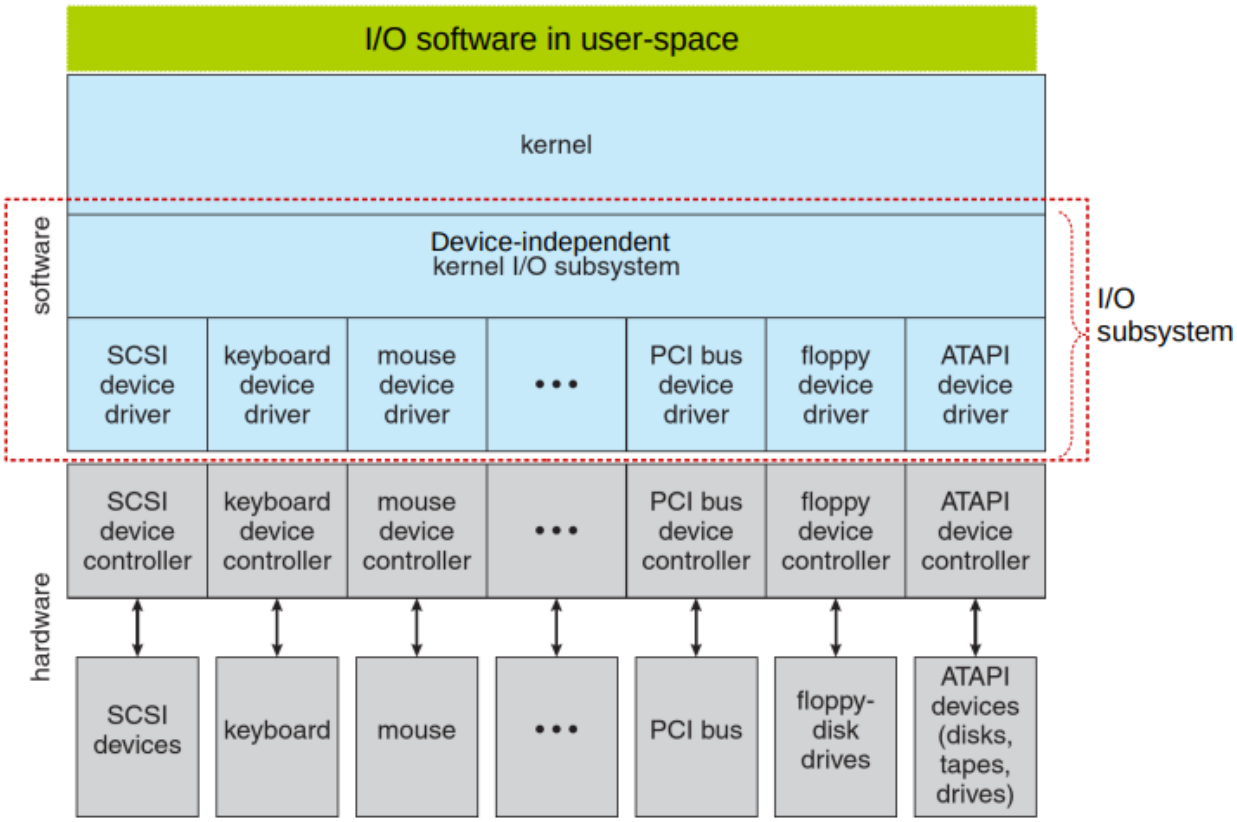
I/O Software

Come viene svolto

L'I/O software è guidato da appositi componenti, il sottosistema kernel di I/O è una parte del sistema operativo che separa il SO dal compito di gestire le periferiche di I/O parte software.

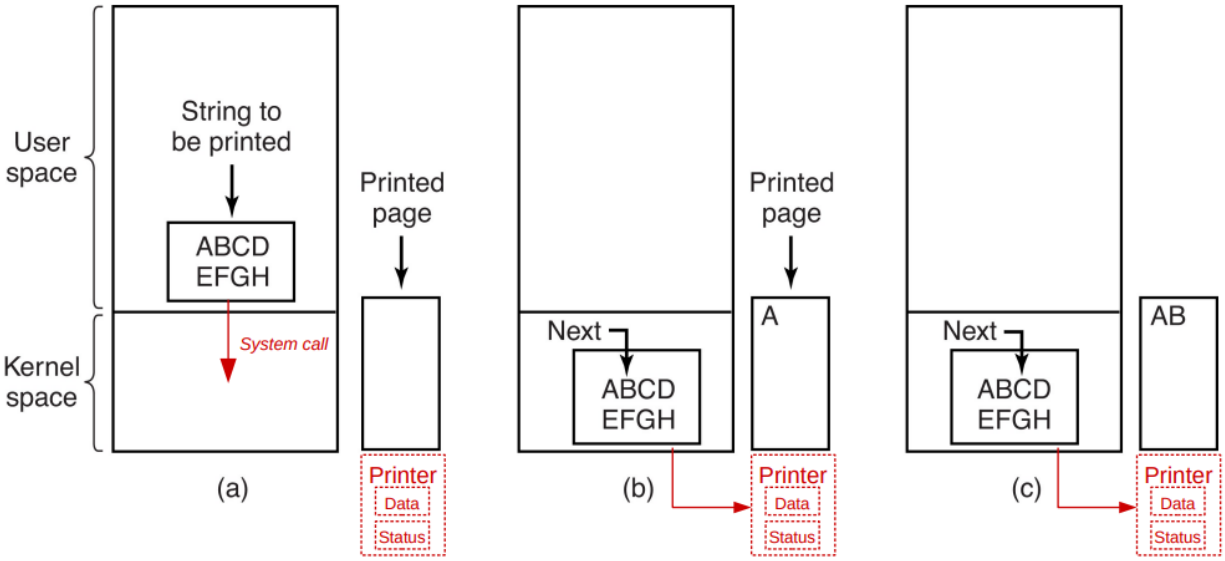


(La parte verde gira in user mode, la parte in verde in kernel mode)

Il codice che gira in **user space**, con apposite funzioni come **fopen**, **fprintf**. Queste ultime ci fanno accedere ai dati di una periferica, vengono tradotte in system calls e invocare il driver che gestisce la periferica. **Kernel fa da intermediario**.

Come fa il SO a fare queste richieste?

Tipicamente il SO quando deve eseguire una funzione ad esempio **printf** la prima cosa che avviene è il passaggio da user mode a kernel come con una system call. Per il secondo passaggio abbiamo il salvataggio in un buffer del kernel e ora avviene la comunicazione, viene letto lo **status register** per vedere se la stampante può o meno prendere in input. Avviene così un ciclo tra **device idle** (quando può ricevere un carattere) e lo stato di **device busy**.



I/O Programmato (PIO o Busy Waiting)

Il SO **legge o scrive** dati da un dispositivo, il SO fa un **busy waiting** continuamente controllando lo stato del dispositivo ed infine il SO esegue questo **polling** (ovvero interrogazioni) fin quando non **termina**.

```
copy_from_user(buffer, p, count); // p è buffer del kernel
for(int i = 0; i < count; i++) { // loop per ogni carattere
    while(*printer_status_reg != READY); // busy waiting sullo s
    *printer_data_register = p[i]; // stampiamo sul data registe
}
return_to_user();
```

Il contenuto dei registri in memoria è l'indirizzo dei registri del device.

Esempio

1. La velocità di una stampante è 10ms per ogni carattere
2. Per ogni carattere la CPU perde 10ms ad aspettare di poter inserire il prossimo
3. Se ci sono 8 caratteri la CPU perde 80ms per una stringa lunga 8 caratteri, pensiamo a grandi dimensioni perdiamo molto di più.

Questo approccio quindi è ottimo quando i tempi di attesa del device sono corti e la CPU non ha altro da fare nei sistemi a singola CPU, ha delle criticità come che la CPU è impegnata fino alla fine dell'I/O.

