Module 6

Concurrency:- Threads, Synchronization. Run-time program Management:- Virtual Machines, Late Binding of Machine Code, Reflection, Symbolic Debugging, Performance Analysis.

Concurrency

A program is said to be *concurrent* if it may have more than one active execution context—more than one "thread of control."
Three important motivations:
1. To capture the logical structure of a problem
2. *To exploit extra processors, for speed*
3.To cope with separate physical devices

We use the word *concurrent* to characterize any system in which two or more tasks may be underway at the same time.
A concurrent system is *parallel* if more than one task can be physically *active* at once; this requires more than one processor.
Concurrent Programming
The term *thread* refers 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.
Task to refer to a well-defined unit of work that must be performed by some thread.
Several languages call their threads processes.
Ada calls them tasks.

Synchronization

In any concurrent programming model, two of the most crucial issues to be addressed are *communication* and *synchronization*.
Communication refers to any mechanism that allows one thread to obtain information produced by another.
Communication mechanisms for imperative programs are generally based on either *shared memory* or *message passing*.
In a shared-memory programming model, some or all of a program's variables are accessible to multiple threads.
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 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.
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.
In blocking synchronization -also called *scheduler-based* synchronization, the waiting thread voluntarily relinquishes its processor to some other thread.
OpenMP comprises a set of *pragmas* (compiler directives) to create and synchronize threads, and to schedule work among them.

<u>Threads</u>

<u>Thread Creation Syntax</u>

Every concurrent system allows threads to be created dynamically.

There are 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 subroutines.

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.

A co-begin construct calls instead for concurrent execution:

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

The compiler checks to make sure that a variable written by one thread is neither read nor written by any concurrently active thread.

In OpenMP, optional “clauses” on parallel directives can specify how many threads to create, and which iterations of the loop to perform in which thread.

A private variable should be *reduced* across all threads at the end of the loop, using a commutative operator.

schedule(static) clause indicates that the compiler should divide the iterations evenly among threads, in contiguous groups.

The default(shared) clause indicates that all variables should be shared by all threads, unless otherwise specified.

reduction(+:sum) clause makes sum an exception: every thread should have its own copy and the copies should be combined (with +) at the end.

In Ada, when the declaration is elaborated, a thread is created to execute the code.

It calls its threads as tasks, we may write,

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

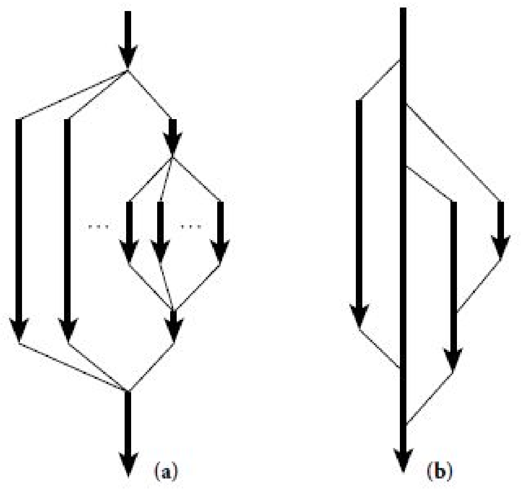


Fig: Lifetime of concurrent threads. With co-begin, parallel loops, or launch-at elaboration (a), threads are always properly nested.
With fork/join (b), more general patterns are possible.
Task T has its own begin. . . end block, which it begins to execute as soon as control enters procedure P.
If P 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 P.

Fork/Join
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.
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.
In Java one obtains a thread by constructing an object of some class derived from a predefined class called Thread:

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

The use of new resembles the creation of dynamic tasks in Ada.
rend.start();
Start makes the thread runnable, arranges for it to execute its run method, and returns to the caller.
There is also a join method:
rend.join(); // wait for completion _
The constructor for a Java thread typically saves its arguments in fields that are later accessed by run.
The class derived from Thread functions as an *object closure*.
Tasks to be accomplished are represented by objects that support the Runnable interface, and these are passed to an Executor object.
The Executor in turn farms them out to a managed pool of threads.

```
class ImageRenderer implements Runnable {
...
}
...
Executor pool = Executors.newFixedThreadPool(4);
...
pool.execute(new ImageRenderer( constructor args ));
```

Each task specified in a call to pool.execute will be run by one of these threads.
To fork a logically concurrent task in Cilk, one simply prepends the keywork spawn to an ordinary function call:
spawn foo(*args*);
At some later time, invocation of the built-in operation sync will join with all tasks previously spawned by the calling task.
Rather than have an existing thread execute a receive operation, a server can *bind* a communication channel to a local thread body or subroutine.
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 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.

Implementation of Threads

The threads of a concurrent program are usually implemented on top of one or more *processes* provided by the operating system.
The problem with putting every thread on a separate process is that processes are simply too expensive in many operating systems.
Because they are implemented in the kernel, performing any operation on them requires a system call.
The typical implementation, starts with coroutines.
Coroutines are a sequential control-flow mechanism: the programmer can suspend the current coroutine and resume a specific alternative by calling the transfer operation.

