计算原理导论

Introduction to Computing Principles

天津大学　计算机科学与技术学院　刘志磊

Software

- CPU implements "machine code" instructions
  - Each machine code instruction is extremely simple
  - e.g. add 2 numbers
  - e.g. compare 2 numbers

- Javascript code we've used: pixel.setRed(10)
  - Javascript is not machine code, so it does not run on the CPU directly.
  - It must be translated to ~10 machine instructions to actually run on the CPU (detail on this later)

- A CPU understands a low level "machine code" language
  - also known as "native code"

- Machine code is hardwired into the design of the CPU hardware
  - it is NOT something that can be changed at will.
  - Each family of compatible CPUs (e.g. the very popular Intel x86 family) has its own, idiosyncratic machine code which is not compatible with the machine code of other CPU families.

- The machine code defines a set of individual instructions.
  - Each machine code instruction is extremely primitive (原始的)
  - such as adding two numbers or testing if a number is equal to zero

- When stored, each instruction takes up just a few bytes.

- A CPU can execute 2 billion operations per second
  - "operations" refers to simple machine code instructions.

- "Program" (程序)
  - e.g. Firefox/Chrome, millions of simple machine code instructions
  - Instructions, like grains of sand making up sculpture

- CPU runs a "fetch/execute cycle"
  - fetch one instruction in sequence
  - execute (run) that instruction, e.g. do the addition
  - fetch the next instruction, and so on

