

---

# Device Drivers

---

Witawas Srisa-an

Embedded Systems Design and  
Implementation

---

# Device Drivers

- What are device drivers?
  - What are they used for?
  - Why are they important?

Abstraction of underlying hardware from the OS and applications.

Provide uniform APIs to access hardware.

---

---

# Introduction

## ■ Device drivers

- ❑ Black boxes to hide details of hardware devices
  - ❑ Use standardized calls
    - Independent of the specific driver
  - ❑ Main role
    - Map standard calls to device-specific operations
  - ❑ Can be developed separately from the rest of the kernel
    - Plugged in at runtime when needed
-

---

# The Role of the Device Driver

- Implements the *mechanisms* to access the hardware
    - E.g., show a disk as an array of data blocks
  - Does not force particular *policies* on the user
    - Examples
      - Who many access the drive
      - Whether the drive is accessed via a file system
      - Whether users may mount file systems on the drive
-

# Introduction

- ✗ A device driver is computer program that allows a system to interface with hardware devices.
- ✗ Example driver: printer driver, bluetooth driver
- ✗ Example devices: your USB stick, sensors: accelerometer
- ✗ It is a translator between the operating system and applications the use the devices and the devices.
- ✗ A typical operating system has many device drivers built into it.
- ✗ A device driver converts general IO instructions into device specific operations.
- ✗ Device drivers operate in a privileged mode → requires careful design

# Why Device Driver?

- ✖ A typical computing system (lap top, computer, cell phone, PDA, Point of sale system) deals with a variety of devices.
- ✖ Making a hardware device work as expected is a cumbersome task.
- ✖ Instead adding this code every application, operating system provides a single point interface for all devices by hosting the device drivers.
- ✖ Adding it under the operating systems provides the protection and security needed for the device drivers from malicious use.
- ✖ The device drivers are essentially shared dynamically linked libraries.

---

# Policy-Free Drivers

- A common practice
    - Support for synchronous/asynchronous operation
    - Be opened multiple times
    - Exploit the full capabilities of the hardware
  - Easier user model
  - Easier to write and maintain
  - To assist users with policies, release device drivers with user programs
-

---

# Why Do We Need Drivers?

- Custom platforms
    - Contain many peripheral devices and kernel (e.g. CE) supported CPU
      - OAL development to get kernel to boot on the board
      - Device drivers to allow applications to access peripheral devices
-



---

# An Example

- To access a serial port in Windows CE
    - calls `CreateFile( )` on COMx
    - calls `WriteFile( )` to write some bytes of data to the serial port
    - calls `CloseHandle( )` to close the serial port.
-

---

# Drivers in CE

- A driver is simply a dynamic-link library (DLL)
    - DLLs are loaded into a parent process address space
    - Run in user mode (different than most OSs)
    - the parent process can then call any of the interfaces exposed from the DLL
      - LoadLibrary( )
      - LoadDriver( ) --- memory resident
-

---

# Writing Device Drivers

- Most of the time spent on system integration is in this process
    - Some devices have support on standard CE release and some don't
      - Onboard or installable
        - Onboard devices may require platform dependent modification to the kernel image
        - Installable devices may require code to be more platform independent
    - Drivers APIs can also be used to provide application-level services
-

# Device Driver Models

- Device drivers, over the years, have become very complex
  - Drivers are separated into classes
    - Serial, network, audio, video, touch panel, etc.
  - Layer approach is used
    - To support a new device, a layer is modified instead of rewriting the entire driver
    - Processing functions required for a given class often do not require modification

---

# File abstraCtion

- What do you with a device? {read, write}, {read only}, {write only}
- Lets look at some examples: USB device, CD-ROM, LED Display,
- What do you do with a file? open, close, read, write, ..
- File is an excellent abstraction for devices.

# /dev partial listing

```
✱ total 380
✱ lrwxrwxrwx 1 root      30 Mar  7 2004 allkmem -> ../devices/pseudo/mm@0:
✱ allkmem
✱ lrwxrwxrwx 1 root      27 Aug 15 2001 arp -> ../devices/pseudo/arp@0:arp
✱ lrwxrwxrwx 1 root      7 Aug 15 2001 audio -> sound/0
✱ lrwxrwxrwx 1 root     10 Aug 15 2001 audioctl -> sound/0ctl
✱ lrwxrwxrwx 1 root     11 Oct  4 03:06 bd.off -> /dev/term/b
✱ drwxr-xr-x 2 root     512 Aug 17 2001 cfg
✱ lrwxrwxrwx 1 root     31 Aug 15 2001 conslog -> ../devices/pseudo/log@0
✱ :conslog
✱ lrwxrwxrwx 1 root     30 Aug 15 2001 console -> ../devices/pseudo/cn@0:
✱ console
✱ drwxr-xr-x 2 root     512 Aug 15 2001 cua
✱ drwxr-xr-x 2 root    2048 Aug 31 2002 dsk
✱ lrwxrwxrwx 1 root     29 Aug 15 2001 dump -> ../devices/pseudo/dump@0:d
✱ ump
✱ lrwxrwxrwx 1 root     50 Aug 15 2001 ecpp0 -> ../devices/pci@1f,4000/eb
✱ us@1/ecpp@14,3043bc:ecpp0
✱ lrwxrwxrwx 1 root      8 Aug 15 2001 fb0 -> fbs/ffb0
✱ drwxr-xr-x 2 root     512 Aug 15 2001 fbs
✱ dr-xr-xr-x 2 root    528 Nov  9 11:51 fd
✱ lrwxrwxrwx 1 root     30 Apr  7 2002 fssnapctl -> ../devices/pseudo/
```

