Chapter 8 – Concurrency

Introduction to Programming Languages

First Edition, 2013

Author: Arvind Bansal

© Chapman Hall / CRC Press

ISBN: 978-146-6565142

Topics Covered

- Concurrent Execution and Abstractions
  - Race Condition; Threads and Dependencies; Synchronization and Mutual Exclusion; Sequential Consistency
- Program Dependency and Automatic Parallelization
  - Control Dependency; Data Dependency; Program Dependency Graph; Parallelization Techniques; Granularity and Execution Efficiency; Program Slicing
- Task and data Parallelism
- Distributed computing
  - Remote Procedures and Parameter Passing in RPC
- Communicating Sequential Processes
- Memory Models of Concurrency
- Concurrent Programming Constructs
- Concurrent Programming in Ada, Java and Emerald

Introduction

- Goal is to speed up computationally slow programs
- Advantages
  - Efficient execution of programs
  - Efficient utilization of multiple resources
- Levels of parallelism
  - Design new parallel algorithms
  - At algorithm level, identify subtasks that can be executed concurrently
  - Develop smart compilation to automatically incorporate parallelism
  - Write programs with parallel constructs
- Restrictions in exploiting parallelism
  - Waiting for conditional statement on which following statements depend
  - Waiting for statements that produce the data being consumed
  - Waiting when a shared data is being used by one of the processes
- Sequential consistency: the output after exploiting concurrency must be the same as executing sequentially
  - Parallelization should not violate the property of causality

Concurrent Execution and Abstraction

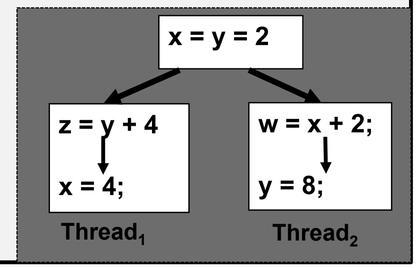
- Approaches to exploit concurrency
  - Parallelizing compilers: sequential programs → concurrent programs
  - Develop concurrent programs using concurrency constructs
- Dependencies limit concurrency
  - Causality of actions
  - Dependency due to control abstractions
  - Dependencies due to update and flow of data
  - Due to order imposed by the programmer in uniprocessor execution
- Dependency imposes sequential execution
  - Dependency to be minimized without violating sequential consistency
  - Causility based sequentiality is inherent and unavoidable
- Causes of sequentiality
  - Limited hardware and data access resources than needed by subtasks
  - Need to avoid racing condition
  - Need to manage shared resources

Race Conditions

- Race condition violates sequential consistency
  - Caused by the presence of shared variables updated in random order
  - A high level instruction is translated to multiple low level instructions
  - The sharing of result of partial computation by one subtask to another subtask may produce different outcomes inconsistently different than sequential execution
- Maintaining sequential consistency
  - Multiple actions together should be executed as one atomic action
  - To ensure atomicity a Boolean variable called semaphore / lock is set
- <u>Example of race condition</u> (x and w are aliases)
  - x = 4; y = 8; z = x + y; w = 5; y = 2 * w
  - Sequential execution gives *x = w = 5; y = 10; z = 12*
  - ***Concurrently executed and terminating in the order x = 4; w = 5; y = 8; z = x + y, y = 2* w gives the value x = w = 5; y = 10*, and *z = 13***

Threads and Dependencies

- Thread is a sequence of actions
  - light weight process that shares memory space with parent process
- Properties of threads
  - Parent process spawn multiple threads that merge after termination
  - Multiple threads execute concurrently in the same memory space
  - Shared variables need to be handled mutually exclusively by threads
- Dependencies
  - Shared variables introduce dependencies in thread actions
- <u>Example</u>
  - X and y are shared
  - Thread 1 produces value of x
  - Thread 2 produces value of y
  - Thread 1 consumes value of y
  - Thread 2 consumes value of x
  - Only one possibility of execution
  - Follows sequential consistency



Mutual Exclusion

- Mutual exclusion ensures sequential consistency
  - Only one process at a time. Others wait for their turn.
  - Lack of enforcement may cause incorrect program behavior.
- Mechanism to ensure mutual exclusion
  - Associate a separate lock with every shared resource.
  - Thread using the shared resource sets the lock before entering critical section, and releases after the end of the critical section.
  - Only one thread can grab the lock at a time.
- Problems with locks
  - Starvation of threads if locks are used aggressively.
  - Improper declaration violates mutual exclusion or causes starvation.
  - May cause excessive overhead of waiting if critical section is bigger.
- Monitor provides mutual exclusion among threads
  - Passive high level construct includes all mutually exclusive processes.
  - Uses **critical section** – a small chunk of code where lock is kept.
  - Operations within critical section are **atomic operations.**
  - Uses two operations: **lock** and **release** to ensure mutual exclusion

Atomic Operations Issues

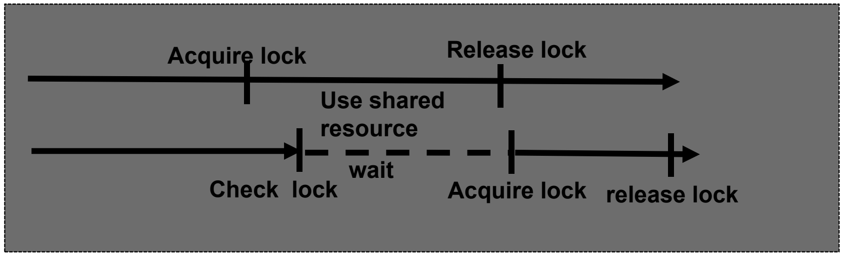
- Multiple threads work simultaneously.
- All the operations updating shared variables and using the updated value are placed together in a critical section.
- All the operations in a critical section are treated as one single instruction.
- Atomicity is enforced using locks.
- During atomic operation currently subtask will keep the control.

```
integer counter; max = 3000;
Thread vote-count:
{ ask; read(vote);
  counter = counter + 1;
if (counter < max)
     vote_array[counter] = vote; }
```

