1 Device Management and Input/Output

Objectives :

- Provide a *uniform* and *simple* view of I/O (hide complexity of device handling)
- Provide *device independence* from : *device type* - terminal, disk or tape, *device instance* : which terminal, which tape etc
- *Fair access* to shared devices
- *Smooth allocation* of dedicated devices
- **Important** : *Ability to exploit parallelism of I/O devices for multiprogramming*

Devices can however vary considerably :

- Differences in *speed*, *synchronous* or *asynchronous*
- *Unit* of data transfer - *character* or *block*
- Character codes - are they all the same ? Does everyone use the same representation
- Operations supported : *read, write, seek, print etc*
- Error Conditions
- Sharable or is it only for a single user (e.g. printer, tape, kettle ?)

2 An overview of common devices

Terminals

- Connect to computers via serial lines (RS-232) : 9600 baud (bits/sec) $\equiv$ 900 bytes/sec
- Although small rate of transfer - can swamp CPU die to overhead of handling each byte
- Say 10 users (running vi or emacs) = 900 interrupts per user : overhead of handling interrupt 100 microseconds
- *Direct Memory Access* : Instead of interrupt on every block - do a block transfer - then interrupt (overall - faster transfer rates)
- *Redirection* : Bitmap displays : CPU writes to video memory - display then reads video memory

Disks

- Options : choice of sector size (tradeoff between space and time efficiency), usually big blocks mean greater transfer rate (time efficiency) as time to find data on disk is minimised

Trends in Disk Technology
for a typical disk today :

- Disk capacity : 2.1 GB
- Number of surfaces per pack : 18
- Number of tracks per surface : 2048
- Number of sectors per track : 32
- Number of bytes per sector : 1 KB
- Number of revolutions per minute : 7200

Transfer rate = 1 KB bytes/sector * 32 sectors/cycle * 120 cycles/second = **4 MB/sec**

3 I/O Layers

- **User Interface** : OS I/O procedures
  Access to virtual devices (such as *files* on Unix, *streams* on DEC10 and *datasets* on IBM - device independent
  Operations supported : *read, write, open, close, seek etc*
  Synchronous or asynchronous

- **Device Independent Layer** : Allocation of dedicated devices
  Protection : user access validation
  Naming and switching : map a virtual device to a real device
  Validate request against operations perform able on device
  Buffering and Spooling
  Error Reporting

- **Device Handler** : Access device registers
  Initiate operations
  Schedule requests
  Handle errors
  Perform character translation if needed

- **Interrupt Handler** : Process each interrupt,
  On the completion of a transfer, signal device handler

4 Where does the I/O part fit in the overall structure

4.1 As independent processes

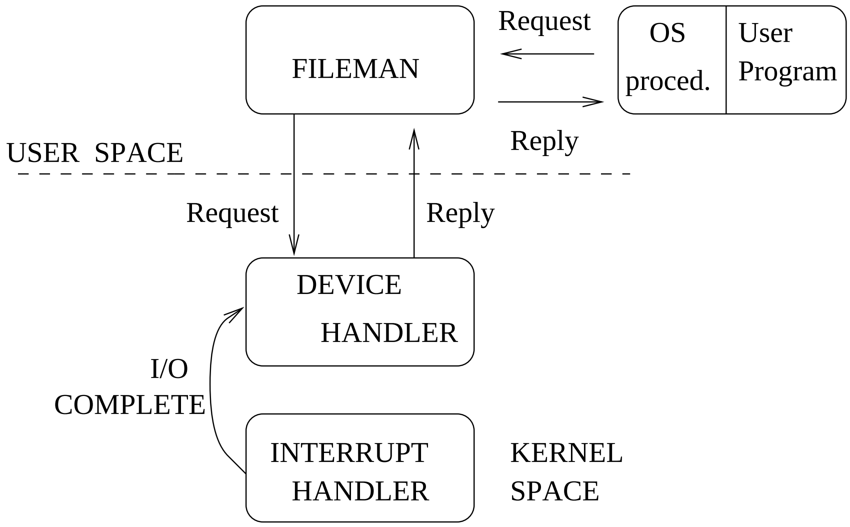

Figure 1: I/O independent processes

Interprocess Communication achieved in this case by

- Request queues and synchronisation via semaphores
- Via message passing between processes

4.2 Monolithic processes

- Kernel and user are different parts of the same process
- Interface between the various I/O components is not well segregated (clean)
- This is in many ways more efficient, because procedure calls do not involve a context switch for communication between the I/O components