I/O Guidato dalle Interruzioni

Il SO operativo viene interrotto da una operazione di I/O da un processo, viene servito e il processo viene bloccato, il SO fa altro nel mentre fin tanto che non arriva un segnale di ritorno dal dispositivo compreso dal processo, completamento o di errore. Dopo che la gestione dell'interruzione è terminata, il SO ripete fin tanto che non ha finito, a questo punto il processo è sbloccato.

```
copy_from_user(buffer, p, count); // p è buffer del kernel
enable_interrupts();
while(*printer_status_reg != READY);
*printer_data_register = p[0];
scheduler(); // Il processo diventa waiting
```

Interrupt Handler

```
if(count == 0) {
    unblock_user(); // Lo stato del processo passa da waiting a
}
else {
    *printer_data_register = p[i];
    count = count - 1;
    i = i + 1;
}
```

```
acknowledge_interrupt(); // Diciamo alla periferica che abbiamo
return_from_interrupt(); // RETINT: ripristina lo stato precede
```

Quello che accade è: quando viene scritto il primo carattere avviene un interrupt gestito dall'interrupt handler, il quale vede se ci sono ancora dei blocchi da scrivere, se ci sono continua altrimenti .

Questo cambio di contesto viene fatto ad **ogni carattere,** il processo cambia da waiting a ready e poi running.

Esempio

1. La velocità di una stampante è 10ms per ogni carattere
2. La CPU impiega 4ms per lo switch
3. Se ci sono 8 caratteri la CPU perde 64ms + 80ms per il context switch per una stringa lunga 8 caratteri, pensiamo a grandi dimensioni perdiamo molto di più.

I punti di forza sono che la CPU può fare altro quando il dispositivo richiede un I/O, i punti critici sono che richiede tempo fare lo switch di contesto.

I/O usando il DMA

Il DMA porta il I/O senza disturbare la CPU, essenzialmente il DMA fa busy waiting invece che farlo fare alla CPU.

```
copy_from_user(buffer, p, count); // p è buffer del kernel
set_up_DMA_controller(); // Qui vengono svolte le operazioni
scheduler();
```

Interrupt Handler

```
acknowledge_interrupt();
unblock_user();
return_from_interrupt();
```

I punti di forza è che la CPU riceve un solo interrupt quando il DMA ha terminato la sua esecuzione, ci sono alcune criticità come che l'uso del DMA richiede

dispositivi che lo implementino, ci vuole una complessità maggiore sul DMA.

Design Goal

In riferimento alla progettazione dei driver e dispositivi per i sistemi operativi dobbiamo tener conto di diversi fattori:

1. Indipendente dal dispositivo

Ad esempio un programma che **legge** (scrive) **un file** come **input** (output) e deve essere in grado di **scrivere** (leggere) **un file** di qualunque disco (hard disk, ssd, chiavetta) senza modificare ogni volta

> 💡 **sort** <input> <output>

2. Schema di nome

Un nome di un file o dispositivo non deve dipendere dal dispositivo, l'utente non deve essere preoccupato di come si chiama il dispositivo. Per esempio in UNIX tutti i dispositivi possono essere riconosciuti da un nome di path come file regolari.

> 💡 **cp** <path_file_to_copy> <path_destination>

3. Gestione degli errori

Bisogna gestire gli errori nella maniera più **trasparente** possibile ovvero **non visibili all'utente**, in oltre la gestione degli errori deve avvenire **vicina all'hardware possibile**, è il **driver che deve gestirli**, proprio per questo verranno **mascherati dal controller** in modo che non siano visibili dal driver per fare meno operazioni inutili.

4. Supporto di modalità multiple di trasferimento

La maggior parte dei dispositivi usa **I/O asincroni** ovvero guidati alle interruzioni, i programmi utente sono molto più semplici invece se operano in maniera **sincrona** e per mettersi d'accordo queste due modalità abbiamo dei buffer. Ad esempio nel DMA quando il programma chiama lo scheduler() abbiamo un blocco del programma.

5. Gestione del buffer

Usiamo i buffer quando dobbiamo immagazzinare dati che non sono completi ad esempio nei network salviamo i pacchetti in arrivo che andranno a comporre il pacchetto di rete. Questo perchè il SO non sa dove mettere dati incompleti e quindi deve salvarli momentaneamente fin quando non prendono senso.

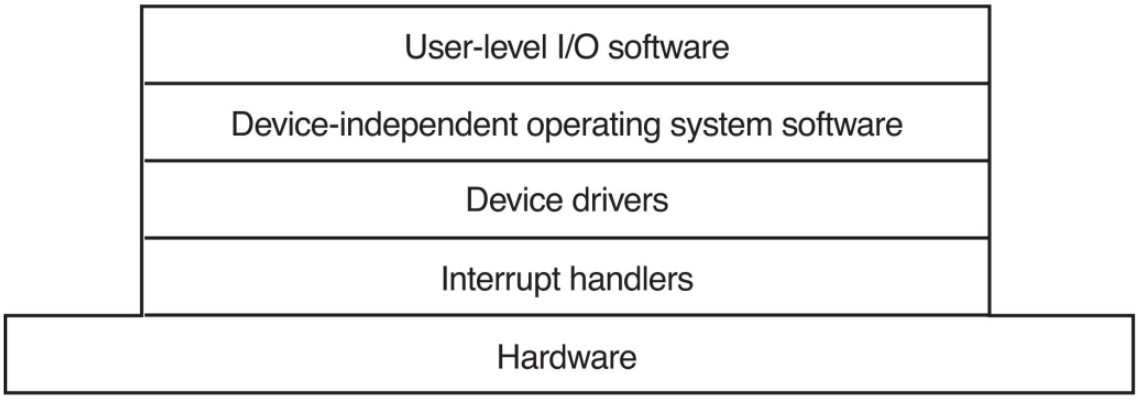
Servono anche quando viaggiamo ad alte velocità e quindi non vogliamo che la CPU svolga più lavoro e che vada sprecato, come quando vediamo i video online abbiamo un buffer con i frame del video così che non si blocchi.

6. Dispositivi condivisi e dedicati

Ci sono dispositivi che possono essere condivisi tra gli utenti allo stesso tempo, quelli dedicati invece possono essere usati da un solo processo alla volta. Il SO deve essere in grado di gestire entrambi.

Generalmente un output è organizzato a 4 livelli:

- Software **livello utente**
- Software **sistema operativo indipendente dai dispositivi**
- **Drivers** dei dispositivi
- **Gestori** delle **interruzioni**
- Hardware

