# Instructor

<table>
<thead>
<tr>
<th>Who</th>
<th>Where</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hubertus Franke</td>
<td>Office hours:</td>
</tr>
<tr>
<td></td>
<td>Mon 6:00 - 7:00 pm</td>
</tr>
<tr>
<td>Ph.D. EE Vanderbilt</td>
<td>Room:</td>
</tr>
<tr>
<td>University 1992</td>
<td>WWH 320 (Z)</td>
</tr>
<tr>
<td>Research Staff Member</td>
<td>WWH 328</td>
</tr>
<tr>
<td>and Manager Operating</td>
<td></td>
</tr>
<tr>
<td>Systems at IBM T.J.</td>
<td></td>
</tr>
<tr>
<td>Watson Research Center</td>
<td></td>
</tr>
<tr>
<td>in Yorktown Heights, NY</td>
<td></td>
</tr>
<tr>
<td>(since 1993)</td>
<td></td>
</tr>
</tbody>
</table>

## General Interests

| Cloud Infrastructures   | High Performance Computing:                |
|                         | MPI (Message Passing Interfaces),          |
|                         | Gang Scheduling                            |
| Operating Systems:      | Software Engineering, Compilers and        |
|                         | Robotics.                                  |
| Linux, AIX, object      | ~100 publications in these areas           |
| oriented OS (K42)       | ~30 patents                                |
| Scheduling, memory      |                                            |
| management, ..          |                                            |
| Computer Architecture:  |                                            |
| Multicore processors    |                                            |
| and Systems on a chip   |                                            |
# TA’s

<table>
<thead>
<tr>
<th>A-M</th>
<th>N-Z</th>
</tr>
</thead>
</table>
| Kanwarpreet Randhawa  
ksr287@nyu.edu  
Office hours: (first 2-3 weeks)  
Tue 2-3  
Room:  
Weaver 13th Floor | Bohou Li  
bohou.li@nyu.edu  
Office hours:  
Th 5-6  
Room:  
Weaver 13th Floor |
Formal Goals of This Course

• What exactly is an operating systems?
• How does the OS interact with the hardware and other software applications?
• Main concepts of an OS
• OS in many contexts
Informal Goals of This Course

• To get more than an A
• To learn OS and enjoy it
• To use what you have learned in MANY different contexts
• To be able to develop your own OS if you want to
• To start your research project in OS
The Course Web Page

http://cs.nyu.edu/courses/fall13/CSCI-GA.2250-001/index.html

• Lecture slides
• Info about mailing list, labs, ...
• Useful links (manuals, tools, ... )
The Textbook

Author: Andrew Tannenbaum
Title: Modern Operating Systems 3e

Publisher: Prentice Hall
Grading

• Homework : 00%
• Lab : 55%
• Midterm : 15%
• Final : 30%

• Due at the beginning of the lecture
• In hardcopy
• Will be “graded” and returned to you
• No late submissions accepted
Grading

- Homework: 00%
- Lab: 55%
- Midterm: 15%
- Final: 30%

- Usually due few weeks after assignment
- Submitted as softcopy
- 2 point penalty per day late
Grading

- Homework: 00%
- Lab: 55%
- Midterm: 15%
- Final: 30%

• Cumulative
• No electronic equipment
Integrity

• Academic integrity
  • http://www.nyu.edu/about/policies-guidelines-compliance/policies-and-guidelines/academic-integrity-for-students-at-nyu.html

• Your labs, and exams must be your own - we have a zero tolerance policy towards cheating of any kind and any student who cheats will get a failing grade in the course.

• Both the cheater and the student who aided the cheater will be held responsible for the cheating
Does a programmer need to understand all this hardware in order to write these software programs?
Operating System

- Media Player
- emails
- Games
- Word Processing
The Two Main Tasks of OS

• Provide programmers (and programs) a clean abstract set of resources

• Manage the hardware resources
A Glimpse on Hardware
A Glimpse on Hardware
Basic Elements

Processor(s)

Main Memory

I/O Modules

System Bus
What Is an OS?

