Web Engineering – Fall 2015
Punjab University College of Information Technology
University of the Punjab

Multi-Threading

Table of Contents

1 Introduction

Concurrency

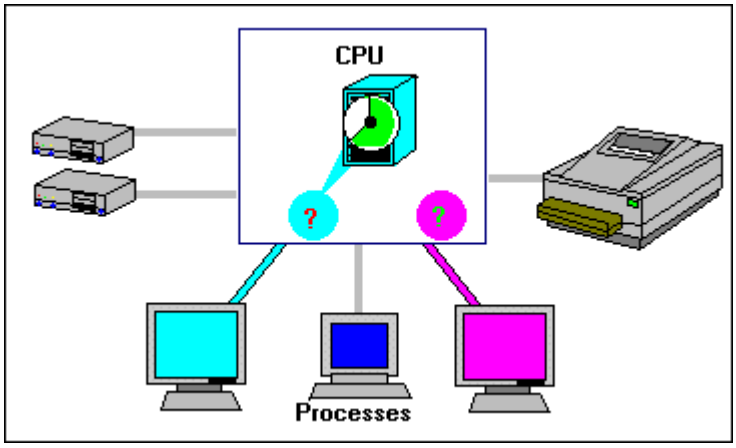
- A sequential program has a single thread of control.
  Its execution is called a process.
- A concurrent program has multiple threads of control.
  They may be executed as parallel processes.

This lesson presents main principles of concurrent programming.

A concurrent program can be executed by

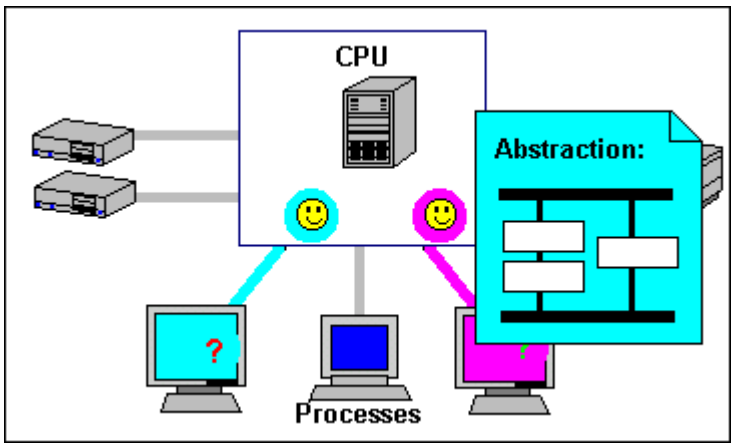
- ***Multiprogramming***: processes share one or more processors
- ***Multiprocessing***: each process runs on its own processor but with shared memory
- ***Distributed processing***: each process runs on its own processor connected by a network to others

In all these cases, the basic principles of concurrent programming are the same.



Operating System usually provides a ***logical abstraction*** for the underlying hardware, and concurrent programs are based on such abstractions.

Thus, on a programming level, programmers do not distinguish multiple and single CPU hardware environments, they write programs using such logical abstractions as ***threads, processes, resources,*** etc.



Why do we need concurrent programs?

- Better performance in a multiprocessing hardware environment.
- Better performance for distributed and input-output intensive applications. (for example, when one process is waiting for a response from a network or input/output device, other threads may continue with CPU)
- Concurrent programs model user-computer interactions and process control applications more naturally than ordinary sequential applications. (for example, a special thread is waiting for a user action and initiates other threads depending on the action and current context).
- Some applications are concurrent by their nature and simply cannot be modelled in a sequential way (for example, multi-media applications).
- Concurrent programs are able to coordinate distributed services

But concurrent applications introduce complexity:

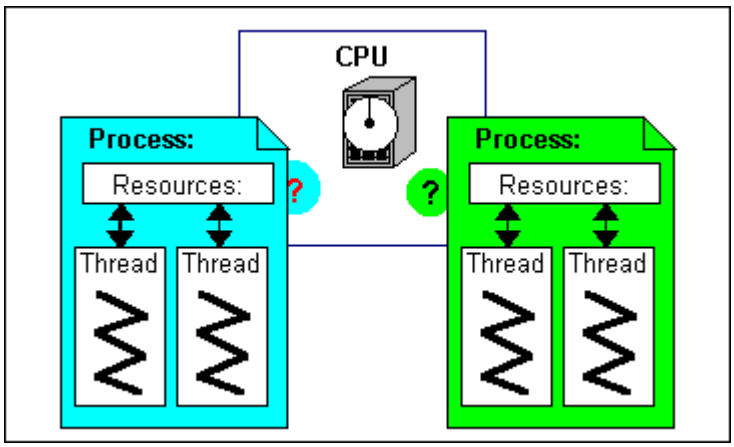
- ***Safety***: concurrent processes may corrupt shared data
- ***Liveness***: processes may "starve" if not properly coordinated
- ***Non-determinism***: the same program run twice may give different results
- ***Run-time overhead***: thread construction, context switching and

Recollect that operating systems support concurrent programming by means of two basic concepts:

- ***Processes***
- ***Threads***

A ***process*** is a running program. The operating system allocates all resources that are needed for the process.

A ***thread*** is a dispatching unit within a process. That means that a process can have a number of threads running within the scope of that particular process.



Main difference between threads and processes can be defined as follows:

> *while a thread has an access to the memory address space and resources of its process, a process cannot access variables allocated for other processes.*

This basic property leads us to a number of conclusions:

- communication between threads created within a single process is simple because the threads share all the variables. Thus, a value produced by one thread is immediately available for all the other threads.
- threads take less time to start, stop or swap than processes, because they use the address space already allocated for the current process.

Different operating systems treat processes and threads differently. For example,

- MS-DOS support a single user process and a single thread,
- UNIX supports multiple user processes with only one thread per process,
- Solaris OS, Linux and Windows NT/2000/XP support multiple processes and multiple threads.

if an operating system supports at least multiple processes, we can say that the ***operating system supports concurrent programming***.

2 Properties of Concurrent Processes

Concurrent programs are governed by two key principles. These are the principles of "safety" and "liveness".

- The "***safety***" principle states "nothing bad can happen".
- The "***liveness***" principle states "eventually, something good happens".

2.1 Safety

Safety in general means that only one thread can access the data at a time, and ensures that the data remain in a consistent state during and after the operation.

Supopose, functions "A" and "B" below run concurrently. What is a resultant value of "x"?

```
var x = 0;
    function A()
    {x = x + 1;}
    function B()
    {x = x + 2;}
```

- **x = 3** if operations **x = x + 1** and **x = x + 2** are atomic, i.e. cannot be interrupted.
- **x = 1,2 or 3** if operations **x = x + 1** and **x = x + 2** can interrupt one another.

If we read/modify/write a file and allow operations to interrupt one annother, the file might be easily corrupted.

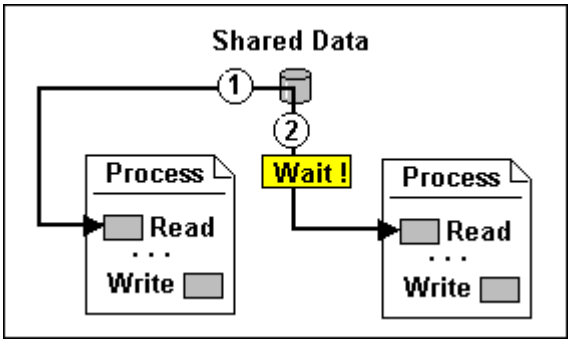
Safety is ensured by implementing

- "***mutual exclusion***" and
- "***condition synchronization***"

when operating on shared data.

"Mutual exclusion" means that that only one thread can access the data at a time, and ensures that the data remain in a consistent state during and after the operation. (atomic update).

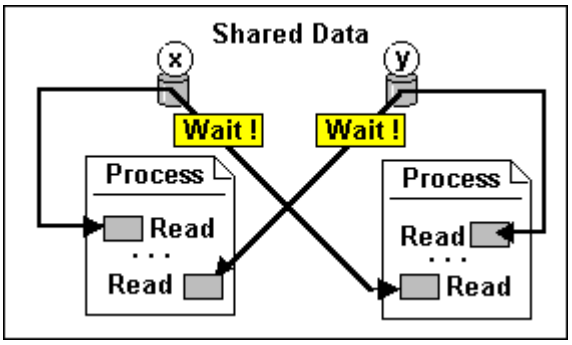
"Condition synchronization" means that operations may be delayed if shared resources are in the wrong state (e.g., read from empty buffer).



