Introduction

- Chapter 8: Subroutines and Control Abstraction
- Chapter 9: Data Abstraction and Object Orientation
- Chapter 10: Functional Languages
- Chapter 11: Logic Languages
- Chapter 12: Concurrency
- Chapter 13: Scripting Languages
Chapter 8: Subroutines and Control Abstraction

- Review of Stack Layout
- Calling Sequences
- Parameter Passing
- Generic Subroutines and Modules
- Exception Handling
- Coroutines
- Events
Abstraction and Subroutines

• Abstraction: a process by which the programmer can associate a name with a potentially complicated program fragment that can be thought of in terms of its purpose, rather than in terms of its implementation:
  • Control abstraction: performs a well-defined operation.
  • Data abstraction: representation of information.

• Subroutine is a principal mechanism for control abstraction:
  • Mostly parameterized:
    • Actual parameters: arguments passed into a subroutine.
    • Formal parameters: parameters in the subroutine definition.
  • Function: a subroutine that returns a value.
  • Procedure: a subroutine that does not return a value.

• Subroutines are usually declared before being used.
Allocation Strategies

- **Static:**
  - Code.
  - Globals.
  - Own variables.
  - Explicit constants (including strings, sets, other aggregates).
  - Small scalars may be stored in the instructions themselves.

- **Stack:**
  - Parameters.
  - Local variables.
  - Temporaries.
  - Bookkeeping information: return program counter (dynamic link), saved registers, line number, saved display entries, static link.

- **Heap:**
  - Dynamic allocation.
Typical Stack Frame

- Usually grows downward toward lower addresses.
- Arguments are accessed as positive offsets from the frame pointer.
- Local variables and temporaries are accessed at negative offsets from the frame pointer.
- Arguments to be passed to called routines are assembled at the top of the frame using positive offsets from the stack pointer.
Parameter Modes

- Parameter-passing mode and related semantic details are heavily influenced by implementation issues.
- The two most common parameter-passing modes (mostly for languages with a value model of variable):
  - Call-by-value: each actual parameter is assigned into the corresponding formal parameter when a subroutine is called and then the two are independent.
  - Call-by-reference: each formal parameter introduces, within the body of subroutine, a new name for the corresponding actual parameter.
    - Aliases: If the actual parameter is also visible within the subroutine under its original name.
- The distinction between value and reference parameters is fundamentally an implementation issue.
Values and Reference Parameters

- Call-by-value/result:
  - Copies the actual parameters into the corresponding formal parameters at the beginning of subroutine execution.
  - Copies the formal parameters back to the corresponding actual parameters when the subroutine returns.
- Pascal: parameters are passed by value by default.
  - Reference is preceded by the keyword `var`.
- C: always passed by value.
- Fortran: always passed by reference.
Generic Subroutines and Modules

- Performing the same operation for a variety of different objects types.
- Provide an explicitly polymorphic generic facility that allows a collection of similar subroutines or modules (with different types) to be created from a single copy of the source code.
- Generic modules (classes): very useful for creating containers – data abstractions that hold a collection of objects.
- Generic subroutine (methods): needed in generic modules.
- Generic parameter:
  - Java, C#: only types.
  - Ada, C++: more general, including ordinary types, including subroutines and classes.
Closures as Parameters

- A closure is a reference to a subroutine together with its referencing environment.
- It may be passed as a parameter.
- A closure needs to include both a code address and a referencing environment.
- Subroutines are routinely passed as parameters (and returned as results) in functional languages.
- Object closure: in object-oriented language a method is packaged with its environment within an explicit object.
- C# delegates: provide type safety without the restrictions of inheritance.
Exception

• Exception: an unexpected – or at least unusual – condition that arises during program execution, and that cannot easily be handled in the local context:
  • Detected automatically by the language implementation.
  • Program may raise it explicitly.

• The most common exceptions: various run-time errors:
  1. “Invent” a value that can be used by the caller when a real value could not be returned.
  2. Return an explicit “status” value to the caller, who must inspect it after every call (extra parameter in a global variable or encoded as otherwise invalid but patterns of function’s regular return value).
  3. Rely on the caller to pass a closure (if supported) for an error-handling routine the normal routine can call when it runs into trouble.
Coroutines

- Coroutines are execution contexts that exist concurrently, but that execute one at a time, and that transfer control to each other explicitly, by name.

- Coroutines can be used to implement:
  - Iterators (Section 6.5.3).
  - Threads (to be discussed in Chapter 12).

- Coroutines uses transfer operation: saves the current program counter in the current coroutine object and resumes the coroutine specified as a parameter.

