Managed Runtime Technology:

General Introduction

Xiao-Feng Li
(xiaofeng.li@gmail.com)

2012-10-10

Agenda

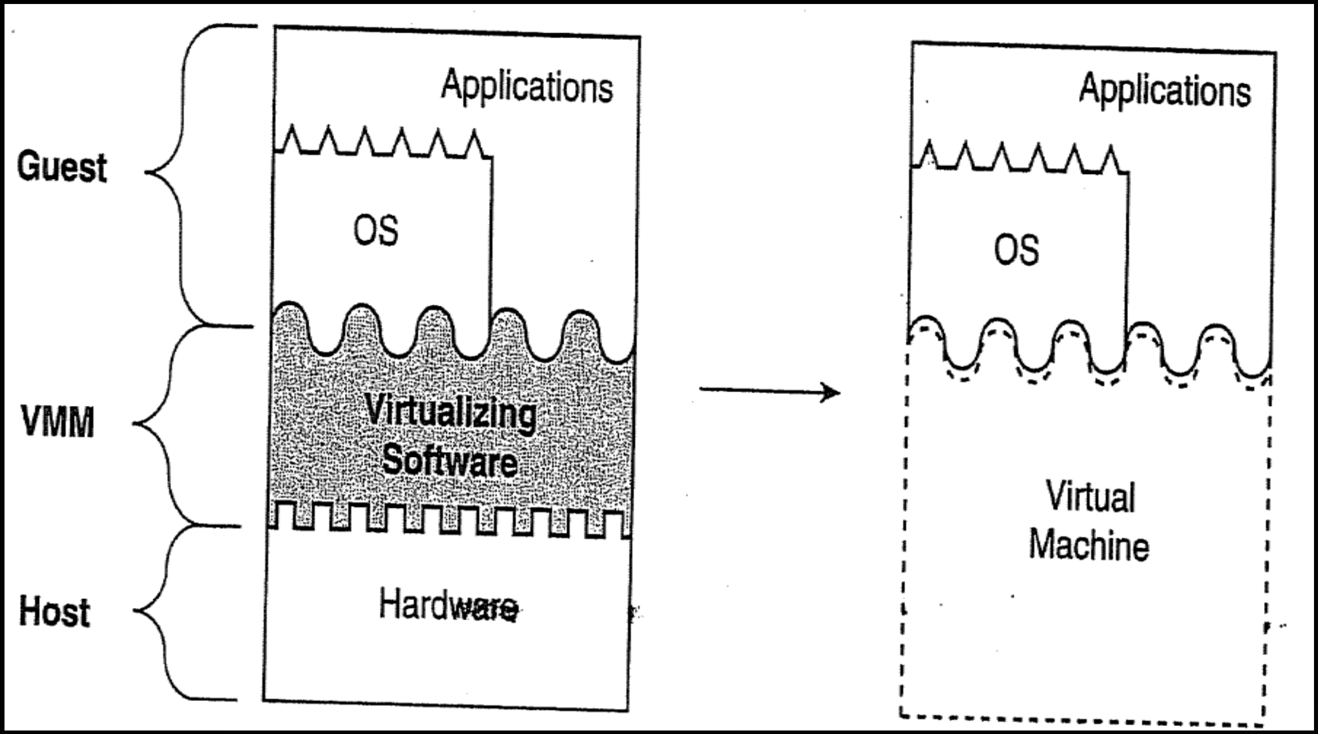
- Virtual machines
- Managed runtime systems
- EE and MM (JIT and GC)
- Summary

What is Virtual Machine?

- A virtual machine is implemented by adding a layer of software to a real machine to support the desired virtual machine’s architecture.
  - Virtualization constructs an isomorphism that maps a virtual *guest* system to a real *host*.
- Virtualization vs. abstraction
  - Virtualization does not necessarily hide details
  - Abstraction uses interfaces to express the abstract model

System (ISA) Virtual Machine

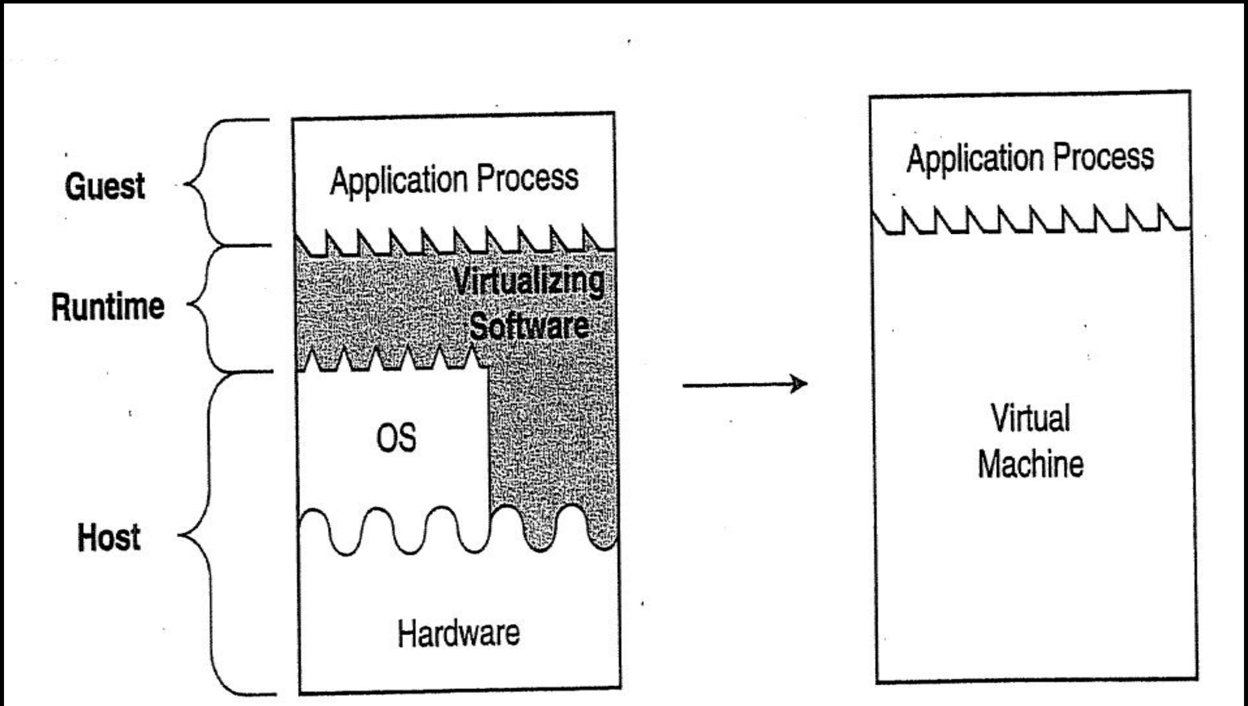
- Such as Vmware, Xen, VirtualBox, etc.