2.2 Liveness

Mutual exclusion solves many safety issues, but gives rise to other problems, in particular ***deadlock*** and ***starvation***.

The problem of ***deadlock*** arises when a thread holds a lock on one object and blocks attempting to gain a lock on another object, because the second object is already locked by a different thread, which is blocked by the lock the original thread currently holds.



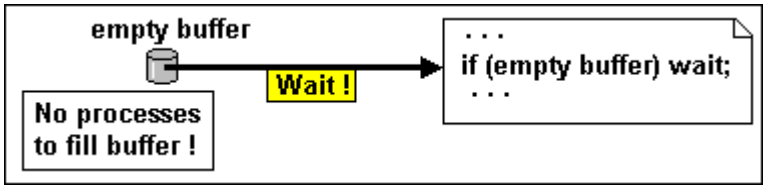
Both threads settle down to wait for the other to release the necessary lock, but neither thread will ever release their own lock because they are both blocked waiting for the other lock to be released first.

Stated like this, it may seem an unlikely occurrence, but in fact, deadlock is one of the most common concurrent programming bugs.

The trouble is that deadlock spreads out through more than two threads and can involve complex interdependencies.

Deadlock is an extreme form of ***starvation***.

Starvation occurs when a thread cannot proceed because it cannot gain access to a resource it requires.



The problems of deadlock and starvation bring us to the next big topic in concurrent programming – ***liveness***.

Concurrent programs are also described as having a 'liveness' property if there are:

- No Deadlock: some process can always access a shared resource
- No Starvation: all processes can eventually access shared resources

The liveness property states that eventually something good happens. Deadlocked programs don't meet this requirement.

Liveness is gradational. Programs can be 'nearly' dead or 'not very' live. Every time you use a synchronized method, you force sequential access to an object. If you

have a lot of threads calling a lot of synchronized methods on the same object, your program will slow down a lot.

3 Basic Concepts

A programming language must provide mechanisms for ***Expressing Concurrency***:

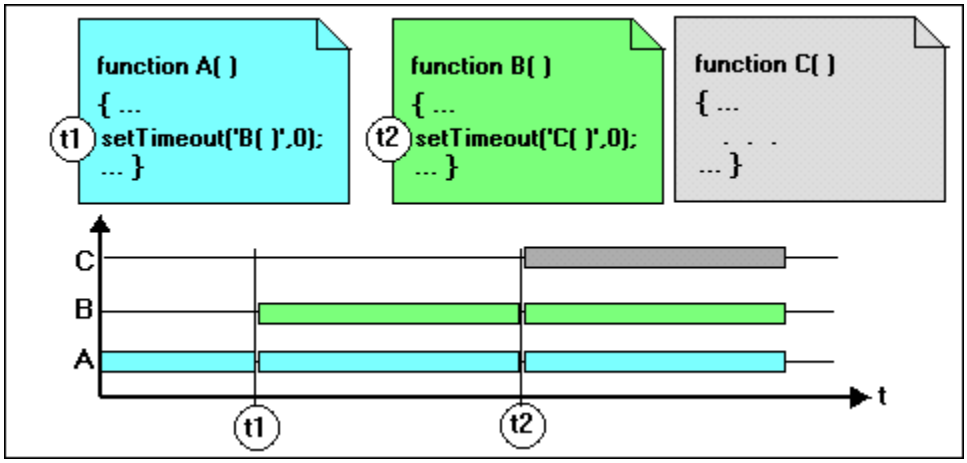
- ***Process creation*** how do you specify concurrent processes?
- ***Communication***: how do processes exchange information?
- ***Synchronization***: how do processes maintain consistency?

