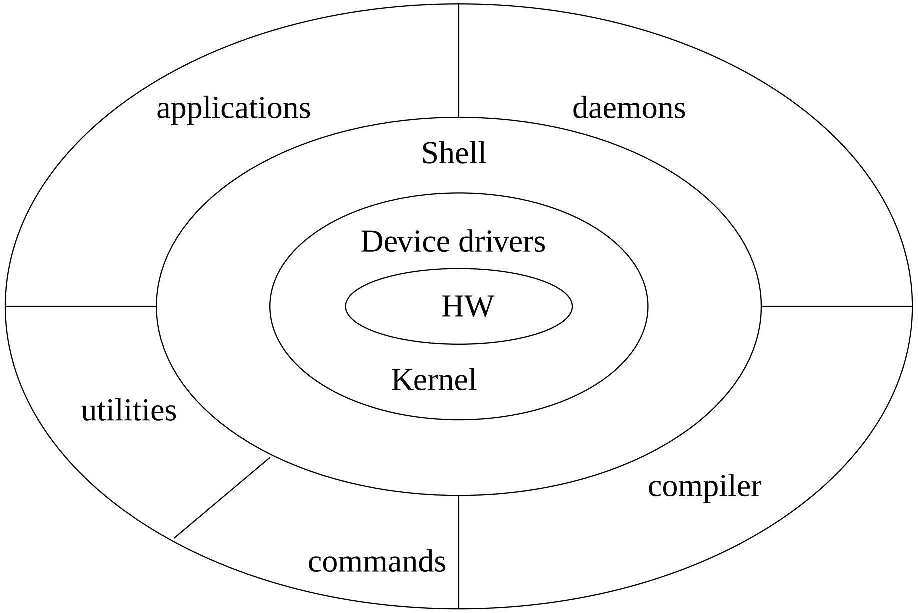
Interrupt handling and context switching

- these two topics are separate and we will examine them in turn

Interrupts

- 

- the user programs and hardware communicates with the kernel through interrupts

Four different kinds of interrupts

- device interrupt, such as a hardware timer, for example the 8253 `counter0` reaching `0` on an IBM-PC
- user code issuing a software interrupt, often called a **system call**
- an illegal instruction (divide by zero, or an opcode which the processor does not recognise)
- or a memory management fault interrupt (occurs when code attempts to read from non existent memory)

First level interrupt handler

- the kernel must detect which kind of interrupt has occurred and call the appropriate routine
  - this code is often termed the **first level interrupt handler**

- the pseudo code for the FLIH follows:

First level interrupt handler

```
save program registers and disable interrupts
k = get_interrupt_kind ();
if (k == source 1) service_source1 ();
else if (k == source 2) service_source2 ();
else if (k == source 3) service_source3 ();
else if (k == source 4) service_source4 ();
else if (k == source 5) service_source5 ();
  etc
restore program registers and enable interrupts
return
```

- you may find the hardware on the microprocessor performs the save and restore program registers and disabling/enabling interrupts
  - possibly by one instruction

First level interrupt handler

- you might also find the hardware enables you to determine the source of the interrupt easily
    - most microprocessors have an interrupt vector table
        - typically one vector per source is implemented

- equally, however the code can be ugly as it depends upon the hardware specifications

Example of FLIH in GNU LuK

- GNU LuK (Lean uKernel) is a very small microkernel which allows premptive processes, interrupt driven devices and semaphores

Example of FLIH in GNU LuK

```
    IsrTemplate[ 0] := 0FCH ;    (* cld (disable interrupts) *)
    IsrTemplate[ 1] := 050H ;    (* push eax *)
    IsrTemplate[ 2] := 051H ;    (* push ecx *)
    IsrTemplate[ 3] := 052H ;    (* push edx *)
    IsrTemplate[ 4] := 01EH ;    (* push ds *)
    IsrTemplate[ 5] := 006H ;    (* push es *)
    IsrTemplate[ 6] := 00FH ;    (* push fs *)
    IsrTemplate[ 7] := 0A0H ;
    IsrTemplate[ 8] := 0B8H ;    (* movl 0x00000010, %eax *)
    IsrTemplate[ 9] := 010H ;
    IsrTemplate[10] := 000H ;
    IsrTemplate[11] := 000H ;
    IsrTemplate[12] := 000H ;
    IsrTemplate[13] := 08EH ;    (* mov  ax, ds *)
    IsrTemplate[14] := 0D8H ;
    IsrTemplate[15] := 08EH ;    (* mov  ax, es *)
    IsrTemplate[16] := 0C0H ;
    IsrTemplate[17] := 08EH ;    (* mov  ax, fs *)
    IsrTemplate[18] := 0E0H ;
```

