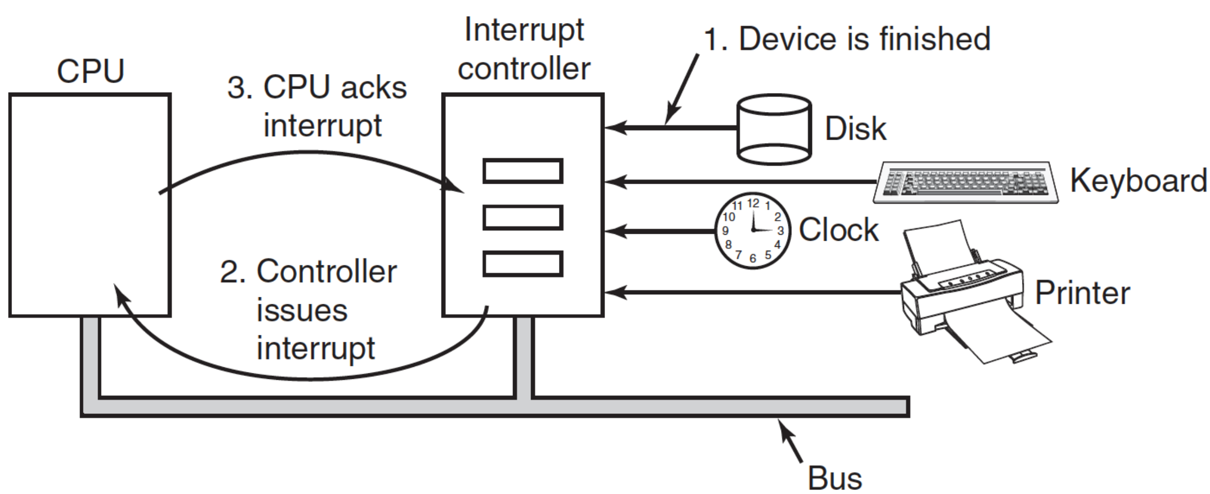
Lecture 12 (Interrupts & algorithms)

Interrupts

Interrupts: device finished → causes interrupt → if no pending interrupts when interrupt controller handles interrupt immediately →interrupt controller puts number on address line specifying device & asserts signal to interrupt to CPU → CPU stop doing its job & start doing smth else

The number on the address lines is used as an index into a table called the **interrupt vector** to fetch a new program counter with corresponding interrupt-service procedure.

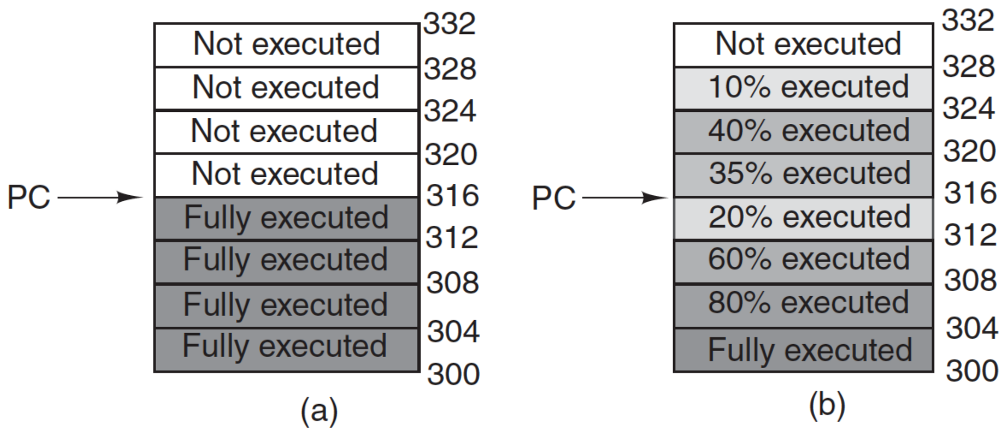


Precise interrupt - interrupt that leaves the machine in a well-defined state. Any changes of registers & memory must be done before interrupt occurs.

Precise interrupt properties:

- PC(program counter) saved in known place
- All instructions before the one pointed to by PC have completed
- No instruction beyond the one pointed to by PC has finished
- The execution state of the instruction pointed to by the PC is known

Imprecise interrupt doesn't meet properties(example: division by zero cause no need to restart program).