Resources
- Allocation
- Protection
- Reclamation
- Virtualization

Services
- Abstraction
- Simplification
- Convenience
- Standardization

CONTAINER

Makes computer usage simpler
What Is an OS?

Resources
- **Allocation**
- Protection
- Reclamation
- Virtualization

Finite resources
Competing demands

Examples:
- CPU
- Memory
- Disk
- Network

Government
- Limited budget,
  - Land,
  - Oil,
  - Gas,
What Is an OS?

Resources
- Allocation
- Protection
- Reclamation
- Virtualization

- You can’t hurt me
  I can’t hurt you
- Implies some degree of safety & security
What Is an OS?

Resources
- Allocation
- Protection
- Reclamation
- Virtualization

The OS gives
- Voluntary at run time
- Implied at termination
- Involuntary
- Cooperative

The OS takes away

Government
- Income Tax
What Is an OS?

Resources
- Allocation
- Protection
- Reclamation
- Virtualization

- illusion of infinite, private resources
- Memory versus disk
- Timeshared CPU
- More extreme cases possible (& exist)

Government
Social security
Booting Sequence

• BIOS starts
  – checks how much RAM
  – keyboard
  – other basic devices

• BIOS determines boot Device

• The first sector in boot device is read into memory and executed to determine active partition

• Secondary boot loader is loaded from that partition.

• This loaders loads the OS from the active partition and starts it.
• Mainframe/supercomputer OS
  • batch
  • transaction processing
  • timesharing
  • e.g. OS/390
• Server OS
• Multiprocessor OS
• PC OS
• Embedded OS
• Sensor node OS
• RTOS
• Smart card OS
OS

Types
- Mainframe OS/supercomputer
  - batch
  - transaction processing
  - timesharing
  - e.g. OS/390
- Server OS
- Multiprocessor OS
- PC OS
- Embedded OS
- Sensor node OS
- RTOS
- Smart card OS

Concepts
- Processes
  - Its address space
  - Its resources
  - Process table
- Address space
- File system
- I/O
- Protection

Different Structures
OS

Types

- Mainframe OS/supercomputer
  - batch
  - transaction processing
  - timesharing
  - e.g. OS/390

- Server OS
- Multiprocessor OS
- PC OS
- Embedded OS
- Sensor node OS
- RTOS
- Smart card OS

Concepts

- Processes
  - Its address space
  - Its resources
  - Process table

- Address space
- File system
- I/O
- Protection

Different Structures

- Monolithic
- Layered systems
- Microkernels
- Client-server
- Virtual machines
Types
- Mainframe OS
  - batch
  - transaction processing
  - timesharing
    - e.g. OS/390
- Server OS
- Multiprocessor OS
- PC OS
- Embedded OS
- Sensor node OS
- RTOS
- Smart card OS

Main objectives of an OS:
- Convenience
- Efficiency
- Ability to evolve

Different Structures
- Monolithic
- Layered systems
- Microkernels
- Client-server
- Virtual machines

Concepts
- Processes
  - Its address space
  - Its resources
  - Process table
- Address space
- File system
- I/O
- Protection
OS

Processes

USER

Hardware
OS Services

- Program development
- Program execution
- Access I/O devices
- Controlled access to files
- System access
- Error detection and response
- Accounting
In a nutshell

• OS is really a manager:
  – programs, applications, and processes are the customers
  – The hardware provide the resources
• OS works in different environments and under different restrictions (supercomputers, workstations, notebooks, tablets, smartphones, real-time, …)
History of Operating Systems

• "We can chart our future clearly and wisely only when we know the path which has led to the present."
  – dlai E. Stevenson, Lawyer and Politician

• First generation 1945 - 1955
  – vacuum tubes, plug boards (no OS)

• Second generation 1955 - 1965
  – transistors, batch systems

• Third generation 1965 - 1980
  – ICs and multiprogramming

• Fourth generation 1980 - present
  – server computers
  – personal computers, hand-held devices, sensors
History of Operating Systems (1945-55)

