1 Kernel functions

```
last pid:  2988;  load averages:  0.88,  0.54,  0.49    15:40:50
82 processes:  76 sleeping, 1 zombie, 3 stopped, 2 on cpu

Memory: 59M real, 940K free, 84M swap, 180M free swap


  PID USERNAME PRI NICE  SIZE   RES STATE   TIME   WCPU    CPU COMMAND
 2987 pp       -25    0 5184K 3152K sleep   0:01  1.72%  1.72% smtpsrvr
 2988 pp       -25    0 5188K 3032K cpu     0:00  0.89%  0.89% submit
 2986 scmofr   -17    0 1264K  976K cpu     0:00  0.87%  0.87% top
 2962 scmofr    23    0 1156K  904K sleep   0:01  0.78%  0.78% tcsh
 2963 pp       -25    0 6072K 3924K sleep   0:01  0.69%  0.69% local
 2964 pp        35    0 5208K 3516K sleep   0:01  0.67%  0.67% smtp
  529 root      28    0 1584K  428K sleep  38:33  0.45%  0.45% fingerd
  482 root       6    0 3232K 1620K sleep  33:17  0.38%  0.38% nscd
21938 oracle     5    0   29M 2884K sleep   3:01  0.25%  0.25% oracle
17264 root      33    0 6380K 1812K sleep   1:39  0.23%  0.23% qmgr
 2958 root      25    0 1368K  888K sleep   0:00  0.17%  0.17% in.rlogi
17259 pp        30    0 5144K  844K sleep   0:42  0.13%  0.13% pptsapd
 2965 scmmm1   -25    0 6284K 1848K sleep   0:00  0.10%  0.10% local
  184 root      33    0 2056K  644K sleep   5:29  0.08%  0.08% rpcbind
 2967 scmmm1    33    0 1316K  964K sleep   0:00  0.08%  0.08% rsh
  242 root      33    0 4672K 2204K sleep   6:57  0.06%  0.06% in.named
  816 root      33    0 1796K  728K sleep   4:34  0.06%  0.06% mountd
25221 htbin     33    0 2044K  856K sleep   0:04  0.06%  0.06% oraweb
```

2 Back to the Kernel

Kernel responsible for managing *process states*.

3 Process States

A process can be

- *New* (a new process is being created)
- *Running* (executing on the CPU at that instant)
- *Waiting* (unable to run, blocked or waiting for an external event)
- *Ready* (runnable; temporarily stopped to let another process run)
- *Terminated* (process is dead ; being deleted)

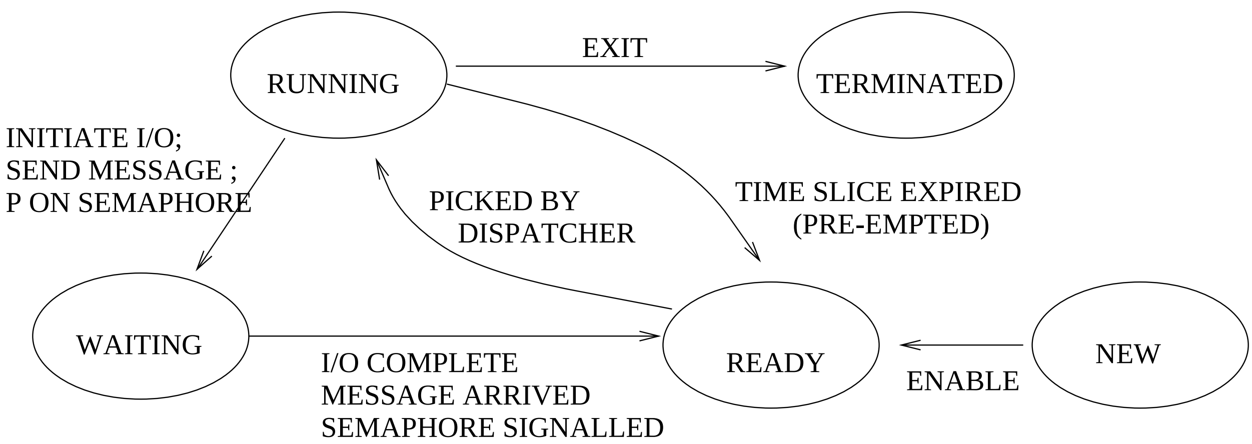

Figure 1:

Why no transitions from Waiting to Running ?