- Explanation
  - Shared variable is index variable counter
  - counter is used access the vote-array
  - If the control is passed to another voter thread after incrementing then counter is incremented by 2, and vote is not recorded

Synchronization

- Waiting for other subtasks before executing the next instruction
- Needed to maintain sequential consistency
- Synchronization is needed
  - In the presence of shared variables among multiple threads using lock
  - When a subtask is waiting for an input value to be produced
  - To avoid race condition



Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Program Dependency

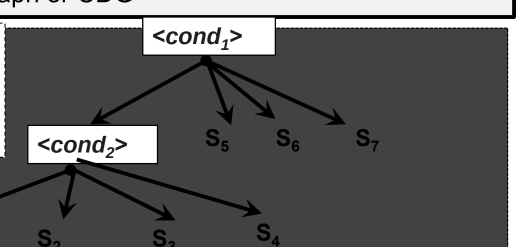
- Sequential consistency
  - The result of concurrent execution is same as sequential execution
  - Possible if the store attained by different permutation is the same
  - Possible if the operations are commutative
  - Store can be split into N mutually exclusive partitions each effected independently by different statement
  - The instructions do not update the store
- Program dependency
  - Statements are executed sequentially to maintain sequential consistency
  - Two types of dependency: control dependency and data dependency
  - Dependency relationship is transitive, antisymmetric and anti-reflexive
- Exploiting concurrency
  - Program's execution order is modeled as a directed acyclic graph
  - Edges between the statements shows control or data dependency
  - Dependent statements are executed sequentially

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Control Dependency

- Caused by control abstraction, separate from dataflow
- Sequentialty in control dependency
  - Conditional statement and actions. A high level control abstraction translated to low level instructions reveals the control dependency
  - Calling subprogram and called program
- S_1 dominates S_2 means S_2 is always executed after S_1
- S_2 post-dominates S_1 if all the paths from S_1 to end go through S_2
- The directed acyclic graph showing control dependency is called control dependency graph or CDG

```
If (<cond1>) {
   If(<cond2>){S1; S2; S3;}
   else  S4;
else {S5; S6; S7;}
```

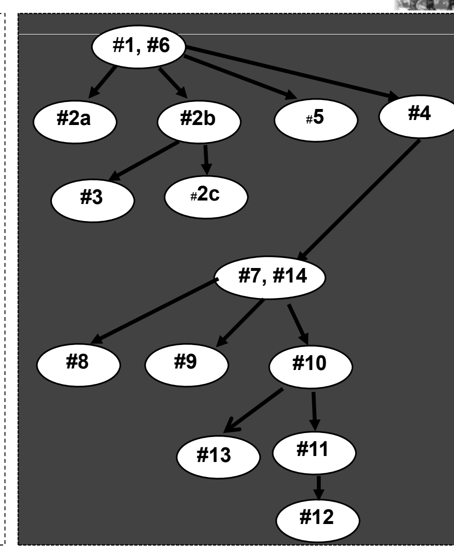


Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Control Dependency Graph

```
   integer  m[4], max;
   program main ( )  % start
1)  { integer i, j;
2)  for (i = 0; i =< 3; i++)
3)        read(m[i]);
4)  call find_max
5)  write(max);
6)  }  % stop

   procedure find_max ( )  % start
7)  {integer  i;
8)  max = m[0];
9)  i = 0;
10) while (i =< 3)
11)      if (m[i] > max )
12)            max = m[i];
13)      i++;
14) }  % stop
```

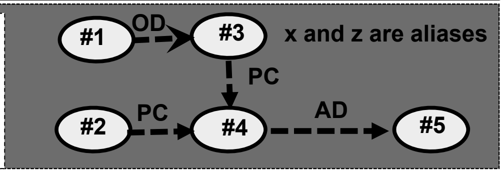


Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Data Dependency

- Sequential**on a graph paper** ty due to the presence of shared variables
- Types of dependencies
  - **Producer-consumer**: the value can not be used until produced
  - **Anti-dependence:** all the consumers of previous values must be executed before rewriting the shared variable
  - **Output dependence**: maintaining the sequential order of aliased variable to ascertain that consumers use the actual values
- Data dependency graph
  - Acyclic directed graph made of data dependencies
  - Loops should be unrolled to establish exact data dependencies

```
1. x = 4;   % producer
2. y = 5;   % producer
3. z = 8;   % OD
4. w = x + y;   % PC
5. z = 9  % OD and AD
```