Picture from "Virtual Machines: Versatile Platforms for Systems and Processes"

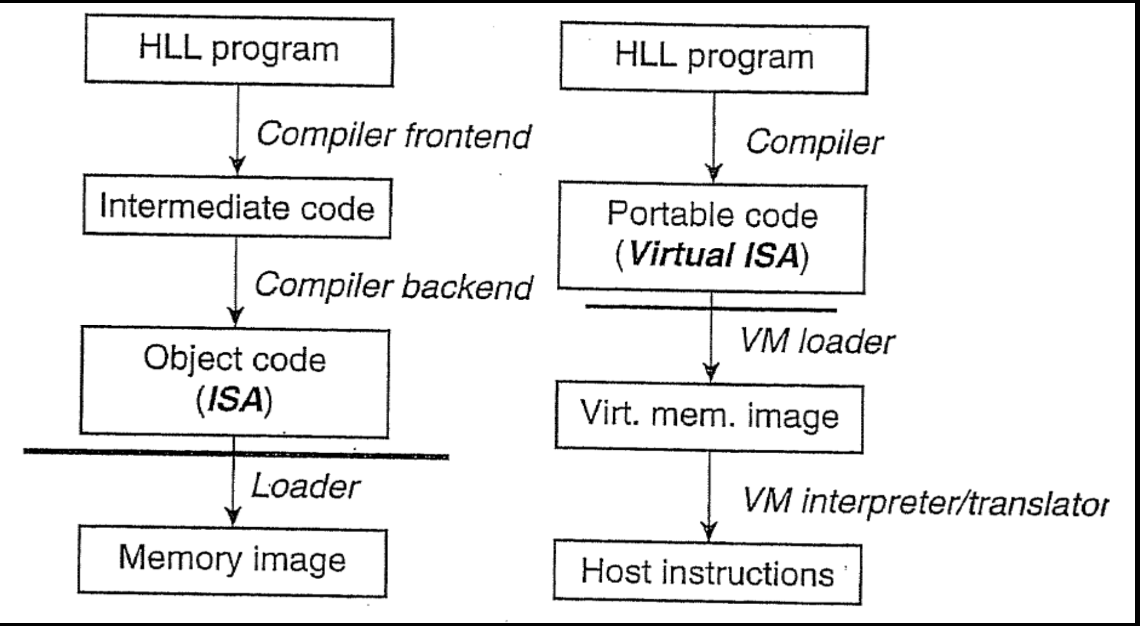
Process (ABI) Virtual Machine

- Different ISAs: Digital FX!32, IA32 EL, etc.
- Same ISAs: such as Dynamo, etc.



Virtual-ISA Virtual Machine

- Such as JVM, CLI, etc.
- Virtual machine → Runtime engine



Language Runtime Engine

- Such as Javascript, Python, HTML, etc.
- Levels: ISA → ABI → Virtual-ISA -> Language
- Abstractions:
  - From all architecture details to language interface
  - Usually depend on other supportive functions
- Our focus: "Managed Runtime System"
  - Virtual-ISA virtual machine, and language runtime engine

Link Runtime to Services

- Language
  - Programming language → interface description language → service description language
  - Such as AIDL (Android), WSDL (web service)
- Service entity
  - Process local service → Inter-process service → cross-machine service → Internet → cloud computing

Agenda

- Virtual machines
- Managed runtime systems
- EE and MM (JIT and GC)
- Summary

Managed Runtime Systems

- Why?
  - Portability
  - Security
  - Productivity

Modern Programming

- Programming in safe languages
  - No illegal pointer dereference
    - a = 0; b = *a;
  - No illegal control flow
    - jmp format_disk;
- Then what's safe language?
  - "type-safe" language and plus
    - If it does not allow operations or conversions which lead to erroneous conditions
  - Memory safety and Control safety
  - Can download-and-run safely ☺

Safe Languages

- Almost all the modern languages are safe
  - Commercial: Java, C#, SQL, etc.
  - Academic: LISP, Haskell, ML, etc.
  - Scripting: Ruby, Javascript, PHP, etc.
  - Widely used today and probably tomorrow
- Unsafe languages
  - C, C++, asm, etc.
  - Will only be used by system programmers
  - Specifically, unsafe languages are commonly used to implement safe languages

