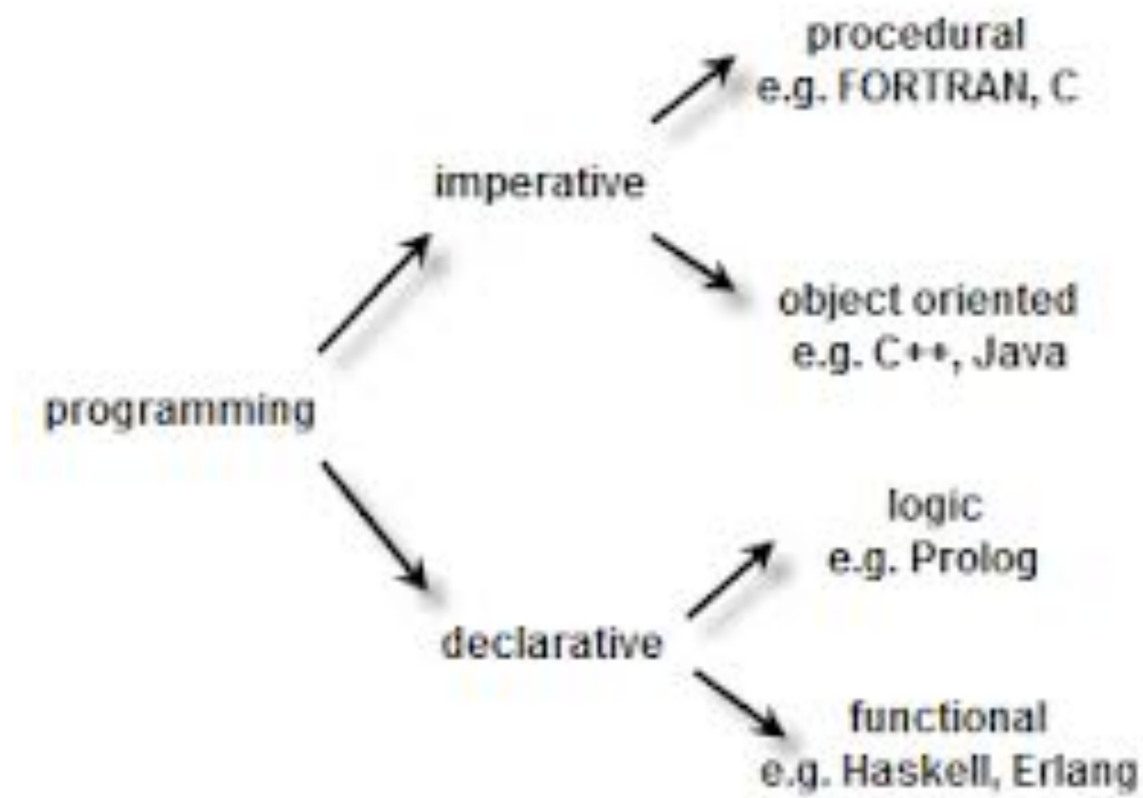
Embedded Systems
Lecture 5: Imperative Programming Languages

Michael O'Boyle
University of Edinburgh

Overview

- Desirable features in a programming language
- Comparison by language
    - Parallelism and Communciation
        - Tasks and Message passing
        - Threads and shared memory
    - Determinancy
- Summary



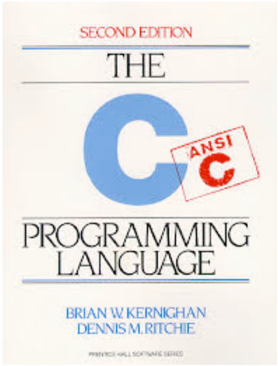
Translating design into software

- Embedded systems are processor based
  - Execute machine code instructions compiler from high level programming languages
- Design has to embodied in a language as in all software development
  - Embedded and real time constraints add complexity to any programming language
  - Most popular are imperative languages with special provision for time and concurrency

