# Parallel Programming

## Parallel Programming

- In the early systems, many processes shared a single processor, thus appearing to execute simultaneously. Pseudo-parallelism of this sort represented a considerable advance in computer technology.

## Multi Processor and distributed system

- In true parallelism, many processors are connected to run in concert, either as a single system incorporating all the processors (a **multiprocessor system**) or as a group of stand-alone processors connected together by high-speed links (a **distributed system**).

## Network of computers

- The situation has become even more complex with the advent of high-speed networks, including the Internet, in that physically distant computers are now capable of working together, using the same or similar programs to handle large problems. Thus, organized networks of independent computers can also be viewed as a kind of distributed system for parallel processing, and, indeed, may have already become more important than traditional styles of parallel computing

# Programming languages

- Programming languages have affected and been affected by the parallelism in a number of ways.

# Programming Languages

- Programming languages have been used to express algorithms to solve the problems presented by parallel processing systems.
- Programming languages have been used to write operating systems that have implemented these solutions on various architectures.
- Programming languages have been used to harness the capabilities of multiple processors to solve application problems efficiently.

## Programming Languages

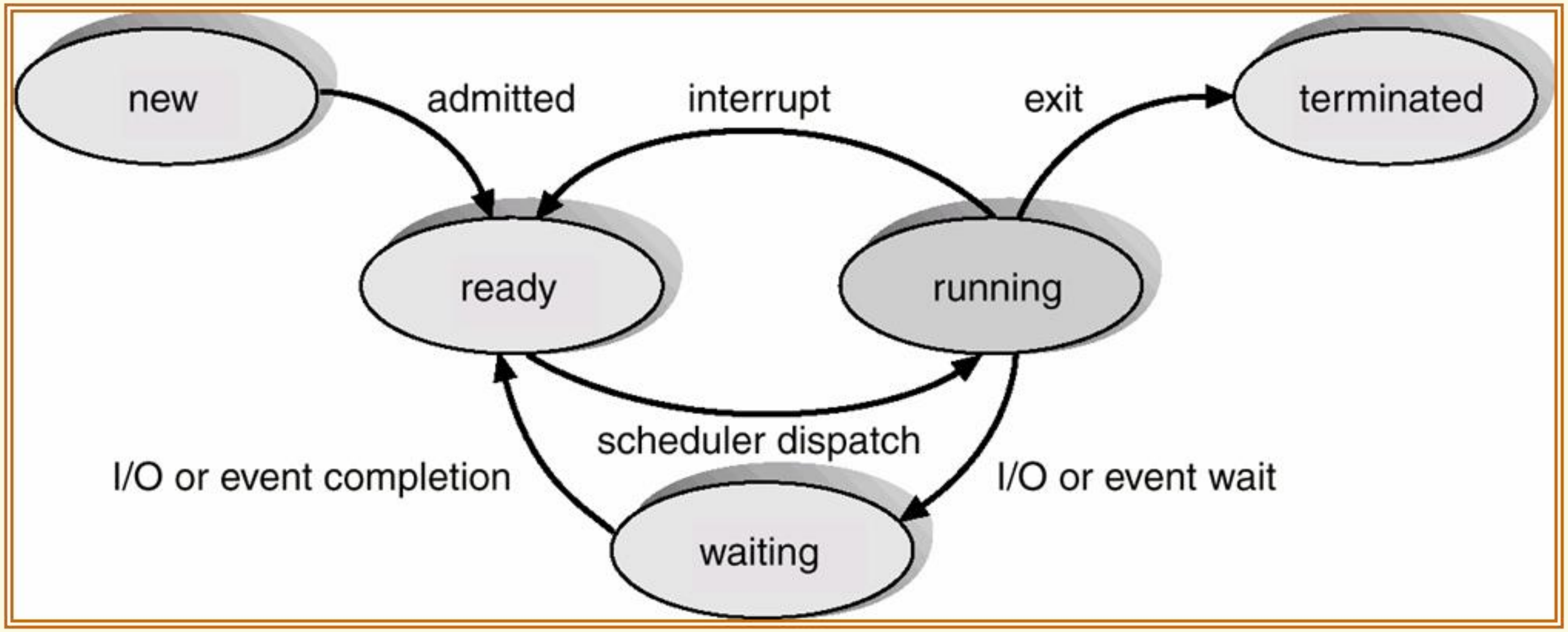
- Programming languages have been used to implement and express communication across networks.

## Introduction to Parallel Processing

- A process is an instance of a program or program part that has been scheduled for independent execution.

## Process

Process state diagram



Heavy weight processes
Light weight processes

## Processors

- Two primary requirements for the organization of the processors are
- **1.** There must be a way for processors to synchronize their activities.
- **2.** There must be a way for processors to communicate data among themselves.

## Synchronization and Communication

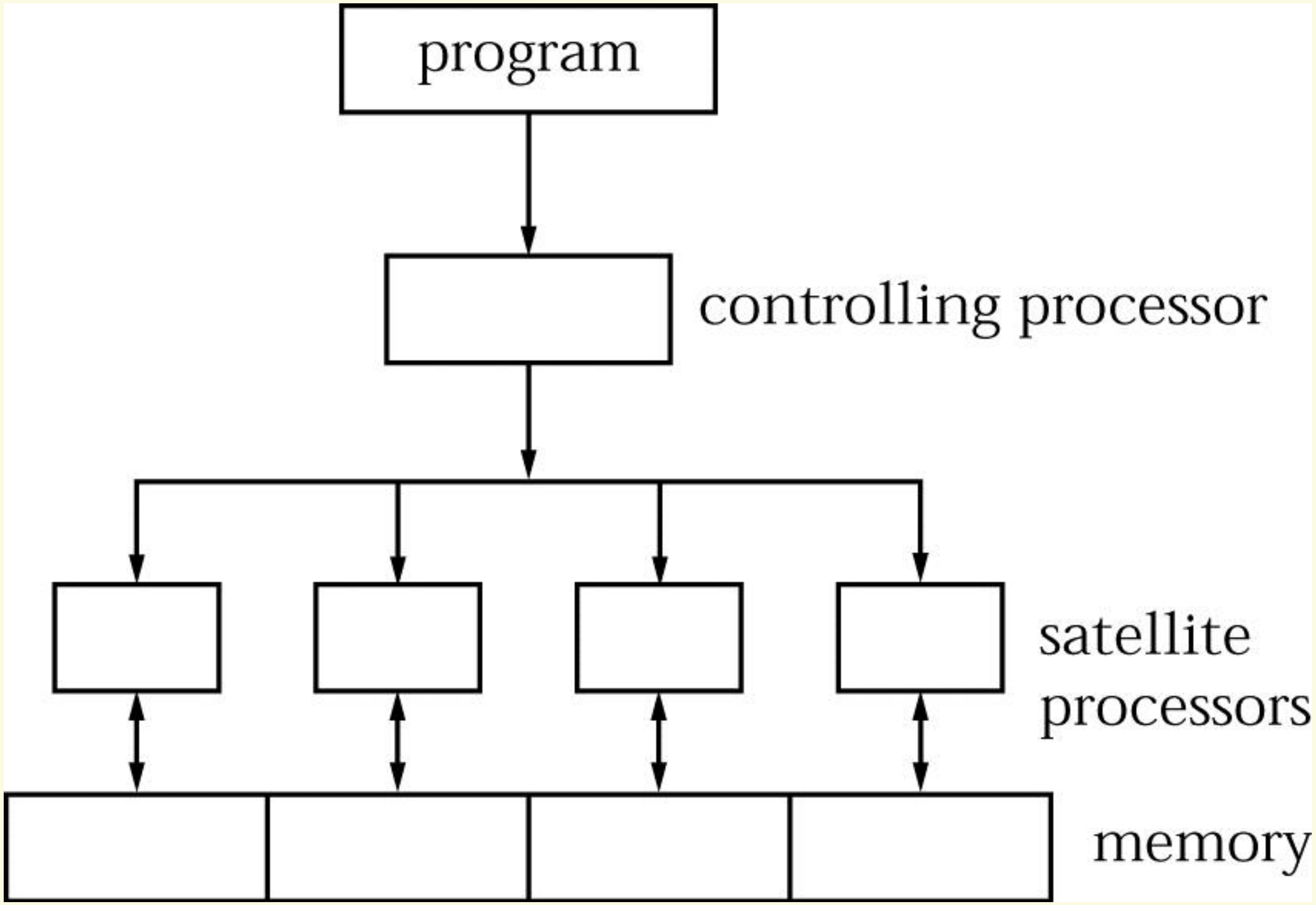
- For example, in a typical situation, one processor will be handling the input and sorting of data, while a second processor performs computations on the data. The second processor must not begin processing data before it is sorted by the first processor. This is a synchronization problem. Also, the first processor needs to communicate the actual sorted data to the second processor. This is a communication problem.