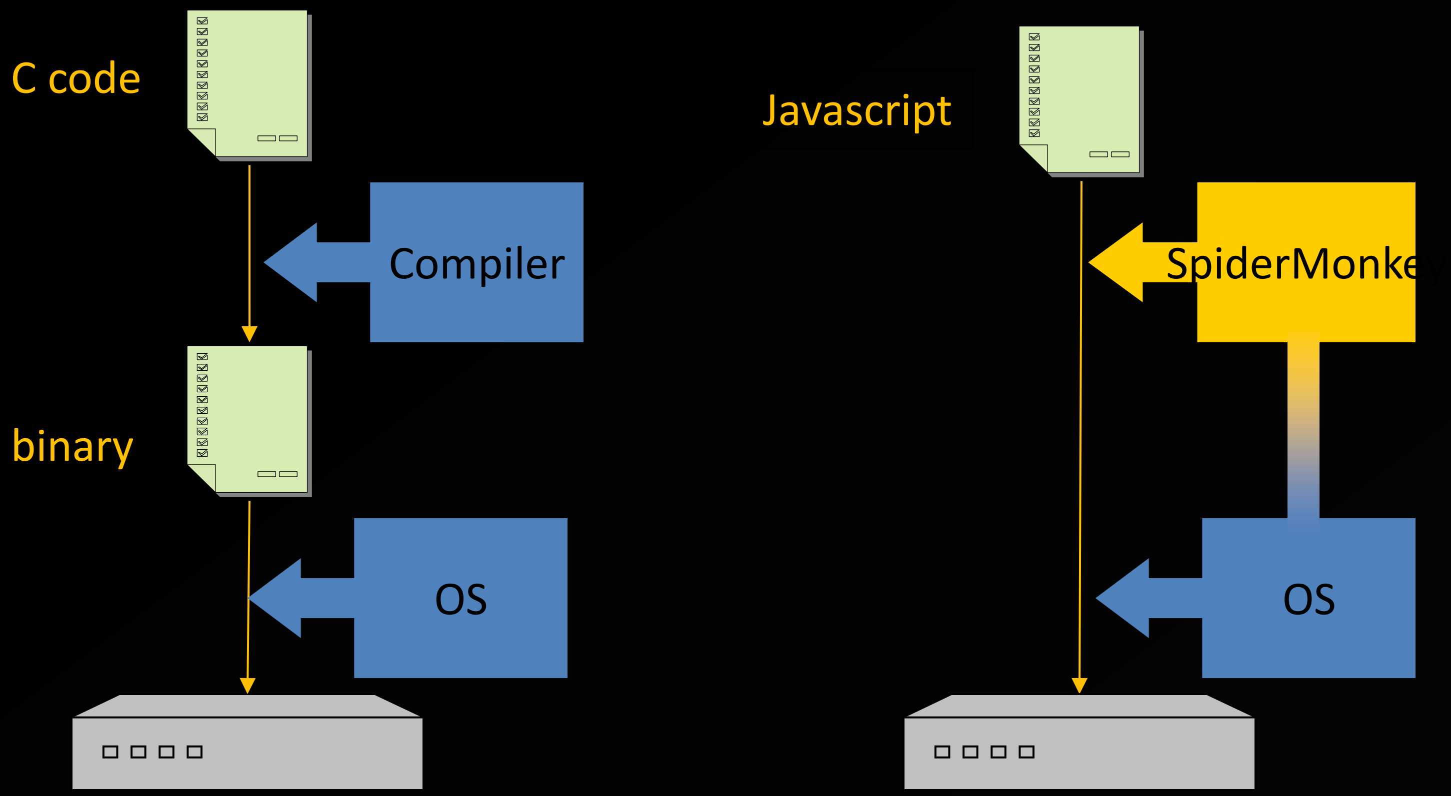
Modern Programming Is Good

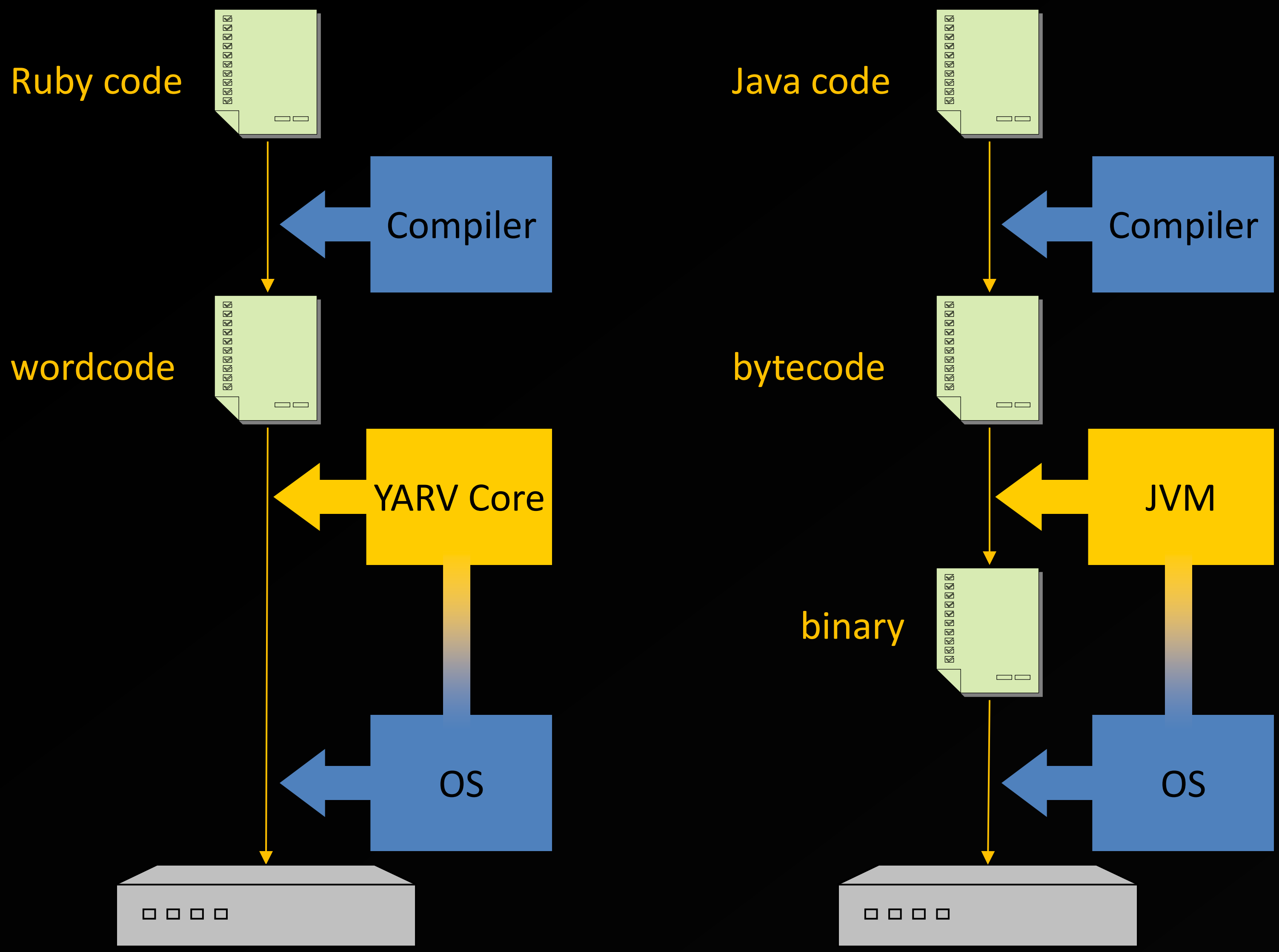
- Programming is becoming higher-level
  - Programmers care only logics
    - "What I want to achieve?"
  - Leave the details to system
    - "How it is achieved?"
  - E.g., details like memory allocation/release, thread scheduling, etc.

- The question:
  - How the system provides the supports?
  - This is our work

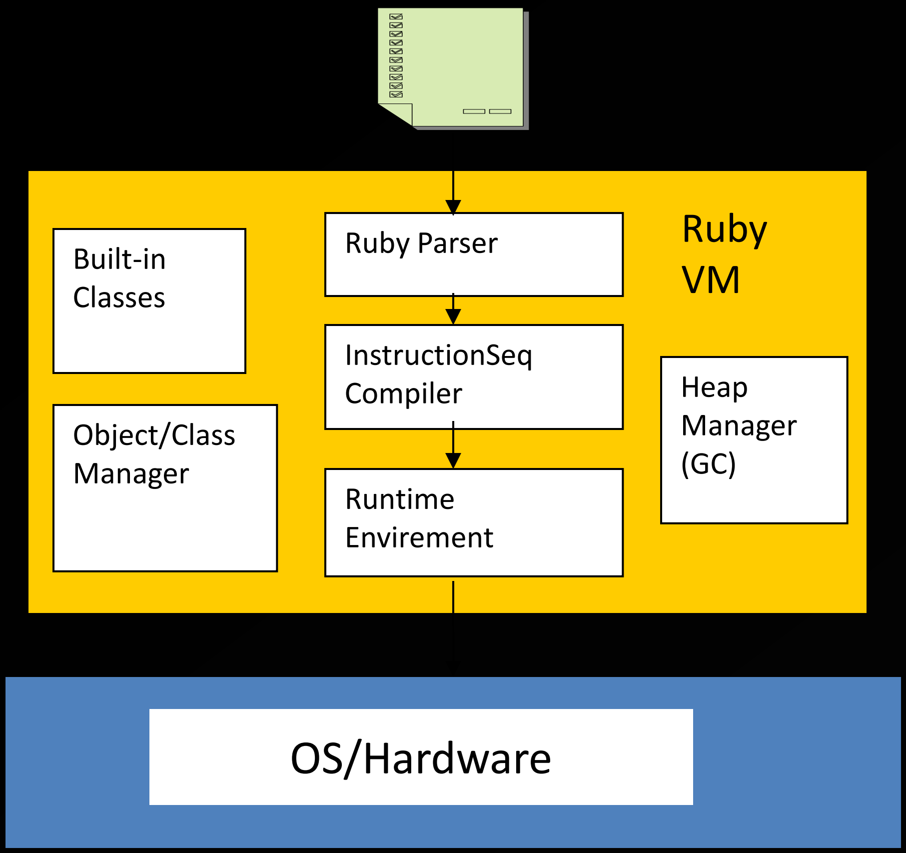
Supports to Modern Prog.