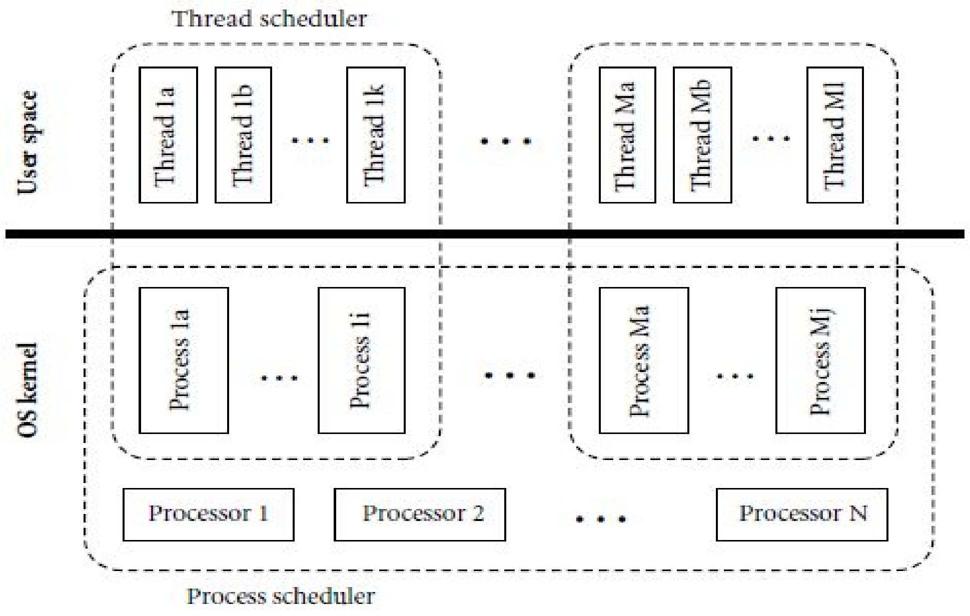


Fig. shows Two-level implementation of threads.

A thread scheduler, implemented in a library or language run-time package, multiplexes threads on top of one or more kernel-level processes.

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.

Uniprocessor Scheduling

At any particular time, a thread is either *blocked* or *runnable*.

A runnable thread may actually be running on some process or it may be awaiting its chance to do so.

Context blocks for threads that are runnable but not currently running reside on a queue called the *ready list.*

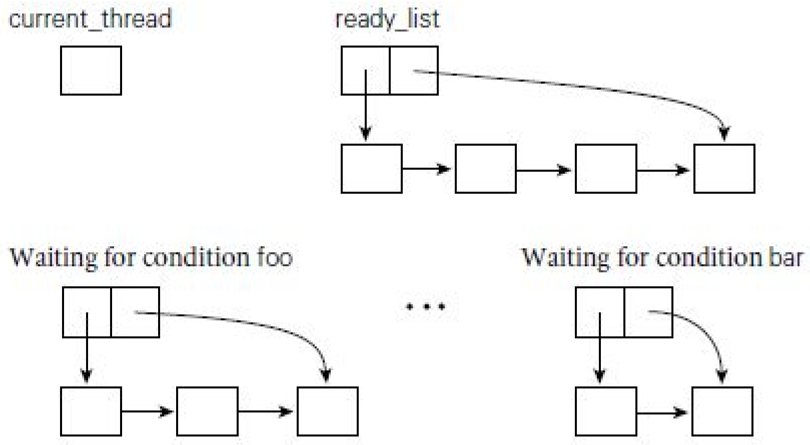


Fig: Data structures of a simple scheduler.

A designated current thread is running.

Threads on the ready list are runnable.

Other threads are blocked, waiting for various conditions to become true.

If threads run on top of more than one OS-level process, each such process will have its own current thread variable.

To yield the processor to another thread, a running thread calls the scheduler:

```
procedure reschedule
t : thread := dequeue(ready list)
transfer(t)
```

If it is blocking for the sake of fairness—to give some other thread a chance to run—then it enqueues its context block on the ready list.

To block for synchronization, a thread adds itself to a queue associated with the awaited condition.

With *cooperative multithreading*, any long-running thread must yield the processor explicitly from time to time (e.g., at the tops of loops), to allow other threads to run.

<u>Preemption:</u> On many systems we can use timer signals for *preemptive multithreading*.

When switching between threads we ask the operating system -which has access to the hardware clock to deliver a signal to the currently running process at a specified time in the future.

The OS delivers the signal by saving the context -registers and pc of the process and transferring control to a previously specified *handler* routine in the language run-time system.

The thread had just executed a call to the standard yield routine, and was about to execute its prologue.

The handler then "returns" into yield, which transfers control to some other thread.

Suppose that a signal arrives when the currently running process has just enqueued the currently running thread onto the ready list in yield, and is about to call reschedule.

When the signal handler "returns" into yield, the process will put the current thread into the ready list a second time.

To resolve the race and avoid corruption of the ready list, thread packages commonly disable signal delivery during scheduler calls:

```
procedure yield
disable signals
enqueue(ready list, current thread)
reschedule
reenable signals
```

To close a *Timing window*—the interval in which a concurrent event may compromise program correctness, the caller must ensure that signals are disabled before checking the condition.

<u>Multiprocessor Scheduling</u>

We can extend preemptive thread package to run on top of more than one OS-provided process by arranging for the processes to share the ready list and related data structures.

If the processes share a single processor, then the program will be able to make forward progress even when all but one of the processes are blocked in the operating system.