3.1 Process creation

Most concurrent languages offer some variant of the following ***Process Creation*** mechanisms:

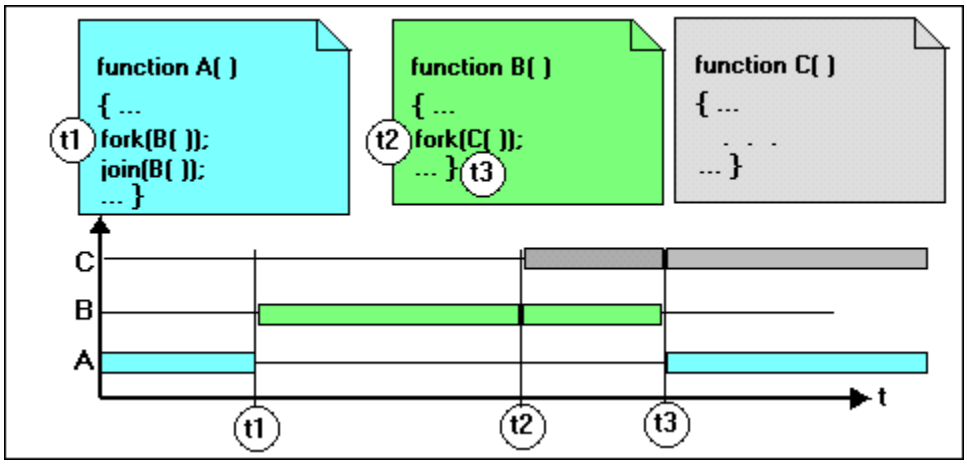
- ***Co-routines*** how do you specify concurrent processes?
- ***Fork and Join***: how do processes exchange information?
- ***Cobegin/coend***: how do processes maintain consistency?

Co-routines are only pseudo-concurrent and require explicit transfers of control:



Co-routines can be used to implement most higher-level concurrent mechanisms.

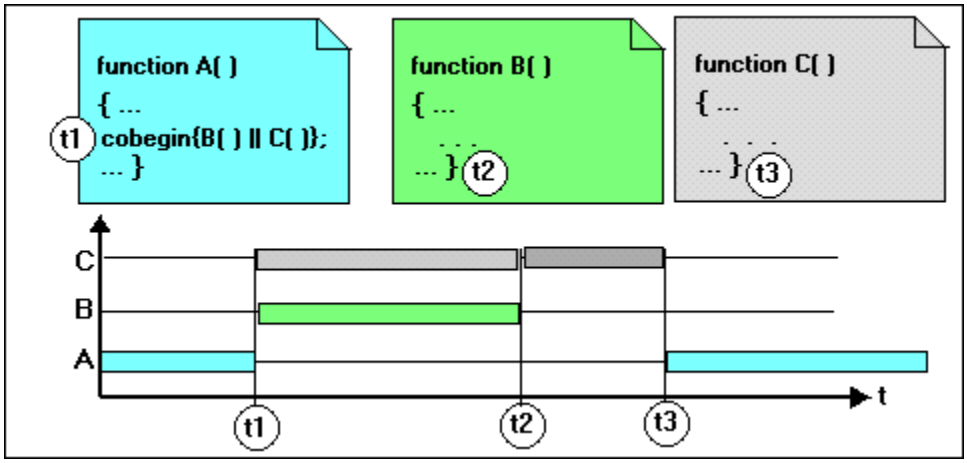
Fork can be used to create any number of processes:



Join waits for another process to terminate.

Fork and join are unstructured, so require care and discipline.

Cobegin/coend blocks are better structured, but they can only create a fixed number of processes.



The caller continues when all of the coblocks have terminated.

3.2 Communication & Synchronization

There are different ***Synchronization Techniques*** that are roughly equivalent in expressive power and can be used to implement each other.

Each approach emphasizes a different style of programming.

- Procedure Oriented (Busy-Waiting, Semaphores, Monitors, Path Expressions)
- Message Oriented (Message Passing)
- Operation Oriented (Remote Procedure Call)

Busy-waiting is primitive but effective. Processes atomically set and test shared variables.

Condition synchronization is easy to implement:

- to signal a condition, a process sets a shared variable
- to wait for a condition, a process repeatedly tests the variable

```
var iAmReady = false;
    function A()
    {
    . . .
    // signal (!)
    iAmReady = true;
    . . .
    }
```

```
function B()
    {
    . . .
    if (!iAmReady){
    //   wait 300 mseconds
      setTimeout("B()",300);return;}
    . . .
    }
```

Mutual exclusion is more difficult to realize correctly and efficiently.