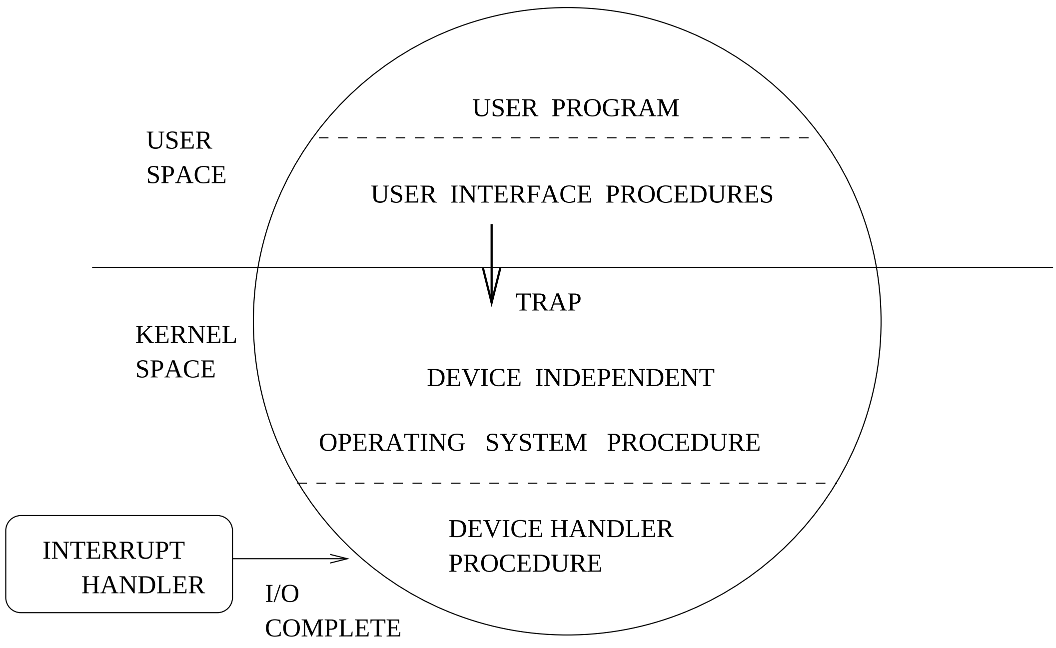

Figure 2: Monolithic structure

5 The Unix I/O Interface to the User

- Devices are considered to be part of the file system - mapped as special files : /dev/tty1 or /dev/lpr (the special case : /dev/null)
- *Device independence* :

```
cat file.txt > /dev/tty
cat file.txt > /dev/lpr
cat file.au > /dev/audio
```

- *I/O Classes* :
  Characters (unstructured) : files and devices
  Block (structured) : files
  Socket (message) : network interface
- **I/O Calls** are **synchronous**

```
fd = open(filename, mode)
close(fd)

nbytesread = read(fd,buffer,nbytes)
nbyteswritten = write(fd, buffer, nbytes)
pos = lseek(fd,offset,offsettype)
```

- *where* : **fd** : index to a file descriptor and **mode** : 0,1 or 2 (for read, write or read/write)
- **offsettype** : relative or absolute (bytes or 512 byte blocks)
- **How are requests made to perform activities on devices ? How are these requests managed ?**
- *Answer:* Need a **File Manager** process which performs these activities

Requests are made to the file manager via a **request queue**

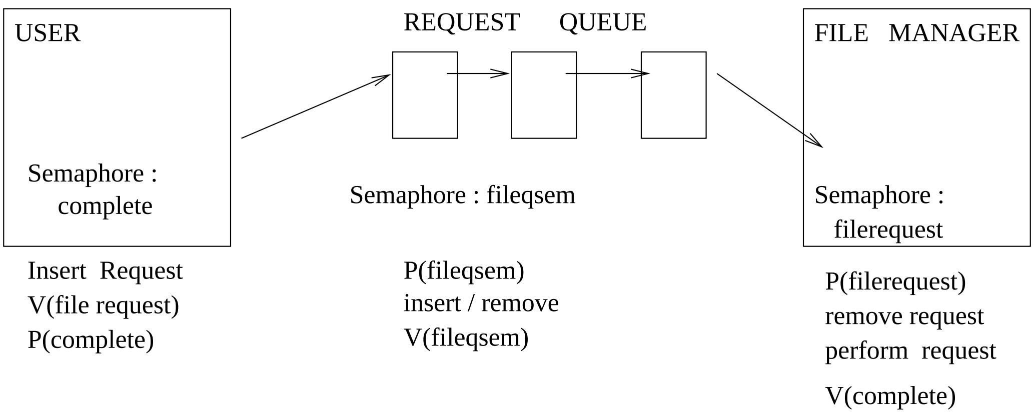

Figure 3: Filemanager

Each descriptor within the request queue is as follows :