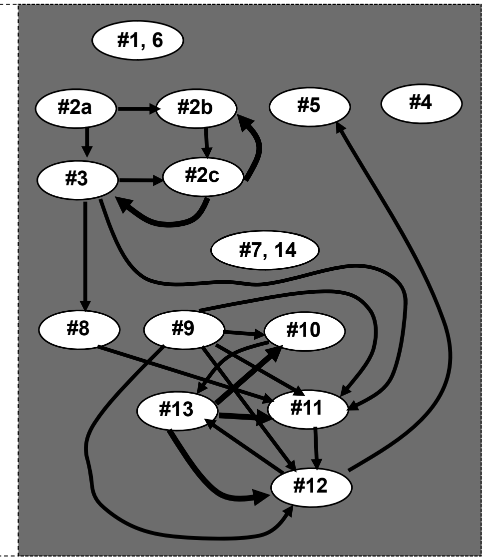


Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Data Dependency Graph

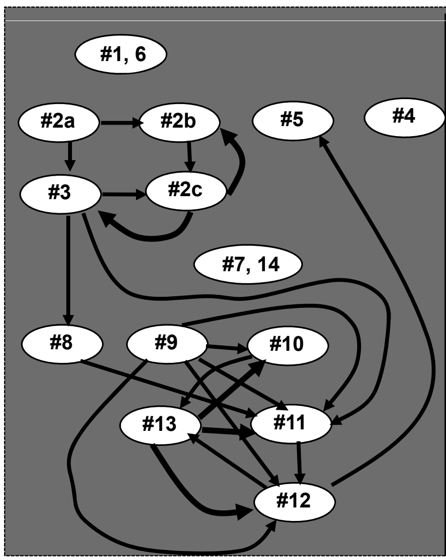
```
   program main ( ) % start
1) { integer i, j;
2)    for (i = 0; i =< 3; i++)
3)       read(m[i]);
4)    call find_max
5)    write(max);
6) } % stop

   procedure find_max ( ) % start
7) { integer i;
8)    max = m[0];
9)    i = 1;
10)   while (i =< 3)
11)      if (m[i] > max )
12)         max = m[i];
13)      i++;
14)} % stop
```



Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Types of Edges in the Graph

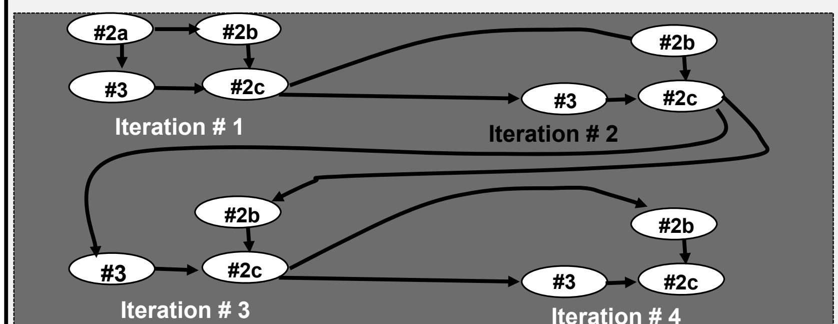


Edge	Type	Edge	Type
(#2a, #2b)	PC	(#9, #11)	PC
(#2a, #3)	PC	(#9, #12)	PC
(#2b, #2c)	AD	(#10, #13)	AD
(#2c, #2b)	PC	(#11, #12)	AD
(#2c, #3)	PC	(#12, #5)	PC
(#3, #2c)	AD	(#12, #13)	AD
(#3, #8)	PC	(#13, #10)	PC
(#3, #11)	PC	(#13, #11)	PC
(#8, #11)	PC	(#13, #12)	PC
(#9, #10)	PC		

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Loop Unrolling Example

- Loop must be unrolled to see all the dependencies
- The following example establishes the acyclic nature
  - *{(iter$_k$: # 2b, iter$_k$: #2c),* **(iter$_k$: *#2c, iter$_{k+1}$: #2b*)}**
  - **{(iter$_k$:** ***#2c, iter$_{k+1}$: #3*),** (iter$_k$: *#3, iter$_k$:* #2c)}
- Edges are between two different iteration cycles

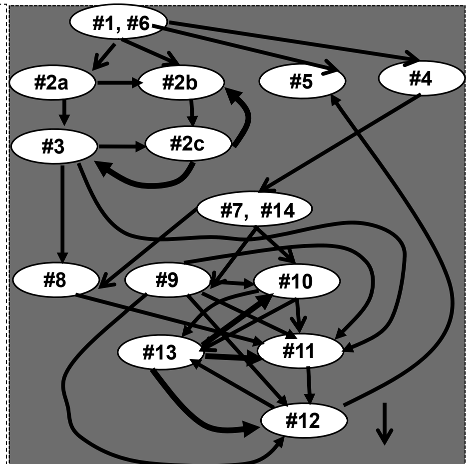


Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Program Dependency Graph

- Actual dependency between the statements in a program
- Formed by superimposing control and data dependency graphs

```
    program  main ( )  % start
1)  { integer i, j;
2)     for (i = 0; i =< 3; i++)
3)        read(m[i]);
4)     call find_max
5)     write(max);
6)  }  % stop
   procedure find_max ( )  % start
7)  { integer  i;
8)    max = m[0];
9)     i = 1;
10)   while (i =< 3)
11)      if (m[i] > max )
12)         max = m[i];
13)      i++;
14)}  % stop
```