Semaphores were introduced by Dijkstra (1968) as a higherlevel primitive for process synchronization.

A ***semaphore*** is a non-negative, integer-valued variable s with two operations:

- **P(s):** delays until s>0
  then, atomically executes s := s-1
- **V(s):** atomically executes s:= s+1

Many problems can be solved using binary semaphores, which take on values 0 or 1.

```
var i_have_message = new Semaphore(0);
    var message = new String();
    function Sender(mX)
    {
    . . .
    message = mX;
    // semaphor down (!)
    V(i_have_message);
    . . .
    }
function receiver()
    {
    . . .
    P(i_have_message);
    // message is available
    . . .
    }
```

A ***monitor is a single process*** that encapsulates ***resources*** and provides ***Lock/Unlock operations*** that manipulate them.

- **lock(r):** delays until "r" is marked as "unlocked"
  then, atomically marked it as "locked".
- **unLock(r):** atomically mark the resource as "unlock".

Nested monitor calls must be specially handled to prevent deadlock

```
function editDocument(dX)
    {
    . . .
    lock(dX);
    // read, edit and save
    // document "dX";
    . . .
    unLock(dX);
    . . .
    }
function editDocument(dX)
    {
    . . .
    lock(dX);
    // read, edit and save
    // document "dX";
    . . .
    unLock(dX);
    . . .
    }
```

Path expressions express the allowable sequence of operations as a kind of regular expression:

buffer : ((put && get) || (write && read))*

Although they elegantly express solutions to many problems, path expressions are too limited for general concurrent programming.

Message passing combines communication and synchronization:

- the ***sender*** specifies the message and a destination: a process, a port, a set of processes, ...
- the ***receiver*** specifies message variables and a source: source may or may not be explicitly identified.

```
function sender(mX)
    {
    . . .
    m = mX;
    send(m,"receiver");
    . . .
    }
function receiver()
    {
    var Messages = new Buffer(); . . .
    while(Messages.length > 0);
    {
    // take and process a message
    // message = Messages[0];
    . . .
    }
    . . .
```

Message transfer may be:

- asynchronous: send operations never block
- buffered: sender may block if the buffer is full
- synchronous: sender and receiver must both be ready

The synchronous message transfer method is also known as ***"Rendezvous"***

```
function sender(mX)
    {
    . . .
    m = mX;
    // point of Rendezvous
    // wait if the receiver is not ready
    send(m,"receiver");
    . . .
    }
function receiver()
    {
    var Message = new Buffer(); . . .
    // point of Rendezvous
    // wait if the sender is not ready
    x = Message;
    . . .
    }
```

4 Concurrent Programming in Java

Java programming paradigm supports multithreading process model as an integral part of the programming language, i.e. there are special operations to create, start, execute and stop multiple threads within a scope of a Java application.

In this lesson, we are using Java programming language to illustrate main principles of concurrent programming.

4.1 Basic Thread Support in Java

Java supports threads with the following standard library classes:

- ***Timer and TimerTask classes***
- ***Thread class***
- ***Runnable interface***

The ***Timer*** class offers two ways to schedule a task:

- Perform the task once after a delay
- Perform the task repeatedly after an initial delay.

```
public TimerTaskExample()
    {
    timer = new Timer();
    timer.schedule(new myTask(),0,1000);
    // after delay "0" msec
    // create an instance of myTask
    // invoke method "run" defined for myTask
    // repeat the process every second
    }
```

Subclasses of the ***TimerTask class*** override the abstract ***run method*** to implement a particular task.

```
class myTask extends TimerTask
    {
    . . .
    public void run() { ... }
    }
```

The ***Timer*** class is just a subclass of a more general ***Thread*** class.

Basically, the Thread class provides the abstraction of an execution thread within a Java program.

Thus, if programmers want to start their own threads of execution, they implement subclasses of the ***Thread*** class and override its ***run*** method to implement the execution sequence for that particular thread.

However, the Timer and TimerTask classes provide a useful implementation of a simple task scheduling mechanism, and they should be used whenever their functionality is sufficient for the application.

