Sistemas de Operação / Fundamentos de Sistemas Operativos

I/O

Artur Pereira <artur@ua.pt>

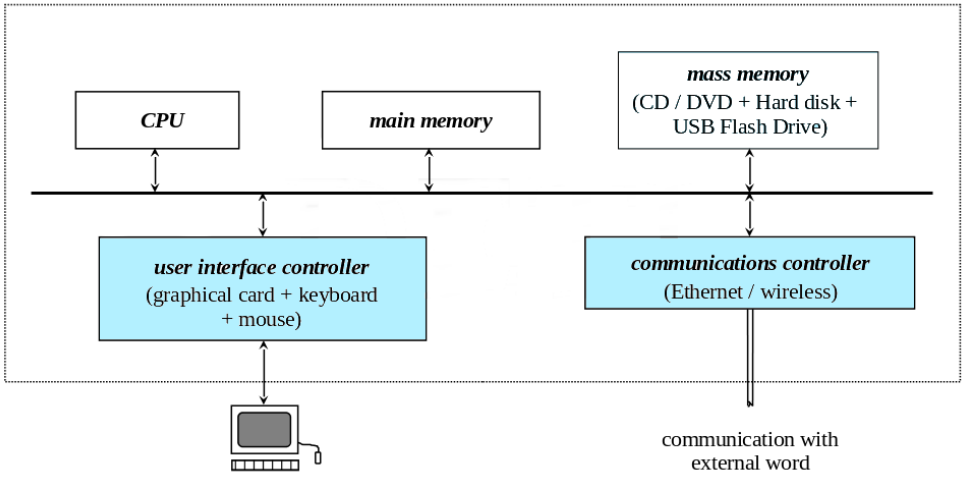
DETI / Universidade de Aveiro

Outline

1. Overview
2. Device controllers
3. I/O programming
4. Types of access

Overview
The I/O components

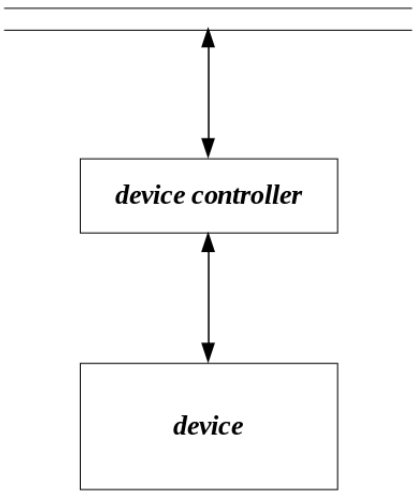
- Simple view of a computational system, highlighting the I/O components



Overview
Role of the operating system

- Two distinct perspectives are usually considered for the role played by the operating system in managing the input / output devices of a computer system
  - user perspective – providing the application developer with a device interface (API) that is conceptually simple, reasonably uniform, and as much as possible independent of the specific device
  - system perspective – isolating the different devices from direct access by user processes by introducing a functional layer that directly controls the devices
    - send commands, transfer data, handle interrupts, and handle error conditions

Device controllers
Device and device controller

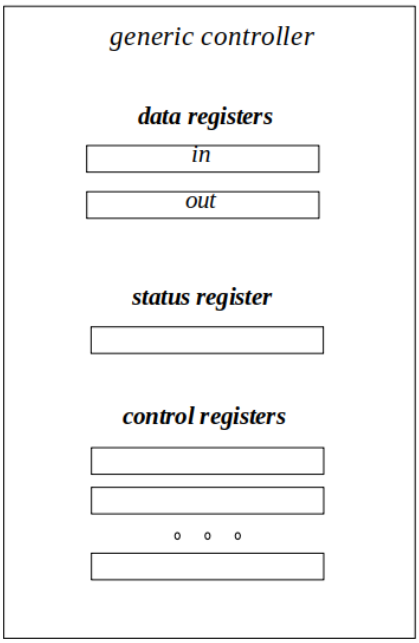


- There are 2 different components to consider
  - The device itself – physical system (electromechanical, optical-mechanical, ...), which stores information and converts it from, or to, an externally accessible form
  - The device controller – electronic circuit, more or less complex, part of the computational system, which works as an interface with the device
- From the point of view of the operating system, the device controller is the only relevant component
- Nowadays, controllers are very versatile, minimizing the role of the operating system in its management (programming)