Any thread that is runnable is placed in the ready list, where it becomes a candidate for execution by any of the application's processes.

When a process calls reschedule, the queue-based ready list will give it the longest-waiting thread.

To resolve the races, we must implement additional synchronization to make scheduler operations in separate processes atomic.

<u>**Implementing Synchronization**</u>

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.

Implementation of parallel threads, requires both atomicity and condition synchronization.

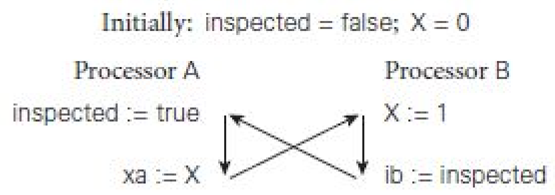
If two processes are racing to dequeue the last thread from the ready list, we *do* care that the implementation of dequeue does not have *internal*, instruction-level races that might compromise the ready list's integrity.
In *Nonblocking synchronization*, atomicity is achieved without the need for mutual exclusion.
<u>Busy-Wait Synchronization</u>
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.
<u>Spin Locks</u>
A simple test-and-test_and_set lock is that, waiting processes spin with ordinary read (load) instructions until the lock appears to be free, then use test_and_set to acquire it.
The very first access is a test_and_set.
Hardware designers began to equip their processors with instructions that read, modify, and write a
memory location as a single atomic operation.
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
```

Overdemand for hardware resources is known as *contention*, and is a major obstacle to good performance on large machines.
On a large machine, contention can be further reduced by implementing a *backoff* strategy, in which a thread that is unsuccessful in attempting to acquire a lock waits for a while before trying again.
Using instructions like atomic_add or compare_and_swap, one can build spin locks that are *fair*, in the sense that threads are guaranteed to acquire the lock in the order in which they first attempt to do so.
An important variant on mutual exclusion is the *reader–writer lock* .
Reader–writer locks recognize that if several threads wish to *read* the same data structure, they can do so simultaneously without mutual interference.
<u>Barriers</u>
A *barrier* serves to provide 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.
<u>Nonblocking Algorithms</u>
Suppose we wish to make an arbitrary update with CAS to a shared location:
x := foo(x);
Note that this update involves at least two accesses to x: one to read the old value and one to write the new.
We can also do this without a lock, using compare_and_swap:

```
start:
r1 := x
r2 := foo(r1) – – probably a multi-instruction sequence
r2 := CAS(x, r1, r2) – – replace x if it hasn't changed
```

if !r2 goto start
This example illustrates that CAS is a *universal* primitive for single-location atomic update.
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.
The CAS-based code of is *nonblocking* : if the CAS operation fails, it is because some other thread has made progress.
Performing cleanup for another thread's operation is often referred to as *helping*.
The dequeue operation does not require cleanup, but the enqueue operation does.
Nonblocking algorithms have several advantages over blocking algorithms.
They are inherently tolerant of page faults and pre-emption.
They can also safely be used in signal (event) and interrupt handlers.
<u>Memory Consistency Models</u>
Most programmers expect a multiprocessor to be *sequentially consistent*—to make all writes visible to all processors in the same order, and to make any given processor's writes visible in the order they were performed.
In *relaxed memory models*, certain loads and stores may appear to occur "out of order".
Stores that are not yet visible in even the L1 cache -or that occurred after a store that is not yet visible are kept in a queue called the *write buffer*.



A's write of inspected precedes its read of X in program order.
B's write of X precedes its read of inspected in program order.
B's read of inspected appears to precede A's write of inspected, because it sees the unset value.
To avoid temporal loops, implementors of concurrent languages and libraries must generally use special *synchronization* or *memory fence* instructions.
Both A and B must prevent their read from *bypassing* -completing before; the logically earlier write.

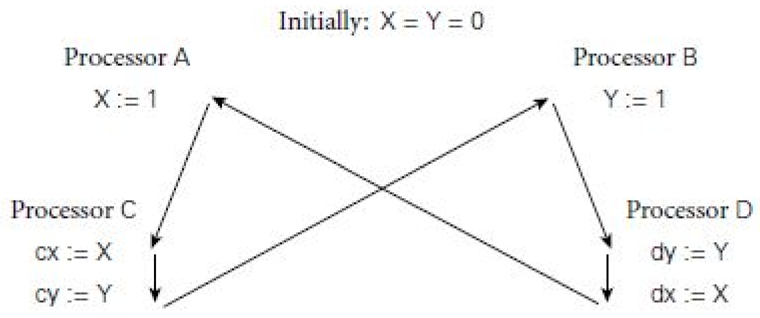


Figure shows the concurrent propagation of writes.
On some machines,it is possible for concurrent writes to reach processors in different orders.
Arrows show apparent temporal ordering.
Here C may read cy = 0 and cx = 1, while D reads dx = 0 and dy = 1.
Multiprocessor memory behavior can in general be described in terms of a transitive *happens before* relationship between instructions.
Every manufacturer ensures that instructions within a given processor appear to occur in program order.
We can write *data-race–free* programs according to some -language-specific *memory model*.

The model specifies which language-level operations are guaranteed to be ordered across threads. A data-race–free program never performs *conflicting* operations unless they are ordered by the model.
Memory consistency models tend to distinguish between *dataraces*-races between ordinary loads and stores) and synchronization races -races between lock operations, volatile loads and stores, or other distinguished operations.
Data races also known as *memory races* are generally considered program bugs;
<u>Scheduler Implementation</u>
To implement user-level threads, OS-level processes must synchronize access to the ready list and condition queues, generally by means of spinning.
In the code for a simple *reentrant* thread scheduler, 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.
A spin-then-yield lock is a busy-wait mechanism: the currently running process relinquishes the processor, but remains on the ready list.
It will perform a test_and_set operation every time every time the lock appears to be free, until it finally succeeds.
Semaphores are the most common form of scheduler-based synchronization.
We use a *bounded buffer* abstraction to illustrate the semantics of various scheduler-based synchronization mechanisms.
A **bounded buffer** is a concurrent queue of limited size into which *producer* threads insert data, and from which *consumer* threads remove data.
The buffer serves to even out fluctuations in the relative rates of progress of the two classes of threads, increasing system throughput.

<u>Semaphores</u>
Semaphores are the oldest of the scheduler-based synchronization mechanisms.
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.
A semaphore-based solution to the bounded buffer problem uses a binary semaphore for mutual exclusion, and two general -or *counting* semaphores for condition synchronization.
<u>Monito</u>r:-
A monitor is a module or object with operations, internal state, and a number of *condition variables*. Only one operation of a given monitor is allowed to be active at a given point in time.
Any operation may suspend itself by *wait*ing on a condition variable.
An operation may also *signal* a condition variable, in which case one of the waiting threads is resumed.
Monitors employs one thread queue for every condition variable, plus two bookkeeping queues: the *entry queue* and the *urgent queue*.
Mesa specifies that signals are only *hints*: the language runtime system moves some waiting thread to the ready list, but the signaler retains control of the monitor, and the waiter must recheck the condition when it awakes.