- Original system supports
  - Compiler:
    - Syntax checking
    - Compilation into binary
  - OS
    - Page allocation and management
    - Process scheduling
- What are missing?
  - Runtime safety checking
  - Runtime interpretation or compilation
  - Memory management at object granularity


C code
Compiler
binary
OS
Javascript
SpiderMonkey
OS

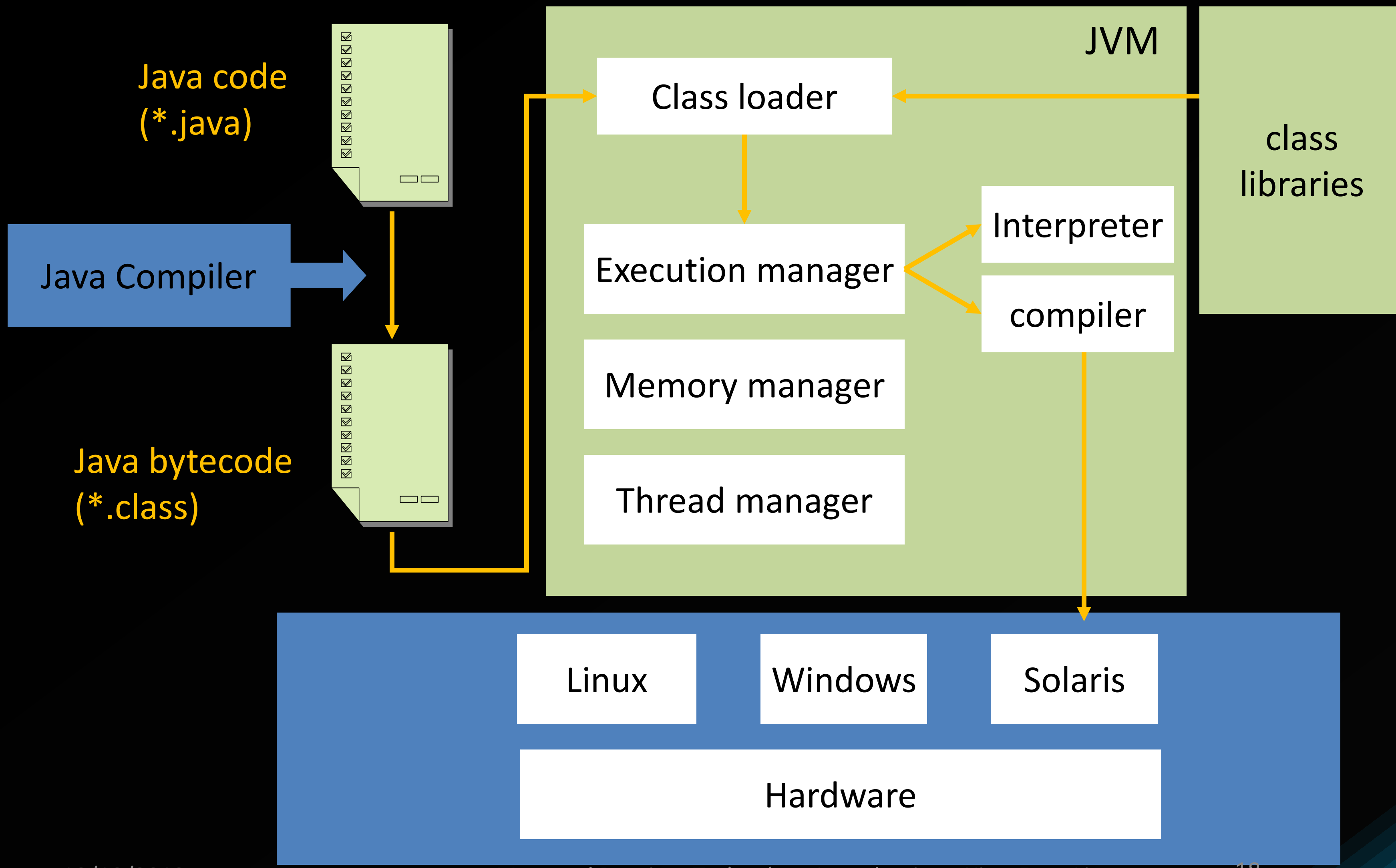

Ruby code
Compiler
wordcode
YARV Core
OS
Java code
Compiler
bytecode
JVM
binary
OS

Example: Ruby Engine



- Ruby VM
  - Parser: convert scripts into AST
  - Compiler: compile AST to wordcode sequence
  - Runtime environment: Yarv core
  - Object/class manager
  - Heap manager

Example: Java Virtual Machine



JVM: a Computing Environment

- For any computing environment, you have:
  - Programming language: C/C++
  - Compiler + OS + machine: gcc/g++ + Linux + X86
  - Runtime library: glibc/glibc++
- Java provides the same things:
  - Programming language: defined by Gosling/Joy/Steele in 1995
  - Virtual machine: JVM has execution engine + threading/MM + ISA
  - Library: class libraries with standard API
  - All are specified into *standards*

JVM Specification

- Mainly defines four things:
  1. Instruction set for the VM (bytecodes)
  2. JVM executable file format (Java class file)
  3. Class loading and verification (ensure the program does not compromise the JVM integrity)
  4. Java threading and memory model (the system)

- Specification says nothing on implementation method
  - Can be implemented as a process in OS, or
  - Part of OS kernel, or OS is part of JVM, or
  - In hardware, or anything