Device controllers
Types of devices

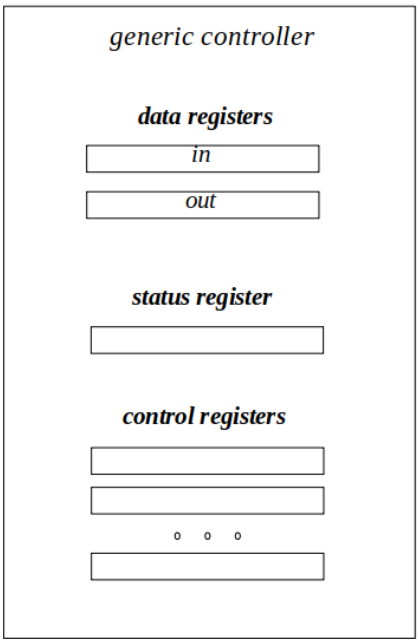
- In terms of transferring information, I/O devices fall into two broad categories
  - character-type devices – the transfer of information is based on a stream of bytes, whose length can be variable
  - block-type devices – the transfer of information is based on a constant and pre-defined number of bytes, the block, typically with a value equal to a power of 2 between 512 and 16K
- The way the transfer is done depends on the bus used
- The rate of transfer depends on the type of device
  - can vary from tens of bytes (keyboard, for example) to thousands of megabytes (SATA or USB3 disk)

Device controllers
Internals



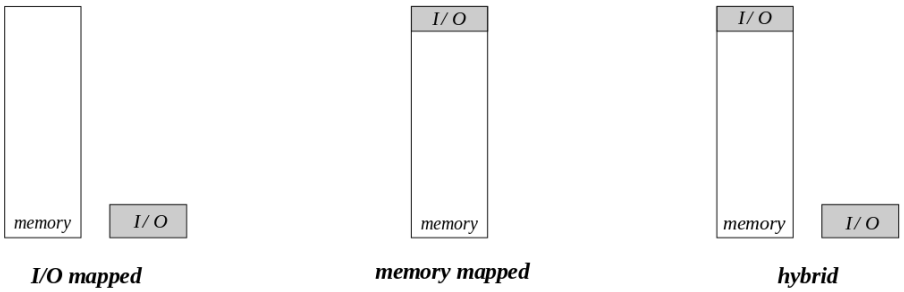
- A generic controller, from a programming point of view, can be seen as set of registers:
  - control registers – playing different functions
    - to configure the device
    - to define the type of interaction with the processor (polled I/O, interrupt-driven I/O or DMA-based I/O)
    - in complex controllers, to execute a command
  - a status register – representing the internal state of the device
    - to indicate the success of the last operation
    - to indicate the failure and erros of the last operation
    - to indicate it is ready to receive a new command

Device controllers
Internals (2)



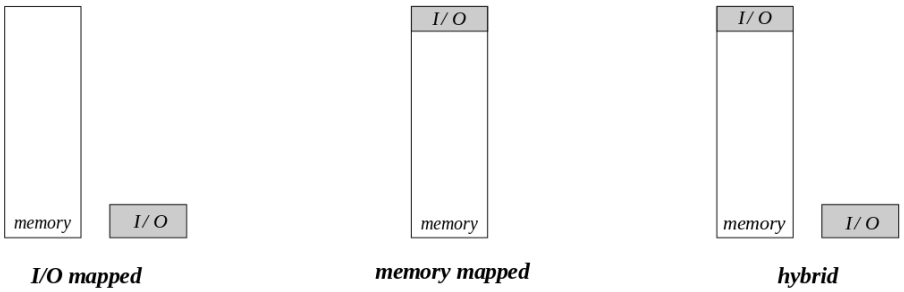
- data registers – used for the communication itself
  - In character-type devices, the write and read commands are implicit
    - a value written in the out register is sent to the device
    - a value received from the device is put in the in register
  - In block-type devices, the transfer starts based on an explicit command
    - the data register is in general unique, in-out, and the direction of the transfer depends on the command given

Device controllers
Address modes



- There are 3 different possible ways to access the internal registers of a controller
  - I/O-mapped – controllers are mapped in a specific I/O address space
    - registers are accessed through specific intructions (in e out)
  - memory-mapped I/O – controllers are mapped in the memory address space
    - registers are accessed through the memory access intructions (load e store)
  - hybrid – controllers are mapped in a specific I/O address space, but data buffers are mapped in the memory address space to facilitate communication

