1 Kernel functions

- Forms the core of the OS, and must contain minimal functionality.
- Should be implementable in a high level language such as C.

The **Kernel performs**

- Management of processes
  Process Data structures and context
  the creation, deletion, starting and stopping of processes.
- First level interrupt handling
  Support for both interrupt and device handlers.
- Multiplex (share) processor between processes
  Act as a *Dispatcher* and as a *Low level scheduler*
- Provide a mechanism of interaction between processes
  Achieved via a number of different primitives
  semaphores, locks and/or message passing.
- Timer functions
  delay for a period, time and date and accounting (measuring run time performance).
- Interrupt handling

Note: In some systems, I/O is part of kernel (Unix).

2 How is the kernel entered or invoked

Kernel is invoked as a result of :

1. **External interrupt** - such as a device (printer, disk drive, kettle) or a timer (clock).
2. **Internal interrupt** - error condition (such as divide by zero, memory protection violation, page fault, execution of an illegal or previledged instruction etc).
3. **Kernel call from a running process**.

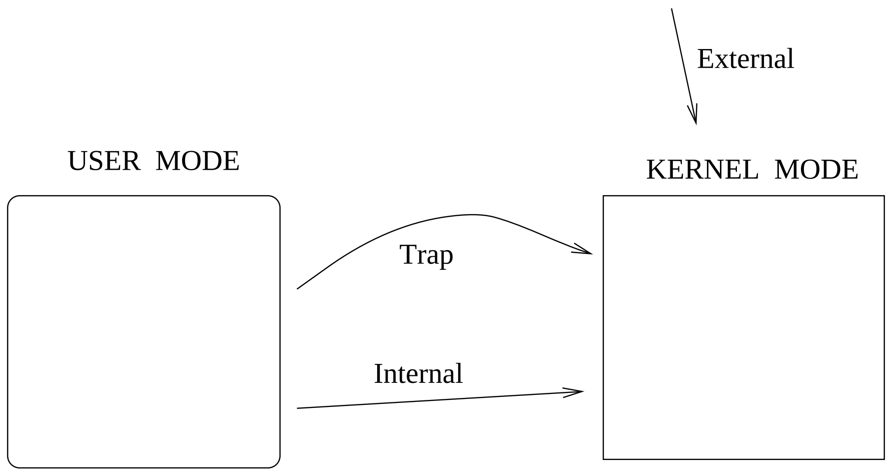

Figure 1:

Kernel entry switches processor into *previledged mode*

Access to :

- Kernel general purpose registers
- Program status word
- Memory management registers
- I/O instructions (on some processors)
- Enable and disable of interrupts
- Special instructions (such as *halt*)

Note: some system processes may need to execute in previledged mode (such as device handlers).

How is kernel entered from a user processes

1. Procedure Calls
2. Special Instructions (such as Supervisor Call (SVC) or trap)
3. Message passing (as in *Mach*)

For procedure call, **Kernel** resides in shared address space (or ROM) and

a) Kernel procedures are linked into process code
b) Single entry point into kernel, and parameters indicate kernel operation

The *special instructions* are similar to hardware interrupts, and register parameters determines operation.

In a message passing case, each process has default ports for sending and receiving messages to or from the Kernel.

Where are the kernel parameters stored ?

On the Kernel stack, which is used for variable and parameter passing within the Kernel.

1. Kernel has a stack like any other process - kernel registers
2. Two stacks are per process - a user mode and a kernel mode stack, however, this required extra memory (Unix)
3. Kernel runs on a user stack (as done in monitors), but in this case it is possible that the data may become corrupted

3 Monolithic - 'the BIG MESS' - systems

- No structure. No abstraction into individual operations.
- Each procedure with a well defined interface (in terms of parameters and results)
- The services provided by the OS (system call) are requested by putting requests in a special place, or using special methods of invocation (kernel calls).

4 Layering - 'PUTTING A STRUCTURE'-

- A number of layers within the operating system, each performing a specific function (such as process handling, memory management, I/O etc).
- Also useful as a security model (come to later)
- Example include MULTICS (concentric rings - parameter checking for validity before proceeding with the call).

