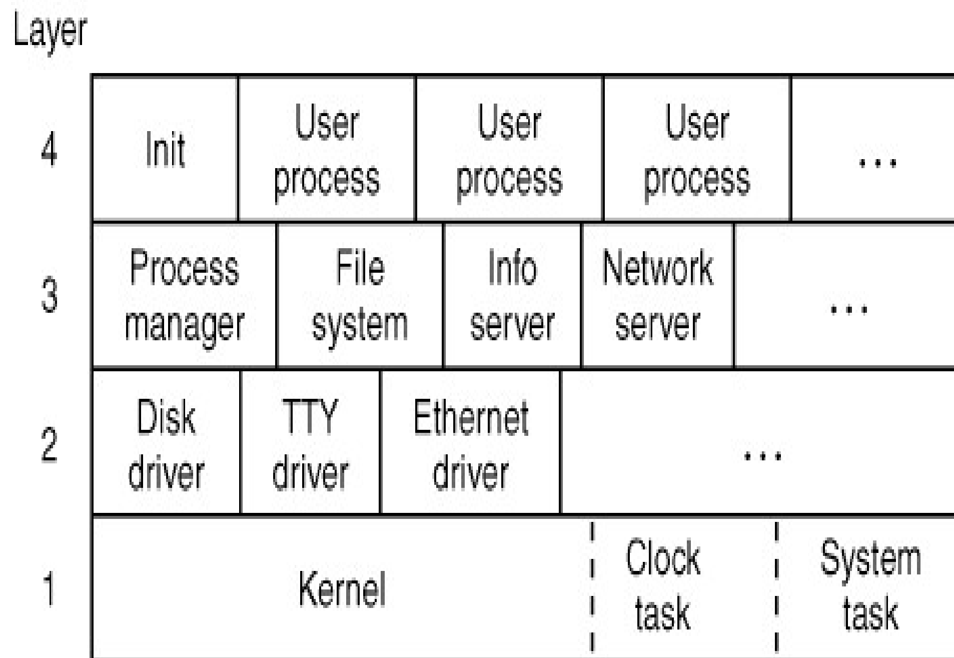
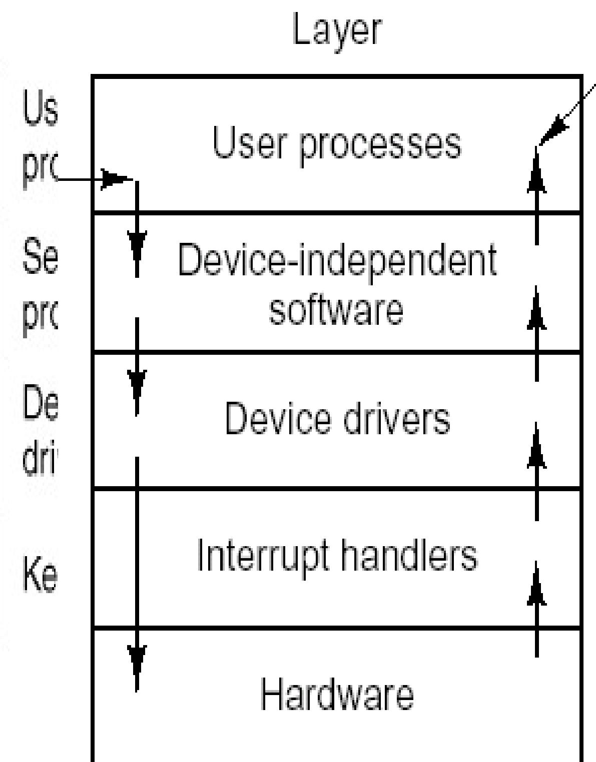
I/O in Minix3