Models of computation

Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous	
Undefined components	Plain text, use cases | (Message) sequence charts		
Communicating finite state machines	StateCharts		SDL
Data flow	Scoreboarding, Dataflow architectures		Kahn networks
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC*, …	Only experimental systems, e.g. distributed DE in Ptolemy	
Imperative (Von Neumann) model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	

Common languages and features

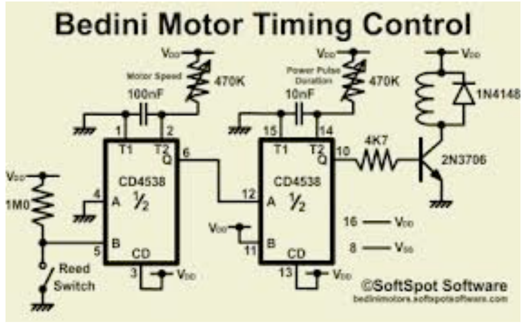


- Focus on just three languages C, Java and Ada.
- C - currently the most popular language used
  - Lacks support for embedded software development
  - Makes direct use of very low level posix threads. Little support for abstraction and exceptions
- Java - de facto standard for programming desktop applications
  - Explicit support for modules concurrency and exceptions
  - Problems for embedded s/w - unpredictability and lack of direct control
  - Real Time Java tries to overcome this
- Ada - used in safety-critical applications
  - Programming in the large and code reuse
  - Tasking Features for concurrency. High level exceptions. Real time facilities

Desirable features

Time access and control:

- Mechanisms/primitives for dealing with absolute & relative time to control & monitor program timing behaviour
- Basic operations: set a clock or timer, read value of timer object
- Higher-level - instructions to delay a task, generate timeout signals

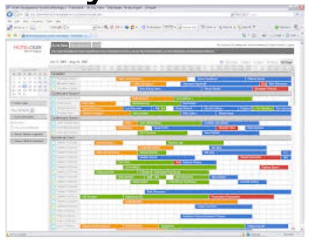


Exception Handling:

- Unusual behaviours in both h/w & s/w should be detected & handled gracefully
- Should also be easy to distinguish between unusual & normal ones
- Useful language structures: define, test and recover from exceptions

Software Management:

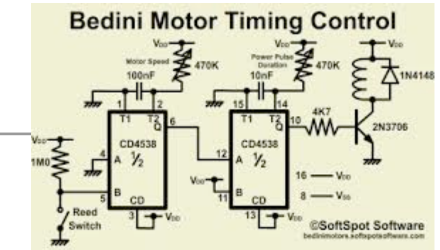
- Embedded software is complex - large amount of code, a variety of activities & requirements
- Language features must provide help with key to managing complexity of large embedded systems i.e. decomposition & abstraction



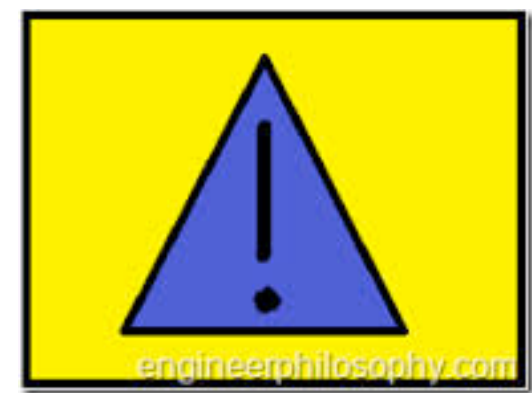
Parallelism and determinancy

- Embedded system/real world is inherently parallel. Deadlock and race conditions a real problem
- Is program behaviour predictable and repeatable? A problem for parallel systems