```
while not desired condition
wait(condition variable)
```

A wait in a nested sequence of monitor operations will release mutual exclusion on the innermost monitor, but will leave the outer monitors locked.
This situation can lead to *deadlock* if the only way for another thread to reach a corresponding signal operation is through the same outer monitors.
A *conditional* critical region also specifies a Boolean condition, which must be true before control will enter the region:

```
region protected variable, when Boolean condition do
. . .
end region
```

Synchronization in Ada 95
Ada 95 augments this mechanism with a notion of *protected object*.
A protected object can have three types of methods: functions, procedures, and *entries*.
Functions can only read the fields of the object; procedures and entries can read and write them.
An entry can have a Boolean expression *guard*, for which the calling task (thread) will wait before beginning execution.
An entry supports three special forms of call: *timed* calls, which abort after waiting for a specified amount of time.
Conditional calls, execute alternative code if the call cannot proceed immediately, and *asynchronous* calls, begin executing alternative code immediately, but abort it if the call is able to proceed before the alternative completes.
Synchronization in Java
In Java, every object accessible to more than one thread has an implicit mutual exclusion lock, acquired and released by means of synchronized statements:

```
synchronized (my_shared_obj) {
... // critical section
}
```

All executions of synchronized statements that refer to the same shared object exclude one another in time.
Java allows a thread to be awoken for spurious reasons; programs must therefore embed the use of wait within a condition-testing loop:

```
while (!condition) {
wait();
}
```

As an alternative to synchronized statements and methods, the programmer may create explicit Lock variables.

```
Lock l = new ReentrantLock();
l.lock();
try {
... // critical section
} finally {
l.unlock();
}
```

Atomicity without Locks
The basic idea of Transactional Memory is very simple: the programmer labels code blocks as atomic.

```
atomic {
– – your code here
```

}
The underlying system takes responsibility for executing these blocks in parallel whenever possible.

<u>Message Passing</u>**:** Shared-memory concurrency has become ubiquitous on multicore processors and multiprocessor servers.
Message passing, however, dominates both distributed and high-end computing.

<u>Run-time program Management</u>

We use the term *run-time system* to refer to the set of libraries on which the language implementation depends for correct operation.
In more complex cases, the compiler generates program-specific *metadata* that the runtime must inspect to do its job.
Some languages eg: C have very small run-time systems: most of the user-level code required to execute a given source program.
This is either generated directly by the compiler or contained in language-independent libraries.
C#, for example, is heavily dependent on a run-time system defined by the Common Language Infrastructure (CLI) standard.
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.
Such a runtime is known as a *virtual machine*.
Many virtual machines use a *just-in-time* (JIT) compiler to translate their instruction set to
that of the underlying hardware.
Some virtual machines—notably the Java Virtual Machine (JVM)—are language-specific.
Microsoft introduced the term *managed code* to refer to programs that run on top of a virtual machine.

<u>Virtual Machines</u>

A *virtual machine* (VM) provides a complete programming environment: its application programming interface (API) includes everything required for correct execution of the programs that run above it.
Every virtual machine API includes an instruction set architecture (ISA) in which to express programs.
This may be the same as the instruction set of some existing physical machine, or it may be an artificial instruction set.
Virtual machines tend to be characterized as either *system* VMs or *process* VMs.
A system VM faithfully emulates all the hardware facilities needed to run a standard OS, including both privileged and unprivileged instructions, memory-mapped I/O etc.
System VMs are sometimes called *virtual machine monitors* (VMMs), because they multiplex a single physical machine among a collection of “guest” operating systems.
They monitor the execution of multiple virtual machines.
VMMs are also increasingly popular on personal computers, where products like Parallels Desktop and VMware Fusion allow users to run programs on top of more than one OS at once.
It is process VMs, that have had the greatest impact on programming language design and implementation.
Process VMs were originally conceived as a way to increase program portability and to quickly “bootstrap” languages on new hardware.

The Java Virtual Machine

A JIT compiler was added in 1998 with the release of Java 2.
Java byte code (JBC) need to be produced from Java source.
The principal requirement, for both compilers and assemblers, is that they generate correct *class files*.

These have a special format understood by the JVM, and must satisfy a variety of structural and semantic constraints.
A JVM is typically given the name of a class file containing the static method main.
It loads this class into memory, verifies that it satisfies a variety of required constraints, allocates any static fields, links it to any preloaded library routines.
It invokes any initialization code provided by the programmer for classes or static fields.
Finally, it calls main in a single thread.
There is a global *constant pool*, a set of registers and a stack for each thread, a *method area* to hold executable byte code, and a heap for dynamically allocated objects.
The constant pool contains both program constants and a variety of symbol table information needed by the JVM and other tools.

