Module 6

Concurrency

- A Program is concurrent if it has more than one active execution context
  - Have more than 1 thread of control
- A system is concurrent if 2 or more tasks are executed at same time
- If threads are executing in different processes it is said to have parallelism

Communication and Synchronization

- Communication refers to any
- mechanism that allows one thread to obtain information produced by another

- 2 mechanisms
  - shared memory
  - Message Passing

- shared-memory
- A thread can communicate with others by writing to a common variable

- Message Passing
- A thread can communicate with another by an explicit send operation

- Synchronization the relative order in which operations occur in different threads

- Synchronization can be implemented by
  - spinning (also called busy-waiting) or
  - blocking

- Busy-wait or Spinning,
- 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

Blocking synchronization or scheduler based Synchronization

- Waiting thread voluntarily reliquishes its processor to some other thread
- Before doing this, it leaves a note in some data structure associated with Synchronization condition
- A thread that makes the condition true in future will find the note and take the needed steps to make the blocked thread run

Languages and Libraries

- Thread-level concurrency can be provided by
  - explicit concurrent languages
  - Compiler supported extensions
  - Library packages
- All parallel programming use libraries for synchronisation or message passing
- UNIX
- Shared memory parallelism which is obtained using POSIX pthread standards
- Windows
- Win32 thread package provide similar functionality

- Language support for concurrency was provided in ALgol 68 and Ada
- Fortran facilitated parallel execution of loops using OpenMP
- Concurrent language is advantageous than library packages in compiler support

Thread Creation Syntax

- All concurrent system allows threads to be created dynamically
- There are 6 options
  - Co-begin
  - Parallel loops
  - Launch at elaboration
  - Fork
  - Implicit receipt
  - Early reply

Co-begin

- 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
- It must appear immediately before a loop construct or a compound statement delimited with curly braces

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,

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

- The forall loop of High Performance Fortran (HPF) was subsequently incorporated into Fortran 95.
- Supports automatic, internal synchronization on the constituent statements of the loop
  - Reads must occur before any write to the left-hand side, in any iteration.
  - The writes of the left-hand side must occur before any reads in the next assignment statement

- Eg: Fortran

```
forall (i=1:n-1)
A(i) = B(i) + C(i)
A(i+1) = A(i) + A(i+1)
end forall
```

Launch-at-Elaboration

- Declare thread of control as parameterless procedure
- In languages like Ada, code for thread is declared similar to a subroutine without parameters
- When declaration is elaborated, a thread is created to execute the code
- In Ada threads are called tasks(T)

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

- Task T has its own begin. . . end block
- T 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.

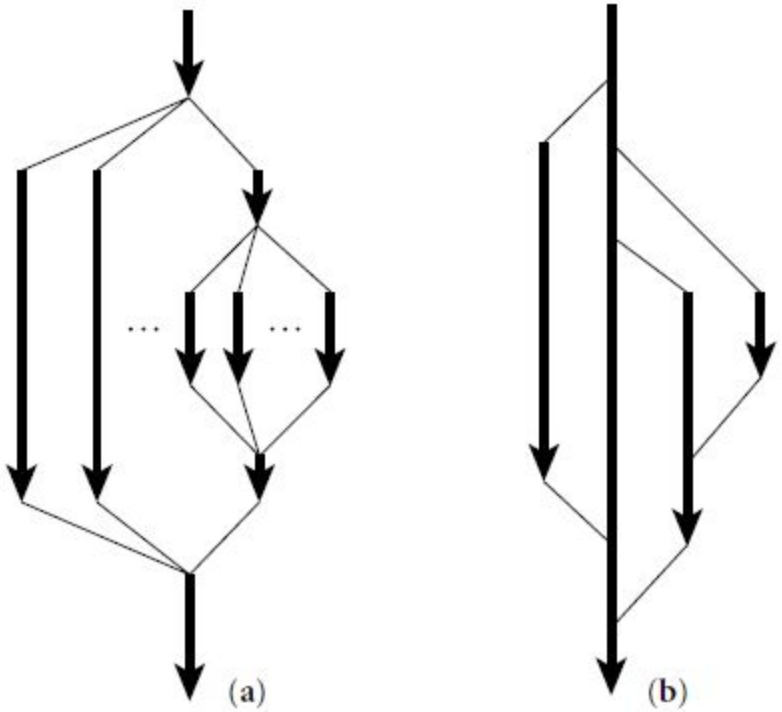


Figure 12.5 Lifetime of concurrent threads. With co-begin, parallel loops, or launch-at-elaboration (a), threads are always properly nested. With fork/join (b), more general patterns are possible.

