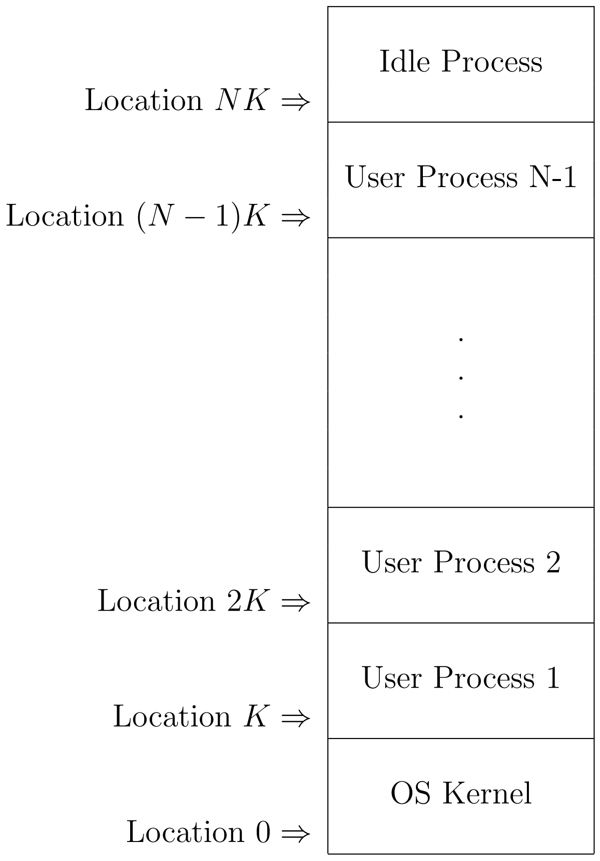
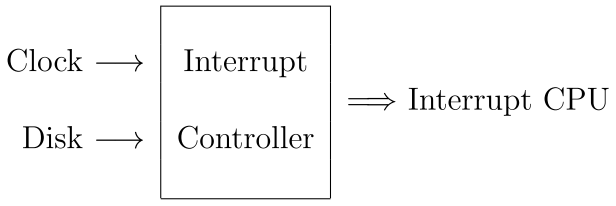
Notes on Processes

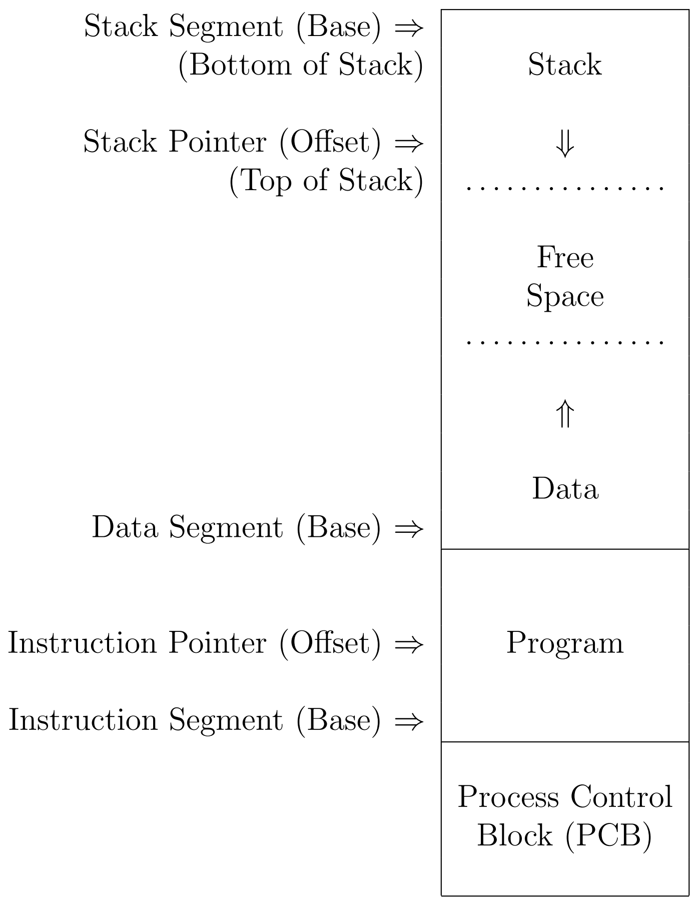
Date: 2 February, 1993

Overview:





Process Anatomy:



Process Control Block:

Each process has a *Process Control Block,* PCB, which maintains important information about the process including the state it was in when interrupted by the operating system.