Device controllers
Address modes (2)



- The Intel Pentium has an I/O address space of 64 KB
  - Some computer systems, based on Pentium, use this space to address controllers
  - But, a region in memory, between addresses 640 KB e 1 MB, is also reserved to implement data buffers for the devices

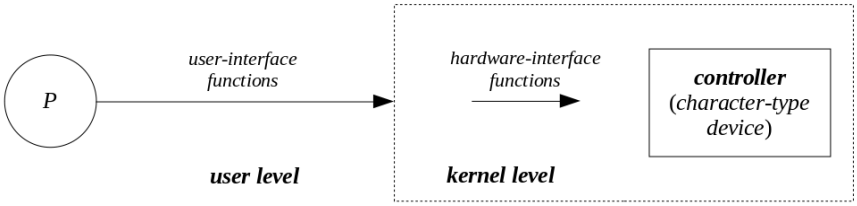
I/O programming
Objectives

- The environment provided by the operating system for the communication with the I/O devices should:
  - be independent of the device specifics
    - devices must be seen in a generic way
    - I/O redirecting, for example, should be possible in a natural way
  - support a uniform naming mechanism
    - device names must consist of strings of characters without any particular meaning
  - decouple devices from user processes
    - vast majority of I/O devices work in an asynchronous manner – data transfers to and from main memory are triggered by interrupts
    - from the user's perspective, however, it is simpler to design communication in a synchronous way – the process blocks until conditions are met for communication to take place

I/O programming
Objectives (2)

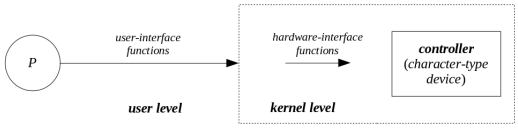
- The environment provided by the operating system for the communication with the I/O devices should (cont.):
  - manage access to preemptable and non-preemptable devices in a uniforme way
    - communication with preemptible devices can be shared by multiple users simultaneously
    - communication with non-preemptible devices must take place in a mutual exclusion, or dedicated regime
    - the operating system therefore has to identify the different situations and ensure proper coordination
  - perform error management in an integrated manner
    - the detection of errors must be carried out as close to the device as possible in order to allow its [possible] recovery in a transparent way
    - the general policy should be to only report the error to the upper layer if the lower layer cannot handle it

Access types
Polled I/O



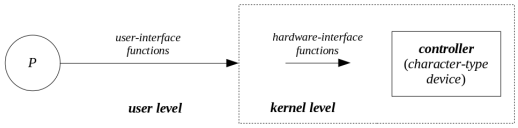
- In a polled-I/O approach there is no decoupling
  - It is the user process that is directly responsible for the communication
- Device communication routines are system calls that directly implement hardware access
- The simplest solution, but little efficient
  - The user processor enters a busy waiting, waiting for the completion of the operation

Access types
Polled I/O (2)



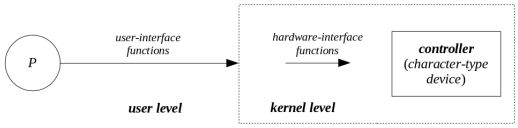

```
/* access routine; assumes a character-type device */

void control(unsigned short add, unsigned char prog []);

#define  RXRDY      ...     /* there are data to be read */
#define  TXRDY      ...     /* transmitter register is empty */
#define  ERROR      ...     /* error status */

unsigned char status(unsigned short add);

unsigned char in(unsigned short add);

void out(unsigned short add, unsigned char val);
```

Access types
Polled I/O (3)



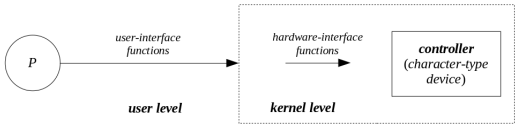

```
/* possible routines for interaction with the device
   system calls - running at kernel level
   It is assumed that the communication channel was established already */

/**
  \brief read N bytes
  \param dd   --- device descriptor, which represents the communication channel
  \param N    --- number of bytes to be read
  \param buff --- pointer to storage area
  \return 0 on success; -1 on error (errno is set accordingly)
*/
int readNBytes (int dd, int N, unsigned char buff[]);

/**
  \brief write N bytes
  \param dd   --- device descriptor, which represents the communication channel
  \param N    --- number of bytes to be written
  \param buff --- pointer to storage area
  \return 0 on success; -1 on error (errno is set accordingly)
*/
int writeNBytes (int dd, int N, unsigned char buff[]);
```