Runnable Interface is an interface class from the standard Java package. This class has just one public method – ***run*** method.

Thus, a class implementing this interface provides the definition of the run method for a Java thread.

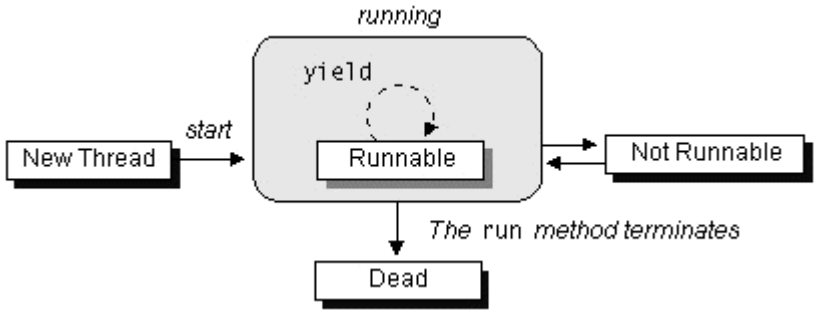
```
public class myThread implements Runnable
    {
      private String name_;
      public myThread(String name)
      {
      name_ = name;
      }
    . . .
    public void run() { . . .}
    }
```

However, this thread should be started somewhere else in code and pointed to the run method of the class implementing the Runnable interface.

```
myThread myThreadOne = new myThread("myThreadOne");
myThread myThreadTwo = new myThread("myThreadTwo");
```

4.2 The Life Cycle of a Thread

A so-called execution state, is one of basic thread properties: a thread can be running, it can be stopped, ready, etc.



There are special methods that cause the execution state transitions. We can identify the following states and corresponding methods:

- "***Created***", after we created a new instance of the Thread class
- "***Runnable***", after we either called the start method of the newly created Thread instance or resumed a "Not Runnable" thread instance
- "***Not Runnable***", after we called the "sleep" method of the "runnable" thread instance, the "runnable" thread instance calls the "wait" method, or the runnable thread instance blocks on an I/O operation
- "***Dead***", after the run method of the thread instance terminated.

Practically speaking, in order to facilitate transitions of a thread states, the Thread class should provides a public interface that can be used to manipulate instances of the class.

```
public class myThread implements Runnable
    {
      private String name_;
      private Thread thread_ = null;
      public myThread(String name)
      {
      name_ = name;
      }
    . . .
    public void run() { . . .}
    // Inteface to start the thread
    public void start(){
      if(thread_ == null){
        thread_ = new Thread(this);
    // thread is created
        thread_.start();}}
    // thread is running
    //
    // Inteface to stop the thread
    public void stop(){thread_ = null;}
    }
```

The ***start*** method

- creates the system resources necessary to run the thread,
- schedules the thread to run, and
- calls the thread's run method.

```
. . . // Inteface to start the thread
    public void start()
    {
      if(thread_ == null)
      {
        thread_ = new Thread(this);
    // thread is created
        thread_.start();
      }
    // thread is running
    //
    }
```

After the start method have been evaluated, the thread is "running".

Yet, it's somewhat more complex than that. As the previous figure shows, a thread that has been started is actually in the ***Runnable state***. Many computers have a single processor, thus making it impossible to run all "running" threads at the same time.

The Java runtime system must implement a scheduling scheme that shares the processor between all "running" threads. So at any given time, a "running" thread actually may be waiting for its turn in the CPU.

A thread becomes ***"Not Runnable"*** when one of the following events occurs:

- its "sleep" method is invoked

- the thread calls the "wait" method to wait for a specific condition to be satisfied;
- the thread is blocking on I/O.

For example, if a thread has been put to sleep, then the specified number of milliseconds must elapse before the thread becomes "Runnable" again.

The following list describes the escape route for every entrance into the "Not Runnable" state:

- If a thread has been put to sleep, then the specified number of milliseconds must elapse
- If a thread is waiting for a condition, then another object must notify the waiting thread of a change in condition by calling notify or notifyAll. More information is available in "Synchronizing Threads" section.
- If a thread is blocked on I/O, then the I/O must complete.