1. Internal CPU Registers:
    - Accumulator.
    - CPU Flags and Status registers.
    - Other general purpose registers, E.g. B, C, D, E.

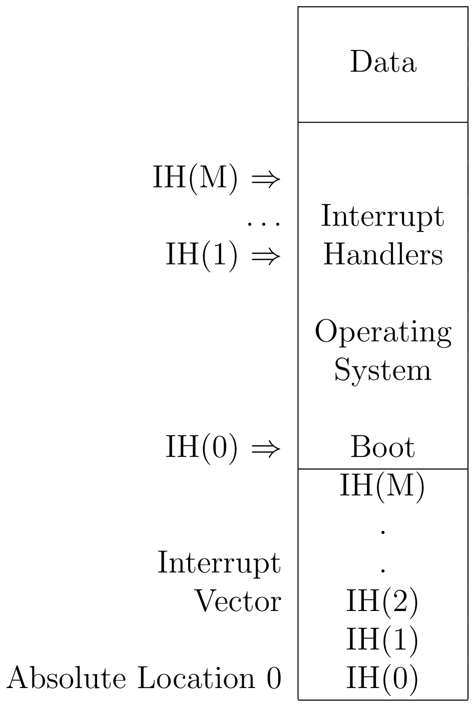
- Instruction Segment and Instruction Pointer registers.
- Data Segment register.
- Stack Segment and Stack Pointer registers.

2. Process Flags:

   - Empty PCB
   - Ready
   - Blocked
   - Executing

3. Semaphore Data Structures: a link to the next blocked process, if this process is blocked.

The Kernel Anatomy:



Semaphore Implementation:

1. Data Structures:
   (a) Process : In each PCB is a pointer *SemaphoreChain*. These are used to form a queue of processes waiting on semaphores.
   (b) Kernel : For each semaphore, there is a record containing:
      i. the value of the semaphore, *Semaphore[i].value*,
      ii. head and tail pointers to a queue of processes waiting on the semaphore, *Semaphore[i].head* and *Semaphore[i].tail.*
2. Code:
   (a) P[i]: When a process does a P on semaphore i, the kernel is invoked. The value i is checked for validity and, if OK, interrupts are disabled. What happens next depends on the value of *Semaphore[i].value.*
      - If it is not zero: it is decremented, interrupts are re-enabled and control returns to the caller.
      - If it is zero: the calling process is added to the end of a queue of processes waiting on the semaphore. For a non empty queue, *Semaphore[i].head* is not nil, this is done as follows:

        ```
        Semaphore[i].tail^.SemaphoreChain = caller's PCB
        Semaphore[i].tail = caller's PCB
        Semaphore[i].tail^.SemaphoreChain = nil
        ```

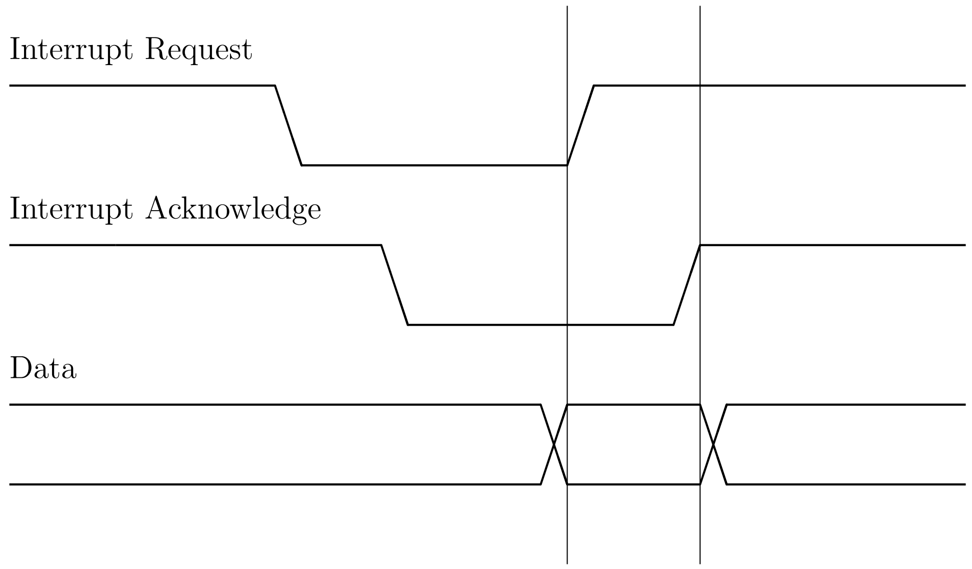
        How this is done if *Semaphore[i].head* is nil is left to the reader. The process is placed in the "blocked" state and interrupts are re-enabled.
   (b) V[i]: When a process does a V on semaphore i, the kernel is invoked. The value i is checked for validity and, if OK, interrupts are disabled. What happens next depends on whether there are blocked processes on Semaphore i.
      - If *Semaphore.value* is not zero or if *Semaphore[i].head* is nil, *Semaphore.value* is incremented, interrupts are re-enabled and control is returned to the caller.