- Some instructions may affect the order that the CPU takes through the instruction sequence
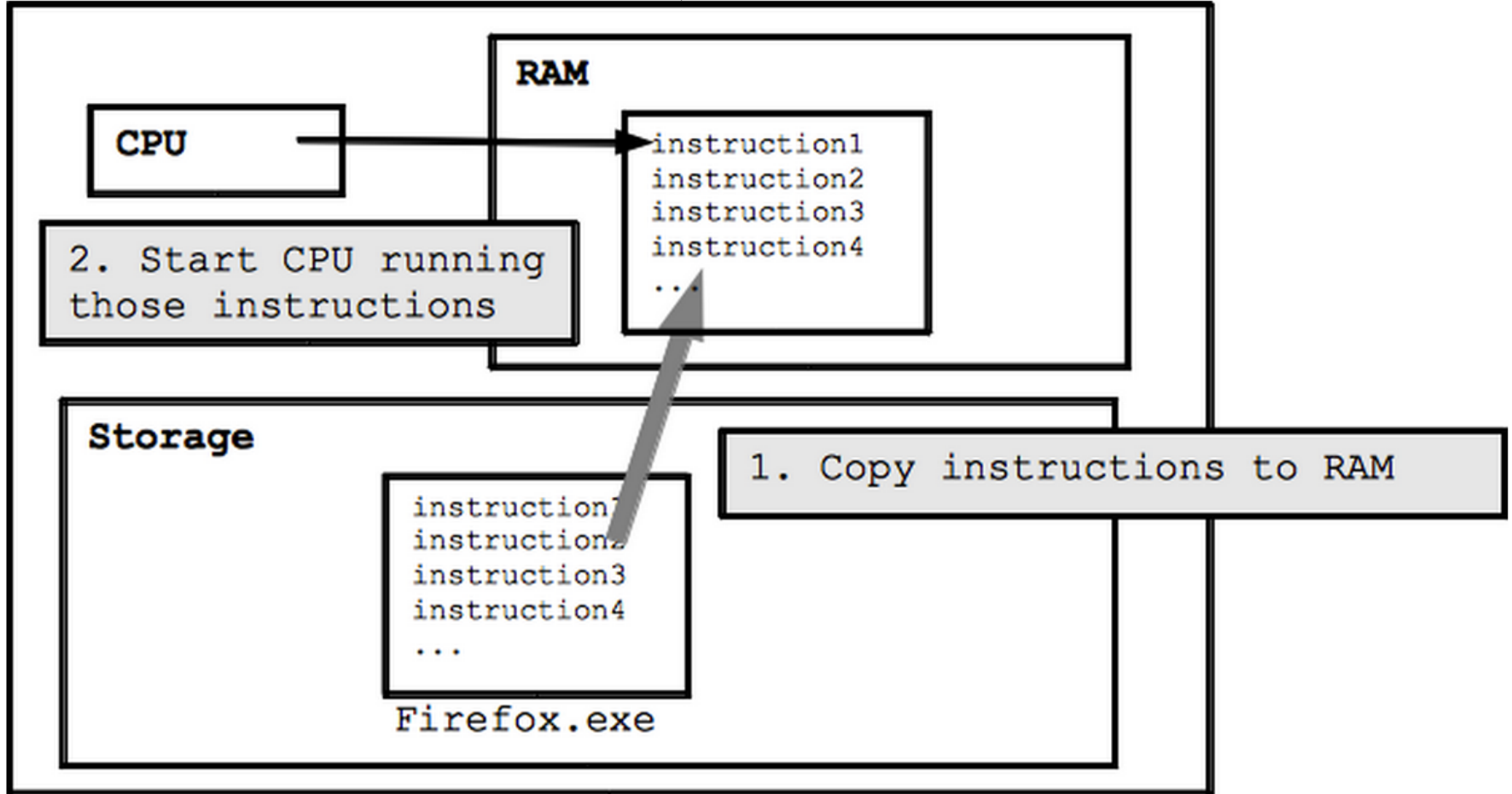
- "loop" instruction
  - jump back 10 instructions
  - Loops are implemented this way
- "if" instruction
  - skip ahead if a certain condition is true
  - If statements are implemented this way


```
                          RAM
    CPU                        instruction1        program =
                               instruction2          series of CPU
                               instruction3          instructions
                               instruction4
                               ...
run =
  CPU fetch/execute
  cycle runs through
  the instruction
  series
```
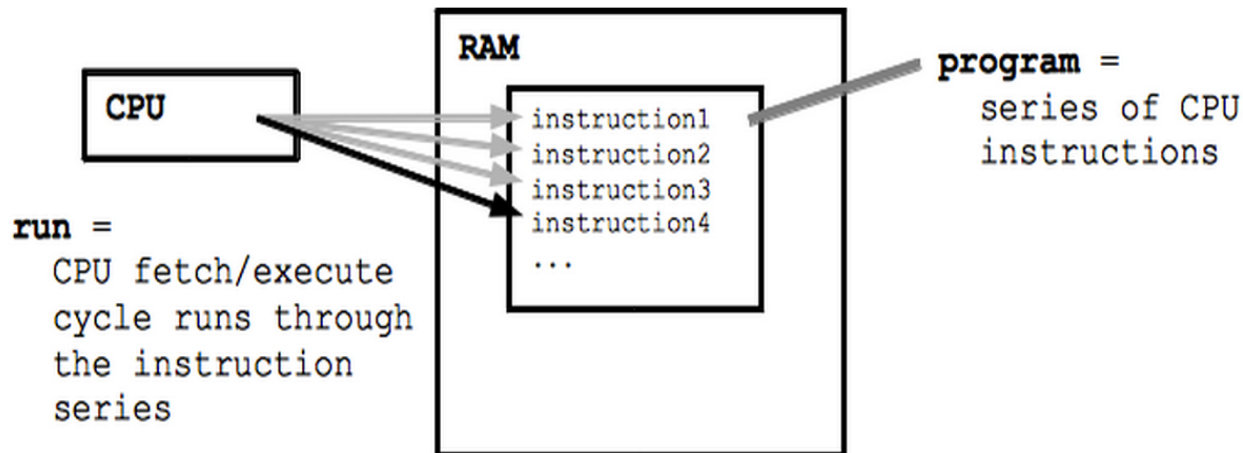
- A program, like Firefox.exe (.exe is a Windows convention), which is a big collection of bytes

- The Firefox.exe bytes are basically the millions of instructions
  - Double click Firefox.exe to Run
  - The instruction bytes are copied up into RAM
  - The CPU is directed to start running at the first instruction

Who did this?

RAM

CPU

instruction1
instruction2
instruction3
instruction4
...

2. Start CPU running
those instructions

Storage

1. Copy instructions to RAM

instruction1
instruction2
instruction3
instruction4
...