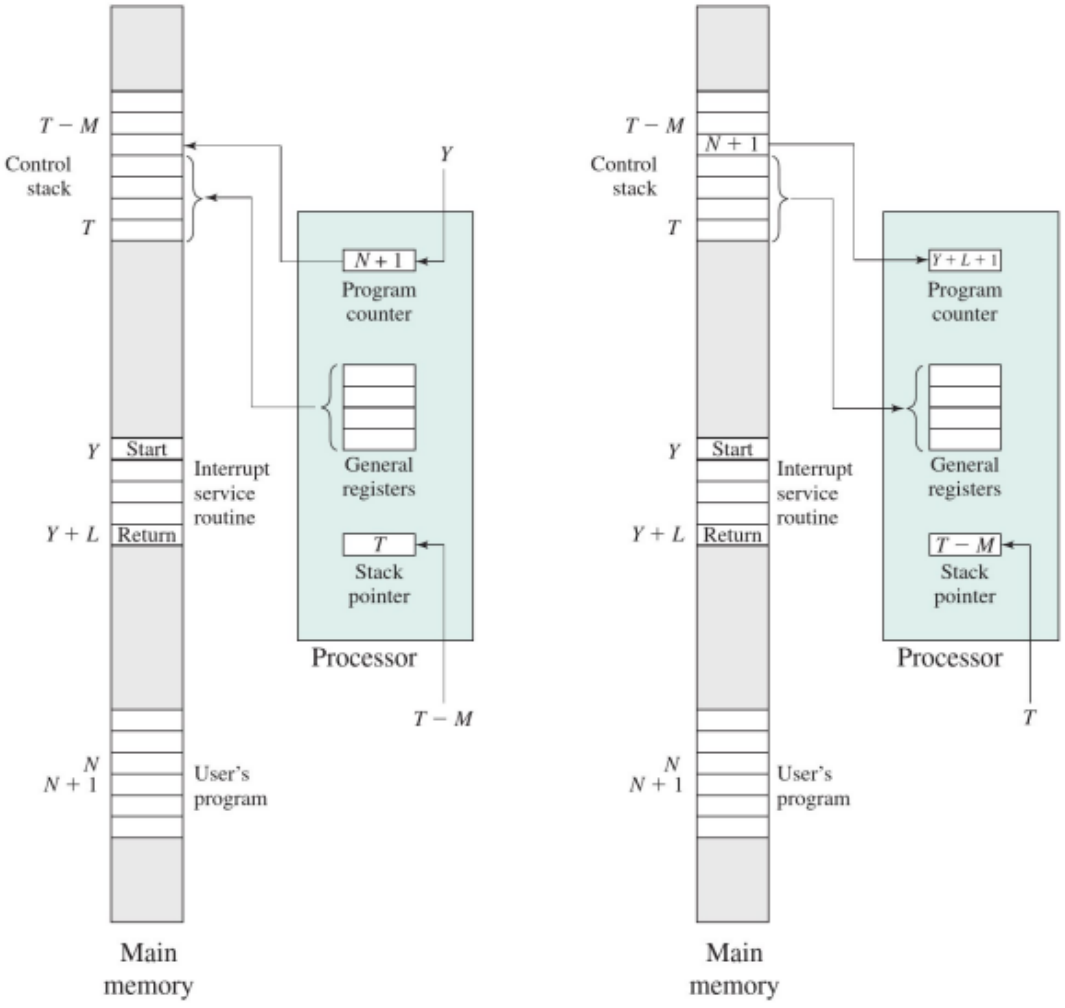


I livelli servono per rendere trasparente il modo in cui un meccanismo viene svolto nel livello, quello che importa è che input e output del livello siamo sempre uguali.

1. Gestori delle interruzioni

Il meccanismo predominante è quello guidato alle interruzioni tramite DMA, in questo livello mascheriamo ai livelli superiori l'avvenimento di una interruzione:

1. Arrivo della interruzione, il **SO salva lo stato della CPU** (attenzione è il SO e non la CPU perchè è il SO ad avere lo stato dei processi in corso e non, quindi il PCB è salvato dal SO).
2. Il SO **prepara** l'arrivo del **programma gestore delle interruzioni** caricandolo dalla RAM. Ed avviene il context switch.
3. Il SO invia un **ACK** al **controller delle interruzioni** per notificare l'inizio della gestione.
4. Il SO avvia il programma di gestione e quando finisce avvia il comando **RETINT** (per effettuare il cambio di contesto).
5. Il SO **sceglie** il prossimo processo da avviare.
6. Vengono **caricati** i dati del prossimo processo.
7. Viene **lanciato** il processo.



(a) Interrupt occurs after instruction at location N

(b) Return from interrupt

Per prima cosa quando **carichiamo il programma di gestione** dobbiamo caricare il suo indirizzo in memoria dall'interrupt vector, in questo caso inizia a Y e finisce a Y+L. Lo **stato attuale viene salvato in T-M.** Al ritorno **dall'interruzione vengono caricati da T-M** dal **control stack** di nuovo in CPU con il comando RETINT, nella figura vediamo che nel PC c'è Y+L+1 che non verrà mai eseguita perchè torniamo direttamente a N+1.

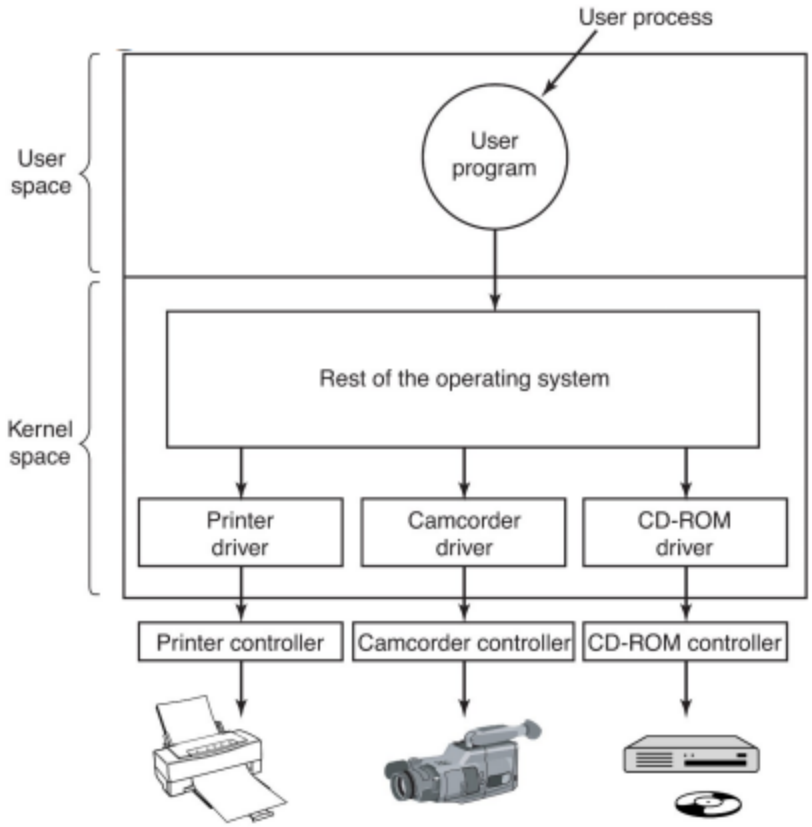
2. Driver dei dispostivi

Il livello del driver dei dispositivi serve per mascherare al livello successivo come vengono controllati i dispositivi. Interagisce con il controller del dispositivo e ha

altre funzioni come l'inizializzazione o da energia al dispositivo.

Tipicamente **ogni driver gestisce un dispositivo**, sono dipendenti dal SO ovvero da Windows a Linux con la stessa periferica cambia il programma.

I **driver sono codici che gestiscono** le **interruzioni del dispositivo** a cui sono **associati, accettano chiamate virtuali di read e write e le trasformano in in comandi a basso livello**. Infatti devono potersi interfacciare direttamente con la periferica e devono quindi essere eseguiti in kernel mode (perchè si vanno a usare i registri).



Come fanno a diventare parte del sistema operativo? Ci sono due modi per integrarli:

- **Compilazione statica** al SO: carichiamo i driver e compiliamo il kernel questo però rende più pesante il caricamento e avremo meno spazio per la RAM e altri processi utente, se cambiamo le periferiche dovremmo ricompilare il kernel che non è proprio il massimo.

- **Compilazione dinamica** al SO: quando installiamo una periferica stiamo caricando sul disco il programma, così quando il BIOS identifica la periferica viene caricato anche il suo programma.

> 💡 **Compilazione statica**: prendiamo direttamente il codice di quella funzione e la inseriamo nel nostro codice avendo così un eseguibile più grande.
>
> **Compilazione dinamica**: prendiamo il codice solo quando leggiamo il codice dall'eseguibile viene associata e caricata la funzione.

Alcuni driver sono caricati in user mode

Quando carichiamo un driver viene caricato il programma di gestione, quindi se viene scritto male va in crash il sistema operativo e quindi se gira in kernel mode fa crashare tutto BSOD (**Blue Screen Of Death**). Ci sono punti di forza nel far girare un driver in user mode: è più semplice da sviluppare e aumenta la stabilità, ma ci sono criticità come l'invio della richiesta di interruzione (PIO e quindi busy waiting).

