Computer Systems

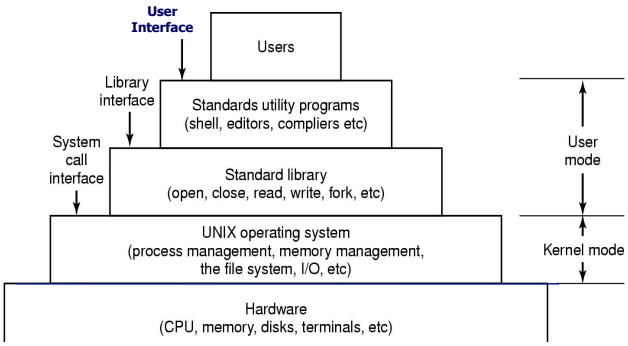
MTSU CSCI 3240

Spring 2016

Dr. Hyrum D. Carroll

Materials from CMU and Dr. Butler

A software view



How it works

hello.c program

```
#include <stdio.h>
int main()
{
   printf("hello, world\n");
}
```

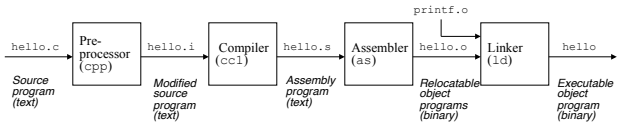
The Compilation system

gcc is the *compiler driver*

gcc invokes several other *compilation phases*

1) Preprocessor
2) Compiler
3) Assembler
4) Linker

What does each one do? What are their outputs?



1. Preprocessor (cpp)

First, gcc compiler driver invokes cpp to generate expanded C source

- cpp just does text substitution
- Converts the C source file to another C source file
- Expands #defines, #includes, etc.
- Output is another C source

1. Preprocesser

Included files:

```
#include <foo.h>
#include "bar.h"
```

Defined constants:

```
#define MAXVAL   40000000
```

By convention, all capitals tells us it's a constant, not a variable.

Macros:

```
#define MIN(x,y)    ((x)<(y) ? (x):(y))
#define RIDX(i, j, n)   ((i) * (n) + (j))
```

1. Preprocesser

Conditional compilation:

```
#ifdef …   or  #if defined( … )
#endif
```

- Code you think you may need again (e.g. debug print statements)
  - Include or exclude code based on #define/#ifdef
  - More readable than commenting code out
- Portability
  - Compilers have "built in" constants defined
  - Operating system specific code
    - » #if defined(__i386__) || defined(WIN32) || …
  - Compiler-specific code
    - » #if defined(__INTEL_COMPILER)
  - Processor-specific code
    - » #if defined(__SSE__)

2. Compiler (cc1)

Next, gcc compiler driver invokes cc1 to generate assembly code

- Translates high-level C code into assembly
  - Variable abstraction mapped to memory locations and registers
  - Logical and arithmetic functions mapped to underlying machine opcodes

3. Assembler (as)

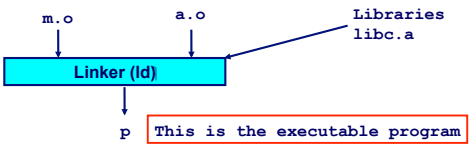
Next, gcc compiler driver invokes as to generate object code

- Translates assembly code into binary object code that can be directly executed by CPU

4. Linker (ld)

Finally, gcc compiler driver calls linker (ld) to generate executable

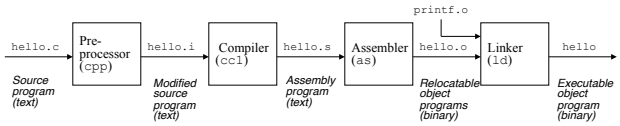
- Links together object code and static libraries to form final executable



Summary of compilation process

Compiler driver **(cc or gcc) coordinates all steps**

- Invokes preprocessor (`cpp`), compiler (`cc1`), assembler (`as`), and linker (`ld`).
- Passes command line arguments to appropriate phases



The run-time system

Program runs on top of an operating system that implements

- File system
- Memory management
- Processes
- Device management
- Network support
- etc.

Operating system functions

Protection

- Protects the hardware/itself from user programs
- Protects user programs from each other
- Protects files from unauthorized access

Resource allocation

- Memory, I/O devices, CPU time, space on disks

Operating system functions

Abstract view of resources

- Files as an abstraction of storage devices
- System calls an abstraction for OS services
- Virtual memory a uniform memory space abstraction for each process
  - Gives the illusion that each process has entire memory space
- A process (in conjunction with the OS) provides an abstraction for a virtual computer
  - Slices of CPU time to run in

Unix file system

Key concepts

- Everything is a file
  - Keyboards, mice, CD-ROMS, disks, modems, networks, pipes, sockets
  - One abstraction for accessing most external things
- A file is a stream of bytes with no other structure.
  - on the hard disk or from an I/O device
  - Higher levels of structure are an application concept, not an operating system concept
    - » No “records” (contrast with Windows/VMS)

Unix file systems

Managed by OS on disk

- Dynamically allocates space for files
- Implements a name space so we can find files
- Hides where the file lives and its physical layout on disk
- Provides an illusion of sequential storage

All we have to know to find a file is its name

Process abstraction

A fundamental concept of operating systems.

A process is an instance of a program when it is running.

- A program is a file on the disk containing instructions to execute
- A process is an instance of that program loaded in memory and running
  - Like you baking the cookies, following the instructions

A process includes

- Code and data in memory, CPU state, open files, thread of execution

How does a program get executed?

The operating system creates a process.