5 Virtual Machine

- A virtual machine monitor provides multiple views to the users
- Contain copies of bare hardware functions such as registers, I/O interrupts, kernel/user mode etc
- Switch between machine models to obtain multiprogramming
- Each virtual machine could potentially run a different OS

6 MicroKernel Approach

- aka Client-Server organisation.
- The organising structure currently in vogue is the **microkernel** OS.
- The goal is to minimise what goes in the kernel, and implement much of the operating system as user-level processes. This results in :
  Better reliability
  Ease of extension and customisation
  Unfortunately - mediocre performance
- Examples of such systems include : Chorus (a French Unix-like system), (in many ways) MS-WinNT, CMU Mach

7 Concurrency and Processes

The OS must coordinate all the activity on a machine - such as the presence of multiple users, I/O interrupts etc

How can it keep all these things straight ?

Answer : It decomposes hard problems into simpler ones. Instead of dealing with everything going on at once, it separates the activities so it can deal with one at a time.

8 How are the problems decomposed into smaller ones ? Is there a basic unit of computation

YES!!!! - its called a **PROCESS**.

- All runnable software on a computer, including the user and kernel processes, is organised into a number of **sequential processes** or just **processes** for short.
- It is an *abstraction* to represent what is needed to run a single program.

The difference between a program and process is subtle, but crucial.

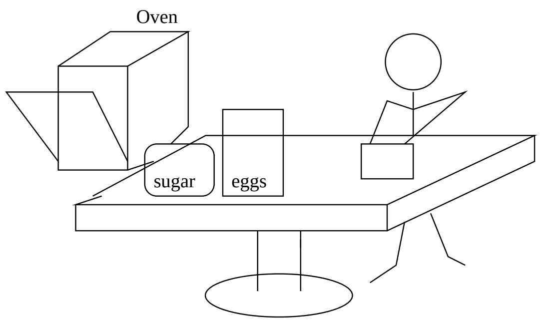

Figure 2:

9 An analogy ...

You are learning to make blueberry muffins :

- Find a recipe and acquire the necessary ingredients (flour, eggs, sugar, butter, blueberries etc)
- Here, recipe is the program
- You are the CPU (the processor of the recipe or information or program)
- The ingredients constitute the inputs, and the muffins the output
- Process is the activity of actually reading the recipe, fetching the ingredients and baking the cake.

Interruptions from the outside world - the television, the neighbours etc - start from where you left off.

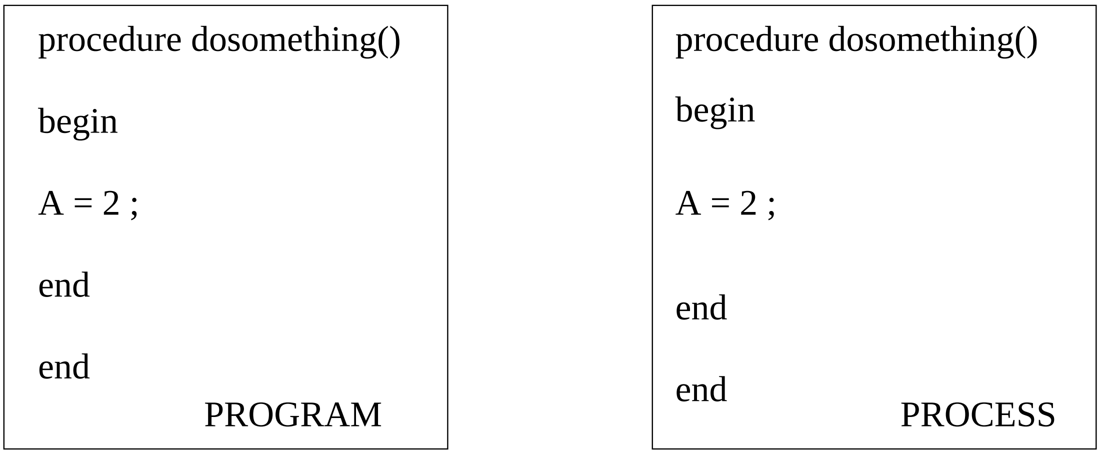

Figure 3:

- A process is more than just a program
- I run **ls** - you run **ls** - same program, but different processes (in Unix)
- Less to a process than a program - a single program can invoke more than one process to complete a task
- Process must be able to create other processes (dynamic process creation) and destroy those not needed.
- The start-up or initialisation processes present on all OS.