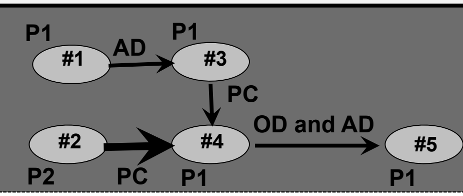


Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Parallelization Techniques

- Find independent paths in a program dependency graph
- Issues in incorporating concurrency
  - allocation for nodes sharing data-dependency edges
  - removing redundant dependencies caused by control abstractions
  - keeping the data transfer overhead between processors minimal
- Mapping program dependency graph on processors
  - Unroll the loops depending upon processor availability
  - Map statements on processors so that data transfer between processors is minimal to reduce data transfer overhead
  - Dependent statements can be mapped on the same processor

```
1) x = 4;    % producer
2) y = 5;   %  producer
3) z = 8;   % OD
4) w = x + y;   % PC
5) z = 9  % OD and AD
```



Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

For-Loop Unrolling

- Control abstractions introduce redundant sequentialty
  - Due to the presence of index variables and incrementing them
- <u>Examples</u>
  - **for** (i = 0; i =< 3; i++) read(m[i]); is sequential. However unrolled version is concurrent***: read(m[1]), read(m[2]), read(m[3]), read(m[4])***
  - **for** (i = 0; i =< n, i++) a[i] = b[i] + 4 is sequential when not unrolled. However, unrolled version is concurrent
- Unrolling with limited processors

for (i = 0; i =< 1000; i++)
  {a[i] = 10;   b[i] = a[i] + 4; c[i] = 2 * b[i];}
is translated to (for four processors)
for (i = 0; i =< 250; i++)
{ a[i] = 10;   b[i] = a[i] + 4; c[i] = 2 * b[i]; % on processor 1
   a[i + 1] = 10;   b[i + 1] = a[i + 1] + 4; c[i + 1] = 2 * b[i + 1]; % on processor 2
   a[i + 2] = 10;   b[i + 2] = a[i + 2] + 4; c[i + 2] = 2 * b[i + 2]; % on processor 3
   a[i + 3] = 10;   b[i + 3] = a[i + 3] + 4; c[i + 3] = 2 * b[i + 3];} % on processor 4

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

While-loop Unrolling

- Issues
  - Due to the indefinite size, needs to exit out of the unrolled block
- Mechanism
  - Unroll the block as many times as number of processors
  - Implement an conditional exit statement with each statement
- Illustration after unrolling with four processors

```
while (<cond>)
  {     <block>;   if (<cond>) exit;
        <block>;   if (<cond>) exit;
        <block>;   if (<cond>) exit;
        <block>;
  }
```

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Granularity and Execution Efficiency

- Granularity is the size of statement blocks executed sequentially on the same processor
- Types of granularity
  - Fine-grain: number of statements is very small
  - Coarse-grain: number of sequentially executed statements on a processor is large
- Problems with fine-grain concurrency
  - Too much interprocessor data-transfer overhead.
  - The advantage gained by distributing statements is lost due to excessive data transfer overhead
- Advantages of coarse-grain concurrency
  - Data transfer overhead is significantly reduced
- Concurrency gains only sublinear speed up due to
  - Limited hardware resources such as data bus and memory ports
  - Shared variables in a program
  - Packing-unpacking cost and use of system routines in data transfer
  - Need to exchange the objects and environments in distributed computing

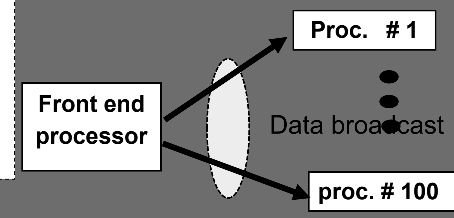
Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Program Partitioning

- Partition the programs such that
  - Parts of program with lots of data dependency are grouped together and executed on the same processor to remove data transfer overhead
  - Nested loops should be unrolled and the corresponding data should be distributed on processors at compile time

```
read(a[i]);
for (I = 1; I<1000;I++) read(b[j]);
for (I = 0;  I < 100000; I++)
  for (j = 0; j < 1000; j++)
     c[I, j] = a[I] * b[j];
```

Sequential program




```
for (I = 0; I < 100000; I++)
read(a[i]);
for (I = 1; I < 1000; I++)  read(b[j]);
```

Front-end partition

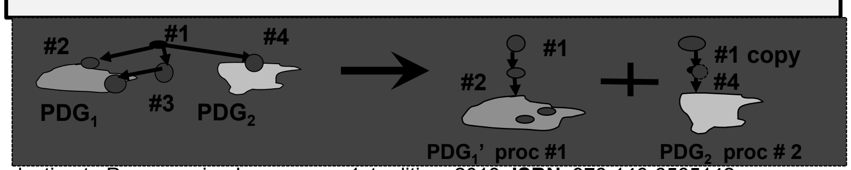
```
for (I = 0;  I < 100000; I++)
   for (j = 10*(k -1); j < 10*k - 1; j++)
       c[I, j] = a[I] * b[j];
```