Struttura generale di un driver

1. **Accetta richieste I/O da livelli sovrastanti**
2. **Controlla la validità dei parametri di input**
    a. Se è valido traduce il parametro astratto in un tipo specifico
    b. Se non valido ritorna errore
3. **Controlla che il dispositivo sia attualmente in uso**
    a. Se è occupato la richiesta viene accodata
    b. Altrimenti esamina lo stato dei registri e poi inserisce i dati
4. **Invio nella giusta sequenza di comandi a basso livello**
    a. Controllo che sia stato accettato l'attuale
    b. Invio di un nuovo comando

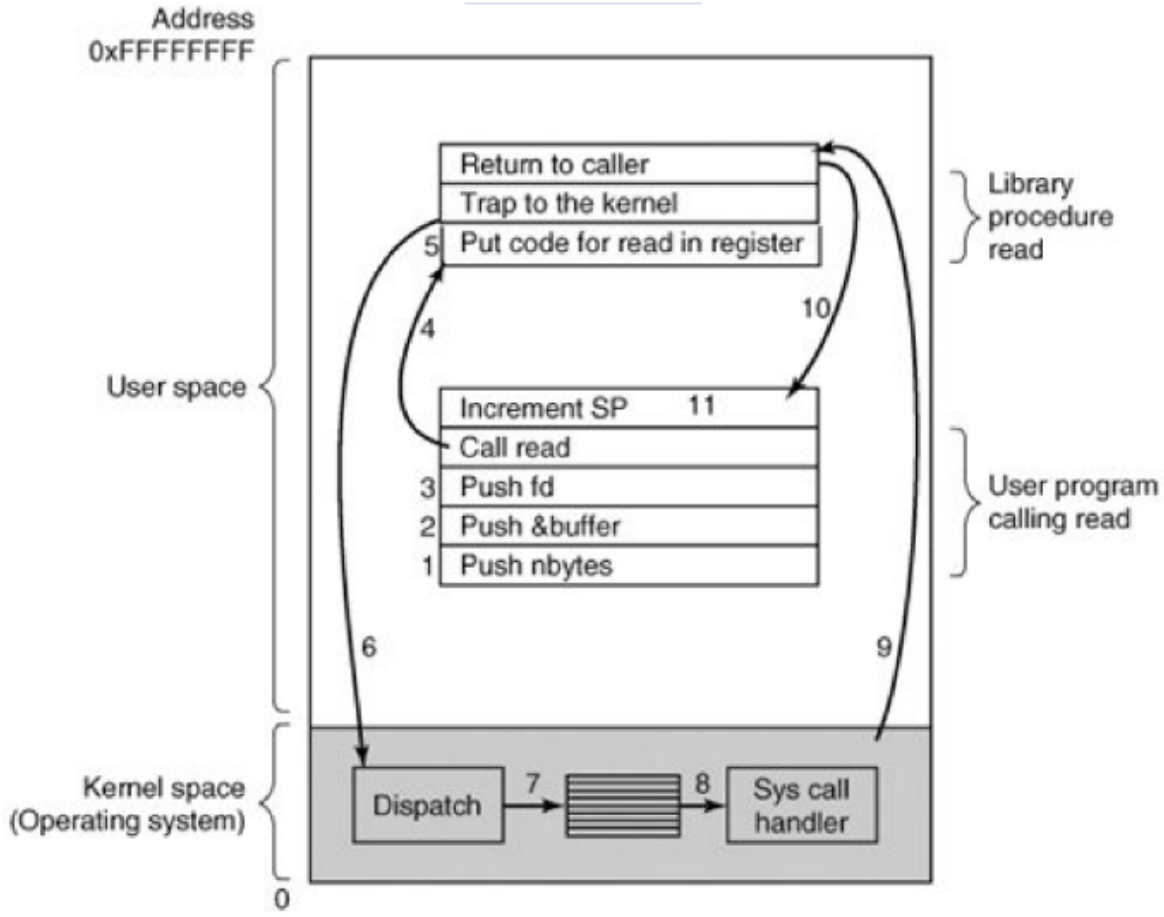
5. **Quando il comando viene eseguito**
    a. Se PIO (Programmed I/O) aspetta il completamento
    b. Se guidato dalle interruzioni si blocca fino a che una interruzione arriva
6. **Finita l'operazione avviene un controllo di errori**
7. **Ritorna lo stato nel caso di errore**
8. **Seleziona il prossimo blocco di dati**

Se avviene una interruzione quando avviene la gestione di un'altra interruzione dobbiamo esser attenti a questa evenienza. Si dice che il driver è **reentrant** quando può essere interrotto nel mezzo dell'esecuzione e poi fatto ripartire senza problemi.

I così detto "**hot-pluggable**" ovvero che un dispositivo può essere inserito e rimosso durante l'esecuzione del SO. Viene letto il contenuto e disponibile runtime.

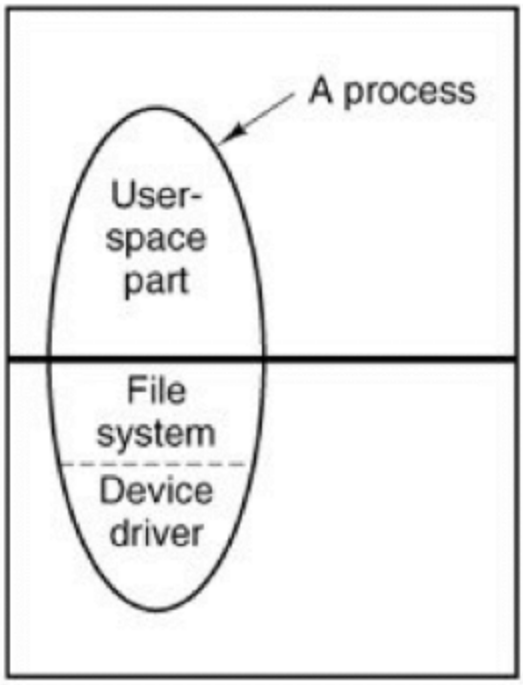
Quando viene inserito il driver della periferica viene caricato in memoria e può gestirla, così come quando viene scollegata il driver dovrà fare in modo che non vi siano errori. Ci deve essere una **gestione dinamica** della periferica.

Come viene effettuata una system call:



1. Viene chiamata la “**read**()” quindi chiamando una funzione dobbiamo **allocare uno stack frame** per la funzione. La read chiama una funzione di libreria che fa il context switch.
2. Viene messo il **numero della system call**, ogni funzione ha il suo numero come se fosse un ID.
3. Viene eseguita una trap verso il kernel, chiaramente **l’hardware gestisce la trap** attraverso il dispatcher che effettua il cambio contesto.
4. Viene gestita la funzione e viene di nuovo caricato lo stato precedente.

Il driver è un pezzo di codice che viene agganciato al programma chiamante che ne ha bisogno.



Ogni processo ha uno **spazio che gira in kernel mode**, ad esempio per il device driver che viene caricato e viene copiata una istanza del codice.

Riassunto sull'I/O e gli Interrupt

L'I/O si divide in due categorie principali: hardware e software. Il primo si verifica quando una periferica fisica comunica con il sistema operativo o viceversa (ad esempio monitor, mouse o tastiera), mentre l'I/O software avviene quando il codice comunica con il sistema operativo o viceversa tramite system calls. In entrambi i casi, è necessario passare al "kernel mode" per la gestione attraverso interrupt vector.
Gli interrupt hardware sono gestiti da appositi driver, che sono porzioni di codice che comunicano con il controller del dispositivo. Quando un dispositivo hardware richiede un'operazione di I/O, il suo controller rileva l'interruzione e la segnala sulla sua linea dedicata. Se l'interrupt controller è libero, può iniziare la gestione. L'identità del dispositivo che ha generato l'interruzione viene determinata attraverso un vettore di interruzioni, che contiene l'indirizzo del codice che gestisce l'interruzione.
Per gli interrupt software, i driver comunicano con il controller del dispositivo per impartire istruzioni. I driver e i controller del dispositivo sono due entità separate:

quando si verifica un'interruzione hardware, essa viene propagata sulla linea dedicata, e il sistema operativo associa un interrupt vector che avvia il programma dell'handler dell'interruzione descritto dal driver.

Visualizzazione di un driver

Sistema operativo didattico xv6.

```
Control Register:
  Address 0x3F6 = 0x08 (0000 1RE0): R=reset, E=0 means "enable interrupt"

Command Block Registers:
  Address 0x1F0 = Data Port
  Address 0x1F1 = Error
  Address 0x1F2 = Sector Count
  Address 0x1F3 = LBA low byte
  Address 0x1F4 = LBA mid byte
  Address 0x1F5 = LBA hi  byte
  Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive
  Address 0x1F7 = Command/status

Status Register (Address 0x1F7):
    7      6    5      4    3    2    1     0
   BUSY  READY FAULT SEEK  DRQ  CORR IDDEX ERROR

Error Register (Address 0x1F1): (check when Status ERROR==1)
    7      6    5      4    3    2    1    0
   BBK    UNC   MC   IDNF  MCR  ABRT T0NF AMNF

   BBK  = Bad Block
   UNC  = Uncorrectable data error
   MC   = Media Changed
   IDNF = ID mark Not Found
   MCR  = Media Change Requested
   ABRT = Command aborted
   T0NF = Track 0 Not Found
   AMNF = Address Mark Not Found
```

Ci sono 4 tipi di registri: **controllo**, **comandi**, **stato** ed **errore**, in questo caso abbiamo una mappatura a porte ovvero con istruzioni IN o OUT.