Example of FLIH in GNU LuK

```
    IsrTemplate[19] := 068H ;    (* push  interruptnumber *)
    IsrTemplate[20] := 000H ;    (* vector number to be overwritten.  *)
    IsrTemplate[21] := 000H ;    (* this is the single parameter.  *)
    IsrTemplate[22] := 000H ;    (* to function.  *)
    IsrTemplate[23] := 000H ;
    IsrTemplate[24] := 0B8H ;    (* movl function, %eax *)
    IsrTemplate[25] := 000H ;    (* function address to be overwritten *)
    IsrTemplate[26] := 000H ;
    IsrTemplate[27] := 000H ;
    IsrTemplate[28] := 000H ;
```

Example of FLIH in GNU LuK

```
    IsrTemplate[29] := 0FFH ;    (* call  %eax *)
    IsrTemplate[30] := 0D0H ;
    IsrTemplate[31] := 058H ;    (* pop  %eax   // remove parameter *)
    IsrTemplate[32] := 00FH ;    (* pop  %fs *)
    IsrTemplate[33] := 0A1H ;
    IsrTemplate[34] := 007H ;    (* pop  %es *)
    IsrTemplate[35] := 01FH ;    (* pop  %ds *)
    IsrTemplate[36] := 05AH ;    (* pop  %dx *)
    IsrTemplate[37] := 059H ;    (* pop  %cx *)
    IsrTemplate[38] := 058H ;    (* pop  %ax *)
    IsrTemplate[39] := 0CFH ;    (* iret *)
```

Example of FLIH in GNU LuK

- GNU LuK uses a routine `ClaimIsr` which will copy the `IsrTemplate` into the correct interrupt vector and then overwrite the vector number and function address in the template

Context switching

- the scheduler runs inside the kernel and it decides which process to run at any time
  - processes might be blocked waiting on a semaphore or waiting for a device to respond
  - a process might need to be preemptively interrupted by the scheduler if it were implementing a round robin algorithm

- the minimal primitives to manage context switching in a microkernel or operating system were devised by Wirth 1983 (Programming in Modula-2)
  - `NEWPROCESS`, `TRANSFER` and `IOTRANSFER` (covered later on)

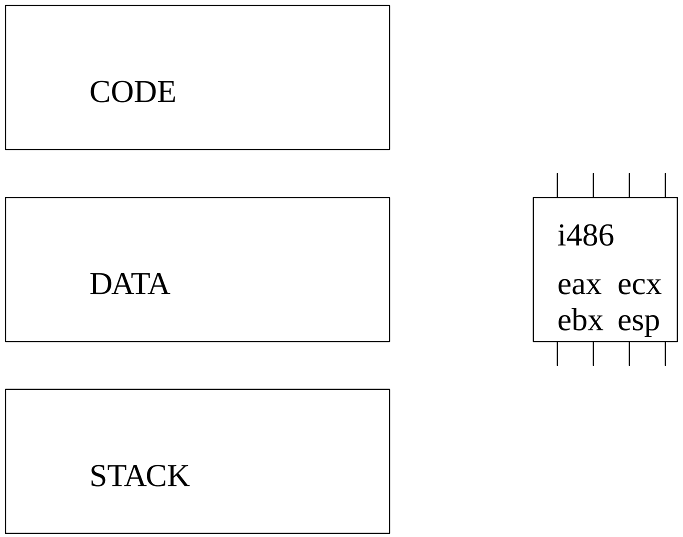
A tiny example of two simple processes in an operating system

```
void Process1 (void)
{
   while (TRUE) {
      WaitForACharacter();
      PutCharacterIntoBuffer();
   }
}

void Process2 (void)
{
   while (TRUE) {
      WaitForInterrupt();
      ServiceDevice();
   }
}
```

Primitives to manage context switching

- firstly let us look at a conventional program running in memory (single program running on a computer)



