are declared in. Like global variables, uninitialized static local variables are stored in the BSS section of the address space (and implicitly initialized to zero), initialized static local variables are stored in the data section of the address space.

Reconciling Program Scope (1)

tentative definition

```
int X;  
  
void func1( ...) {  
    ...  
}
```

file1.c

(complete) definition

```
int X=1;  
  
void func2( ...) {  
    ...  
}
```

file2.c

**Where does X go?
What's its initial value?**

- tentative definitions overridden by compatible (complete) definitions
- if not overridden, then initial value is zero

X goes in the data section and has an initial value of 1. If file2.c did not exist, then X would go in the bss section and have an initial value of 0. Note that the textbook calls tentative definitions “weak definitions” and complete definitions “strong definitions”. This is non-standard terminology and conflicts with the standard use of the term “weak definition,” which we discuss shortly.

Reconciling Program Scope (2)

```
int X=2;  
  
void func1( ...) {  
    ...  
}
```

file1.c

```
int X=1;  
  
void func2( ...) {  
    ...  
}
```

file2.c

What happens here?

In this case we have conflicting definitions of X — this will be flagged (by the ld program) as an error.

Reconciling Program Scope (3)

```
int X=1;

void func1( ...) {
    ...
}
```

file1.c

```
int X=1;

void func2( ...) {
    ...
}
```

file2.c

Is this ok?

No; it is flagged as an error: only one file may supply an initial value.

Reconciling Program Scope (4)

```
extern int X;  
  
void func1( ...) {  
    ...  
}
```

file1.c

```
int X=1;  
  
void func2( ...) {  
    ...  
}
```

file2.c

What's the purpose of "extern"?

The “extern” means that this file will be using X, but it depends on some other file to provide a definition for it, either initialized or uninitialized. If no other file provides a definition, then ld flags an error.

If the “extern” were not there, i.e., if X were declared simply as an “int” in file1.c, then it wouldn't matter if no other file provided a definition for X — X would be allocated in bss with an implicit initial value of 0.

Note: this description of extern is how it is implemented by gcc. The official C99 standard doesn't require this behavior, but merely permits it. It also permits “extern” to be essentially superfluous: its presence may mean the same thing as its absence.

The C11 standard more-or-less agrees with the C99 standard. Moreover, it explicitly allows a declaration of the form “extern int X=1;” (i.e., initialization), which is not allowed by gcc.

For most practical purposes, whatever gcc says is the law ...

Does Location Matter?

```
int main(int argc, char *[]) {  
    return(argc);  
}
```

```
main:  
    pushq %rbp    ; push frame pointer  
    movq %rsp, %rbp    ; set frame pointer to point to new frame  
    movl %edi, %eax    ; put argc into return register (eax)  
    movq %rbp, %rsp    ; restore stack pointer  
    popq %rbp    ; pop stack into frame pointer  
    ret          ; return: pops end of stack into rip
```

This rather trivial program references memory via only `rsp` and `rip` (`rbp` is set from `rsp`). Its code contains no explicit references to memory, i.e., it contains no explicit addresses.

Location Matters ...

```
int X=6;
int *aX = &X;

int main() {
    void subr(int);
    int y=*aX;
    subr(y);
    return(0);
}

void subr(int i) {
    printf("i = %d\n", i);
}
```

We don't need to look at the assembler code to see what's different about this program: the machine code produced for it can't simply be copied to an arbitrary location in our computer's memory and executed. The location identified by the name **aX** should contain the address of the location containing **X**. But since the address of **X** will not be known until the program is copied into memory, neither the compiler nor the assembler can initialize **aX** correctly. Similarly, the addresses of **subr** and **printf** are not known until the program is copied into memory — again, neither the compiler nor the assembler would know what addresses to use.

Coping

- **Relocation**
 - **modify internal references according to where module is loaded in memory**
 - **modules needing relocation are said to be *relocatable***
 - » **which means they *require* relocation**
 - **the compiler/assembler provides instructions to the linker on how to do this**

A Revised Version of Our Program

```
extern int X;
int *aX = &X;
int Y = 1;

int main() {
    void subr(int);
    int y = *aX+Y;
    subr(y);
    return(0);
}
```

main.c

```
#include <stdio.h>
int X;

void subr(int XX) {
    printf("XX = %d\n", XX);
    printf("X = %d\n", X);
}
```

subr.c

```
gcc -o prog -O1 main.c subr.c
```

Note that what we did, in order to obtain what's in the next few slides, was:

```
gcc -S -O1 main.c subr.c
```

```
gcc -c main.s subr.s
```

```
gcc -o prog main.o subr.o
```

main.s (1)

```
0:      .file   "main.c"
0:      .text
0:      .globl main
0:      .type   main, @function
0: main:
0: .LFB0:
0:      .cfi_startproc
0:      subq   $8, %rsp
4:      .cfi_def_cfa_offset 16
4:      movq   aX(%rip), %rax
11:     movl   (%rax), %edi
13:     addl   Y(%rip), %edi
19:     call  subr
24:     movl   $0, %eax
29:     addq   $8, %rsp
33:     .cfi_def_cfa_offset 8
33:     ret
34:     .cfi_endproc
34: .LFE0:
34:     .size   main, .-main
```

must be replaced with aX's address, expressed as an offset from the next instruction

must be replaced with Y's address, expressed as an offset from the next instruction

must be replaced with subr's address, expressed as an offset from the next instruction

Note that a symbol's value is the location of what it refers to. The compiler/assembler knows what the values (i.e., locations) of **aX** and **Y** are relative to the beginning of this module's data section (next slide), but has no idea what **subr**'s value is. It is the linker's job to provide final values for these symbols, which will be the addresses of the corresponding C constructs when the program is loaded into memory. The linker will adjust these values to obtain the locations of what they refer to relative to the value of register `rip` when the referencing instructions are executed.