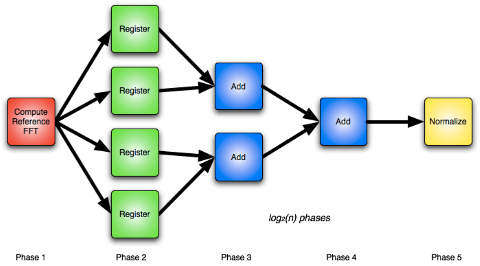
Comparison by certain features



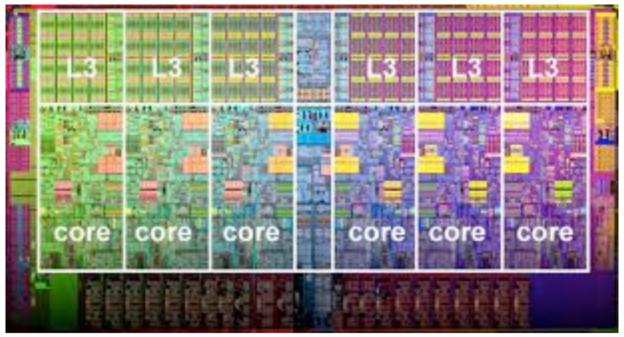
- Time access and control
  - Ada comprehensive set of timing packages. Calendar and Real-time. Delay function
  - Java elaborate Date class. Coarse clock granularity - but Real Time Java has access to a nanosecond clock
  - C standard libraries for interfacing to calendar time. Posix thread library or pthreads has a   nano second clock
- Exceptions
  - Ada has clean scheme for declaring, raising and handling
  - Java extends this and integrates in OO model. C has none
- Abstraction
  - Ada and Java support modules in form of packages
  - C does not really apart from separate compilation of files
- Real issues: parallelism and determinancy



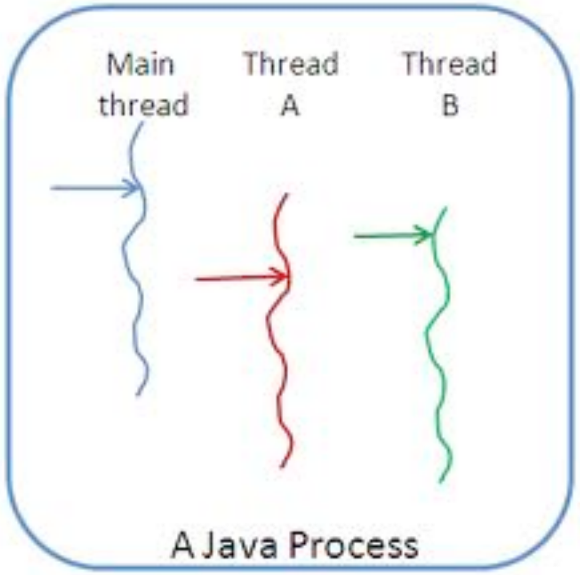
Parallelism and Communication



- Concurrency control: inherent feature of embedded systems
  - Software constructs for defining, synchronising, communication among parallel activities & scheduling their execution
  - In addition, to above higher level facilities, need mechanisms for finer degree of h/w control and timing
    - e.g. declarations or statements that directly deal with interrupts, IO, etc.
- Java provides threads and shared memory plus synchronisation
- C has to incorporate real-time POSIX primitives (fork, wait, spawn, etc.) for concurrency.
  - Can have either shared memory or use message-passing via MPI
- Ada provides tasks and uses a message-passing approach



Parallel Java threads

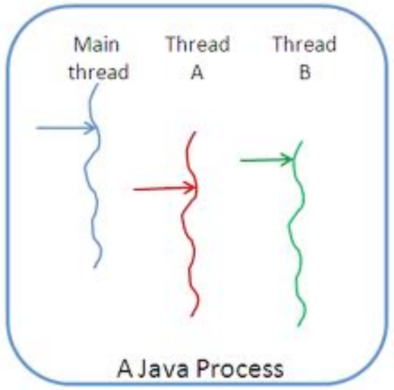


- Threads are the active objects of concurrency
    - Threads are derived from the Java `Threads` class
- A Thread can be specified by subclassing the `Thread` class with the `extends` keyword & specifying a `run` method for it
    - The `run` method contains the thread's executable code
- A thread is activated by calling the `start` method, which invokes its `run` method
    - e.g. `Producer.start();` makes `Producer` ready for execution