Il registro di controllo ci consente di controllare il device, il blocco di comandi ci dice cosa deve scrivere o leggere e in quale settore. Il registro di stato ci dice cosa sta facendo il dispositivo ed infine il registro di errore ci dice se ci sono errori o meno.

```
Abbiamo una maschera per lo status register:
[BUSY, READY, FAULT, SEEK, DRQ, CORR, IDDEX, ERROR]
                 ^
Il bit va a 1 quando viene impostato dal controller nel device.
```

Nel command block regsister vediamo come all'indirizzo 0x1F7 possiamo trovare il registro di stato.

- **In**: legge dati da un registro.
- **Out**: scrive i dati su un registro.

Ci sono 3 modalità per 8, 16 o 32 bit si scrittura o lettura. Se abbiamo una "**s**" nel nome della funzione significa che legge sequenze più lunghe di bit.

```
static int ide_wait_ready() {
  while (((int r = inb(0x1f7)) & IDE_BSY) || !(r & IDE_DRDY))
    ;                                  // loop until drive isn't busy
}

static void ide_start_request(struct buf *b) {
  ide_wait_ready();
  outb(0x3f6, 0);                      // generate interrupt
  outb(0x1f2, 1);                      // how many sectors?
  outb(0x1f3, b->sector & 0xff);       // LBA goes here ...
  outb(0x1f4, (b->sector >> 8) & 0xff);    // ... and here
  outb(0x1f5, (b->sector >> 16) & 0xff);  // ... and here!
  outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
  if(b->flags & B_DIRTY){
    outb(0x1f7, IDE_CMD_WRITE);        // this is a WRITE
    outsl(0x1f0, b->data, 512/4);  // transfer data too!
  } else {
    outb(0x1f7, IDE_CMD_READ);         // this is a READ (no data)
  }
}

void ide_rw(struct buf *b) {
  acquire(&ide_lock);
  for (struct buf **pp = &ide_queue; *pp; pp=&(*pp)->qnext)
    ;                                  // walk queue
  *pp = b;                             // add request to end
  if (ide_queue == b)                  // if q is empty
    ide_start_request(b);              // send req to disk
  while ((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
    sleep(b, &ide_lock);               // wait for completion
  release(&ide_lock);
}

void ide_intr() {
  struct buf *b;
  acquire(&ide_lock);
  if (!(b->flags & B_DIRTY) && ide_wait_ready() >= 0)
    insl(0x1f0, b->data, 512/4);    // if READ: get data
  b->flags |= B_VALID;
  b->flags &= ~B_DIRTY;
  wakeup(b);                           // wake waiting process
  if ((ide_queue = b->qnext) != 0) // start next request
    ide_start_request(ide_queue);  // (if one exists)
  release(&ide_lock);
}
```

Questo codice è un esempio di driver, ci sono diverse funzioni che servono per gestire il dispositivo, il controller userà parti del driver così come il SO caricherà in memoria porzioni di codice del driver.

La struttura `buf` è descritta così:

```
struct buf {
    int flags; // Quale operazione va svolta
    uint dev; // Numero del dispositivo su cui leggere o scriver
    uint sector; // Quale settore andiamo a leggere o scrivere
    struct sleeplock ide_lock; // Serve per la mutua esclusione
    uchar data[BSIZE]; // BSIZE è una macro  impostata a 512 byt
    struct buf *prev; // Puntatore al nodo precedente
    struct buf *next; // Puntatore al nodo successivo
    struct buf *qnext; // Cosa richieste in attesa per il disco
}
```

Nella struttura sleeplock vengono usate le funzioni TSL ad esempio per garantire la muta esclusione.

Abbiamo anche gli **spinlock** che generano **busy waiting** mentre la sleeplock manda a dormire il processo.

3. Software indipendente dal dispositivo

Avvio delle funzioni del driver

Il processo che vuole effettuare una richiesta I/O esegue la system call, si passa da user mode a kernel mode. A questo punto il controller esegue la prima funzione ovvero `ide_rw()` che va a fare busy waiting sullo status register.

> 💡 Il valore di **IDE_BSY** è 0x80 (1000 0000) così viene preso il valore del primo bit
> Il valore di
> **IDE_DRDY** è 0x40 (0100 0000) così viene preso il valore del secondo bit

Il valore del device viene dato dal suo **stato**, ad esempio se fosse **BUSY** avremmo **1000 0000** per il suo stato, messo in & con i due registri avremmo while(true || true) e così avremo una attesa attiva. Usare le macro serve perchè quando viene

compilato il codice vengono rimpiazzate la parti di codice con macro con gli effettivi valori.

La funzione `ide_start_request()` serve per inviare richiesto con una struttura dati di tipo `buf` al disco. Vengono scritti i registri con i valori attraverso il protocollo sopra riportato. Quando scriviamo 0 significa che stiamo abilitando le interruzioni (E = 0 'interruzioni abilitiate').

Supponiamo che venga chiesto al driver una operazione di lettura al disco numero 1, iniziamo a inserire nel blocco 0x1F3 i primi 8 bit meno significativi.

Abbiamo il blocco 0111 0101 1011 1110 1101 0001 0101

I primi 8 bit vanno scritti in **0x1F3**: 0001 0101

I successivi 8 bit vanno scritti in **0x1F4**: 1110 1101

I successivi 8 bit vanno scritti in **0x1F5**: 0101 1011

Ed infine abbiamo gli ultimi 4 bit in **0x1F6**: 0111

Il blocco va messo in & con 0xFF per poter ricavare gli ultimi 8 bit ovvero quelli in **0x1F3, porta** dove dovranno essere messi questi dati.

Successivamente dobbiamo fare uno shift di 8 bit verso sinistra per poter fare di nuovo 0xFFx sul blocco di numeri che andranno in **0x1F4,** e poi ancora uno shift di 16 bit a sinistra con maschera 0xFF.

A questo punto rimane l'ultimo pezzo per **0x1F6,** con i 4 bit più significativi sono 1B1D per dire che ci troviamo nel primo disco così da poter unire questi 4 bit con i 4 rimanenti ovvero 0111.

Dobbiamo comporre il restante byte abbiamo solo i LSB ovvero 0111, si ragiona sempre con 8 bit quindi abbiamo 000 0111, lo mettiamo in & con 0xF. 1B1D abbiamo una assegnazione B con 1 e D con 1 perchè è il disco 1.

```
(b->dev & 1) << 4 = 0000 0001 << 4 = 0001 0000
```

Il campo dev contiene il disco su cui vogliamo fare la scrittura o lettura

```
(b->sector >> 24) & 0x0f = 0000 0111
```

Come risultato avremo 1111 0111 dove i più alti sono TOP4LBA e i più bassi invece sono del blocco dati.

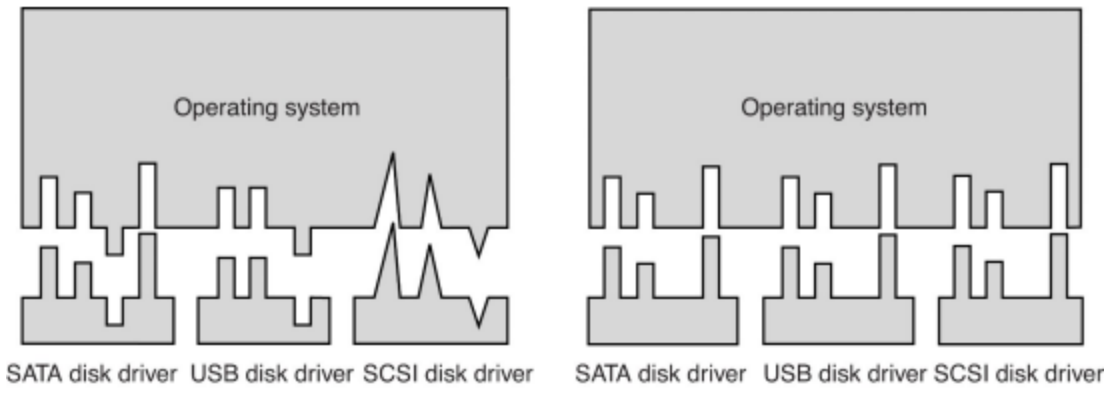
Adesso verifichiamo il **dirty bit** se vale 1 abbiamo una scrittura e quindi scriviamo in 0x1F7 diamo il comando.