Processor k partition

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

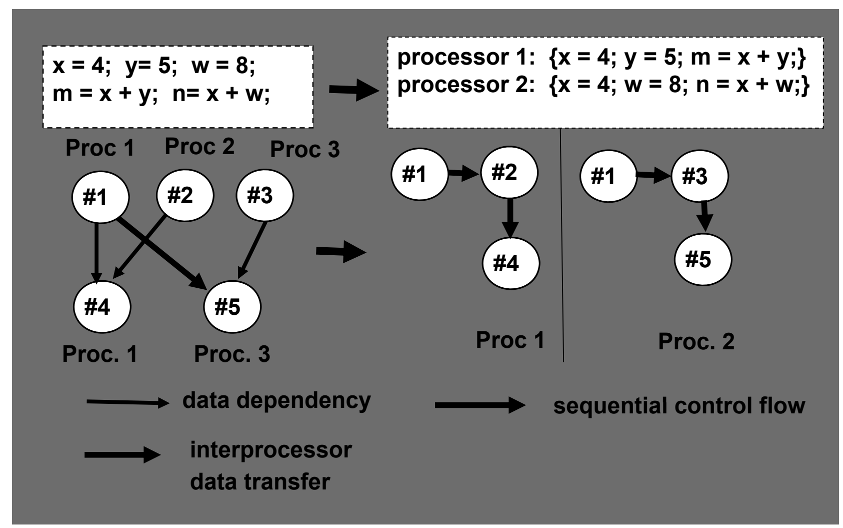
Program Slicing

- Compile-time analysis of PDG to optimize program properties
  - Splits a program in multiple slices to optimize the program properties
- Application
  - matching programs; identifying duplicated code in programs; debugging the programs; software maintenance; automated parallelization
- Program slicing for automatic parallelization
  - Minimize the data transfer overhead by dividing the programs into multiple blocks with minimal data transfer overhead
- Technique
  - Identify statement-nodes in PDG that sends data to multiple processors, and create clones of those statements
  - Identify data-dependency edges connecting two different processors and map the sink node in the same processor



Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Program Slicing Example



Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Task Parallelism

- Spawns and manages multiple threads executing concurrently
- Threads spawned using thread-pool that contains inactive threads
  - Thread-pool removes the overhead of creating threads
- Implementation in various languages
  - C, C++ use a thread library such as Posix
  - Java uses synchronized methods
  - Ada uses task construct to spawn multiple subtasks
- Issues in handling multiple concurrent threads
  - synchronization and handling shared resources
  - communication between the threads and the parent processes and communication among threads
  - resource allocations to avoid starvation and deadlocks
  - resource deallocation when the process or threads are terminated
- Starvation is indefinite waiting by a process/thread for CPU-time
- Deadlock is when two more processes are waiting for resources other processes are holding without releasing them timely

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Handling Concurrent Threads

- Approaches
  - Without mutual communication that does not used any shared resources
  - the use of shared resources
- Shared resource approach
  - Use of lock in critical sections to enforce mutual exclusion
  - Shared resources can be handled using high level constructs such as monitors as in Concurrent Pascal or synchronized methods as in Java
- Operations on locks
  - **Wait_and_lock** to acquire the shared resource
  - Release to free the shared resource
- Problem with threads
  - Critical section has to be restarted if interrupted/aborted in between
  - Large boundary of critical section causes excessive wait
  - Blocking large objects (ex: arrays) causes unnecessary sequentiality
  - Objects should not be blocked for read operations
  - Overhead of checking the status of the locks
  - Improper placement of locks can cause starvation or incorrect execution

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Solving Concurrent Thread Issues

- Reading problem
  - Use different types of locks: **shared-read** lock allows concurrent reading; **exclusive locks** are traditional locks
- Critical section boundary problem
  - Use of **transactional memory** and **shadow copy** to allow reading before update and after update provided no read-write conflict exists
  - Update is done only after the successful transaction is committed
  - Each transaction keeps its **read-set** and **write-set** to avoid conflict
- Read-set and Write-set conflict resolution
  - **Eager resolution**: pessimistic, finds more conflict, no rollback problem
  - **Lazy resolution**: finds less conflicts, optimistic but has overhead of transaction rollback if the conflict resolution is late
- Improper placement of lock problem: use monitors
- Alternative Information exchange between threads
  - Asynchronous producer-consumer information exchange using buffer

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Data Parallelism

- Applying the same instruction on multiple elements concurrently
  - The data-elements should not have data-dependency
  - Works well with flat data structures: arrays, sequences, and sets
- <u>Example of loop with data-parallelism</u>
  - **for** (i = 0; i =< 1000; i++) a[i] = a[i] + 4 % data_parallel for-loop
- <u>Example of loop with no data-parallelism</u>
  - **for**(i = 1; i =< 999; i++) a[i] = a[i – 1] + a[i + 1] % sequential for-loop
  - It is sequential because a[i] is dependent upon the value of a[i – 1]
- Map-reduce model of data parallelism
  - *map* function sorts a collection of (key, value) pairs
  - *Reduce* function groups the sorted values into groups that can be handled by a user-defined function
- Data parallel constructs
  - a[1:N] = b[1:N] + c[1:N]

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Integrating Task and Data Parallelism

- Task parallelism is supported by MIMD architecture
- Data parallelism is supported by SIMD architecture
- Advantage of integration
  - There are problems that need both task and data parallelism.
- Different approaches of integration
  - Multiple concurrent data parallel computations
  - A coordinator process that spawns multiple data parallel subtasks
  - Shared data abstraction (SDA) written concurrently by multiple data-parallel operations
  - Distributed shared data structures: Shared data structures is a an accumulation of multiple distributed spaces. Each processor writes into its own partition in data parallel manner. A subtask can access data space on other processors too.