Sistemi Operativi

Lez. 15

Architettura di riferimento

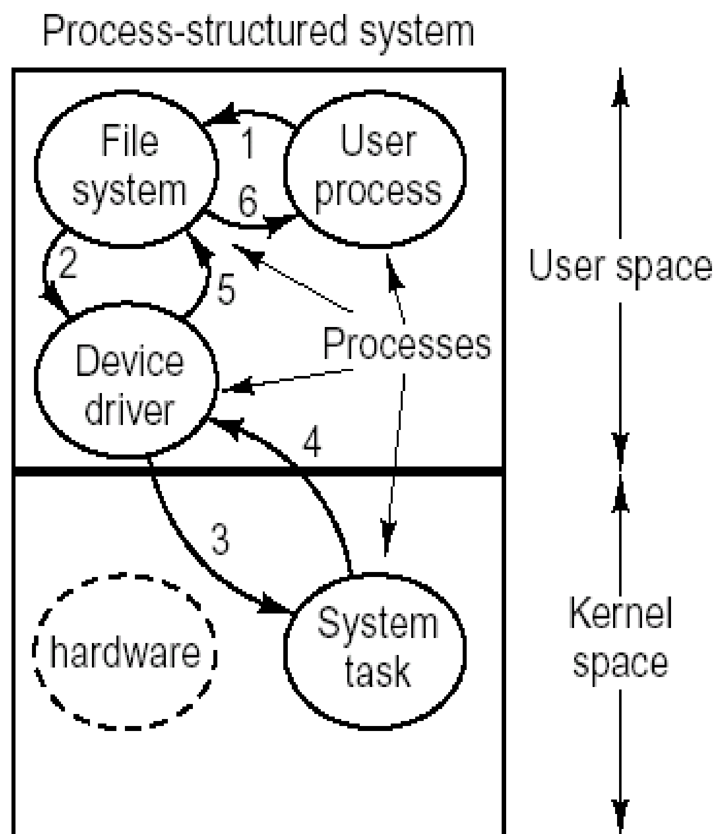




Device Driver

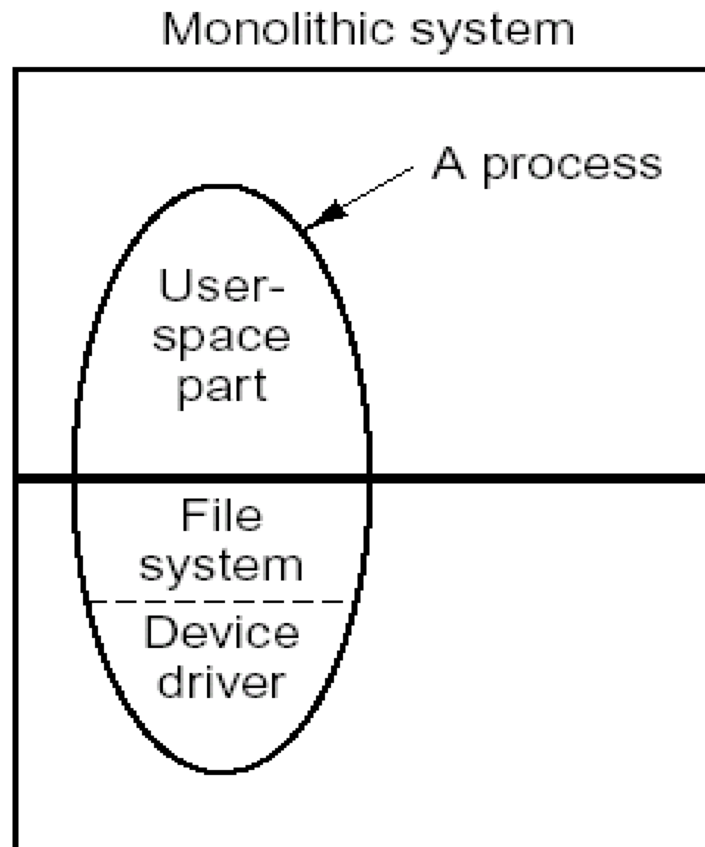
- Per ogni classe di dispositivo di I/O (HD, floppy, RAM disk) esiste un apposito driver che condivide un insieme di routine con i driver dedicati alla stessa tipologia di dispositivo (block, char)
- I driver operano in user mode per poter però operare correttamente hanno bisogno di:
  - Leggere e scrivere sulle porte di I/O
  - Rispondere agli interrupt
  - Accedere a zone di memoria del kernel
- Tutte queste funzioni sono svolte attraverso kernel call gestite dal system task

I/O in minix



1. Un processo che vuole leggere un file manda un messaggio a FS
2. FS, che svolge il ruolo di device-independent code, invia la richiesta al driver
3. Driver inoltra la richiesta al system task, via kernell call
4. Il kernel avvia l'operazione e trasferisce i dati tra i processi

Driver nei sistemi monolitici



Struttura generale di un driver (1)

```
message mess;

void io_driver( void ){
   int rcode, caller;

   initialize(); /* called once */
   while( TRUE ){
      receive( ANY, &mess );
      caller = mess.m_source;

      switch( mess.m_type ){
         case READ :    rcode = do_read(); break;
         case WRITE :   rcode = do_write(); break;
         case OTHER :   rcode = do_other(); break;
         default :      rcode = ERROR;
      }
      mess.m_type = DRIVER_REPLY;
      mess.REP_STATUS = rcode;
      send( caller, &mess );
   }
}
```

Struttura generale di un driver (2)

```
void do_something( void ){
   ...;
   set controller;
   receive( HARDWARE, &mess );
   ...
}
```

- Durante la chiamata di una procedura del tipo do …, il driver solitamente si interrompe in attesa di un messaggio di interrupt, non sono invece accettate richieste per lo svolgimento di nuove operazioni, sino al compimento di quella in corso

Operazioni comuni a block device

- OPEN
- CLOSE
- READ
- WRITE
- IOCTL: consente di modificare alcuni parametri operazionali (n.ro e dimensioni partizioni)
- SCATTERED_IO:abilita il file system a richiedere la lettura/scrittura di blocchi multipli

Memory driver

- Gestire gli accessi alla memoria centrale (RAM)
- Scopo principale: abilitare l'uso di una parte della RAM come se fosse una porzione di disco
- Gli accessi sono gestiti considerando la memoria un dispositivo a blocchi (RAM DISK) RAM DISK: parte della memoria centrale può essere usata per contenere parte del file system
- Il memory driver riceve la richiesta per la lettura/scrittura di un blocco dati, ricerca il blocco dati in memoria ed esegue l'operazione richiesta su RAM invece che su disco
- La root del file system è solitamente "montata" sul RAM disk

Memory devices

- Il memory driver di Minix gestisce diversi dispositivi:
  - 0:/dev/ram: è un vero RAM disk la cui dimensione è definita in fase di boot
  - 1:/dev/mem:consente di accedere a tutta le memoria fisica a partire dall'indirizzo 0
  - 2:/dev/kmem consente l'accesso alla memoria occupata dal kernel
  - 3:/dev/null: cestino
  - 5:/dev/zero: una write su /dev/zero equivale a una write su /dev/null mentre una read da /dev/zero equivale ad azzerare una zona

Block device driver

- Ogni driver ha la stessa struttura:
  - Inizializzazione specifica per il driver
  - Chiamata della procedura driver_task che esegue una fase di inizializzazione comune a tutti i device a blocchi ed esegue il loop di ricezione richieste

```
11669  /*===========================================================================*
11670   *                                 main                                       *
11671   *===========================================================================*/
11672  PUBLIC int main(void)
11673  {
11674  /* Main program. Initialize the memory driver and start the main loop. */
11675    m_init();
11676    driver_task(&m_dtab);
11677    return(OK);
11678  }
```