I driver come interfaccia per il sistema operativo

I driver non possono essere gestiti in maniera personalizzata a seconda del dispositivo altrimenti il SO dovrebbe gestire troppi fattori, per questo si è creato un'interfaccia che fornisse gli stessi blocchi di funzionamento per tutti:

Uniform interfacing for device drivers
Buffering
Error reporting
Allocating and releasing dedicated devices
Providing a device-independent block size

Così facendo i driver forniscono delle API per il SO con le quali possono collaborare e qualunque driver se creato con questa interfaccia potrà comunicare con il SO:



In Unix ci sono dei file appositi che ci danno il nome del file con l'interfaccia di quel dispositivo vediamo un esempio qui sotto:

```
crw-rw-rw-. 1 root root      1,   3 Feb 28 13:56 null
…
brw-rw----. 1 root disk      8,   0 Feb 28 13:56 sda
brw-rw----. 1 root disk      8,   1 Feb 28 13:56 sda1
brw-rw----. 1 root disk      8,   2 Feb 28 13:56 sda2
brw-rw----. 1 root disk      8,   3 Feb 28 13:56 sda3
…
crw-rw-rw-. 1 root tty       5,   0 Feb 28 13:57 tty
crw--w----. 1 root tty       4,   0 Feb 28 13:56 tty0
crw--w----. 1 root tty       4,   1 Feb 28 13:56 tty1
…
crw-rw-rw-. 1 root root      1,   5 Feb 28 13:56 zero
```

Come abbiamo detto ci sono tabelle con nomi dei file per i vari dispositivi, abbiamo in **rosso** se sono dispositivi a **caratteri** o **blocchi**, in **blu** il **major number** e in verde il **minor number**. **L'arancione** identifica il **nome in stringa**. I file contengono l'interfaccia (ovvero il major number e minor number) per poter mettere in **comunicazione** il **sistema operativo** con i **driver del dispositivo**. L'interfaccia del driver è fornita dall'SDK di sviluppo dei driver chiaramente, per la sua implementazione abbiamo:

```
/* Describe the driver */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("John Doe");
MODULE_DESCRIPTION("My simple driver");

#define DEVICE_NAME “foobar”
static int major_num; /* Holds the device major number */
static ssize_t my_drv_read(struct file*, char __user*, size_t, l
static ssize_t my_drv_write(struct file*, const char __user*, si

static struct file_operations my_drv_fops = {
    .owner = THIS_MODULE,
    .read = my_drv_read,
```

```
    .write = my_drv_write
};

static int __init my_drv_init(void) {
    /* 0 -> dynamically get major number */
    major_num = register_chrdev(0, DEVICE_NAME, &my_drv_fops);
    printk(KERN_INFO "My driver registered");
    return 0;
}

static void __exit my_drv_exit(void) {
    unregister_chrdev(major_num, DEVICE_NAME)
    printk(KERN_INFO "My driver unregistered");
}

/* Implementation of my_drv_read, my_drv_write, … */
module_init(my_drv_init);
module_exit(my_drv_exit);
```

La prima funzione server per dire che quando viene caricato il driver dobbiamo chiamare **my_drv_init** e quando viene smontato dobbiamo chiamare **my_drv_exit.** In inizializzazione viene creato il file che vediamo nella tabella sopra riportato. Il nome è passato nel codice in questo caso “foobar” mentre il **major number** viene assegnato in fase di creazione, **se viene passato 0** sarà il **programma del kernel a scegliere il numero**. Quando il driver viene tolto dalla memoria viene rimosso anche il file speciale.

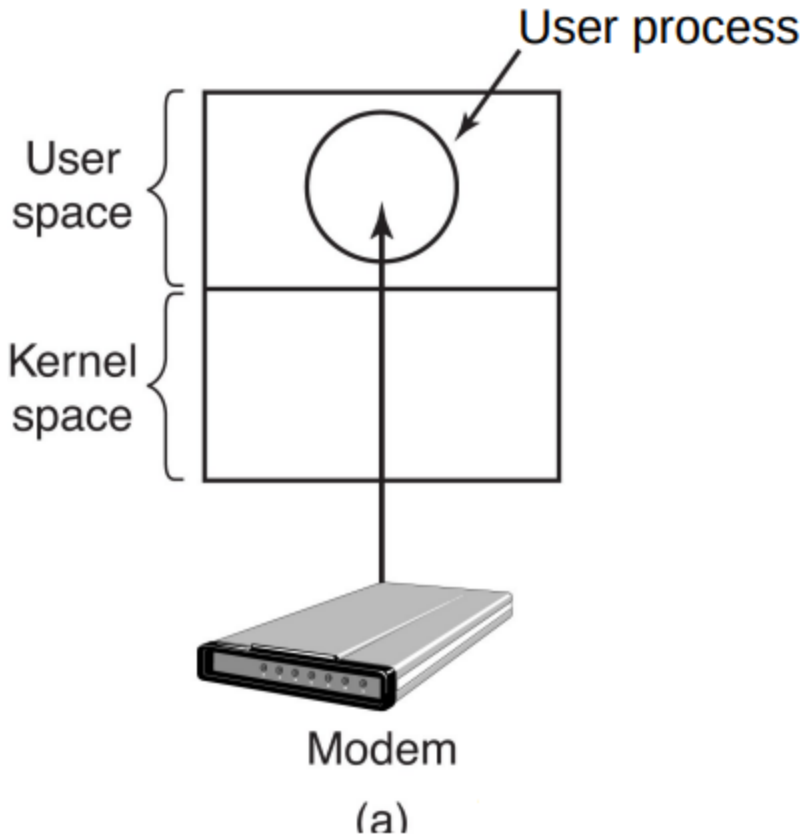
Per sapere quante istanze abbiamo di un dispositivo ci sono degli script lanciati in fasi di avvio e servono per assegnare il minor number.

Quando il sistema operativo vuole parlare con il dispositivo, viene invocato il file in `/dev/nome` e grazie a questo sappiamo quale file driver `nome.ko` andremo a interpellare.

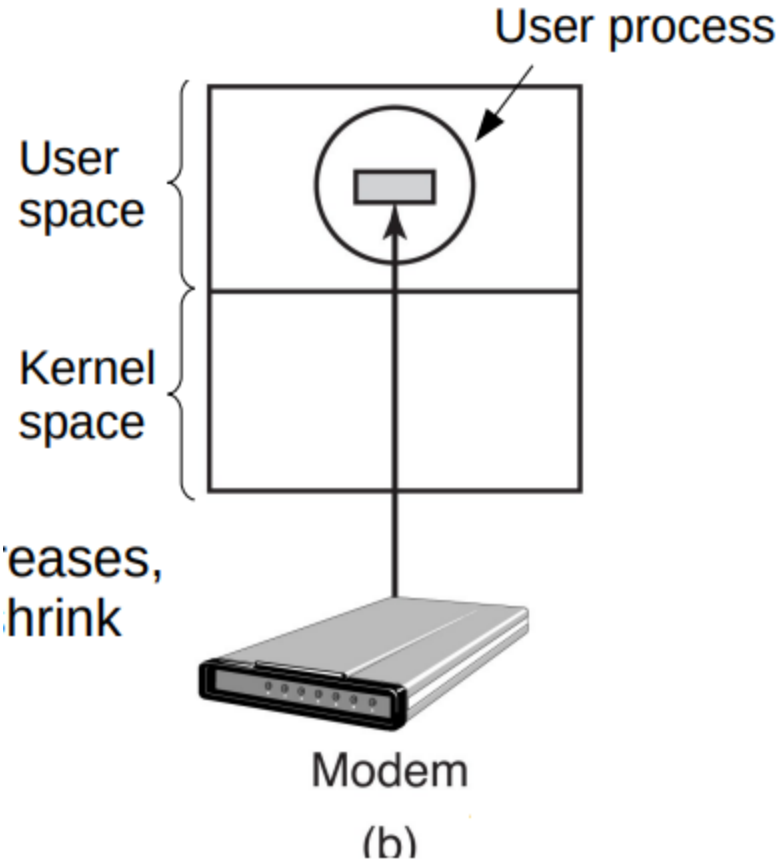
Buffering

Come possiamo gestire l'arrivo dei dati con la loro spedizione? Vediamolo in un problema semplice.

