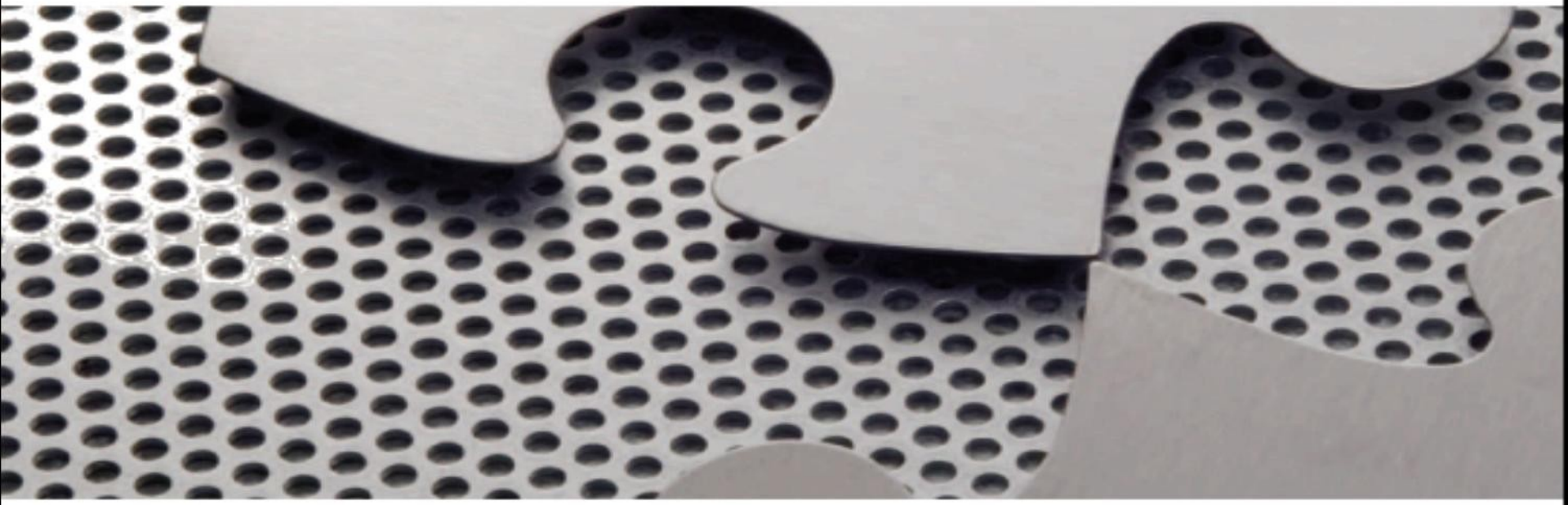
Programming Languages
Third Edition



Chapter 13

Parallel Programming

Objectives

- Understand the nature of parallel processing
- Understand the relationship of parallel processing to programming languages
- Understand threads
- Understand semaphores
- Understand monitors

Objectives (cont’d.)

- Understand message passing
- Become familiar with parallelism in non-imperative languages

Introduction

- **Parallel processing**: executing many computations in parallel
- **Multiprogramming**: many processes share a single processor, thus appearing to execute simultaneously (pseudoparallelism)
- **True parallelism**: many processors are connected to run in concert
  - **Multiprocessor system**: a single system incorporates all the processors
  - **Distributed system**: a group of standalone processors connected by high-speed links

Introduction (cont'd.)

- Implementation of parallel processing is not simple, and parallel systems are relatively uncommon
- High-speed networks, including the Internet, present new possibilities for using physically distant computers in parallel processing
- Organized networks of independent computers can be viewed as a kind of distributed system for parallel processing
- A comprehensive study of parallel processing is beyond the scope of this course

Introduction (cont'd.)

- Programming languages have affected and been affected by parallelism in several ways
- Languages have been used to:
  - Express algorithms to solve parallel processing issues
  - Write operating systems for parallel processing
  - Harness the capabilities of multiple processors to solve application problems efficiently
  - Implement and express communication across networks

Introduction (cont'd.)

- Must also consider the basic ways that programmers have used programming languages to express parallelism
- Must distinguish parallelism as expressed in a language, and the parallelism that actually exists in the underlying hardware
- Programs that are written with parallel programming constructs do not necessarily result in actual parallel processing
  - May be implemented by pseudoparallelism, even in a system with multiple processors

Introduction (cont'd.)

- Parallel programming is sometimes called **concurrent programming**
  - Emphasizes the fact that parallel constructs express only the **potential** for parallelism

Introduction to Parallel Processing

- **Process**: the fundamental notion of parallel processing
  - The basic unit of code executed by a processor
  - An instance of a program or program part that has been scheduled for independent execution
- **Jobs**: an earlier name for processes
- In the early days, jobs were executed sequentially in **batch** fashion
  - Only one process in existence at a time

Introduction to Parallel Processing (cont'd.)

- With the advent of pseudoparallelism, processes could exist simultaneously in one of three states:
  - **Executing**: in possession of the processor
  - **Blocked**: waiting for some activity, such as I/O, to complete
  - **Waiting**: ready for the processor and waiting for it
- The operating system needs to apply some algorithm to schedule processes and manage waiting and blocked processes
  - Principal method for this is the **hardware interrupt**

Introduction to Parallel Processing (cont'd.)

- **Heavyweight process**: corresponds to the earlier notion of a program in execution
  - Full-fledged independent entity, together with all the memory and other resources allocated to an entire program
- **Lightweight process**: shares its resources with the program it comes from
  - Does not have an independent existence
  - Also called a **thread**
  - Can be very efficient since there is less overhead in their creation and management

Introduction to Parallel Processing (cont'd.)

- In true parallel processing, each processor may individually be assigned to a process
  - Each processor may or may not be assigned its own queues for maintaining blocked and waiting processes
- Two primary requirements for the organization of processors in a parallel processing system:
  - Must be a way for processors to synchronize their activity
  - Must be a way for processors to communicate data among themselves

Introduction to Parallel Processing (cont'd.)

- **Single-instruction, multiple-data (SIMD) systems**: central control by one processor, with each other processor executing the same instructions on their respective registers or data sets
  - Are multiprocessing rather than distributed systems
  - May be synchronous, operating at the same speed, with the controlling processor determining when each instruction is executed

Introduction to Parallel Processing (cont'd.)

- **Multiple-instruction, multiple-data (MIMD) systems**: processors operator at different speeds and are asynchronous
  - May be either multiprocessor or distributed processor systems
  - Synchronization of the processors is a critical problem
- Hybrid systems are also possible

