

Language for a microkernel

- many potential languages could be used to build a microkernel
 - C, Modula-2, Ada, Pascal, Java
- we will build components of a microkernel in C
- C has been around for 30 years although ANSI C was created in 1983
 - C is used for many industrial projects, the favoured language for operating systems and microkernels

Language for a microkernel

- language choice is *very* subjective
 - the pros/cons presented here are also subjective and maybe wrong.. from your point of view
 - a useful exercise is to write out your own cons/pros together with your justifications

The C Programming language

- Pros:
 - not a large language
 - no preconceived process synchronization primitives
 - no process creation primitives
 - low level operations (bit manipulation possible)
 - reasonably strong typing available - but it can be forced if desired
 - get a error if you use an `int` rather than a `char`
 - no OO and memory management is via `kmalloc/kfree`

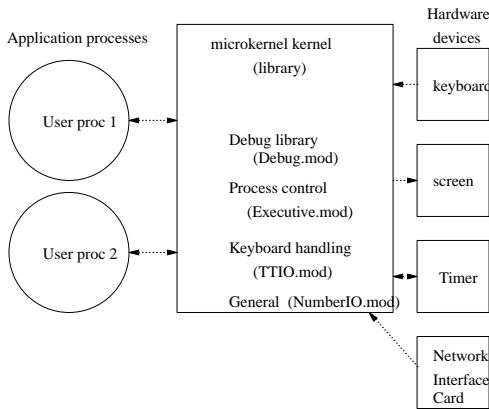
The C Programming language

- Cons:
 - no preconceived process synchronization primitives
 - the programmer needs to be disciplined when using memory management (`kmalloc/kfree`)
 - pointer complexity!
 - actually two languages the C preprocessor and the C compiler

microkernel software

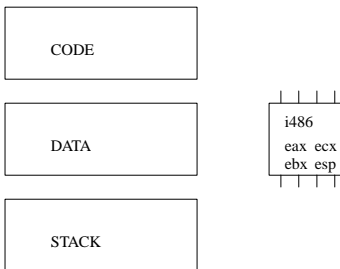
- what are the software components of a microkernel?

- typical microkernel might consist of:



High level facilities for low level control

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



A tiny example of two simple processes in the microkernel

```

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

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

High level facilities for low level control

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

Concurrency

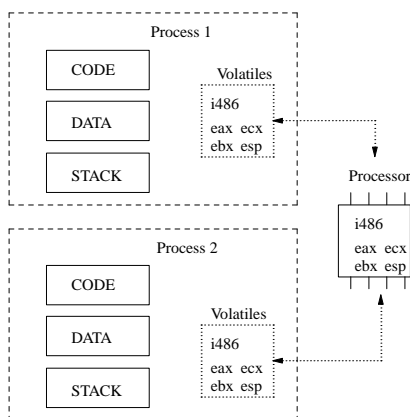
- suppose we want to run two programs concurrently?
 - we could have two programs in memory. (Two stacks, code, data and two copies of a volatile environment)
 - on a single processor computer we can achieve apparent concurrency by running a fraction of the first program and then run a fraction of the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

Thread primitives in the microkernel

- 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 (including context switching) using these primitives without having to descend into assembly language**
- see the module [SYSTEM](#) `<src.html#SYSTEM-h>` in our microkernel

Thread primitives in the microkernel

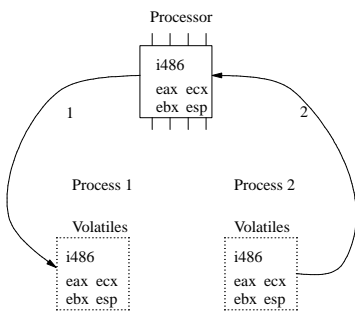
- the primitives NEWPROCESS, TRANSFER and IOTRANSFER are concerned with copying *Volatiles between process and processor*
- the procedure TRANSFER transfers control from one process to another process
- these primitives are *low level* primitives
 - later on we will build higher level process control functions

TRANSFER

- the C definition is:

```
extern void
SYSTEM_TRANSFER(void **p1, void *p2);
```

- and it performs the following action:



TRANSFER

- (examine the [SYSTEM](#) `<src.html#SYSTEM-h>` module in our microkernel)

IOTRANSFER

- 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

Creating processes in the microkernel

- concurrent applications inside the microkernel will *always* use the module `Executive` as it presents a high level interface to thread/process manipulation
- but `Executive` has to create processes using the primitives seen on the previous slides

Creating a process using SYSTEM

- a mechanism is needed to initially create a process
- the Modula-2 library defined the procedure `NEWPROCESS` as

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

Creating processes in the microkernel

- examine the function `InitProcess` in the module `Executive` ([src.html#Executive-c](#))

Creating processes in the microkernel

The Module Executive

```
Descriptor *
Executive_InitProcess (void (*p)(void),
                      unsigned int StackSize,
                      char *Name,
                      const int Name_High)
{
    Descriptor *d;
    OnOrOff ToOldState;

    ToOldState = SYSTEM_TurnInterrupts(Off);
    SysStorage_ALLOCATE((void **)&d,
                       sizeof(Descriptor));
    d->Size = StackSize;

    SysStorage_ALLOCATE((void **)&d->Start,
                       StackSize);
    SYSTEM_NEWPROCESS(p, d->Start,
                     StackSize, &d->Volatiles);
    ...
}
```

- the primitives: TRANSFER, IOTRANSFER and NEWPROCESS are present in the SYSTEM module
 - they are built from a high level language and assembly language
- the module Executive sits on top of SYSTEM and presents a higher level of abstraction
- examine the Executive source code in [C](#) (`<src.html#Executive-c>`)