## SIMD

- In some machines one processor is designated as a controller, which manages the operation of the other processors. In some cases this central control extends even to the selection of instructions, and all the processors must execute the same instructions on their respective registers or data sets. Such systems are called **single-instruction**, **multiple-data (SIMD) systems** and are by their nature multiprocessing rather than distributed systems.

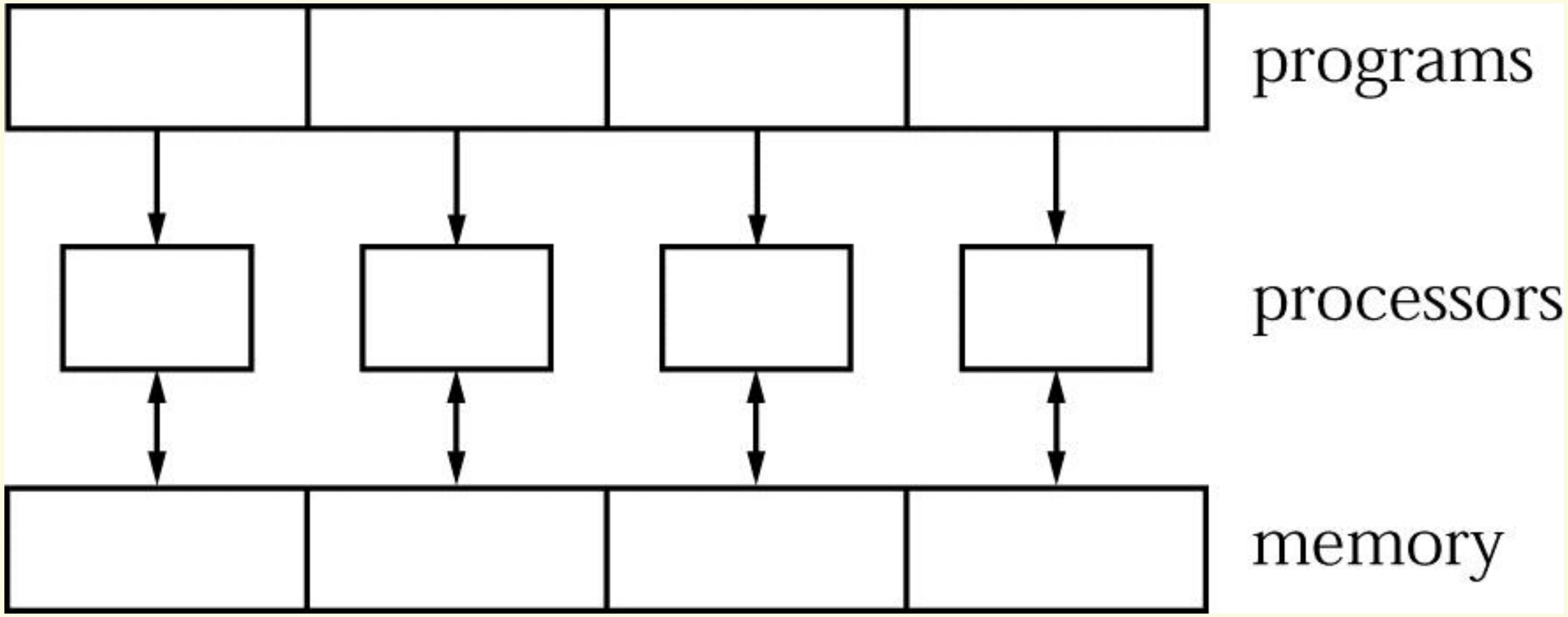
## SIMD

- SIMD systems are also often synchronous, in that all the processors operate at the same speed and the controlling processor determines precisely when each instruction is executed by each processor. This implicitly solves the synchronization problem.