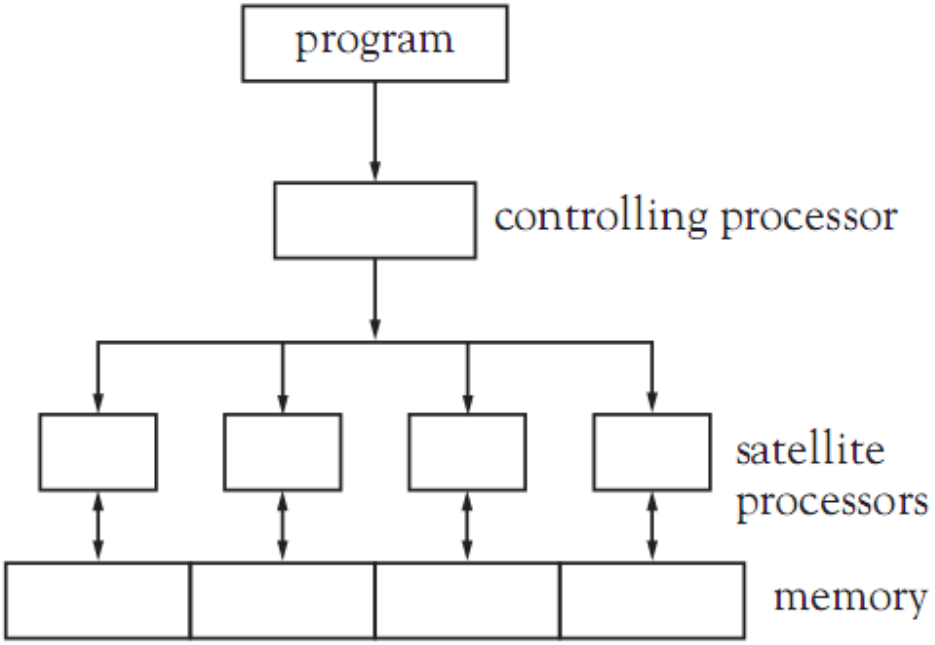


Figure 13.1a Schematic of an SIMD processor

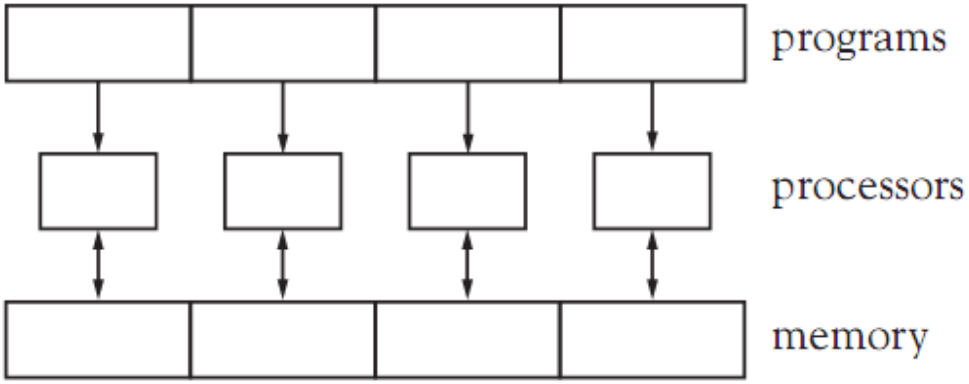


Figure 13.1b Schematic of an MIMD processor

Introduction to Parallel Processing (cont'd.)

- **Shared-memory system**: one central memory is shared by all processors
  - By nature, this is a multiprocessor rather than a distributed system
- In a shared-memory system:
  - Processors communicate through changes made to shared memory
  - Each processor must have exclusive access to those areas of memory that it is changing, to avoid race conditions

Introduction to Parallel Processing (cont'd.)

- **Race condition**: when different processes modify the same memory in unpredictable ways
  - Solving the mutual exclusion problem typically means blocking processes when one is already accessing shared data, using a locking mechanism
- **Deadlock**: occurs when processes end up waiting for each other to unblock
  - Detecting or preventing deadlock is a difficult problem in parallel processing

Introduction to Parallel Processing (cont'd.)

- **Distributed-memory system**: each processor has its own independent memory
  - In distributed-memory systems, each processor has its own memory that is inaccessible to other processors
- Distributed processors have a **communication problem**
  - Each must be able to send and receive messages from all other processes asynchronously
  - Processes may block while waiting for a needed message

Introduction to Parallel Processing (cont'd.)

- Communication depends on the configuration of links between processors
  - Processors may be connected in sequence, and need to forward information to other processors farther along the link
  - If few processors, they may be fully linked to each other
- Deadlock problems in a distributed-memory system are also extremely difficult
  - Each process may have little information about the status of other processes

Introduction to Parallel Processing (cont'd.)