- The main body of the program plays the roles of an initial, default coroutine.
Cactus Stack

- Used when two or more corutines are declared in the same nonglobal scope: they must share access to objects in that scope.
- Example: The main stack (MQR) and the coroutines A, B, C and D.
  - Each branch off the stack contains the frames of a separate coroutine.
  - The dynamic chain of a given coroutine ends in the block in which coroutine began execution.
  - The static chain extends down into the remainder of the cactus.
Events

- Event is something to which a running program needs to respond but which occur outside the program, at an unpredictable time (e.g., inputs to GUI).
  - Synchronous input is generally not acceptable.
- An event handler (callback) is invoked (asynchronously) when a given event occurs.
- Then run-time system calls back into the program instead of being called from it.
Chapter 9: Data Abstraction and Object Orientation

- Object-Oriented Programming
- Encapsulation and Inheritance
- Initialization and Finalization
- Dynamic Method Binding
- Multiple Inheritance
- Object-Oriented Programming Revisited
Data Abstraction Development

- We talked about data abstraction some back in the unit on naming and scoping.
- Recall that we traced the historical development of abstraction mechanisms:
  - Static set of variables: Basic.
  - Locals: Fortran.
  - Statics: Fortran, Algol 60, C.
  - Modules: Modula-2, Ada 83.
  - Module types: Euclid.
  - Objects: Smalltalk, C++, Eiffel, Java, Oberon, Modula-3, Ada 95.
- Statics allow a subroutine to retain values from one invocation to the next, while hiding the name in-between.
Modules

- Modules allow a collection of subroutines to share some statics, still with hiding:
  - If you want to build an abstract data type, though, you have to make the module a manager.

- The abstraction provided by modules and module types has at least three important benefits:
  - It reduces conceptual load by minimizing the amount of detail that the programmer must deal with.
  - It provides fault containment:
    - Prevents the programmer to use a program component the wrong way.
    - Limits the portion of a program’s text where the component can be used.
  - It provides a significant degree of independence among program components – difficult to achieve.
Object-Oriented (OO) Programming

• OO Programming can be seen as an attempt to enhance opportunities for code reuse by making it easy to define new abstractions as extensions or refinements of existing abstractions.
• Objects add inheritance and dynamic method binding.
• Simula 67 introduced these, but didn't have data hiding.
• The 3 key factors in OO programming:
  • Encapsulation (data hiding).
  • Inheritance.
  • Dynamic method binding.
• The class contains:
  • Data members (fields).
  • Subroutine members (methods).
Using Classes

• Derived (child, subclass) classes – extend class hierarchy by creating new classes from base (parent, superclass) classes:
  • A single root superclass:
    • Smalltalk, Java: Object.
    • C++: no such class.
  • General-purpose base class: contains only the fields and methods need to implement common operations.

• Modifying base class methods:
  • Redefinition: exposes implementation details.
  • Leave the implementation details to the base class by invoking the method of the parent class:
    • Java, Smalltalk, Objective-C use super.
    • C# uses base.
    • C++ uses :: (why not super?).
    • Eiffel: Explicitly rename methods inherited from a base class.
Encapsulation and Inheritance

• Encapsulation mechanism:
  • Grouping together in one place data and subroutines that operate on them.
  • Hiding irrelevant details from the users of an abstraction.

• OO programming:
  • An extension of the “module-as-type” mechanism.
  • A “module-as-manager” framework.

• Data hiding mechanisms of modules in non-object-oriented languages.

• Data-hiding issues when adding inheritance to make classes.

• Adding inheritances to records: allowing (static) modules to continue to provide data hiding.
Constructors

• The lifetime of an object to be the interval during which it occupies space and can hold data.
• Most object-oriented languages provide some sort of special mechanism to initialize an object automatically at the beginning of its lifetime.
• When written in the form of a subroutine, this mechanism is known as a constructor.
• A constructor does not allocate space.
• A few languages provide a similar destructor mechanism to finalize an object automatically at the end of its lifetime.
Dynamic Method Binding

• Virtual functions in C++ are an example of dynamic method binding: you don't know at compile time what type the object referred to by a variable will be at run time.
• Simula also had virtual functions (all of which are abstract).
• In Smalltalk, Eiffel, Modula-3, and Java all member functions are virtual.
• Note that inheritance does not obviate the need for generics:
  • You might think: hey, I can define an abstract list class and then derive int_list, person_list, etc. from it, but the problem is you won't be able to talk about the elements because you won't know their types.
  • That's what generics are for: abstracting over types.
• Java doesn't have generics, but it does have (checked) dynamic casts.
Member Lookup

- Virtual functions are the only thing that requires any trickiness (Figure).
- They are implemented by creating a dispatch table (vtable) for the class and putting a pointer to that table in the data of the object.
- Objects of a derived class have a different dispatch table.
- In the dispatch table, functions defined in the parent come first, though some of the pointers point to overridden versions.
- You could put the whole dispatch table in the object itself.
- That would save a little time, but potentially waste space.
OO Programming Revisited

• Anthropomorphism is central to the OO paradigm - you think in terms of real-world objects that interact to get things done.

• Many OO languages are strictly sequential, but the model adapts well to parallelism as well.

• Strict interpretation of the term:
  • Uniform data abstraction: everything is an object.
  • Inheritance.
  • Dynamic method binding.