- Else, there are blocked processes on the semaphore. The first process on the queue is removed and unblocked:

```
Semaphore[i].head^.BlockFlag is cleared
Semaphore[i].head =
        Semaphore[i].head^.SemaphoreChain
```

  Interrupts are re-enabled and control is returned to the caller.

Interrupt Handshaking



Interrupt Processing

There are three pairs of Interrupt Request (IR), Interrupt Acknowledge (IA) lines which use the handshaking diagramed above. They are between the Disk Device and the Interrupt Controller, between the Clock Device and the Interrupt Controller and between the Interrupt Controller and the CPU. Between the Controller and the CPU there are also Data lines. These lines inform the CPU which of the two devices caused the interrupt. We give an example of the Disk Device requesting an interrupt.

1. The Disk asserts the IR between it and the Controller.
2. The Controller sees the IR asserted and in turn asserts the IR between it and the CPU. If the Controller is already in the middle of a handshake

with the CPU, perhaps the Clock asserted its IR just a moment earlier, the Disk's request is held pending.

3. If the CPU has interrupts enabled, it responds to the Controller by asserting the IA. If interrupts are not enabled, the handshake stalls at this point.
4. The controller sees that the CPU has asserted IA and chooses between the Disk and Clock using, for instance, Dekker's algorithm. Suppose Disk is chosen. A code for Disk is placed on the Data lines and IR is released.
5. The CPU sees the release of the IR, reads the Data and then releases the IA.
6. The CPU disables interrupts and uses the Data to index the Interrupt Vector.
7. At this point, the handshake between the CPU and Controller is completed. The Controller can assert the IR any time after the IA returns high.
8. However, the Disk has not been informed that its interrupt is being serviced. Any time after the Controller asserts the IA between itself and the CPU, it asserts the IA between itself and the Disk.
9. The Disk sees the IA asserted and releases its IR.
10. The Controller sees the Disk's IR released and releases the IA. This completes the handshake between the Disk and the Controller. The Disk can interrupt again any time after it sees the IA return high.

Scheduling and Disk Access

The processes are scheduled round robin. Then can be found in one of three states: Executing, Ready or Blocked. The PCB will be marked Empty when there is no process residing at that memory address.

Time-slicing is accomplished by attaching the scheduler to the Clock Device's interrupt handler. When the clock interrupts, the scheduler fills in the

PCB of the currently executing process and selects a new process for execution. It can do this by simply marching through all PCB's looking for one marked "Ready". It unloads the information from the selected PCB into the CPU, re-enables interrupts and launches the new process.

To communicate with the Disk, there are two semaphores:

1. DiskResource: to gain access to the disk. It is initialized to 1.

2. DiskWait: to wait for completion of the disk operation. It is initialized to 0.

A process wanting the disk must first acquire it by means of *P(DiskResource)*. Once access is gained, the process begins the disk transfer. The process will wait for an interrupt from the Disk Device before continuing. It performs a *P(DiskWait)* and is suspended. When the Disk Device interrupts, its interrupt handler copies important information to a buffer and performs a *V(DiskWait)*. When the requesting process continues, it finishes the transfer and then releases control of the disk via *V(DiskResource)*.

If a process begins the disk operation and is interrupted by the clock, it could happen the the disk operation completes before the process is resumed. In which case the *V(DiskWait)* by the interrupt handler will occur before the *P(DiskWait)* of the disk process. However, the interrupt will not be lost: the *P(DiskWait)* will decrement *DiskWait* from 1 to 0 and will not block.