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
- Parallel loops are data parallel
- Co-begin, and launch-at-elaboration are task parallel

- The fork makes the creation of threads an explicit, executable operation.
- The companion join operation, 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

Java

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

- new thread does not begin execusion when first created
- To start it, call the method named start, which is defined in Thread

```
rend.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 join method

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

Thread creation in C#

- Rather than require every thread to be derived from a common Thread class, C# allows one to be created from an arbitrary ThreadStart delegate:

```
class ImageRenderer {
...
public ImageRenderer( args ) {
// constructor
}
public void Foo() { // Foo is compatible with ThreadStart;
// its name is not significant
// code to be run by the thread
}
}
...
ImageRenderer rendObj = new ImageRenderer( constructor args );
Thread rend = new Thread(new ThreadStart(rendObj.Foo));
```

Spawn and sync in Cilk

- 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.

Implicit Receipt

- Newly created threads will run in address space of creator
- In RPC a new thread is created automatically in response to an incomming request from some other address space
- Rather than have an existing thread execute a receive opration
- 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

Early Reply

- Sequential subroutines are executed in single threads
- It saves its current context, executes subroutine and returns
- If we have 2 threads,
  - One that executes caller
  - Another that executes callee
- Call is done using fork/join pair
- Caller waits for callee to terminate

- Nothing dictates, however, that the callee has to terminate in order to release the caller;
- 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.

Implementation of Threads (Thread Architectures)

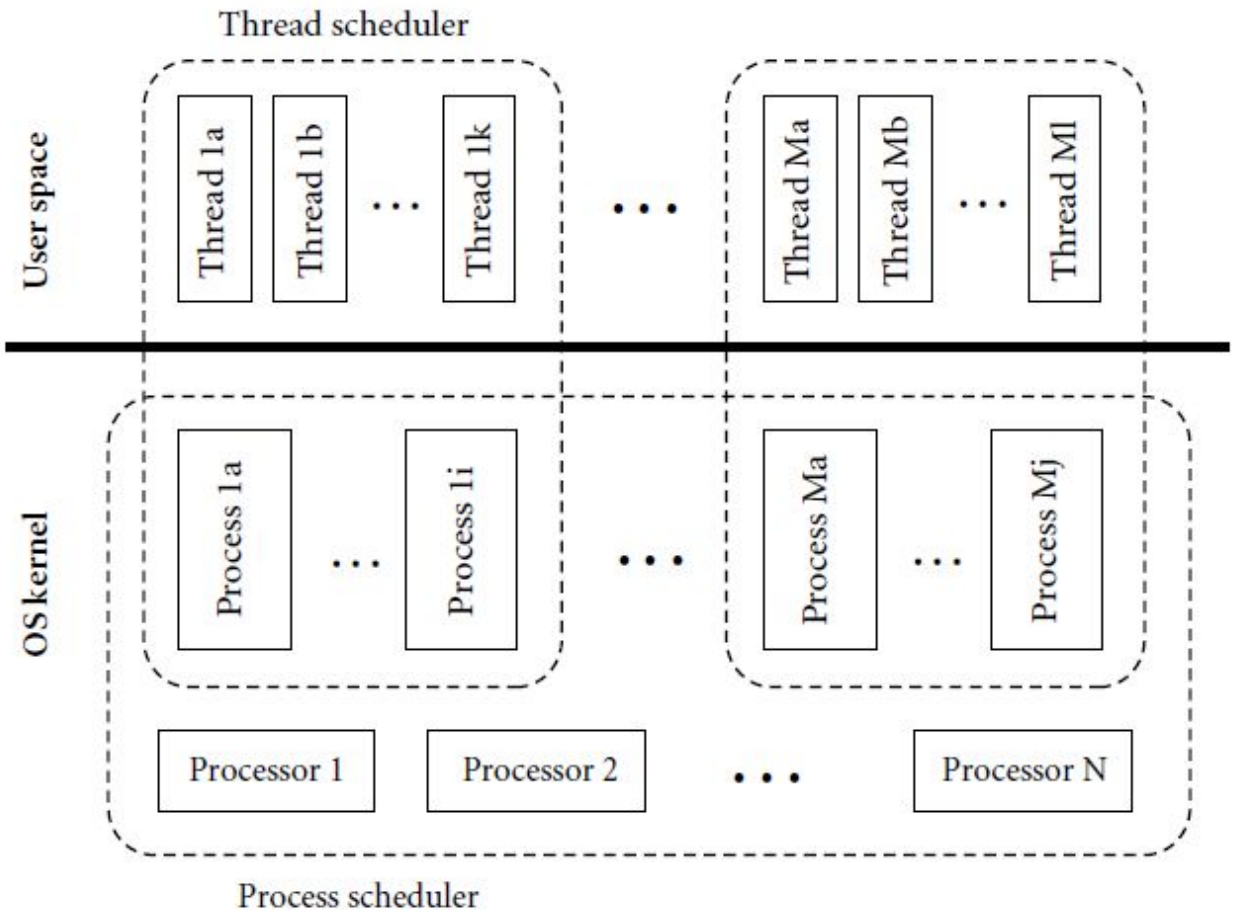
- the threads of a concurrent program are usually implemented on top of one or more processes provided by

  the operating system.
- We can either use
  - a separate OS process for every thread
  - multiplex all of a program's threads on top of a single process

- 1st case is too expensive
  - Sine they are implemented on kernel, any operation requires a system call
- 2nd case preludes parallel execution and if current thread makes a system call that blocks then no other threads run

Another Approach

- Intermediary approach is followed by many languages
- Large number of threads are run on top of small number of processes


Thread scheduler
User space
Thread 1a
Thread 1b
Thread 1k
Thread Ma
Thread Mb
Thread Ml
OS kernel
Process 1a
Process 1i
Process Ma
Process Mj
Processor 1
Processor 2
Processor N
Process scheduler

Threads via Coroutines

- Implemented using coroutines
- Coroutines are sequential control flow mechanism
- 'Programmer can suspend current coroutine and resume specific coroutine by calling transfer operation

- To turn coroutines into thread 3 steps are followed:
- Hide the argument to transfer by implementing a scheduler that chooses which thread to run next when the current thread yields the processor

- implement a preemption mechanism that suspends the current thread automatically on a regular basis, giving other threads a chance to run
- allow the data structures that describe our collection of threads to be shared by more than one OS process.

Uniprocessor Scheduling

- 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 run
- Context blocks for runnable threads reside on a queue called the ready list.
- Context blocks for threads that

  are blocked thread reside conditions for which they are waiting

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

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

- Before calling the scheduler, if a thread that wants to run again at some point in the future it must place its own context block in some appropriate data structure.
- If it is blocking for the sake of fairness(Voluntarily giving control) then it enqueues its context block on the ready list:

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

- To block for synchronization, a thread adds itself to a queue of awaited condition:

```
procedure sleep on(ref Q : queue of thread)
    enqueue(Q, current thread)
    reschedule
```

Preemption

- Multiplexing the processor should be done fairly without requiring that threads call yield explicitly
- This can be achieved by **timer signals** for preemptive multithreading
- When switching between threads,OS delivers a signal to current thread by saving context of process and transfer control to a handler
- The handler calls yield routine that transfers control to some other thread.

- The arrival of signal is arbitrary
- This can introduce race condition between voluntary calls to scheduler and automatic calls triggered by preemption
- To resolve race thread packages disable signal delivery during scheduler calls

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

- sleep_on routine must also assume that signals are disabled and enabled by caller

  disable signals

  if not desired condition

  sleep on(condition queue)

  reenable signals

- On a uniprocessor, disabling signals allows the check and the sleep to occur as a single, atomic operation.

Multiprocessor Scheduling

- Preemptive scheduling can be done by sharing the ready list and related data structures
- If the process run on different processors then more than 1 thread will be able to run at once
- When process calls reschedule, it selects one from ready list
- Here also race condition can appear
- To resolve races additional synchronization needs to be implemented to make scheduler operations in separate processes atomic

Synchronization

- Synchronization either
  - makes some operation atomic or
  - delays that operation until some necessary operation 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.
- 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.

Busy-Wait Synchronization

- It is a technique in which a process repeatedly checks to see if a condition is true

Spin Locks

- This is a lock that causes a thread trying to acquire it to wait in a loop while repeatedly checking if the lock is available
- This is achieved by the means of a special instruction called test and set
- It sets a Boolean variable to true and returns an indication of whether the variable was previously false

TSL Instruction

- test and set instruction used to write 1(set) to a memory location and return its old value as a single atomic
- The calling process obtains the lock if the old value was 0,otherwise the while loop spins waiting to acquire the lock

while not test and set(L)

– – nothing – – spin

Disadvantages

- When a processor P1 has obtained a lock and processor P2 is also waiting for the lock
  - P2 will keep incurring bus transactions in attempts to acquire the lock
- When the processor has obtained a lock, all other processors which also wish to obtain the same lock keep trying to obtain the lock by initiating the bus transactions repeatedly until they get hold of lock

- This increases the bus traffic requirement of test and set
- This slows down all other traffic
- Test and test and set is an improvement over TSL since it donot initiate lock aquisition request continuously

Test-and-test and set

- It spins with ordinary reads until it appears that the lock is free