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Distributed Computing

- Distributed computing
  - execute procedures concurrently on different processors
  - Supports either **data migration, code migration** or a combination of two
- Data migration: transfer data to remote processors for computation
  - Useful when information exchange is not too large
- Code migration: transfer code to remote processor for computation
  - Useful when overhead of code transfer is less than overhead of data transfer; and to reduce the computational overhead on servers
- Communication overhead between distributed processors
  - Overhead of multiple layers of network protocol
  - Computational overhead of linearizing the data at source and delinearizing the data at the destination
  - Synchronous communication blocks the process until the acknowledgement sent by the receiver is received
  - Asynchronous communication: sender deposits in the mailbox that is retrieved using a system routine
  - Need for access of local resources by remote processors

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Executing Remote Procedure

- Executing subprograms on remote processors
  - More than one memory address apaces
  - Information is physically transferred across machines using network
  - The result may be passed back across the processors.
- Information exchange mechanism to remote processors
  - **Passing the reference**: significant overhead of information access; easy to pass the references
  - **Passing the value:** overhead of object details, channel, and buffer address needs to be transferred
- Mechanism to invoke remote procedure
  - Pack (**marshall)** all the information into a packet called **stub**
  - Send the information using system call to network layer using a channel
  - Unpack the information at the remote end, and execute
  - Pack the result back, and send through the stub to calling subprogram
- Mechanism works fine in uniform operating system

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Parameter Passing in RPC

- Types of parameter passing
  - Call by move similar to call by value in single address space
  - Call by reference similar to call by reference in single address space
  - Call by copy-restore similar to call by value-result in single space
  - Scheme that copies the object only once and then uses the remote copy by looking at the object-id when requested for object. The scheme is an integration of call by reference + call by move
- Passing parameters has overhead due to
  - System calls, communication layer, delay in transmission, marshaling and demarshaling, conversion of data structure to different format in heterogeneous address space
  - Undoing the effect of changes during call by reference if the called procedure does not terminate properly or communication fails
  - Suspension of the calling program to preserve the data structure when called program is executing in call by copy-restore
- Languages supporting different parameter scheme
  - Emerald supports all forms
  - Java supports Remote method Invocation

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Communicating Sequential Process

- A model of concurrent programming using guarded commands
  - Guards are used as input sources; commands are used as output
  - Concurrent processes (P || Q) are disjoint and do not share variables
  - No need for synchronization because there are no shared variables
- Notations for operations on concurrent processes
  - αP : set of events seen by a process P
- Operations with same alphabet
  - commutativity: $P \parallel Q \equiv Q \parallel P$
  - associativity: $P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$
  - deadlock: $P \parallel Stop_{\alpha P} \equiv Stop_{\alpha P}$
  - running: $P \parallel Run_{\alpha P} \equiv P$
  - agreement: $(c \rightarrow P) \parallel (c \rightarrow Q) \equiv (c \rightarrow (P \parallel Q))$
  - disagreement: $(c \rightarrow P) \parallel (d \rightarrow Q) \equiv Stop$
- Operations on different alphabet
  - $(a \rightarrow P) \parallel (c \rightarrow Q) \equiv (a \rightarrow (P \parallel (c \rightarrow Q))$
  - $(c \rightarrow P) \parallel (b \rightarrow Q) \equiv (b \rightarrow (Q \parallel (c \rightarrow P))$
  - $(a \rightarrow P) \parallel (b \rightarrow Q) \equiv (a \rightarrow (P \parallel (b \rightarrow Q)) \mid (b \rightarrow (Q \parallel (a \rightarrow P))$

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

More Operations on Processes

- Sequential composition
  - is_associative $(P; Q); R \equiv P; (P; R)$
  - with_unit $skip; P \equiv P$
  - with_zero $Abort; P \equiv Abort$
  - distributes $(a \rightarrow P); Q \equiv a \rightarrow (P; Q)$
- Guarded command
  - is_associative (P □ Q) □ R ≡ P □ (Q □ R)
  - commutative P □ Q ≡ Q □ P
  - distributes (P □ Q); R ≡ (P; R) □ (Q; R)

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

CSP Language

- uses laws of concurrency, sequential composition and guarded commands
- CSP language statement
  - Input → output command
  - input: declaration, a Boolean condition, or a process with input data.
  - Output command: skip, assignment, alternative command, parallel command, iterative command, or a process that outputs data.
  - Syntax for input process supply data: *<process>*?*<input-data>*
  - Syntax for output process generating data: *<process>*!*<output-data>*
  - Output processes terminate when all the input processes terminate or guards are no more true
- Language support and semantics
  - single data entities, arrays, structured data, array of processes, assignment, alternative commands, iterative commands, parallel commands, recursive processes, and subroutines.
  - Iterative loop continues until all input processes stop supplying data
  - Semantics of guards is the same as guarded commands

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Syntax of CSP Language

<parallel-command> ::= '['(*<process>* {|| *<process>* }* ']'
<process> ::= *<identifier>* {(*<command>* | *<declaration>*}+
<command> ::= *skip* | *<assignment>* | *<input-statement>* |
<output-command> |*<alternative-command>* |
<iterative-command> | *<parallel-command>*
<assignment> ::= *<variable>* = *<expression>*
<input-statement> ::= *<process-name>*?*<variable>*
<output-command> ::= *<process-name>*!*<variable>*
<iterative-command> ::= '*'*<alternative-command>*
<alternative-command> :: '['*<guarded-command>*
{'□' *<guarded-command>*}* ']'

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Example

```
iobuffer::
        m = 80; buffer(0..m - 1) character;  c character;
        rear, front: integer; rear = 0; front = 0; count = 0;
        full, empty: Boolean; full = false; empty = true;
        * [ count == m - 1→  full = true □
           count == 0 → empty = true □
           not(full); producer?c →   % read  c from  the producer
                buffer[front] = c;
                front = front + 1 mod m;
                count = count + 1;
                empty = false □
                not(empty)   → consumer ! buffer(rear);
                rear = rear + 1 mod m;
                count = count  - 1;
                full = false ]
```

