Software

System
- Background
- Not a lot of user interaction

Application
- Foreground
- user input

Real Time System
- Response time is important
- Real world unexpected

Hard:
- Time critical
- Worst case
- pacemaker

Soft:
- Won't crash sys!

Event Driven: Needs input (Button, polling)
Time Driven: Set schedule

Event - Driven

Advantages of interrupts:
- system is notified immediately of event, so can be handled immediately
- deterministic response times
- interrupts can be prioritized
- response time is independent of number of sensors

Disadvantages of interrupts:
- complex hardware is needed (sensors, system-level handlers for ISRs, etc)
- difficult to test and debug
  - interrupts are unpredictable, ISR and main code run asynchronously

Advantages of polling:
- Simpler code, more flexibility
- Easier to debug since event detection is predictable
  - Sensor events may be missed
  - Difficult to prioritize events
  - Adding more sensors degrades performance and response times
  - Consumes a lot of CPU time, particularly if events are few and far between

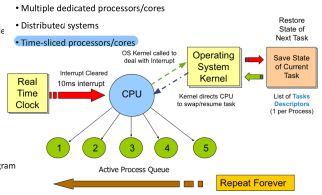
Multi-Tasking

Speed-up Factor: $S_p = \frac{T_1}{T_p}$

Advantages:
- greater flexibility – system can be distributed across several servers
- greater scalability – a single program can be instantiated as many times as needed

Disadvantages:
- challenges in decomposing system into appropriate concurrent units
- challenges in communication between parallel units
- challenges in synchronization between parallel units
- challenges in debugging and testing

4 Multi-Tasking Implementations
- Pseudo-multitasking
- Multiple dedicated processors/cores
- Distributed systems
- Time-sliced processors/cores



Processes vs Threads

Advantages of Processes:
- If one process fails, others continue
- New process instances can be started manually, or outside of a main program
- Processes can be run on separate machines

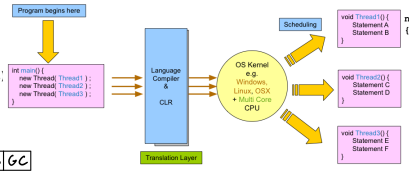
Disadvantages of Processes:
- Less efficient than threads due to additional overhead (more memory, state info)
- Communication/synchronization happens across process boundaries

Creating a Process (Windows)

```
namespace MyProcess
{
    class MyProcess
    {
        public static void Main()
        {
            using (Process myProcess = new Process())
                {
                    myProcess.StartInfo.UseShellExecute = false;
                    myProcess.StartInfo.FileName = "C:\\HelloWorld.exe";
                    myProcess.StartInfo.CreateNoWindow = true;
                    myProcess.Start();
```

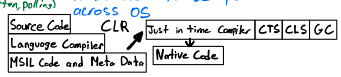
CPEN 333

A **kernel** is the **heart** of an **operating system**, consists of a collection of applications, services and **software system calls** that implement
- Creating processes/threads
- Mutual exclusion,
- Process communication,
- Process synchronisation,
- Sharing memory,
- Scheduling and prioritization of processes,
- Handling the GUI (windows, mouse, keyboard etc.,
- Dealing with files & directories, networks etc.



Library Abstractions

- Allows code to be portable across OS



Exceptions

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            string s = null;
            double numerator = 10; double denominator = 0;
            if (s == null)
                throw new NullValueCustom();
            if (denominator == 0)
                throw new NotDividbyZero();

        }
        catch(NullValueCustome)
        {
            Console.WriteLine("My custom Exception says no Null Value");
        }
        catch(NotDividbyZero)
        {
            Console.WriteLine("Denominator Cannot be zero");
```

Custom Exception in C#

```
is Program

static void Main(string[] args)
{
    try
    {
        string s = null;
        if (s == null)
            throw new NullValueCustom();
    }
    catch
    {
        Console.WriteLine("My custom Exception says no Null Value");
    }
```

```
public class NullValueCustom : Exception
{
    public NullValueCustom()
    {
    }
```
Your code here

Specifications

Preconditions: set of conditions that must be satisfied by the input arguments. The onus is on the caller to guarantee these, otherwise the behaviour of the code is left unspecified.

Postconditions: if the preconditions hold, precisely specifies the outputs: which variables are modified, if exceptions are thrown, and any other *effects*. The implementer has the obligation to guarantee these.