```
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
```

Backoff

- On a large machine, contention can be further reduced by implementing a backoff
- A thread that is unsuccessful in attempting to acquire a lock waits for a while before trying again

Barriers

- A barrier for a group of threads means any thread must stop at this point and cannot proceed until all other threads reach this barrier Implemented using a shared counter modified by an atomic fetch and decrement instruction
- Counter begin at n
- As each thread reaches a barrier. counter is decremented
- When counter becomes 0, it allows other threads to proceed

```
shared count : integer := n
shared sense : Boolean := true
per-thread private local sense : Boolean := true
procedure central barrier
local sense := not local sense
– – each thread toggles its own sense
if fetch and decrement(count) = 1
– – last arriving thread
count := n – – reinitialize for next iteration
sense := local sense – – allow other threads to proceed
else
repeat
– – spin
until sense = local sense
```

Nonblocking Algorithms

- When a lock is acquired at the beginning of a critical section, no other thread can execute until the lock is released
- x:=foo(x);
- This can be protected by lock

```
acquire(L)
r1 := x
r2 := foo(r1) – – probably a multi-instruction sequence
x := r2
release(L)
```

- But 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
```

- CAS has 3 arguments
  - Memory location where value is to be replaced(X)
  - Old value read(r1)
  - New value to be written(r2)
  - CAS compares if x==r1 and if so r2 is written to x
- After writing it return true if successful and false if failed

Generalised CAS

repeat
prepare
CAS – – (or some other atomic operation)
until success
clean up

- The "prepare" part is harmless if we need to repeat;
- The "cleanup," if needed, can be performed by any thread if the original thread is delayed.
- Performing cleanup for another thread's operation is often referred to as helping

Scheduler implementation

```
shared scheduler lock : low level lock
shared ready list : queue of thread
per-process private current thread : thread
procedure reschedule
– – assume that scheduler lock is already held
– – and that timer signals are disabled
t : thread
loop
t := dequeue(ready list)
if t = null
exit
– – else wait for a thread to become runnable
release lock(scheduler lock)
– – window allows another thread to access ready list
– – (no point in reenabling signals;
– – we're already trying to switch to a different thread)
acquire lock(scheduler lock)
transfer(t)
– – caller must release scheduler lock
– – and reenable timer signals after we return
```

```
procedure yield
disable signals
acquire lock(scheduler lock)
enqueue(ready list, current thread)
reschedule
release lock(scheduler lock)
reenable signals

procedure sleep on(ref Q : queue of thread)
– – assume that caller has already disabled timer signals
– – and acquired scheduler lock, and will reverse
– – these actions when we return
enqueue(Q, current thread)
reschedule
```

Scheduler-Based Synchronization

- The problem with busy-wait synchronization is that it consumes processor cycles,
- cycles that are therefore unavailable for other computation.
- Busy-wait synchronization
- makes sense only if

(1) one has nothing better to do with the current processor, or

(2) the expected wait time is less than the time that would be required to switch contexts to some other thread and then switch back again.

Algorithm

```
type lock = Boolean := false;
procedure acquire lock(ref L : lock)
while not test and set(L)
count :=TIMEOUT
while L
count –:= 1
if count = 0
OS yield – – relinquish processor and drop priority
count :=TIMEOUT
procedure release lock(ref L : lock)
L := false
```

Semaphores

- 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.

