PROGRAMMING PARADIGMS

MODULE 6

TOPICS:

Concurrency:-

- **Threads**
- **Synchronization**

Run-time program Management:-

- **Virtual Machines**
- **Late Binding of Machine Code**
- **Reflection**
- **Symbolic Debugging**
- **Performance Analysis**

1. Concurrency

- a program is said to be *concurrent* if it may have more than one active execution context—more than one "thread of control." Concurrency has at least three important motivations:

  1. *To capture the logical structure of a problem.* Many programs, particularly servers and graphical applications, must keep track of more than one largely independent "task" at the same time. Often the simplest and most logical way to structure such a program is to represent each task with a separate thread of control.

  2. *To exploit extra processors, for speed.* Long a staple of high-end servers and super computers, multiple processors have recently become ubiquitous in desktop and laptop machines. To use them effectively, programs must generally be written (or rewritten) with concurrency in mind.

  3. *To cope with separate physical devices.* Applications that run across the Internet or a more local group of machines are inherently concurrent.

Likewise, many embedded applications-the control systems of a modern automobile, for example-often have separate processors for each of several devices.

- In general, we use the word *concurrent* to characterize any system in which two or more tasks may be underway (at an unpredictable point in their execution) at the same time.
- Under this definition, coroutines are not concurrent, because at any given time, all but one of them is stopped at a well-known place.
- A concurrent system is *parallel* if more than one task can be physically *active* at once; this requires more than one processor.

1.1 Concurrent Programming Fundamentals

- Within a concurrent program, we will use the term *thread* to refer to the active entity that the programmer thinks of as running concurrently with other threads.
- In most systems, the threads of a given program are implemented on top of one or more *processes* provided by the operating system. OS designers often distinguish between a *heavyweight* process, which has its own address space, and a collection of *lightweight* processes, which may share an address space. Lightweight processes were added to most variants of Unix in the late 1980s and early 1990s, to accommodate the proliferation of shared-memory multiprocessors.
- We will sometimes use the word *task* to refer to a well-defined unit of work that must be performed by some thread.
- In one common programming idiom, a collection of threads shares a common "bag of tasks"—a list of work to be done.
- Each thread repeatedly removes a task from the bag, performs it, and goes back for another.
- Sometimes the work of a task entails adding new tasks to the bag.
- Unfortunately, terminology is inconsistent across systems and authors.
- Several languages call their threads processes. Ada calls them tasks. Several operating systems call lightweight processes threads. The Mach OS, from which OSF Unix and Mac OS X are derived, calls the address space shared by lightweight processes a task.
- A few systems try to avoid ambiguity by coining new words, such as"actors"or "filaments."

1.1.1 Communication and Synchronization

- In any concurrent programming model, two of the most crucial issues to be addressed are *communication* and *synchronization*.

Communication

- Communication refers to any mechanism that allows one thread to obtain information produced by another.
- Communication mechanisms for imperative programs are generally based on
    - *shared memory* or
    - *message passing*.
- In a **shared-memory programming model**, some or all of a program's variables are accessible to multiple threads. For a pair of threads to communicate, one of them writes a value to a variable and the other simply reads it.
- In a **message-passing programming model**, threads have no common state. For a pair of threads to communicate, one of them must perform an explicit send operation to transmit data to another.

Synchronization

- Synchronization refers to any mechanism that allows the programmer to control the relative order in which operations occur in different threads.
- Synchronization is generally implicit in message-passing models: a message must be sent before it can be received.
    - If a thread attempts to receive a message that has not yet been sent, it will wait for the sender to catch up.
- Synchronization is generally not implicit in shared-memory models: unless we do something special, a "receiving" thread could read the "old" value of a variable, before it has been written by the "sender."
- In both shared-memory and message-based programs, synchronization can be implemented either by *spinning* (also called *busy-waiting*) or by *blocking*.
- In busy-wait synchronization, a thread runs a loop in which it keeps reevaluating some condition until that condition becomes true (e.g., until a message queue becomes nonempty or a shared variable attains a particular value)-presumably as a result of action in some other thread, running on some other processor.

- Note that busy-waiting makes no sense on a uniprocessor: we cannot expect a condition to become true while we are monopolizing a resource (the processor) required to make it true. (A thread on a uniprocessor may sometimes busy-wait for the completion of I/O, but that's a different situation: the I/O device runs in parallel with the processor.)
- In blocking synchronization (also called *scheduler-based* synchronization), the waiting thread voluntarily relinquishes its processor to some other thread.
- Before doing so, it leaves a note in some data structure associated with the synchronization condition.
- A thread that makes the condition true at some point in the future will find the note and take action to make the blocked thread run again.