Firefox.exe

- Who starts Firefox?
  - Operating System: Windows, Linux, Android, iOS
  - Set of supervisory programs run when computer first starts

- Invisible administration behind the scenes

- Starting/managing/ending other programs
  - Multitasking: run multiple programs at the same time
  - Operating system keeps each program run isolated
  - Program has its own RAM, its own windows on screen
  - vs. accidental or malicious action between programs
  - e.g. Laptop, Digital camera

- # Who runs the operating system?
  - Chicken and egg problem..
  - When first powered on, computer runs a tiny "get started" program. That program typically looks for a disk containing an operating system to run

- # Boot up (启动)
  - Start

- # Reboot （重启）
  - shutdown/start-fresh cycle

- From Programmer to the CPU:
  - We've written small Javascript programs
  - We've seen large program like Firefox
  - Computer language used by a person (e.g. Javascript)
  - vs. the simple machine code instructions in the CPU

- What's the connection?
- Here the basic themes, not the details

- It is extremely rare to write machine code by hand.

- Instead, a programmer writes code in a more "high level" computer language with features that are more useful and powerful than the simple operations found in machine code.

- For CS101, we write code in Javascript which supports high level features such as *strings, loops, and the print()* function.

- None of those high level features are directly present in the low level machine code; they are added by the Javascript language.

- There are two major ways that a computer language can work.

• Computer languages
  • "high level" features
  • e.g. loops, if-statements, strings
  • e.g. C, C++, Javascript, Java
  • Programmer writes "source code" of a program in a language

```
// C++ code
a = "hi";
b = a + "!";
```

• How to get program to the CPU?
  • Having the .exe allows one to run the program
  • To add feature or fix a bug, ideally you want the source code
  • Add a feature in the source code, then run the compiler again to make a new version of the .exe

- Compiler (编译器) **translates** the source code into a large number of machine code instructions
    - Suppose a high level construct, like an *if-statement*, can be implemented by a sequence of 5 machine code instructions
    - e.g. Firefox: written in C++, compiler takes in Firefox C++ source code, produces Firefox.exe
    - The compilation step can be done once.
    - The end user does not need to the source code or the compiler.
    - Does not work backwards: having the .exe, you cannot recover the source code (well)

```
Source code

a = "hello";
b = a + "!";
```

Compiler

```
program.exe

instruction1
instruction2
instruction3
instruction4
...
```
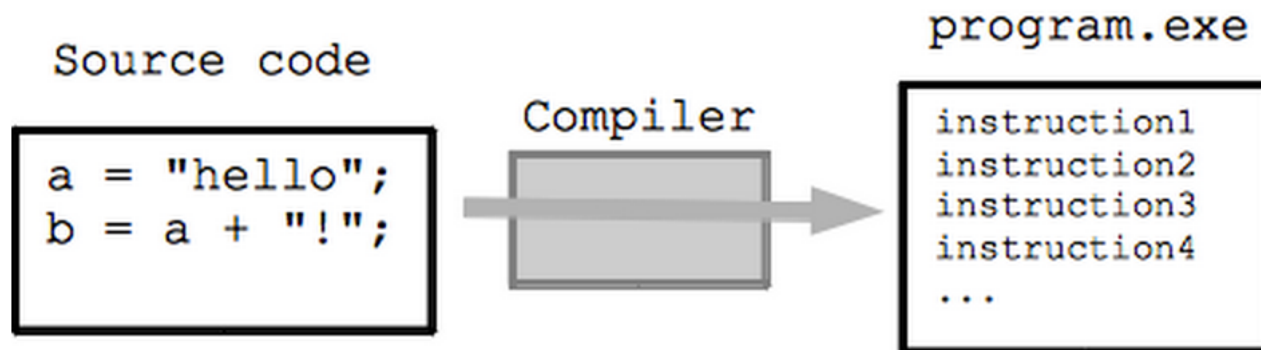
- The source code (源代码) is available to all, along with the right to make modifications
  - Typically the software does not cost anything
  - **Freedom:** the end user is not dependent on the original vendor to fix bugs, or perhaps the vendor goes out of business
  - The user can make the change themselves (since they have access to the source)

  - Insurance/freedom policy
  - Often the license terms will require the change to made available to the wider community in some cases
  - **Open source** is a very successful model for some cases

- "Open Source" (开源) refers to software where the program includes access to its source code, and a license where the user can make their own modifications. Typically open source software is distributed for free.