There are two methods to stop a thread:

- a run method may terminate naturally.

For example, the run method may incorporate a finite while loop – the thread will iterate a predefined number of times and then the run method exits.

```
. . .
      public void run()
       {
       int i = 0;
       while (i < 10)
        {
        . . .
        i++;
        }
       }
      . .
```

- a thread may define a special public method to be used to "stop" the thread.

```
public void stop()
       {
       thread_ = null;
       }
```

4.3 Thread Priority

Conceptually, threads run concurrently. Practically, this is not entirely correct. Most computer configurations have a single CPU, so threads actually run one at a time in such a way as to provide an ***illusion of concurrency***.

Execution of multiple threads on a single CPU, in some order, is called ***scheduling***. Java supports a very simple, deterministic scheduling algorithm known as fixed priority scheduling. This algorithm schedules threads based on their priority relative to other runnable threads.

When a Java thread is created, it inherits its priority from the thread that created it.

A thread's priority can be altered at any time using the ***setPriority*** method. Thread priorities are integers. The higher integer means the higher priority. At any given time, when multiple threads are ready to be executed, the runtime system chooses the runnable thread with the highest priority for execution.

A lower priority thread is running only when all higher priority threads stop, or becomes not runnable for some reason.

Thus, a chosen thread runs until one of higher priority thread becomes runnable, or it is stoped. Then the second thread is given a chance to run, and so on, until the interpreter exits.

The thread priority may be set dynamically, if it is predefined by corresponding class definition.

```
public class myThread implements Runnable
    {
      private String name_;
      private Thread thread_ = null;
      public myThread(String name, int priority)
      {
      name_ = name;
      priority_ = priority;
      }
    . . .
    public void run() { . . .}
    // Inteface to start the thread
    public void start(){
      if(thread_ == null){
        thread_ = new Thread(this);
    // thread is created
        thread_.setPriority(priority_);
         thread_.start();}}
    // thread is running with priority ("priority_")
    //
    // Inteface to stop the thread
    public void stop(){thread_ = null;}
    }
```

Note, that the myThread's constructor takes now an integer argument for setting the priority of its thread. We set the thread's priority in the start method.

```
myThread myThreadOne = new myThread("myThreadOne",1);
myThread myThreadTwo = new myThread("myThreadTwo",2);
```

Suppose, the run method is defined as follows:

```
public void run()
      {
      int i = 0;
      String r = "";
      while ((thread_ == Thread.currentThread()) && (i < 10))
        {
        r = r + i + " " + name_;
        i++;
        }
      System.out.println(r);
      }
```

The while loop in the run method is in a tight loop. Once the scheduler chooses a thread with this thread body for execution, the thread never voluntarily relinquishes control of the CPU – the thread continues to run until the while loop terminates naturally or until the thread is preempted by a higher priority thread. Threads demonstrating such behaviour are called ***selfish threads***.

In some situations, having selfish threads doesn't cause any problems because a higher priority thread preempts the selfish one. However, in other situations, threads with CPU-greedy run methods, can take over the CPU and cause other threads to wait for a long time before getting a chance to run.

Some systems, such as Windows NT/2000/XP, fight selfish thread behavior with a strategy known as ***time slicing***. Time slicing comes into play when there are multiple "Runnable" threads of equal priority and those threads are the highest priority threads competing for the CPU.

```
myThread myThreadOne = new myThread("myThreadOne",2);
myThread myThreadTwo = new myThread("myThreadTwo",2);
```

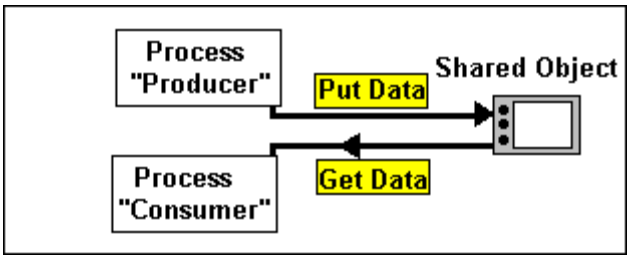
Now the "myThreadOne" and "myThreadTwo" threads will be subject of time slicing algorithm, if such algorithm is supported by the current implementation of the Java Virtual Machine.