2. Thread Creation Syntax

- Almost every concurrent system allows threads to be created (and destroyed) dynamically.
- Syntactic and semantic details vary considerably from one language or library to another, but most conform to one of six principal options: **co-begin, parallel loops, launch-at-elaboration, fork (with optional join), implicit receipt, and early reply**.
- The first two options delimit threads with special control-flow constructs.
- The others use syntax resembling (or identical to) subroutines.
- At least one language (SR) provides all six options. Most others pick and choose.
- Most libraries use fork/join, as do Java and C#. Ada uses both launch-at-elaboration and fork.
- OpenMP uses co-begin and parallel loops. RPC systems are typically based on implicit receipt.

Co-begin

- The usual semantics of a compound statement (sometimes delimited with begin. . . end) call for sequential execution of the constituent statements.
- A co-begin construct calls instead for concurrent execution:

```
co-begin – – all n statements run concurrently
stmt 1
stmt 2
. . .
```

stmt n
end

- Each statement can itself be a sequential or parallel compound, or (commonly) a subroutine call.
- Co-begin was the principal means of creating threads in Algol-68. It appears in a variety of other systems as well, including OpenMP:

```
#pragma omp sections
{
# pragma omp section
{ printf("thread 1 here\n"); }
# pragma omp section
        { printf("thread 2 here\n"); }
}
```

- In C, OpenMP directives all begin with #pragma omp.

Parallel Loops

- Many concurrent systems, including OpenMP, several dialects of Fortran, and the recently announced Parallel FX Library for .NET, provide a loop **whose iterations are to be executed concurrently.**
- In OpenMP for C, we might say

```
#pragma omp parallel for
for (int i = 0; i < 3; i++) {
        printf("thread %d here\n", i);
}
```

- In C# with Parallel FX, the equivalent code looks like this:

```
Parallel.For(0, 3, i => {
        Console.WriteLine("Thread " + i + " here");
});
```

- The third argument to Parallel. For is a delegate, in this case a lambda expression.
- In many systems it is the programmer's responsibility to make sure that concurrent execution of the loop iterations is safe, in the sense that correctness will never depend on the outcome of race conditions.
- Access to global variables, for example, must generally be synchronized, to make sure that iterations do not conflict with one another.
- The compiler checks to make sure that a variable written by one thread is neither read nor written by any concurrently active thread.

Launch-at-Elaboration

- In several languages, Ada among them, the code for a thread may be declared with syntax resembling that of a subroutine with no parameters.
- When the declaration is elaborated, a thread is created to execute the code.
- In Ada (which calls its threads tasks) we may write

```
procedure P is
task T is
...
end T;
begin -- P
...
end P;
```

- Task T has its own begin. . . end block, which it begins to execute as soon as control enters procedure P.
- If P is recursive, there may be many instances of T at the same time, all of which execute concurrently with each other and with whatever task is executing (the current instance of) P.
- The main program behaves like an initial default task.
- When control reaches the end of procedure P, it will wait for the appropriate instance of T (the one that was created at the beginning of this instance of P) to complete before returning.
- This rule ensures that the local variables of P are never deallocated before T is done with them.

Fork/Join

- Co-begin, parallel loops, and launch-at-elaboration all lead to a concurrent control-flow pattern in which thread executions are properly nested.
- The **fork** operation is more general: it makes the creation of threads an explicit, executable operation.
- The companion **join** operation, when provided, allows a thread to wait for the completion of a previously forked thread.
- Because fork and join are not tied to nested constructs, they can lead to arbitrary patterns of concurrent control flow
- In addition to providing launch-at-elaboration tasks, Ada allows the programmer to define task *types*:

```
task type T is
```

```
...
begin
...
end T;
```

- The programmer may then declare variables of type access T (pointer to T), and may create new tasks via dynamic allocation:

```
pt : access T := new T;
```

- The **new operation is a fork**; it creates a new thread and starts it executing.
- There is no explicit join operation in Ada, though parent and child tasks can always synchronize with one another explicitly if desired (e.g., immediately before the child completes its execution).
- As with launch-at-elaboration, control will wait automatically at the end of any scope in which task types are declared for all threads using the scope to terminate.
- Any information an Ada task needs in order to do its job must be communicated through shared variables or through explicit messages sent after the task has started execution.
- In Java one obtains a thread by constructing an object of some class derived from a predefined class called **Thread**:

```
class Image extends Thread {
        ...
        Image( args ) {
                // constructor
        }
        public void run() {
                // code to be run by the thread
        }
}
...
Image ob = new Image( constructor args );
```

- In Java, however, the new thread does *not* begin execution when first created.
- To **start** it, the parent (or some other thread) must call the method named start, which is defined in Thread:

```
ob.start( );
```

- Start makes the thread runnable, arranges for it to execute its run method, and returns to the caller.
- The programmer must define an appropriate run method in every class derived from Thread. The run method is meant to be called only by start.
- There is also a **join** method:

```
ob.join( ); // wait for completion
```

- A particularly elegant realization of fork and join appears in the Cilk programming language, developed by researchers at MIT
- To fork a logically concurrent task in Cilk, one simply prepends the keyword spawn to an ordinary function call:

```
spawn foo( args );
```

Implicit Receipt

- We have assumed in all our examples so far that newly created threads will run in the address space of the creator.
- In RPC systems it is often desirable to create a new thread automatically in response to an incoming request from some *other* address space.
- Rather than have an existing thread execute a receive operation, a server can *bind* a communication channel to a local thread body or subroutine.
- When a request comes in, a new thread springs into existence to handle it.
- In effect, the bind operation grants remote clients the ability to perform a fork within the server's address space, though the process is often less than fully automatic.