4 Process Management Operations

Process Creation

- Requires :
  Process name,
  Process code + start address
  data + stack
- Kernel creates a PCB (Process Control Block), assigns enough memory and makes the process runnable

Process Deletion ('Killing and removing a process').

- Stop process
- clean up - outstanding I/O, messages etc.
  recover resources (such as memory)
  delete the PCB
- *This is difficult to achieve - WHY ?*
- *What happens when a parent is killed before the child can respond with the result ?*

Process Control

- *Movement of processes between states* - such as halting a process (make unrunnable)
- *Schedule* a runnable process
- *Continue process* - make runnable

Also possible to change *priorities* of processes

Associated with interrupts are processes which perform specific functions, and after completing the activity pass over control to something else.

- *process context is saved in PCB when process stops running*
- *Context is then re-loaded into processor registers when the process starts running again*

Kernel maintains an array or linked list of PCBs

5 Representing a process - the PCB

A process is represented by the PCB (Process Control Block) or PD (Process Descriptor), which contains the following :

- **Process Identifier** (parent identifier)
- **Context**
  Program Counter (PC)
  Processor Status Word (PSW)
  Stack Pointer (SP)
  Registers (perhaps implemented as a stack)
  Memory addressing registers
- **Process State**
- **Priority**
- **Accounting Information** : start time, CPU used
- **Resources Allocated**
  memory allocated (base and limit regs, page and segment tables)
  root directory
  open files
  other i/o devices
- Exception address
- *Pointer for queues* - why is this needed ?

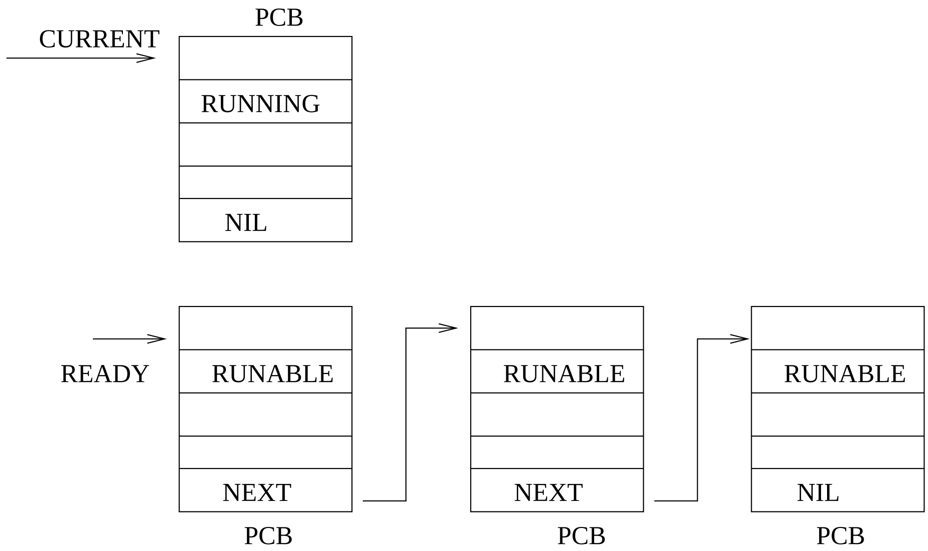

Figure 2:

Assume - ReadyQ is priority ordered

```
BEGIN
  disable interrupts
  initialise data structures
  create initial processes

  REPEAT (FOREVER)
    Dispatch process



  ENTER :
   save registers of current process
   determine the source of interrupt
   service interrupt or call
 END (repeat)
END
```

Things to consider :

- The **Dispatcher** selects the highest priority process, enables the interrupts and runs it. The priority at which a process is run is defined by the PSW (Processor Status Word)
- The saving of registers and determining interrupt source is determined by the *First Level Interrupt Handler* (FLIH)
- **Note** : Servicing an interrupt or call may state the change of the current or another process, and cause a reschedule to take place. Hence, it is possible that another process may be run after an external event has occurred instead of the one intended prior to the event. Hence, the dynamic element of scheduling
- The amount of stuff that is stored in the PCB depends on the need. Hence, it is possible to control the degree of information that needs to be recorded on a *context switch*