Principles of I/O software

Goals of I/O software:

- Device independence (access device w/o specification of device)
- Uniform naming (name of file or device should be string or integer and doesn't depend on device)
- Error handling (Errors should be handled as close to the hardware as possible)
- Synchronous vs. asynchronous:
  1. Synchronous (blocking): After read() syscall program is automatically suspended until data are available in buffer
  2. Asynchronous (interrupt driven): CPU starts transfer and do smth else until the interrupt arrives
- Buffering (data may put in buffer if needed)

Sharable device: many users may use it simultaneously(disks), **dedicated:** one user wait until another user finished(printers)

Ways of performing I/O:

- Programmed (CPU do al work): CPU continuously polls the device - **polling** or **busy waiting**

```
//STRING TO THE PRINTER PROGRAMMED I/O
copy_from_user(buffer, p,count);                      /* p is the ker nel buffer */
for (i = 0; i < count; i++) {                         /* loop on every character */
  while (*printer_status_reg != READY) ;              /* loop until ready */*
  printer_data_register = p[i];                       /* output one character */
}
return_to_user();
```

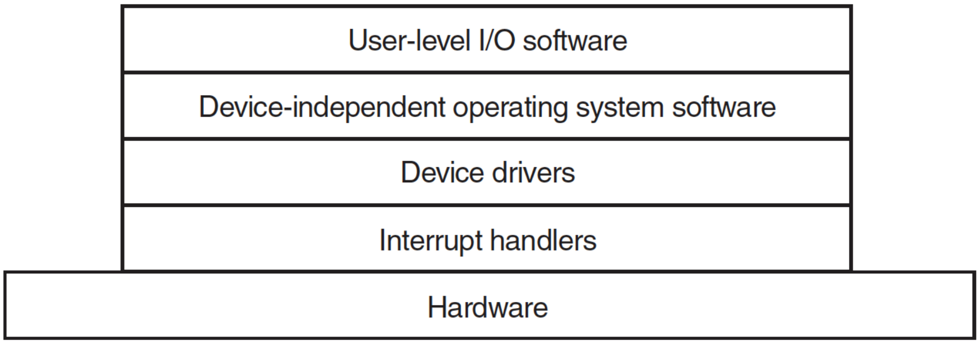
- Interrupt-driven (device did it job → interrupt → process stops & state is save → interrupt-service procedure run; interrupt occurs at every character while printing):

```
copy_from_user(buffer, p, count);                    if (count == 0) {
enable_interrupts();                                   unblock_user();
while (*printer_status_reg != READY) ;               } else {
*printer_data register = p[0];                         *printer_data_register = p[i];
scheduler();                                            count = count − 1;
                                                       i = i + 1;
/*during print syscall*/                             }
                                                     acknowledge_interrupt();
/*interrupt-service procedure ->      */             return_from_interrupt();
```

- Using DMA (DMA is programmed I/O, only with the DMA controller doing all the work, instead of the main CPU, decrease number of interrupts to one per buffer period)

```
copy_from_user(buffer, p, count);                     acknowledge_interrupt();
set_up_DMA_controller();                              unblock_user();
scheduler();                                          return_from_interrupt( );
//during print syscall                                interrupt-service procedure
```

I/O software layers



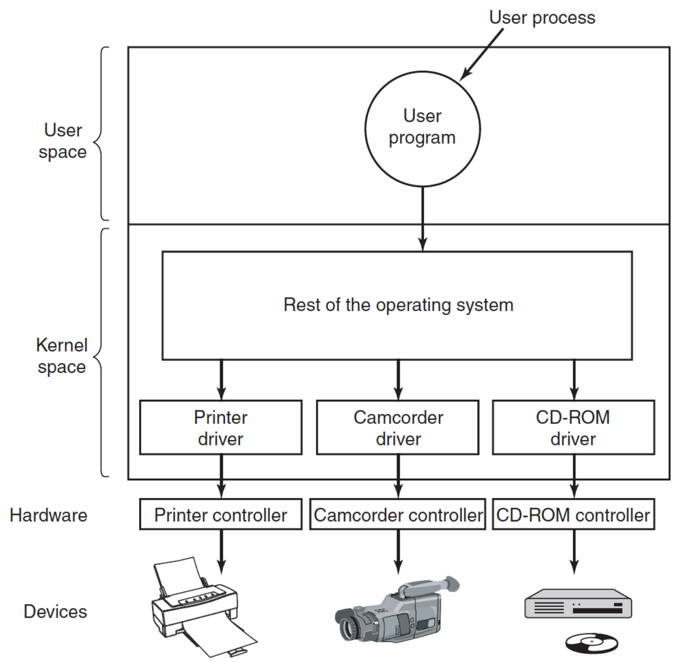
Interrupt handlers

Steps after interrupt completes:

1. Save registers (including PSW) not already saved by interrupt hardware
2. Set up context for interrupt service procedure
3. Set up stack for interrupt service procedure
4. Acknowledge interrupt controller. If no centralized interrupt controller, reenable interrupts
5. Copy registers from where saved to process table
6. Run interrupt service procedure. Extract information from interrupting device controller's
registers
7. Choose which process to run next
8. Set up the MMU context for process to run next
9. Load new process' registers, including its PSW
10. Start running the new process

Device drivers

Driver - specific code for controlling device, normally handles one device type or closely related devices.



OS - single binary program that contains all of the drivers it will need compiled into it. If a new device was added, the system administrator simply recompiled the kernel with the new driver to build a new binary - outdated structure. Drivers were dynamically loaded into the system during execution - new structure.