Early Reply

- We normally think of sequential subroutines in terms of a single thread, which saves its current context, executes the subroutine, and returns to what it was doing before.
- The effect is the same, however, if we have two threads—one that executes the caller and another that executes the callee.
- In this case, the call is essentially a fork/join pair. The caller waits for the callee to terminate before continuing execution.
- In several languages, including SR and Hermes, the callee can execute a reply operation that returns results to the caller *without* terminating.

- After an early reply, the two threads continue concurrently.
- Semantically, the portion of the callee prior to the reply plays much the same role as the constructor of a Java or C# thread; the portion after the reply plays the role of the run method.

2.1 Implementation of Threads

- The threads of a concurrent program are usually implemented on top of one or more *processes* provided by the operating system.
- At one extreme, we could use a separate OS process for every thread; at the other extreme we could multiplex all of a program's threads on top of a single process.
- The problem with putting every thread on a separate process is that processes (even "lightweight" ones) are simply too expensive in many operating systems.
    - Because they are implemented in the kernel, performing any operation on them requires a system call.
    - Because they are general purpose, they provide features that most languages do not need, but have to pay for anyway.
- At the other extreme, there are two problems with putting all threads on top of a single process:
    - first, it precludes parallel execution on a multi core or multiprocessor machine
    - second, if the currently running thread makes a system call that blocks, then none of the program's other threads can run, because the single process is suspended by the OS.
- In the common two-level organization of concurrency (user-level threads on top of kernel-level processes), similar code appears at both levels of the system:
    - the language run-time system implements threads on top of one or more processes in much the same way that the operating system implements processes on top of one or more physical processors.
- To turn coroutines into threads, we proceed in a series of three steps.
    - First, we hide the argument to *transfer* by implementing a *scheduler* that chooses which thread to run next when the current thread yields the processor.
    - Second, we implement a *preemption* mechanism that suspends the current thread automatically on a regular basis, giving other threads a chance to run.

- Third, we allow the data structures that describe our collection of threads to be shared by more than one OS process, possibly on separate processors, so that threads can run on any of the processes.