Strutture dati

```
10828  /* Info about and entry points into the device dependent code. */
10829  struct driver {
10830    _PROTOTYPE( char *(*dr_name), (void) );
10831    _PROTOTYPE( int (*dr_open), (struct driver *dp, message *m_ptr) );
10832    _PROTOTYPE( int (*dr_close), (struct driver *dp, message *m_ptr) );
10833    _PROTOTYPE( int (*dr_ioctl), (struct driver *dp, message *m_ptr) );
10834    _PROTOTYPE( struct device *(*dr_prepare), (int device) );
10835    _PROTOTYPE( int (*dr_transfer), (int proc_nr, int opcode, off_t position,
10836                                              iovec_t *iov, unsigned nr_req) );
10837    _PROTOTYPE( void (*dr_cleanup), (void) );
10838    _PROTOTYPE( void (*dr_geometry), (struct partition *entry) );
10839    _PROTOTYPE( void (*dr_signal), (struct driver *dp, message *m_ptr) );
10840    _PROTOTYPE( void (*dr_alarm), (struct driver *dp, message *m_ptr) );
10841    _PROTOTYPE( int (*dr_cancel), (struct driver *dp, message *m_ptr) );
10842    _PROTOTYPE( int (*dr_select), (struct driver *dp, message *m_ptr) );
10843    _PROTOTYPE( int (*dr_other), (struct driver *dp, message *m_ptr) );
10844    _PROTOTYPE( int (*dr_hw_int), (struct driver *dp, message *m_ptr) );
10845  };
```

driver_task

```
11057 PUBLIC void driver_task(dp)
11058 struct driver *dp;       /* Device dependent entry points. */
11059 {
11062   int r, proc_nr;
11063   message mess;
11066   init_buffer(); /* Get a DMA buffer */
.....
11071   while (TRUE) {
11073         /* Wait for a request to read or write a disk block. */
11074         if(receive(ANY, &mess) != OK) continue;
11076         device_caller = mess.m_source;
11077         proc_nr = mess.PROC_NR;
11080         switch(mess.m_type) {
11081             case DEV_OPEN:              r = (*dp->dr_open)(dp, &mess);   break;
11082             case DEV_CLOSE:             r = (*dp->dr_close)(dp, &mess);  break;
11083             case DEV_IOCTL:             r = (*dp->dr_ioctl)(dp, &mess);  break;
11084             case CANCEL:                r = (*dp->dr_cancel)(dp, &mess);break;
11085             case DEV_SELECT:            r = (*dp->dr_select)(dp, &mess);break;
11086
11087             case DEV_READ:
11088             case DEV_WRITE:      r = do_rdwt(dp, &mess);   break;
11089             case DEV_GATHER:     /* Reading/writing multiple blocks in one go */
11090             case DEV_SCATTER:    r = do_vrdwt(dp, &mess); break;
11091
11092             case HARD_INT:       if(dp->dr_hw_int) (*dp->dr_hw_int)(dp, &mess);
11096                                  continue;
.....
11107         }
```

driver_task

```
11109         /* Clean up leftover state. */
11110         (*dp->dr_cleanup)();
11111
11112         /* Finally, prepare and send the reply message. */
11113         if (r != EDONTREPLY) {
11114                 mess.m_type = TASK_REPLY;
11115                 mess.REP_PROC_NR = proc_nr;
11116                 /* Status is # of bytes transferred or error code. */
11117                 mess.REP_STATUS = r;
11118                 send(device_caller, &mess);
11119         }
11120    }
11121 }
```

read/write

- Svolte attraverso la funzione do_rdwt, che a sua volta richiama due funzioni dr_prepare e dr_transfer
- dr_prepare: individua indirizzo di partenza e dimensione del disco (partizione o sottopartizione su cui deve essere effettuata l'operazione)
- dr_transfer:
  - RAM DISK: chiede al kernel di trasferire fisicamente byte da una zona di memoria all'altra
  - Dischi: alle operazioni precedenti va aggiunta anche la gestione della periferica da parte del kernel

```
11148 PRIVATE int do_rdwt(dp, mp)
11149 struct driver *dp;                  /* device dependent entry points */
11150 message *mp;                        /* pointer to read or write message */
11151
11152 /* Carry out a single read or write request. */
11153   iovec_t iovec1;
11154   int r, opcode;
11155   phys_bytes phys_addr;
11156
11157   /* Disk address?  Address and length of the user buffer? */
11158   if (mp->COUNT < 0) return(EINVAL);
11159
11160   /* Check the user buffer. */
11161   sys_umap(mp->PROC_NR, D, (vir_bytes) mp->ADDRESS, mp->COUNT, &phys_addr)
11162   if (phys_addr == 0) return(EFAULT);
11163
11164   /* Prepare for I/O. */
11165   if ((*dp->dr_prepare)(mp->DEVICE) == NIL_DEV) return(ENXIO);
11166
11167   /* Create a one element scatter/gather vector for the buffer. */
11168   opcode = mp->m_type == DEV_READ ? DEV_GATHER :  DEV_SCATTER;
11169   iovec1.iov_addr = (vir_bytes) mp->ADDRESS;
11170   iovec1.iov_size = mp->COUNT;
11171
11172   /* Transfer bytes from/to the device. */
11173   r = (*dp->dr_transfer)(mp->PROC_NR, opcode, mp->POSITION, &iovec1, 1);
11174
11175   /* Return the number of bytes transferred or an error code. */
11176   return(r == OK ? (mp->COUNT - iovec1.iov_size) :  r);
11177
```

m_transfer per RAM Disk

- Implementa fisicamente le operazioni di lettura/scrittura sui dispositivi gestiti dal memory driver
- Le operazioni gestite sono:
  - DEV_GATHER: nel caso di lettura di uno o più blocchi
  - DEV_SCATTER: nel caso di scritture di uno o più blocchi