Parallel Java threads

- To avoid all threads having to be child classes of `Thread`, Java also provides a standard interface called `Runnable`

```
public   interface Runnable{
         public abstract void run();
         }
```

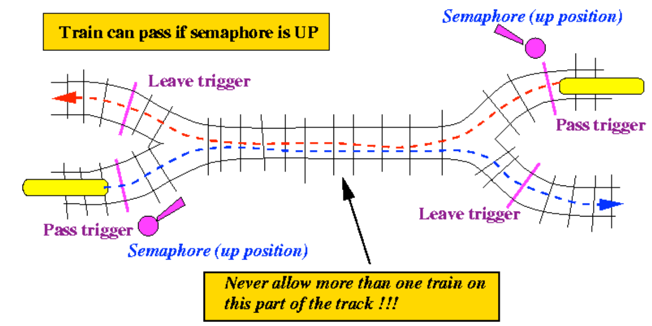


- Any class which wishes to express concurrency must implement this interface & provide the `run` method.

- The `join` method is available from the `Thread` class for managing threads

    - e.g. the thread `Process_Data`, which needs to wait for thread `Get_Data` to terminate before it can continue, must call: `Get_Data.join();`

Parallelism in C using Pthread library




```
#include <stdio.h>
#include <pthread.h>
main()  {
  pthread_t f2_thread, f1_thread;
  void *f2(), *f1();
  int i1,i2;
  i1 = 1;
  i2 = 2;
  pthread_create(&f1_thread,NULL,f1,&i1);
  pthread_create(&f2_thread,NULL,f2,&i2);
  pthread_join(f1_thread,NULL);
  pthread_join(f2_thread,NULL);
}
void *f1(int *x){
  int i = *x;
  sleep(1);
  printf("f1: %d",i);
  pthread_exit(0);
}
void *f2(int *x){
  int i = *x;
  sleep(1);
  printf("f2: %d",i);
  pthread_exit(0);
}
```

What happens if f1 and f2 try to write to the same variable y?

```
main () {
int y;
...
pthread_create(...f1,&y);
pthread_create(...f2,&y);
pthread_join(...);
pthread_join(...);
printf("f1: %d",y);
}

void *f1(int *x,*y){
  *y=1;
  pthread_exit(0);
}
void *f2(int *x,*y){
  *y=2;
  pthread_exit(0);
}
```

Race condition!!

Parallel tasks in Ada

```
procedure example1 is
    task a;
    task b;
    task body a is
     -- local declarations for a
     begin
      -- statements for a
     end a;
    task body b is
     -- local declarations for b
     begin
      -- statements for b
     end b;
begin
 -- Tasks a and b will start before the first
 -- statement of the body of example1
end;
```

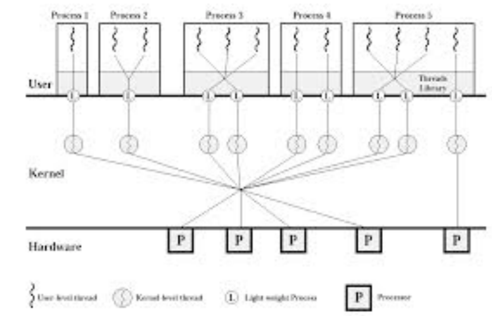
Communication: Shared memory and synchronisation: Java and synchronized methods

```
public class SynchronizedCounter{
  private int c=0;
  public synchronized void incr(){
     c++;
  }
  public synchronized void decr(){
     c--;
  }
 }

new Thread(...t.incr()...).start();
new Thread(...t.decr()...).start();
```