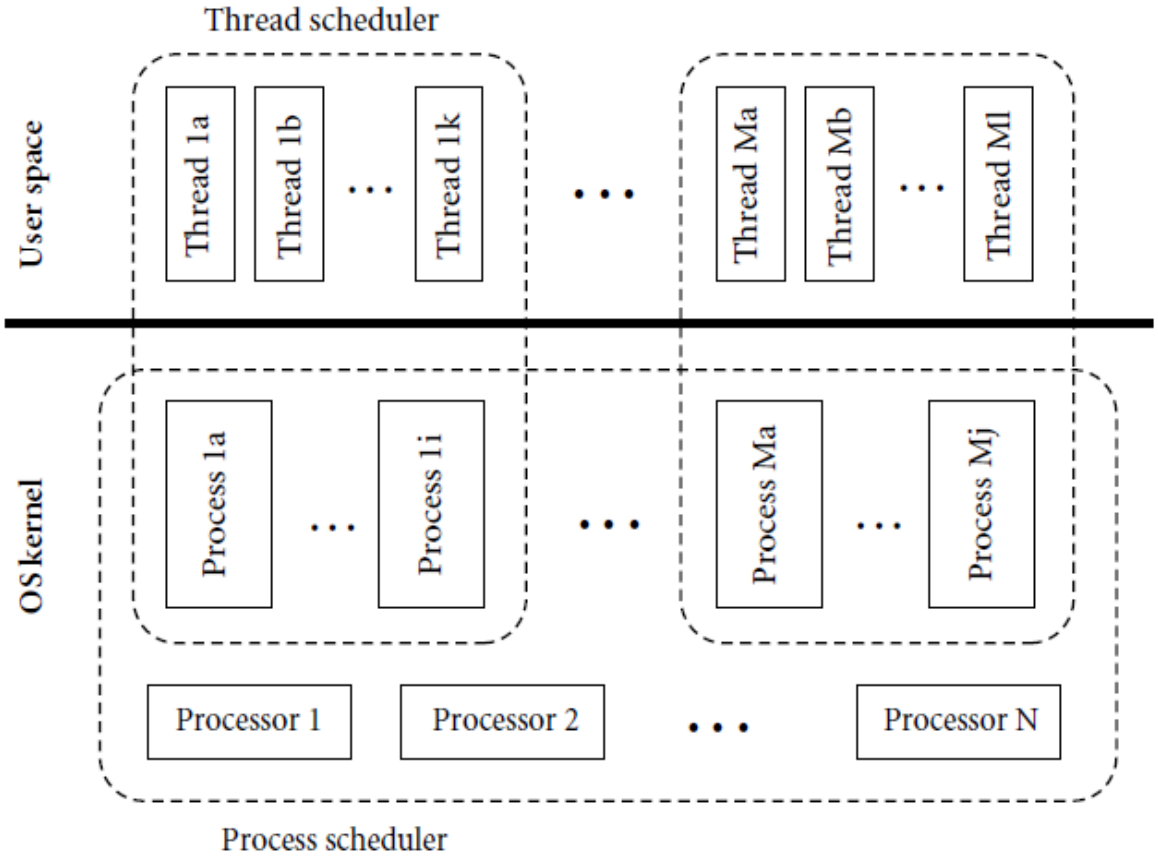

Figure: Two-level implementation of threads

3. Implementing Synchronization

- Synchronization is the principal semantic challenge for shared-memory concurrent programs.
- Typically, synchronization serves either to make some operation *atomic* or to delay that operation until some necessary precondition holds.
- Atomicity is most commonly achieved with *mutual exclusion locks*. Mutual exclusion ensures that only one thread is executing some *critical section* of code at a given point in time.

- Critical sections typically transform a shared data structure from one consistent state to another.
- *Condition synchronization* allows a thread to wait for a precondition, often expressed as a predicate on the value(s) in one or more shared variables.
- In general, our goal is to provide only as much synchronization as is necessary to eliminate bad races—those that might otherwise cause the program to produce incorrect results.

3.1 Busy-Wait Synchronization

- Busy-wait condition synchronization is easy if we can cast a condition in the form of "location *X* contains value *Y* ": a thread that needs to wait for the condition can simply read *X* in a loop, waiting for *Y* to appear.
- To wait for a condition involving more than one location, one needs atomicity to read the locations together, but given that, the implementation is again a simple loop.

Spin Locks

- *spin locks*, which provide mutual exclusion, and *barriers*, which ensure that no thread continues past a given point in a program until all threads have reached that point.

```
type lock = Boolean := false;
procedure acquire lock(ref L : lock)
        while not test and set(L)
                while L
                        – – nothing – – spin
procedure release lock(ref L : lock)
        L := false
```

Example: A simple test-and-test_and_set lock.

- a practical spin lock needs to run in constant time and space, and for this one needs an atomic instruction that does more than load or store.
- The simplest such instruction is known as test_and_set. It sets a Boolean variable to true and returns an indication of whether the variable was previously false.
- Given test_and_set, acquiring a spin lock is almost trivial:

```
while not test and set(L)
        – – nothing – – spin _
```

- Many processors provide atomic instructions more powerful than test_and_ set.
- Several can swap the contents of a register and a memory location atomically. A few can add a constant to a memory location atomically, returning the previous value.
- Several processors, including the x86, the IA-64, and the SPARC, provide a particularly useful instruction called compare_and_swap (CAS).
- This instruction takes three arguments: a location, an expected value, and a new value. It checks to see whether the expected value appears in the specified location, and if so replaces it with the new value, atomically.

Barriers

- Data-parallel algorithms are often structured as a series of high-level steps, or *phases*, typically expressed as iterations of some outermost loop.
- Correctness often depends on making sure that every thread completes the previous step before any moves on to the next.
- A *barrier* serves to provide this synchronization.
- The simplest way to implement a busy-wait barrier is to use a globally shared counter, modified by an atomic fetch_and_decrement instruction.
- The counter begins at *n*, the number of threads in the program. As each thread reaches the barrier it decrements the counter.
- If it is not the last to arrive, the thread then spins on a Boolean flag.
- The final thread (the one that changes the counter from 1 to 0) flips the Boolean flag, allowing the other threads to proceed.
- To make it easy to reuse the barrier data structures in successive iterations (known as barrier *episodes*), threads wait for alternating values of the flag each time through.

3.2 Nonblocking Algorithms

- A thread is said to be “blocked” if it cannot make forward progress without action by other threads.
- Conversely, an operation is said to be *nonblocking* if in every reachable state of the system, any thread executing that operation is guaranteed to complete

in a finite number of steps if it gets to run by itself (without further interference by other threads).

- In this theoretical sense of the word, locks are inherently blocking, regardless of implementation: if one thread holds a lock, no other thread that needs that lock can proceed.
- We can generalize to design special-purpose concurrent data structures that operate without locks.
- Modifications to these structures generally follow the pattern

```
repeat
    prepare
    CAS
until success
clean up
```

- If it reads more than one location, the "prepare" part of the algorithm may need to double-check to make sure that none of the values has changed before moving on to the CAS.
- A read-only operation may simply return once this double-checking is successful.
- Nonblocking algorithms have several advantages over blocking algorithms.
    - They are inherently tolerant of page faults and preemption:
    - They can also safely be used in signal (event) and interrupt handlers,
- They can also be faster than locks.