• Programming and Controlled tied to the Computer

<table>
<thead>
<tr>
<th>Name</th>
<th>First operational</th>
<th>Numeral system</th>
<th>Computing mechanism</th>
<th>Programming</th>
<th>Turing complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zuse Z3 (Germany)</td>
<td>May 1941</td>
<td>Binary floating point</td>
<td>Electro-mechanical</td>
<td>Program-controlled by punched 35 mm film stock (but no conditional branch)</td>
<td>Yes (1998)</td>
</tr>
<tr>
<td>Atanasoff–Berry Computer (US)</td>
<td>1942</td>
<td>Binary</td>
<td>Electronic</td>
<td>Not programmable—single purpose</td>
<td>No</td>
</tr>
<tr>
<td>Colossus Mark 1 (UK)</td>
<td>February 1944</td>
<td>Binary</td>
<td>Electronic</td>
<td>Program-controlled by patch cables and switches</td>
<td>No</td>
</tr>
<tr>
<td>Harvard Mark I – IBM ASCC (US)</td>
<td>May 1944</td>
<td>Decimal</td>
<td>Electro-mechanical</td>
<td>Program-controlled by 24-channel punched paper tape (but no conditional branch)</td>
<td>No</td>
</tr>
<tr>
<td>Colossus Mark 2 (UK)</td>
<td>June 1944</td>
<td>Binary</td>
<td>Electronic</td>
<td>Program-controlled by patch cables and switches</td>
<td>No</td>
</tr>
<tr>
<td>Zuse Z4 (Germany)</td>
<td>March 1945</td>
<td>Binary floating point</td>
<td>Electro-mechanical</td>
<td>Program-controlled by punched 35 mm film stock</td>
<td>Yes</td>
</tr>
<tr>
<td>ENIAC (US)</td>
<td>July 1946</td>
<td>Decimal</td>
<td>Electronic</td>
<td>Program-controlled by patch cables and switches</td>
<td>Yes</td>
</tr>
<tr>
<td>Manchester Small-Scale Experimental Machine (Baby) (UK)</td>
<td>June 1948</td>
<td>Binary</td>
<td>Electronic</td>
<td>Stored-program in Williams cathode ray tube memory</td>
<td>Yes</td>
</tr>
<tr>
<td>Modified ENIAC (US)</td>
<td>September 1948</td>
<td>Decimal</td>
<td>Electronic</td>
<td>Read-only stored programming mechanism using the Function Tables as program ROM</td>
<td>Yes</td>
</tr>
<tr>
<td>EDSAC (UK)</td>
<td>May 1949</td>
<td>Binary</td>
<td>Electronic</td>
<td>Stored-program in mercury delay line memory</td>
<td>Yes</td>
</tr>
<tr>
<td>Manchester Mark 1 (UK)</td>
<td>October 1949</td>
<td>Binary</td>
<td>Electronic</td>
<td>Stored-program in Williams cathode ray tube memory and magnetic drum memory</td>
<td>Yes</td>
</tr>
<tr>
<td>CSIRAC (Australia)</td>
<td>November 1949</td>
<td>Binary</td>
<td>Electronic</td>
<td>Stored-program in mercury delay line memory</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Source: wikipedia
History of Operating Systems
(1945-1955)

• Vacuum tubes, plug boards (no OS)
  – ENIAC (UPenn 1944)
    • 30 tons, 150m, 5000 calcs/sec
  – Zuse’s Z3 (1941)
    • 2000 relays
    • 22 bit words
    • 5-10 Hz

• What’s a bug?
History of Operating Systems (1955-65)

- Emergence of the Mainframe
- Programmers isolated from machine
- Programming Languages emerge
  - Fortran
  - Cobol

- Structure of a typical JCL job – 2nd generation
- Single user
- Programmer/User as the operator
- Secure, but inefficient use of expensive resources
- Low CPU utilization-slow mechanical I/O devices
History of Operating System
(1955-65)

Early batch system
- bring cards to 1401
- read cards to tape
- put tape on 7094 which does computing
- put tape on 1401 which prints output.
History of Operating Systems (1965-80)