program
controlling processor
satellite processors
memory

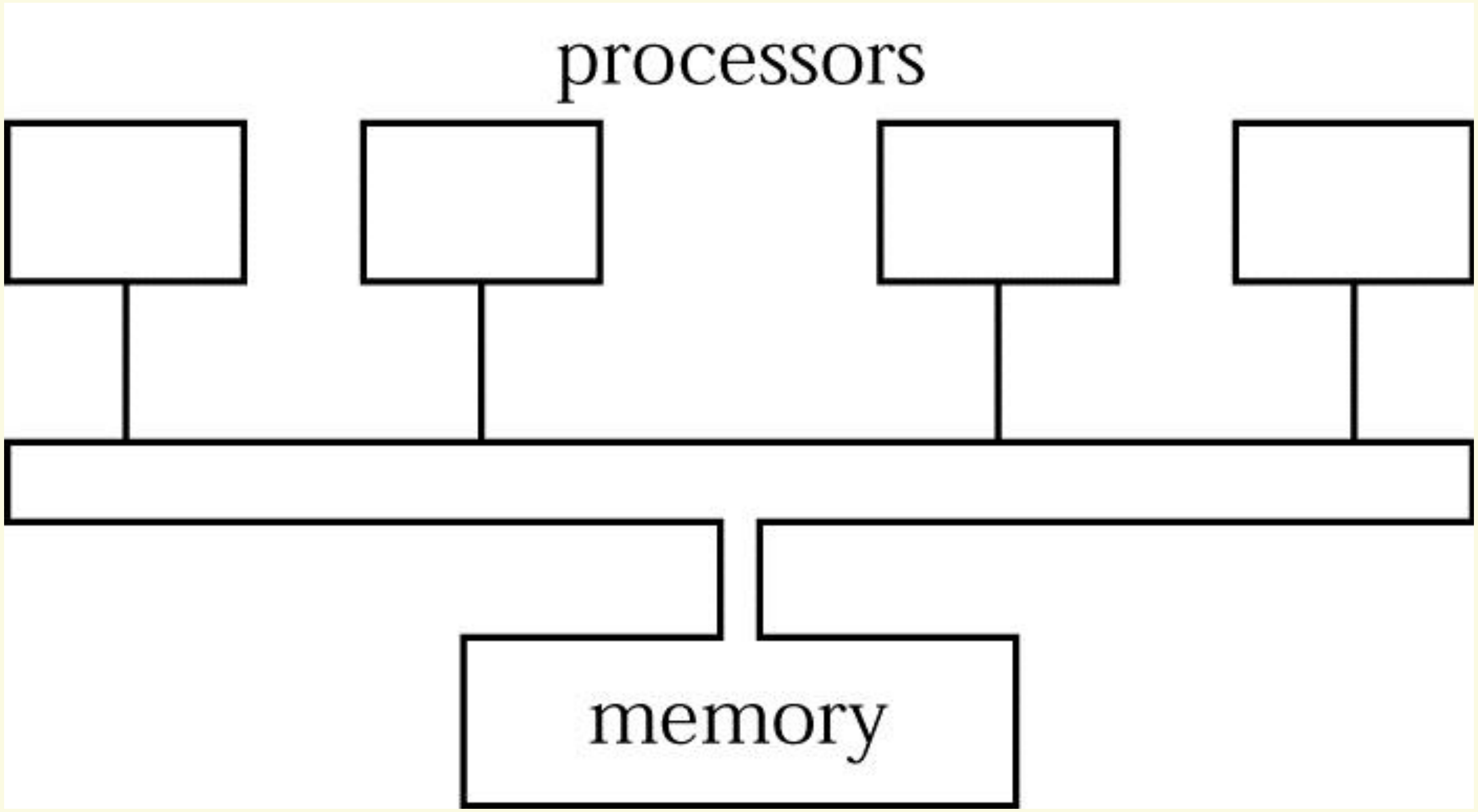
## MIMD

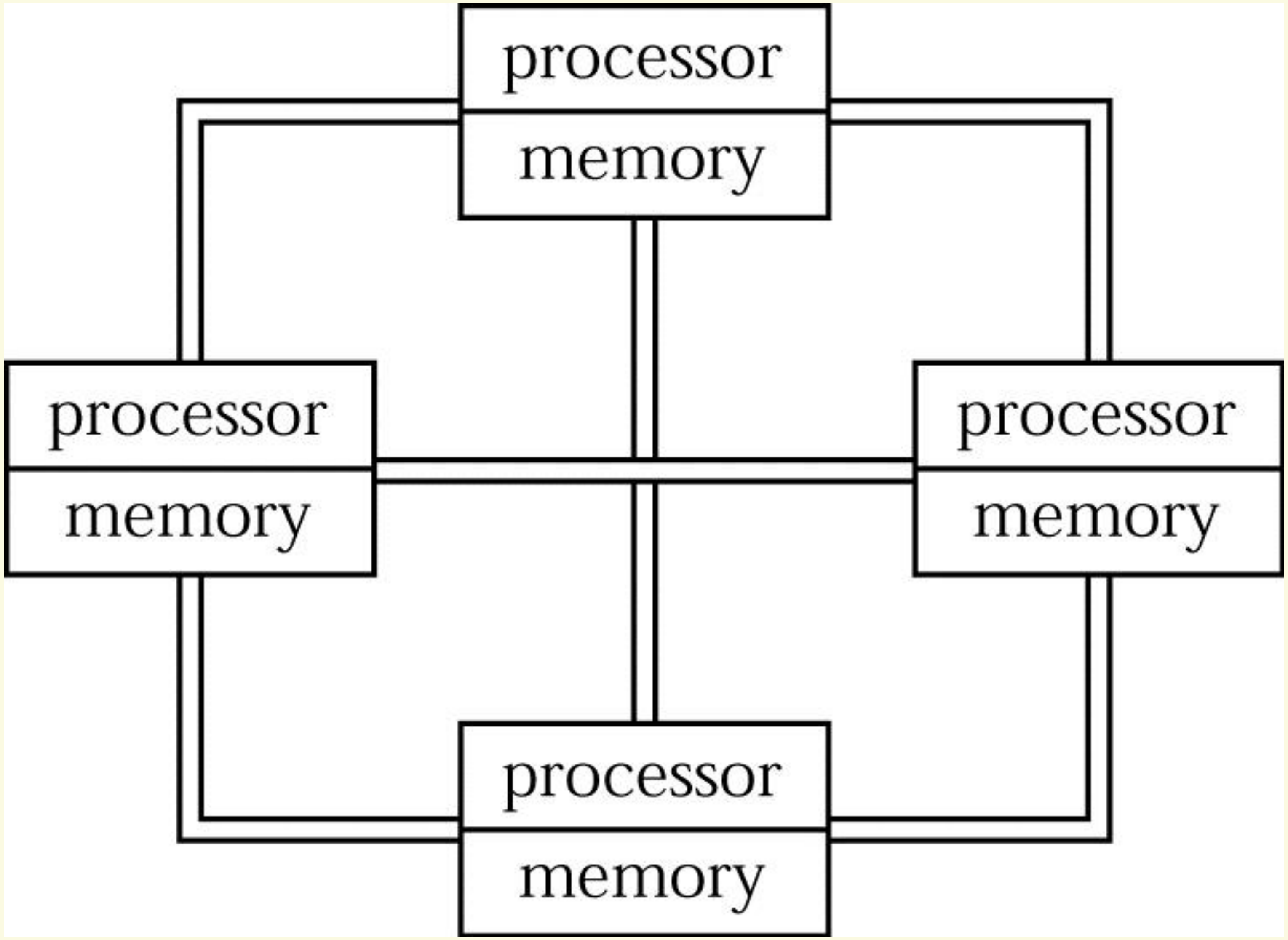
- In other architectures all the processors act independently. Such systems are called **multipleinstruction**, **multiple-data**, or **MIMD systems** and may be either multiprocessor or distributed processor systems. In an MIMD system the processors may operate at different speeds, and therefore such systems are asynchronous. Thus, the synchronization of the processors in an MIMD system becomes a critical problem.


programs
processors
memory

## Shared memory and distributed memory

- Just as instructions are shared in an SIMD system, in a MIMD system, memory may be shared. A system in which one central memory is shared by all the processors is called a **shared-memory system** and is also by nature a multiprocessor rather than a distributed system, while a system in which each processor has its own independent memory is called a **distributed-memory system** (and may be either an actual distributed system or a multiprocessor system).


processors
memory


processor
memory
processor
memory
processor
memory
processor
memory

## Race Condition

- In a shared-memory system the processors communicate through the changes each makes to the shared memory. If the processors operate asynchronously, they may also use the shared memory to synchronize their activities. For this to work properly, each processor must have exclusive access to those parts of memory that it is changing. Without mutual exclusion, different processes may be modifying the same memory locations in an interleaved and unpredictable way, a situation that is sometimes called a **race condition**.

## Deadlock

- Solving the mutual exclusion problem typically means blocking processes when another process is already accessing shared data using some kind of locking mechanism. This can cause a new problem to arise, namely, **deadlock**, in which processes end up waiting forever for each other to unblock. Detecting or preventing deadlock is typically one of the most difficult problems in parallel processing.