3.3 Scheduler Implementation

- To implement user-level threads, OS-level processes must synchronize access to the ready list and condition queues, generally by means of spinning.
- In a simple *reentrant* thread scheduler code, we disable timer signals before entering scheduler code, to protect the ready list and condition queues from concurrent access by a process and its own signal handler.
- Our code assumes a single "low-level" lock (scheduler lock) that protects the entire scheduler.
- Before saving its context block on a queue (e.g., in yield or sleep on), a thread must acquire the scheduler lock.
- It must then release the lock after returning from reschedule.
- Of course, because reschedule calls transfer, the lock will usually be acquired by one thread (the same one that disables timer signals) and released by another

- The code for yield can implement synchronization itself, because its work is self-contained.

3.4 Semaphores

- Semaphores are the oldest of the scheduler-based synchronization mechanisms.
- They are still heavily used today, particularly in library-based implementations of concurrency.
- A semaphore is basically a counter with two associated operations, P and V.
- A thread that calls P atomically decrements the counter and then waits until it is non-negative.
- A thread that calls V atomically increments the counter and wakes up a waiting thread, if any.
- It is generally assumed that semaphores are fair, in the sense that threads complete P operations in the same order they start them.
- A semaphore whose counter is initialized to 1 and for which P and V operations always occur in matched pairs is known as a *binary semaphore*.
- It serves as a scheduler-based mutual exclusion lock: the P operation acquires the lock; V releases it.
- More generally, a semaphore whose counter is initialized to k can be used to arbitrate access to k copies of some resource.
- The value of the counter at any particular time is always k more than the difference between the number of P operations ($\#P$) and the number of V operations ($\#V$) that have occurred.
- A P operation blocks the caller until $\#P \leq \#V + k$.

RUN-TIME PROGRAM MANAGEMENT

Ques 8) What do you mean by run time?

Ans: Run Time

Run time is a phase of a computer program in which the program is run or executed on a computer system. Run time is part of the program life cycle, and it describes the time between when the program begins running within the memory until it is terminated or closed by the user or the operating system.

Programming languages can be classified based on the time binding takes place between program variables and their memory locations. If this happens early on, at compile time, and this binding stays in place throughout the execution of the program, then we say that these languages are **static languages**. Two **examples** of such languages are Fortran 77 and Cobol. Fortran 90 has evolved to a point where not all binding takes place at compile time. Hence it cannot be considered a static language.

At the other extreme, there are languages where all bindings take place during run time. In other words, no variable locations are decided when the program is compiled. Such languages are called **dynamic languages**. **Examples** in this category include LISP, ML, Perl.

Some languages (notably C) have very small run-time systems. Most of the user-level code required to execute a given source program is either generated directly by the compiler or contained in language-independent libraries. Other languages have extensive run-time systems. C#, **for example,** is heavily dependent on a run-time system defined by the Common Language Infrastructure (CLI) standard.

Like any run-time system, the **CLI** (Common Language Interface or Common Language Runtime) depends on data generated by the compiler (e.g., type descriptors, lists of exception handlers, and certain content from the symbol table). It also makes extensive assumptions about the structure of compiler-generated code (e.g., parameter-passing conventions, synchronization mechanisms, and the layout of run-time stacks). The coupling between compiler and runtime runs deeper than this, however: the CLI programming interface is so complete as to fully hide the underlying hardware.1 Such a runtime is known as a **virtual machine**. Virtual machines are part of a growing trend toward run-time management and manipulation of programs using compiler technology. To avoid the overhead of emulating a non-native instruction set, many virtual machines use a **just-in-time (JIT)** compiler to translate their instruction set to that of the underlying hardware.

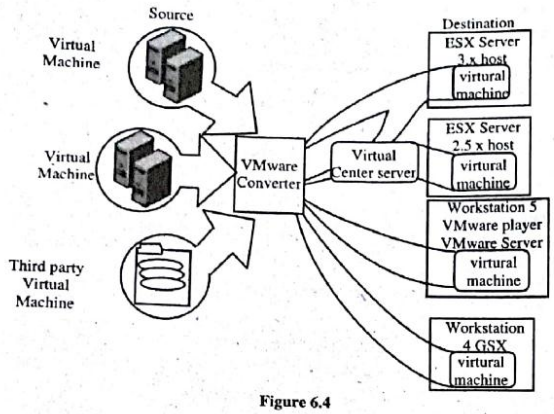

Figure 6.4

Ques 9) Write a note on virtual machines.

Ans: Virtual Machines

A virtual machine (VM) is a software implementation of a machine (i.e. a computer) that executes program like a physical machine. Virtual machines are separated into two major categories, based on their use and degree of correspondence to any real machine:

1) A system virtual machine provides a complete system platform which supports the execution of a complete operating system (OS). These usually emulate an existing architecture, and are built with either the purpose of providing a platform to run programs where the real hardware is not available for use (for example, executing software on otherwise obsolete platforms), or of having multiple instances of virtual machines lead to more efficient use of computing resources.