```
class Hello {
public static void main(String args[]) {
System.out.println("Hello, world!");
}
};
```

When compiled with Sun's javac compiler, the constant pool for this program has 28 separate entries.
Four of the entries (8, 12, 25, 28) are *type signatures* for methods and fields.
In the format shown here, "V" indicates void; "L*name*;" is a fully qualified class.
A program running on the JVM begins with a single thread.
Each thread has a small set of base registers, a stack of method call frames, and an optional traditional stack on which to call native (non-Java) methods.
Each frame on the method call stack contains an array of local variables, an *operand stack* for evaluation of the method's expressions, and a reference into the constant pool that identifies information needed for dynamic linking of called methods.
Heap:- In keeping with the type system of the Java language, a datum in the local variable array or the operand stack is always either a reference or a value of a built-in scalar type.
Structured data (objects and arrays) must always lie in the heap.
When a thread releases a monitor or writes a volatile variable, the JVM must ensure that all previous updates to the thread's cache have been written back to memory.

<u>Class Files</u>

Physically, a JVM class file is stored as a stream of bytes.
Multiple class files may be combined into a Java archive (.jar) file.
Logically, a class file has a well-defined hierarchical structure.
Class files contain extensive information not typically found in an executable object file.
Examples include access flags for classes, fields, and methods -public, private, protected, static, final, synchronized, native, abstract, strictfp; symbol table information that is built into to the structure of the file etc..

<u>Byte Code</u>

The byte code for a method or for a constructor or a class initializer appears in an entry in the class file's method table.
It is accompanied by:
An indication of the number of local variables, including parameters
The maximum depth required in the operand stack
A table of exception handler information.

Instruction Set

Java byte code was designed to be both simple and compact.
Orthogonality was a strictly secondary concern.
Every instruction begins with a single-byte *opcode.*

Load/store: Move values back and forth between the operand stack and the local variable array.
Arithmetic: Perform integer or floating point operations on values in the operand stack.
Operand stack management: Push and pop; duplicate; swap.
Control transfer: Perform conditional, unconditional, or multiway branches (switch).
Method calls: Call and return from ordinary and static methods -including constructors and initializers of classes and interfaces.
Verification: When it first *loads* a class file, the JVM checks the top-level structure of the file.
Among other things, it verifies that the file begins with the appropriate "magic number," that the specified sizes of the various sections of the file are all within bounds.
These sizes add up to the size of the overall file.
When it *links* the class file into the rest of the program, the JVM checks additional constraints.
These checks require *data flow analysis* to determine that desired properties-initialization status, types of slots in the local stack frame, depth of the operand stack are the same on every possible path to a given point in the program.
The Common Language Infrastructure
The Common Language Infrastructure (CLI) benefitted from experience with Java and the JVM.
Microsoft developed increasingly sophisticated versions of its Component Object Model (COM), first to communicate with, then to call, and finally to share data across program components written in multiple languages.
In .NET, Common Type System (CTS)- provides a superset of what any particular language needs, while requiring common semantics and implementation wherever the type systems of more than one language intersect.
In addition to the CTS, the CLI defines a virtual machine architecture, the VES (Virtual Execution System); an instruction set for that machine, the CIL (Common Intermediate Language); and a portable file format for code and metadata, PE (Portable Executable) assemblies.
Microsoft calls its CLI implementation the Common Language Runtime (CLR); it refers to CIL as Microsoft Intermediate Language (MSIL).
Richer Type System- The Common Type System supports both value and reference variables of structured types -the JVM is limited to references.
Richer Calling Mechanisms- To facilitate the implementation of functional languages, the CLI provides explicit tail-recursive function calls.
Unsafe Code- For the benefit of C, C++, and other non–type-safe languages, the CLI supports explicitly unsafe operations: non converting type casts, dynamic allocation of non–garbage-collected memory etc..
Verifiable code, cannot use these features, and *unverifiable* code, which can use them.
Miscellaneous- For the sake of multiple languages, the CLI supports global data and functions, local variables whose shapes and sizes are not statically known etc..
The CommonType System
The VES and CIL provide instructions to manipulate data of certain built-in types.
A few additional types are predefined, and have built-in names in CLI metadata.