Agenda

- Virtual machines
- Managed runtime systems
- EE and MM (JIT and GC)
- Summary

JVM Execution Engine

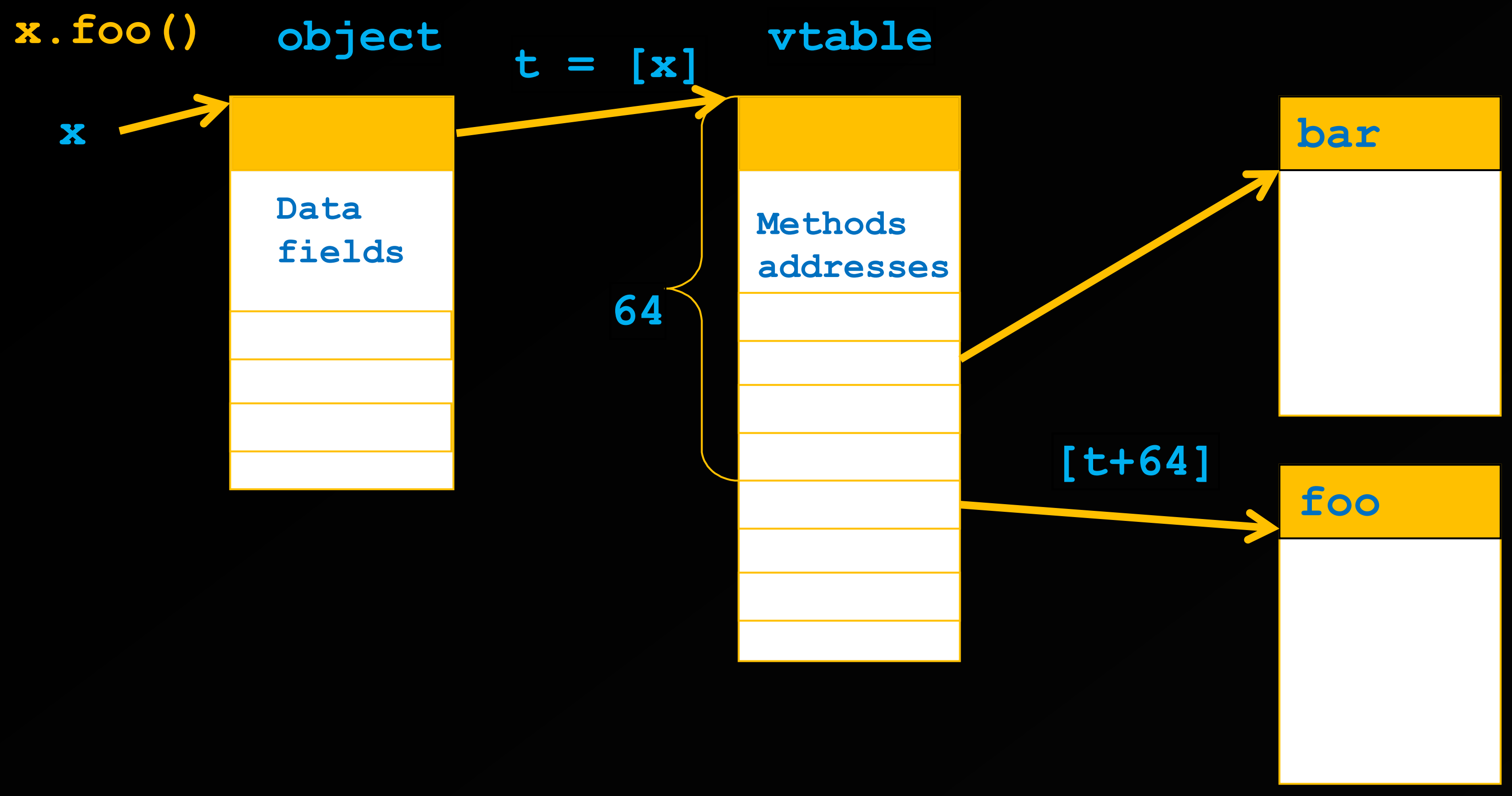
- Executes Java bytecodes either using interpreter or Just-In-Time compiler
- Registers:
  - **PC**: Program Counter
  - **FP**: Frame Pointer
  - **SP**: Operand Stack Top Pointer

- Interpreter: directly interpret the bytecodes
- JIT compiler: compile the bytecode into machine code then execute

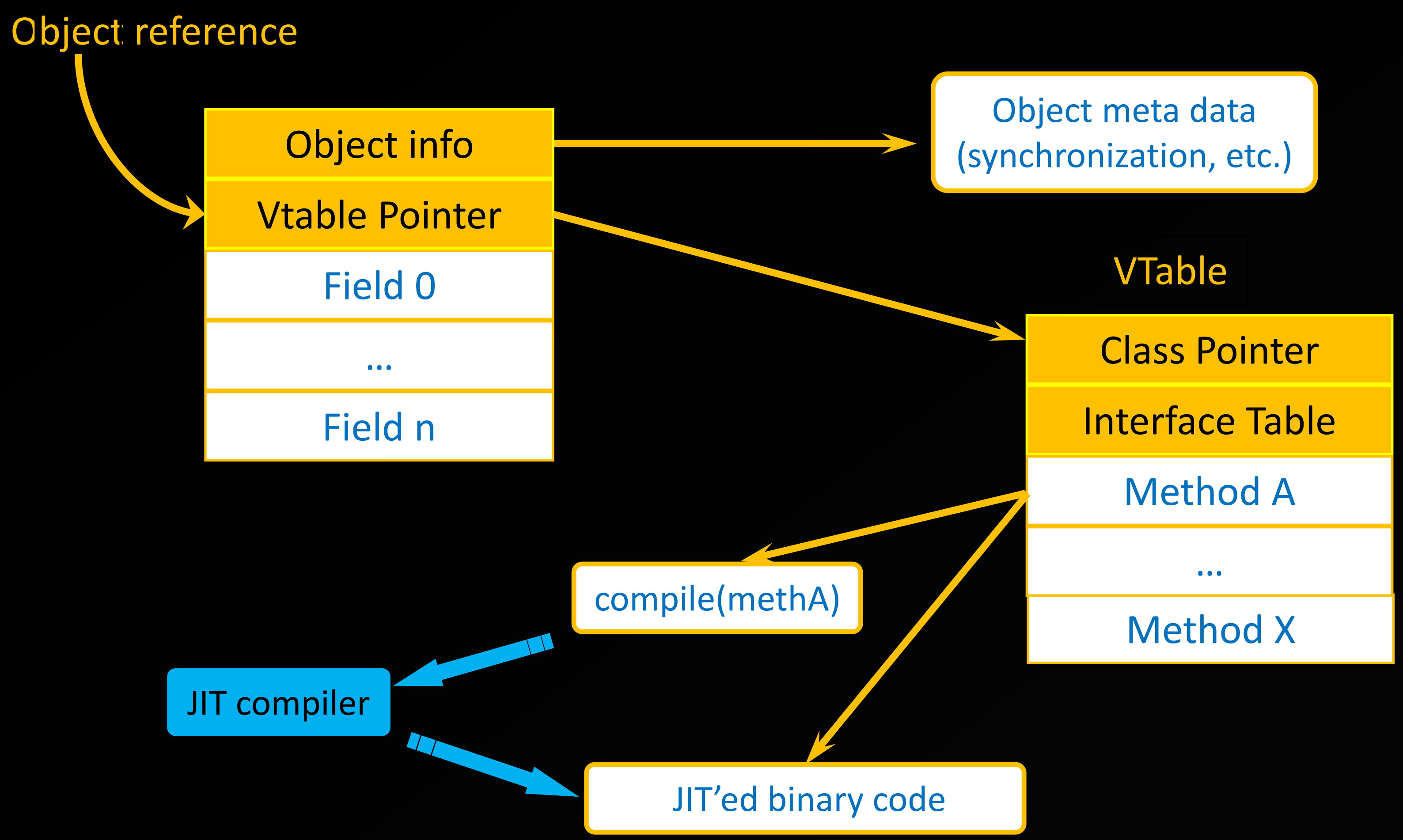
Bytecode Interpreter

```
while( program not end ) {
  fetch next bytecode => b
  switch(b) {
    case ILOAD:
      load an integer from the local variable array and push on top
      of current operand stack;
    case ISTORE:
      pop an integer from the top of  current operand stack and store
      it into the local variable array;
    case ALOAD:
      ... ...
  } // end of switch
} // end of while
```