2) A Process virtual machine (also, language virtual machine) is designed to run a single program, which means that it supports a single process. Such virtual machines are usually closely suited to one or more programming languages and built with the purpose of providing program portability and flexibility (amongst other things). An essential characteristic of a virtual machine is that the software running inside is limited to the resources and abstractions provided by the virtual machine–it cannot break out of its virtual environment. A virtual machine was originally defined by Popek and Goldberg as "an efficient, isolated duplicate of a real machine". Current use includes virtual machines which have no direct correspondence to any real hardware.

Ques 10)What is JVM? Explain the storage management in JVM.

Ans: Java Virtual Machine (JVM)

As we know when a **Java file** compiled the output is not an '.exe' but it's a '.class' file and that '.class' file consists of **Java byte codes** which are understandable by JVM.

Java Virtual Machine interprets the byte code into the machine code depending upon the underlying operating system and hardware combination. It is responsible for all the things like garbage collection, array bounds checking, etc. JVM is **platform dependent.**

The **JVM** is called "virtual" because it provides a machine interface that does not depend on the underlying operating system and machine hardware architecture.

This independence from hardware and operating system is a basis of the write-once run-anywhere (WORA) value of Java programs.

Storage Management in JVM

Figure 6.5 shows a block diagram of the Java virtual machine that includes the major subsystems and memory areas described in the specification.

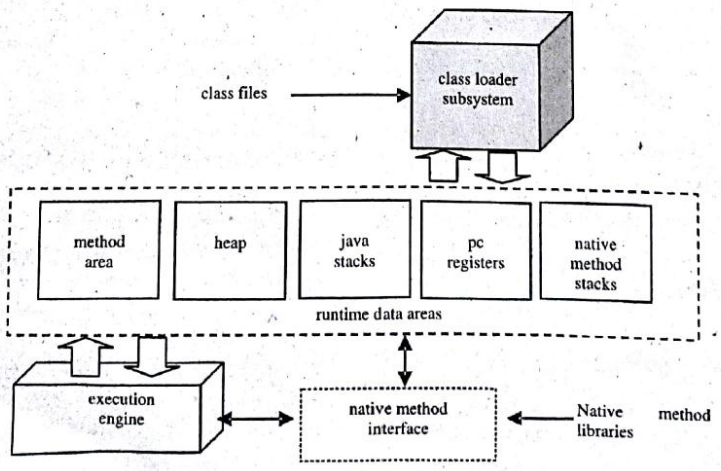

Figure 6.5: Internal Architecture of Java Virtual Machine

Ques 11) Give the elements used in java virtual machine.

Ans: Elements of Java Virtual Machine

1) **Class Loader Subsystem:** it's a mechanism for loading types (classes and interfaces) given fully qualified names.
2) **Method Area:** This area is also shared by all threads running inside the virtual machine. When the virtual machine loads a class file, it parses information about a type from the binary data contained in the class file. It places this type information into the method area
3) **Heap:** Heap is second part of memory and shared by all threads running inside the virtual machine. It contains objects Heap is used for creating objects.
4) **Java Stack:** The Java stack is composed of stack frames (or frames). A stack frame contains the state of one Java method invocation. When a thread invokes a method, the Java virtual machine pushes a new frame onto that thread's Java stack. When the method completes, the virtual machine pops and discards the frame for that method.
5) **PC Register:** Stores the value to indicate the next instruction to execute.
6) **Native Method Stack:** It stores the state of native method invocations in an implementation-dependent way as well as possibly in registers or other implementation-dependent memory areas
7) **Execution Engine:** A mechanism responsible for executing the instructions contained in the methods of loaded classes.

Ques 12) What is a just-in-time (JIT) compiler? Explain in brief.

Ans: JIT

The key of java power "Write once, run everywhere" is bytecode. The way bytecodes get converted to the appropriate native instructions for an application has a huge impact on the speed of an application. These bytecode can be interpreted, compiled to native code or directly executed on a processor whose Instruction Set Architecture is the bytecode specification. Interpreting the bytecode which is the standard implementation of the Java Virtual Machine (JVM) makes execution of programs slow.

To improve performance, JIT compilers interact with the JVM at run time and compile appropriate bytecode sequences into native machine code. When using a JIT compiler, the hardware can execute the native code, as opposed to having the JVM interpret the same sequence of bytecode repeatedly and incurring the penalty of a relatively lengthy translation process. This can lead to performance gains in the execution speed, unless methods are executed less frequently.

The time that a JIT compiler takes to compile the bytecode is added to the overall execution time, and could lead to a higher execution time than an interpreter for executing the bytecode if the methods that are compiled by the JIT are not invoked frequently. The JIT compiler performs certain optimizations when compiling the bytecode to native code.