- 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:
- P operation acquires the lock;
- V releases it

type semaphore = record
N : integer – – usually initialized to something nonnegative
Q : queue of threads
procedure P(ref S : semaphore)
disable signals
acquire lock(scheduler lock)
S.N –:= 1
if S.N < 0
sleep on(S.Q)
release lock(scheduler lock)
reenable signals
procedure V(ref S : semaphore)
disable signals
acquire lock(scheduler lock)
S.N +:= 1
if N ≤ 0
– – at least one thread is waiting
enqueue(ready list, dequeue(S.Q))
release lock(scheduler lock)
reenable signals

Bounded buffer Problem

shared buf : array [1..SIZE] of bdata
shared next full, next empty : integer := 1, 1
shared mutex : semaphore := 1
shared empty slots, full slots : semaphore := SIZE, 0
procedure insert(d : bdata)
P(empty slots) P increments value wait
P(mutex)
buf[next empty] := d
next empty := next empty mod SIZE + 1
V(mutex)
V(full slots)

function remove : bdata

P(full slots)

P(mutex)

d : bdata := buf[next full]

next full := next full mod SIZE + 1

V(mutex)

V(empty slots)

return d

Run-time Program Management

Late Binding of Machine Code

- Compiler produces a target code in machine language that can be executed several times for different inputs.
- In some environments this translation and execution are combined
  - Just in time compiler

JIT compiler

- Java Byte Codes can be run on different platforms.
- Byte code is interpreted without additional preprocessing
- Cost of JIT compiler can be reduced by existance of an earlier source to byte code compiler
- JIT compilers are faster

Dynamic Compilation

- In some cases JIT compilation must be delayed bze
  - The source or byte code was not created or discovered until run time
  - 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.
  - Code paths that are executed most often
  - Where most of the execution time is spent
- Optimizes this

Binary Translation

- 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.
- Run binary image of old architecture on to a new architecture
- 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.
- Eg: Apple's Rosetta system, which allows programs compiled for older PowerPC-based Macintosh computers to run on newer x86-based Macs

Difficulty

- Principal challenge is the loss of information in the original source to byte code translation
  - Object code typically lacks both

  type information and the clearly delineated subroutines and control-flow constructs of source code and byte code.

- While most of this information appears in the compiler's symbol table, and may sometimes be included in the object file

  for debugging purposes, vendors usually delete it before shipping commercial

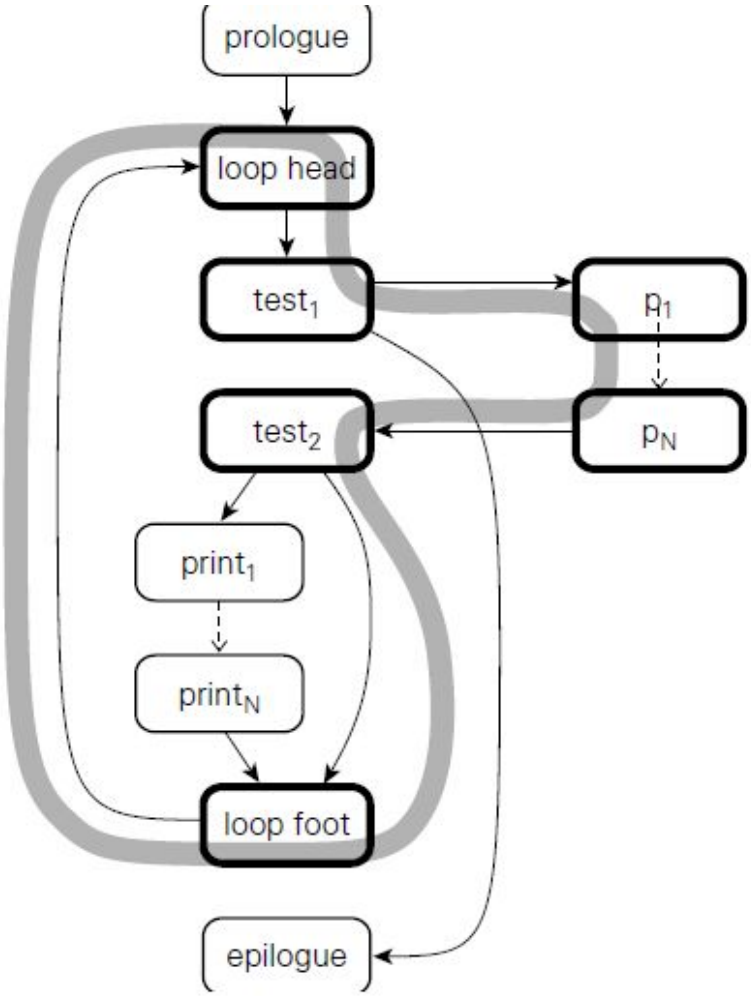
  products, and a binary translator cannot assume it will be present