Object and Vtable



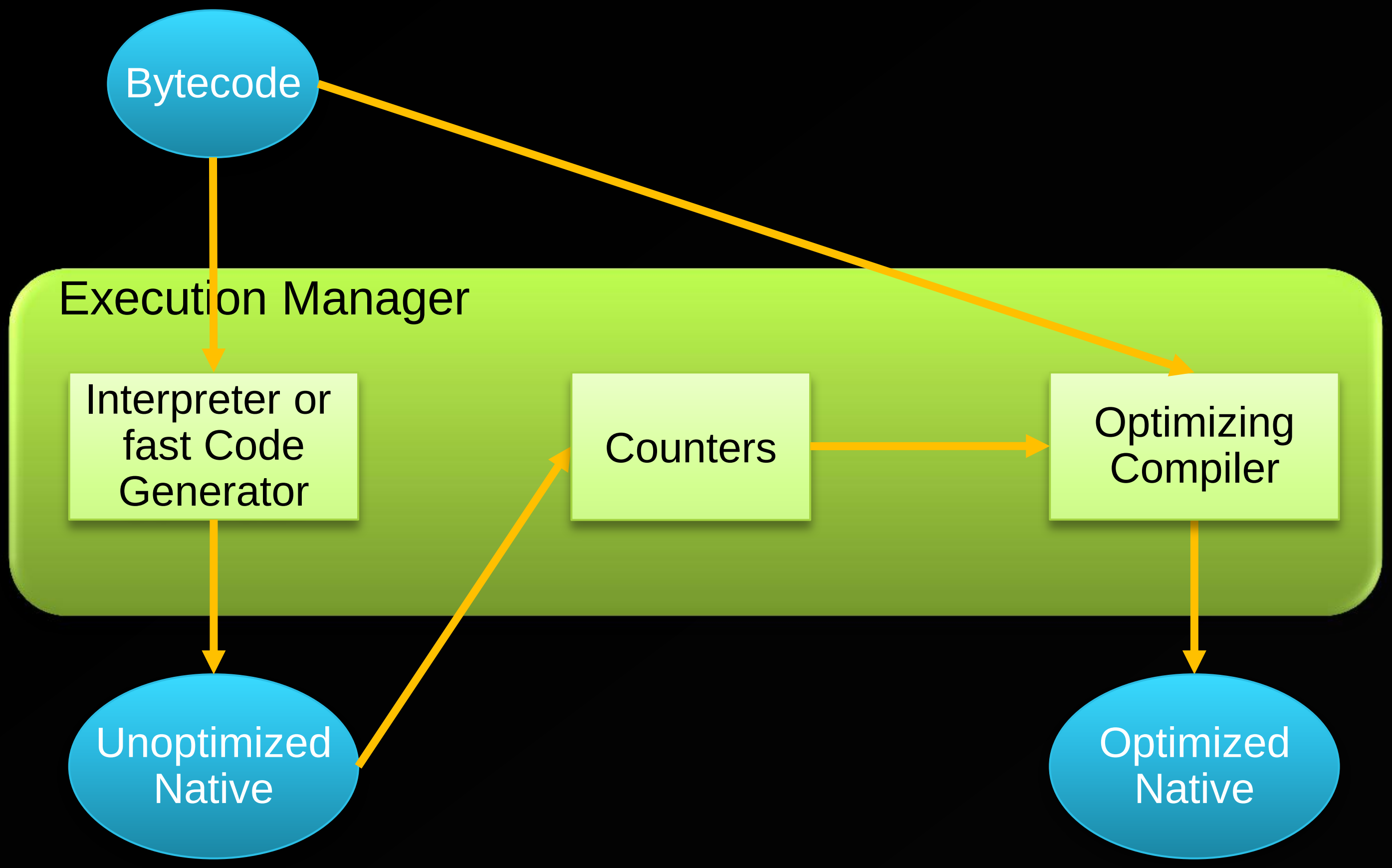
JIT Compiler



Interpreter vs. Compiler

- Key difference
  - Whether the engine needs to generate native code during execution
  - Interpreter: Portable code execution
    - Easy to implement, no extra memory for compiled code
  - JIT Compiler: Faster code execution
- Tradeoffs between execution time and compilation time
  - Also tradeoffs between optimization levels

Adaptive Compilation



Method-based vs. Trace-based JIT

- Method-based JIT
  - The method to compile is dynamically identified
  - Pros: large optimization scope
  - Cons: not all the method code is executed
    - If execution is very dynamic, optimization is difficult
- Trace-based JIT
  - The code in the trace is dynamically identified
  - Pros: only compile hot traces and no control flow
  - Cons: hot trace has to be stable
    - If execution is very dynamic, potential trace explosion

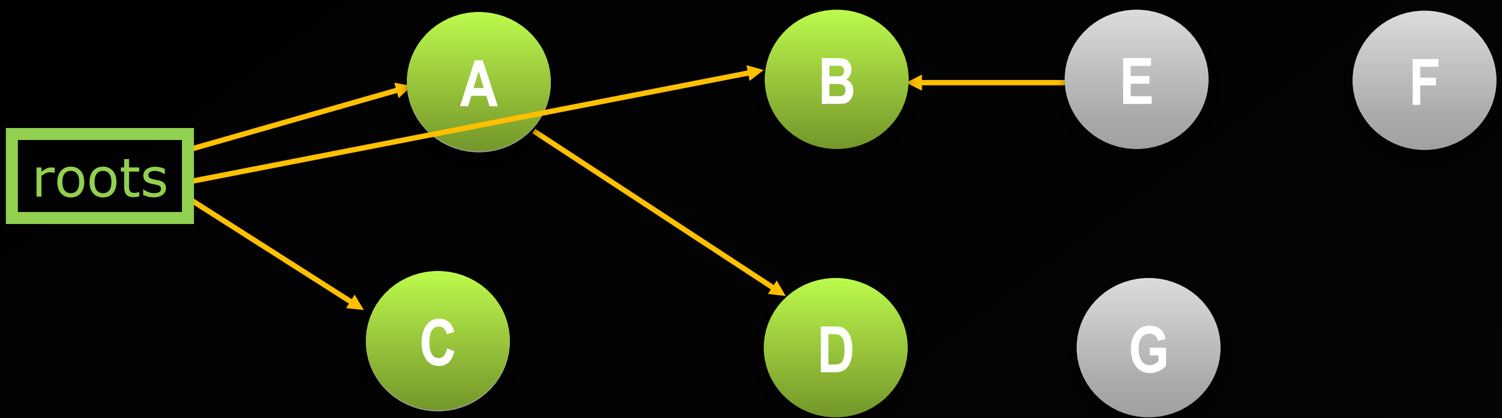
Garbage Collection

- Reference counting vs. Reachability analysis
- Reference counting:
  – Object dead when reference count is zero
  – Pros: Real-time death identification
  – Cons: runtime overhead to maintain counters
- Reachability analysis
  – Object alive when it is reachable from app
  – Pros: no need to maintain counters
  – Cons: need suspend app to find reachable objects