Since the JIT compiler translates a series of bytecode into native instructions, it can perform some simple optimizations. Some of the common optimizations performed by JIT compilers are data-analysis, translation from stack operations to register operations, reduction of memory accesses by register allocation, elimination of common sub-expressions etc. The higher the degree of optimization done by a JIT compiler, the more time it spends in the execution stage. Therefore a JIT compiler cannot afford to do all the optimizations that is done by a static compiler, both because of the overhead added to the execution time and because it has only a restricted view of the program.

The Just-In-Time (JIT) compiler is a component of the Java Runtime Environment that improves the performance of Java applications at run time. Java programs consist of classes, which contain platform neutral bytecode that can be interpreted by a JVM on many different computer architectures. At run time, the JVM loads the class files, determines the semantics of each individual bytecode, and performs the appropriate computation.

The additional processor and memory usage during interpretation means that a Java application performs more slowly than a native application. The JIT compiler helps improve the performance of Java programs by compiling bytecode into native machine code at run time.

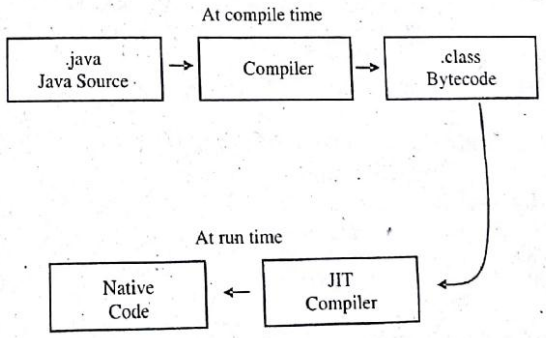

Figure 6.6

The JIT compiler is enabled by default, and is activated when a Java method is called. The JIT compiler compiles the bytecode of that method into native machine code, compiling it "just in time" to run. When a method has been compiled, the JVM calls the

compiled code of that method directly instead of interpreting it. Theoretically, if compilation did not require processor time and memory usage, compiling every method could allow the speed of the Java program to approach that of a native application.

JIT compilation does require processor time and memory usage. When the JVM first starts up, thousands of methods are called.

Compiling all of these methods can significantly affect startup time, even if the program eventually achieves very good peak performance.

Ques 13)Why Common Language Runtime (CLR) is better than Java Virtual Machine (JVM)?

Ans: Comparison between CLR and JVM

Similarities between the CLR and JVM include:

1). Both Virtual Machines (VMs).
2) Both include garbage collection.
3) Both employ stack-based operations.
4) Both include runtime-level security.
5) Both have methods for exception handling.

Differences between the CLR and JVM include:

1) CLR was designed to be language-neutral, JVM was designed to be Java-specific.

2) CLR was originally only Windows-compatible, JVM works with all major OSs.

3) CLR uses a JIT compiler, JVM uses a specialized JIT compiler called Java HotSpot.

4) CLR includes instructions for closures, coroutines and declaration/manipulation of pointers, the JVM does not.

5) JVM is compatible with more robust error resolution and production monitoring tools.

Ques 14)Explain late binding of machine code.

Ans: Late Binding of Machine Code

Late Binding (Dynamic building) is a computer programming mechanism in which the method being called upon an object is looked up by name at runtime. This is informally known as duck typing or name binding.

With Early binding (Static binding) the compilation phase fixes all types of variables and expressions. This is usually stored in the compiled program as an offset in a virtual method table ("v-table") and is very efficient. With late binding the compiler does not have enough information to verify the method even exists, let alone bind to its particular slot on the v-table. Instead the method is looked up by name at runtime.

4. Inspection/Introspection

- Symbol table metadata makes it easy for utility programs-to *inspect* a program and reason about its structure and types.
- Lisp has long allowed a program to reason about its own internal structure and types (this sort of reasoning is sometimes called ***introspection***).
- Java and C# provide similar functionality through a *reflection* API that allows a program to peruse its own metadata.
- Reflection appears in several other languages as well, including Prolog and all the major scripting languages.
- In a dynamically typed language such as Lisp, reflection is essential: it allows a library or application function to type check its own arguments.
- In a statically typed language, reflection supports a variety of programming idioms that were not traditionally feasible.

4.1 Reflection

- Reflection can be useful when printing diagnostics.
- More significantly, reflection is useful in programs that manipulate other programs.
- In a language with reflection, there is no need to examine source code:
- if they load the already-compiled program into their own address space, they can use the reflection API to query the symbol table information created by the compiler.
- Interpreters, debuggers, and profilers can work in a similar fashion.
- In a distributed system, a program can use reflection to create a general-purpose *serialization* mechanism, capable of transforming an almost arbitrary structure into a linear stream of bytes that can be sent over a network and reassembled at the other end.
- There are dangers, associated with the undisciplined use of reflection.
    - Because it allows an application to peek inside the implementation of a class, reflection violates the normal rules of abstraction and information hiding.
    - It may be disabled by some security policies.
    - By limiting the extent to which target code can differ from the source, it may preclude certain forms of code improvement.
- Perhaps the most common pitfall of reflection, at least for object-oriented languages, is the temptation to write case (switch) statements driven by type information.