- Open source software also includes freedom or independence since the user is not dependent on the original vendor to make changes or fixes or whatever to the source code.

```
// Javascript code
a = 1;
b = a + 2;
```

- **Dynamic languages**
  - e.g. Java, Javascript, Python
  - Does not use the compiler/machine-code strategy
  - Can be implemented by an "interpreter" (解释器)

- **Interpreter** is a program which "runs" other code
  - e.g. web browsers includes a Javascript interpreter, browser "runs" bits of Javascript code in a web page
  - Interpreter looks at one line at a time
  - Deconstructs what each line says to do
  - The interpreter then does that action in the moment
  - Then proceeds to the next line

Disclaimer: there are many languages, no one "best" language, it depends!

- Interpreter (解释器)
  - A program that reads a source program and produces the results of executing that program

- Compiler (编译器)
  - A program that translates a program from one language (the *source*) to another (the *target*)

- The compiler **translates** source code to equivalent machine code
- The interpreter **does** the code, looking at each line and doing it

- Compiled code tends to run faster than Interpreter code
    - The compiler has pre-processed the source code, leaving the *program.exe* as lean (精干) and direct as it can be to just run.
    - many questions are resolved by the compiler at compile time
        - how to append to this string, how many bytes do I need here…

- Dynamic/interpreter languages
    - Tend to have a greater number of programmer-friendly features
        - e.g. Memory Management （存储管理）
        - C and C++: partially manual, some programmer input required
        - Dynamic languages: automatic memory management, no programmer input needed
        - Automatic memory management not free: spending CPU cycles to lighten programmer workload

    - Programmers are often more productive in dynamic languages
    - The resulting program tends to run slower than compiled code

    Tradeoff

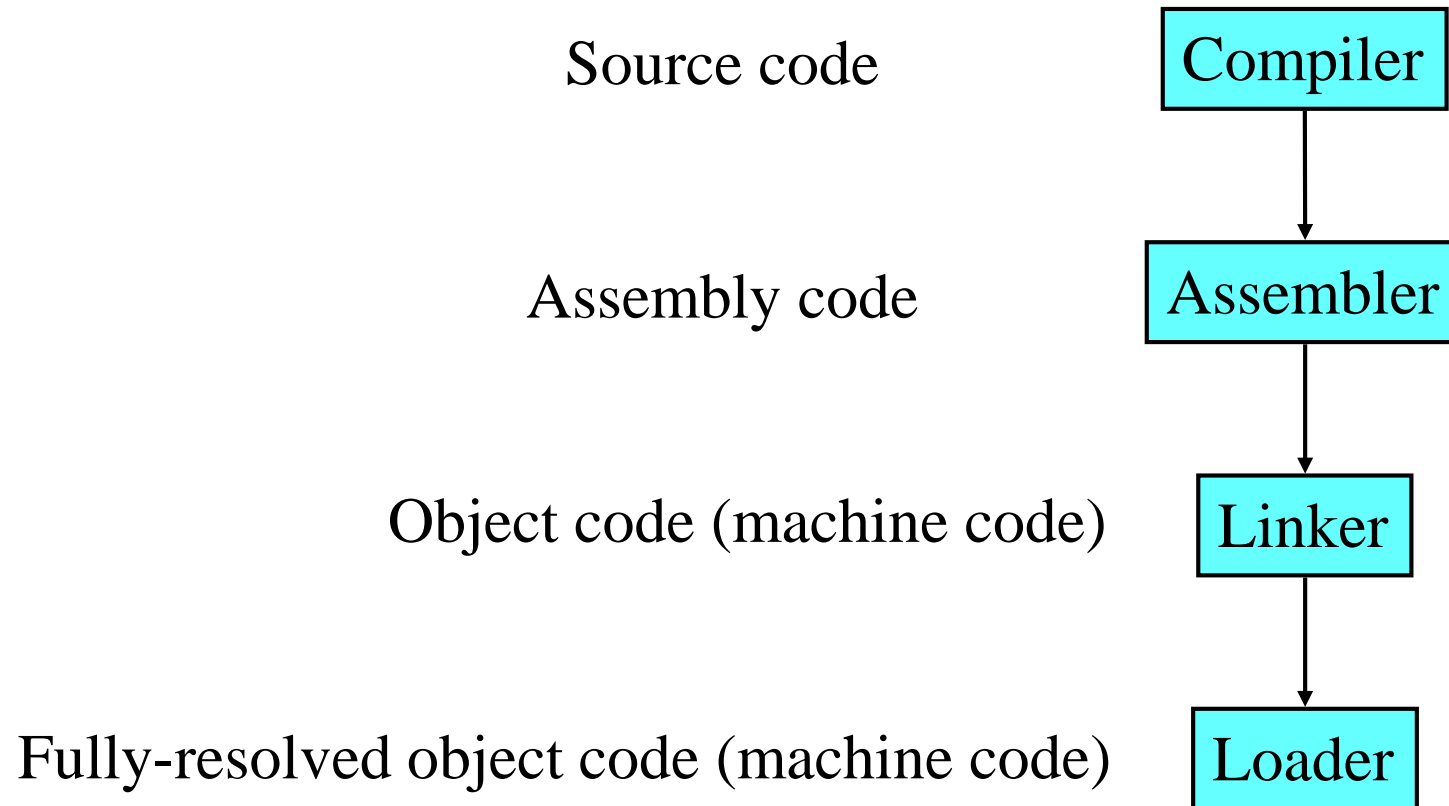Current trend is towards dynamic languages ← Moore's Law