Primitives to manage context switching

- four main components
  - code
  - data
  - stack
  - processor registers (volatiles)

Concurrency

- suppose we want to run two programs concurrently?
  - we could have two programs in memory. (Two stacks, code, data and two copies of a volatile environment)
  - on a single processor computer we can achieve apparent concurrency by running a fraction of the first program and then run a fraction of the second.
  - if we repeat this then apparent concurrency will be achieved
  - in operating systems multiple concurrent programs are often called *processes*

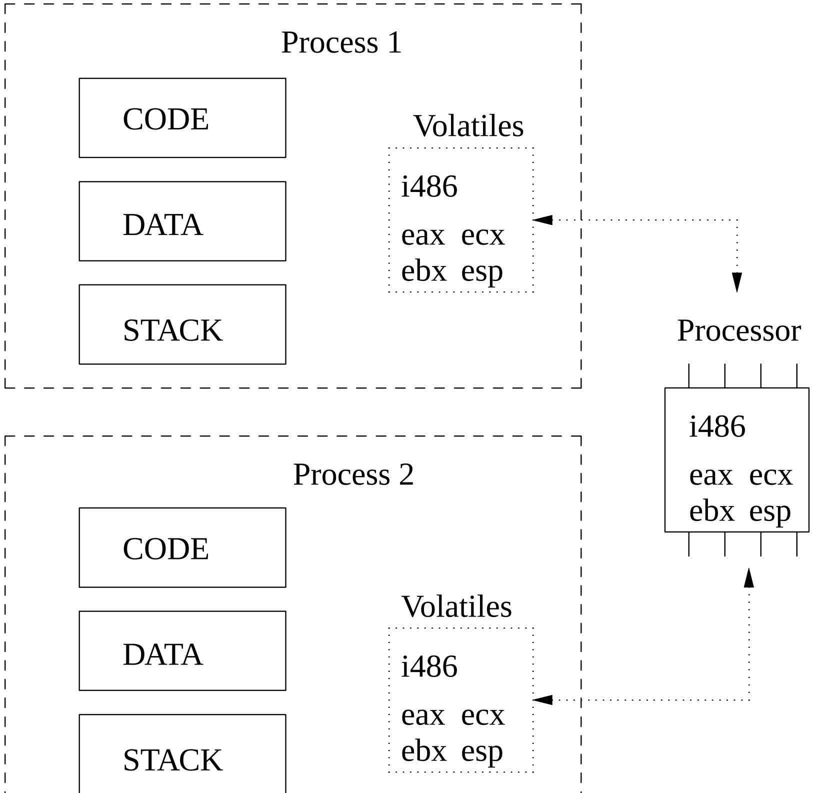
Concurrency

- what technical problems need to be solved so achieve apparent concurrency?
  - require a mechanism to switch from one process to another

- remember our computer has one processor but needs to run multiple processes
  - the information about a process is contained within the volatiles (or simply: processor registers)

Implementing concurrency

- we can switch from one process 1 to process 2 by:
  - copying the current volatiles from the processor into an area of memory dedicated to process 1
  - now copying some new volatiles from memory dedicated to process 2 into the processor registers



Implementing concurrency

- this operation is call a context switch (as the processors context is switched from process 1 to process 2)
  - by context switching we have a completely new set of register values inside the processor
  - so on the i486 we would change **all** the registers. Some of which include: `EAX, EBX, ECX, EDX, ESP` and `flags`
  - note that by changing the `ESP` register (stack pointer) we have effectively changed stack

Context switching primitives in GNU LuK

- the previous description of context switching is very low level
- in a high level language it is desirable to avoid the assembler language details as far as possible
  - `NEWPROCESS`
  - `TRANSFER`
  - `IOTRANSFER`
- **it is possible to build a microkernel which implements context switching and interrupt driven devices using these primitives without having to descend into assembly language**
  - these are the primitives as defined by Wirth in 1983

Context switching primitives in GNU LuK

- the primitives `NEWPROCESS`, `TRANSFER` and `IOTRANSFER` are concerned with copying ***Volatiles between process and processor***