Java 5 Reflection

- Java's root class, Object, supports a getClass method that returns an instance of java.lang.Class.
- Objects of this class in turn support a large number of reflection operations, among them the getName method used.
- A call to getName returns the *fully qualified* name of the class, as it is embedded in the package hierarchy.
- For array types, naming conventions are taken from the JVM:

```
int[] A = new int[10];
System.out.println(A.getClass().getName()); // prints "[I"
String[] C = new String[10];
System.out.println(C.getClass().getName()); // "[Ljava.lang.String;"
Foo[][] D = new Foo[10][10];
System.out.println(D.getClass().getName()); // "[[LFoo;"
```

- Here Foo is assumed to be a user-defined class in the default (outermost) package.
- A left square bracket indicates an array type; it is followed by the array's element type.
- One can even use reflection to call a method of an object whose class is not known at compile time. Suppose that someone has created a stack containing a single integer:

```
Stack s = new Stack();
s.push(new Integer(3));
```

- Now suppose we are passed this stack as a parameter u of Object type.
- We can use reflection to explore the concrete type of u. In the process we will discover that its second method, named pop, takes no arguments and returns an Object result.
- We can call this method using Method.invoke:

Other Languages

- C#'s reflection API is similar to that of Java:
  - System.Type is analogous to java.lang.Class;
  - System.Reflection is analogous to java.lang.reflect.
  - Pseudofunction **typeof** plays the role of Java's pseudofield .class.
- More substantive differences stem from the fact that PE assemblies contain a bit more information than is found in Java class files.

4.2 Symbolic Debugging

- Symbolic debuggers are built into most programming language interpreters, virtual machines, and integrated program development environments.
- They are also available as stand-alone tools, of which the best known is GNU's gdb.
- ***symbolic*** refers to a debugger's understanding of high-level language syntax-the symbols in the original program.
- In a typical debugging session, the user starts a program under the control of the debugger, or *attaches* the debugger to an already running program. The debugger then allows the user to perform two main kinds of operations.
    - One kind inspects or modifies program data;
    - The other controls execution: starting, stopping, stepping, establishing ***breakpoints*** **and** ***watchpoints*****.**
- A **breakpoint** specifies that execution should stop if it reaches a particular location in the source code.
- A **watchpoint** specifies that execution should stop if a particular variable is read or written.
- Both breakpoints and watchpoints can typically be made *conditional*, so that execution stops only if a particular Boolean predicate evaluates to true.
- Both data and control operations depend critically on symbolic information.
- A symbolic debugger needs to be able both to parse source language expressions and to relate them to symbols in the original program.
- Both data and control operations also depend on the ability to manipulate a program from outside: to stop and start it, and to read and write its data.
- This control can be implemented in at least three ways.
    - The easiest occurs in interpreters. Since an interpreter has direct access to the program's symbol table and is "in the loop" for the execution of every statement.
    - The technology of dynamic binary rewriting can also be used to implement debugger control.
    - The third implementation of debugger control is depends on support from the operating system.
- Perhaps the most mysterious parts of debugging from the user's perspective are the mechanisms used to implement breakpoints, watchpoints, and single stepping.
- Some processors provide hardware support to make breakpoints a bit faster.
- The x86, for example, has four *debugging registers* that can be set (in kernel mode) to contain an instruction address.

- Watchpoints implementation depends on hardware support.

4.3 Performance Analysis

- Before placing a debugged program into production use, one often wants to understand and if possible improve its performance.
- Perhaps the simplest way to measure, at least approximately, the amount of time spent in each part of the code is to *sample* the program counter (PC) periodically.
- This approach was exemplified by the classic prof tool in Unix.
- By linking with a special prof library, a program could arrange to receive a periodic timer signal-once a millisecond, say-in response to which it would increment a counter associated with the current PC.
- While simple, prof had some serious limitations. Its results were only approximate, and could not capture fine-grain costs. It also failed to distinguish among calls to a given routine from multiple locations.
- We can use the more recent gprof tool, which relies on compiler support to instrument procedure prologues.
- If our program is underperforming for algorithmic reasons, it may be enough to know where it is spending the bulk of its time.
- We can focus our attention on improving the source code in the places it will matter most.
- As an example of instrumentation, consider the task of identifying basic blocks that execute an unusually small number of instructions per cycle.
- To find such blocks we can combine
    - (1) the aggregate time spent in each block
    - (2) a count of the number of times each block executes and
    - (3) static knowledge of the number of instructions in each block.
- Most modern processors provide a set of *performance counters* that can be used to good effect by performance analysis tools. The Intel Pentium M processor, for example, has two performance counters that can be configured by the kernel to count any of 47 different kinds of *events*, including branch mispredictions; TLB (address translation) misses; and various kinds of cache misses, interrupts, executed instructions, and pipeline stalls.
- Unfortunately, performance counters are generally a scarce resource: their number, type, and mode of operation varies greatly from processor to processor