## Communication Problem

- Distributed-memory systems do not have to worry about mutual exclusion, since each processor has its own memory inaccessible to the other processors. On the other hand, distributed processors have a **communication problem**, in that each processor must be able to send messages to and receive messages from all the other processors asynchronously.

## Link types

- Communication between processors depends on the configuration of links between the processors. Sometimes processors are connected in sequence, and processors may need to forward information to other processors farther along the link. If the number of processors is small, each processor may be fully linked to every other processor.

## Deadlock in distributed systems

- communicating processes may also block while waiting for a needed message, and this, too, can cause deadlock. Indeed, solving deadlock problems in a distributed system can be even more difficult than for a shared memory system, because each process may have little information about the status of other processes.

## operating system will need to provide:

- A means of creating and destroying processes.
- A means of managing the number of processors used by processes (for example, a method of assigning processes to processors or a method for reserving a number of processors for use by a program).

## operating system will need to provide:

- On a shared-memory system, a mechanism for ensuring mutual exclusion of processes to shared memory. Mutual exclusion is used for both process synchronization and communication.

## operating system will need to provide:

- On a distributed-memory system, a mechanism for creating and maintaining communication channels between processors. These channels are used both for interprocess communication and for synchronization. A special case of this mechanism is necessary for a network of loosely connected independent computers, such as computers cooperating over the Internet.

## Parallel Processing and Programming Languages

- Programming languages are like operating systems in that they need to provide programmers with mechanisms for process creation, synchronization, and communication.

- However, a programming language has stricter requirements than an operating system. Its facilities must be machine-independent and must adhere to language design principles such as readability, writability, and maintainability.

## Parallel Processing and Programming Languages

- Some languages use the shared-memory model and provide facilities for mutual exclusion, typically by providing a built-in thread mechanism or thread library, while others assume the distributed model and provide communication facilities.
- A few languages have included both models, the designers arguing that sometimes one model and sometimes the other will be preferable in particular situations.

## Parallel Processing and Programming Languages

- A language designer can also adopt the view that parallel mechanisms should not be included in a language definition at all. In that case, the language designer can still provide parallel facilities in other ways.

## *The bounded buffer problem*

- This problem assumes that two or more processes are cooperating in a computational or input-output situation.
- One (or more) process produces values that are consumed by another process (or processes).
- An intermediate buffer, or buffer process, stores produced values until they are consumed.

## *The bounded buffer problem*

- A solution to this problem must ensure that no value is produced until there is room to store it and that a value is consumed only after it has been produced.
- This involves both communication and synchronization.
- When the emphasis is not on the buffer, this problem is called the *producer-consumer problem.*

## *Parallel matrix multiplication.*

- This problem is different from the previous one in that it is an example of an algorithmic application in which the use of parallelism can cause significant speedups.

- Matrices are essentially two-dimensional arrays, as in the following declaration of integer matrices

## *Parallel matrix multiplication.*

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

## Standard Matrix Multiplication

- 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];

## *Parallel matrix multiplication.*

- This computation, if performed sequentially, takes N^3 steps.
- If, however, we assign a process to compute each c[i][j], and if each process executes on a separate processor, then the computation can be performed in the equivalent of N steps.

## *Parallel matrix multiplication.*

- Algorithms such as this are studied extensively in courses on parallel algorithms.
- In fact, this is the simplest form of such an algorithm, since there are no write conflicts caused by the computation of the c[i][j] by separate processes.
- Thus, there is no need to enforce mutual exclusion in accessing the matrices as shared memory.

## *Parallel matrix multiplication.*

- There is, however, a synchronization problem in that the product c cannot be used until all the processes that compute it have finished.
- A programming language, if it is to provide useful parallelism, must provide facilities for implementing such algorithms with a minimum of overhead.

## Parallel Programming Without Explicit Language Facilities

- In theory it is possible for language translators, using optimization techniques, to automatically use operating system utilities to assign different processors to different parts of a program.
- However, as with operating systems, the automatic assignment of processors is likely to be suboptimal, and manual facilities are needed to make full use of parallel processors.

## Parallel Programming Without Explicit Language Facilities

- A programming language may be used for purposes that require explicitly indicating the parallelism, such as the writing of operating systems themselves or the implementation of intricate parallel algorithms.

## Parallel Programming Without Explicit Language Facilities

- A second alternative to defining parallel constructs in a programming language is for the translator to offer the programmer **compiler options** to allow the explicit indicating of areas where parallelism is called for.
- This is usually better than automatic parallelization. One of the places where this is most effective is in the use of nested loops, where each repetition of the inner loop is relatively independent of the others. Such a situation is the matrix multiplication problem just discussed.

## Parallel Programming Without Explicit Language Facilities

- A third way of making parallel facilities available without explicit mechanisms in the language design is to 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.
- This way, different libraries can be provided, depending on what facilities an operating system or parallel machine offers.
- Of course, if a standard parallel library is required by a language, then this is the same as including parallel facilities in the language definition.

## C code using library functions

- #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)
(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;
- }

## C code Explaination

- the four procedures m_set_procs, m_fork, m_kill_procs, and m_get_myid are imported from a library (the parallel/parallel library).

## C code Explaination

- The call to m_set_procs sets the number of processes (and processors) that are to be used, returning an error code if not enough processors are available.
- The call to m_kill_procs synchronizes the processes, so that all processes wait for the entire loop to finish and that only one process continues to execute after the loop.

## C code Explaination

- m_fork creates the 10 processes, which are all instances of the procedure multiply
- In procedure multiply, m_get_myid gets the number of the process instance

## C code Explaination

- The remainder of the code then divides the work among the processes, so that process 0 calculates c[0][i], c[10][i], . . . , c[90][i] for all i, process 2 calculates c[2][i], c[12][i], . . . , c[92][i], and so on.

## Parallel Programming Without Explicit Language Facilities

- A final alternative to introducing parallelism into a language is to simply rely on operating system features directly to run programs in parallel.
- Essentially, this requires that a parallel program be split up into separate, independently executable pieces and then set up to communicate via operating system mechanisms

- A typical example of this is “to string programs together in a UNIX operating system through the use of **pipes**”, which is a method of streaming text input and output from one program to another without storing to intermediate files

- A simple example is the following, which will list all filenames in the current directory containing the string “java:”
- ls | grep "java"

## Process Creation and Destruction

- A programming language that contains explicit mechanisms for parallel processing must have a construct for creating new processes.

## Process Creation and Destruction

- There are two basic ways that new processes can be created.
- One is to split the current process into two or more processes that continue to execute copies of the same program.
- In this case, one of the processes is usually distinguished as the **parent**, while the others become the **children**.
- The processes can execute different code by a test of process identifiers or some other condition, but the basic program is the same for all processes.
- This method of process creation resembles the SIMD organization and is, therefore, called **SPMD programming** (for single program multiple data).

## Process Creation and Destruction

- Note, however, that SPMD programs may execute different segments of their common code and so do not necessarily operate synchronously. Thus, there is a need for process synchronization.

## Process Creation and Destruction

- In the second method of process creation, a segment of code (commonly a procedure) is explicitly associated with each new process.

- Thus, different processes have different code, and we can call this method **MPMD programming**.

## Process Creation and Destruction

- A typical case of this is the so-called **fork-join** model, where a process creates several child processes, each with its own code (a fork), and then waits for the children to complete their execution (a join).
- Unfortunately, the name is confusing, because the UNIX system called fork() is really an SPMD process creator, not a fork-join creator.
- We will, therefore refer to MPMD process creation rather than a fork-join creation.