- Just In Time Compiler (JIT, 即时编译器)
  - Compile code of a dynamic language on the fly
  - The interpreter is used for simple cases
  - For important sections of dynamic code, the JIT creates a block of machine code in RAM for that section
  - All major browsers now have a JIT for the Javascript code they run
  - Best of both worlds
  - Flexibility of dynamic languages
  - Combined with most of the performance of the compiled world
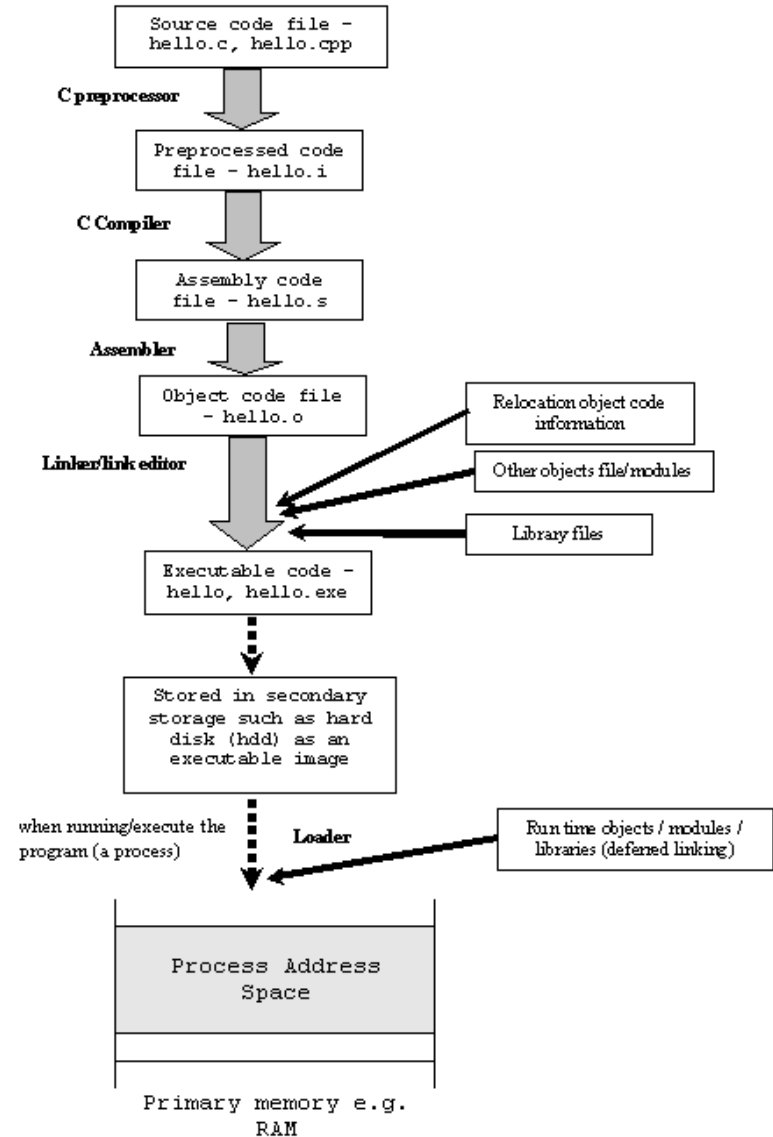 - Active area of research, works pretty well