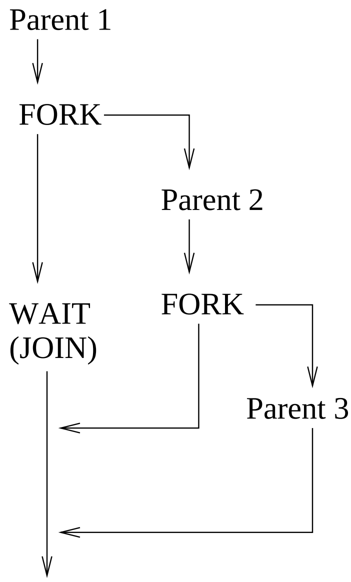

Figure 4:

- Process creating by **Fork** and **Join** in Unix
- To create different child processes, child must execute
  exec(filename, arguments)
  join = wait(pid)
- Concurrency and synchronisation is mixed
- When a **create process** call is made to the OS, the OS allocates :
  Process id (Pid) = create(filename,arguments)
  Allocate storage space
  Use other mechanisms to achieve synchronisation with processes already active
- WinNT supports both duplicating parent's address space and loading a new program into a process

10 Before we go further, some definitions

Uniprogramming : Running one process at one time (such as on MS/DOS or the Macintosh (old)).

Easier for OS builder. Gets rid of the problems associated with concurrency, by removing it! For a PC, the idea was - one user does only one thing at a time - so whats the point of having multiple operations active

Harder for user; need to take a coffee break every time something prints.

Mutliprogramming : More than one process active at a time (Unix and OS/2)

often called multitasking, but that could mean something else, as we shall soon see.

Multithreading : A single program made up of a number of different concurrent activities (also called *multitasking* - as in Ada).

Anh hunh ... so ... aaaa ... what's a **thread ?**

"the best way to avoid a temptation is to yield to it!

11 Threads

- A **Thread** is a sequential execution stream within a process, sometimes also called *a 'lightweight' process*
- There are therefore, multiple threads within a process
  - each has a stack, PC and registers (own copy)
  - share data and code
  - sharing of *state*, like contents of memory (global variables, heap), file system
- The **address space** constitutes all the addresses that are needed by a program - providing the illusion that the program is running on its own machine
- Possible to have multiple threads per address space (such as in Solaris)
- Although if only a single thread in an address space, there is a large similarity between threads and processes
- Keep the distinction between *threads* and *address space* : The **address space** encapsulates **protection** (preventing a buggy program from messing everything else on the system), whereas a **thread** encapsulates concurrency.

- Threads within a process belong to a single user
- Threads, like processes, can either belong to the user or the kernel
  (user level threads are in invisible to kernel, and provide faster switching)
- Very efficient switching
- Usually no preemption within threads
- Also, sometimes protection between threads is revoked

Examples of multithreaded applications include :

- *Robot control :* single program, multiple concurrent operation.
- *Network server :* single program, must handle concurrent requests from multiple users (web servers).
- *Windowing systems :* one thread per window.
- *Parallel Programming :* split program into multiple threads to make it run faster. Often called **multiprogramming**

Hence :

Multiprogramming : Multiple jobs or processes
Multiprocessing : Mutiple processors or CPUs.

To **confuse matters** even further, some multiprocessors are in fact uniprogrammed – i.e. use of multiple threads in one address space, but only run one program at a time.

Examples of multithreaded operating systems :

- **MS/DOS** : one thread, one address space
- **traditional Unix** : one thread per address space, many address spaces
- **WinNT, Unix (Solaris, HPUX), CMU Mach** - many threads per address space, many address spaces
- **Pilot** (the OS on the first personal computer ever built)- many threads, one address space (no protection! - single user no *need* for protection).

Hence :
Depending on the context, **Processes** *could be viewed as the unit of resource allocation and* **Threads** *the unit of execution and the active entities which need to communicate.*