• Lots of conflicting uses of the term out there object-oriented style available in many languages:
  • Data abstraction crucial.
  • Inheritance required by most users of the term object-oriented.
  • Centrality of dynamic method binding a matter of dispute.
Polymorphism

- Dynamic method binding introduces subtype polymorphism into any code that expects a reference to an object of a specific class.
- An object of the derived class that supports the operations of the base class can be also used.
- A combination of inheritance and dynamics methods still does not eliminate the need for generics (see Example 9.45):
  - Needed to avoid tedious type casting.
  - Needed to avoid potentially unsafe code.
- Generics exist for the purpose of abstracting over unrelated types, something that inheritance does not support.
- Eiffel, Java, and C# also provide generics.
- Virtual methods often preclude the in-line expansion of subroutines at compile time.
Chapter 10: Functional Languages

- Historical Origins
- Functional Programming Concepts
- A Review/Overview of Scheme
- Evaluation Order Revisited
- Higher-Order Functions
- Theoretical Foundations
- Functional Programming in Perspective
Church’s Model

• These results led Church to conjecture that any intuitively appealing model of computing would be equally powerful as well: this conjecture is known as Church’s thesis.

• Church’s model of computing is called the lambda calculus:
  • Based on the notion of parameterized expressions with each parameter introduced by an occurrence of the letter $\lambda$ — hence the notation’s name.
  • Lambda calculus was the inspiration for functional programming.
  • One uses it to compute by substituting parameters into expressions, just as one computes in a high level functional program by passing arguments to functions.
Functional Languages

• The design of the functional languages is based on mathematical functions:
  • A solid theoretical basis that is also closer to the user, but relatively unconcerned with the architecture of the machines on which programs will run.
• Functional languages such as Lisp, Scheme, FP, ML, Miranda, and Haskell are an attempt to realize Church's lambda calculus in practical form as a programming language.
• The key idea: do everything by composing functions:
  • No mutable state.
  • No side effects.
Functional Programming Concepts

• Necessary features, many of which are missing in some imperative languages:
  • 1st class and high-order functions.
  • Serious polymorphism.
  • Powerful list facilities.
  • Structured function returns.
  • Fully general aggregates.
  • Garbage collection.

• So how do you get anything done in a functional language?
  • Recursion (especially tail recursion) takes the place of iteration.
  • In general, you can get the effect of a series of assignments
    \[
    x := 0 \\
    x := expr1 \\
    x := expr2 \\
    \]
    from \( f3(f2(f1(0))) \), where each \( f \) expects the value of \( x \) as an argument, \( f1 \) returns \( expr1 \), and \( f2 \) returns \( expr2 \).
Higher-Order Functions

• Even more important than recursion is the notion of higher-order functions.
• Take a function as argument, or return a function as a result.
• Great for building things.
• Why higher-order functions are not more common in imperative programming languages?
  • Depends on the ability to create new functions on the fly: we need a function constructor – a significant departure from the syntax and semantics of traditional imperative languages.
  • The ability to specify functions as return values or to store them in variables requires one of the following:
    • Eliminate function nesting.
    • Give local variables unlimited extent.
Lisp

- The first functional language.
- Lisp also has (these are not necessary present in other functional languages):
  - Homo-iconography: Homogeneity of programs and data – a program in Lisp is itself a list, and can be manipulated with the same mechanisms used to manipulate data.
  - Self-definition: the operational semantics of Lisp can be defined elegantly in terms of an interpreter written in Lisp.
  - Read-evaluate-print: interaction with the user.
- Variants of Lisp:
  - Pure (original) Lisp.
  - Interlisp, MacLisp, Emacs Lisp.
  - Common Lisp.
  - Scheme.
Other Functional Languages

• Pure Lisp is purely functional; all other Lisps have imperative features.
• All early Lisps dynamically scoped:
  • Not clear whether this was deliberate or if it happened by accident.
• Scheme and Common Lisp statically scoped:
  • Common Lisp provides dynamic scope as an option for explicitly-declared special functions.
  • Common Lisp now THE standard Lisp:
    • Very big; complicated (The Ada of functional programming).
• Scheme is a particularly elegant Lisp.
• Other functional languages: ML, Miranda, Haskell, FP.
• Haskell is the leading language for research in functional programming.
Evaluation Order Revisited

- **Applicative order** - evaluate function arguments before passing them to a function:
  - Scheme: functions use applicative order defined with lambda.
  - What is usually done in imperative languages.
  - Usually faster.
  - Scheme uses applicative order in most cases.

- **Normal order** - pass function arguments unevaluated:
  - Scheme: special forms (hygienic macros) use normal order defined with syntax-rules.
  - Arises in the macros and call-by-name parameters of imperative languages.
  - Like call-by-name: don't evaluate argument until you need it.
  - Sometimes faster.
  - Terminates if anything will (Church-Rosser theorem).
Lazy Evaluation

• Lazy evaluation gives the best of both worlds: the advantage of normal-order evaluation while running within a constant factor of the speed of applicative-order evaluation.
• Particularly useful for “infinite” data structures.
• Scheme: available through explicit use of \texttt{delay} and \texttt{force}:
  • \texttt{delay} creates a “promise”.
• But not good in the presence of side effects.
  • If an argument contains a reference to a variable that may be modified by an assignment, then the value of the argument will depend on whether it is evaluated before or after the assignment.
  • If the argument contains an assignment, values elsewhere in the program may depend on when evaluation occurs.
• Scheme requires that every use of \texttt{delay}-ed expression be enclosed in \texttt{force}. 
Perspective: Advantages

- Lack of side effects makes programs easier to understand.
- Lack of explicit evaluation order (in some languages) offers possibility of parallel evaluation (e.g. MultiLisp).
- Lack of side effects and explicit evaluation order simplifies some things for a compiler (provided you don't blow it in other ways).
- Programs are often surprisingly short.
- Language can be extremely small and yet powerful.
Chapter 11: Logic Languages

- Logic Programming Concepts
- Prolog
- Theoretical Foundations
- Logic Programming in Perspective
Logic Programming Concepts

• Logic programming systems allow the programmer to state a collection of axioms from which theorems can be proven.