- **Multiprogramming systems**
  - Multiple jobs in memory - 3rd generation
  - Allow overlap of CPU and I/O activity
  - Polling/Interrupts, Timesharing
  - Spooling

- **Different types**
  - Epitomized by the IBM 360 machine
  - MFT (IBM OS/MFT) Fixed Number of Tasks
  - MVT (IBM OS/MVT) Variable Number of Tasks

- **Birth of Modern Operating System Concepts**
  - Time Sharing: when and what to run → scheduling
  - Resource Control: memory management, protection
The Operating System Jungle / Zoo (1980-present)

- Mainframe operating systems
- Server operating systems
- Multiprocessor operating systems
- Personal computer operating systems
- Real-time operating systems
- Embedded operating systems
- Smart card operating systems
- Cellphone/tablet operating systems
- Sensor operating systems
Computer Architecture
(a closer look)

We must know and understand what is actually managed by an OS
Processors

• Each CPU has a specific set of instructions
  – ISA (Instruction Set Architecture)
    • RISC: Sparc, MIPS, PowerPC
    • CISC: x86, zSeries

• All CPUs contain
  – General registers inside to hold key variables and temporary results
  – Special registers visible to the programmer
    • Program counter contains the memory address of the next instruction to be fetched
    • Stack pointer points to the top of the current stack in memory
    • PSW (Program Status Word) contains the condition code bits which are set by comparison instructions, the CPU priority, the mode (user or kernel) and various other control bits.
How Processors Work

• Execute instructions
  – CPU cycles
    • Fetch (from mem) → decode → execute
    • Program counter (PC)
      – When is PC changed?
    • Pipeline: fetch \( n+2 \) while decode \( n+1 \) while execute \( n \)
  – Two modes of CPU (why?)
    • User mode (a subset of instructions)
    • Privileged mode (all instruction)
  – Trap (special instruction)
Memory-Storage Hierarchy

Typical access time

<table>
<thead>
<tr>
<th>Access Time</th>
<th>Capacity</th>
<th>Typical capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 nsec</td>
<td>Registers</td>
<td>&lt;1 KB</td>
</tr>
<tr>
<td>2 nsec</td>
<td>Cache</td>
<td>1 MB</td>
</tr>
<tr>
<td>10 nsec</td>
<td>Main memory</td>
<td>64-512 MB</td>
</tr>
<tr>
<td>10 msec</td>
<td>Magnetic disk</td>
<td>5-50 GB</td>
</tr>
<tr>
<td>100 sec</td>
<td>Magnetic tape</td>
<td>20-100 GB</td>
</tr>
</tbody>
</table>

• Other metrics:
  – Bandwidth (e.g. MemBandwidth 30GB/s → 200GB/s, Disk ~70-200MB/s)
• What can an OS do to increase its “performance”
  – Active Cache management...
Memory Access

• Memory read:
  – Assert address on address lines
  – Wait till data appear on data line
  – Much slower than CPU!

• How many mem access for one instruction?
Memory Access

• Memory read:
  – Assert address on address lines
  – Wait till data appear on data line
  – Much slower than CPU!

• How many mem access for one instruction?
  – Fetch instruction
  – Fetch operand (0, 1 or 2)
  – Write results (0 or 1)

• How to speed up instruction execution?
CPU Caches

- **Principle:**
  - Data/Instruction that were recently used are “likely” used again in short period
  - Used in “many” subsystems (I/O, filesystems, ...)

- **Cache hit:**
  - no need to access memory

- **Cache miss:**
  - data obtained from mem, possibly update cache

- **Issues**
  - Operation MUST be correct
  - Cache management for Memory done in hardware
Example of Device
(resource and operation)

• Disk:
  – Multiple-subdevice
  – Translations
    • Block -> sector
  – Head Movement
  – Seek Time
  – Data Placement
  – Power Management

OS Major Components

• Process and thread management
• Resource management
  – CPU
  – Memory
  – Device
• File system
• Bootstrapping
Process: a running program

- A process includes
  - Address space
  - Process table entries (state, registers)
    - Open files, thread(s) state, resources held