- Including among other things, a virtual memory space

System loader reads program from file system and loads its code into memory

- Program includes any statically linked libraries
- Done via DMA (direct memory access)

System loader loads dynamic shared objects/libraries into memory

Then it starts the thread of execution running

- Note: the program binary in file system remains and can be executed again

Where are programs loaded in memory?

To start with, imagine a primitive operating system.

- Single tasking.
- Physical memory addresses go from zero to N.

The problem of loading is simple

- Load the program starting at address zero
- Use as much memory as it takes.
- Linker binds the program to absolute addresses
- Code starts at zero
- Data concatenated after that
- etc.

Where are programs loaded, cont'd

Next imagine a multi-tasking operating system on a primitive computer.

- Physical memory space, from zero to N.
- Applications share space
- Memory allocated at load time in unused space
- Linker does not know where the program will be loaded
- Binds together all the modules, but keeps them relocatable

How does the operating system load this program?

- Not a pretty solution, must find contiguous unused blocks

How does the operating system provide protection?

- Not pretty either

Where are programs loaded, cont'd

Next, imagine a multi-tasking operating system on a modern computer, with hardware-assisted virtual memory

The OS creates a virtual memory space for each user's program.

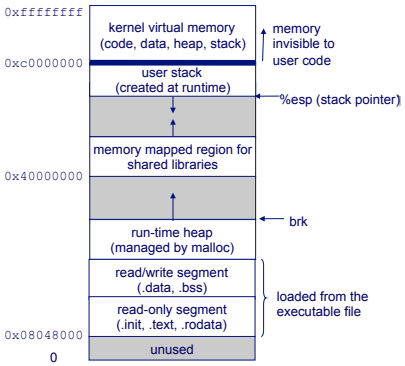
- As though there is a single user with the whole memory all to itself.

Now we're back to the simple model

- The linker statically binds the program to virtual addresses
- At load time, the operating system allocates memory, creates a virtual address space, and loads the code and data.
- Binaries are simply virtual memory snapshots of programs

Example memory map

Nothing is left relocatable, no relocation at load time



The memory hierarchy

Operating system and CPU memory management unit gives each process the "illusion" of a uniform, dedicated memory space

- i.e. 0x0 – 0xFFFFFFFF for IA32
- Allows multitasking
- Hides underlying non-uniform memory hierarchy

Memory heirarchy motivation

In 1980

- CPUs ran at around 1 mhz.
- A memory access took about as long as a CPU instruction
- Memory was not a bottleneck to performance

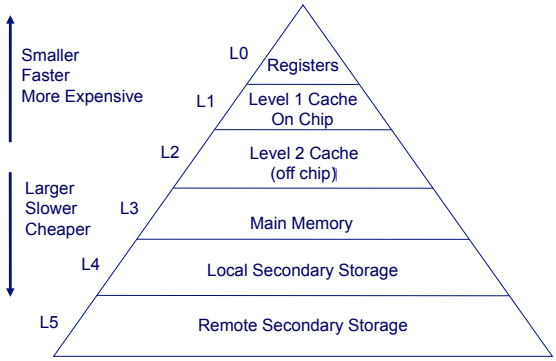
Today

- CPUs are about 3000 times faster than in 1980
- DRAM Memory is about 10 times faster than in 1980

We need a small amount of faster, more expensive memory for stuff we'll need in the near future

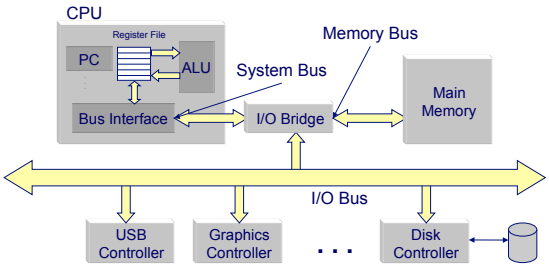
- How do you know what you'll need in the future?
- Locality
- L1, L2, L3 caches

The memory heirarchy

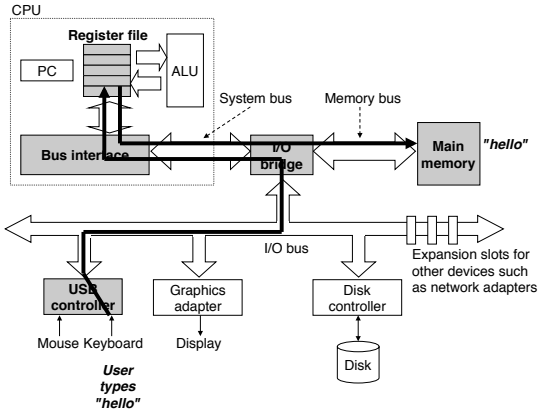


Hardware organization

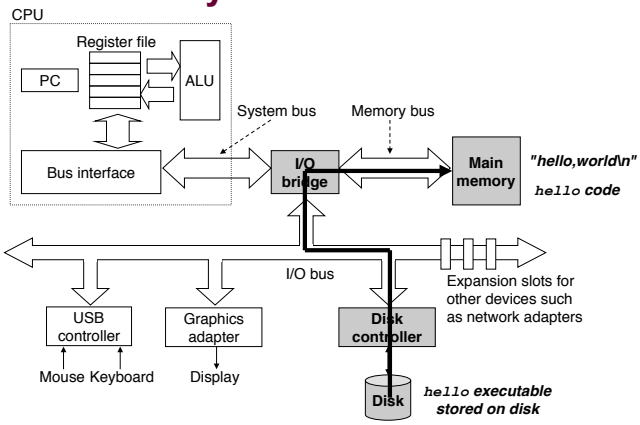
The last piece…how does it all run on hardware?



Reading the hello command from the keyboard



Loading the executable from disk into main memory



Writing the output string from memory to the display