Xiao-Feng Li, et al, Cycler: Improve Heap Management for Allocation-intensive Applications with On-the-fly Object Reuse, PDCS'2011

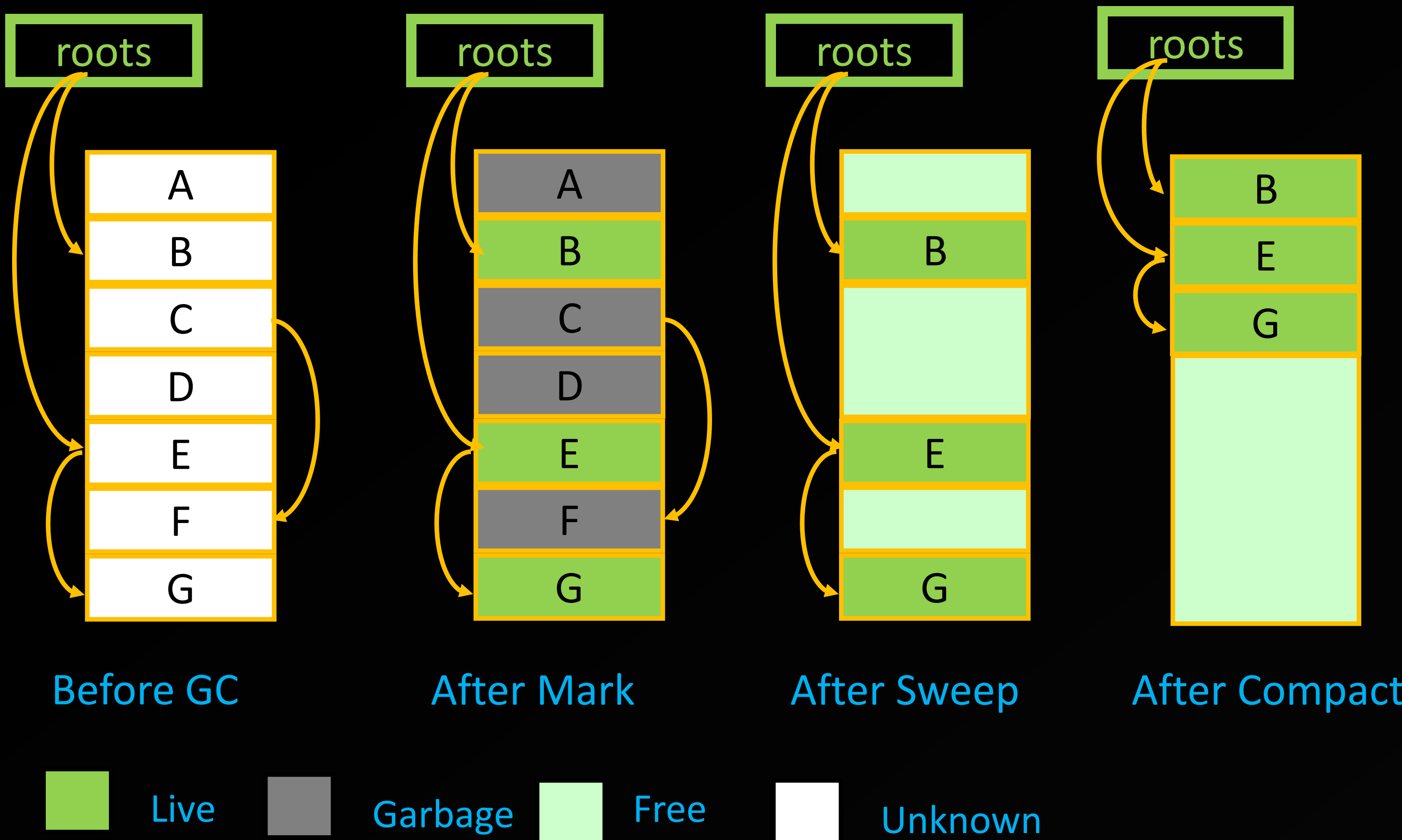
Reachability Analysis

- Roots
  - object references in threads' execution context
- Live object
  - The object that is reachable from root reference or other live object
- Dead object (garbage)
  - Objects that are not accessible to the application