```
// REQUIRES:  any array data, integer val
// EFFECTS:   returns the index of the first occurrence
//            of val in data, throws an ItemNotFound
//            exception if not found
int find(int[] data, int val);
```

Validation includes:
- **Formal reasoning**: also called verification, constructs a formal proof that the program is correct.
- **Code review**: code "proof-reading" by a colleague or peer to look for bugs/inefficiencies.
- **Testing**: running the program on *carefully* selected inputs and checking the results against known answers.

Testing

Unit Testing: testing each method/class (i.e. a unit) in isolation. Each partition of inputs is usually separated into a separate test so that a failed test will indicate exactly what type of bug to search for.

Integration Testing: testing the system as a whole to ensure the various components of the software function properly in combination. If all unit tests pass, then you know the issue is somewhere in the interfacing or timings.

Regression Testing: testing the system to ensure it performs exactly as it did in a previous functional version. Often important when adding new features or optimizing parts of code, making sure you don't break any part that works.

Critical Section

```
void threadinc(ref int counter) {
  for (i=0; i<5000000; ++i) {
    counter +=1;
  }
}
```
- Reads the current value of counter
- Adds 1 to that value
- Stores the new value back to counter

X3 steps

Atomic Operations

1. Reads and writes of the following data types are atomic: bool, char, byte, sbyte, short, ushort, uint, int, float, and reference types.
2. Reads and writes of enum types with an underlying type in the list are also atomic.
3. Reads and writes of other types, including long, ulong, double, and decimal, as well as user-defined types, are not guaranteed to be atomic.

Mutual Exclusion

assigning a constant byte value	`char x = 'x';`	YES
assigning a constant integer value	`int y = 10;`	YES
copying an integer value	`int z = y;`	No
incrementing an integer	`z += 1;`	No

- Mutex Class
- Interlocked Class
- SpinLock Struct
- ReaderWriterLockSlim
- AutoResetEvent Class

Mutex

```
void threadinc(ref int counter)

// increment
for (int i=0; i<5000000; ++i) {

lock:   // wait until not being used
        mut.WaitOne()

        counter += 1;

unlock: // clear to let others access counter
        mut.ReleaseMutex();
```

InterLocked Class

```
void threadinc(ref int counter)

// increment
lock:  for (int i=0; i<5000000; ++i) {

       Interlocked.Increment (ref counter);
```

SpinLock Struct:

```
void threadinc(ref int counter) {
  bool gotLock = false;
  for (int i=0; i<5000000; ++i) {
    gotLock = false;
    try{
lock:   sl.Enter(ref gotLock);
        counter += 1;
        }
finally{
unlock:  // Only give up the lock if you actually acquired it
         if (gotLock) sl.Exit();
        }
}
```

IPC: Shared Memory Types

Persistent Shared Memory-Mapped Files View: Random access
- The CreateFromFile methods create a memory-mapped file from an existing file on disk.
- One application is to create a memory-mapped view of a part of an extremely large file and manipulates a portion of it.

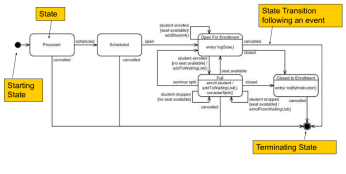
```
// Create the memory-mapped file.
using (var mmf = MemoryMappedFile.CreateFromFile(@"c:\ExtremelyLargeImage.data", FileMode.Open, "ImgA"))
{
    // Create a random access view, from the the offset
    // to the the offset plus length
    using (var accessor = mmf.CreateViewAccessor(offset, length))
    {
        int colorSize = Marshal.SizeOf(typeof(MyColor));
        MyColor color;