Synchronized methods prevent race condition. However if synchronized method requires interaction from another thread, it may lead to deadlock

Communication: Shared memory and synchronisation: C and pthreads



```
main () {
 int y=1;
 ...
 pthread_create(...f1,&y);
pthread_create(...f2,&y);
pthread_join(...);
pthread_join(...);
printf("f1: %d",y);
}

void *f1(int *x,*y){
 *y=1;
 pthread_exit(0);
}
void *f2(int *x,*y){
*y=2;
pthread_exit(0);
}
```

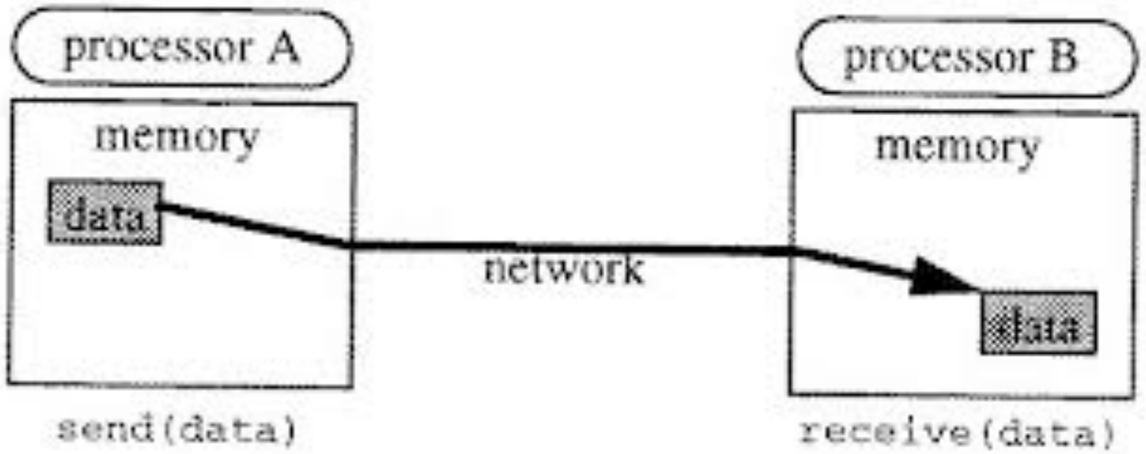
```
main () {
 int y=1;
 ...
 pthread_create(...f1,&y);
pthread_join(...);
pthread_create(...f2,&y);
pthread_join(...);
printf("f1: %d",y);
}

void *f1(int *x,*y){
 *y=1;
 pthread_exit(0);
}
void *f2(int *x,*y){
*y=2;
pthread_exit(0);
}
```

Can use join as a way of ordering. Mutual exclusion allows more efficient but complex and error prone codes `pthread_mutex_lock()`, `pthread_mutex_unlock()`

Communication: Message- Passing

- One of the two approaches to communication
- Assumes no shared state between tasks/processes. One task cannot refer to or access variables in another task - they are not in scope
  - Instead send and receive messages via a channel or pipe
- Key issue is whether synchronous or asynchronous. Can lead to deadlock
  - CSP: communicating synchronous processes [1985] is the originator followed by occam.
- MPI widely used in HPC. Ada uses it too



Synchronous message passing: CSP

- Communicate by shared channels c and d

```
process A
..
var a ...
  a:=3;
  c!a; -- output
end
```

```
process B
..
var b ...
  ...
  c?b; -- input
end
```

No race conditions (!)