m_transfer

```
11706 PRIVATE int m_transfer(proc_nr, opcode, position, iov, nr_req)
11707 int proc_nr;                    /* process doing the request */
11708 int opcode;                     /* DEV_GATHER or DEV_SCATTER */
11709 off_t position;                 /* offset on device to read or write */
11710 iovec_t *iov;                   /* pointer to read or write request vector */
11711 unsigned nr_req;                /* length of request vector */
11712 {
11713 /* Read or write one the driver's minor devices. */
11714   phys_bytes mem_phys;
11715   int seg;
11716   unsigned count, left, chunk;
11717   vir_bytes user_vir;
11718   struct device *dv;
11719   unsigned long dv_size;
11720   int s;
```

m_transfer

Dispositivi con indirizzi virtuali

```
.....
11739        /* Virtual copying. For RAM disk, kernel memory and boot device. */
11740        case RAM_DEV:
11741        case KMEM_DEV:
11742        case BOOT_DEV:
11743            if (position >= dv_size) return(OK);        /* check for EOF */
11744            if (position + count > dv_size) count = dv_size - position;
11745            seg = m_seg[m_device];
11746
11747            if (opcode == DEV_GATHER) {  /* copy from SELF to proc_nr */
11748                sys_vircopy(SELF,seg,position, proc_nr,D,user_vir, count);
11749            } else {
11750                sys_vircopy(proc_nr,D,user_vir, SELF,seg,position, count);
11751            }
11752            break;
```

m_transfer

```
11754           /* Physical copying. Only used to access entire memory. */
11755           case MEM_DEV:
11756               if (position >= dv_size) return(OK);        /* check for EOF */
11757               if (position + count > dv_size) count = dv_size - position;
11758               mem_phys = cv64ul(dv->dv_base) + position;
11759
11760               if (opcode == DEV_GATHER) {                  /* copy data */
11761                   sys_physcopy(NONE, PHYS_SEG, mem_phys, proc_nr, D, user_vir, count);
11763               } else {
11764                   sys_physcopy(proc_nr, D, user_vir, NONE, PHYS_SEG, mem_phys, count);
11766               }
11767               break;
.....
11787           }
11788
11789           /* Book the number of bytes transferred. */
11790           position += count;
11791           iov->iov_addr += count;
11792           if ((iov->iov_size -= count) == 0) { iov++; nr_req--; }
11793
11794     }
11795     return(OK);
11796 }
```

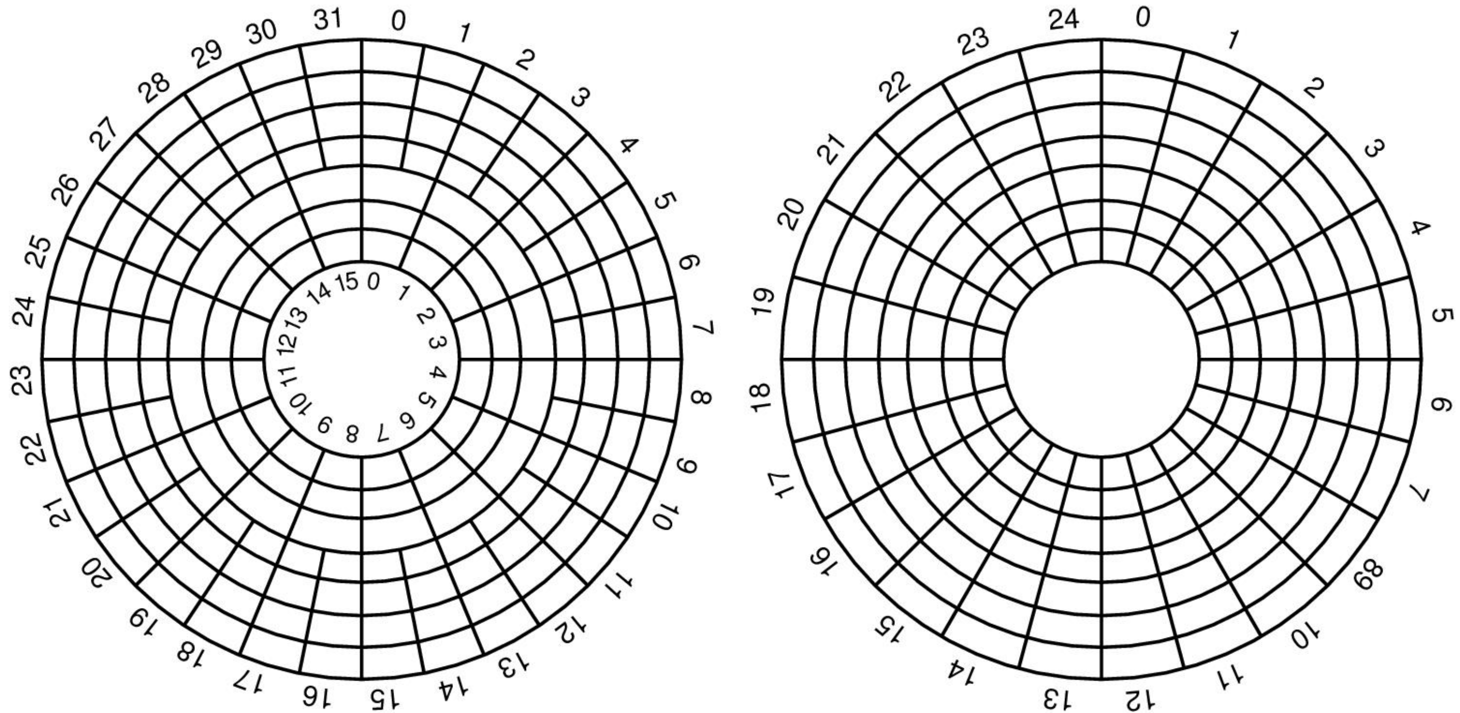
```
00001 #include "syslib.h"
00002
 00003   PUBLIC int sys_vircopy(src_proc, src_seg, src_vir, 00004
   dst_proc, dst_seg, dst_vir,   bytes)
00005 int src_proc; /* source process */
00006 int src_seg; /* source memory segment */
00007 vir_bytes src_vir; /* source virtual address */
00008 int dst_proc; /* destination process */
00009 int dst_seg; /* destination memory segment */
00010 vir_bytes dst_vir; /* destination virtual address */
00011 phys_bytes bytes; /* how many bytes */
00012 {
00017
00018 message copy_mess;
00019
00020 if (bytes == 0L) return(OK);
00021 copy_mess.CP_SRC_ENDPT = src_proc;
00022 copy_mess.CP_SRC_SPACE = src_seg;
00023 copy_mess.CP_SRC_ADDR = (long) src_vir;
00024 copy_mess.CP_DST_ENDPT = dst_proc;
00025 copy_mess.CP_DST_SPACE = dst_seg;
00026 copy_mess.CP_DST_ADDR = (long) dst_vir;
00027 copy_mess.CP_NR_BYTES = (long) bytes;
00028 return(_taskcall(SYSTASK, SYS_VIRCOPY, &copy_mess));
00029 }
```

