CI583: Data Structures and Operating Systems
OS Design Principles

Outline

Managing hardware

In an earlier lecture we discussed device drivers, which contain the code that knows how to interact with a given device.

How do we make sure that the right driver is associated with the right device, that all devices are properly initialised when the system starts, or that new devices are recognised correctly when attached?

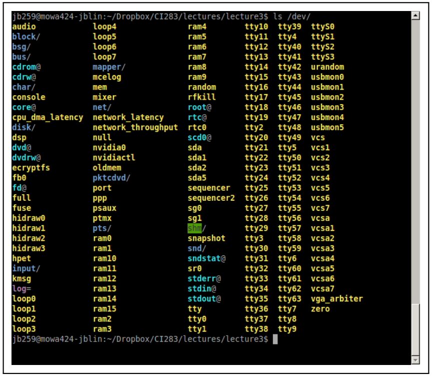
Managing hardware

In early UNIX systems the kernel had hard-coded support for the relevant devices. In order to add a new device, the user needed to recompile the kernel image.

Each device was identified by a device number which was a pair of the major device number, which identified the driver, and minor device number, which identified the particular device among those handled by this driver.

Managing hardware

Special files were created on the file system in the `/dev` directory to refer to devices.



```
jb259@mowa424-jblin:~/Dropbox/CI283/lectures/lecture3$ ls /dev/
audio            loop4               ram4        tty10  tty39  ttyS0
block/           loop5               ram5        tty11  tty4   ttyS1
bsg/             loop6               ram6        tty12  tty40  ttyS2
bus/             loop7               ram7        tty13  tty41  ttyS3
cdrom@           mapper/             ram8        tty14  tty42  urandom
cdrw@            mcelog              ram9        tty15  tty43  usbmon0
char/            mem                 random      tty16  tty44  usbmon1
console          mixer               rfkill      tty17  tty45  usbmon2
core@            net/                root@       tty18  tty46  usbmon3
cpu_dma_latency  network_latency     rtc@        tty19  tty47  usbmon4
disk/            network_throughput  rtc0        tty2   tty48  usbmon5
dsp              null                scd0@       tty20  tty49  vcs
dvd@             nvidia0             sda         tty21  tty5   vcs1
dvdrw@           nvidiactl           sda1        tty22  tty50  vcs2
ecryptfs         oldmem              sda2        tty23  tty51  vcs3
fb0              pktcdvd/            sda5        tty24  tty52  vcs4
fd@              port                sequencer   tty25  tty53  vcs5
full             ppp                 sequencer2  tty26  tty54  vcs6
fuse             psaux               sg0         tty27  tty55  vcs7
hidraw0          ptmx                sg1         tty28  tty56  vcsa
hidraw1          pts/                shm/        tty29  tty57  vcsa1
hidraw2          ram0                snapshot    tty3   tty58  vcsa2
hidraw3          ram1                snd/        tty30  tty59  vcsa3
hpet             ram10               sndstat@    tty31  tty6   vcsa4
input/           ram11               sr0         tty32  tty60  vcsa5
kmsg             ram12               stderr@     tty33  tty61  vcsa6
log=             ram13               stdin@      tty34  tty62  vcsa7
loop0            ram14               stdout@     tty35  tty63  vga_arbiter
loop1            ram15               tty         tty36  tty7   zero
loop2            ram2                tty0        tty37  tty8
loop3            ram3                tty1        tty38  tty9
jb259@mowa424-jblin:~/Dropbox/CI283/lectures/lecture3$ 
```

These "special" files refer to the device using its device number. Thus, when an application opens the "file" `/dev/sda1` the kernel uses the right device and driver.

Managing hardware

This approach was laborious and inadequate for the number of devices that can be used with today's systems.

The modern approach uses what we might call meta-drivers, each responsible for the class of devices that can use a particular bus,

So, the USB driver knows about USB devices and probes the system at start-up time to see which are attached.

Managing hardware

The correct drivers and kernel modules are then loaded to initialise these devices.

Whilst a modern Linux system is running, the `udev` “daemon” listen for the connection and removal of devices and updates the contents of `/dev` accordingly.

Most of what `udev` does is in userland (doesn’t require special privileges).

Interacting with a device

We begin by considering how the OS might interact with a very simple device – a terminal, which takes user input at the command line and displays the results in a simple text-only UI.

Although this might seem like an obsolete example, the way that the OS reads from and writes to this device is something that can be adapted for many contexts.

Interacting with a device

We can see straight away that it's not enough to read one character at a time from the device, or to write data straight to the device. Why not?

1. Data may be sent to the device faster than they can be processed, or generated by the user faster than the application can accept it.

Interacting with a device

2. Chars may arrive from the keyboard when there is no waiting `read` request.

Interacting with a device

3. Input may need to be processed in some way before they read the application.

   For instance, chars need to be echoed to the screen so that the user can see what they are typing.

   Chars may be grouped into lines which can be edited before submitting by pressing enter.

   The terminal may support tab-completion.

Interacting with a device

Items 1 and 2 are examples of the producer-consumer problem, in which two processes need to synchronise their access to a finite queue or buffer.

The producer's job is to generate a piece of data, put it into the buffer and start again.

Interacting with a device

At the same time, the consumer is consuming the data (i.e., removing it from the buffer) one piece at a time.

The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Interacting with a device

The OS will communicate with many devices using the same model of input and output queues (even a window manager, which controls the gui – here the input comes from keyboard and mouse, and the window manager needs to keep track of which application window has the focus, whilst the output is a stream of requests to repaint parts of the display using new data).

We separate this common functionality into something called a line discipline module.

Shared libraries

We have already mentioned the fact that each OS has an API and that high-level languages provide convenient calls to the API in standard libraries.

As well as wrappers for the API, standard libraries contain large numbers of standard functions for convenience.

Shared libraries

These shared libraries are called dynamic-linked libraries on Windows and shared objects on Linux.

One advantage of using them is that they need not be loaded until needed, improving the start-up time of a program.

Shared libraries

The biggest advantage is code reuse – few programmers would attempt GUI programming without a library.

One disadvantage is that they bring increased complexity (DLL A depends on DLL B, which depends on DLL C – are the versions of these DLLs compatible?) – this complexity is greatly tamed by managed runtime environments like .NET or the JVM.

Shared libraries

When a program is converted into machine code that can be executed directly by the OS, it needs to contain everything it needs to run.

When we include a call to a shared function in our program, such as a call to the `C` function `printf` (or, more indirectly, `System.out.println` in Java), a compiler could take one of several choices:

Shared libraries

Approaches to shared libraries:

1. Include a copy of `printf` with our program. This would make our programs much larger than they need to be and be a waste of storage.
2. Load a copy of `printf` at runtime, along with our program. There would only be one copy on disk but each program would load it into memory: again, this would be very inefficient.

Shared libraries

Approaches to shared libraries:

3. Ensure that all processes that make use of `printf` refer to the same known location, L. If some process wants to make use of that location for another purpose, move the single copy of `printf` to another location, M. This is the approach taken by Windows.
4. Ensure that each process that calls on `printf` maintains a table mapping an arbitrary memory location, L_1, to the real location in memory of `printf`, L_2. This approach is called position-independent code and is the approach used by most modern UNIX systems.

After the break

Storage, virtual memory, virtualisation, scheduling.