        // Make changes to the view.
        for (long i = 0; i < length; i += colorSize)
        {
            accessor.Read(i, out color);
            color.Brighten(10);
            accessor.Write(i, ref color);
```

Non-Persistent Shared Memory-Mapped Files View: Stream Access
- The CreateNew and CreateOrOpen methods create a memory-mapped file that is not mapped to an existing file on disk.
- Commonly used for interprocess communications

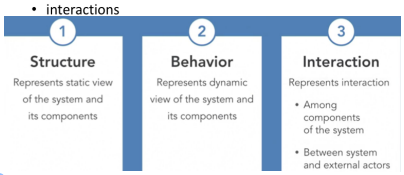
```
using (MemoryMappedFile mmf = MemoryMappedFile.CreateNew("testmap", 10000))
{
    bool mutexCreated;
    Mutex mutex = new Mutex(true, "testmapmutex", out mutexCreated);
    using (MemoryMappedViewStream stream = mmf.CreateViewStream())
    {
        BinaryWriter writer = new BinaryWriter(stream);
        writer.Write(1);
    }
    mutex.ReleaseMutex();
    Console.WriteLine("Start Process B and press ENTER to continue.");
    Console.ReadLine();
    Console.WriteLine("Start Process C and press ENTER to continue.");
    Console.ReadLine();
    mutex.WaitOne();
```

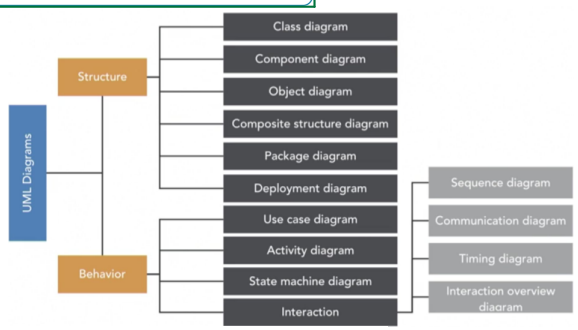
State Chart Diagram



UML: Unified Modelling Language

Its main purpose is to help communicate and plan out
- structure
- behaviour
- interactions





Structure:
- **Class Diagrams:** outline the different entities in a system, and their relationships with each other. It shows the structural breakdown of the software.
- **Deployment Diagrams:** show where each of your software modules are deployed in the physical system and how they communicate.

Behaviour:
- **Use-Case Diagrams:** document high-level functional requirements, and relationships with users and other systems (actors). These outline every observerable function your system must perform.

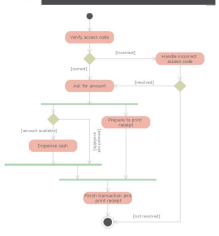
Behaviour (cont.):
- **State Diagrams:** outline the time-dependent changes in state and transitions of each major object or interaction in your system.
- **Activity Diagrams:** model high-level activities and transitions between system states, shows concurrency of activities.

Behavior-Interaction:
- **Sequence Diagrams:** show the detailed flow of execution of events, and relative timings between them; these model the behaviour and the interactions between collaborating objects.

Activity Diagram

Show the procedural flow of control while processing an activity, modelling the logic in a use-case or use-case scenario.

Green bars indicate creation and joining of parallel sections.



Creating a Thread

```
using System;
using System.Threading; Using System.Threading Namespace

namespace concurrency Namespace
{
    class Program Class
    {
        static void Main(string[] args) Entry Point
        {
            for (int i = 0; i < 2; i++)2 new threads spun
            {
                Thread thread = new Thread(MyProcess);
                // worker threads
                thread.Start(); Thread is started
            }
            Console.WriteLine("A process that takes 4s!");
            Thread.Sleep(4000);
            Console.WriteLine("Done!");
        }
        private static void MyProcess(object obj)
        {
            Console.WriteLine("A process that takes 4s!");
            Thread.Sleep(4000);
            Console.WriteLine("Done!");
        }
    }
}
```

New Instantiation — Constructor Function. New created thread will execute MyProcess

Join() : wait for thread(s) to complete

- Main program waits for all threads to finish before proceeding in Main

Non-Static Method – Have to create an object of that class to call that method