But can deadlock

```
process A
var a ...
  c!a; -- output
  d?a; --input
end
```

```
process B
var b ...
  d!b; -- output
  c?b; -- input
end
```

Synchronous message passing: Ada-rendez-vous

```
task screen_out is
 entry call_ch(val:character; x, y: integer);
 entry call_int(z, x, y: integer);
end screen_out;
task body  screen_out is
...
 select
  accept call_ch ...  do ..
  end call_ch;
 or
  accept call_int ... do ..
  end call_int;
 end select
```

Sending a message:

```
begin
 screen_out.call_ch('Z',10,20);
 exception
  when tasking_error =>
             (exception handling)
end;
```

Predictability

Programs must be both functionally predictable and timing predictable

- Timing predictability implies well-defined timing characteristics for constructs, which are statically derivable
- Languages overloaded with facilities & special cases usually too complex to satisfy predictability requirements

Ada 95 standard has been specifically proposed with predictability of tasking & timing features in mind

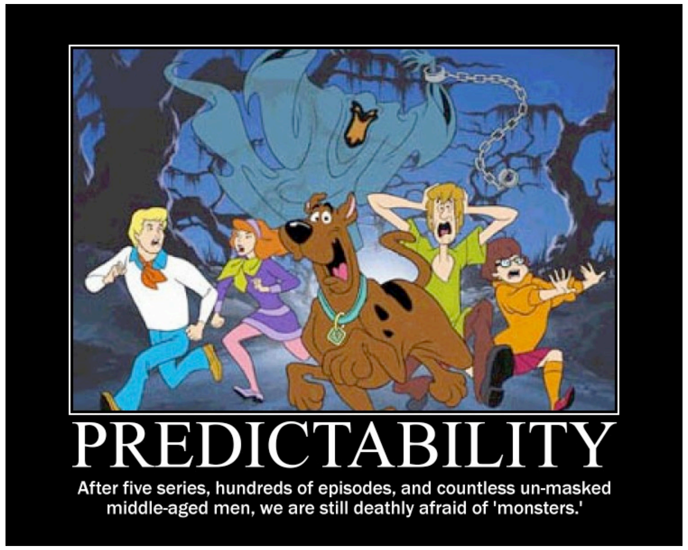
- Features such as recursion & dynamic data structures lead to unpredictable timing
- e.g. dynamic storage management & garbage collection

Java is highly unpredictable

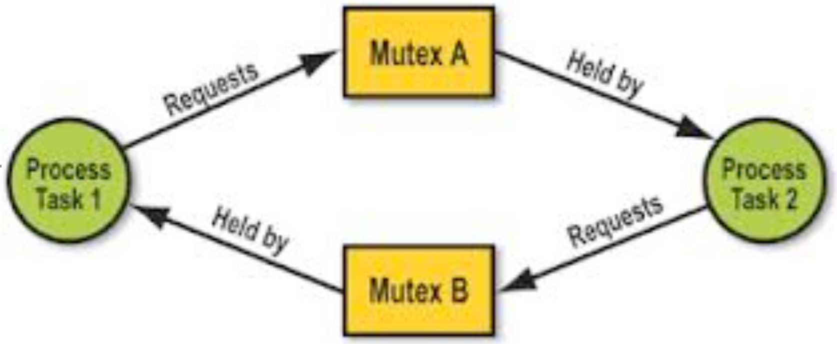
- Garbage collection and dynamic compilation makes perfromance prediction extremely difficult
- Real time Java proposed as a way to overcome this

C potentially unpredictable

- Unrestricted use of dynamic memory allocation the main problem



Problems with imperative languages and shared memory



- Potential deadlocks
  - Specification of total order of operations is an over- specification. A partial order would be sufficient.
  - The total order reduces the potential for optimizations
- Timing cannot be specified
  - Access to shared memory leads to anomalies, that have to be pruned away by mutexes, semaphores, monitors. Messages can be as bad
- Access to shared, protected resources leads to priority inversion
- Termination in general undecidable
  - Preemptions at any time complicate timing analysis

Summary

- Desirable features in a programming language
- Comparison by language
  - Parallelism and Communciation
    - Message passing
    - Threads
  - Determinancy
- Next lecture on embedded hardware