I Dischi

Hard Disk

- composto da una pila di dischi ricoperti di materiale magnetico sovrapposti su ogni facciata del disco opera una testina che effettua le operazioni di lettura/scrittura

Disk Hardware (2)



- Struttura fisica di un disco con doppio settore sulle tracce esterne
- Struttura logica dello stesso dispositivo

Disk Hardware (1)

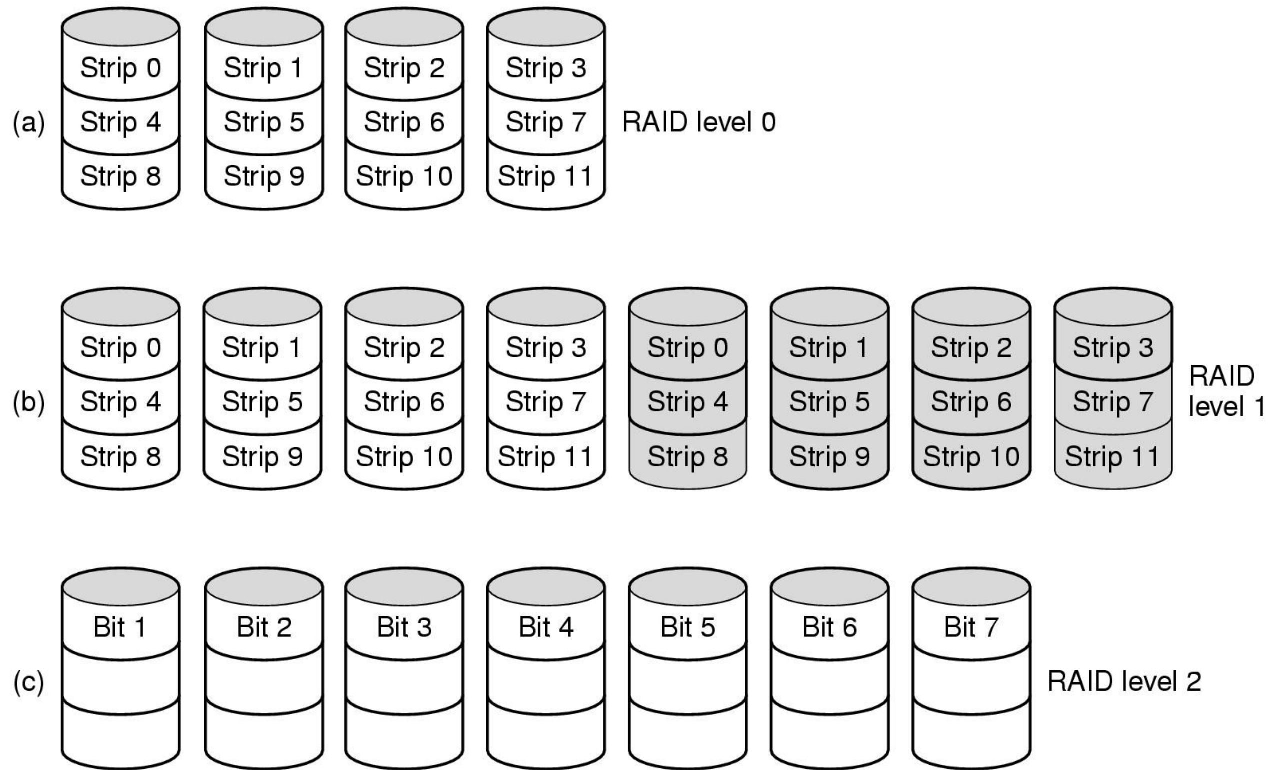
Parameter	IBM 360-KB floppy disk	WD 18300 hard disk
Number of cylinders	40	10601
Tracks per cylinder	2	12
Sectors per track	9	281 (avg)
Sectors per disk	720	35742000
Bytes per sector	512	512
Disk capacity	360 KB	18.3 GB
Seek time (adjacent cylinders)	6 msec	0.8 msec
Seek time (average case)	77 msec	6.9 msec
Rotation time	200 msec	8.33 msec
Motor stop/start time	250 msec	20 sec
Time to transfer 1 sector	22 msec	17 μsec

Evoluzione dei dischi a partire dal dispositivo originale del PC IBM sino al modello Western Digital WD 18300

Disk Formatting (1)