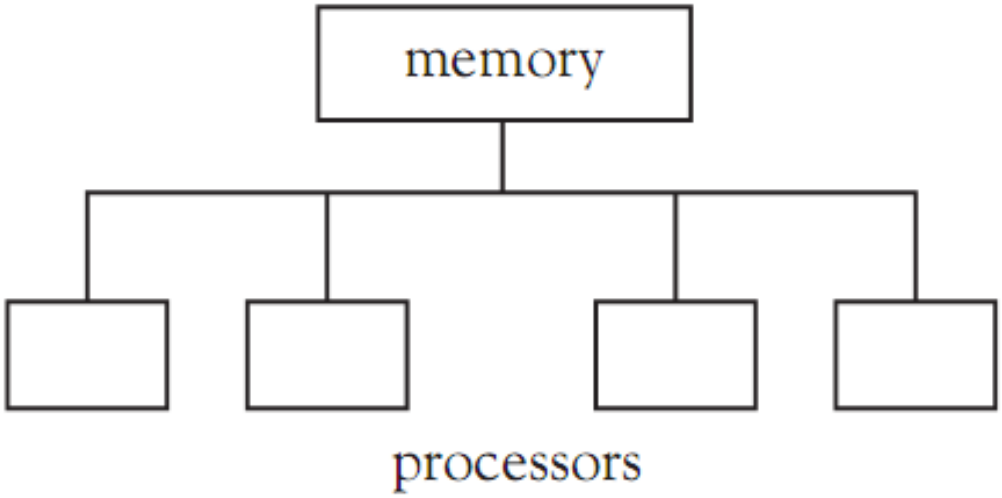


Figure 13.2a Schematic of a shared-memory system

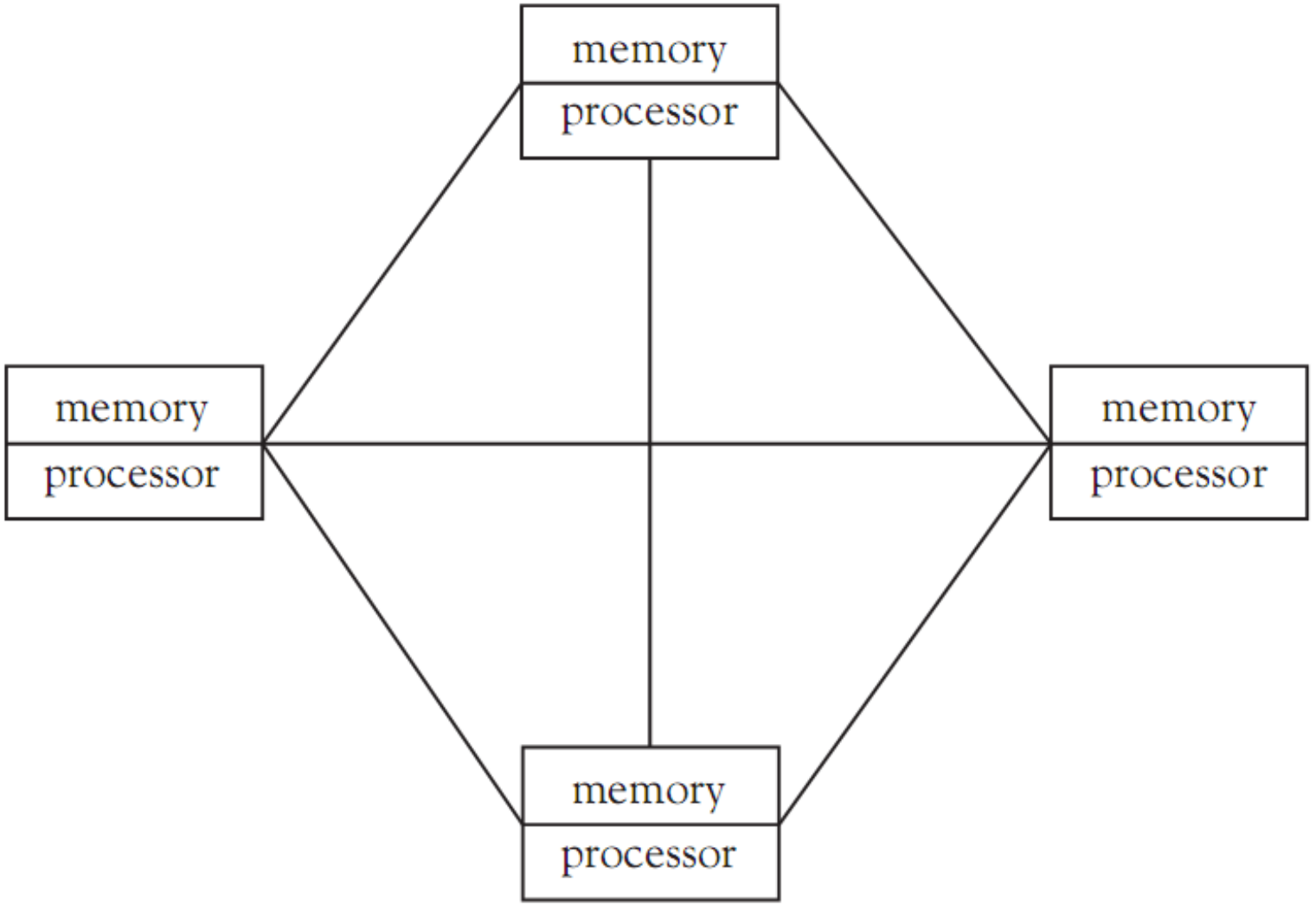


Figure 13.2b Schematic of a fully linked distributed-memory system

Introduction to Parallel Processing (cont'd.)

- It is possible for a system to be a hybrid between a shared and distributed-memory system
  - Each processor maintains some private memory in addition to the shared memory
  - Processors have some communication links separate from shared memory
- It is the task of the operating system to integrate the operation of processors and to shield the user from needing to know too much about the configuration

Introduction to Parallel Processing (cont'd.)

- In general, an operating system should provide:
  - A means of creating and destroying processes
  - A means of managing the number of processors used by processes
  - On a shared-memory system, a mechanism for ensuring mutual exclusion of processes to shared memory (for process synchronization and communication)
  - On a distributed-memory system, a mechanism for creating and maintaining communication channels between processors

Parallel Processing and Programming Languages

- Some programming languages use the shared-memory model
  - They provide facilities for mutual exclusion, usually via a built-in thread mechanism or thread library
- Some languages assume the distributed model
  - They provide communication facilities
- A few languages include both models
- A language may not include parallel mechanisms in its definition at all
  - Parallel facilities may be provided in other ways

Parallel Processing and Programming Languages (cont'd.)

- We will discuss two approaches to shared-memory models:
  - **Threads**
  - **Semaphores** and **monitors**
- We will discuss **message passing** for the distributed model
- Two standard problems in parallel processing will be used to demonstrate these mechanisms:
  - Bounded buffer problem
  - Parallel matrix multiplication

Parallel Processing and Programming Languages (cont'd.)

- **Bounded buffer problem**: assumes that two or more processes are cooperating in a computational or input-output scenario
  - One process produces values consumed by the other, using an intermediate buffer or buffer process
  - No value must be produced until there is room to store it in the buffer
  - A value is consumed only after it has been produced
- This involves both communication and synchronization
- Also called the **producer-consumer problem**

Parallel Processing and Programming Languages (cont'd.)

- **Parallel matrix multiplication**: an algorithmic application in which the use of parallelism can cause significant speedups
- Matrices are essentially two-dimensional arrays
- Example: with both dimensions of size N:

```
typedef int Matrix [N][N];
Matrix a,b,c;
```

Parallel Processing and Programming Languages (cont'd.)

- The standard way of multiplying two matrices a and b to form a third matrix c is done using nested loops:

```
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        c[i][j] = 0;
        for (k = 0; k < N; k++)
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

- If performed sequentially, this computation requires N^3 steps

Parallel Processing and Programming Languages (cont'd.)

- If we assign a separate process to compute each `c[i][j]`, and each process executes on a separate processor, then the computation requires only N steps
  - This is the simplest form of the algorithm, because there are no write conflicts and no need to enforce mutual exclusion in accessing the matrices as shared memory
  - There is a synchronization problem to ensure that `c` is not used until all processes have completed

Parallel Programming without Explicit Language Facilities

- One approach to parallelism is simply not to express it explicitly in a language
- This is easiest in functional, logic, and object-oriented languages
  - They have a certain amount of inherent parallelism implicit in the language constructs
- Language translators may be able to automatically use operating system utilities to assign different processors to different parts of a program
  - Manual facilities are still needed to make full, optimal use of parallel processors

Parallel Programming without Explicit Language Facilities (cont'd.)

- Second approach: the translator may offer **compiler options** to allow explicit indication of areas where parallelism is called for
  - One of the most effective places is in the use of nested loops, since each repetition of the inner loop is relatively independent of the others
- Example: compiler option in FORTRAN

```
C$doacross share(a, b, c), local(j, k)
```

```
      integer a(100, 100), b(100, 100), c(100, 100)
      integer i, j, k, numprocs, err
      numprocs = 10
C code to read in a and b goes here
      err = m_set_procs(numprocs)
C$doacross share(a, b, c), local(j, k)
       do 10 i = 1,100
         do 10 j = 1,100
          c(i, j) = 0
          do 10 k = 1,100
             c(i, j) = c(i, j) + a(i, k) * b(k, j)
  10 continue
       call m_kill_procs
C code to write out c goes here
       end
```

Figure 13.3 FORTRAN compiler options for parallelism

Parallel Programming without Explicit Language Facilities (cont'd.)

- Third approach: provide a library of functions to perform parallel processing
  - This is a way of passing the facilities provided by an operating system directly to the programmer
- If a standard parallel library is required by a language, then it is the same as including parallel facilities in the language definition
- Example: using library functions in C to provide parallel processing for the matrix multiplication problem

```
#include <parallel/parallel.h>
#define SIZE 100
#define NUMPROCS 10

shared int a[SIZE][SIZE], b[SIZE][SIZE], c[SIZE][SIZE];

void multiply(){
    int i, j, k;
    for (i = m_get_myid(); i < SIZE; i += NUMPROCS)
        for (j = 0; j < SIZE; j++)
            for (k = 0 ; k < SIZE; k++)
                c[i][j] += a[i][k] * b[k][j];
}

main(){
    int err;
    /* code to read in the matrices a and b goes here */
    m_set_procs(NUMPROCS);
    m_fork(multiply);
    m_kill_procs();
    /* code to write out the matrix c goes here */
    return 0;
}
```

Figure 13.4 Use of a library to provide parallel processing

Parallel Programming without Explicit Language Facilities (cont'd.)

- Fourth approach: rely on operating system features directly to run programs in parallel
- Requires that a parallel program be split into separate, independently executable pieces that are set up to communicate via operating system mechanisms
  - Allows only program-level parallelism
- Example: use of **pipes** in Unix to string programs together