Access types
Polled I/O (4)



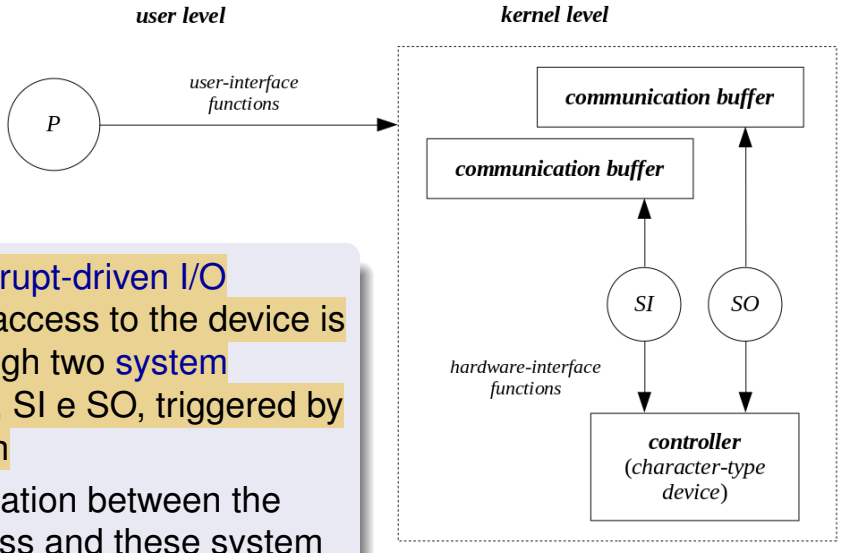

```
/**
  \param dd   --- device descriptor, which represents the communication channel
  \param N    --- number of bytes to be read
  \param buff --- pointer to storage area
  \return 0 on success; -1 on error (errno is set accordingly)
*/
int readNBytes(int dd, int N, unsigned char buff[])
{
    int add = getAdd(dd);
    for (int n = 0; n < N; n++)
    {
        int stat;
        do
        {
            stat = status(add);
            if ((stat & ERROR) != 0)
                /* error handling */
        } while ((stat & RXRDY) == 0);
        buff[n] = in(add);
    }
    return 0;
}
```

Access types
Polled I/O (5)



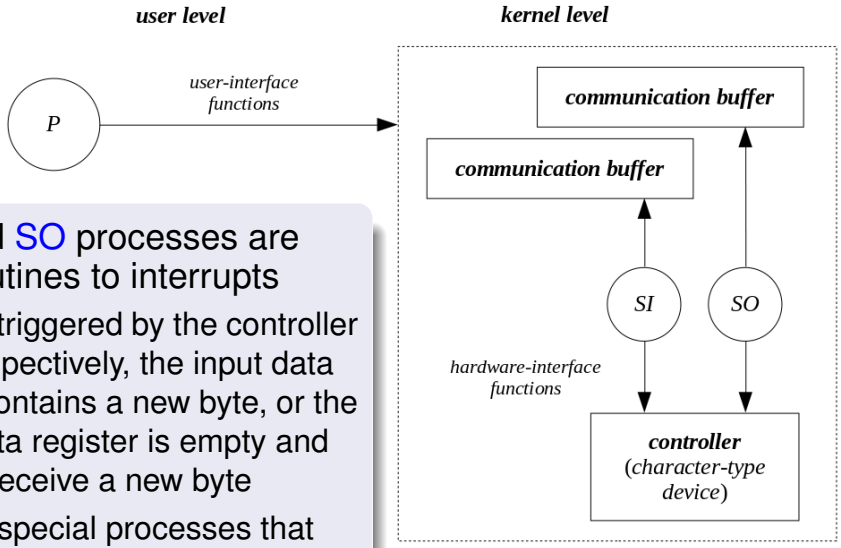

```
/**
  \param dd   --- device descriptor, which represents the communication channel
  \param N    --- number of bytes to be written
  \param buff --- pointer to storage area
  \return 0 on success; -1 on error (errno is set accordingly)
*/
int writeNBytes(int dd, int N, unsigned char buff [])
{
    int add = getAdd(dd);
    for (int n = 0; n < N; n++)
    {
        int stat;
        do
        {
            stat = status(add);
            if ((stat & ERROR) != 0)
                /* error handling */
        } while ((stat & TXRDY) == 0);
        out(add, buff[n]);
    }
    return 0;
}
```