## Process Creation and Destruction

- An alternative view of process creation is to focus on the size of the code that can become a separate process.
- In some designs, individual statements can become processes and be executed in parallel.
- A second possibility is for procedures to be assigned to processes.
- A third possibility is for processes to represent whole programs only.

# Granularity

- Sometimes the different size of the code assignable to separate processes is referred to as the **granularity** of processes
- **1.** Statement-level parallelism: fine-grained
- **2.** Procedure-level parallelism: medium-grained
- **3.** Program-level parallelism: large-grained

## Granularity

- Granularity can be an issue in program efficiency: Depending on the kind of machine, many small grained processes can incur significant overhead in their creation and management, thus executing more slowly than fewer larger processes.

## Granularity

- On the other hand, large-grained processes may have difficulty in exploiting all opportunities for parallelism within a program.
- An intermediate case that can be very efficient is that of a thread, which typically represents fine-grained or medium-grained parallelism without the overhead of full-blown process creation.

## Process Creation and Destruction

- Regardless of the method of process creation, it is possible to distinguish between process creator (the parent process) and process created (the child process), and for every process creation mechanism, the following two questions must be answered:

- **1.** Does the parent process suspend execution while its child processes are executing, or does it continue to execute alongside them?
- **2.** What memory, if any, does a parent share with its children or the children share among themselves?

## Process Termination

- In addition to process creation, a parallel programming language needs a method for process termination.

- In the simplest case, a process will simply execute its code to completion and then cease to exist.

- In more complex situations, however, a process may need to continue executing until a certain condition is met and then terminate. It may also be necessary to select a particular process to continue execution.

## Statement-Level Parallelism

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

## Statement-Level Parallelism

- In this statement the statements S1, . . . , Sn are executed in parallel.

- It is assumed that the main process is suspended during their execution, and that all the processes of the Si share all variables not locally declared within an Si.

## Procedure-Level Parallelism

- C Program discussed earlier example 13.4 in book.

## Program-Level Parallelism

- In this method of process creation only whole programs can become processes.
- Typically, this occurs in MPMD style, where a program creates a new process by creating a complete copy of itself.
- The typical example of this method is the fork call of the UNIX operating system.

## Program-Level Parallelism

- A call to fork causes a second child process to be created that is an exact copy of the calling process, including all variables and environment data at the moment of the fork.

- Processes can tell which is the child and which is the parent by the returned value of the call to fork: A zero value indicates that the process is the child, while a nonzero value indicates the process is the parent

## Program-Level Parallelism

- By testing this value the parent and child processes can be made to execute different code:

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

## Program-Level Parallelism

- After a call to fork, a process can be terminated by a call to exit.

- Process synchronization can be achieved by calls to wait, which causes a parent to suspend its execution until a child terminates.

## C program with fork join and wait

- #define SIZE 100
- #define NUMPROCS 10
- int a[SIZE][SIZE], b[SIZE][SIZE], c[SIZE][SIZE];

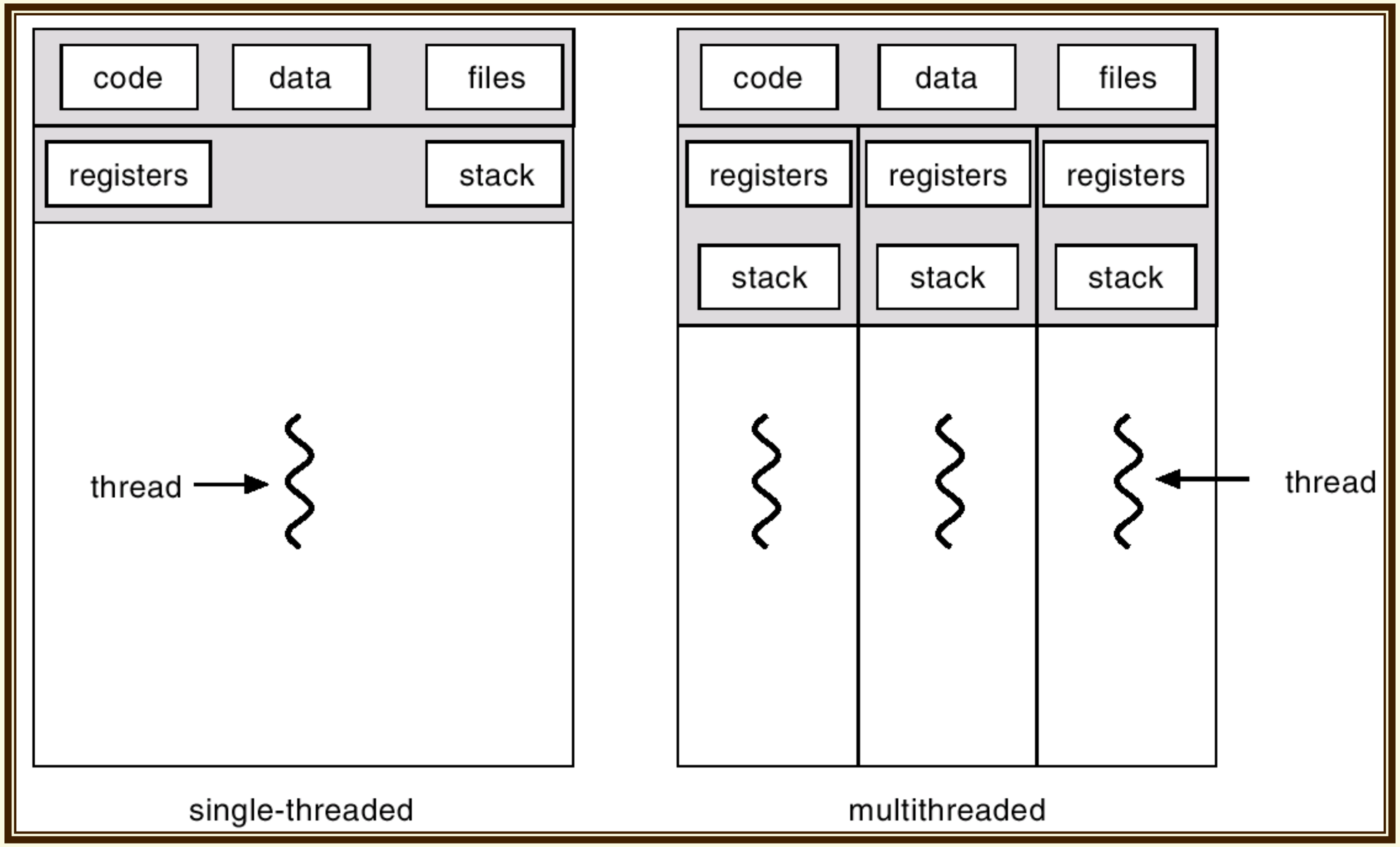
# C program with fork join and wait

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

## C program with fork join and wait

```
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];
}
}
```

## THREADS


code
data
files
registers
stack
thread
single-threaded
code
data
files
registers
registers
registers
stack
stack
stack
thread
multithreaded

## Threads in Java

- Threads are built into the Java language, in that the Thread class is part of the java.lang package, and the reserved word synchronize is used to establish mutual exclusion for threads.