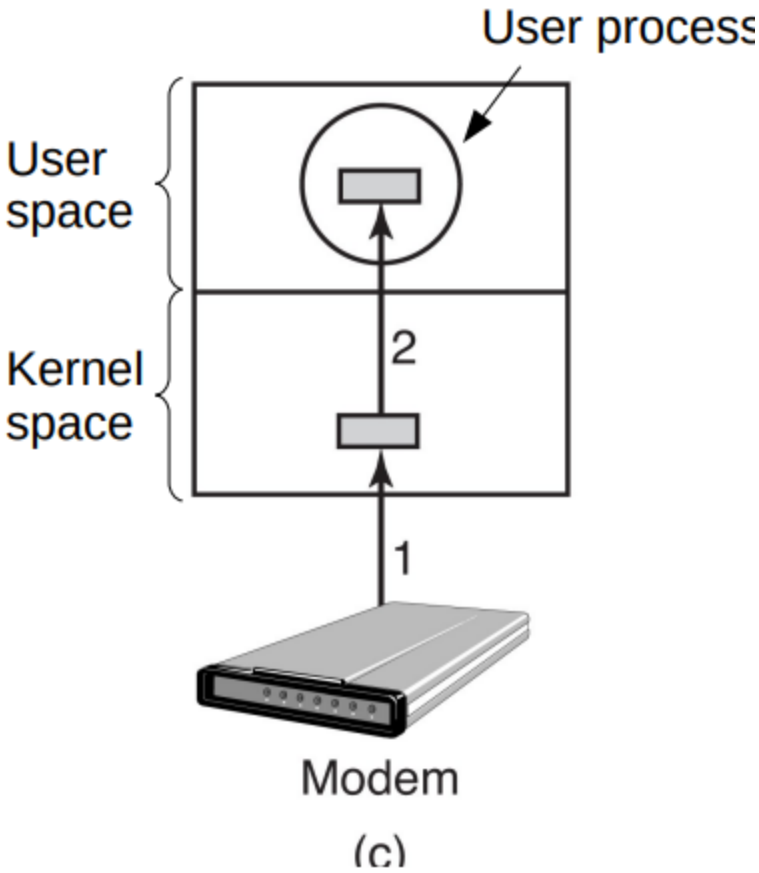
- **Leggere** dati da un modem ADSL, **senza buffering**, se leggessimo un carattere alla volta, avremmo una interruzione per gestire questo arrivo, ogni interruzione ha 2 context switch, ovvero per chiamare l'interrupt handler e riportare tutto alla normalità, quindi bisogna cambiare approccio.



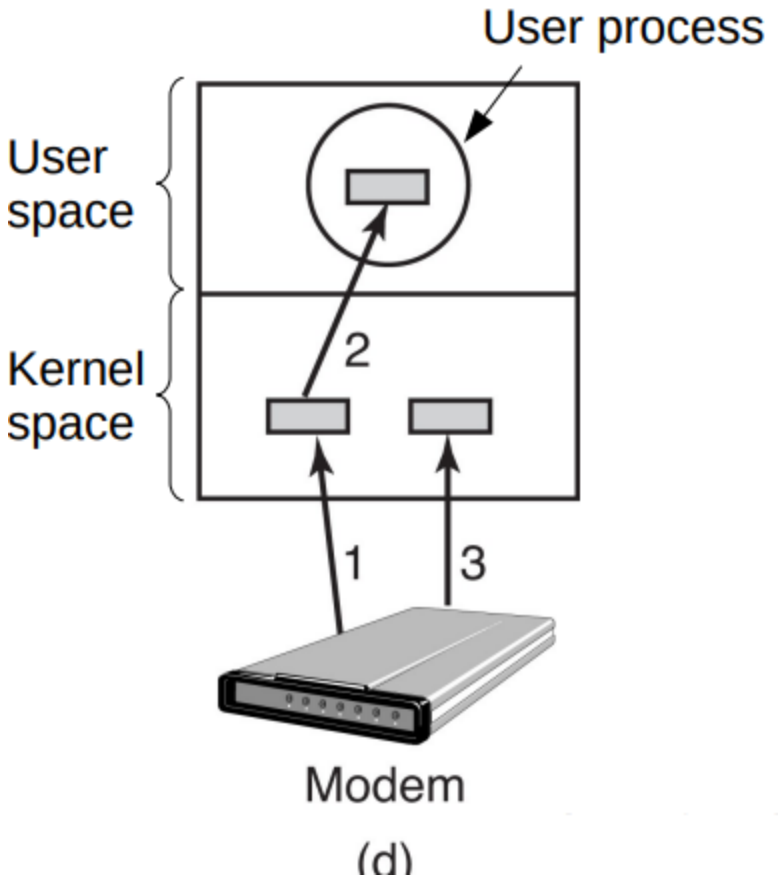
Con il **buffering** possiamo leggere blocchi di dati, come possiamo riempire un buffer mentre i dati stanno arrivando? Attraverso il **bloccaggio delle pagine** in memoria centrale per non poter esser **swappate** in memoria fin quando non sono piene.



Per risolvere questo problema in modo che **non siano scambiate dal gestore delle pagine** nel mentre che sono riempite, possiamo spostare le **pagine nel kernel** (non possono essere swappate per definizione), una volta che sono piene possiamo metterle in memoria:

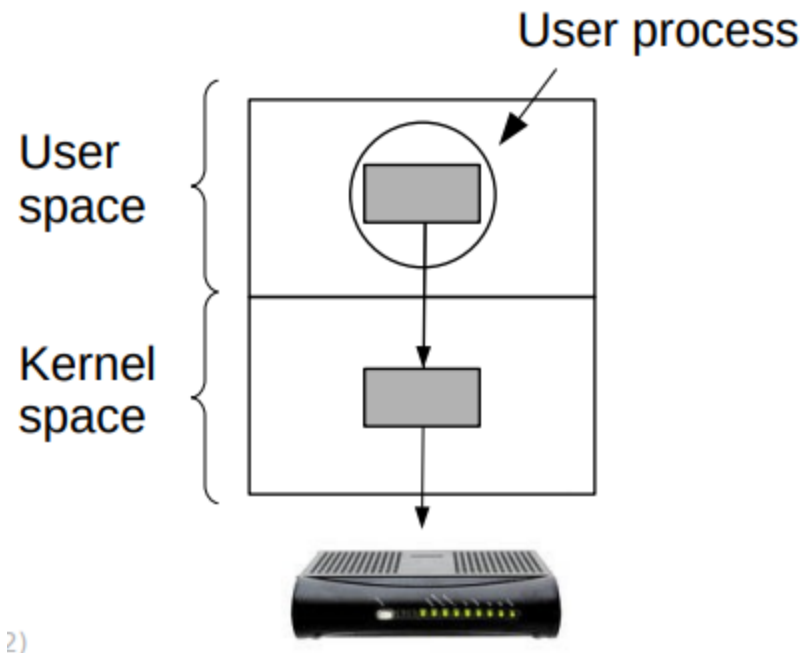


Se però arrivano altri dati nel mentre che stiamo copiando la pagina in memoria centrale, come possiamo gestire questi dati? Semplice attraverso altri buffer.



Chiamiamo **buffer circolare** quando abbiamo k pool di buffer da usare.