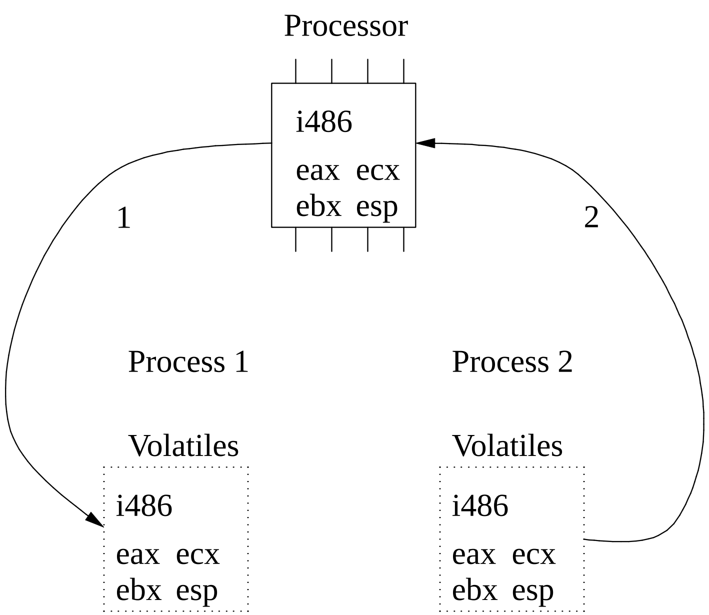
- the procedure `TRANSFER` transfers control from one process to another process

- these primitives are *low level* primitives
  - they are normally wrapped up by higher level functions:
    - for example: `initProcess` uses `NEWPROCESS` which is similar to `new_thread` in Python

TRANSFER

- the C definition is:
- ```
  typedef void *PROCESS;

  extern void SYSTEM_TRANSFER (PROCESS *p1, PROCESS p2);
  ```
- and it performs the following action:


  ```

IOTRANSFER

- ```
  extern void SYSTEM_IOTRANSFER (PROCESS *first,
                                 PROCESS *second,
                                 unsigned int interruptNo);
  ```

- the procedure `IOTRANSFER` allows process contexts to be changed when an interrupt occurs

- its function can be explained in two stages
  - firstly it transfers control from one process to another process (in exactly the same way as `TRANSFER`)
  - secondly when an interrupt occurs the processor is context switched back to the original process

- the implementation of `IOTRANSFER` involves interaction with the `FLIH`
  ```

NEWPROCESS

```
extern void SYSTEM_NEWPROCESS (void (*p)(void), void *a,
                               unsigned long n,
                               PROCESS *new);
```

- `p` is a pointer to a function.
  - this function will be turned into a process
  - `a` the start address of the new processes stack
  - `n` the size in bytes of the stack
  - `new` a variable of type `PROCESS` which will contain the volatiles of the new process

How is TRANSFER implemented?

- or how do we implement a context switch?
  - first we push all registers onto the stack
  - second we need to save the current running processes stack pointer into the running process control block
  - third we need to restore the next process stack pointer into the microprocessors stack pointer
  - fourth we pop all registers from the stack

How is TRANSFER implemented?

- ```
  void SYSTEM_TRANSFER (PROCESS *p1, PROCESS p2)
  {
     onOrOff toOldState;

     toOldState = turnInterrupts(Off);
     asm volatile ("pusha ; pushf");   /* push all registers */
     /* remember p1 is the address of a PROCESS */
     asm volatile ("movl %[p1], %%eax ; movl %%esp, (%%eax)"
        :: [p1] "rm" (p1));            /* p1 := top of stack  */
     asm volatile ("movl %[p2], %%eax ; movl %%eax, %%esp"
        :: [p2] "rm" (p2));            /* top of stack := p2 */
     asm volatile ("popf ; popa");     /* restore all registers */
     toOldState := turnInterrupts(toOldState);
  }
  ```

- `asm volatile`
  - means inline an assembly instruction
  ```

How is TRANSFER implemented?

- the parameters `("movl %[p1], %%eax ; movl %%esp, (%%eax)" :: [p1] "rm" (p1));`
- means
  - move `p1` into register `%eax`
  - move `%esp` into the address pointed to by `%eax`
  - p1 is a variable which may be in a register or in memory
  - p1 is an input to the assembly instruction

Conclusion

- we have seen the structure of a `FLIH`
- we have seen how three primitives can be used to create processes, context switch between processes and react to interrupts
- we have seen how a context switch might be implemented