```
using System;
using System.Threading;

public class NonStaticMethodMultiThreading
{
    public static void Main()
    {
        SomeClass obj = new SomeClass();
        SomeClass th = new Thread(obj.SomeWork);
        th.Start();
        Thread.Sleep(1000);

        th.Join();

    }
}
```

```
public class SomeClass
{
    public void SomeWork()
    {
        var sw = Stopwatch.StartNew();

        do
        {
            Console.WriteLine("Thread {0}: Elapsed {1:N2}(s)",
                              Thread.CurrentThread.ManagedThreadId,
                              sw.ElapsedMilliseconds / 1000.0);

            Thread.Sleep(500);
        } while (sw.ElapsedMilliseconds <= 5000);
        sw.Stop();
    }
}
```

Static Method - Don't have to create an object of that class to call that method

- Can be accessed by other classes from other Namespaces

```
using System;
using System.Diagnostics;
using System.Threading;

public class StaticMethodMultiThreading
{
    public static void Main()
    {
        var th = new Thread(SomeClass.SomeWork);
        th.Start();
        Thread.Sleep(1000);

        th.Join();

    }
}
```

```
public class SomeClass
{
    public static void SomeWork()
    {
        var sw = Stopwatch.StartNew();

        do
        {
            Console.WriteLine("Thread {0}: Elapsed {1:N2}(s)",
                              Thread.CurrentThread.ManagedThreadId,
                              sw.ElapsedMilliseconds / 1000.0);

            Thread.Sleep(500);
        } while (sw.ElapsedMilliseconds <= 5000);
        sw.Stop();
    }
}
```

Method

Thread Communication – Shared Memory

```
using System;
using System.Threading;

namespace SharedResources
{
    class Program
    {
        static static bool isCompleted;
        static readonly object lockCompleted = new object();

        static void Main(string[] args)
        {
            Thread thread = new Thread(HelloWorld);
            //Worked Thread
            thread.Start();
            //Main Thread
            HelloWorld();
        }
```

Need this

```
        private static void HelloWorld()
        {
            lock (lockCompleted)
            {
                if (!isCompleted)
                {
                    Console.WriteLine("Hello World should print only once");
                    isCompleted = true;
                }
            }
        }
    }
}
```

Only 1 thread is allowed in while Locked

Thread Pool

```
using System;
using System.Threading;
public class Example
{
    public static void Main(
    {
        ThreadPool.QueueUserWorkItem(ThreadProc);
        Thread.Sleep(1000);
        Console.WriteLine("Main thread exits.");
    }

    // This thread procedure performs the task.
    static void ThreadProc(Object stateInfo)
    {
        // No state object was passed to QueueUserWorkItem, so stateInfo is null.
        Console.WriteLine("Hello from the thread pool.");
```

capacity finish executing
- How do you know if Thread Pool is being used?
  - Thread.CurrentThread.IsThreadPoolThread
- How to add to Thread Pool?
  - Task Parallel Library
  - Background Work
  - Asynchronous delegate
  - Call ThreadPool.QueueUserWorkItem*

Message Passing

Pass an argument into Thread's Start method:

```
    Thread t = new Thread (Print);
}
static void Print (object messageObj) {
    string message = (string) messageObj; // We need to cast here
    Console.WriteLine (message);
}
```

Use Cases: ATM

ATM Use cases
- Withdraw Cash
- Request Balance
- Request Statement
- Request Cheques

- Withdraw Cash

Benefit: - User goes home with cash money

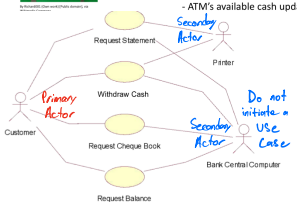
Interaction:
- User identifies him/herself
- Requests amount
- System checks balance and limits
- If valid, dispenses cash, debits account
- Asks user if would like receipt

Effects: - User's account balance updated
- ATM's available cash updated

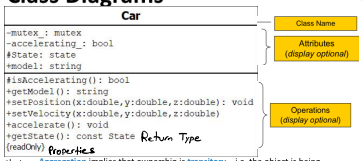
Start of Primary scenario/transaction
1. The user inserts their ID card into the system.
2. The system reads the magnetic strip from the card.
3. If the system cannot read the card then **<<Scenario 1>>**
4. The system contacts the banks central computer to request the PIN number for the card and their account details.
5. If bank central computer cannot access users account then **<<Scenario 2>>**
6. The system prompts the user for their PIN.
7. The user enters their PIN.
8. If PIN cannot be authenticated **<<Scenario 3>>**
9. The user is prompted for the amount of the withdrawal.
10. The user enters the amount of withdrawal.
11. The system checks with the banks central computer
12. If the user has insufficient funds **<<Scenario 4>>**
13. The cash is dispensed and the customer's account at the Bank Central Computer is debited with the withdrawal amount.
14. The card is returned to the user and a receipt issued.

End-Of-Transaction

Scenario 1: The users card is returned. End-of-Transaction
Scenario 2: The users card is returned. End-of-Transaction
Scenario 3: The user is given two more attempts to enter a correct PIN. If this fails the card is kept and the transaction ends. Otherwise resume primary scenario.
Scenario 4: The user is given the opportunity to enter a lesser amount or cancel the transaction. If cancel is chosen, the card is returned and the transaction ends. If the lesser amount is acceptable then resume primary scenario.

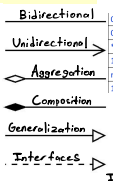


Class Diagrams



Attribute Syntax:
visibility name: type multiplicity = default {properties}
- **visibility**: public (+), protected (#), or private (-)
- **name**: attribute name
- **type**: type or class of the attribute
- **multiplicity**: number of elements (e.g. size of an array)
- **default**: default value if isn't specified in constructor
- **properties**: any additional properties

Operation Syntax:
visibility name (parameters) : return-type {properties}
- **visibility**: public (+), protected (#), or private (-)
- **name**: attribute name
- **parameters**: method arguments
  - **direction**: in, out, or inout
- **return-type**: return type or class of the operation
- **properties**: any additional properties



0..1	Could be 0 or 1 objects (**caution**: could be 0).
0..*	Could be 0 or more objects (**caution**: could be 0).
*	Unknown many (**caution**: could be 0).
1	Guaranteed **always** to be exactly 1 object.
n	Guaranteed **always** to be exactly 'n' objects (e.g. 5).
1..*	Unknown many but **at least** 1 and maybe more objects.

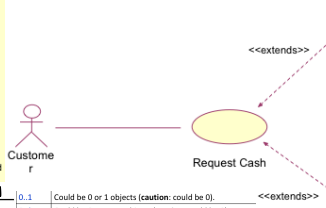
Association: describes a *makes-use-of* or *calls-upon* relationship, implies that one class contains a reference to and can send messages to another.
Encapsulation: captures the *part-of* or *belongs-to* relationship. It implies that one class owns or controls another. There are two 'types': aggregation and composition.
Generalization: describes the *kind-of* relationship, whether a class can be substituted for or inherits from another.

Aggregation implies that ownership is transitory – i.e. the object is being "borrowed". The object can be detached from the owner and given to another one at run-time.
Composition implies a stronger ownership than aggregation: the owner and parts have the same lifetime, implying the owner is responsible for creation/deletion of resource.

Generalization
Describe the kind-of or substitutes-for relationship. This implies two things:
- **Inheritance**: derived class inherits all attributes/operations of the base
- **Substitutability**: anywhere the base is used, the derived class can also be used

Interfaces
An interface describes the contract a class must abide by, but has no explicit inheritance of attributes or operations. An implementation class *implements* the interface.

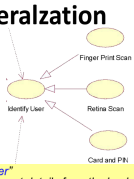
Extends



Use-Case Request Cash
1. . . .
2. . . .
3. If(users wants to chose type of $ notes)
4. Extend use case : Choose $$$ Notes
5. If user chooses to have a receipt
6. Extend use case : Print Receipt
7. . . .
8. . . .
9. The card is returned to the user and a receipt is issued.

End-Of-Transaction

Generalzation



"Identify the user"
obtain their account details from the bank central computer

Includes

1. Include ***Identify User***