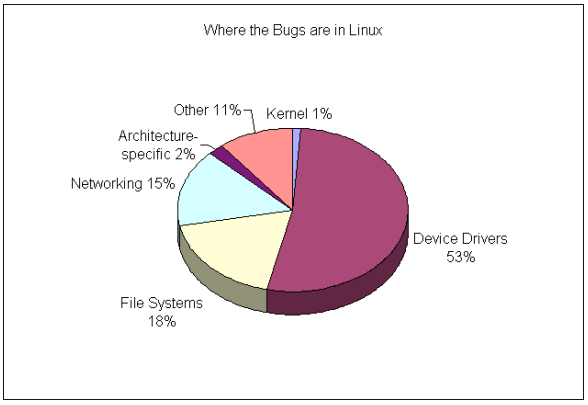
Where the bugs are

Where the bugs are in Linux (other OSs may be worse ☺).

- Graphic from http://linuxbugs.coverity.com/linuxbugs.htm



Reasons:

- Device drivers are written by many people.
- Less experience, less knowledge, less checks, kernel code is read and checked by many more people.
- We would like a way to reduce the impact of bugs in device drivers on the rest of the system.

Providing device drivers

Compiled as part of the kernel.

Loadable (and unloadable) sections of kernel level code.

- The way Linux and Windows do it.

User level service providers

- There may need to be a stub driver at kernel level.
- Can be done with memory-mapped devices by allocating the corresponding real memory addresses to the driver process.
- This is safer because the interactions with the rest of the OS are constrained to specific interfaces.
- In x86 processors, processes can request permission to use particular IO ports and if granted can use them from user mode.

Linux Kernel Modules

Sections of kernel code that can be compiled, loaded, and unloaded independently of the rest of the kernel.

A kernel module may typically implement a device driver, a file system, or a networking protocol.

The module interface allows third parties to write and distribute, on their own terms, device drivers or file systems that could not be distributed under the GPL. (Some people strongly disagree with this idea and think it is illegal.)

Kernel modules allow a Linux system to be set up with a standard, minimal kernel, without any extra device drivers built in.

Three components to Linux module support:

- module management
- driver registration
- conflict resolution

Module Management

The module requestor loads modules; it also regularly queries the kernel to see whether a dynamically loaded module is still in use, and will unload it when it is no longer actively needed.

When the kernel is compiled some symbols (functions, variables) are exported into an internal symbol table.

- These symbols can then be referenced by modules via the linking process.
- e.g. /proc/kallsyms

```
c0100000 T _text
c0100000 T startup_32
c01000b0 T startup_32_smp
c0100130 t checkCPUtype
c01001b1 t is486
c01001b8 t is386
c0100225 t check_x87
c0100260 T setup_pda
c0100282 t setup_idt
c010029f t rp_sidt
c0100322 t early_divide_err
c0100328 t early_illegal_opcode
c0100331 t early_protection_fault
c0100338 t early_page_fault
c010033f t early_fault
...
```

Module loading

The module is scanned for references to kernel symbols, these are located in the symbol table.

- If a symbol can't be found the module is not loaded.

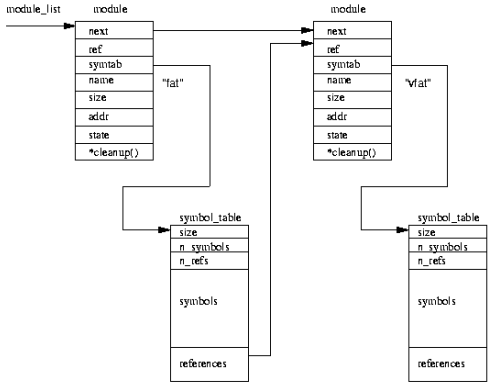
A continuous area of virtual kernel memory is requested.

The module is moved to its location.

The module itself provides more symbol-table entries to the kernel.

- These may be used by other modules.

Module Design



Driver Registration

When a module is loaded the kernel calls its startup routine.