Settore di un disco

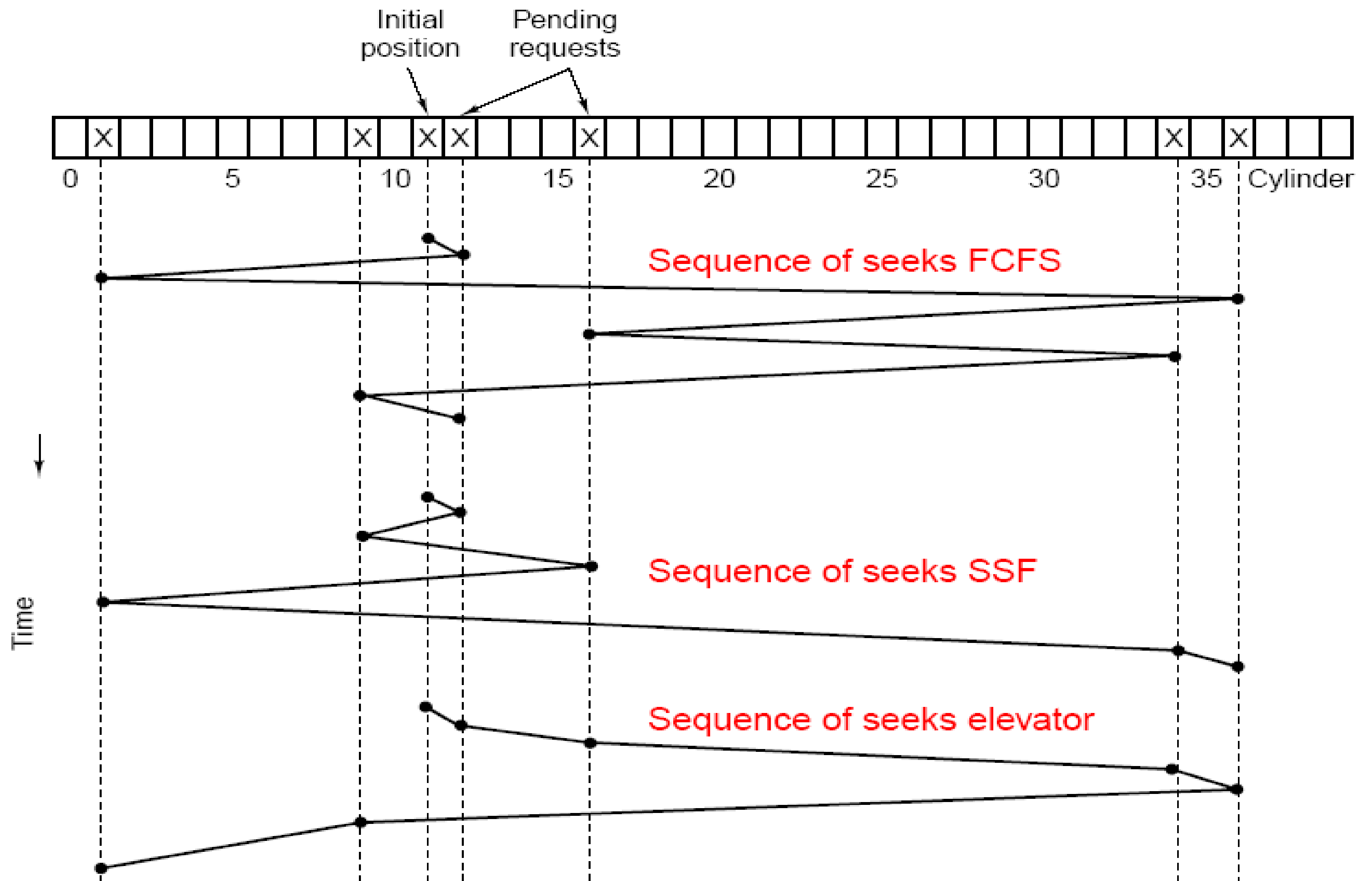
Redundant Array of Independent Disk



- I drive di Backup e parity sono in grigio

Disk Arm Scheduling

- Il tempo richiesto per leggere o scrivere un blocco del disco è determinato da tre fattori
  1. Seek time
  2. Rotational delay
  3. Actual transfer time
- Il termine preponderante è il tempo di seek
- Il controllo degli errori è effettuato dal device controller


Initial position
Pending requests
0
5
10
15
20
25
30
35
Cylinder
Time
Sequence of seeks FCFS
Sequence of seeks SSF
Sequence of seeks elevator

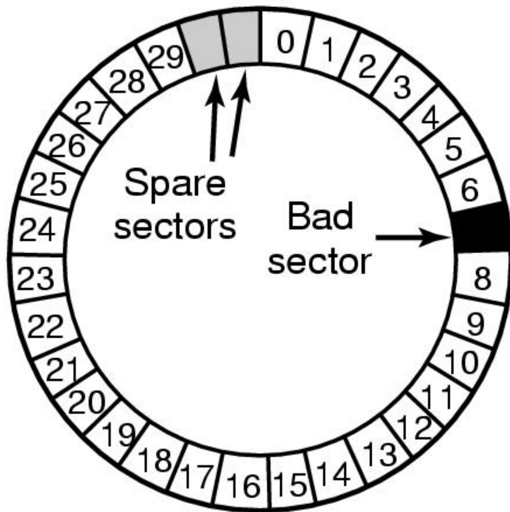
Errori

- Nella gestione di un disco si possono verificare diverse tipologie di errore, alcune di queste possono essere risolte dal controller, nella maggior parte dei casi è il driver che deve trovare una soluzione
  - **Errori di programmazione:** invio di parametri errati (n.ro cilindro, n.ro settore ecc.) al controller. In questi casi l'operazione va interrotta sperando che non si verifichi troppo frequentemente
  - **Transient checksum error:** errori di lettura/scrittura a causa di tracce di polvere. In questi casi si tratta di riprovare un certo numero di volte
  - **Permanent checksum error:** il settore viene marcato come cattivo (bad). Solo l'hw può porre rimedio a questa situazione