The Dispatcher does the following (more later) :

```
    REPEAT (FOREVER)

Run thread (process)
  Save state (into PCB
   or TCB (thread control block))
Choose new thread (process) to run
Load state into PCB (TCB)

    END
```

6 Interrupts and Interrupt Vectors

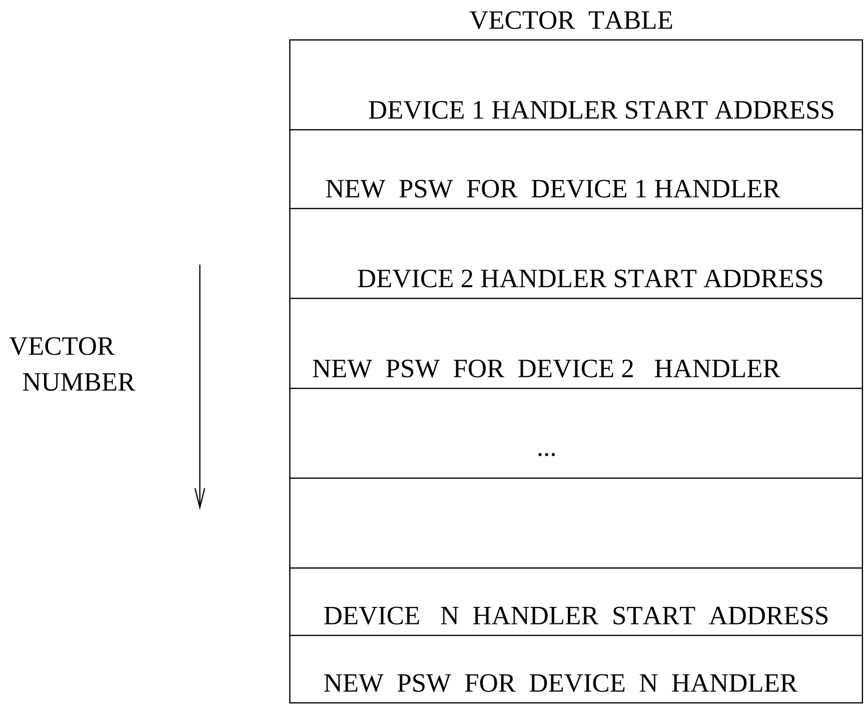

Figure 3:

- When device generates interrupt, it also generates the vector number to identify itself.
- *Trap instruction* = software interrupt
- Kernel sets up vector table of interrupt handlers
- Current PC and PSW = onto stack
- New PC and PSW loaded from **vector table**

if all interrupt handling in **assembly language**, in the Kernel, then interrupt handlers :

- Save and restore interrupted programs context
- Perform device independent processing

Hence, no software FLIH

Note : Interrupt handlers should however be written in a high level language, and be part of device handler process - particularly valid for real time systems.
Also, LIH in kernel to perform actions common to all interrupt handling.

Note: There is a slight distinction between forking a thread as opposed to forking a process (unix 'fork'). A Unix fork creates a new process, so it has to create a new address space, in addition to a new thread.

Forking a thread is very much like an asynchronous procedure call, it means - go do this work - where the calling thread does not wait for the callee to complete.

What if the calling thread needs to wait ?

Thread *Join* – wait for a forked thread to finish.

Hence, a traditional procedure call (in Pascal for instance) :

```
 procedure A()
   B()
 procedure B()

equivalent to

  procedure A()
  Thread t = new Thread ;
  t^Fork(B) ; t^Join()
```