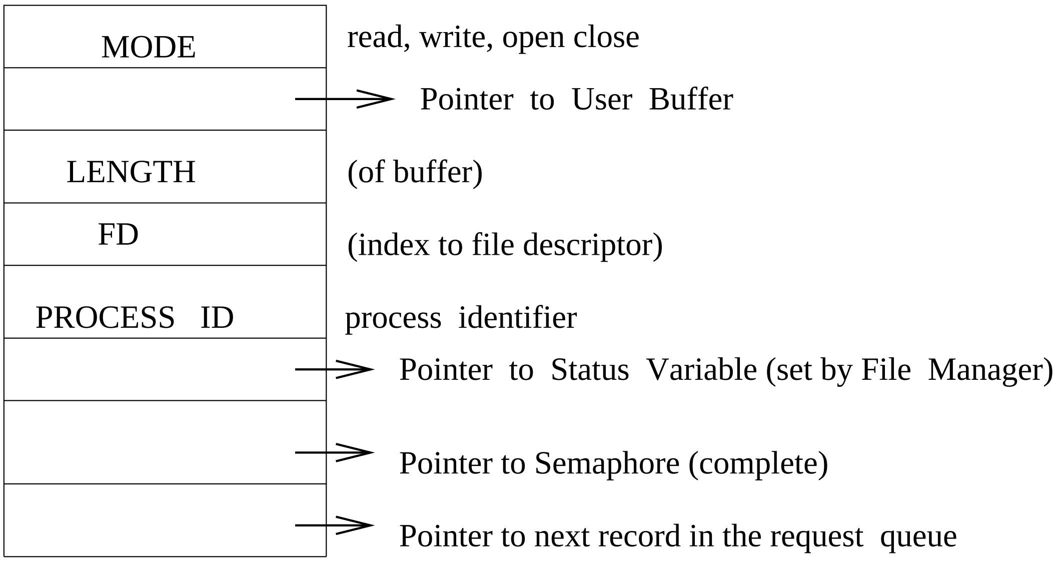

Figure 4: FileMan request queue

What additional information is needed for an open request ?

6 Implementation of Interface Procedures

Can be implemented via *semaphores* or *messages*

semaphore implementation

```
PROCEDURE Read(parameters)
 new(io_request)
 put mode, buffptr, length, fd, process_id,
     pointers into io_request fields
 P(fileqsem) {get mutual exclusion on request queue}
 insert(fileq, io_request)
 V(fileqsem)
 V(filerequest) {tell file manager work available}
 P(complete) {signalled by file manager}
 return(status) {status = -1 for error}
  {if ok, return value specified in call}
END
```

message based request transfer

```
PROCEDURE Read(parameters)
 new(msgbuff)
 put mode, buffptr, length, fd into msgbuff
 SEND(fileman, msgbuff)
 RECEIVE(fileman, status)
 return(status)
END
```

7 Synchronous vs. Asynchronous I/O

Synchronous I/O

- Interface procedure returns when I/O operation is completed ... i.e. no other activity in procedure
- I/O 'appears' instantaneous to users

Asynchronous I/O

- User process executes in parallel with I/O operation
- No *Receive* (message passing) or *P(complete)* (semaphore) in the interface procedure - must add another procedure
- *status = waitio(fd)*
- *Less Secure* : user can access data buffer before operation completes

```
write(fd, buffer, length)
 {continue processing}
status = waitio(fd)
 {blocked waiting for I/O to complete}
 {continue processing}
```

How is *waitio* implemented ? Using semaphores :

```
PROCEDURE waitio(fd)
 P(complete(fd))
 return status
END
```

File Manager Process

```
PROCESS  fileman
  initialise data structures
 REPEAT
  receiveany(source, msg)
  case msg.mode

  open : get device descriptor
      check access permissions and mode
 if device free - allocate file descriptor
 {what if device already open ?}
 insert info. into descriptor
   send(source,fd)

   close : ...

   read : use fd to find the file descriptor
  save source in descriptor
  check if request is valid
  set up devicemsg
  send(device, devicemsg)

   write : ...

   I/O complete: If error - notify
 find descriptor for device
 send(user_process, status)
 FOREVER
END
```