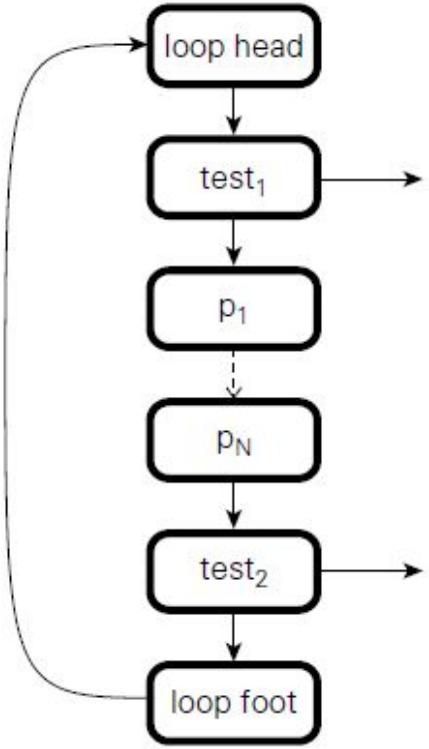
- **The typical binary translator reads an object file and reconstructs a control flow graph**
- This task is complicated by the lack of explicit information about basic blocks.
- While branches (the ends of basic blocks) are easy to identify, beginnings are more difficult
- since branch targets are sometimes computed at run time or looked up in dispatch tables or virtual function tables

Dynamic Optimization

- 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

```
procedure print_matching(S : set, p : predicate)
foreach e in S
if p(e)
print e
```


prologue
loop head
test1
p1
test2
pN
print1
printN
loop foot
epilogue


loop head
test1
p1
pN
test2
loop foot

Binary Rewriting

- *Binary rewriting* is a general technique to modify existing executable programs(Obj Code)
- Evaluate the coverage of test suites, by identifying paths through the code that are not explored by a series of tests.

- It can be used to
- Simulate new architectures
- Implement model checking for parallel programs
  - exposes race conditions by forcing a program through different interleaving
- of operations in different threads.
- “Audit”the quality of a compiler’s optimizations
  - check whether the value loaded into a register is same as before
- Insert dynamic semantic checks into a program that lacks them

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

- Mobile code must be executed in some sort of sandbox
- Sandboxing is a software management strategy that isolates application from critical system resources and other programs
- It provides extra layer of security that prevents malware or harmful applications from negetively affecting your system
- Sandboxing mechanisms lie at the boundary between language implementation
- and operating systems

Reflection

- Lisp allows a program to reason about its own internal structure and types is called introspection
- Java and C# provide similar functionality through a reflection API that allows
  - a program to persue its own metadata
- Useful for printing diagnostics

Java 5 Reflection

- Java's root class is Object
- Object class 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 ,getParameters method
- A call to getName returns the fully qualified name of the class

- int[] A = new int[10];
- System.out.println(A.getClass().getName( )); // prints "[I“
- User-defined types are indicated by an L
- It is followed by the fully qualified class name and terminated by a semicolon
- 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;"

- Object o = new Object();
- System.out.println(o.getClass().getName());
- // "java.lang.Object"

- we can append the pseudofield name .class to the name of the type itself:

- System.out.println(Object.class.getName());
- // "java.lang.Object"

- we can use static method forName of class Class to obtain a Class object for a type with a given (fully qualified) character string name
- Class stringClass = Class.forName("java.lang.String");

- Class intArrayClass = Class.forName("[I");

Reflection In C#

- C# reflection API is similar to Java
- System.type is analogous to Java.lang.Class
- System.Reflection is analogous to java.lang.reflect
- The pseudofunction typeof plays the role of java's pseeudofield.class

```
import static java.lang.System.out;

public static void listMethods(String s)
        throws java.lang.ClassNotFoundException {
    Class c = Class.forName(s);     // throws if class not found
    for (Method m : c.getDeclaredMethods()) {
        out.print(Modifier.toString(m.getModifiers()) + " ");
        out.print(m.getReturnType().getName() + " ");
        out.print(m.getName() + "(");
        boolean first = true;
        for (Class p : m.getParameterTypes()) {
            if (!first) out.print(", ");
            first = false;
            out.print(p.getName());
        }
        out.println(") ");
    }
}
```

Sample output for `listMethods("java.lang.reflect.AccessibleObject")`:

```
public java.lang.annotation.Annotation getAnnotation(java.lang.Class)
public boolean isAnnotationPresent(java.lang.Class)
public [Ljava.lang.annotation.Annotation; getAnnotations()
public [Ljava.lang.annotation.Annotation; getDeclaredAnnotations()
public static void setAccessible([Ljava.lang.reflect.AccessibleObject;, boolean)
public void setAccessible(boolean)
private static void setAccessible0(java.lang.reflect.AccessibleObject, boolean)
public boolean isAccessible()
```

Figure 15.5 Java reflection code to list the methods of a given class. Sample output is shown under code.