Access types
Interrupt-driven I/O



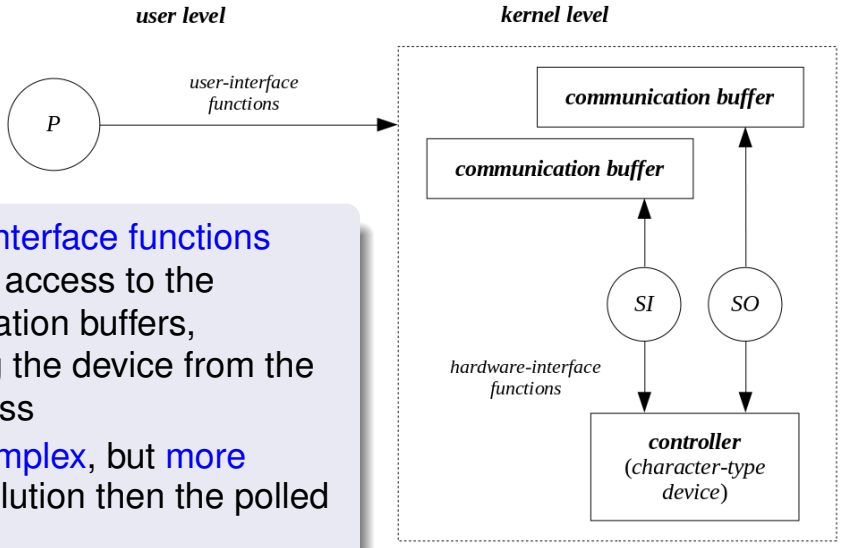
- In the interrupt-driven I/O approach access to the device is done through two system processes, SI e SO, triggered by interruption
- Communication between the user process and these system processes is done through two communication buffers

Access types
Interrupt-driven I/O (2)



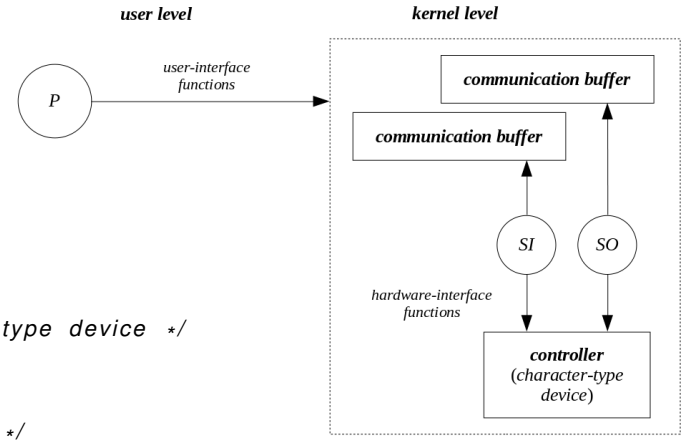
- The SI and SO processes are service routines to interrupts
  - They are triggered by the controller when, respectively, the input data register contains a new byte, or the output data register is empty and ready to receive a new byte
  - They are special processes that run until exhaustion
  - They only lose control of the processor if another, higher priority process is scheduled to run

Access types
Interrupt-driven I/O (3)



- The user-interface functions implement access to the communication buffers, decoupling the device from the user process
- A more complex, but more efficient solution then the polled case
  - User processe blocks, freeing the processor, if conditions for transferring data do not meet