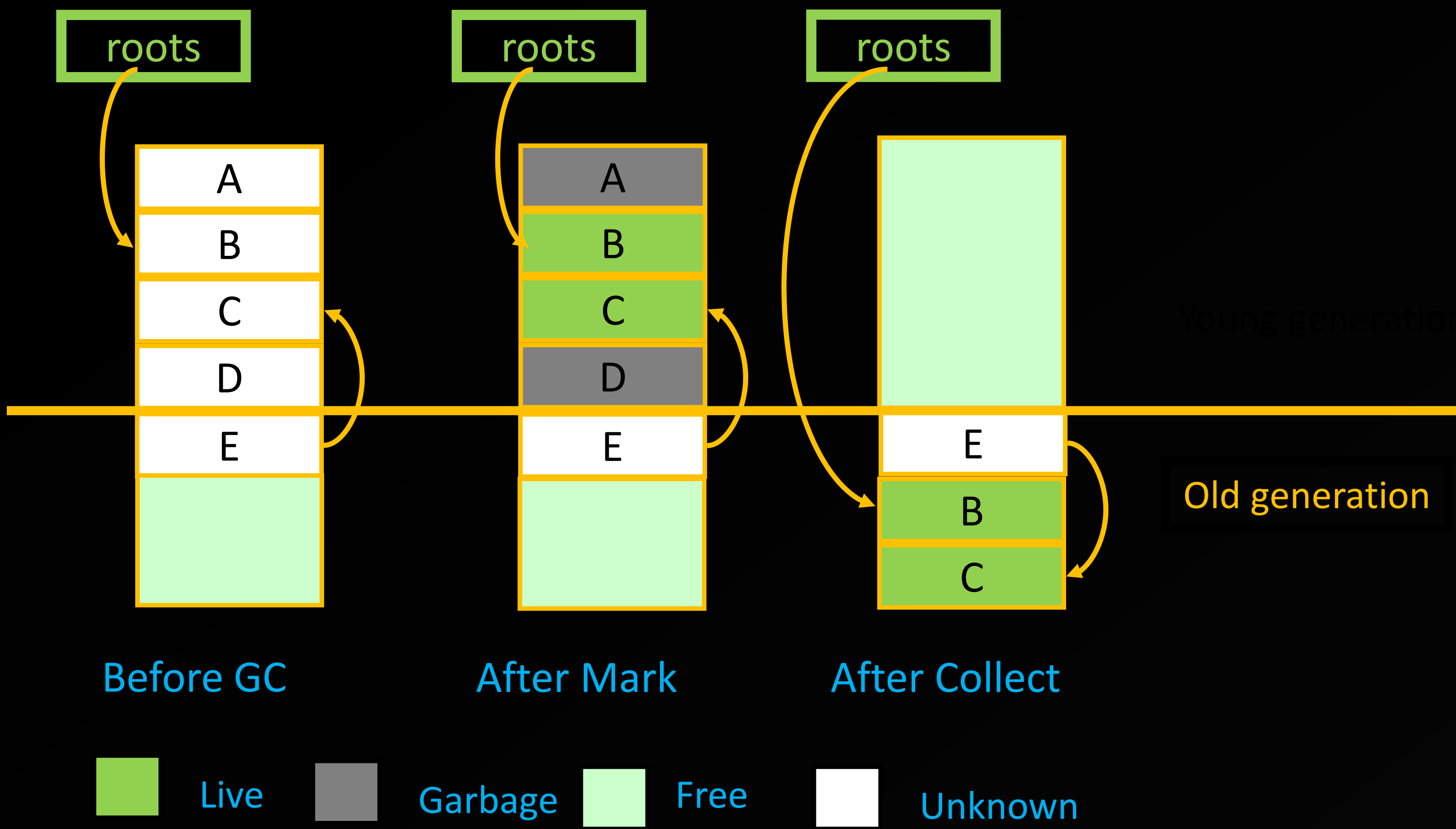
Mark-Sweep-Compaction

- Algorithm
  - Trace heap from roots to mark all live objects
  - Sweep garbage from the heap, reclaiming their space
  - Compact the heap to reduce the fragmentation
- Pros
  - Fast, non-moving
- Cons
  - Space fragmentation
  - Object access locality

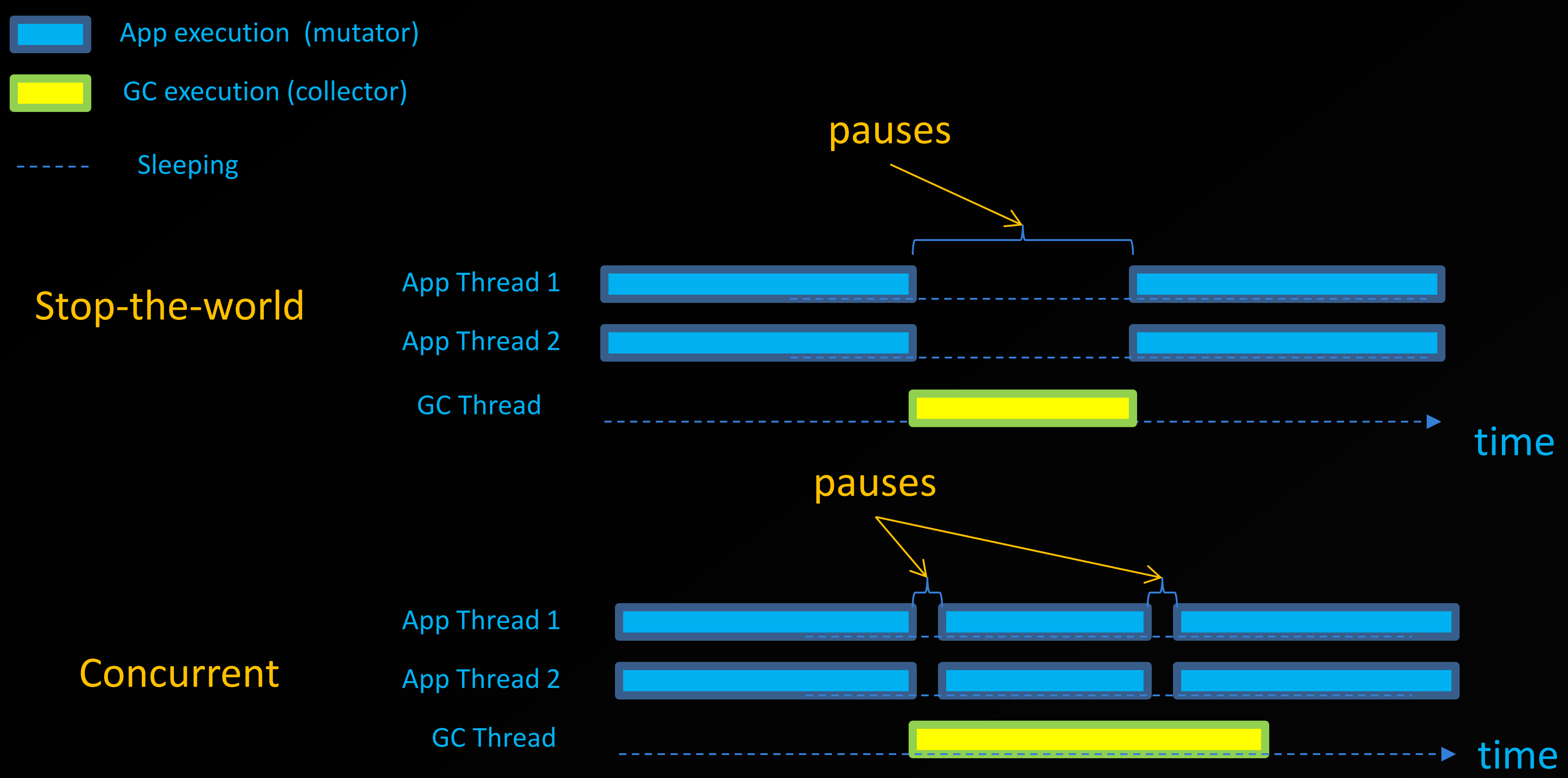

roots
A
B
C
D
E
F
G
Before GC
roots
A
B
C
D
E
F
G
After Mark
roots
B
E
G
After Sweep
roots
B
E
G
After Compact
Live
Garbage
Free
Unknown

Generational Collection

- Hypothesis: most objects die young, small percentage long live
- Algorithm
  - Partition the heap into generations, collect only the younger generation mostly
  - When older generations are full, collect entire heap
- Pros
  - Short pause time
- Cons
  - Floating garbage


roots
roots
roots
A
B
C
D
E
A
B
C
D
E
E
B
C
Old generation
Before GC
After Mark
After Collect
Live
Garbage
Free
Unknown

Concurrent GC



Xiao-Feng Li, et al, Tick: Concurrent GC in Apache Harmony, 2009

Agenda

- Virtual machines
- Managed runtime systems
- EE and MM (JIT and GC)
- Summary

Summary

- Evolution of Virtual Machines and Runtime Systems
- Modern Programming and Managed Runtime Systems
- Differences between a managed runtime system and a traditional runtime system
- Execution Engine and Garbage Collection