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Memory Models of Concurrency

- Specifies how memory behaves under models of concurrency
- Memory model is important because
  - Compilers reorganizes the instructions for optimization many times violating the property of sequential consistency
  - The memory model should provide race free execution without introducing extra sequentiality
  - In the absence of a safe model, hackers may attack Internet languages
- Synchronization properties for sequential consistency
  - Data-race free
  - Follow synchronization order consistent with program order
  - Lock action should be followed by release action
  - After a lock is placed, no process should violate lock before release. This problem is difficult to maintain for many languages since they allow unsynchronized methods to access shared resource
  - A read operations reads the last updated value
  - A write operation should wait unless all read operations on previous write in the program order have taken place – **Anti dependency property**

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Current Memory Models

- Java 5 has an extensive memory model
  - Supports volatile variables to separate locks and shared variables from other regular variables. Program order of the volatile variables is not altered by the optimizing compilers
- Problems in existing memory model
  - Lock set in one thread can not be reset in another thread
  - Use of shared variables in two threads gives rise to cyclic reasoning that can only be broken by global analysis of causality
  - A thread may be blocked for input output interaction. Due to volatile variable maintaining program order, all other threads waiting for volatile variables are also blocked unless the lock is released

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

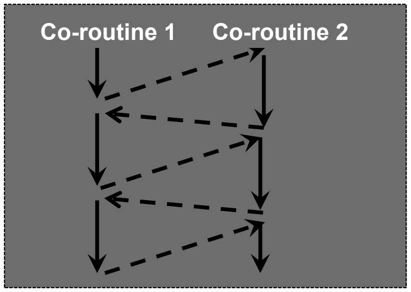
Concurrent Programming Constructs

- Coroutines
  - Control between two processes alternate
  - Each process saves its state before passing the control
  - implemented in Simula, Modula-2, Ruby, Lua and Go
- Constructs for data parallel programming
- Constructs for parallel spawning of subtasks
  - Cobegin and Coend pair: spawns subtasks without shared variables
  - Fork and join: starts multiple processes, parent process suspends, and resumes after all spawned processes terminate
- Constructs for spawning multiple threads
  - thread.**new**( ) ; thread.start (); thread.join ( );
- Synchronization constructs to handle shared resources
  - Lock(v); … release(v)
  - Monitors: support mutual exclusion of procedures
  - Synchronized methods
- Constructs for invoking remote procedures

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Co-routines

- Two processes alternate
- They save their state before passing the control to other
- They resume from the last suspension point
- implemented in Simula, Modula-2, Ruby, Lua




```
fact = Fiber.new
   m, n = 1, 1
   loop do
      Fiber.yield n
      m = m + 1
      n = n * m
   end
end
5.times {puts fact.resume}
```

- Two coroutines: function *fact* generates a factorial and the external loop prints out the value
- External loop starts the fact
- Fact generates value using fiber.yield and suspends and passes the control to external loop
- External loop resumes the fact coroutines again using the method *fact.resume*

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Cobegin / Coend

- Spawns multiple independent subtasks
  - No shared variables between subtasks
  - Main task suspends when subtasks start, and resumes after all subtasks terminate
- Concurrent Pascal syntax

```
x := 0; z := 4;
cobegin
      begin x := 1;  x := x + 1 end;  % concurrent activity 1
      begin z := 2; z := z - 1 end;  % concurrent activity 2
Coend;
```

- **SMIL syntax**

```
<par>
     <text src = "my_resume.html"   region = "text_area"    dur = "60s" />
     <video src = "my_presentation.mpg"  region = "Video_area"   />
</par>
```

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Monitors

- Monitors are passive declarations like modules
- Embedded entities in a monitor
  - Shared resources
  - Mutually exclusive procedures that work on shared resources.
  - An initial body is executed when a monitor is called.
  - A process can access its own variables.
- Any process has to use monitor and the corresponding procedures to access the shared resource.
  - There is no direct access to shared resources.
  - Waiting is done using **spin-lock** or **suspension of a process.**
- Abstract syntax

type <*identifier*> = **monitor** (<*parameter-list*>)
<*shared variable declarations*>
procedure <*identifier*> <*procedure-body*>
procedure <*identifier*> <*procedure-body*>
…
procedure <*identifier*> <*procedure-body*>
<*initial-body of the monitor*>

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

An Example of Monitor

```
monitor iobuffer;
m = 80;
char buffer[0..m - 1];
integer rear, front, count;
condition nonempty, nonfull;
procedure insert(char element)
{  if  (count == m - 1)  then
await(nonfull);
buffer[rear] = element;
rear = (rear + 1) modulo  m;
count = count + 1;
signal(nonempty)}

procedure remove (char element)
{   if (count == 0) then
await(nonempty)
      result = buffer[front];
front = (front + 1) modulo n;
      count = count – 1;
      signal(nonfull)}

% initial body of the monitor
{ rear = 0; front = 0; count = 0;
nonempty = false; nonfull = true }
…
iobuffer  console;
…
{
…
console.insert('a');
…
}
```

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Concurrent Programming in ADA

- Ada uses tasks for concurrent programming
  - Tasks are equivalent to Java threads
  - Task is declared like a module
  - Two important features of task: ***entry-point*** and ***accept***
  - *Entry points-accept* pair is used for parameter passing from other tasks
- Abstract syntax