• Symbolic logic used for the basic needs of formal logic:
  • Express propositions: logical statements that may or may not be true.
  • Express relationships between propositions.
  • Describe how new propositions can be inferred from other propositions.

• Proposition consists of objects and relationships of objects to each other.

• Particular form of symbolic logic used for logic programming is called first-order predicate logic.

• The user of a logic program states a theorem or goal, and the language implementation attempts to find a collection of axioms and inference steps that together imply the goal.
Logic Programming

- Based on predicate calculus.
- Predicates: building blocks $P(a_1, a_2, \ldots, a_K)$, e.g.:
  - $\text{limit}(f, \text{infinity}, 0)$
  - $\text{enrolled}(\text{you}, \text{CS3304})$
  - These are interesting because we attach meaning to them, but within the logical system they are simply structural building blocks, with no meaning beyond that provided by explicitly-stated interrelationships.
- Operators: conjunction, disjunction, negation, implication.
- Universal and existential quantifiers.
- Statements:
  - Sometimes true, sometimes false, often unknown.
  - Axioms: assumed true.
  - Theorems: provably true.
  - Hypotheses (goals): things we'd like to prove true.
Prolog

• Prolog can be thought of declaratively or imperatively:
  • We’ll emphasize the declarative semantics for now, because that's what makes logic programming interesting.
  • We'll get into the imperative semantics later.

• Prolog allows you to state a bunch of axioms:
  • Then you pose a query (goal) and the system tries to find a series of inference steps (and assignments of values to variables) that allow it to prove your query starting from the axioms.

• Example statement:
  
  ```prolog
  mother(mary, fred).
  % you can either think of this as an predicate asserting that
  % mary is the mother of fred - or a data structure (tree)
  % in which the functor (atom) mother is the root,
  % mary is the left child, and fred is the right child
  
  rainy(rochester).
  ```
Resolution and Unification

- Horn clause format:
  \[ H \leftarrow B_1, B_2, \ldots, B_n \]

- Resolution - existing statements are combined, possibly canceling terms, to derive new statements:
  \[ C \leftarrow A, B \]
  \[ D \leftarrow C \]
  \[ D \leftarrow A, B \]

- Unification – matching terms:
  \[ \text{flowery}(X) \leftarrow \text{rainy}(X) \]
  \[ \text{rainy}(\text{Rochester}) \]
  \[ \text{flowery}(\text{Rochester}) \]

- Free variable \((X)\) acquires value \((\text{Rochester})\).
Search/Execution Order

• Bottom-up resolution, forward chaining:
  • Begin with facts and rules of database and attempt to find sequence that leads to goal.
  • Works well with a large set of possibly correct answers.

• Top-down resolution, backward chaining:
  • Begin with goal and attempt to find sequence that leads to set of facts in database.
  • Works well with a small set of possibly correct answers.

• Prolog implementations use backward chaining.

• When goal has more than one subgoal, can use either
  • Depth-first search: find a complete proof for the first subgoal before working on others
  • Breadth-first search: work on all subgoals in parallel
  • Prolog uses depth-first search: can be done with fewer computer resources.
**Imperative Control Flow**

- Cut - a zero-argument predicate ! (exclamation point):
  - Always succeeds.
  - Side effect: commits the interpreter to whatever choices have been made since unifying the parent goal with the left hand side of the current rule.
- Example - list membership:
  - No cut:
    ```prolog
    member(X, [X | _]).
    member(X, [_ | T]) :- member(X, T).
    ```
  - Cut:
    ```prolog
    member(X, [X | _]) :- !.
    member(X, [_ | T]) :- member(X, T).
    ```
  - Alternative:
    ```prolog
    member(X, [X | _]).
    member(X, [H | T]) :- X \= H, member(X, T).
    ```
    - \( X \neq H \) means \( X \) and \( H \) will not unify.
Database Manipulation

- Prolog is homoiconic: it can represent itself.
- It can also modify itself.
  - Add clause with the built-in predicate `assert`.
  - Remove clause with the built-in predicates `retract` and `retractall`.
- `clause` predicate attempts to match its two arguments against the head and body of some existing clause in the database.
- Individual terms can be created, or their contents extracted, using the built-in predicates `functor`, `arg`, and `=..`
  - `functor(T, F, N)` succeeds if and only if `T` is a term with functor `F` and arity `N`.
  - `arg(N, T, A)` succeeds if and only if its first two arguments are instantiated, `N` is a natural number, `T` is a term, and `A` is the `N`th argument of `T`.
- Infix predicate `=..` “equates” a term with a list.
Theoretical Foundations