Normally the program building process involves four stages and utilizes different 'tools' such as a **preprocessor (预处理器), compiler (编译器), assembler (汇编器), and linker (链接器).**

Source code   Compiler

Assembly code   Assembler

Object code (machine code)   Linker

Fully-resolved object code (machine code)   Loader

The right Figure shows the steps involved in the process of building the C program starting from the compilation until the loading of the executable image into the memory for program running.
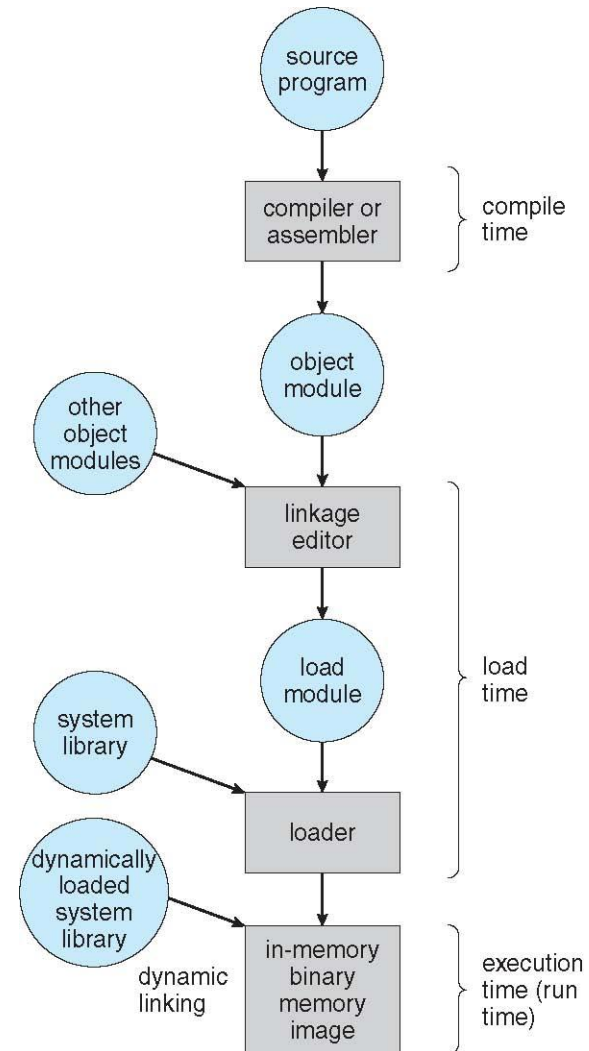
Why Memory Management?


```
Source code file -
hello.c, hello.cpp

C preprocessor

Preprocessed code
file - hello.i

C Compiler

Assembly code
file - hello.s

Assembler

Object code file        Relocation object code
- hello.o               information

Linker/link editor      Other objects file/modules

                        Library files

Executable code -
hello, hello.exe

Stored in secondary
storage such as hard
disk (hdd) as an
executable image

when running/execute the        Run time objects / modules /
program (a process)    Loader   libraries (deferred linking)

Process Address
Space

Primary memory e.g.
RAM
```

Why Memory Management?

- Program must be brought (from disk) into memory and placed within a process（进程）for it to be run

- If only a few processes can be kept in main memory, then much of the time all processes will be waiting for I/O and the CPU will be idle (空闲)

- Hence, memory needs to be allocated efficiently in order to pack as many processes into memory as possible

Running time of a user program

How to do memory management？

➔ Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time:** If memory location known a priori, absolute code can be generated; must recompile code if starting location changes
  - **Load time:** Must generate relocatable code if memory location is not known at compile time
  - **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers)

Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** (逻辑地址) – generated by the CPU; also referred to as **virtual address (虚地址)**
  - **Physical address (虚地址)** – address seen by the memory unit

- Logical and physical addresses are the same in compile-time and load-time address-binding schemes;

- logical (virtual) and physical addresses differ in execution-time address-binding scheme

Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address

- In MMU scheme, the value in the relocation register (寄存器) is added to every address generated by a user process (用户进程) at the time it is sent to memory