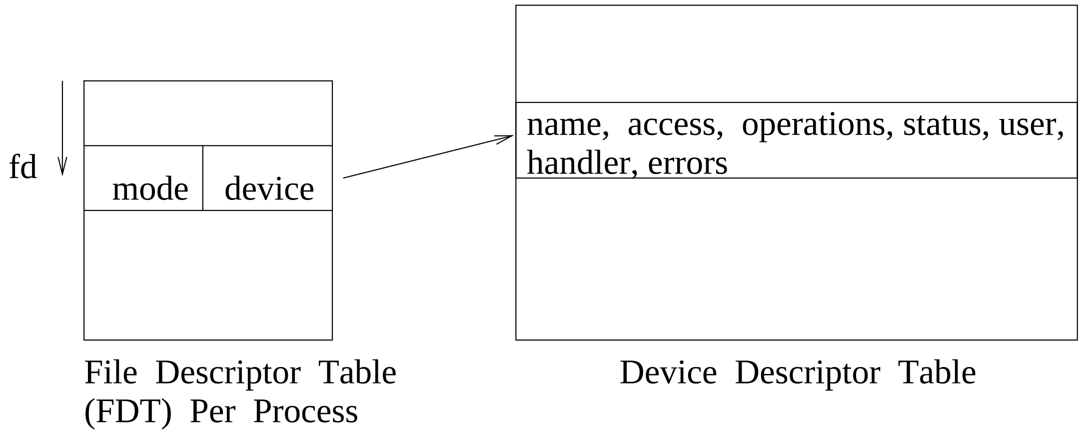

Figure 5: File manager data structures

```
open(device_name, mode)
 find device in device descriptor table
 IF (allocated) or (out of service) or
    (mode incompatible with operations supported)
    or (no access permissions)
 THEN
    send(source, error)
 ELSE
    find free file descriptor
    set mode and device descriptor pointer
    set device status busy
    send(source, fd)

close(fd)
  IF (allocated to process)
  THEN
     set device status free
     deallocate file descriptor
     send(source,ok)
```

The **Device Descriptor** (held by the filemanager) contains :

```
Device Name
Access Control Information
Permitted Operations
Current Status : allocated, free, out of service etc
Current User : process id if allocated
Device Handler process id
Error Handling procedures - who to send what messages
```

8 Device Allocation

Devices can be :

- **Dedicated** : e.g. magnetic tape, terminal, printer - usually *allocated for long periods* - Only allocated to *authorised* processes
  OS must decide when to allocate (avoiding deadlock)
  simple policy - open fails if already opened
- **Shared** : e.g. discs, window terminals.
  OS provides *filing system* for discs (will cover this later)
- **Spooled** : e.g printers
  Provide sharing of non-sharable devices !
  Reduce I/O time and increase throughput

9 Device Handler

```
PROCEDURE Device_Handler

 initialise data structures and device
 REPEAT
  wait for next user request
  initiate requested I/O operation
  User data copied to or from buffer in user process
  wait for I/O to complete
  check for errors
  provide status to fileman
 FOREVER
END
```

The request is made to the device handler, via a *device handler interface* - if implemented as messages the request/reply message contains the following :

Request Message

Device No. - disc drive, terminal etc
Operation : read, write, position heads, rewind etc (device specific)
User Process ID : for identifying process requesting I/O
fd : needed for identifying replies
Buffer Pointer : address of data in user process
NBytes : number of bytes to transfer

Reply Message

User Process ID fd : User PID and fd identify relevant data structure in fileman *Status* : No. of bytes transfered or error

10 Device Manipulation

Devices can be separate or memory mapped :