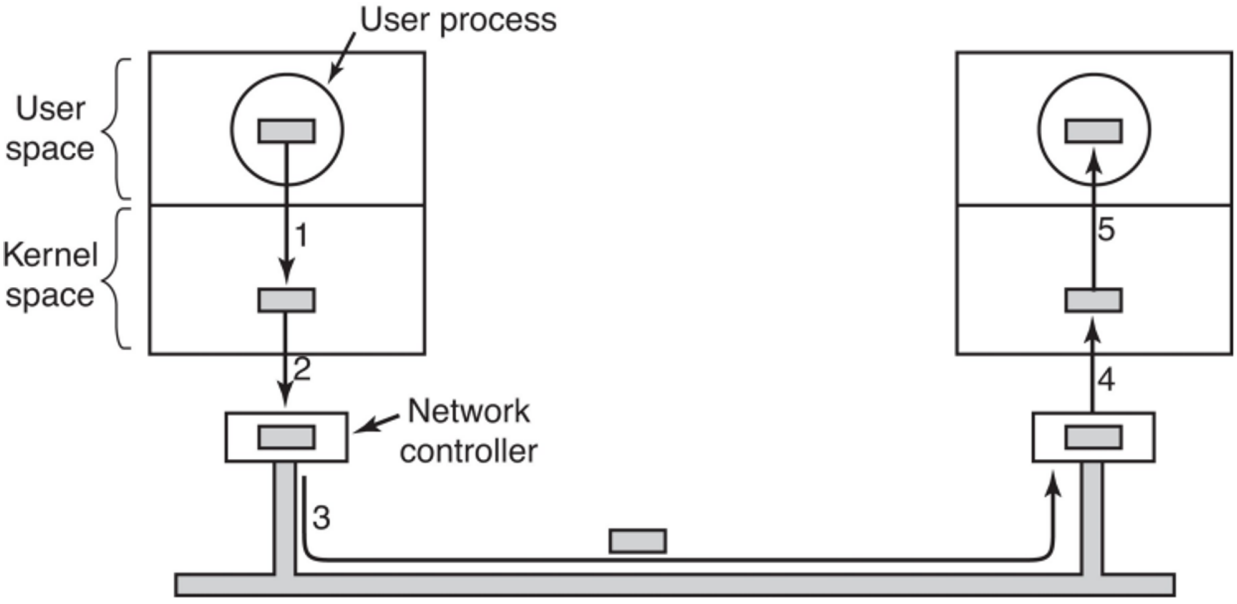
- Dobbiamo anche **scrivere** sul modem, come possiamo fare? Nello stesso modo della lettura possiamo usare più buffer per poter scrivere più dati senza perderne.



2)

Il buffering è davvero la soluzione?

Sì ma in parte, se il buffering viene abusato allora avremo una perdita di performance, proprio per questo deve essere usato nei momenti giusti.



1. **User**-space buffer → **Kernel**-space buffer
2. **Kernel**-space buffer → **Network**-space buffer

3. **Network**-space buffer → **Network**-space buffer
4. **Network**-space buffer → Kernel-space buffer
5. **Kernel**-space buffer → **User**-space buffer

Riportare gli errori

Gli errori sono molto comuni nel contesto degli I/O quando ci sono devono essere gestiti al meglio dal SO, ma principalmente sono i driver a dover fornire un modo per gestire gli errori.

Ci sono diversi tipi di errori che possono verificarsi:

- **Errori di programmazione**: quando il chiamante chiede una operazione invalida (scrivere dati su una tastiera) oppure fornire parametri non corretti. Solitamente viene fornito un codice di errore ad esempio nella `read()` viene fornito -1 quando si verifica un errore.
- **Errori di I/O:** quando proviamo a scrivere su una porzione di memoria danneggiata o proviamo a usare una periferica che è spenta. Solitamente ci si aspetta che sia il driver a gestire l'errore.

Possiamo quindi dire che ci sono 3 modi di riportare gli errori:

- **Leggere l'errore in un programma interattivo:** chiede all'utente cosa fare.
- **Leggere l'errore in un programma batch:** il programma fallisce senza poter far nulla.
- **Errore critico**: mostra un errore e termina

Allocazione e rilascio dei dispositivi

Il sistema operativo deve gestire i dispositivi/periferiche, deve controllare se il dispositivo è libero o meno ad esempio se avessimo due stampe allo stesso momento sulla stessa stampante, avremmo un output ovvero molti fogli mischiati.

- **Approccio 1**: il processo utente prova ad aprire il device file, se non è disponibile la chiamata fallisce. Problema di deadlock nel caso in cui un programma occupi il dispositivo per sempre.

- **Approccio 2:** il SO ha un meccanismo per il quale si gestisce la coda di richieste nel caso in cui il dispositivo non sia disponibile, è senza deadlock perchè è il SO che gestisce il dispositivo e quindi questa risorsa.

Grandezza dei blocchi uniforme

Alcuni dispositivi potrebbero avere una dimensione di blocchi diverse, questo livello serve per fornire una dimensione prefissata dai vari dispositivi del SO.

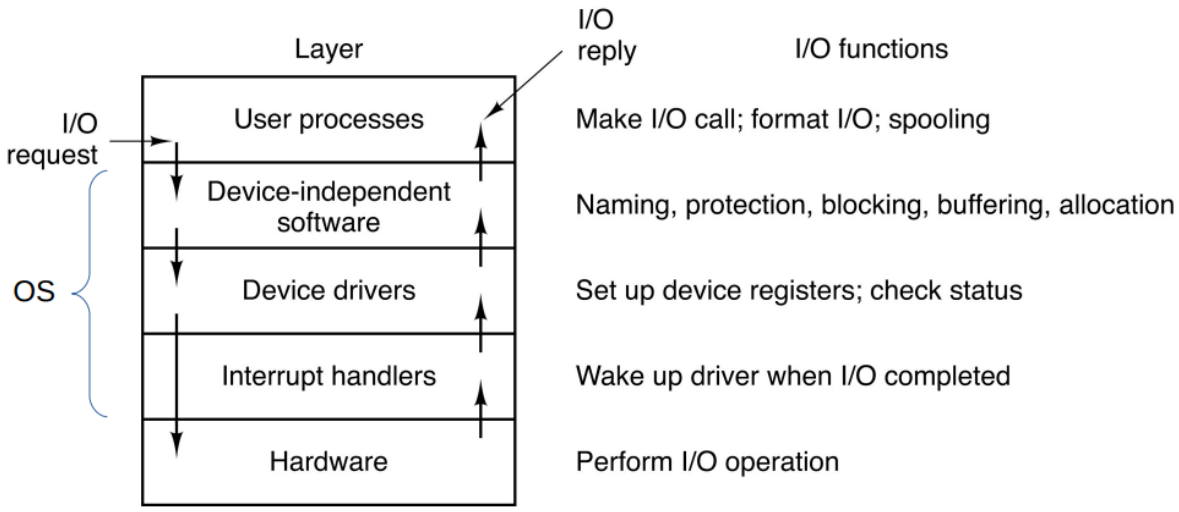
I/O Software: User-space I/O

La maggior parte degli I/O software è compreso nello spazio di esecuzione del kernel, ma una piccola porzione è riservata a **librerie di sistema** e interi programmi che girano in user space.

Le librerie di sistema sono collegate allo spazio utente, possiamo accedere a **system calls attraverso funzioni** fornite dalla **libreria di sistema**.

Sono spesso associati con **spooling system (**Simultaneous Peripheral Operation On-Line**)** ovvero un programma chiamato **spooler** che è avviato come un daemon (programma che gira in background) con una directory ovvero la spooling directory che è usata per gestire input e output in maniera più efficiente.

In sintesi, lo spooling migliora **l'efficienza**, la **reattività** e la **gestione delle risorse** del sistema, rendendolo una scelta preferibile rispetto all'approccio senza spooling in molti contesti.



Nota bene: nella discesa, la freccia salta gli interrupt handlers, proprio perchè i driver comunicano con i controller hardware e saranno poi i controller hardware a parlare con l'interrupt handler che comprende tutti i programmi di gestione delle interruzioni ovvero gli interrupt handlers.