- The user program deals with *logical* addresses; it never sees the *real* physical addresses

Dynamic relocation using a relocation register
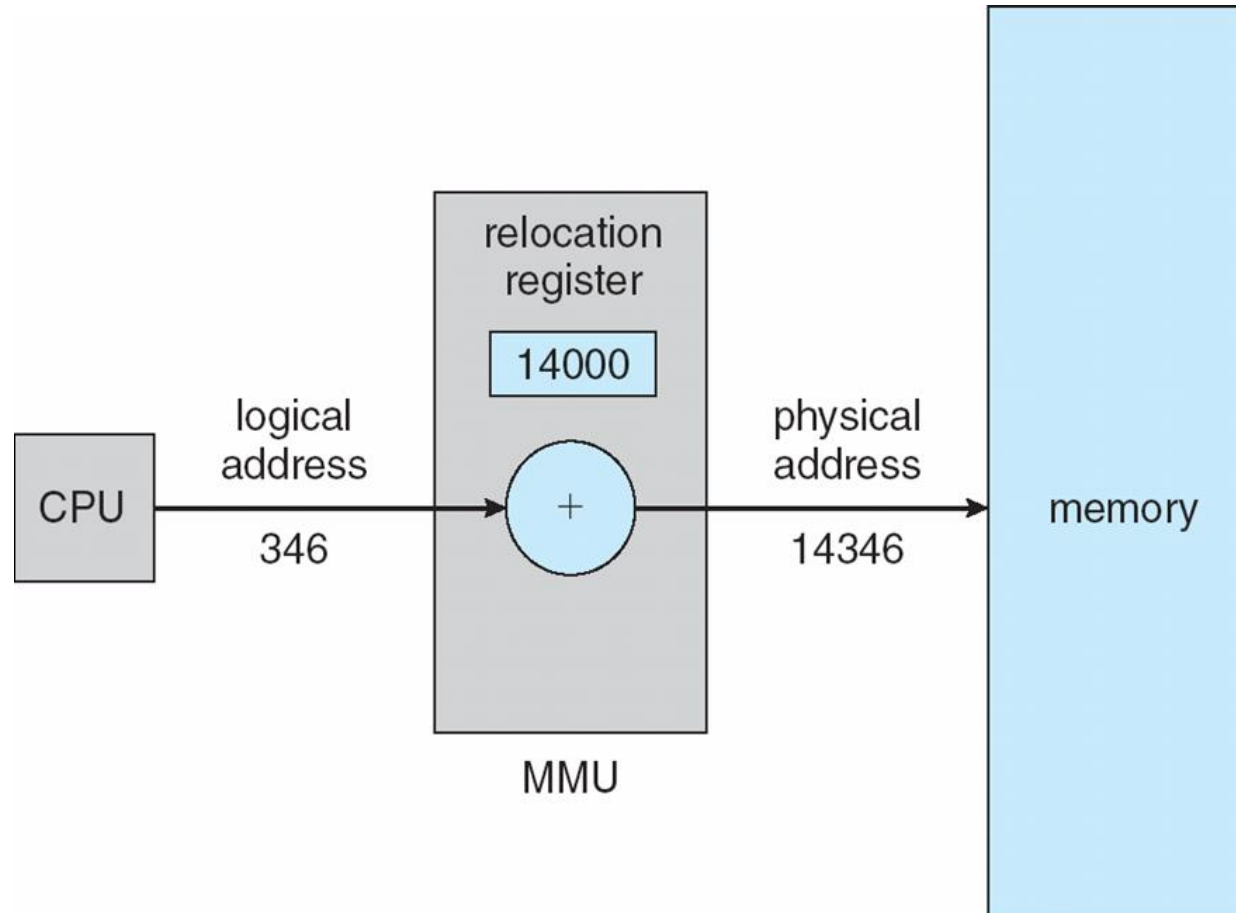
Dynamic Loading

- Routine/Program is not loaded until it is called

- Better memory-space utilization
  - Unused routine/Program is never loaded

- Useful when large amounts of code are needed to handle in frequently occurring cases

- No special support from the operating system is required implemented through program design