- In mathematical logic, a *predicate* is a function that maps constants (atoms) or variables to the values true and false.
- *Predicate calculus* provides a notation and inference rules for constructing and reasoning about *propositions* (statements) composed of predicate applications, *operators*, and the *quantifiers* ∀ and ∃.
  - Operators include and (∧), or (∨), not (¬), implication (→), and equivalence (↔).
  - Quantifiers are used to introduce bound variables in an appended proposition, much as λ introduces variables in the lambda calculus.
    - The *universal* quantifier, ∀, indicates that the proposition is true for all values of the variable.
    - The *existential* quantifier, ∃, indicates that the proposition is true for at least one value of the variable.
  - Clausal form provides a unique expression for every preposition.
“Closed World” Assumption

- Closed world assumption: the database is assumed to contain everything that is true.
- When the database does not have information to prove the query, the query is assumed to be false.
- Prolog can prove that a goal is true but it cannot prove that the goal is false.
  - Assumption: if a goal cannot be proven true, it is false.
  - Prolog is a true/fail system, not true/false system.
- The problem of the closed-world assumption is related to the negation problem.
Negation

• A collection of Horn clauses does not include any things assumed to be false: purely “positive” logic.

• \(+\) predicate is different from logical negation – it can succeed simply because our current knowledge is insufficient to prove it.

• Negation in Prolog occurs outside any implicit existential quantifiers on the right-hand side of the rule:
  • \(\neg(takes(X, \text{his201}))\). where \(X\) is uninstantiated means:
    \(\neg \exists X[takes(X, \text{his201})]\) rather than \(\exists X[\neg takes(X, \text{his201})]\)

• A complete characterization of the values of \(X\) for which \(\neg takes(X, \text{his201})\) is true would require a complete exploration of the resolution tree something that Prolog does only when all goals fails or when repeatedly prompted with semicolons.
likes(jake,chocolate).
likes(jake,apricots).
likes(darcie,licorice).
likes(darcie,apricots).

trace.
likes(jake,X), likes(darcie,X).

(1) 1 Call: likes(jake, _0)?
(1) 1 Exit: likes(jake, chocolate)
(2) 1 Call: likes(darcie, chocolate)?
(2) 1 Fail: likes(darcie, chocolate)
(1) 1 Redo: likes(jake, _0)?
(1) 1 Exit: likes(jake, apricots)
(3) 1 Call: likes(darcie, apricots)?
(3) 1 Exit: Likes(darcie, apricots)

X = apricots
Chapter 12: Concurrency

- Background and Motivation
- Concurrent Programming Fundamentals
- Implementing Synchronization
- Language-Level Mechanisms
- Message Passing
Definitions

- Classic von Neumann (stored program) model of computing has single thread of control.
- Parallel programs have more than one thread of control.
- Motivations for concurrency:
  - To capture the logical structure of a problem.
  - To exploit extra processors, for speed.
  - To cope with separate physical devices.
- Concurrent: any system in which two or more tasks may be underway (at an unpredictable point in their execution).
- Concurrent and parallel: more than one task can be physically active at once (more than one processor).
- Concurrent, parallel and distributed: processors are associated with the devices physically separated in the real world.
Process

• A process or thread is a potentially-active execution context.
• Processes/threads can come from:
  • Multiple CPUs.
  • Kernel-level multiplexing of single physical machine.
  • Language or library level multiplexing of kernel-level abstraction.
• They can run:
  • In true parallel.
  • Unpredictably interleaved.
  • Run-until-block.
• Most work focuses on the first two cases, which are equally difficult to deal with.
• A process could be thought of as an abstraction of a physical processor.
Race Conditions

• A race condition occurs when actions in two processes are not synchronized and program behavior depends on the order in which the actions happen.

• Race conditions are not all bad; sometimes any of the possible program outcomes are ok (e.g. workers taking things off a task queue).

• Race conditions (we want to avoid race conditions):
  • Suppose processors A and B share memory, and both try to increment variable X at more or less the same time
  • Very few processors support arithmetic operations on memory, so each processor executes
    
    • If both processors execute these instructions simultaneously X could go up by one or by two.
Synchronization

- Synchronization is the act of ensuring that events in different processes happen in a desired order.
- Synchronization can be used to eliminate race conditions.
- In our example we need to synchronize the increment operations to enforce mutual exclusion on access to $X$.
- Most synchronization can be regarded as one of the following:
  - Mutual exclusion: making sure that only one process is executing a critical section (e.g., touching a variable) at a time.
  - Usually using a mutual exclusion lock (acquire/release).
  - Condition synchronization: making sure that a given process does not proceed until some condition holds (e.g., that a variable contains a given value).
Shared Memory

• To implement synchronization you have to have something that is atomic:
  • That means it happens all at once, as an indivisible action.
  • In most machines, reads and writes of individual memory locations are atomic (note that this is not trivial; memory and/or busses must be designed to arbitrate and serialize concurrent accesses).
  • In early machines, reads and writes of individual memory locations were all that was atomic.

• To simplify the implementation of mutual exclusion, hardware designers began in the late 60's to build so-called read-modify-write, or fetch-and-phi, instructions into their machines.
• Thread: an active entity that the programmer thinks of as running concurrently with other threads.
• Built on top of one or more processes provided by the operating system:
  • Heavyweight process: has its own address space.
  • Lightweight processes: share an address space.
• Task: a well defined unit of work that must be performed by some thread:
  • A collection of threads share a common “bag of tasks”.
• Terminology inconsistent across systems and authors.
Communication and Synchronization

- Communication - any mechanism that allows one thread to obtain information produced by another:
  - Shared memory: program’s variables accessible to multiple threads.
  - Message passing: threads have no common state.