Symbolic Debugging

- 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 debugger then allows the user to perform two main kinds of operations.
  - 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

Performance Analysis

- Simplest way to measure the amount of time spent in each part of the code is to sample the program counter periodically
- This approach was supported by classic prof tool in Unix
- By linking with prof library, a program can receive time signals once a millisecond

- After execution the tool provides statistical summary of the percentage of time spent in each subroutine and loop

Limitation of Prof tool

- Results are only approaximate
- Could not capture fine grain costs
- Failed to distinguish among calls to a given routine from multiple locations
- If we want to know if A,B, and C is the biggest contributor to program run time
- If all 3 called D, then we cannot identify if A's D , B's D or C's D was expensive

gprof Tool

- This tool relies on compiler support to instrument procedure prologues
- It logs number of times D is called from each location
- More sophiusticated tools logs not only caller and callee but also stack backtrace
- Most modern processors provide a set of performance counters as performance analysis tools

Run time system

- Run time system or runtime refers to the set of libraries on which the language implementation depends for correct operation

Example

- C has small run time systems
  - Most of the code needed to execute a source program is either compiler generated or contained in language independent libraries
- C# depends heavily on runtime systems

Virtual Machines

- Avirtual 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.

- Characterized as either system VMs or process VMs:
- A system VM emulates all the hardware facilities needed to run a standard OS,
- Also called Virtual Machine monitors (VMM)
  - A single physical machine is multiplexed among different guest OS
- Process VM used to increase portability of programs

The Java Virtual Machine

- JVM interface provides direct support for all built in and reference types defined by Java

Steps

- Initially the name of a class file containing
- the static method main is given to JVM
- Loads this to memory
- Verifies if it satisfies required constants

- Allocates static fields
- Links to preloaded libraries
- invokes any initialization code provided by the programmer
- for classes or static fields
- Finally calls main in a single thread

Storage Management

- Has a constant pool
- set of registers
- stack for each thread,
- method area to hold executable byte code
- heap for dynamically allocated objects.

Global data

- 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 javac compiler, the constant pool for this program has 28 separate entries
- Entry 18 contains the text of the output string;
- Entry 3 indicates that this text is indeed a Java string.
- Many of the additional entries (7, 11, 14, 21–24, 26, 27) give the textual names of files, classes, methods, and fields.
- Others (9, 10, 13) are the names of class , file ,methods
- Four of the entries (8, 12, 25, 28) are type signatures for methods and fields

Per-thread data

- A program running on the JVM begins with a single thread.
- Additional threads are created by using the build-in class Thread, and then calling its start method
- Each thread has
  - set of base registers,
  - a stack of method call frames,
  - optional traditional stack to call non java methods
- Each frame has
  - An array of local variables
  - An operand stack
  - A reference to constant pool

Heap

- Structures data must lie in heap
- They are allocated, dynamically, using the new and newarray instructions
- They are reclaimed automatically via garbage collection
- Each object in heap has mutual exclusion lock
- Locks are acquired with monitorenter instruction and released with the monitorexit instruction

Class Files

- A JVM class file is stored as a stream of bytes.
- Multiple class files may be combined into a Java archive (.jar) file.
- Begins with a "magic number" (0x_cafe_babe)
- This is followed by
  - Major and minor version numbers of the JVM for which the file was created
  - The constant pool
  - Indices into the constant pool for the current class and its superclass
  - Tables describing the class's superinterfaces, fields, and methods

Byte Code

- The byte code for a method appears in an entry in the class file's method table.
- It contains
- Number of local variables
- The maximum depth required in the operand stack
- A table of exception handler information, each entry of which indicates
- – The byte code range covered by this handler
- – The address (index in the code) of the handler itself
- – The type of exception caught (an index into the constant pool)
- Optional information for debuggers

Instruction Set

- Every instruction begins with a single-byte opcode.
- Arguments occupy subsequent bytes
- Types of instructions supported:
- **Load/store**: Move values back and forth between the operand stack and the local
- variable array.
- **Arithmetic**: Performinteger or floating point operations on values in the operand
- stack.
- **Type conversion**: “Widen” or “narrow” values among the built-in types (byte, char, short, int, long, float, and double). Narrowing may result in a loss

of precision but never an exception.