Separate I/O Bus

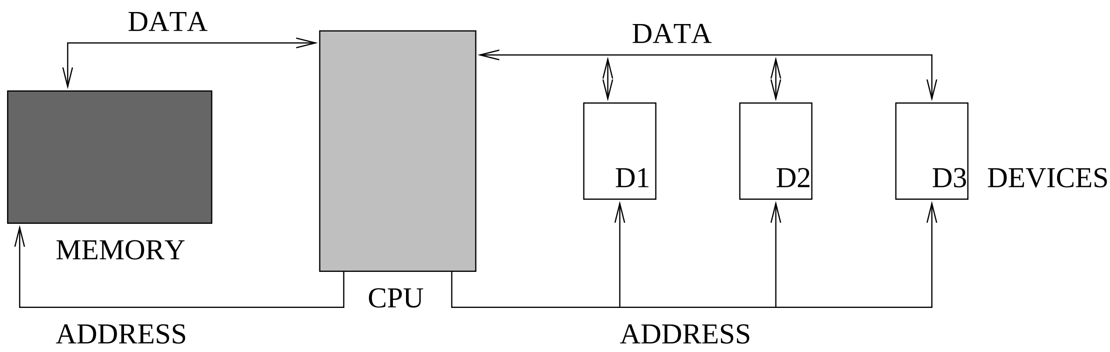

Figure 6: separate I/O Bus

- Special assembly code procedures called by high level language

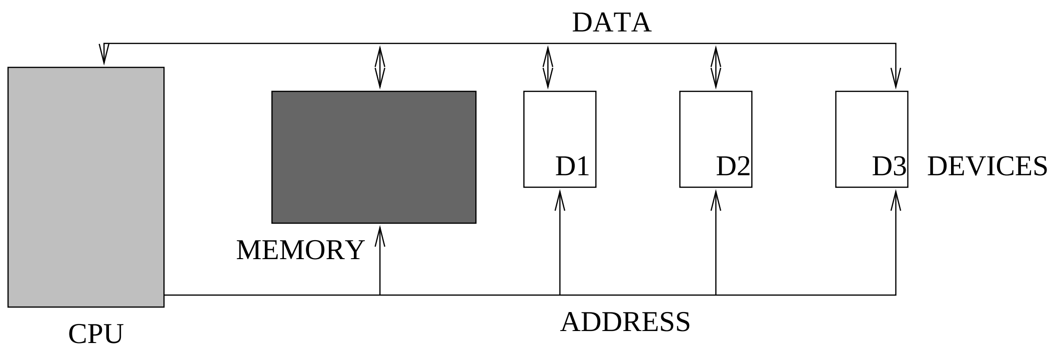

Figure 7: memory mapped I/O

- Device addressed as another *memory location*
- Use of high level language primitives which access memory, e.g. ch := rbuf
- More flexible

11 Polled Input/Output

- CPU checks I/O device repeatedly to check if input is available
- OR if the device is ready to accept output
- Used for 'bootstraps' and *panic* output
- Procedures which would be used for performing operations would include *InCh* and *Outch*
- Concept of *busy wait* : waiting for an event to happen

12 Interrupt Handling for devices

In the kernel process definition, the first thing performed is the initialisation of data structures (PCBs), and the installation of the interrupt vector table. The latter works as follows :

```
PROCEDURE Install(H:proc; vector:address; priority:integer)

 set vector to jump to flih
 set PSW in vector+2 {interrupts disabled}
 handlers[vector] := H {H is the handler}
 set priority field in PSW {needed for nested interrupts}
 pswtable[vector] := PSW {interrupts}
END
```

This allows for variable priority interrupt handlers, but FLIH runs with all interrupts disabled.

13 I/O Buffering

Unbuffered I/O

- Data transfered from (to) user directly to (from) device (no intermediate storage)
- Each *read* or *write* causes an actual (physical) I/O operation
- $\equiv$ run file manager and device handler for each transfer, could cause very high process switching overheads (when transferring characters for instance - possible to get process switch per character)
- Not the *best* way - clearly - any alternatives ?

Buffered I/O

- No direct transfer from (to) user to (from) I/O device
- Used to smooth peaks in I/O traffic
- Particularly useful for dealing with scenarios where there is a difference in data transfer rates - linking fast devices to slow ones for instance
- Also useful for storing data temporarily ...
- *OUTPUT* : User's output data transferred to OS output buffer. User continues operation, and is only suspended when the buffer becomes full
- *INPUT* : OS reads ahead for user. Read normally satisfied from buffer. User is blocked if buffer is empty

Double buffering :