- This routine must register the module with the kernel.

Allows modules to tell the rest of the kernel that a new driver has become available.

The kernel maintains tables of all known drivers, and provides a set of routines to allow drivers to be added to or removed from these tables at any time.

Registration tables include the following items:

- Device drivers
- File systems
- Network protocols
- Binary formats – recognise and load new types of executable files

Conflict Resolution

The conflict resolution mechanism allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver

It aims to:

- Prevent modules from clashing over access to hardware resources
- Prevent *autoprobes* from interfering with existing device drivers
- Resolve conflicts with multiple drivers trying to access the same hardware (they may be allowed to but not at the same time)

The kernel maintains lists of allocated hardware resources (interrupt lines, DMA channels and I/O address space).

- Device drivers reserve resources with the database.
- If it can't get a resource the driver may try alternatives or simply fail and ask to be unloaded.

User level driver

- Linux allows a suid root process to access IO ports – ioperm. Then the process can drop back to the actual user's uid and exec.
- Memory mapped IO also allows user processes to access IO registers. The registers are mapped into the *ordinary* address space. This process is allowed to read and write these addresses.
- Interrupts are a problem. The driver has to somehow respond to and request interrupts. Experimental versions of Linux allow interrupts to be accessed via the /proc file system. (Currently you can read /proc/interrupts.)
- DMA uses interrupts but also requires memory buffers (allocated as contiguous pages of physical memory). There has to be a way of obtaining the bus (usually physical) addresses for the DMA controller.
- Because of kernel/user transitions they can be slower than kernel level.

I/O Performance

Bad things

- Lots of context switches (interrupts, and passing I/O through layers of device drivers or filters)
- Copying of data (from controllers to memory, from user to kernel space and possibly between layers, from one machine to another on a network)

Solutions

- PIO – Programmed I/O
  - Busy waiting can be effective if we can be sure the wait isn't going to be long.
  - This cuts down the number of context switches.
- Put I/O handling at the same level, e.g. move different parts of the procedure into the kernel.
  - This is what Solaris did with the telnet daemon and Microsoft did with the graphics display system in Windows NT 4.
  - Not only does this cut down on context switches, it also reduces the amount of data copying required.
- Transfer data in larger chunks.
  - This can cut down on the number of interrupts (1 per chunk)
- Use DMA controllers or I/O processors to work in parallel with the CPU.

General services

We saw the block buffer cache that UNIX provides for all of its block device drivers. This is only one of several services the OS might provide to devices and device drivers.

Buffers are used to deal with the differences in speed between the processor and devices, and between different devices.

- serial connections or modem connections to disk
- data produced by a process to disk

In both cases the buffer would be the size of a disk block (or a multiple of this) to facilitate writing to disk.

Buffers are also used to package data into different sized chunks for transfer between different devices, especially over networks.

A disk write system call may copy the data from user space to a kernel buffer in order to ensure copy semantics. What is another solution to this problem?

Double buffering is used when data is being read and written.

Caching

Data which is currently being accessed is cached in faster memory for efficiency purposes. e.g.

- disk storage is cached by
- main memory which is cached by
- the CPU's secondary cache which is cached by
- the CPU's primary cache which may be cached by
- processor registers

A cache is like a buffer but holds a copy of data, a buffer might hold the only existing version of the data.

So the UNIX block buffer cache is both. It buffers information and maintains it as a cache.

Spooling and device reservation

Used initially to provide a buffer on a fast device for data on a slow device.
Also used to stop the interleaving of data from different sources on the same device.

Effectively sharing a non-sharable device over several processes.

Commonly used for printing.

- A process opens output to the printer.
- The output is stored in a buffer (usually a file).
- A request to print this output is recorded by the printing system.
- A thread or process maintains the queue of printing

Device reservation – allowing one process exclusive access to the device at a time.

Before next time

Read from the textbook

- 10.2 Disk Structure
- 10.4 Disk Scheduling
- 10.5 Disk Management
- 10.6 Swap-Space Management