---

# Device SPACE

- ✗ Typically there are multiple devices of the same type.
- ✗ All the devices controlled by the same device driver is given the same “major number”
- ✗ A “minor number” distinguishes among the devices of the same type.
- ✗ Example: printers have a major number since purpose is same, minor# is denote a specific printer

---

# Splitting the Kernel

- Process management
    - Creates, destroys processes
    - Supports communication among processes
      - Signals, pipes, etc.
    - Schedules how processes share the CPU
  - Memory management
    - Virtual addressing
-



# Splitting the Kernel

## ■ File systems

- Everything in UNIX can be treated as a file
- Linux supports multiple file systems

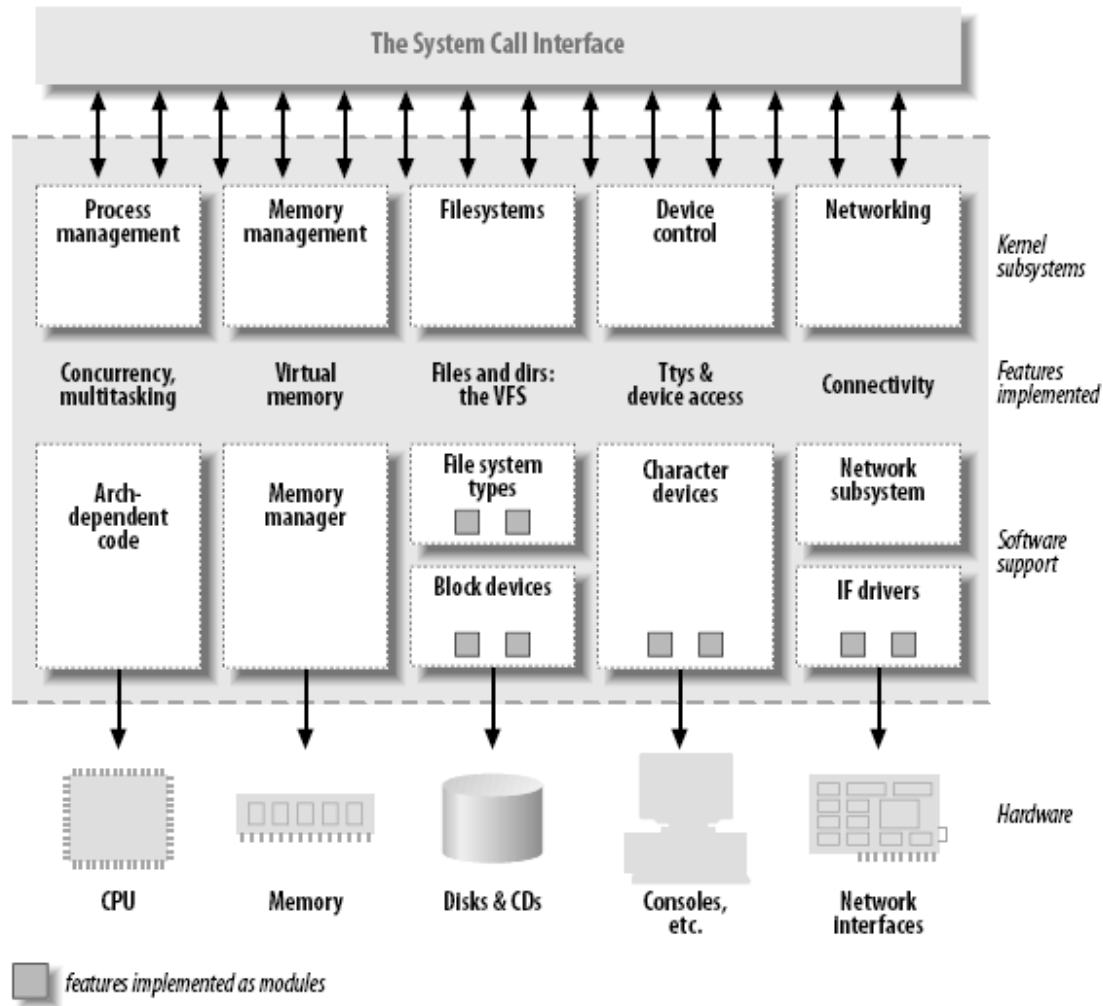
## ■ Device control

- Every system operation maps to a physical device
  - Few exceptions: CPU, memory, etc.