- Synchronization – any mechanism that allows the programmer to control the relative order in which operations occur on different threads.
  - Shared memory: not implicit, requires special constructs.
  - Message passing: implicit.

- Synchronization implementation:
  - Spinning (busy-waiting): a thread runs in a loop reevaluating some condition (makes no sense on uniprocessor).
  - Blocking (scheduler-based): the waiting thread voluntarily relinquishes its processor to some other thread (needed a data structure associated with the synchronization action).
Thread Creation Syntax

- Six principal options:
  - Co-begin.
  - Parallel loops.
  - Launch-at-Elaboration.
  - Fork/Join.
  - Implicit Receipt.
  - Early Reply.

- The first two options delimit thread with special control-flow constructs.
- SR language provides all six options.
- Java, C# and most libraries: fork/join.
- Ada: launch-at-elaboration and fork/join.
- OpenMP: co-being and parallel loops.
- RPC systems: implicit receipt.
Implementation of Threads

- The threads: usually implemented on top of one or more processes provided by the operating system.
- Every thread a separate process:
  - Processes are too expensive.
  - Requires a system call.
  - Provide features are seldom used (e.g., priorities).
- All thread in a single process:
  - Precludes parallel execution on a multicore or multiprocessor machine.
  - If the currently running thread makes a system call that blocks, then none of the program’s other threads can run.
Two-Level Thread Implementation

- User level threads on top of kernel-level processes:
  - Similar code appears at both level of the system:
    - The language run-time system implements threads on top of one or more processes.
    - The operating system implements processes on top of one or more physical processors.
  - The typical implementation starts with coroutines.

- Turning coroutines into threads:
  - Hide the argument to transfer by implementing scheduler.
  - Implement a preemption mechanisms.
  - Allow data structure sharing.
Uniprocessor Scheduling

• A thread is either blocked or runnable:
  • `current_thread`: thread running “on a process”.
  • `ready_list`: a queue for runnable thread.
  • Waiting queues: queues for threads blocked waiting for conditions.
  • Fairness: each thread gets a frequent “slice” of the processor.

• Cooperative multithreading: any long-running thread must yield the processor explicitly from time to time.

• Schedulers: ability to "put a thread/process to sleep" and run something else:
  • Start with coroutines.
  • Make uniprocessor run-until-block threads.
  • Add preemption.
  • Add multiple processors.
Multiprocessors Scheduling

- True or quasi parallelism introduces race between calls in separate OS processes.
- Additional synchronization needed to make scheduler operations in separate processes atomic:

```plaintext
procedure yield:
    disable_signals
    acquire(scheduler_lock)       // spin lock
    enqueue(ready_list, current)
    reschedule
    release(scheduler_lock)
    re-enable_signals

    disable_signals
    acquire(scheduler_lock)       // spin lock
    if not <desired condition>
        sleep_on <condition queue>
    release(scheduler_lock)
    re-enable signals
```
Implementing Synchronization

- Typically, synchronization is used to:
  - Make some operation atomic.
  - Delay that operation until some necessary precondition holds.
- Atomicity: usually achieved with mutual exclusion locks.
  - Mutual exclusion ensures that only one thread is executing some critical section of code at given point in time:
  - Much early research was devoted to figuring out how to build it from simple atomic reads and writes.
  - Dekker is generally credited with finding the first correct solution for two threads in the early 1960s.
  - Dijkstra: a version that works for n threads in 1965.
- Condition synchronization: allows a thread to wait for a precondition: e.g. a predicate on the value(s) in one or more shared variables.
Semaphores

- A semaphore is a special counter:
  - Has an initial value and two operations, P and V, for changing value.
  - A semaphore keeps track of the difference between the number of P and V operations that have occurred.
  - A P operation is delayed (the process is de-scheduled) until \( #P - #V \leq C \), the initial value of the semaphore.
- The semaphores are generally fair, i.e., the processes complete P operations in the same order they start them.
- Problems with semaphores:
  - They're pretty low-level:
    - When using them for mutual exclusion, it's easy to forget a P or a V, especially when they don't occur in strictly matched pairs.
  - Their use is scattered all over the place:
    - If you want to change how processes synchronize access to a data structure, you have to find all the places in the code where they touch that structure, which is difficult and error-prone.
Monitors

• Suggested by Dijkstra as a solution to the problems of semaphores (languages Concurrent Pascal, Modula, Mesa).

• 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 (programmers are relieved of the responsibility of using $P$ and $V$ operations correctly).
  • A thread that calls a busy monitor is automatically delayed until the monitor is free.
  • An operation can suspend itself by waiting on a condition variable (not the same as semaphores – no memory).
  • All operations on the encapsulated data, including synchronization, are collected together.