Dynamic Linking

- Linking postponed until execution time

- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine

- Stub replaces itself with the address of the routine, and executes the routine

- Operating system needed to check if routine is in processes' memory address

- Dynamic linking is particularly useful for libraries

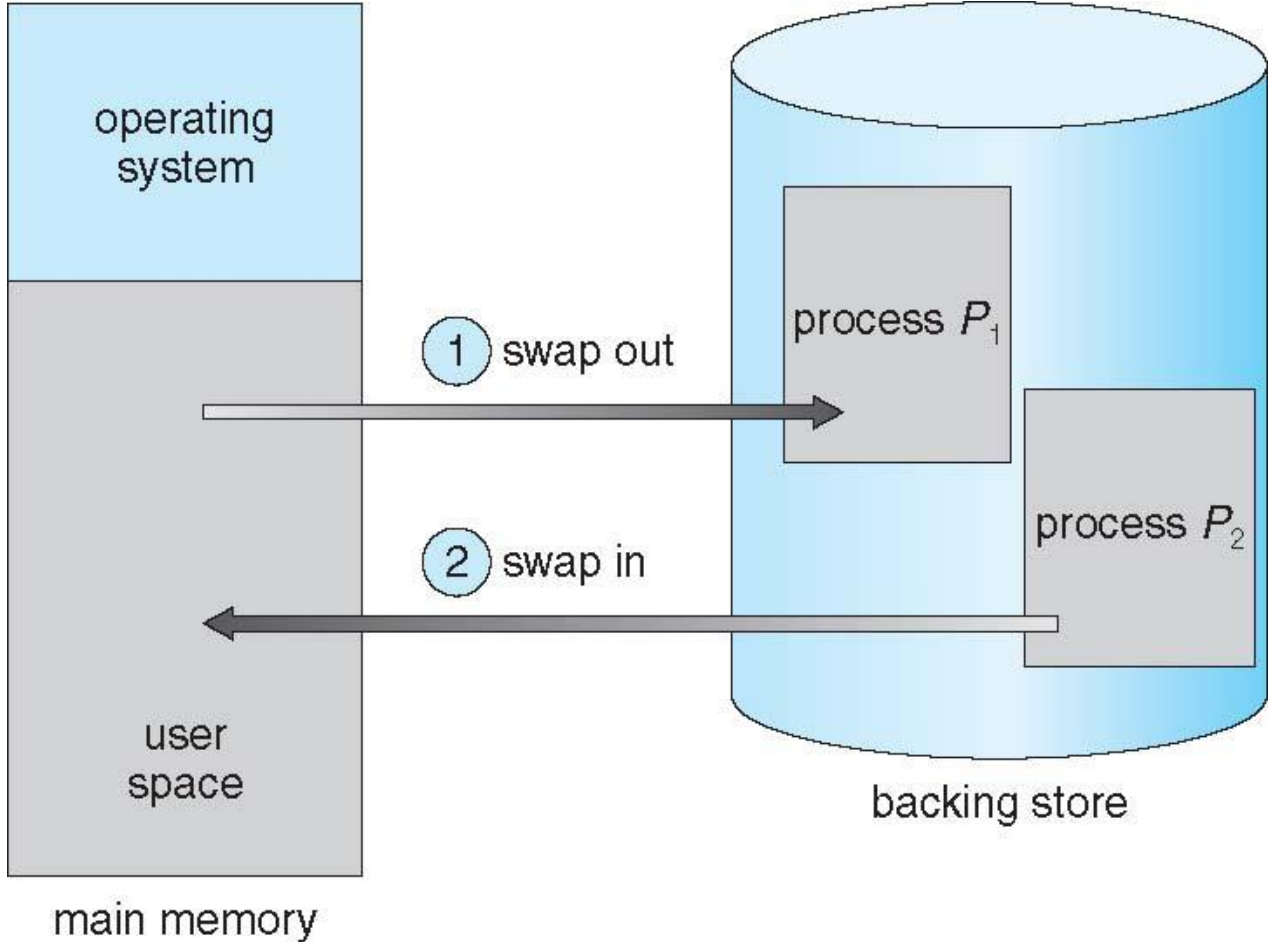- System also known as **shared libraries**

Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution

- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images

- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed

- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped

- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)

- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Schematic View of Swapping

operating
system

① swap out

process P_1

② swap in

process P_2

user
space

backing store

main memory

Thank You!

Q&A