```
ls | grep "java"
```

Process Creation and Destruction

- A language with explicit mechanisms for parallel processing must have a construct for creating new processes
- Two basic ways to create new processes
- First approach: split the current process into two or more processes that continue to execute copies of the same program
  - One process is called the **parent**, while the others are called **children**
  - Processes can execute different code by testing some condition, but the basic program is the same

Process Creation and Destruction (cont'd.)

- This process resembles SIMD organization and is called **SPMD programming** (single program multiple data)
- SPMD programs may execute different segments of their common code
  - Do not necessarily operate synchronously, thus need process synchronization
- Second approach: a segment of code (usually a procedure) is explicitly associated with each new process
  - Different processes have different code

Process Creation and Destruction (cont'd.)

- This approach is called **MPMD programming**
- **Fork-join model**: a process creates several child processes, each with its own code (fork), and then waits for the children to complete (a join)
- **Granularity** of processes: relates to the size of the code that can become a separate process:
  - **Fine-grained**: individual statements can become processes and be executed in parallel
  - **Medium-grained**: procedures are assigned to processes
  - **Large-grained**: whole programs are processes

Process Creation and Destruction (cont'd.)

- Granularity can be an efficiency issue:
  - Many small-grained processes can incur significant overhead in creation and management
  - Large-grained processes may have difficulty exploiting all opportunities for parallelism
- Threads can be very efficient
  - Represent fine-grained or medium-grained parallelism without the overhead of full-blown process creation

Process Creation and Destruction (cont'd.)

- Two questions for process creation mechanisms:
  - Does the parent process suspend execution while its children are executing, or does it continue to execute alongside them?
  - What memory, if any, does a parent share with its children, or the children share among themselves?
- A parallel programming language also needs a method for process termination
  - Process may simply terminate when done
  - Process may need to wait for a condition to be met
  - Process may need to be selected for continuing

Statement-Level Parallelism

- **parbegin-parend** block: a construct for indicating parallel execution for statements
- Example: in Occam:

```
parbegin
  S1;
  S2;
  ...
  Sn;
parend;
```

  - Statements are all executed in parallel
  - Assumes that the main process is suspended
  - All processes of the `Si` share all variables not locally declared within an `Si`

Statement-Level Parallelism (cont'd.)

- **forall** construct in Fortran95: indicates parallel execution of each iteration of a loop
- Example:

```
      forall(i = 1:100, j = 1:100)
        c(i, j) = 0
        do 10 k = 1,100
                          c(i, j) = c(i, j) + a(i, k) * b(k, j)
10      continue
      end forall
```

Procedure-Level Parallelism

- When a procedure is associated with a process, the mechanism has this form:

```
x = newprocess(p);
...
...
killprocess(x);
```

  - p is a declared procedure
  - x is a process designator
- Alternate way is to use declarations to associated procedures to processes:

```
process x(p);
```

Procedure-Level Parallelism (cont'd.)

- Scope of x can be used as the region where x is active
  - x begins executing when the scope of its declaration is entered
  - x is terminated on exit from its scope (if not already completed)
- This is the method used by Ada
  - A process is called a **task** in Ada

Program-Level Parallelism

- In this method of process creation, only whole programs can become processes
  - Typically in MPMD style
  - A program creates a complete copy of itself
- Example: `fork` call of the UNIX operating system
  - Creates a child process that is an exact copy of the calling process, including all variables and environment data at the moment of the fork
  - The return value of the `fork` call indicates whether the process is the parent or child, which can be used to determine which code to execute

Program-Level Parallelism (cont'd.)

- Example:

```
if (fork() == 0)
  {/* ... child executes this part ...*/}
else
  {/* ... parent executes this part ...*/}
```

- A process can be terminated by a call to `exit`
- Process synchronization can be achieved by calls to `wait`, which suspends a parent until a child terminates
- Example: parallel matrix multiplication in C code:

Program-Level Parallelism (cont'd.)

```
#define SIZE 100
#define NUMPROCS 10

int a[SIZE][SIZE], b[SIZE][SIZE], c[SIZE][SIZE];

main(){
    int myid;
    /* code to input a,b goes here */
    for (myid = 0; myid < NUMPROCS; ++myid)
        if (fork() == 0){
            multiply(myid) ;
```

Figure 13.5 Sample C code for the fork construct (*continues*)

```
            exit(0) ;
        }
    for (myid = 0; myid < NUMPROCS; ++myid)
        wait(0) ;
        /* code to output c goes here */
    return 0;
}

void multiply (int myid){
    int i, j, k;
    for (i = myid; i < SIZE; i+= NUMPROCS)
        for (j = 0; j < SIZE; ++j){
            c[i][j] = 0;
            for (k = 0; k < SIZE; ++k)
                c[i][j] += a[i][k] * b[k][j];
    }
}
```

Figure 13.5 Sample C code for the fork construct

Threads

- Threads can be an efficient mechanism for fine- or medium-grained parallelism in the shared memory model
- Java has a widely used thread implementation

Threads in Java

- Threads are built into the Java Language
  - `Thread` class is part of `java.lang` package
  - Reserved word `synchronize` is used to establish mutual exclusion for threads
- To create a thread, instantiate a `Thread` object and define a `run` method in one of these ways:
  - Extend `Thread` through inheritance and override the empty `Thread.run` method
  - Instantiate an object of a class that implements the `Runnable` interface and pass it to the `Thread` constructor

Threads in Java (cont'd.)

- Example:

```
class MyThread extends Thread{

    public void run()
    { ... }
}

...

Thread t = new MyThread();

...
```

Threads in Java (cont'd.)

- Example:

```
class MyRunner implements Runnable{

    public void run()
    { ... }
}

...

MyRunner m = new MyRunner();
Thread t = new Thread(m);

...
```

Threads in Java (cont'd.)

- `start` method: begins running a thread and calls the `run` method
  - Main program continues to execute, but entire program will not finish execution until all of its threads complete
- `join` method: used to cause a thread to wait for another thread to finish before continuing

```
Thread t = new Thread(m);
t.start(); // t begins to execute
// do some other work
t.join(); // wait for t to finish
// continue with work that depends on t being finished
```

Threads in Java (cont'd.)

- `interrupt` method: sets an internal flag in the thread object that received the interrupt
  - Thread can test whether some other thread called its interrupt method
  - Allows a thread to continue to execute some cleanup code before actually exiting

```
Thread t = new Thread(m);
t.start(); // t begins to execute
// do some other work
t.interrupt(); // tell t that we are waiting for it
t.join(); // continue to wait
// continue
```

Threads in Java (cont'd.)

- Any method such as `join` that blocks the current thread has a timeout version
  - Java runtime system makes no attempt to discover or prevent deadlock; it is up to the programmer
- Java threads will share some memory or other resources by passing objects to be shared to constructors on the `Runnable` objects used to create threads

Threads in Java (cont'd.)

```
class Queue{

    ...
    public Object dequeue(){
        if (empty()) throw EmptyQueueException ;
        ...
    }

    public void enqueue(Object obj) { ... }
    ...
}

class Remover implements Runnable{

    public Remover(Queue q){ ... }
    public void run(){ ... q.dequeue() ... }
    ...
}
```

```
class Inserter implements Runnable{

    public Inserter(Queue q){ ... }
    public void run(){ ... q.enqueue(...) ... }
    ...
}

Queue q = new Queue(...);
...
Remover r = new Remover(q);
Inserter i = new Inserter(q);
Thread t1 = new Thread(r);
Thread t2 = new Thread(i);
t1.start();
t2.start();
...
```

- Queue `q` is shared between threads `t1` and `t2`
- Need a mechanism for ensuring mutual exclusion

Threads in Java (cont'd.)

- `synchronize` keyword is used for method synchronization
- Example:

```
class Queue{
    ...
    synchronized public Object dequeue(){
        if (empty()) throw EmptyQueueException ;
        ...
    }

        synchronized public void enqueue(Object obj){ ... }
        ...
}
```

Threads in Java (cont'd.)

- Every object in Java has a single **lock** available to threads
- When a thread attempts to execute a synchronized method, it must first acquire the lock on the object
  - If lock is held by another thread, it must wait
  - After exiting the method, it releases the lock
- `wait()` method: used to manually stall a thread based on a testable condition
- `notify()` or `notifyAll()` method: used to wake up a thread in a waiting condition

Threads in Java (cont'd.)