- A process tree
  - A created two child processes, B and C
  - B created three child processes, D, E, and F
Some System Calls For Process Management

<table>
<thead>
<tr>
<th>Process management</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid = fork()</td>
<td>Create a child process identical to the parent</td>
</tr>
<tr>
<td>pid = waitpid(pid, &amp;statloc, options)</td>
<td>Wait for a child to terminate</td>
</tr>
<tr>
<td>s = execve(name, argv, environp)</td>
<td>Replace a process’ core image</td>
</tr>
<tr>
<td>exit(status)</td>
<td>Terminate process execution and return status</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signal</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Call</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>kill(pid, signal)</td>
<td>Deliver signal to the process pid</td>
</tr>
<tr>
<td>signal( signal, function )</td>
<td>Define which function to call for signal</td>
</tr>
</tbody>
</table>
Address Space

- Defines where sections of data and code are located in 32 or 64 address space
- Defines protection of such sections
- ReadOnly, ReadWrite, Execute
- Confined “private” addressing concept
  - requires form of address virtualization
Recap: What is an OS?

• **Code that:**
  – Sits between programs & hardware
  – Sits between different programs
  – Sits betweens different users

• **Job of OS:**
  – **Manage hardware resources**
    • Allocation, protection, reclamation, virtualization
  – **Provide services to app.**
    • Abstraction, simplification, standardization
Recap: What is an OS?

- **Code that:**
  - Sits between programs & hardware
  - Sits between different programs
  - Sits between different users

- **Job of OS:**
  - Manage hardware resources
    - Allocation, protection, reclamation, virtualization
  - Provide services to app. How? → system call
    - Abstraction, simplification, standardization
A peek into Unix/Linux

- Application
- Libraries
- Portable OS Layer
- Machine-dependent layer

User space/level

Kernel space/level

- User/kernel modes are supported by hardware
- Some systems do not have clear user-kernel boundary
Unix: Application

- Application (E.g., emacs)
- Libraries
- Portable OS Layer
- Machine-dependent layer

Written by programmer
Compiled by programmer
Uses function calls
Unix: Libraries

Application

Libraries (e.g., stdio.h)

Portable OS Layer

Machine-dependent layer

Provided pre-compiled
Defined in headers
Input to linker (compiler)
Invoked like functions
May be “resolved” when program is loaded
Typical Unix OS Structure

Application

Libraries

Portable OS Layer

Machine-dependent layer

system calls (read, open..)
All “high-level” code
Typical Unix OS Structure

Application

Libraries

Portable OS Layer

Machine-dependent layer

- Bootstrap
- System initialization
- Interrupt and exception
- I/O device driver
- Memory management
- Kernel/user mode switching
- Processor management
System Call

• Invoked via non-priviliged instruction
  – TRAP
  – Treated often like an interrupt, but its “somewhat” different

• Synchronous transfer control

• Side-effect of executing a trap in userspace is that an exception is raised and program execution continues at a prescribed instruction in the kernel → syscall_handler()
Steps in Making a System Call

Example:
read (fd, buffer, nbytes)
System Calls (POSIX)

- System calls for process management
- Example of fork used in simplified shell program

```c
#define TRUE 1
while(TRUE) {
    type_prompt();
    read_command(command, parameters);
    if (fork() != 0) {
        /* some code*/
        waitpid(-1,&status, 0);
    } else {
        /* some code*/
        execve(command, parameters,0);
    }
}
```

Portable Operating System Interface for Unix (IEEE standard 90’s)
System Calls (POSIX)

- System calls for file/directory management
  - \texttt{fd=open(file,how,....)}
  - \texttt{n=wride(fd,buffer,nbytes)}
  - \texttt{s=rmdir(name)}

- Miscellaneous
  - \texttt{s=kill(pid,signal)}
  - \texttt{s=chmod(name,mode)}
System Calls (Windows Win32 API)

- **Process Management**
  - `CreateProcess` - new process (combined work of fork and `execve` in UNIX)
    - In Windows - no process hierarchy, event concept implemented
  - `WaitForSingleObject` - wait for an event (can wait for process to exit)