4.4 Synchronizing Threads

In a general case, concurrent threads share data and analyze activities of other threads.

One such set of programming situations are known as producer/consumer scenarios where the producer generates a stream of data that then is consumed by a consumer.

In Object-Oriented programming, sharing data is equivalent to ***sharing an object***.




```
public class Producer
   extends Thread {
private IS storage_;
  public Producer(IS storage)
  {
  storage_ = storage;
  }
  public void run()
  {
  for (int i = 0; i < 10; i++)
    {
    storage_.put(i);
    System.out.println
    ("Producer put: " + i);
    }
  }
}
public class Consumer
    extends Thread {
private IS storage_;
  public Consumer(IS storage)
  {
  storage_ = storage;
  }
  public void run()
  {
  int value;
  for (int i = 0; i < 10; i++)
    {
    value = storage_.get();
    System.out.println
    ("Consumer got: " + value);
    }
  }
}
```

```
public class IS
{
private int value_;
public void put(int value){value_ = value;}
public int get(){return value_;}
}
```

The main program starting all the threads and creating a shared instance of IS might look as follows:

```
IS storage = new IS();
Producer producer = new Producer(storage);
Consumer consumer = new Consumer(storage);
producer.start();
consumer.start();
```

If we run the above example we might notice the following problems. One problem arises when the Producer is quicker than the Consumer and generates two numbers before the Consumer has a chance to consume the first one. Thus the Consumer would skip a number. Part of the output might look like this:

```
Producer put: 0
Producer put: 1
Consumer got: 1
```

Another problem that might arise is when the Consumer is quicker than the Producer and consumes the same value twice. In this situation, the Consumer would print the same value twice and might produce output that looked like this:

```
Producer put: 0
Consumer got: 0
Consumer got: 0
```

Either way, the result is wrong. You want the Consumer to get each integer produced by the Producer exactly once.

Problems like this one are called ***race conditions***. They arise from multiple, asynchronously executing threads trying to access a single object at the same time and getting the wrong result.

The activities of the Producer and Consumer must be synchronized in two ways:

- two threads must not simultaneously access the shared object. A Java thread can prevent such simultaneous access by locking an object.
  When an object is locked by one thread and another thread invokes a synchronized method on the same object, the second thread will block until the object is unlocked.

- two threads must coordinate their Put/Get operations. For example, the Producer indicates that the value is ready, and the Consumer indicate that the value has been retrieved.

The Thread class provides a collection of methods - ***wait***, ***notify***, and ***notifyAll*** - to build such co-ordination mechanisms.

The code segments within a program that access the same object from separate, concurrent threads are called ***critical sections***.

A critical section is a block or a method identified with the ***synchronized*** keyword. Java associates a lock with every object that has synchronized code.

```
public class IS
{
   private int value_;
   public synchronized void put(int value){value_ = value;}
   public synchronized int get(){return value_;}
}
```

Note that the method declarations for both put and get contain the synchronized keyword. Hence, the system associates a unique lock with every instance of IntegerStorage (including the one shared by the Producer and the Consumer).

Now, we want the Consumer to wait until the Producer puts a value into the IS and the Producer must notify the Consumer when it happened.

Similarly, the Producer must wait until the Consumer takes a value (and notifies the Producer) before replacing it with a new value.

```
public class IntegerStorage
{
      private int value_;
      private boolean available_ = false;
      public synchronized void put(int value){
      while(available_)
      {
      try {wait();}
// wait for Consumer to get value
      catch (InterruptedException exc) {exc.printStackTrace();}
      }
      value_ = value;
      available_ = true;
// notify Consumer that value has been set
      notifyAll();
}
public synchronized int get()
      {
      while(!available_){try {wait();}
// wait for Producer to put value
      catch (InterruptedException exc) {exc.printStackTrace();}
      }
      available_ = false;
// notify Producer that value has been retrieved
      notifyAll();
      return value_;
}
}
```

5 References

- **Head First Java** By *Kathy Sierra, Bert Bates*
- **Java Programming Language** By *Ken Arnold, James Gosling, David Holmes*
- **O'Reilly® Programming the Be Operating System (Dan Sydow)**