```
(1)  class Queue
(2)  {   ...
(3)      synchronized public Object dequeue()
(4)                      {  try  // wait can generate InterruptedException
(5)         {  while (empty()) wait();
(6)            ...
(7)         }
(8)                          catch (InterruptedException e) // reset interrupt
(9)         { Thread.currentThread().interrupt(); }
(10)     }
(11)     synchronized public void enqueue(Object obj)
                             // tell waiting threads to try again
(12)     { // add obj into the queue
(13)       ...
(14)       notifyAll();
(15)     }
(16)     ...
(17) }
```

A Bounded Buffer Example in Java

- Bounded buffer example: the producer reads characters from standard input and inserts them into a buffer
- The consumer removes characters from the buffer and writes them to standard output
- Additional requirements:
  - Producer thread will continue to read until an end of file is encountered, whence it will exit
  - Consumer thread will continue to write until the producer has ended and the buffer is empty

Semaphores

- **Semaphore**: a mechanism to provide mutual exclusion and synchronization in a shared-memory model
  - Is a shared integer variable that can be accessed only via three operations: **InitSem**, **Signal**, and **Delay**
- **Delay** operation: tests the semaphore for a positive value
  - Decrements it if it is positive
  - Suspends the calling process if it is zero or negative

Semaphores (cont'd.)

- **Signal** operation: tests whether processes are waiting
  - Causes one of them to continue if so
  - Increments the semaphore if not
- **Signal** is analogous to `notify` in Java, and **Delay** is analogous to `wait`
- The system must ensure that each of these operations executes **atomically** (by only one process at a time)

Semaphores (cont'd.)

- Can ensure mutual exclusion by defining a critical region
- **Critical region**: a region of code that can be executed by only one process at a time
- Example:

```
Delay(S);
{critical region}
Signal(S);
```

- Semaphores are sometimes called **locks**
  - Can also be used to synchronize processes

Semaphores (cont'd.)

- An important question is the method used to choose a suspended process for continued execution when a call to Signal is made
- Possibilities include:
  - Making a random choice
  - Using a first in, first out strategy
  - Using some sort of priority system
- This choice has a major effect on the behavior of concurrent programs using semaphores

Semaphores (cont'd.)

```
class Semaphore{

    private int count;

    public Semaphore(int initialCount){
       count = initialCount;
    }

    public synchronized void delay() throws InterruptedException{
        while (count <= 0) wait();
        count--;
    }

    public synchronized void signal(){
        count++;
        notify();
    }
}
```

Figure 13.7 Simplified Java code for a semaphore

A Bounded Buffer Using Semaphores

- Semaphores can be used to enforce synchronization and mutual exclusion
- Use of semaphores can eliminate the need for Java synchronization mechanisms:
  - Synchronized methods
  - Calls to `wait` and `notifyAll`

A Bounded Buffer Using Semaphores (cont'd.)

```
(1)  class Buffer{
(2)      private final char[] buf;
(3)      private int start = -1;
(4)      private int end = -1;
(5)      private int size = 0;
(6)      private Semaphore nonFull, nonEmpty, mutEx;

(7)      public Buffer(int length) {
(8)          buf = new char[length];
(9)          nonFull = new Semaphore(length);
(10)         nonEmpty = new Semaphore(0);
(11)         mutEx = new Semaphore(1);
(12)     }

(13)     public boolean more()
(14)     { return size > 0; }
```

```
(15)     public void put(char ch){
(16)      try {
(17)         nonFull.delay();
(18)         mutEx.delay();
(19)         end = (end+1) % buf.length;
(20)         buf[end] = ch;
(21)         size++;
(22)         mutEx.signal();
(23)         nonEmpty.signal();
(24)       }
(25)       catch (InterruptedException e)
(26)       { Thread.currentThread().interrupt(); }
(27)     }
(28)     public char get(){
(29)      try{
(30)         nonEmpty.delay();
(31)         mutEx.delay();
(32)         start = (start+1) % buf.length;
(33)         char ch = buf[start];
(34)         size--;
(35)         mutEx.signal();
(36)         nonFull.signal();
(37)         return ch;
(38)       }
(39)       catch (InterruptedException e)
(40)       { Thread.currentThread().interrupt(); }
(41)       return 0;
(42)     }
(43) }
```

Difficulties with Semaphores

- Although semaphores themselves are protected, there is no protection against their incorrect use by programmers
- Example: failure to create a critical region:

```
Signal(S);
...
Delay(S);
```

- Example: this process will likely block at the second Delay and never resume execution:

```
Delay(S);
...
Delay(S);
```

Difficulties with Semaphores (cont'd.)

- Example: can cause deadlock:
  - If Process 1 executes `Delay(S1)` at the same time that Process 2 executes `Delay(S2)`, then each will block waiting for the other to issue a Signal
  - Deadlock has occurred
- The monitor was invented to remove some of these insecurities in the use of semaphores

Process 1: *Delay*(S_1);
Delay(S_2);
. . .
Signal(S_2);
Signal(S_1);

Process 2: *Delay*(S_2);
Delay(S_1);
. . .
Signal(S_1);
Signal(S_2);

Implementation of Semaphores

- Semaphores are generally implemented with some form of hardware support
- Example: TestAndSet machine instruction on a single-processor system tests a memory location and simultaneously increments or decrements the location if the test succeeds
- Assuming TestAndSet returns the value of its location parameter and decrements its location parameter if it is > 0, we can implement Signal and Delay with this code:

```
Delay(S): while TestAndSet(S) <= 0 do {nothing};
Signal(S) : S : = S + 1;
```

Implementation of Semaphores (cont'd.)

- This implementation causes a blocked process to **busy-wait** or **spin** in a while loop until S becomes positive again through a call to Signal by another process
  - Semaphores implemented this way are sometimes called **spin-locks**
- Leaves unresolved the order in which waiting processes are reactivated
  - May be random, or in some order imposed by the operating system

Implementation of Semaphores (cont'd.)

- A waiting process may be preempted by many incoming calls to Delay from new processes, and never get to execute despite calls to Signal
  - This situation is called **starvation**
- Starvation is prevented by using a scheduling system that is **fair**
  - Guarantees that every process will execute within a finite period of time
- Avoiding starvation is more difficult than avoidance of deadlocks

Implementation of Semaphores (cont'd.)

- Modern shared-memory systems often provide facilities for semaphores that do not require busy-waiting
  - Semaphores are special memory locations accessible by only one processor at a time, with a queue to store the processes that are waiting for it

Monitors

- **Monitor**: an abstract data type mechanism with the added property of mutual exclusion
  - It encapsulates shared data and operations on these data
  - At most, one process at a time can be using any of the monitor's operations
- Monitor has an associated wait queue to track processes that are waiting to use its operations

Monitors (cont'd.)

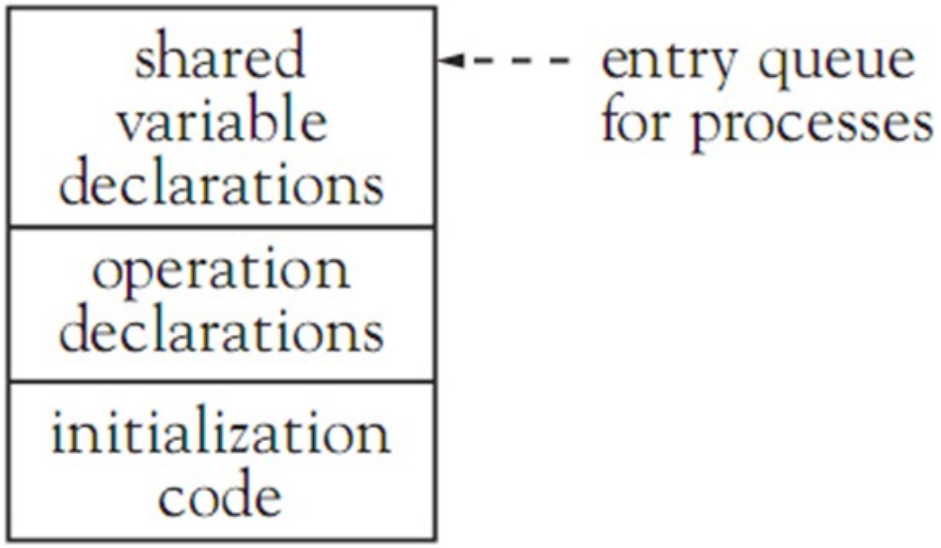


Figure 13.9 The structure of a monitor

Monitors (cont'd.)

- Monitor must also provide condition variables to handle situations like the bounded buffer problem
- **Condition variables**: shared variables within the monitor resembling semaphores
  - Each has a queue of processes waiting for the condition
  - Each has associated **suspend** and **continue** operations
- If a condition queue is empty, a call to continue will have no effect
- Suspend always suspends the current process

Monitors (cont'd.)

- When a continue call is issued to a waiting process by a process currently in the monitor, there are two choices:
  - Suspended process just awakened by the continue call must wait until the calling process has left the monitor
  - Process that issues the continue call must suspend until the awakened process has left the monitor
- It is possible to imitate the behavior of a monitor using semaphores

Monitors (cont'd.)

- Monitors and semaphores are equivalent in terms of the kinds of parallelism they can express
- Monitors provide a more structured mechanism for concurrency than semaphores, and they ensure mutual exclusion
  - They cannot guarantee the absence of deadlock
- Java and Ada have monitor-like mechanisms

Java Synchronized Objects as Monitors

- Java objects whose methods are synchronized are essentially monitors
  - Sometimes called **synchronized objects**
- Java provides an entry queue for each synchronized object
  - A thread inside the synchronized object has a lock on the object
- These Java queues do not operate in a fair fashion
- Any unsynchronized methods may be executed without acquiring the lock or going through the entry queue

Java Synchronized Objects as Monitors (cont'd.)

- Java's synchronized objects do not have separate condition variables
  - Only one wait queue per synchronized object for any and all conditions
- Java `wait` and `sleep` calls are suspend operations
- Java `notify` and `notifyAll` calls are continue operations
- When a thread comes off the wait queue, it must again acquire the object's lock
  - Awakened threads in Java must wait for the awakening thread to exit the synchronized code

The Java `Lock` and `Condition` Interfaces

- `java.util.concurrent.locks` package: includes interfaces and classes that support a more authentic monitor mechanism
  - Programmer represents a monitor as an explicit lock object associated with a shared data resource
- `lock` method: acquires the lock
- `unlock` method: releases the lock
- `await` method: wait for an explicit condition object
- Multiple condition objects can be associated with a single lock

The Java `Lock` and `Condition` Interfaces (cont'd.)

- Each condition object has its own queue of threads waiting on it
- When a method is finished, it can signal other threads waiting on a condition object by using the `signal` or `signalAll` methods

The Java `Lock` and `Condition` Interfaces (cont'd.)

```
import java.concurrent.locks.*;

class Buffer{

    private final char[] buf;
    private int start = -1;
    private int end = -1;
    private int size = 0;

    private Lock lock;                    // The lock for this resource
    private Condition okToGet;            // Condition with queue of
                                             waiting readers
    private Condition okToPut;            // Condition with queue of
                                             waiting writers

    public Buffer(int length){
        buf = new char[length];
        lock = new ReentrantLock();       // Instantiate the lock and its
                                             two conditions
        okToGet = lock.newCondition();
```

The Java `Lock` and `Condition` Interfaces (cont'd.)

```
        okToPut = lock.newCondition();

    }

    public void put(char ch){              // Not synchronized on the Buffer
                                              object,
        ...                                // but unlocks and locks its lock
                                              instead
    }

    public char get(){
        ...
    }
}
```

Ada95 Concurrency and Monitors

- Concurrency in Ada is provided by independent processes called **tasks**
- A task is declared using specification and body declaration similar to the package mechanism

```
task T; -- task specification
-- see next section for more elaborate versions of this

task body T is
-- declarations
begin
-- code executed when task runs
end;
```

Ada95 Concurrency and Monitors (cont'd.)

- An Ada task begins to execute as soon as the scope of its declaration is entered
- When the end of the scope of the task declaration is reached, the program waits for the task to terminate before continuing execution
- Can declare task types and variables

```
task type T is
...
end;

task body T is
...
begin
...
end T;

p,q: T;
```

Ada95 Concurrency and Monitors (cont'd.)

- Ada95 has monitors, called **protected objects**
  - Correspond to synchronized objects in Java

```
protected type Queue is -- specification
   procedure enqueue (item: in ...);
   entry dequeue (item: out ...);
   function empty return boolean;
private:
   -- private data here
   size: natural;
   ...
end Queue;

protected body Queue is
  -- implementation of services here
end Queue;
```

Ada95 Concurrency and Monitors (cont'd.)

- Three kinds of operations within a protected object:
  - Functions
  - Procedures
  - **Entries**
- Functions may not change the local state of a protected object but can be executed by any number of callers
- Procedures and entries may only be executed by a single caller at a time
- No functions can execute simultaneously with a procedure or entry

Ada95 Concurrency and Monitors (cont'd.)

```
protected body Queue is
   procedure enqueue (item: in ...) is
   begin
      ...
   end enqueue;

   entry dequeue (item: out ...) when size > 0 is
   begin
      ...
   end dequeue;

   function empty return boolean is
   begin
       return size > 0;
   end empty;

end Queue;
```

Ada95 Concurrency and Monitors (cont'd.)

- A procedure can always be executed, while an entry can only be executed under a certain condition, called the **entry barrier**
  - If the entry barrier is closed (false), the task is suspended and placed in a wait queue for that entry
- Ada-protected object entries correspond to monitor condition variables
- Ada runtime system automatically recomputes the entry barrier at appropriate times and wakes a waiting task

Message Passing

- **Message passing**: a mechanism for process synchronization and communication using the distributed model of parallel processing
- In its most basic form, it consists of two operations, send and receive
- Example: in C code:

```
void send(Process to, Message m);
void receive(Process from, Message m);
```

  - Assumes every sender knows its receiver and vice versa
  - Must have names within scope of each other

Message Passing (cont'd.)

- Less restrictive form of send and receive removes the requirement to name sender and receiver
- Example: in C code:

```
void send(Message m);
void receive(Message m);
```

  - A sent message will go to any process willing to receive it, and a message will be received from any sender
- Commonly, a name is required for the send but not for the receive

Message Passing (cont'd.)

- Other questions to be answered revolve around synchronization of processes communicating with send and receive
  - Must a sender wait for a receiver to be ready before sending a message, or can the sender continue to execute even if there is no available receiver? If so, are messages stored in a buffer for later receipt?
  - Must a receiver wait until a message is available to be sent, or can a receiver receive a null message and continue to execute?

Message Passing (cont'd.)

- **Rendezvous** mechanism: when both sender and receiver must wait until the other is ready
  - If messages are buffered, must determine if there is a size limit on the buffer, and who manages the buffer
- **Mailbox** mechanism: a separate process manages the buffer, where processes drop off and receive messages from named (or numbered) mailboxes
  - Mailbox may be assigned an **owner** process that manages it

Message Passing (cont'd.)

- **Control facilities**: permit processes to test for messages, to accept messages only on certain conditions, and to select specific messages
- Various forms of message passing include:
  - Hoare's **Communicating Sequential Processes (CSP)** language framework and **occam** language
  - **Remote Procedure Call** (**RPC**) or **Remote Method Invocation** (**RMI**) in Java
  - **CORBA** (Common Object Request Broker Architecture)
  - **COM** (Common Object Model)

Task Rendezvous in Ada

- Ada tasks can pass messages to each other via a rendezvous mechanism
- **Task entries**: define rendezvous points