Procedure <*proc_identifier*>

task <*task-name*$_1$> **is** <*entry-points*$_1$> **end** <*task-name*$_1$>

task body <*task-name*$_1$> **is** <*block*$_1$> **end** <*task_name*$_1$>

…

task <*task-name*$_N$> **is** <*entry-point*$_N$> **end** <*task-name*$_N$>

task body <*task-name*$_N$> **is** <*block*$_N$> **end** <*task_name*$_N$>

begin *null* **end** <*proc_identifier*>

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Example in ADA

```
WITH Ada.Text_IO;
USE Ada.Text_IO;
PROCEDURE Assignment   IS
  TASK   SolveProblem   IS    ENTRY start_thinking (Problem_index: INTEGER);
   END   SolveProblem;
  TASK BODY SolveProblem    IS
 BEGIN
      ACCEPT start_thinking (Problem_index: INTEGER) DO
        delay 240.0;  -- Put delay to simulate time taken to solve a problem
         Put_Line("write answer ");
      END start_thinking;
  END SolveProblem;

BEGIN
  FOR Index    IN 1..5    LOOP
    SolveProblem.start_thinking(Index)
    Put_Line("Solving  the next problem");
  END loop;
  Put_Line("Assignment done");
END Assignment
```

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Concurrent Programming in JAVA

- Concurrency constructs in Java
  - Thread primitives and synchronized methods
  - <*thread-name*>.**star**t ( ), <*thread-name*>.**yield** ( ), **<**thread-name>**.sleep**(<*duration*>), <*thread-name*>.**setPriority** (<*priorityValue*>)
  - Synchronized methods are used for shared objects that are visible to other threads. Confined objects need not be synchronized.
  - Methods using a shared variable are declared as *synchronized methods*
- Problems with synchronized methods
  - Locks cannot be set in one method and released in another method.
  - Locks are not at the variable level but at the method level
  - An unsynchronized method can also update the shared variables
  - Synchronized method causes sequentialty slowing down the execution.

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Distributed Programming in Emerald

- Emerald supports fine grain object mobility
- Components of Emerald objects
  - *Unique network id* that can be generated by concatenating *host-name, process-name* and *local identifier*
  - Data representation local to the object
  - Methods working on the local data
  - An optional process that may invoke other objects
- Types of objects
  - Global objects – objects are completely mobile
  - Local objects – immovable objects embedded under another object
  - Direct objects – basic type used to build another object
- Advantages of fine-grain mobility in Emerald
  - Enhances load balancing
  - Object mobility provides robustness against processor failure
  - Active objects can be moved to other processors for better efficiency
  - Better utilization of special software on specialized processors

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Author: Arvind Bansal © Chapman Hall/CRC Press, 2013, All rights reserved

Object Mobility in Emerald

- Operations for object mobility
  - Locate an object
  - Move an object
  - Fix an object at a particular node
  - Unfix an object to make it mobile again
  - Refix an object that is a combination of unfix, move, and fix at a node
- Parameter passing in Emerald
  - ***Call by object-reference****:* remote objects can be accessed only by going through the operating system. Remote objects reference can be passed to distributed nodes easily
  - ***Call by visit:*** object is moved to remote processor, and after the computation, object is copied back
  - ***Call by move:*** object is moved to remote processor. However, it is not copied back from the remote processor

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Summary I

- Concurrent programming is concerned about dividing a task into multiple subtasks executing concurrently
  - Concurrent execution should maintain sequential consistency
- Concurrency can be exploited using task parallelism, data parallelism, or the integration of the two
  - Data parallelism is about the same operation on multiple data
  - Task parallelism is about different subtasks on different or same data
  - Integration can be done using: 1) spawning multiple data parallel tasks concurrently; 2) distributed data structure and subtasks working concurrently
- Subtasks can be executed concurrently using:
  - Multiple processes / threads without shared variables
  - Threads with shared variables: needs synchronization
- There are three types of dependencies :
  - Control dependency and data dependency

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Summary II

- Three types of data dependencies
  - Producer-consumer, anti dependency, output dependency
- Control dependency is dependent upon
  - Conditional statements and procedure calls
  - Conditional statement is embedded in if-then-else statement, while-loop, for-loop and case statements
- Concurrency is exploited at
  - Fine-grain concurrency with packing / unpacking overhead
  - Coarse-grain parallelism groups multiple statements on single processor to reduce the data transfer and packing-unpacking overhead
  - Program slicing reduces communication overhead by replicating statements and grouping statements to execute on one processor
- Shared variable synchronization is done using
  - Locks and monitors
  - Locks are associated with a shared resource, and captured by a single process and released after the end of the critical section
  - Locks can cause additional sequentialty, deadlocks and starvation

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Summary III

- Critical sections and atomic operations
  - All operations in a critical section are treated as one atomic operation
  - Critical section should be as small as possible to avoid sequentiality
- CSP is an algebraic treatment of processes
  - Based upon guarded commands and algebraic theory of processes
- CSP language is based upon CSP algebra
  - Input part is guard includes declarations and input streams
  - Output is command and output streams
  - All input processes must terminate for a process to terminate
- Distributed computing
  - Is based upon code and data mobility for better resource utilization
  - Code mobility uses code migration or object migration
  - Emerald uses object migration. Emerald objects are flat
  - There is additional overhead of accessing distributed objects due to the involve of operating systems and computer network
  - Emerald uses both reference and object movement for parameter passing

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142