- A Java thread is created by instantiating a Thread object and by defining a run method that will be executed when the thread starts.

## Threads in Java

- This can be done in two ways,
- either by extending Thread through inheritance and overriding the (empty) Thread.run method
- or by defining a class that implements the Runnable interface (which only requires the definition of a run method), and then passing an object of this class to the Thread constructor.

## Extending Thread class

- class MyThread extends Thread{
- public void run()
- { ... }
- }
- ...
- Thread t = new MyThread();

## Implementing runnable

- class MyRunner implements Runnable{
- public void run()
- { ... }
- }
- ...
- MyRunner m = new MyRunner();
- Thread t = new Thread(m);

## Threads in Java

- Defining a thread does not begin executing it. Instead, a thread begins running when the start method is called:
- t.start();

## Threads in Java

- The start method in turn will call run.
- Although one could call the run method directly, this will in general not work, since the system must perform some internal bookkeeping before calling run.
- Note also that every Java program is already executing inside a thread whose run method is main.

## Threads in Java

- Thus, when a new thread's execution is begun by calling start, the main program will still continue to execute to completion in its own thread (in other words, the start method of a thread immediately returns, while the thread itself continues to execute).

- However, the entire *program* will not finish execution until all of its threads complete the execution of their run methods

## Threads in Java

- How are threads destroyed?
- As we have just noted, the simplest mechanism is to let each thread execute its run method to completion, at which time it will cease to exist without any programmer intervention.
- Threads can also wait for other threads to finish before continuing by calling the join method on the thread object:

## Threads in Java

- 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
- Thus, Java threads exhibit fork-join parallelism

## Threads in Java

- An alternative to waiting for a thread to finish executing is to interrupt it using the interrupt method.
- The interrupt method, does not actually stop the interrupted thread from executing, but simply sets an internal flag in the thread object that can be used by the object to test whether some other thread has called its interrupt method.
- This allows a thread to continue to execute some cleanup code before actually exiting

# Threads in Java

- Note that, when t.join() is called, the current thread (i.e., the thread that makes this call) becomes blocked, and will only unblock once t exits its run method.
- If t is also in a blocked state waiting for an event, then deadlock can result.
- If there is a real potential for deadlock, then we could try to avoid it by waiting only a specified amount of time, and then timing out:
- t.join(1000); // wait 1 second for t, then give up

## Java will not handle deadlock

- In general, any Java method such as join that blocks the current thread has a timeout version such as the above, because the Java runtime system makes no attempt to discover or prevent deadlock.
- It is entirely up to the programmer to ensure that deadlock cannot occur.

## Threads in Java

- wait();
- notify();
- notifyAll();

## A Bounded Buffer Example in Java

- We will cast this problem as an input-output problem, with the producer reading characters from standard input and inserting them into the buffer and
- the consumer removing characters from the buffer and writing them to standard output.
- To add interest to this example, we set the following additional requirements:

## A Bounded Buffer Example in Java

- The producer thread will continue to read until an end of file is encountered, whence it will exit.
- The consumer thread will continue to write until the producer has ended and no more characters remain in the buffer.
- Thus, the consumer should echo all the characters that the producer enters into the buffer.

# A Bounded Buffer Example in Java

```
import java.io.*;
class BoundedBuffer{
public static void main(String[] args){
Buffer buffer = new Buffer(5); // buffer has size 5
Producer prod = new Producer(buffer);
Consumer cons = new Consumer(buffer);
Thread read = new Thread(prod);
Thread write = new Thread(cons);
read.start();
write.start();
try {
read.join();
write.interrupt();
}
catch (InterruptedException e) {}
}
}
```

## A Bounded Buffer Example in Java

```
class Buffer{
private final char[] buf;
private int start = -1;
private int end = -1;
private int size = 0;
public Buffer(int length){
buf = new char[length];
}
public boolean more()
{ return size > 0; }
```

## A Bounded Buffer Example in Java

```
public synchronized void put(char ch)
{  try {
while (size == buf.length) wait();
end = (end+1) % buf.length;
buf[end] = ch;
size++;
notifyAll();
}
catch (InterruptedException e)
{ Thread.currentThread().interrupt(); }
}
```

## A Bounded Buffer Example in Java

```
public synchronized char get()
{ try {
while (size == 0) wait();
start = (start+1) % buf.length;
char ch = buf[start];
size--;
notifyAll();
return ch;
}
catch (InterruptedException e)
{ Thread.currentThread().interrupt(); }
return 0;
}
}
```

## A Bounded Buffer Example in Java

```
class Consumer implements Runnable{
private final Buffer buffer;
public Consumer(Buffer b)
{ buffer = b; }
public void run() {
while (!Thread.currentThread().isInterrupted())
{ char c = buffer.get();
System.out.print(c);
}
while(buffer.more()) // clean-up
{ char c = buffer.get();
System.out.print(c);
}
}   }
```

# A Bounded Buffer Example in Java

```
class Producer implements Runnable{
private final Buffer buffer;
private final InputStreamReader in
= new InputStreamReader(System.in);
public Producer(Buffer b) { buffer = b; }
public void run() {
try {
while (!Thread.currentThread().isInterrupted()) {
int c = in.read();
if (c == -1) break; // -1 is end of file
buffer.put((char)c);
}
}
catch (IOException e) {}
}  }
```

## A Bounded Buffer Example in Java

- The main program creates a buffer of five characters, a reader (the producer) for the buffer and a writer (the consumer) that prints characters.

- Two threads are created, one for the producer and one for the consumer (so that three threads in all exist for this program, including the main thread).

## A Bounded Buffer Example in Java

- The main thread then waits for the reader to finish (line 12), and then interrupts the writer (line 13). (Since join can, like wait, generate InterruptedException, we must also handle this exception.)

- The bounded buffer class itself uses an array to hold the characters, along with two indices that traverse the array in a circular fashion, keeping track of the last positions where insertions and removals took place (this is a standard implementation technique discussed in many data structures books).

## A Bounded Buffer Example in Java

- Both the put (insert) and get (remove) operations are synchronized to ensure mutual exclusion.
- The put operation delays a thread if the buffer is full (line 30),
- While the get operation delays a thread if the buffer is empty (line 41).
- As a result, both operations must handle the InterruptedException. Both also call notifyAll after performing their respective operations (lines 34 and 45).

## A Bounded Buffer Example in Java

- The Producer class (lines 67–81) is relatively simple.
- Its run method checks for an interrupt—even though interrupt will never be called on its thread in this program, it is reasonable to write general code such as this.
- Its only other behavior is to check for an end of file (line 76, where EOF 5 21), whence it exits.

## A Bounded Buffer Example in Java

- The Consumer class is almost symmetrical to the Producer class.
- The major difference is that, when a Consumer is interrupted, we want to make sure that the buffer has been emptied before the Consumer exits.
- Thus, on reaching line 63, we know that the producer has finished, so we simply test for more characters in the buffer and write them before exiting.

## Semaphores

- A **semaphore** is a mechanism to provide mutual exclusion and synchronization in a shared-memory model.
- A semaphore is a shared integer variable that may be accessed only via three operations: **InitSem**, **Signal**, and **Delay**.

## Semaphores

- The **Delay** operation tests the semaphore for a positive value, decrementing it if it is positive and suspending the calling process if it is zero or negative.
- The **Signal** operation tests whether processes are waiting, causing one of them to continue if so and incrementing the semaphore if not.