One might ask why these locations are referred to using offsets from the instruction pointer (also known as the program counter), rather than simply using their addresses. The reason is to save space: the addresses would be 64 bits long, but the offsets are only 32 bits long.

The `.file` directive supplies information to be placed in the object file and the executable of use to debuggers — it tells them what the source-code file is.

The `.globl` directive indicates that the symbol, defined here, will be used by other modules, and thus should be made known to the linker.

The `.type` directive indicates how the symbol is used. Two possibilities are function and object (meaning a data object).

The `.size` directive indicates the size that should be associated with the given symbol.

The directives starting with `.cfi_` are there for the sake of the debugger. They generate auxiliary information stored in the object file (but not executed) that describes the relation between the stack pointer (`%rsp`) and the beginning of the stack frame. Thus

they compensate for the lack of a standard frame-pointer register (%esp for IA32). In particular, they emit data going into a table that is used by a debugger (such as gdb) to determine, based on the value of the instruction pointer (%rip) and the stack pointer, where the beginning of the current stack frame is.

main.s (2)

```
0:      .globl Y
0:      .data
0:      .align 4
0:      .type Y, @object
0:      .size Y, 4
0: Y:
0:      .long 1
4:      .globl aX
8:      .align 8
8:      .type aX, @object
8:      .size aX, 8
8: aX:
8:      .quad X
8:      .ident "GCC: (Debian 4.7.2-5) 4.7.2"
0:      .section .note.GNU-stack,"",@progbits
```

Annotations:

- Y should be made known to others
- aX should be made known to others
- must be replaced with address of X

The symbol **X**'s value is, at this point, unknown.

The “.data” directive indicates that what follows goes in the data section.

The “.long” directive indicates that storage should be allocated for a long word.

The “.quad” directive indicates that storage should be allocated for a quad word.

The “.align” directive indicates that the storage associated with the symbol should be aligned, in the cases here, on 4-byte and 8-byte boundaries (i.e., the least-significant two bits and three bits of their addresses should be zeroes).

The “.ident” directive indicates the software used to produce the file and its version.

The “.section” directive used here is supplied by gcc by default and indicates that the program should have a non-executable stack (this is important for security purposes).

subr.s (1)

```
        .file   "subr.c"
0:      .section .rodata.str1.1,"aMS",@progbits,1
0: .LC0:      .string "XX = %d\n"
9: .LC1:      .string "X = %d\n"
```

The “.section” directive here indicates that what follows should be placed in read-only storage (and will be included in the text section). Furthermore, what follows are strings with a one-byte-per-character encoding that require one-byte (i.e., unrestricted) alignment. This information will ultimately be used by the linker to reduce storage by identifying strings that are suffixes of others.

subr.s (2)

```
0:      .text
0:      .globl subr
0:      .type subr, @function
0: subr:
0:      .LFB11:
0:      .cfi_startproc
0:      subq $8, %rsp
4:      .cfi_def_cfa_offset 16
4:      movl %edi, %esi
6:      movl $.LC0, %edi
11:     movl $0, %eax
16:     call printf
21:     movl X(%rip), %esi
27:     movl $.LC1, %edi
32:     movl $0, %eax
37:     call printf
42:     addq $8, %rsp
46:     .cfi_def_cfa_offset 8
46:     ret
47:     .cfi_endproc
47: .LFE11:
47:     .size subr, .-subr
```

subr should be made known to others

must be replaced with .LC0's address

must be replaced with .LC1's address

must be replaced with printf's address, expressed as an offset from the next instruction

Note that the compiler has generated **movl** instructions (copying 32 bits) for copying the addresses of `.LC0` and `.LC1`: it's assuming that both addresses will fit in 32 bits (in other words, that the text section of the program will be less than 2^{32} bytes long — probably a reasonable assumption).

subr.s (3)

```
0:      .comm X,4,4
0:      .ident "GCC: (Debian 4.7.2-5) 4.7.2"
0:      .section .note.GNU-stack,"",@progbits
```

reserve 4 bytes of 4-byte aligned storage for X

The “.comm” directive indicates here that four bytes of four-byte aligned storage are required for X in BSS. “comm” stands for “common”, which is what the Fortran language uses to mean the same thing as BSS. Since Fortran predates pretty much everything (except for Eratosthenes), its terminology wins (at least here).

Quiz 3

```
int X;  
int func(int arg) {  
    static int Y;  
    int Z;  
  
    ...  
}
```

Which of *X*, *Y*, *Z*, and *arg* would the compiler know the addresses of at compile time?

- a) all
- b) just *X* and *Y*
- c) just *arg* and *Z*
- d) none