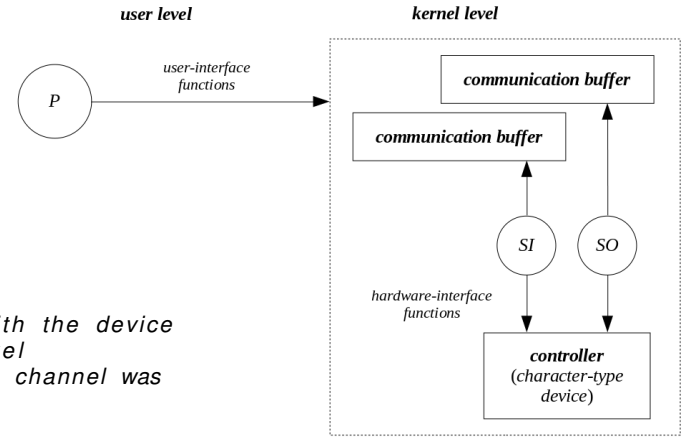
Access types
Interrupt-driven I/O (4)




```
/* access routine; assumes a character-type device */

typedef struct COM_BUFF
{
    FIFO fifo;          /* storage area */
    SEMAPHORE wait;     /* user process' blocking semaphore */
    bool noMoreInt;     /* end of interruptions (only for output) */
    int errnumb;        /* error status */
} COM_BUFF;
```

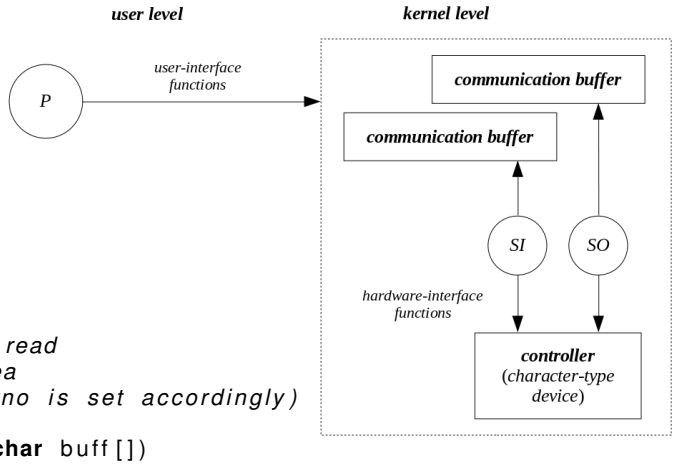
- in an entry buffer, the semaphore is initialized at 0
  - meaning that the FIFO is empty, so no data can be read
- in an output buffer, the semaphore is initialized at N, the size of the `fifo`
  - meaning that the FIFO is empty, so N bytes can be put there
- flag `noMoreInt` signals the need to prime the controller output data register so that interrupts are generated again
  - initialized to true

Access types
Interrupt-driven I/O (5)



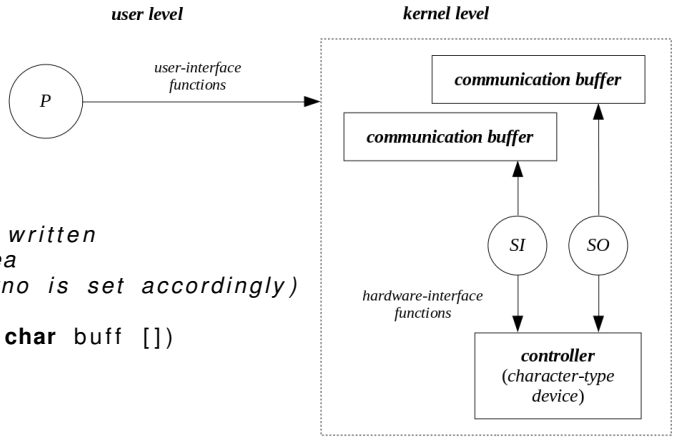

```
/* possible routines for interaction with the device
   system calls - running at kernel level
   It is assumed that the communication channel was
   established already
*/

/**
  \brief read N bytes
  \param dd   --- device descriptor, which represents the communication channel
  \param N    --- number of bytes to be read
  \param buff --- pointer to storage area
  \return 0 on success; -1 on error (errno is set accordingly)
*/
int readNBytes (int dd, int N, unsigned char buff[]);

/**
  \brief write N bytes
  \param dd   --- device descriptor, which represents the communication channel
  \param N    --- number of bytes to be written
  \param buff --- pointer to storage area
  \return 0 on success; -1 on error (errno is set accordingly)
*/
int writeNBytes (int dd, int N, unsigned char buff[]);
```

Access types
Interrupt-driven I/O (6)