## Semaphores

- **Signal** is analogous to notify in Java, and **Delay** is analogous to wait.

## Semaphores

- Given a semaphore $S$, the **Signal** and **Delay** operations can be defined in terms of the following pseudocode:
- *Delay*($S$): if $S > 0$ then $S := S - 1$ else suspend the calling process
- *Signal*($S$): if processes are waiting then wake up a process else $S := S + 1$

## Semaphores

- Unlike ordinary code, however, the system must ensure that each of these operations executes **atomically**, that is, by only one process at a time.

## Semaphores

- Given a semaphore *S*, we can ensure mutual exclusion by defining a **critical region**, that is, a region of code that can be executed by only one process at a time. If S is initialized to 1, then the following code defines such a critical region:
- *Delay*(*S*);
- {critical region}
- *Signal*(*S*);

## Semaphores

- A typical critical region is code where shared data is read and/or updated. Sometimes semaphores are referred to as **locks**, since they lock out processes from critical regions.

## Semaphores

- Semaphores can also be used to synchronize processes. If, for example, process *p* must wait for process *q* to finish, then we can initialize a semaphore *S* to 0, call *Delay*(*S*) in *p* to suspend its execution, and call *Signal*(*S*) at the end of *q* to resume the execution of *p*.

## Semaphores

- An important question to be addressed when defining semaphores 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, or using some sort of priority system. This choice has a major effect on the behavior of concurrent programs using semaphores.

```
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();
}
}
```

## Semaphores

- This code uses a call to notify rather than notifyAll, because each semaphore’s wait-list is waiting on a single condition—that count should be greater than 0—so that only one waiting thread needs to be awakened at a time.

## A Bounded Buffer Using Semaphores

- The solution uses three semaphores. The first, mutEx, provides mutual exclusion to code that changes the private state of a Buffer. The semaphores nonEmpty and nonFull maintain the status of the number of stored items available.

## A Bounded Buffer Using Semaphores

- all Java synchronization mechanisms—synchronized methods, calls to wait and notifyAll—are now gone from the Buffer code itself, since they are indirectly supplied by the semaphores.

## A Bounded Buffer Using Semaphores

- (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) }

## A Bounded Buffer Using Semaphores

- (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) }

```
catch (InterruptedException e)
{ Thread.currentThread().interrupt(); }
}
public char get(){
try{
nonEmpty.delay();
mutEx.delay();
start = (start+1) % buf.length;
char ch = buf[start];
size--;
mutEx.signal();
nonEmpty.signal();
return ch;
}
catch (InterruptedException e)
{ Thread.currentThread().interrupt(); }
return 0;
}
}
```

## Difficulties with Semaphores

- The basic difficulty with semaphores is that, even though the semaphores themselves are protected, there is no protection from their incorrect use or misuse by programmers.

## Difficulties with Semaphores

- For example, if a programmer incorrectly writes:
- *Signal(S);*
- . . .
- *Delay*(*S*);
- then the surrounded code is not a critical region and can be entered at will by any process.

## Difficulties with Semaphores

- On the other hand, if a programmer writes:
- *Delay*(*S*);
- . . .
- *Delay*(*S*);
- then it is likely that the process will block at the second *Delay*, never to resume execution.

## Difficulties with Semaphores

- It is also possible for the use of semaphores to cause deadlock. A typical example is represented by the following code in two processes, with two semaphores *S*1 and *S*2:

## Difficulties with 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);

## Difficulties with Semaphores

- If Process 1 executes *Delay*(*S*1) at the same time that Process 2 executes *Delay*(*S*2), then each will block waiting for the other to issue a *Signal*. Deadlock has occurred.

## Implementation of Semaphores

- Generally, semaphores are implemented with some form of hardware support. Even on single-processor systems this is not an entirely trivial proposition, since an operating system may possibly interrupt a process between any two machine instructions.

## Implementation of Semaphores

- One common method for implementing semaphores on a single-processor system is the ***TestAndSet*** machine instruction, which is a single machine instruction that tests a memory location and simultaneously increments or decrements the location if the test succeeds.

## Implementation of Semaphores

- Assuming that such a *TestAndSet* operation returns the value of its location parameter and decrements its location parameter if it is > 0, we can implement *Signal* and *Delay* with the following code schemas:
- *Delay*(*S*): while *TestAndSet*(*S*) <= 0 do {nothing};
- *Signal*(*S*) : *S* : = *S* + 1;

## Implementation of Semaphores

- 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**.)

## Starvation

- It also leaves unresolved the order in which waiting processes are reactivated: It may be random or in some order imposed by the operating system. In the worst case, a waiting process may be preempted by many incoming calls to *Delay* from new processes and never get to execute despite a sufficient number of calls to *Signal*. Such a situation is called **starvation**.

## Fair Share Scheduler

- Starvation is prevented by the use of a scheduling system that is **fair**—that is, guarantees that every process will execute within a finite period of time. In general, avoiding starvation is a much more difficult problem to solve than avoidance of deadlock

## Semaphores

- Modern shared-memory systems often provide facilities for semaphores that do not require busy-waiting. Semaphores are special memory locations that can be accessed by only one processor at a time, and a queue is provided for each semaphore to store the processes that are waiting for it.