Drivers functions: accept abstract read and write requests from the device-independent software; driver must initialize the device, if needed; manage its power requirements and log events

Drivers structure: checking input parameters → if valid translate it from abstract to concrete terms if needed → check if the device is currently in use → if yes, request will be queued for later processing → device is idle then hardware status will be examined to see if the request can be handled now → if no, may need to switch on device or start a motor before transfers → if device is ready then actual control can begin;

Controlling of device: command sequence determined → write it to controller's device registers → check acceptance of command & readiness for next command → all commands issued → if driver waits controller then device driver block itself & wait until interrupt → error checking → may pass some data to device-independent software → return status information to caller → if queue of requests ≠ empty then next selected & started, otherwise driver blocks waiting for the next request.

Drivers have to be **reentrant** - running driver has to expect that it will be called a second time before the first call has completed.

Device-independent software

Although some of the I/O software is device specific, other parts of it are device independent. Boundary depends on system.

Functions of device-independent software:

- Uniform interfacing for device drivers (for each class of devices OS defines a set of functions that driver must supply; naming of devices; protection)

  Major device number - number in i-node(of file uniquely specified by device name) to locate appropriate driver; **minor device number** - number in i-node for passing as parameter to driver to specify unit for read/write.

- Buffering (n size buffer fill ⇒ wake up process - user space; buffer in kernel & interrupt put characters to it, buffer full ⇒ copy to user space)

  Double buffering - two buffers for data in kernel; first full → use second. Then second full ⇒ first copied to user space & first = empty

  Circular buffer - pointer on next free word & pointer to first not removed word.

  If data buffered too many times ⇒ performance decrease

- Error reporting (software determines nature of error & it reports to user)

  Programming errors - occur when a process asks for smth impossible ⇒ report to caller. Other errors: providing invalid buffer address or other parameters.

  Actual I/O errors - write on damaged, read from switched off. Driver pass problem to device-independent software

  ---

- Allocating & releasing dedicated devices (attempt to acquire not available device ⇒ caller blocks instead of device failing; blocked processes put into queue ⇒ when device is idle first from queue process will execute)

- Providing device-independent block size (higher layers deal only with abstract devices that all use the same logical block size, independent of the physical sector size)

User space I/O software

Small portion of I/O software consists of **libraries linked** together with **user** programs.

Another type is spooling systems. **Spooling(Simultaneous Peripheral Operation On Line)** is d way of dealing with dedicated I/O devices in a multiprogramming system. Instead what is done is to create a special process, called a **daemon**, and a special directory, called a **spooling directory.** Printer's work: process generates entire file → file puts into spooling directory → daemon decide which process can use printer's special file to print from directory.

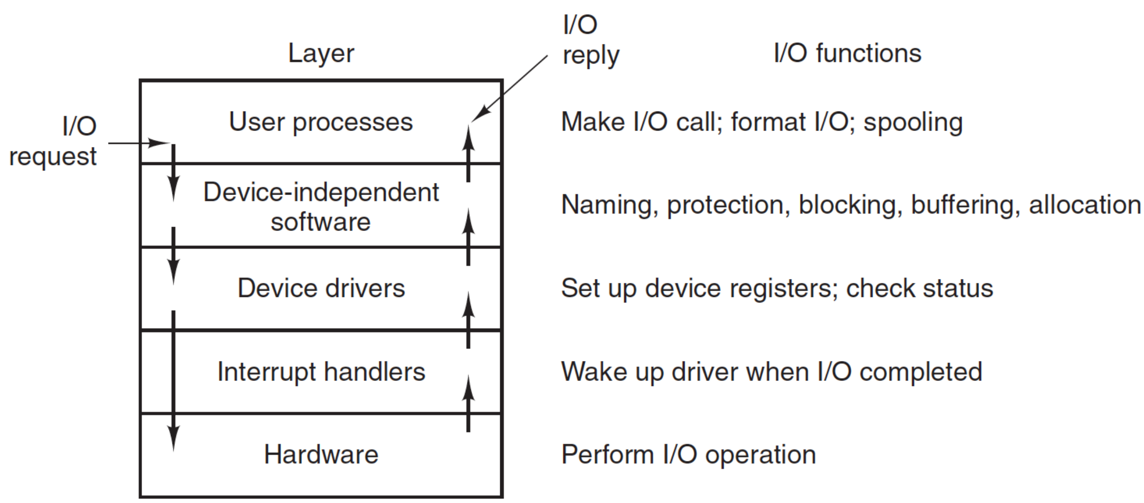


Figure 5-17. Layers of the I/O system and the main functions of each layer.

Read a block from a file → OS is invoked to carry out the call → needed block is not in cache → calls the device driver to issue the request to the hardware to go get it from the disk → process is then blocked until the disk operation has been completed and the data are safely available in the caller's buffer → disk is finished, the hardware generates an interrupt → interrupt handler is run to discover what has happened & device for attention → extracts the status from the device → wakes up the sleeping process to finish off the I/O request and let the user process continue