- **File Management**
  - `CreateFile`, `CloseHandle`, `CreateDirectory`, ...
  - Windows does not have signals, links to files, ..., but has a large number of system calls for managing GUI
List of important syscalls

<table>
<thead>
<tr>
<th>Posix</th>
<th>Win32</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Process Management</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fork</td>
<td>CreateProcess</td>
<td>Clone current process</td>
</tr>
<tr>
<td>execve</td>
<td>CreateProcess</td>
<td>Replace current process</td>
</tr>
<tr>
<td>wait(pid)</td>
<td>WaitForSingleObject</td>
<td>Wait for a child to terminate.</td>
</tr>
<tr>
<td>exit</td>
<td>ExitProcess</td>
<td>Terminate process &amp; return status</td>
</tr>
</tbody>
</table>

**File Management**

| open        | CreateFile     | Open a file & return descriptor                  |
| close       | CloseHandle    | Close an open file                               |
| read        | ReadFile       | Read from file to buffer                        |
| write       | WriteFile      | Write from buffer to file                       |
| lseek       | SetFilePointer | Move file pointer                               |
| stat        | GetFileAttributesEx | Get status info                      |

**Directory and File System Management**

| mkdir       | CreateDirectory | Create new directory                            |
| rmdir       | RemoveDirectory | Remove *empty* directory                        |
| link        | (none)          | Create a directory entry                        |
| unlink      | DeleteFile      | Remove a directory entry                        |
| mount       | (none)          | Mount a file system                             |
| umount      | (none)          | Unmount a file system                           |

**Miscellaneous**

| chdir       | SetCurrentDirectory | Change the current working directory           |
| chmod       | (none)              | Change permissions on a file                   |
| kill        | (none)              | Send a signal to a process                     |
| time        | GetLocalTime       | Elapsed time since 1 Jan 1970                  |

A Few Important Posix/Unix/Linux and Win32 System Calls
OS Service Examples

• Services that need to be provided at kernel level
  – System calls: file open, close, read and write
  – Control the CPU so that users won’t stuck by running
    while ( 1 ) ;
  – Protection:
    • Keep user programs from crashing OS
    • Keep user programs from crashing each other

• Services that can be provided at user level
  – Read time of the day
Is Any OS Complete?
(Criteria to Evaluate OS)

Portability
Security
Fairness
Robustness
Efficiency
Interfaces
Source Code to Execution

1. C source
2. Compiler
3. Assembly
4. Assembler
5. Object File
6. Library
7. Linker
8. Loader
9. DLL
10. Executable
What happens to your program after it is compiled but before it can be executed?
The OS Expectation