• Monitors have the highest-level semantics, but a few sticky semantic problem - they are also widely used.
Conditional Critical Regions

- Proposed as an alternative to semaphores by Brinch Hansen.
- Critical region - a syntactically delimited critical section in which the code is permitted to access a protected variable:
  - Specifies a Boolean condition that must be true before control enters:
    ```
    region protected_variable, when Boolean_condition do
      ...
    end region
    ```
  - No thread can access the protected variable except within a region statement.
  - Any thread that reaches a region statement waits until the condition is true and no other is currently in a region for the same variable.
  - Nesting regions: a deadlock is possible.
- Languages – Edison:
  - Influenced synchronization mechanism of Ada 95, Java, and C#.
Message Passing

• Most concurrent programming on large multicomputers and networks is currently based on messages.

• To send/receive a message, one must generally specify where to send it to, or where to receive it from: communication partners need names for one another:
  • Addressing messages to processes: Hoare’s CSP (Communicating Sequential Processes).
  • Addressing messages to ports: Ada.
  • Addressing messages to channels: Occam.

• Ada’s comparatively high-level semantics for parameter modes allows the same set of modes to be used for both subroutines and entries (rendezvous).

• Some concurrent languages provide parameter modes specifically designed with remote invocation in mind.
Transactional Memory

• Locks (semaphors, monitors, CCRs) make it easy to write data-race free programs but they do not scale:
  • Adding processors and threads: the lock becomes a bottleneck.
  • We can partition program data into equivalence classes: a critical section must acquire lock for every accessed equivalence class.
  • Different critical sections may locks in different orders: deadlock can result.
  • Enforcing a common order can be difficult.
• Locks may be too low level a mechanism.
• The mapping between locks and critical sections is an implementation detail from a semantic point of view:
  • We really want is a composable atomic construct: transactional memory (TM).
Chapter 13: Scripting Languages

• What is a Scripting Language?
• Problem Domains
• Scripting the World Wide Web
• Innovative Features
Scripting Language

- Modern scripting languages have two principal sets of ancestors:
  - Command interpreters or “shells” of traditional batch and “terminal” (command-line) computing:
    - IBM’s JCL, MS-DOS command interpreter, Unix sh and csh.
  - Various tools for text processing and report generation
    - IBM’s RPG, and Unix’s sed and awk.
- From these evolved:
  - Perl: originally devised by Larry Wall in the late 1980s, and now the most widely used general purpose scripting language.
  - Other general purpose scripting languages include Tcl (“tickle”), Python, Ruby, VBScript (for Windows) and AppleScript (for the Mac).
Common Characteristics

• Both batch and interactive use.
• Economy of expression: avoid the extensive declarations and top-level structure.
• Lack of declarations; simple scoping rules.
• Flexible dynamic typing.
• Easy access to system facilities (other programs).
• Sophisticated pattern matching and string manipulation: usually extended regular expressions.
• High level data types: frequently built into the syntax and semantics of the language itself.
Shell (Command) Languages

- They have features designed for interactive use.
- Provide mechanisms to manipulate file names, arguments, and commands, and to glue together other programs:
  - Most of these features are retained by more general scripting languages.
  - We use `bash` Unix shell to illustrate these features.
  - There is also `csh` family of shells.
- We consider a few of them - full details can be found in the `bash` man page, or in various on-line tutorials:
  - Filename and Variable Expansion.
  - Tests, Queries, and Conditions.
  - Pipes and Redirection.
  - Quoting and Expansion.
  - Functions.
  - The `#!` Convention.
Text Processing / Report Generation

- Shell languages tend to be heavily string-oriented.
  - Commands are strings parsed into lists of words.
  - Variables are string-valued.
  - Not intended for editor-like text operations (e.g., *emacs* or *vi*).
- Tools needed to provide for search, substitution, etc.:
  - The second principal class of ancestors for modern scripting languages.
  - Some representative tools:
    - *sed*
    - *awk*
    - *Perl*
Mathematics and Statistics

- A one-line mathematics and statistics computation

- APL - A Programming Language:
  - Interactive, matrix oriented.
  - Concise expression of mathematical algorithms.
  - Code structured as a sequence of unary/binary operators/functions acting on matrices/arrays.
  - A large number of special characters for operators: \( x \left[ \uparrow x \leftarrow 6 \, ? \, 4 \, 0 \right] \)

- Modern successors:
  - Mathematical computing: Maple, Mathematica, and Matlab.
  - Statistical computing: S and R.
“Glue” Languages / General Purpose Scripting

- Scripting languages - shell- and text-processing mechanisms:
  - Can prepare input and parse output from processes.
- An extensive library of built-in operations to access the features of underlying OS.
- Rich set of features for internal computation:
  - Arbitrary precision arithmetic (Python, Ruby).
  - Higher-level types.
  - Modules and dynamic loading (Perl, Tcl, Python, Ruby).
- The philosophy of general-purpose scripting is to make it as easy as possible to construct the overall framework of a program:
  - External tools are used only for special-purpose tasks.
  - Compiled languages only when performance is at a premium.
Extension Languages

• Most applications accept some sort of commands:
  • These commands are entered textually or triggered by user interface events such as mouse clicks, menu selections, and keystrokes.
  • Commands in a graphical drawing program might save or load a drawing; select, insert, delete, or modify its parts; choose a line style, weight, or color; zoom/rotate the display; or modify user preferences.