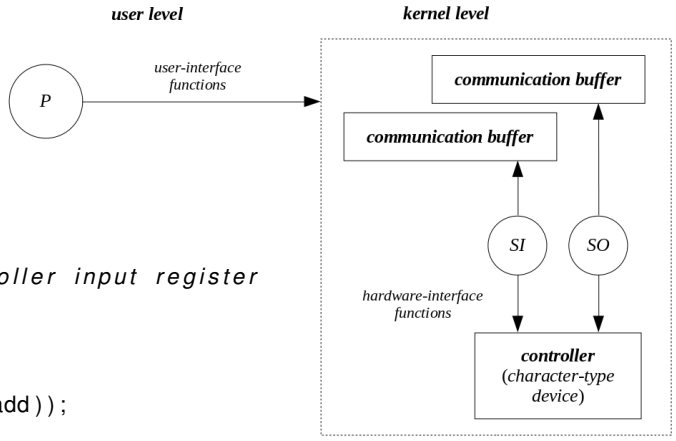

```
/**
  \brief read N bytes
  \param dd   --- device descriptor
  \param N    --- number of bytes to be read
  \param buff --- pointer to storage area
  \return 0 on success; -1 on error (errno is set accordingly)
*/
int readNBytes(int dd, int N, unsigned char buff[])
{
    COMM_BUF * inbuf = getBuff(dd);

    for (int n = 0; n < N; n++)
    {
        if (inbuf->errNumb != 0)
            /* error handling */

        sem_down(inbuf->wait);

        disable_interrupts();
        buff[n] = fifoOut(inbuf->fifo);
        enable_interrupts();
    }
    return 0;
}
```


Access types
Interrupt-driven I/O (7)



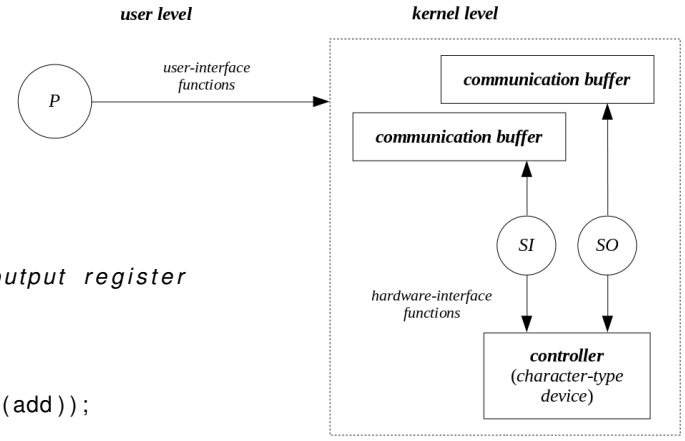

```
/**
  \brief write N bytes
  \param dd   --- device descriptor
  \param N    --- number of bytes to be written
  \param buff --- pointer to storage area
  \return 0 on success; -1 on error (errno is set accordingly)
*/
int writeNBytes(int dd, int N, unsigned char buff [])
{
    COMM_BUF * outbuf = getBuff(dd);

    for (int n = 0; n < N; n++)
    {
        if (outbuf->errNumb != 0)
            /* error handling */

        sem_down(outbuf->wait);

        disable_interrupts();
        fifoIn(outbuf->fifo, buff[n]);
        if (outbuf->noMoreInt)
        {
            writeDReg(getAdd(dd));
            Outbuf->noMoreInt = false;
        }
        enable_interrupts();
    }
    return 0;
}
```

Access types
Interrupt-driven I/O (8)