```
task userInput is
    entry buttonClick (button: out ButtonID);
    entry keyPress (ch: out character);
end userInput;
```

- A task exports entry names to the outside world
- Entries do not have code bodies but must appear inside an `accept` statement that provides the code to be executed

Task Rendezvous in Ada (cont'd.)

- `Accept` statements can only appear inside the body of a task
  - Caller of an entry waits for the task to reach a corresponding `accept` statement
  - A task that reaches an `accept` statement waits for a corresponding call
- Each entry has an associated queue to maintain processes that are waiting for an `accept` statement to be executed
  - This is a FIFO queue

Task Rendezvous in Ada (cont'd.)

```
task body userInput is
begin
    ...
    -- respond to a button click request:
    accept buttonClick (button: out ButtonID) do
       -- get a button click while the caller waits
    end buttonClick; -- caller can now continue
    -- continue with further processing
    ...
    -- now respond to a keypress request:
    accept keyPress (ch: out character) do
      -- get a keypress while the caller waits
    end keyPress; – caller can now continue
    -- continue with further processing
end userInput;
```

Task Rendezvous in Ada (cont'd.)

- A task that operates as a server for other client tasks will maintain a set of entries that it will accept and wait for one of these entries to be called
- A `select` statement is used for the set of entries
  - All conditions in the `select` statement are evaluated
  - Those that are true have the corresponding select alternatives tagged as **open**
  - An open accept statement is selected for execution if another task has executed an entry call for its entry

```
(1)  task body userInput is
(2)  begin
(3)    loop
(4)      select
(5)        when buttonClicked =>
(6)          -- respond to a button click request:
(7)                    accept buttonClick (button: out ButtonID) do
(8)                      -- get and return a button click
(9)                      -- while the caller waits
(10)         end buttonClick; -- caller can now continue
(11)         -- do some buttonClick cleanup
(12)       or when keypressed =>
(13)         -- now respond to a keypress request:
(14)                   accept keyPress (ch: out character) do
(15)                     -- get a keypress while the caller waits
(16)                   end keyPress; -- caller can now continue
(17)         -- do some keyPress cleanup
(18)       or terminate;
(19)     end select;
(20)   end loop;
(21) end userInput;
```

Task Rendezvous in Ada (cont'd.)

- Termination of tasks in Ada can occur in two ways:
  - Task executes to completion and has not created any child processes that are still executing
  - Task is waiting with an open terminate alternative in a select statement, and its master has executed to completion
- **Master**: the block of the parent task in which a task was created
  - All tasks created by the same master must also have terminated or be waiting at a terminate alternative (they will all terminate simultaneously)

Task Rendezvous in Ada (cont'd.)

- Tasks cannot have functions that return values
  - Must return values through parameters in entries

```
task buf is
  entry insert(ch: in character);
  entry delete(ch: out character);
  entry more (notEmpty: out boolean);
end;
task body buf is
   MaxBufferSize: constant integer := 5;
   store: array (1..MaxBufferSize) of character;
   bufferStart: integer := 1;
   bufferEnd: integer := 0;
   bufferSize: integer := 0;
begin
   loop
     select
       when bufferSize < MaxBufferSize =>
          accept insert(ch: in character) do
             bufferEnd := bufferEnd mod MaxBufferSize + 1;
             store(bufferEnd) := ch;
```

Figure 13.11 A bounded buffer as an Ada task (*continues*)

```
            end insert;
            bufferSize := bufferSize + 1;
         or when bufferSize > 0 =>
            accept delete(ch: out character) do
               ch := store(bufferStart);
            end delete;
            bufferStart := bufferStart mod MaxBufferSize + 1;
            bufferSize := bufferSize - 1;
         or
            accept more(notEmpty: out boolean) do
                 notEmpty := bufferSize > 0;
            end more;
         or terminate;
      end select;
   end loop;
end buf;
```

Figure 13.11 A bounded buffer as an Ada task

```
            end insert;
            bufferSize := bufferSize + 1;
         or when bufferSize > 0 =>
            accept delete(ch: out character) do
               ch := store(bufferStart);
            end delete;
            bufferStart := bufferStart mod MaxBufferSize + 1;
            bufferSize := bufferSize - 1;
         or
            accept more(notEmpty: out boolean) do
                 notEmpty := bufferSize > 0;
            end more;
         or terminate;
      end select;
   end loop;
end buf;
```

Figure 13.11 A bounded buffer as an Ada task

```
task type Semaphore is
  entry initSem (n: in integer);
  entry wait; -- use wait instead of delay, which is reserved in Ada
  entry wait;
end;
task body Semaphore is
  count : integer;
begin
  accept initSem (n: in integer) do
   count := n;
  end initSem;
  loop
   select
    when count > 0 =>
     accept wait;
     count := count - 1 ;
   or
    accept signal;
    count := count + 1;
   or
    terminate;
   end select;
  end loop;
end Semaphore;
```

Figure 13.12 A semaphore task type in Ada

Parallelism in Non-Imperative Languages

- Parallel processing in functional or logic programming languages is still in an experimental and research phase
- A number of good implementations of research proposals exist, including MultiLisp, Qlisp, Parlog, and FGHC
- Erlang: a non-imperative language that supports message-passing

And-Parallelism

- In and-parallelism, a number of values are computed in parallel by child processes, while the parent process waits for them to finish and return their values
- Example: a process calls a function in Lisp:

```
(f a b c d e)
```

  - Can create six parallel processes to compute the values of `f, a, … e`
  - Suspends execution until all values are computed
  - Then calls the (function) value of `f` with the returned values of `a` through `e` as arguments

And-Parallelism (cont'd.)

- In a let-binding: `(let ((a e1) (b e2) (c e3)) (...))`
  - The values of `e1`, `e2`, and `e3` can be computed in parallel
- In Prolog: `q :- p1, p2, ..., pn.`
  - `p1` through `pn` can be executed in parallel
  - `q` succeeds if all the `pi` succeed
- Noninterference is guaranteed since the evaluation of arguments in a purely functional language causes no side effects
- Most functional languages are not pure, however

And-Parallelism (cont'd.)

- Example: in Prolog:

```
process(N,Data) :-
            M is N - 1,
            Data = [X|Data1],
            process(M,Data1).
```

  - Cannot be executed in parallel since each of the first two goals contribute instantiations to the last
  - Synchronization is needed here

Or-Parallelism

- In or-parallelism, execution of several alternatives can occur in parallel, with the first to finish (or succeed) causing all other alternative processes to be ignored (and to terminate)
- Example: in Lisp: `(cond (p1 e1) (p2 e2)...(pn en))`
- Or-parallelism makes the `cond` into a nondeterministic construct, which may change the overall behavior of the program if the order of evaluation is significant

Or-Parallelism (cont'd.)

- In Prolog, a system may try to satisfy alternative clauses for a goal simultaneously
- Example: if there are two or more clauses for the same predicate:

```
p(X) :- q(X).
p(X) :- r(X).
```

  - The correct execution of a program may depend on the order in which the alternatives are tried
- Synchronization and order problems are difficult to solve automatically by a translator
- Some language designers have included some manual parallel constructs

Parallelism in Lisp

- And-parallelism is most often implemented in Lisp
- MultiLisp offers parallel evaluation of function calls with the `pcall` construct: `(pcall f a b c...)`
  - Provides parallel evaluation of its subexpressions
- **Future**: a construct that returns a point to the value of a not-yet-finished parallel computation
- Example: in MultiLisp: `(pcall f (future a) (future b)...)`
  - Allows execution of `f` to proceed before the values of `a` and `b` have been computed
  - When the values are needed, it suspends evaluation until they are available

Parallelism in Prolog

- The more natural form for Prolog is or-parallelism, although both forms have been implemented
- Or-parallelism fits well with the semantics of logic programming
  - Often performed automatically by a parallel Prolog system
- One version of and-parallelism uses guarded Horn clauses to eliminate backtracking
- Example:

```
h :- g1, ..., gn | p1, ..., pm.
```

  - The `g1, …gn` are **guards** that are executed first and prohibited from establishing instantiations

Parallelism in Prolog (cont'd.)

- If the guards succeed, the system **commits** to this clause for `h`, and the `p1,…pm` are executed in parallel
  - Such a system is FGHC (flat guarded Horn clauses)
- Example: FGHC program

```
generate(N,X) :- N = 0 | X = [].
generate(N,X) :- N > 0 | X = [N|T], M is N-1,
                 generate(M,T).
```

Parallelism in Prolog (cont'd.)

- FGHC places severe constraints on variable instantiation and thus on unification
  - A significant reduction in the expressiveness of the language
- **Variable annotations**: specify the kind of instantiations allowed and when they may occur
- **Parlog**: a language that does this
- **Mode declaration**: distinguishes input variables from output variables
  - "?" for input and "^" for output

Parallelism in Prolog (cont'd.)

- Example: quicksort in Parlog:

```
mode qsort (P?,S^).
qsort([],[]).
qsort([H|T] S) :- partition(H,T,L,R),
                                    qsort(L,L1),
                                    qsort(R,R1),
                                    append(L1,[ H|R],S).
mode partition(P?,Q?,R^,S^).
partition(P,[A|X],[A|Y],Z) :- A < P:
                                          partition(P,X,Y,Z).
partition(P,[A|X],Y,[A|Z]):- A >= P:
                                          partition(P,X,Y,Z).
partition(P,[], [], []).
```

Parallelism in Prolog (cont'd.)

- Parlog selects an alternative from among clauses by the usual unification process and also by using guards
  - It then commits to one alternative

Parallelism with Message Passing in Erlang

- Erlang: developed by Ericcson for distributed, fault-tolerant, real-time applications needed in the telecommunications industry
  - Uses strict evaluation and dynamic typing
  - Its syntax is similar to that of Haskell, with powerful pattern-matching capability
- Erlang's additional features include:
  - Variable binding via single assignment
  - Distinguishes between atoms and variables
  - Supports concurrency and parallelism based on message passing

Parallelism with Message Passing in Erlang (cont'd.)

- Absence of side effects and the presence of message passing produces a very simple model of parallel programming in Erlang
- Erlang implementation includes an interactive interpreter and a compiler

Parallelism with Message Passing in Erlang (cont'd.)

- Example: in Erlang code:

```
1> Rectangle = {rectangle, 8, 4}
{rectangle, 8, 4}
2> Circle = {circle, 3.5}
{circle, 3.5}
3> {rectangle, Width, Height} = Rectangle.
{rectangle, 8, 4}
4> Width.
8
5> {circle, Radius} = Circle.
{circle, 3.5}
6> Radius.
3.5
```

Parallelism with Message Passing in Erlang (cont'd.)

- Pattern matching and variable binding form the basis of function applications in Erlang
  - Example: creates a `geometry` module:

```
-module(geometry).
-export([area/1]).
area({rectangle, Width, Height}) -> Width * Height;
area({circle, Radius}) -> 3.14159 * Radius * Radius.
```

  - Example: compiles and tests the `geometry` module

```
1> c(geometry).
{ok, geometry}
2> geometry:area({rectangle, 8, 4}).
32
3> geometry:area({circle, 3.5}).
38.4844775
```

Parallelism with Message Passing in Erlang (cont'd.)

- Functions can be passed as arguments to other functions, with mapping:

```
1> lists:map(fun(N) -> 2 * N end, [2, 4, 6]).
[4, 8, 12]
2> Double = fun(N) -> 2 * N end.
#Fun<erl_eval.6.57006448>
3> lists:map(Double, [2, 4, 6]).
[4, 8, 12]
4> lists:map(fun math:sqrt/0, [4, 9, 16]).
[2, 3, 4]
```

- `fun`: a special type of function, used like lambda to build an anonymous function to pass as an argument

Parallelism with Message Passing in Erlang (cont'd.)

- An Erlang process is a lightweight software object that runs independently of any other process
  - Communicates with other processes via messages
- **Spawn**: creates a new process
- **Send**: sends messages to other processes
- **Receive**: receives messages
- A function can act as a **server** that executes in its own process
  - A **client** makes a request of this server

Parallelism with Message Passing in Erlang (cont'd.)

Table 13.1 The essential process operations in Erlang

Operation	What It Does
`Pid = spawn(Fun)`	Creates a new process to execute the function `Fun` in parallel with the caller and returns the process identifier for this process. `Pid` can then be used to send messages to the new process.
`Pid ! message`	Sends `message` (a pattern) to the process identified by `Pid`. The sender does not wait, but continues its execution.

`receive`
`  Pattern1 [ when Guard1 ] ->`
`  Expressions1`
`  Pattern2 [ when Guard2 ] ->`
`  Expressions2`
`  . . .`
`end`
	Receives a message (a pattern). If the message matches one of the given patterns, the corresponding code is executed. Otherwise, the message is saved for later processing.

```
-module(area_server).
-export([area/2, loop/0]).

area(Pid, shape) ->
   Pid ! {self(), shape},
   receive
      {Pid, Result} ->
         Result
   end.

loop() ->
   receive
      {From, {rectangle, Width, Height}} ->
         From ! {self(), Width * Height},
         loop();
      {From, {circle, Radius}} ->
         From ! {self(), 3.14159 * Radius * Radius},
         loop();
      {From, Other} ->
         From ! {self(), error, Other},
         loop()
   end.
```

Parallelism with Message Passing in Erlang (cont'd.)

- Interactions between processes are not always tightly coupled
  - A process may hand off work to other processes and go on without waiting for a response
  - A process may wait longer than is desirable to receive a message (if the server has a problem)
- Receiver process can include an `after` clause to time out the `receive` operation

Parallelism with Message Passing in Erlang (cont'd.)

- Example: `receive` operation:

```
receive
   Pattern1 [ when Guard1 ] ->
      Expressions1
   Pattern2 [ when Guard2 ] ->
      Expressions2
   . . .
   after Time ->
      Expressions
end
```

- **Mailbox**: a queue-like data structure associated with a process

Parallelism with Message Passing in Erlang (cont'd.)

- To remove a message from its mailbox, the process's receive operation must be able to match the message to one of its patterns
  - It a match occurs, the message is removed from the mailbox and discarded, and the expressions in the clause are executed
  - If not, the message is removed and put at the rear of the process's **save queue**
- If the mailbox becomes empty or a timeout occurs, the process is suspended until a new message arrives

Parallelism with Message Passing in Erlang (cont'd.)

- When a new message arrives, the process itself is rescheduled for execution to examine the message
  - If a match, all messages in the saved queue are transferred to the mailbox in their original order
- Erlang includes resources for developing distributed applications to run on a network of independent computers
- Independent processes are embedded in software objects called nodes, which allow messages to be sent and received across a network of machines