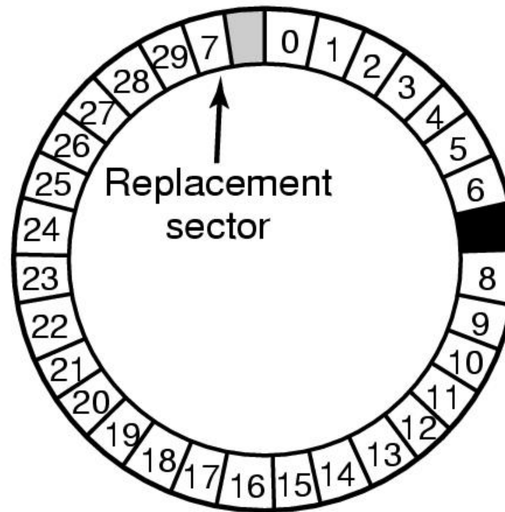
Errori

- **Seek errors:** il braccio non è ben calibrato e non si posiziona correttamente sulle tracce. È necessario avviare un’operazione per ricalibrare il braccio, quando il controller lo rende possibile
- **Controller error:** anche in questo caso va avviata un’operazione di reset del controller, che può comunque risultare non risolutiva

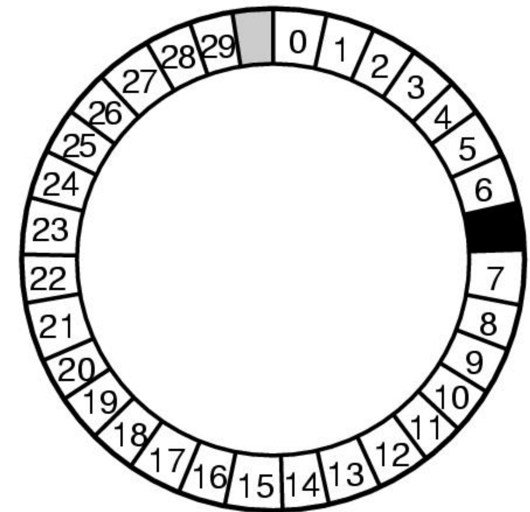
Bad Block



(a)



(b)

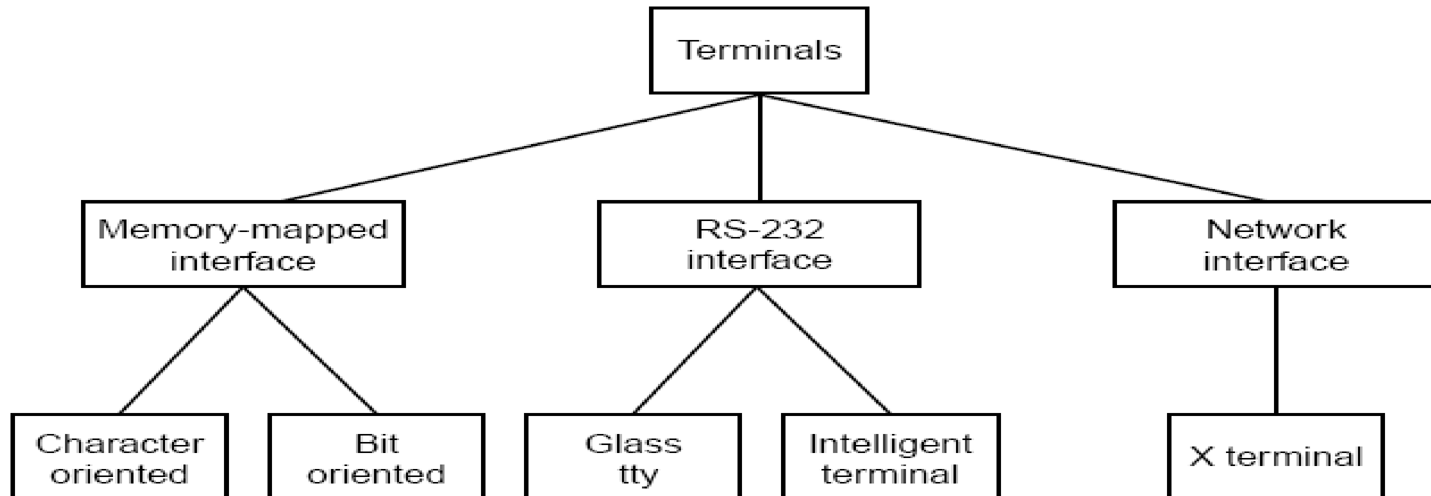

(c)

a. Una traccia con settore rovinato
b. Sostituzione di un settore disponibile con il contenuto del settore rovinato
c. Shifting di tutti i settori per bypassare quello rovinato