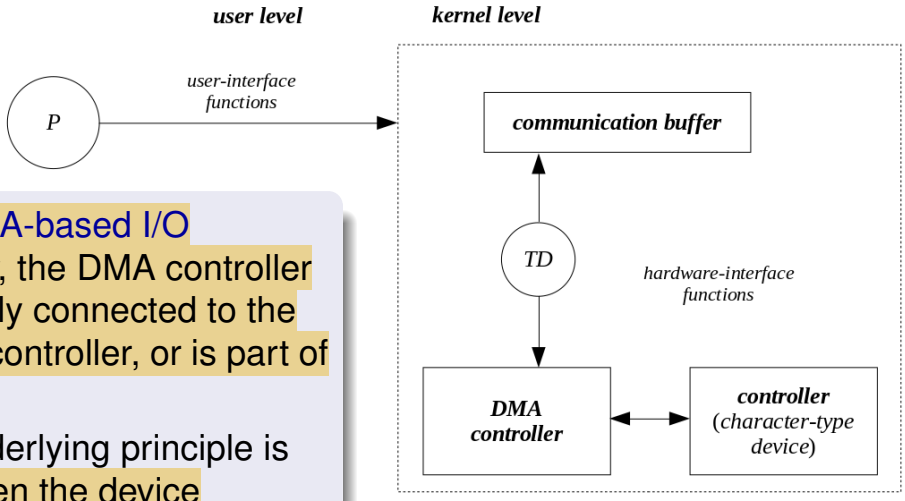

```
/** SI process -
 *   reading of one byte from the controller input register
 *   add --- controller address
 */
void readDReg(unsigned short add)
{
    COM_BUFF * inbuf = getBuff(getDesc(add));

    int stat = status(add);                /* read the status register */
    if ((stat & ERROR) == ERROR)
       inBuff->errnumb = stat & ERROR;     /* set error status */
    if ((stat & RXRDY) == RXRDY)           /* there are data to be read */
    {
        char val = in(add);
        if ((stat & ERROR) != ERROR)
        {
            if (not fifo_full(inbuf->fifo))
            {
                fifo_in(inbuf->fifo, val);     /* store byte in buffer */
                sem_up(inbuf->wait);           /* signal there is data in buffer */
            }
            else
                inbuf->errNumb = OVERRUN;      /* overrun error */
        }
    }
}
```


Access types
Interrupt-driven I/O (9)



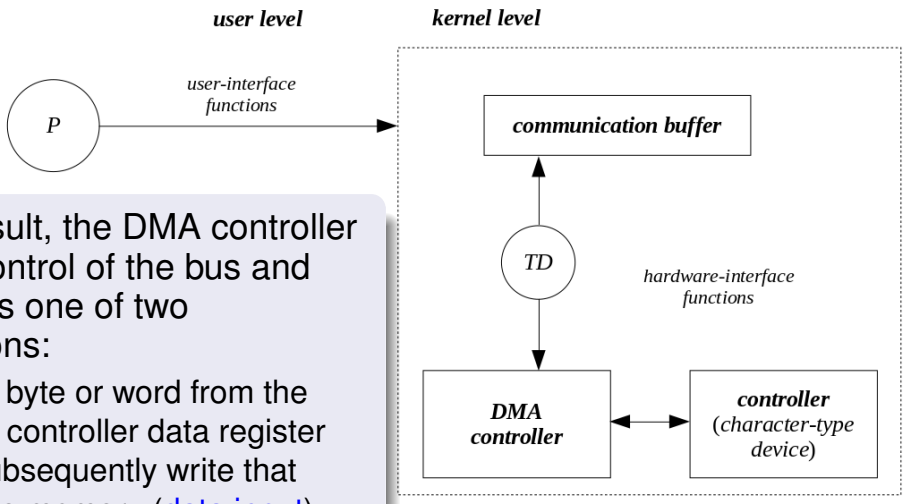

```
/* SO process -
 *   write one byte to the controller output register
 *   add --- controller address
 */
void writeDReg(unsigned short add)
{
    COM_BUFF * outbuf = getBuff(getDesc(add));

    int stat = status(add);                /* read status register */
    if ((stat & ERROR) == ERROR)
       outbuf->errnumb = stat & ERROR;     /* set error status */
    if ((stat & TXRDY) == TXRDY)           /* can write ? */
    {
        if (not fifo_empty(outbuf->fifo))
        {
            char val = fifo_out(outbuf->fifo);  /* retrieve byte from buffer */
            sem_up(outbuf->wait);   /* signal there is room in buffer */
            out(add, val);    /* put byte in controller output register */
        }
        else
            outbuf->noMoreInt = true;   /* set end of interruptions */
    }
}
```

Access types
DMA-based I/O



- In a DMA-based I/O strategy, the DMA controller is directly connected to the device controller, or is part of it
- The underlying principle is that when the device controller wants to transfer data, it activates a request transfer input in the DMA controller

Access types
DMA-based I/O (2)



- As a result, the DMA controller takes control of the bus and performs one of two operations:
  - read a byte or word from the device controller data register and subsequently write that value to memory (data input)
  - read a byte or word from memory and subsequently write that value to the device controller data register (data output)