The Common Language Specification (CLS) defines a subset of the CTS intended for cross-language interaction.
Built-inTypes: The VES and CIL provide instructions to manipulate the following types:-
Integers in 8-, 16-, 32-, and 64-bit lengths, both signed and unsigned
"Native" integers, of the length supported by the underlying hardware, both signed and unsigned
IEEE floating point, both single and double precision.
Object references and "managed" pointers.
ConstructedTypes- To the built-in types, the CTS adds the following.
Dynamically allocated instances of class, interface, array, and delegate types
Methods—Function types.
Properties—Getters and setters for objects.

Events—Lists of delegates, associated with an object, that should be called in response to changes to the object.
Typed references—Pointers bundled together with a type descriptor, used for C-style variable argument lists.
Unmanaged pointers—As in C, these can point to just about anything, and support pointer arithmetic etc..
<u>CLS:-</u>The Common Language Specification (CLS) defines a subset of the CTS that most (though not all) languages can accommodate.
It omits several of the types provided by the CTS, including signed 8-bit integers; unsigned native, 16-, 32-, and 64-bit integers etc..
<u>Generics</u>:- Java generics were defined in terms of *type erasure*, which effectively converts all generic types to Object before generating byte code.
C# generics were defined in terms of *reification*, which creates a new concrete type every time a generic is instantiated with different arguments.
<u>Metadata and Assemblies</u>
Portable Executable (PE) *assemblies* are the rough equivalent of Java .jar files: they contain the code for a collection of CLI classes.
PE is based on the Common Object File Format (COFF), originally developed for AT&T's System V Unix.
The metadata of an assembly has a complex internal structure.
The metadata begins with a *manifest* that specifies the files included and directly referenced, the types exported and imported, versioning information etc..
<u>The Common Intermediate Language</u>
Just as the CLI VES bears a strong resemblance to the JVM, CIL bears a strong resemblance to JBC.
Version 4 of the ECMA standard defines approximately 250 instructions, most with single-byte opcodes.
Verification: The CLI distinguishes between *verifiable* and *unverifiable* code.
Verifiable code must satisfy a large variety of constraints that guarantee type safety and catch many common programming errors.
Unverifiable code can make use of unsafe language features e.g., unions and pointer arithmetic in C, but must still conform to more basic rules for validity i.e well formedness of CIL.
<u>Late Binding of Machine Code</u>
Just-in-time (JIT) compilation, which translates a program from source or intermediate form into machine language immediately before each separate run of the program.
We consider *binary translation* and *binary rewriting* systems, which perform compiler-like operations on programs *without* access to source code.
<u>Just-in-Time and Dynamic Compilation</u>
A JIT system compiles programs immediately prior to execution, it can add significant delay to program start-up time.
JIT compilers tend to focus on the simpler forms of target code improvement.
Specifically, they often limit themselves to so-called *local* improvements, which operate within individual control flow constructs.
Improvements at the *global* (whole method) and *interprocedural* (whole program) level are usually too expensive to consider.
Like a lazy linker, a JIT compiler may perform its work incrementally.
It begins by compiling only the class file that contains the program entry point (i.e., main), leaving *hooks* in the code that call into the run-time system wherever the program is supposed to call a method in another class file.
To eliminate the latency of compiling even the original class file, the language implementation may incorporate both an interpreter and a JIT compiler.
Execution begins in the interpreter.

When a class file is JIT compiled, the language implementation can cache the resulting machine code for later use.
JIT compilation affords the opportunity to perform certain kinds of code improvement that are usually not feasible in traditional compilers.

Dynamic Compilation

Compilation *must* be delayed, either because the source or byte code was not created or discovered
until run time, or because we wish to perform optimizations that depend on information gathered during execution.
In these cases we say the language implementation employs *dynamic compilation*.
A dynamic compiler can use statistics gathered by run-time *profiling* to identify *hot paths* through the
code, which it then optimizes in the background.
By rearranging the code to make hot paths contiguous in memory, it may also improve the performance of the instruction cache.
If foo is a static method of class C, then calls to C.foo can safely be in-lined.
Similarly, if bar is a final method of class C -one that cannot be overridden, and o is an object of class
C, then calls to o.bar can safely be in-lined.
A dynamic compiler can perform the optimization if it verifies that there exists no class derived from c anywhere in the -current version of the program.
The compiler might choose to generate code along the following lines:

```
static void f(C o) {
if (o.getClass() == C.class) {
... // code with in-lined calls -- much faster
} else {
... // code without in-lined calls
}
}
```

An Example System: the Hot Spot Java Compiler

HotSpot is Sun’s principal JVM and JIT compiler for desktop and server systems.
It was first released in 1999, and is available as open source.
HotSpot takes its name from its use of dynamic compilation to improve the performance of hot code paths.
Newly loaded class files are initially interpreted.
Methods that are executed frequently are selected by the JVM for compilation and are subsequently patched into the program on the fly.
The Hot Spot compiler can be configured to operate in either “client”or “server”
mode.
Client mode is optimized for lower start-up latency.
Server mode is optimized to generate faster code.

Other Example Systems

Like Hot Spot, Microsoft’s CIL-to-machine-code compiler performs dynamic optimization of hot code paths.
The .NET source-to-CIL compilers are also explicitly available to programs through the System.CodeDom.Compiler API.
Many Lisp dialects and implementations have employed an explicit mix of interpretation and compilation.

Common Lisp includes a compile function that takes the name of an existing function as argument. As a side effect, it invokes the compiler on that function, after which the function will presumably run much faster:

```
(defun square (x) (* x x))            ; outermost level function declaration
(square 3)                           ; 9
(compile ’square)
(square 3)                            ; also 9 (but faster :-)
```

<u>Binary Translation</u>

Just-in-time and dynamic compilers assume the availability of source code or of byte code that retains all of the semantic information of the source.
There are times, however, when it can be useful to recompile object code.
This process is known as *binary translation*.
It allows already-compiled programs to be run on a machine with a different instruction set architecture.
The principal challenge for binary translation is the loss of information in the original source-to-object code translation.
Object code typically lacks both type information and the clearly delineated subroutines and control-flow constructs of source code and byte code.
The typical binary translator reads an object file and reconstructs a control flow graph.
Static binary translation is not always possible for arbitrary object code.
In addition to computed branches, problems include self-modifying code –programs that write to their own instruction space, dynamically generated code e.g., for single-pointer closures, and various forms of introspection.

<u>Where and When toTranslate</u>

By placing the interpreter (emulator) in the lowest levels of the operating system, Apple was able to significantly reduce its time to market.
Binary translators display even more diversity in their choice of what and when to translate.
In the simplest case, translation is a one-time, off-line activity akin to conventional compilation.
Digital Equipment Corp. (DEC) in the early 1990s constructed a pair of translators for their newly developed Alpha processor.
For the sake of transparency -to avoid modification of programs on disk, and to accommodate dynamic linking, translated code is usually not retained from one execution to the next.
Systems in this style include Apple’s Rosetta; HP’s Aries etc..

<u>Dynamic Optimization</u>

In a long-running program, a dynamic translator may revisit hot paths and optimize them more aggressively.
A similar strategy can also be applied to programs that don’t need translation—that is, to programs that already exist as machine code for the underlying architecture.
This sort of *dynamic optimization* has been reported to improve performance by as much as 20% over already-optimized code, by exploiting run-time profiling information.
Much of the technology of dynamic optimization was pioneered by the Dynamo project at HP Labs in the late 1990s.
Dynamo’s key innovation was the concept of a *partial execution trace*: a hot path whose basic blocks can be reorganized, optimized, and cached as a linear sequence.

<u>Binary Rewriting</u>

Binary rewriting is a general technique to modify existing executable programs, typically to insert instrumentation of some kind.
The most common form of instrumentation collects profiling information.

One might count the number of times that each subroutine is called, for example, or the number of
times that each loop iterates.

Binary rewriting can be used to:-
-Simulate new architectures
-Evaluate the coverage of test suites, by identifying paths through the code that are not explored by a series of tests.
-Implement *model checking* for parallel programs, a process that exposes race conditions
-"Audit"the quality of a compiler's optimizations.
ATOM was a static tool that modified a program for subsequent execution.
With the demise of the Alpha processor, use of ATOM has been largely supplanted by Pin, a binary rewriter developed by researchers at Intel in the early 2000s, and distributed as open source.
Pin can even be *attached* to an already-running application, much like the symbolic debuggers.

Mobile Code and Sandboxing

Portability is one of the principal motivations for late binding of machine code.
Code that has been compiled for one machine architecture or operating system cannot generally be run on another.
Code in a byte code (JBC, CIL) or scripting language (JavaScript, Visual Basic), however, is compact and machine independent:
It can easily be moved over the Internet and run on almost any platform.
Such *mobile code* is increasingly common.
Cell phone platforms are using mobile code to distribute games, productivity tools, and interactive media that run on the phones themselves.
In some sense, mobile code is nothing new: almost all our software comes from other sources; we buy it on a DVD or download it over the Internet and install it on our machines.
Mobile code carries a variety of risks.
It may access and reveal confidential information (spyware).
It may interfere with normal use of the computer in annoying ways (adware).
It may damage existing programs or data, or save copies of itself that run without the user's intent (malware of various kinds).
In the worst cases, it may use the host machine as a "zombie" from which to launch attacks on other users.
To protect against unwanted behavior, both accidental and malicious, mobile code must be executed in some sort of *sandbox.*
Sandbox creation is difficult because of the variety of resources that must be protected.
At a minimum, one needs to monitor or limit access to processor cycles, memory outside the code's own instructions and data, the file system, network interfaces, other devices etc..
Sandboxing mechanisms lie at the boundary between language implementation and operating systems.
The two most common techniques are binary rewriting and execution in an untrusting interpreter.

Reflection

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.
Reflection is essential: it allows a library or application function to type check its own arguments.
Reflection can be useful when printing diagnostics.
Suppose we are trying to debug an old-style -nongeneric queue in Java, and we want to trace the objects that move through it.

In the dequeue method, just before returning an object rtn of type Object, we might write

```
System.out.println("Dequeued a " + rtn.getClass().getName());
```

If the dequeued object is a boxed int, we will see