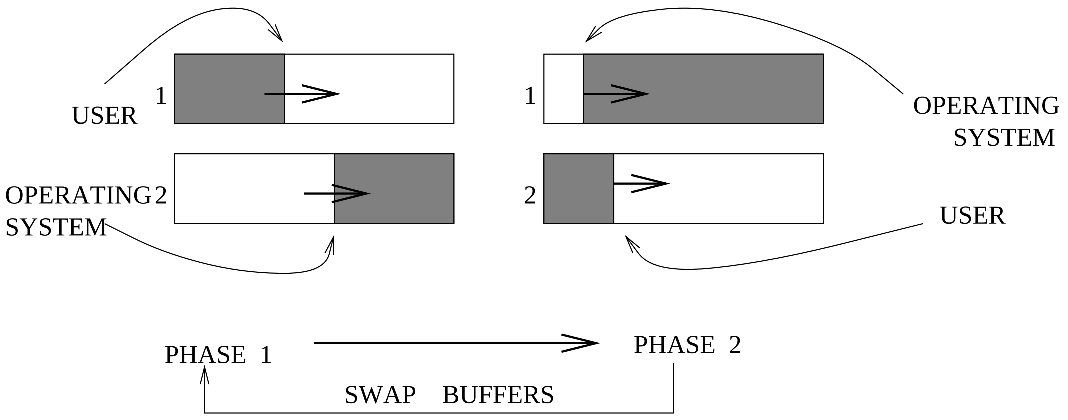

Figure 8: Double Buffering between OS and User

The Spooler :

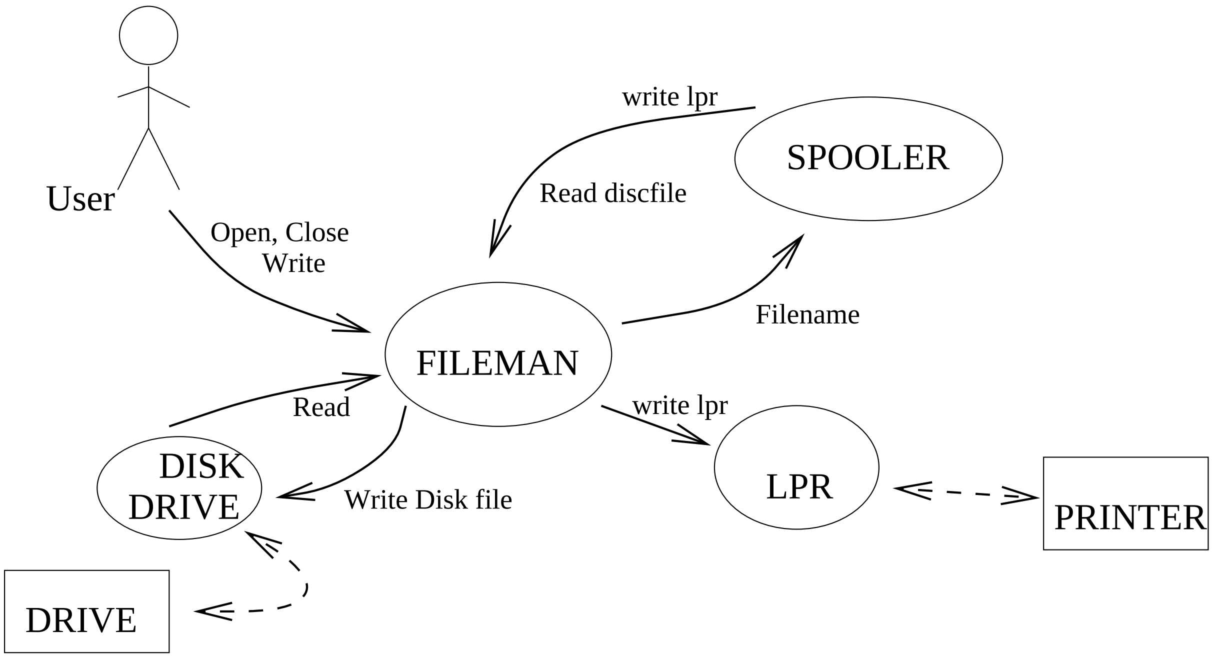

Figure 9: The Spooler Process

14 Spooling

- Used for sharing of *non-sharable* devices !!
- Blocking user when opening a file to an allocated non-sharable device causes delays and bottlenecks
- Use a spool to the an intermediate medium - such as a disk file
- Example includes a printer (lpr) spooler
- Only the printer can open certain files on the system, such as /dev/lpr
- Privledged access to files (associated with particular devices)

Here's a series of events that occur when a user (scm6mrw) tries to print a file :

1. The user (scm6mrw) opens */dev/lpr*
2. Fileman opens */tmp/scm6mrw*
3. *scm6mrw* writes to lpr file, Fileman diverts output to disc file
4. *scm6mrw* closes *dev/lpr*
5. Fileman closes */tmp/scm6mrw*
6. Fileman sends filename "scm6mrw" to spooler
7. Spooler eventually opens */tmp/scm6mrw* for reading
8. Spooler copies file to lpr
9. Spooler closes and deletes */tmp/scm6mrw*