• An extension language serves to increase the usefulness of an application by allowing the user to create new commands, generally using the existing commands as primitives.

• Increasingly seen as an essential feature:
  • Adobe’s graphics suite (Illustrator, etc.) can be extended (scripted) using JavaScript, Visual Basic (on Windows), or AppleScript.
  • AOLserver, an open-source web server from America On-Line, can be scripted using Tcl. Disney and Industrial Light and Magic use Python to extend their internal (proprietary) tools.
World Wide Web

- Dynamically created World Wide Web content:
  - Does the script that creates the content run on the server or the client machine?
- Server-side and client-side web scripting.
  - Server side scripting: used when the service provided wants to retain complete control over the content of the page but does not create the content in advance (e.g., search engines, Internet retailers).
  - Client-side scripts are typically used for tasks that don’t need access to proprietary information, and are more efficient if executed on the client’s machine (e.g., interactive animation, error-checking, fill-in forms).
CGI Scripts

• The original mechanism for server-side web scripting is the Common Gateway Interface (CGI).
• A CGI script is an executable program residing in a special directory known to the web server program.
• When a client requests the URI corresponding to such a program, the server executes the program and sends its output back to the client:
  • This output needs to be something that the browser will understand: typically HTML.
• CGI scripts may be written in any language available:
  • Perl is particularly popular:
    • Its string-handling and “glue” mechanisms are suited to generating HTML.
    • It was already widely available during the early years of the web.
Embedded Server-Side Scripts

• Though widely used, CGI scripts have several disadvantages:
  • The web server must launch each script as a separate program, with potentially significant overhead (though, CGI script compiled to native code can be very fast once running).
  • Scripts must generally be installed in a trusted directory by trusted system administrators (they cannot reside in arbitrary locations as ordinary pages do).
  • The name of the script appears in the URI, typically prefixed with the name of the trusted directory, so static and dynamic pages look different to end users.
  • Each script must generate not only dynamic content, but also the HTML tags that are needed to format and display it (his extra “boilerplate” makes scripts more difficult to write).
  • Most web servers now use a “module loading” mechanism that allows interpreters for one or more scripting languages.
Client Side Scripts

• Embedded server-side scripts are generally faster than CGI script, at least when startup cost predominates:
  • Communication across the Internet is still too slow for interactive pages.

• Because they run on the web designer’s site, CGI scripts and, to a lesser extent, embeddable server-side scripts can be written in many different languages:
  • All the client ever sees is standard HTML.

• Client-side scripts, by contrast, require an interpreter on the client’s machine:
  • There is a powerful incentive for convergence in client-side scripting languages: most designers want their pages to be viewable by as wide an audience as possible.
While Visual Basic is widely used within specific organizations - all the clients of interest are known to run Internet Explorer.

Pages intended for the general public almost always use JavaScript for interactive features:

- Developed by Netscape in the mid 1990s.
- All major browser implement JavaScript.
- Standardized by ECMA (the European standards body) in 1999.

The HTML Document Object Model (DOM) standardized by the World Wide Web Consortium specifies a very large number of elements, attributes, and user actions, all of which are accessible in JavaScript:

- Scripts can, at appropriate times, inspect or later almost any aspect of the content, structure, or style of a page.
Java Applets

• An applet is a program designed to run inside some other program.
• The term is most often used for Java programs that display their output in (a portion of) a web page:
  • Does not produce HTML output.
  • Directly controls a portion of the page.
  • Java GUI libraries (Swing or AWT) are used to display information.
• To support the execution of applets, most modern browsers contain a Java virtual machine.
• Subject to certain restrictions (security).
• Mostly do not interact with the browser or other programs so they generally not considered a scripting mechanism.
XSLT

- XML (extensible markup language) is a more recent and general language in which to capture structured data:
  - More regular and consistent syntax and semantics (compared to HTML).
  - Extensibility: users can define their own tags.
  - Clear distinction between the content of a document (the data it captures) and the presentation of that data.
  - Presentation is deferred to a companion standard known as XSL (extensible stylesheet language).
- XSLT is a portion of XSL devoted to transforming XML:
  - Selecting, reorganizing, and modifying tags and the elements they delimit.
  - Scripting the processing of data represented in XML.
Features of Scripting Languages

1. Both batch and interactive use.
2. Economy of expression.
3. Lack of declarations; simple scoping rules.
4. Flexible dynamic typing.
5. Easy access to other programs.
6. Sophisticated pattern matching and string manipulation.
7. High level data types.
Object Orientation

• Perl 5 has features that allow one to program in an object-oriented style.
• PHP and JavaScript have cleaner, more conventional-looking object-oriented features:
  • Both allow the programmer to use a more traditional imperative style.
• Python and Ruby are explicitly and uniformly object-oriented.
• Perl uses a value model for variables; objects are always accessed via pointers.
• In PHP and JavaScript, a variable can hold either a value of a primitive type or a reference to an object of composite type:
  • In contrast to Perl, however, these languages provide no way to speak of the reference itself, only the object to which it refers.
Summary

• This lectures provide overview of the Chapters 8-13.
• The material covered presents most, but not all the topics from Chapters 8-13 that will be covered in the final exam.
• Chapters 8-13 related material will constitute 80% of the final exam.