```
Dequeued a java.lang.Integer
```

In a language with reflection, these tools have 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.
The most common pitfall of reflection, at least for object-oriented languages, is the temptation to write case (switch) statements driven by type information:

```
procedure rotate(s : shape)
case shape.type of       – – don’t do this in Java!
square: rotate square(s)
triangle: rotate triangle(s)
circle:              – – no-op
```

<u>Java 5 Reflection</u>
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.
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.
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.
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:

```
Method uMethods[] = u.getClass().getMethods();
Method method1 = uMethods[1];
Object rtn = method1.invoke(u); // u.pop()
```

The principal difference between reflection in Java and C# on the one hand, and in scripting languages on the other, is that the scripting languages—like Lisp—are dynamically typed.
In Ruby, for example, we can discover the class of an object, the methods of a class or object, and the number of parameters expected by each method.
The parameters themselves are untyped until the method is called.

Both Java and C# allow the programmer to extend the metadata saved by the compiler.
In Java, these extensions take the form of *annotations* attached to declarations.
In general, a Java annotation resembles an interface whose methods take no parameters, throw no exceptions, and return values of one of a limited number of predefined types.
Java annotations were inspired at least in part by experience with the earlier javadoc tool, which produces HTML-formatted documentation based on structured comments in Java source code.
JML, the Java Modeling Language, allows the programmer to specify preconditions, post-conditions, and invariants for classes, methods, and statements.
Java 5 introduced a program called apt designed to facilitate the construction of annotation processing tools.
The functionality of this tool was subsequently integrated into Sun's Java 6 compiler.

Symbolic Debugging

Most programmers are familiar with symbolic debuggers: they 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.
The adjective *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.
In gdb, the command print a.b[i] needs to parse the to-be-printed expression; it also needs to recognize that a and i are in scope at the point where the program is currently stopped.
b is an array-typed field whose index range includes the current value of i.
Similarly, the command break 123 if i+j == 3 needs to parse the expression i+j; it also needs to recognize that there is an executable.
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.
The technology of dynamic binary rewriting -as in Dynamo and Pin can also be used to implement debugger control.
In Unix, debugger control mechanism employs a kernel service known as ptrace.
The ptrace kernel call allows a debugger to "grab" (attach to) an existing process or to start a process under its control.
The debugger(tracing process) can specify the address at which it should resume execution, and can ask the kernel to run it for a single instruction -a process known as *single stepping* or until it receives another signal.
Suppose the traced process is currently stopped, and that before resuming it the debugger wishes to set a breakpoint at the beginning of function foo.
It does so by replacing the first instruction of the function's prologue with a special kind of trap.
For a conditional breakpoint, the debugger evaluates the condition's predicate when the breakpoint occurs.

If the breakpoint is unconditional, or if the condition is true, the debugger jumps to its command loop
and waits for user input.
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.
In *single-stepping mode*, it simulates a trap after every user-mode instruction.
Watch points are a bit trickier.
The easiest implementation depends on hardware support.
Suppose we want to drop into the debugger whenever the program modifies some variable x.
The debugging registers of the x86 and other modern processors can be set to simulate a trap whenever the program writes to x's address.
The overhead of repeated context switches between the traced process and the debugger dramatically impacts the performance of software watchpoints: slowdowns of 1000× are not uncommon.
Debuggers based on dynamic binary rewriting have the potential to support arbitrary numbers of watch points.

Performance Analysis

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.
After execution, the prof post-processor would correlate the counters with an address map of the program's code and produce a statistical summary of the percentage of time spent in each subroutine and loop.
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.
If we want to know which of A, B, and C is the biggest contributor to program run time, it is not particularly helpful to learn that all three of them call D where most of the time is actually spent.
If we want to know whether it is A's Ds, B's Ds, or C's Ds that are so expensive, we can use the more recent gprof tool, which relies on compiler support to instrument procedure prologues.
The gprof post-processor assumes that the total time spent in D can accurately be apportioned among the call sites according to the relative number of calls.
More sophisticated tools log not only the caller and callee but also the stack *backtrace* -the contents of the dynamic chain, allowing them to cope with the case in which D consumes twice as much time when called from A as it does when called from B or C.
If the program is underperforming for other reasons, however, we generally need to know *why*.
Is it cache misses due to poor locality, perhaps? Branch mispredictions? Poor pipeline performance?
Tools to address these and similar questions generally rely on more extensive instrumentation of the code or on some sort of hardware support.
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 -obtained by statistical sampling, (2) a count of the number of times each block executes obtained via instrumentation, and (3) static knowledge of the number of instructions in each block.
If basic block *i* contains *ki* instructions and executes *ni* times during a run of a program, it contributes *kini* dynamic instructions to that run.

Let $N = \sum_i k_i n_i$ be the total number of instructions in the run.
If statistical sampling indicates that block *i* accounts for *xi*% of the time in the run and *xi* is significantly larger than (*kini*)/*N*, then probably an unusually large number of cache misses take place.
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.
Performance counters are generally a scarce resource -one might often wish for many more of them.
Their number, type, and mode of operation varies greatly from processor to processor; direct access to them is usually available only in kernel mode.
Operating systems do not always export that access to user-level programs with a convenient or uniform interface.