- **Object management**: Create or query the properties of objects and arrays; access fields and array elements.
- **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.
- **Exceptions**: throw (no instructions required for catch).
- **Monitors**: Enter and exit (wait, notify, and notify All are invoked via method calls).

Verification

- JVM checks at load time if references to local variables are within bounds
- Check at load time if top level structure of file
- Checks at link time additional constraints
- Verifies if all entries in constant pool are well formed
- Performs host of checks on byte code of class methods

The Common Language Infrastructure(CLI)

- Provides libraries that allow to interoperate with older programs
- Contains
- VM
- Libraries ,servers and tools for UI management
- Database access
- Security services

Similarities to the JVM

- Both systems define a multithreaded, stack-based VM, with built-in support for garbage collection, exceptions, virtual method dispatch, and mix-in inheritance

- Both represent programs using a platform-independent, self-descriptive, byte code notation

Comparison to the JVM

- **Richer Type System**
  - The Common Type System supports both value and reference variables of structured types
  - JVM is restricted to reference types
  - CLI Supports multidimensional arrays function pointers, generics,structural type equivalence.

- **Richer Calling Mechanisms**
  - To implement functional languages,
  - CLI provides explicit tail-recursive function calls
  - Supports both value and reference parameters, variable numbers of parameters multiple return values, and non virtual methods, all of which the JVM lacks.

Unsafe code

- For the benefit of C, C++, and other non–type-safe languages, the
- CLI supports explicitly unsafe operations: nonconverting type casts, dynamic
- allocation of non–garbage-collected memory, pointers to non-heap data, and pointer arithmetic.

Miscellaneous

- For the sake of multiple languages, the CLI supports global
- data and functions, local variables whose shapes and sizes are not statically
- known, optional detection of arithmetic overflow, and rich facilities for “scoped” security and access control.

- CLI specification describes the following four aspects:

- The Common Type System (CTS)
- A set of data types and operations that are shared by all CTS-compliant programming languages.

- The Metadata
- Information about program structure is language-agnostic, so that it can be referenced between languages and tools, making it easy to work with code written in a language the developer is not using.

- The Common Language Specification (CLS)
- A set of base rules to which any language targeting the CLI should conform in order to interoperate with other CLS-compliant languages. The CLS rules define a subset of the Common Type System.

- The Virtual Execution System (VES)
- The VES loads and executes CLI-compatible programs, using the metadata to combine separately generated pieces of code at runtime.

Common Type System

- Virtual Execution System(VES) defines by CLI VM and CIL provides instructions to manipulate data of certain built in types

Built-in Types

- Integers in 8-, 16-, 32-, and 64-bit lengths, both signed and unsigned
- "Native" integers, of the length supported by the underlying hardware, again
- both signed and unsigned
- IEEE floating point, both single and double precision
- Object references and "managed" pointers

ConstructedTypes

- 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.
- **Value types**—Records (structures), unions, and enumerations

- **Boxed value types**—Values embedded in a dynamically allocated object so that one can create references to them.
- **Function pointers**—References to static functions: type-safe, but without a referencing environment.
- **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. They cannot point to garbage-collectible objects in the heap.

The Common Language Specification

- Defines a subset of CTS that most language can accommodate
- It omits several types provided by CTS
- Establishes naming convensions
- Limits use of overloading
- Defines operators and conversions that programs can assume as supported on built in types

- Requires a lower bound of zero on each dimension of array indexing.
- Prohibits fields and static methods in interfaces.
- Insists that a constructor be called exactly once for each created object
- Each constructor begin with a call to a constructor of its base class.

Generics

- Java generics converts all generic types to Object before generating byte code
- C# generics defined in terms of reification
- Creates a new concrete type every time a generic is instantiated with different arguments

Metadata and Assemblies

- 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)
- PE assemble contains
  - General purpose PE header
  - a special CLI header
  - metadata describing the assembly's types and methods
  - CIL code for the methods.

The Common Intermediate Language

- CIL resembles JBC
- Provides value types, reference parameters,optional overflow checking on arithmetic and make subroutine calls
- It has
  - Non virtual method calls
  - Indirect calls
  - Tail calls which discard callers frame
  - jumps