• The OS expects executable files to have a specific format
  – Header info
    • Code locations and size
    • Data locations and size
  – Code & data
  – Symbol Table
    • List of names of things defined in your program and where they are defined
    • List of names of things defined elsewhere that are used by your program, and where they are used.
Example of Things

```c
#include <stdio.h>
extern int errno;

int main () {

    printf ("hello, world\n")

    <check errno for errors>
}

• Symbol defined in your program and used elsewhere
  • main

• Symbol defined elsewhere and used by your program
  • printf
  • errno
Two Steps Operation: Parts of OS

**Linking**
- Stitches independently created object files into a single executable file (i.e., a.out)
- Resolves cross-file references to labels
- Listing symbols needing to be resolved by loader

**Loading**
- copying a program image from hard disk to the main memory in order to put the program in a ready-to-run state
- Maps addresses within file to physical memory addresses
- Resolves names of dynamic library items
- schedule program as a new process
Libraries (I)

• Programmers are expensive.
• Applications are more sophisticated.
  – Pop-down menus, streaming video, etc
• Application programmers rely more on library code to make high quality apps while reducing development time.
  – This means that most of the executable is library code
Libraries (II)

- A collection of subprograms
- Libraries are distinguished from executables in that they are not independent programs
- Libraries are "helper" code that provides services to some other programs
- Main advantages: reusability and modularity
Static Libraries

• These libraries are stored on disk.
• Linker links only the libraries referenced by the program.
• Main disadvantage: needs a lot of memory (for example, consider standard functions such as printf and scanf. They are used almost by every application. Now, if a system is running 50-100 processes, each process has its own copy of executable code for printf and scanf. This takes up significant space in the memory.)
Dynamic Link Libraries (Shared Libraries)

• Why not keep those shared library routines in memory and link at object file when needed? (DLLs)
• A shared library is an object module that can be loaded at run time at an arbitrary memory address, and it can be linked to by a program in memory.
• An application can request a dynamic library during execution
• Main advantage: saving memory
• Main disadvantage: ~10% performance hit
A Bit About Relocation

• modifies the object program so that it can be loaded at an address different from the location originally specified

• The compiler and assembler (mistakenly) treat each module as if it will be loaded at location zero

(e.g. **jump 120** is used to indicate a jump to location 120 of the current module)
A Bit About Relocation

• To convert this relative address to an absolute address, the linker adds the base address of the module to the relative address.

• The base address is the address at which this module will be loaded.

Example: Module A is to be loaded starting at location 2300 and contains the instruction
  jump 120
  The linker changes this instruction to
  jump 2420
A Bit About Relocation

• How does the linker know that Module A is to be loaded starting at location 2300?
  – It processes the modules one at a time. The first module is to be loaded at location zero. So relocating the first module is trivial (adding zero). We say that the relocation constant is zero.
  – After processing the first module, the linker knows its length (say that length is L1).
  – Hence the next module is to be loaded starting at L1, i.e., the relocation constant is L1.
  – In general the linker keeps the sum of the lengths of all the modules it has already processed; this sum is the relocation constant for the next module.
A Bit About Relocation

Module M5 will go here. Its relocation constant is L1+L2+L3+L4.
A Bit About Relocation

Base $M4 = L1 + L2 + L3$

Value of $f = \text{Base} + \text{rel}$
LAB assignment #1

Due 6/17
LAB #1: Write a Linker

- Link “==merge” together multiple parts of a program

- What problem is solved?
  - External references need to be resolved
  - Module relative addressing needs to be fixed

```
289:~/NYU/lab1/example> cat print.c
#include <stdio.h>

void print_hello()
{
    printf("Hello world\n");
}

291:~/NYU/lab1/example> cat main.c
#include <stdio.h>

extern void print_hello();
int main(int argc, char **argv)
{
    print_hello();
}
```
LAB #1: Write a Linker

- Simplified module specification
  - List of symbols defined and their value by module
  - List of symbols used in module (including external)
  - List of "instructions"

# of (symbol,value) defined: xy=2

Addressing
I: Immediate
R: Relative
A: Absolute
E: External
Lab #1: Write a Linker

Fancy Output (not req)

Symbol Table
xy=2
z=15

Memory Map
+0
0: R 1004 1004+0 = 1004
1: I 5678 5678
2: xy: E 2000 ->z 2015
3: R 8002 8002+0 = 8002
4: E 7001 ->xy 7002
+5
0: R 8001 8001+5 = 8006
1: E 1000 ->z 1015
2: E 1000 ->z 1015
3: E 3000 ->z 3015
4: R 1002 1002+5 = 1007
5: A 1010 1010
+11
0: R 5001 5001+11= 5012
1: E 4000 ->z 4015
+13
0: A 8000 8000
1: E 1001 ->z 1015
2: z: E 2000 ->xy 2002

Required output

Symbol Table
xy=2
z=15

Memory Map
000: 1004
001: 5678
002: 2015
003: 8002
004: 7002
005: 8006
006: 1015
007: 1015
008: 3015
009: 1007
010: 1010
011: 5012
012: 4015
013: 8000
014: 1015
015: 2002

input
1 xy 2
2 z xy
5 R 1004 I 5678 E 2000 R 8002 E 7001
0
1 z
6 R 8001 E 1000 E 1000 E 3000 R 1002 A 1010
0
1 z
2 R 5001 E 4000
1 z 2
2 xy z
3 A 8000 E 1001 E 2000