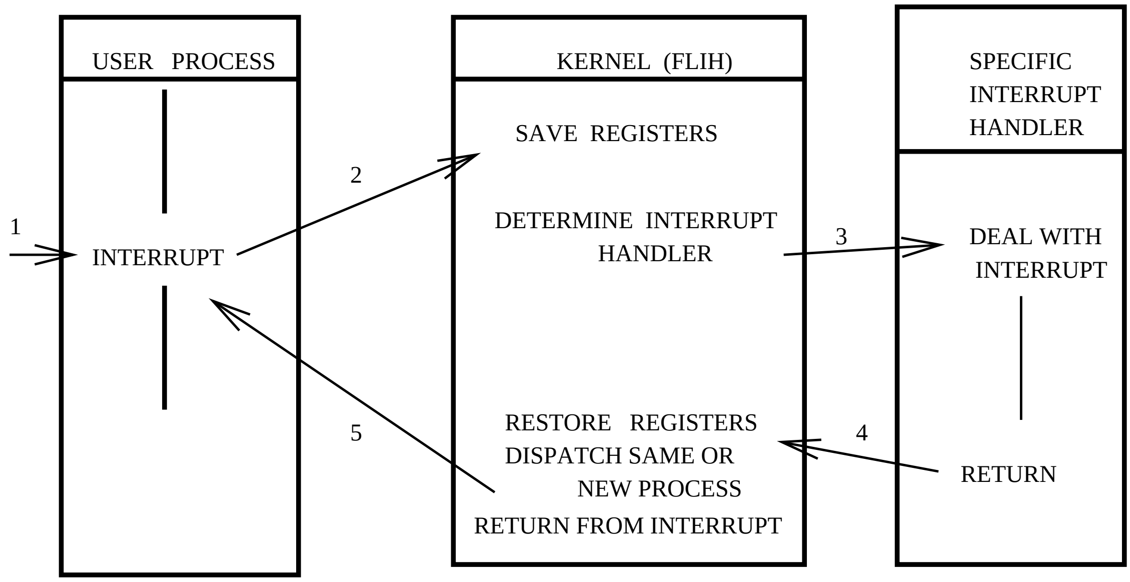

Figure 4:

7 First Level Interrupt Handler (FLIH)

- save registers of current process in PCB
- Determine the source of interrupt
- Initiate service of interrupt - calls interrupt handler
  Hardware dependent - implemented in assembler

Note : FLIH must run with interrupts disabled - Why ?
Enable or disable interrupts by priority level in PSW.

- Interrupt handler may call kernel to unblock another process (e.g. waiting for I/O) which could result in a different process - rather than interrupted one eventually being run by the dispatcher
- Handler could be a kernel procedure to service a normal kernel call, e.g. send a message, delay a process etc.

Nested Interrupts

- This occurs when the interrupt handler runs with interrupts enabled - in what scenarios is this likely to occur ?
- If interrupt handler runs in stack of interrupted procedure, ALL context must be saved on stack. The ALL in this case refers to the level of nesting.
- The stack space of ALL processes must be increased to cope with the additional level of nesting.

8 Dispatcher

- Allocates **processes** or **threads** to **processors**
- Selects the *highest priority ready* process from the head of *Ready* queue, and moves it to the *running* state - i.e. executing on the processor
- Invoked after every entry into the kernel

How is control returned to the dispatcher ?

The current process continues unless one of the following occurs :

- **Kernel call** moves it into *waiting* state, e.g. *P(sem)*
- An **Error Trap** occurs - such as a *memory protection violation*
- Its **Time slice** expires - give up CPU to someone else
- It is **interrupted** by a **higher priority** process

Hence, two types of events : **Internal events** (Sleeping Beauty - go to sleep and hope in the future a Prince Charming will wake you), or an **external event** - (unpredictable) occurs.

Dispatcher keeps a list of *ready* processes (threads) - how does it choose ?

a) Zero ready threads - just loops
b) One ready thread - easy
c) More that one thread - use a strategy (FIFO, LIFO, Priority)

A sample dispatcher :

```
PROCEDURE  DISPATCH

BEGIN

If current^.state <> running OR
    current^.priority < readyq^.priority THEN

BEGIN /* choose new process */
  IF current^.state = running THEN
    BEGIN
current^.state=ready
schedule(current)
 /* add current to ready queue
    in priority order */
    END
  current := readyq
  readyq := readyq^.next
        END
load registers from PCB or stack
put PC and PSW on stack

return from interrupt
 /* loads PC and PSW from stack */

END
```

what should dispatcher do if there is no runnable process ?