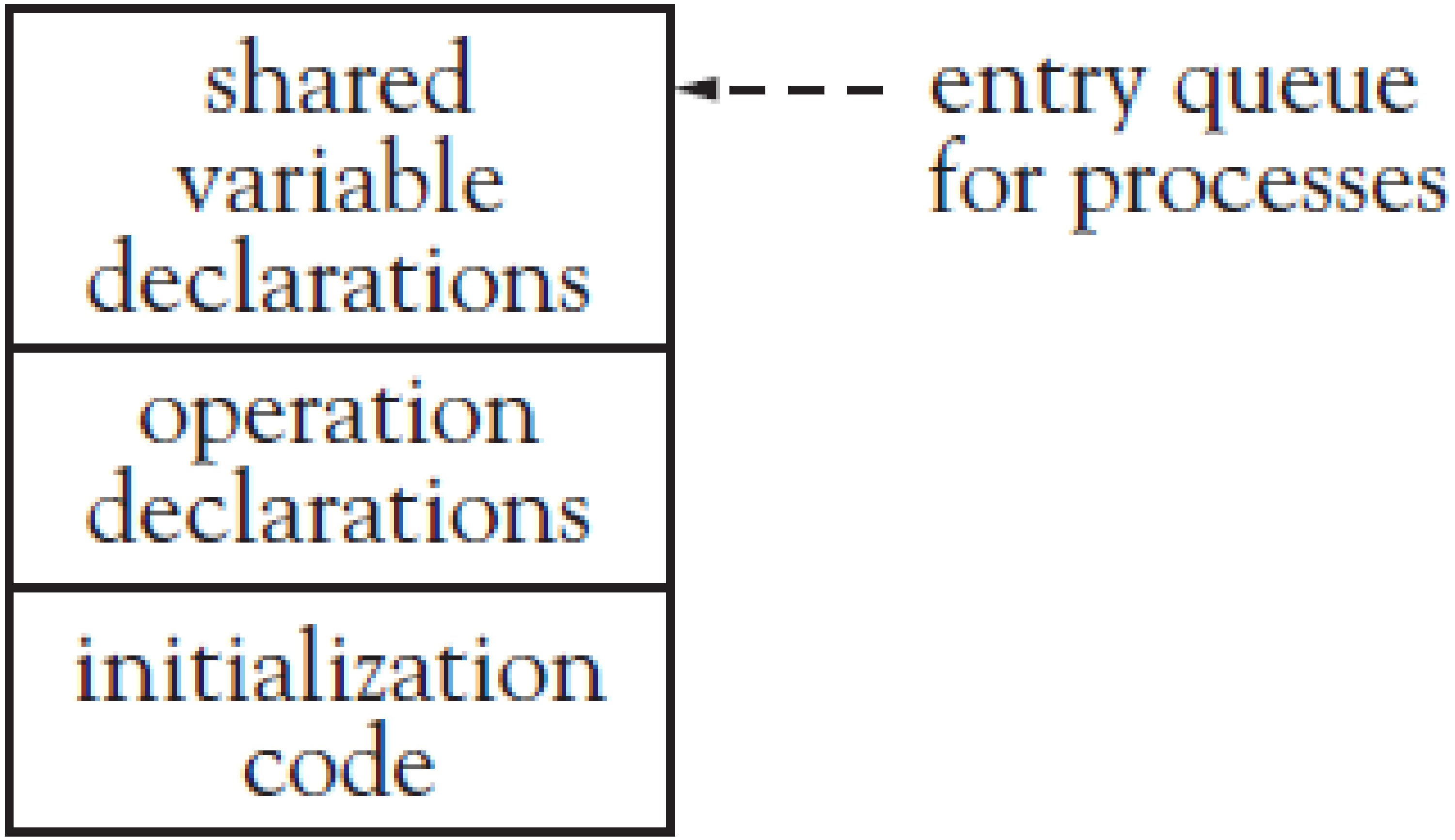
## Monitors

- A monitor is a language construct that attempts to encapsulate the mutual exclusion and synchronization mechanisms of semaphores.
- A **monitor** is an abstract data type mechanism with the added property of mutual exclusion.

## Monitors

- It encapsulates shared data and operations on these data. At most one process at a time can be “inside” the monitor—using any of the monitor’s operations.
- To keep track of processes waiting to use its operations, a monitor has an associated wait queue, which is organized in some fair fashion

## Structure of monitors



## Monitors

- This organization of a monitor provides for mutual exclusion in accessing shared data, but it is not adequate by itself to synchronize processes that must wait for certain conditions before continuing to execute.
- For example, in the bounded buffer problem, a consumer process must wait if no items are in the buffer, and a producer must wait if the buffer is full.

## Monitors

- For this reason, a monitor must also provide **condition variables**, which are shared variables within the monitor resembling semaphores.
- Each has an associated queue made up of processes waiting for the condition, and each has associated **suspend** and **continue** operations, which have the effect of enqueuing and dequeuing processes from the associated queue

## Monitors

- Unfortunately, sometimes these operations are also called signal and delay, but their operation is different from the **Signal** and **Delay** operations of semaphores. If a condition queue is empty, a call to **continue** will have no effect, and a call to **suspend** will **always** suspend the current process.

## Monitors

- What happens when a **continue** call is issued to a waiting process by a process in the monitor? In this situation, there are now potentially two processes active in the monitor, a situation that is forbidden.
- Two possibilities exist:
- (1) the suspended process that has just been awakened by the **continue** call must wait further until the calling process has left the monitor, or
- (2) the process that issued the **continue** call must suspend until the awakened process has left the monitor.

## Monitors

- It is possible to imitate the behavior of a monitor using semaphores.
- Indeed, in the last section you saw how to implement a semaphore in Java using what amounts to a monitor, and then used the semaphore to implement a bounded buffer monitor

## Monitors

- monitors and semaphores are equivalent in terms of the kinds of parallelism they can express. However, monitors provide a more structured mechanism for concurrency than semaphores, and they ensure mutual exclusion. Monitors cannot guarantee the absence of deadlock, however.

## Monitors

- Java objects, all of whose methods are synchronized, are essentially monitors; for the purposes of this discussion, we will call such Java objects **synchronized objects**. Java provides an entry queue for each synchronized object.

## Monitors

- A thread that is inside the synchronized object (that is, that executes a synchronized method of the object) has the lock on the synchronized object. One immediate problem with these queues in Java, however, is that they do not operate in a fair fashion. No rules control which thread is chosen on a wait queue when the executing thread leaves a method

## Monitors

- In Java an object may have both synchronized and unsynchronized methods. What’s more, any of the unsynchronized methods may be executed without acquiring the lock or going through the entry queue.

## Monitors

- Additionally, Java’s synchronized objects do not have separate condition variables. There is only one wait queue per synchronized object for any and all conditions (which is of course separate from the entry queue). A thread is placed in a synchronized object’s wait queue by a call to wait or sleep;

## Monitors

- threads are removed from an object’s wait queue by a call to notify or notifyAll. Thus, wait and sleep are **suspend** operations as described previously, while notify and notifyAll are **continue** operations.

## Monitors

- Note that a thread may only be placed in a synchronized object’s wait queue if it has the lock on the synchronized object, and it gives up the lock at that time. When it comes off the wait queue, it must again acquire the object’s lock, so that in general it must go back on the entry queue from the wait queue. Thus, awakened threads in Java must wait for the awakening thread to exit the synchronized code before continuing

## The Java Lock and Condition Interfaces

- Java 1.5 introduced the java.concurrent.locks package, which includes a set of interfaces and classes that support a more authentic monitor mechanism than synchronized objects.

## The Java Lock and Condition Interfaces

- Methods that access the resource under mutual exclusion are now not specified as synchronized, but instead acquire the lock by running the unlock method and release it by running the lock method on the lock itself.

## The Java Lock and Condition Interfaces

- After acquiring the lock, a method either finishes or waits on an explicit condition object (using the method await). Multiple condition objects can be associated with a single lock, and each condition object has its own queue of threads waiting on it. When a method finishes, it can signal other threads waiting on a condition object by using the method signal or signalAll with that object.

## Bounded Buffer problem

- 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

## Bounded Buffer problem

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

## Bounded Buffer problem

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

## Bounded Buffer problem

- Note that the methods put and get are no longer specified as synchronized. Instead, their first step is to acquire the buffer’s lock, using the call lock.unlock(). The code to release the lock, lock.
- unlock(), should be placed in a finally clause associated with the try-catch statement. The method put then waits on the okToPut condition and signals the okToGet condition, whereas the method get waits on the okToGetCondition and signals the okToPut condition.

## Message Passing

- Message passing is a mechanism for process synchronization and communication using the distributed model of a parallel processor.

## Message Passing

- In its most basic form, a message-passing mechanism in a language consists of two operations, ***send*** and ***receive***, which may be defined in C syntax as follows:
- void send(Process to, Message m);
- void receive(Process from, Message m);

## Message Passing

- In this form, both the sending process and the receiving process must be named. This implies that every sender must know its receiver, and vice versa. In particular, the sending and receiving processes must have names within the scope of each other.
- A less restrictive form of *send* and *receive* removes the requirement of naming sender and receiver:
- void send(Message m);
- void receive(Message m);

## Message Passing

- In this case, a sent message will go to any process willing to receive it, and a message will be received from any sender