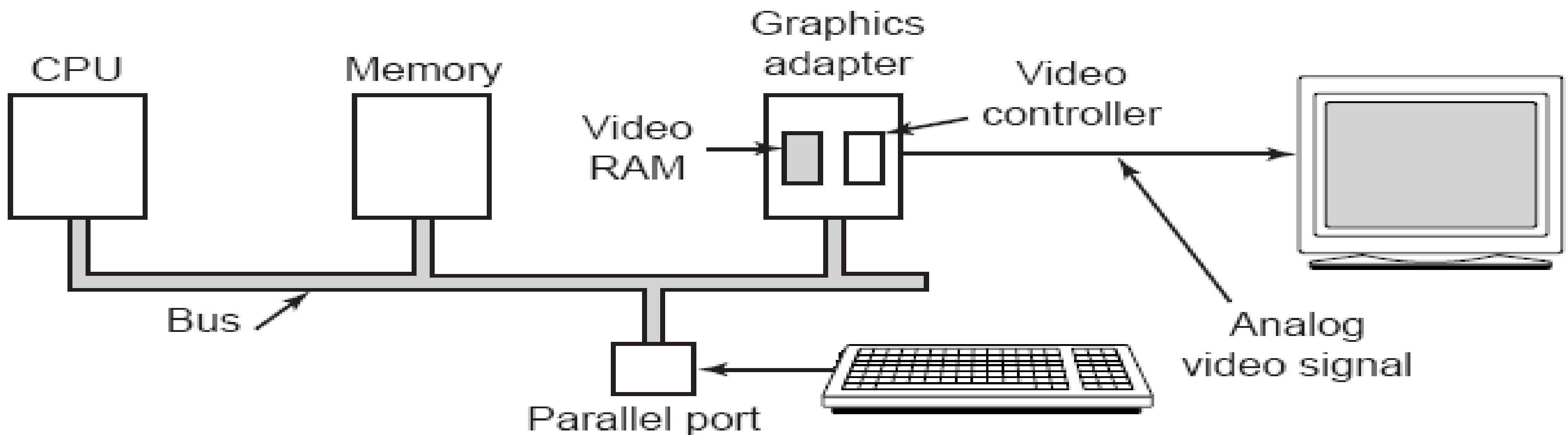
Terminali

Terminali

- Formati da una tastiera ed un video che sono solitamente gestiti separatamente, si distinguono tre tipologie di terminali in funzione della modalità con cui sono collegati



Memory-Mapped

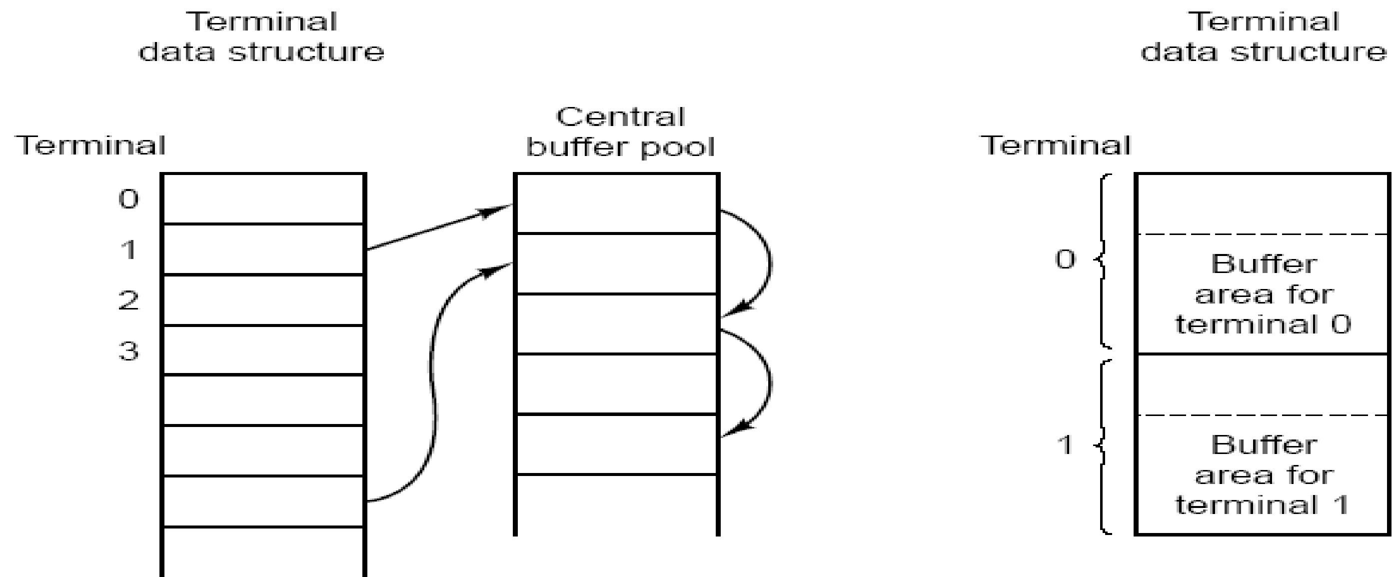


- **Character-oriented**: il display visualizza caratteri e ogni carattere sullo schermo occupa 2 byte nella video RAM. Un video 25x80 necessita quindi di una video RAM di 4000 byte
- **Bit-oriented**: l'elemento di visualizzazione elementare è il pixel, Nelle configurazioni più semplici ad ogni bit della VIDEO RAM è associato un pixel, nei video più elaborati sono necessari 24 bit per descrivere ciascun pixel. Un (800×1024) a colori richiede quindi una video RAM di 2.400 Kbyte

Keyboard Driver

- In un dispostivo memory mapped la tastiera e completamente disaccoppiata dal video. Ad ogni tasto premuto genera un interrupt, che risveglia l'interrupt handler e conseguentemente il tty driver che a sua volta deve memorizzare il dato inserito e passarlo all'applicazione
  - **raw**: i caratteri sono passati uno alla volta così come inseriti, senza alcuna interpretazione da parte del driver, inclusi backspaces, delete, etc.
  - **cooked**: il driver gestisce una linea alla volta e quindi si fa carico di interpretare i caratteri di controllo prima di passarli all'applicazione (**canonical mode** in POSIX).
  - In entrambi i casi si rende necessario da parte del driver un'attività di bufferizzazione per consentire la sovrapposizione di attività di input con quelle di elaborazione

Buffering



- **Due appocci:**
  - I buffer sono reperiti all'interno di un pool comune presente nella memoria centrale
  - Ad ogni terminale viene preassegnato il proprio buffer (dispendioso in termini di spazio)

Controllo del video

- Più semplice della tastiera:
- Basta inserire i caratteri appropriati nella corrispondente posizione della video RAM- il controller effettua il resto
- **Ulteriori supporti forniti dal controller:**
  - Scroll register: che punta alla prima linea da visualizzare, sarà il controller che that points to the first line that is to be displayed. The hardware then just wraps around video RAM.
  - Cursor register: usati per fornire le coordinate del cursore gestito dal mouse o dalle arrow keys