## ■ Networking

- Handles packets
- Handles routing and network address resolution issues

# Splitting the Kernel

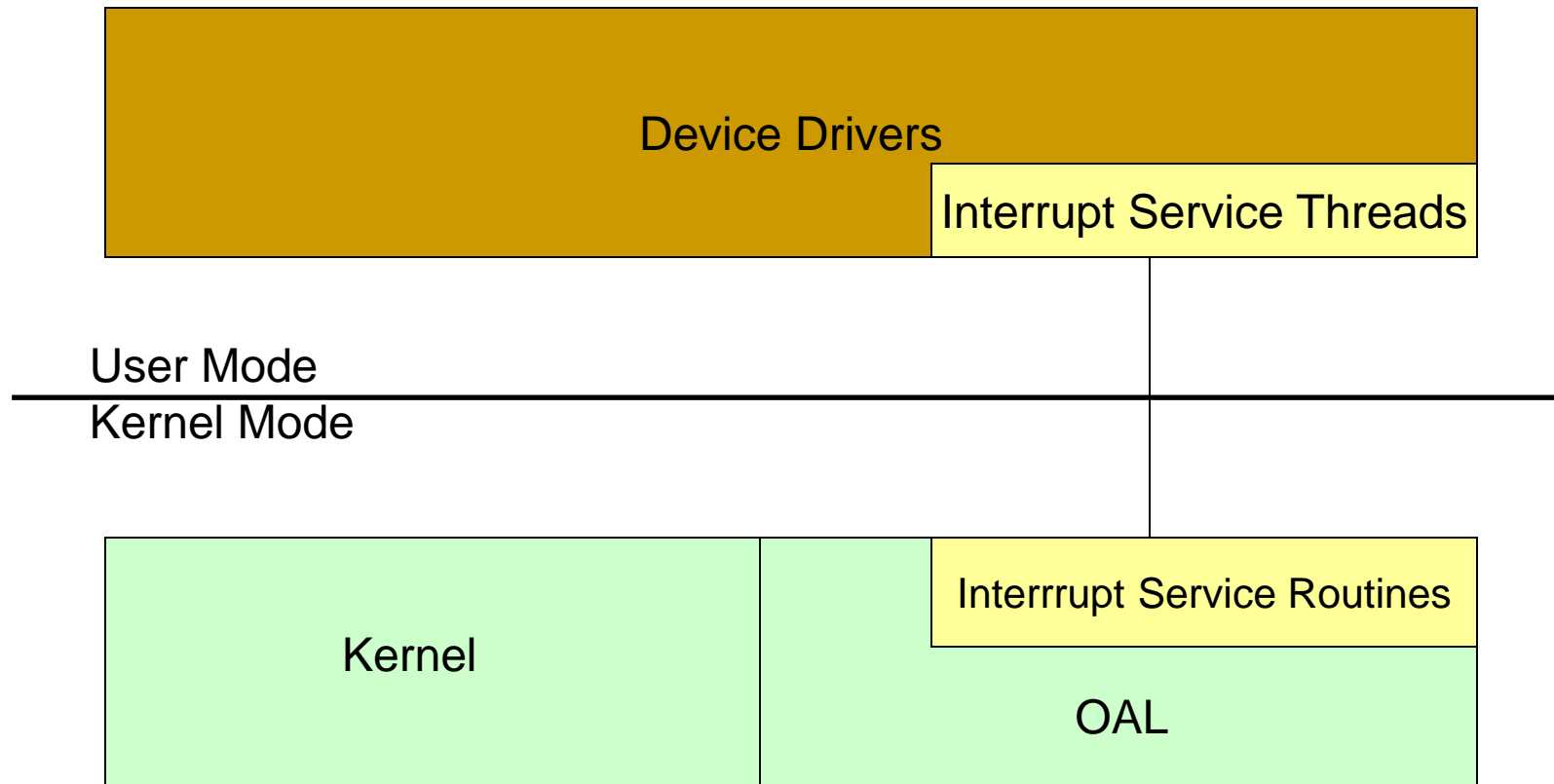


---

# Loadable Modules

- The ability to add and remove kernel features at runtime
  - Each unit of extension is called a *module*
  - Use `insmod` program to add a kernel module
  - Use `rmmmod` program to remove a kernel module
-

# Device Driver Architecture



From Wilson and Havawala, Building Powerful Platform with Windows CE  
Addison-Wesley, 2001 page 227.

# Stream-Interface

- Allow accesses to an array of devices through basic file I/O interfaces
  - Often used for installable devices
  - The device manager (DM) manages stream-interface drivers
    - DM operates as its own process
      - Device driver DLLs are mapped into the address space

---

# Other Driver Models

- Native Interface
  - USB Interface
  - Network Device Interface Specification
-

---

# Classes of Devices and Modules

- Character devices
  - Block devices
  - Network devices
  - Others
-

# Character Devices

- Abstraction: a stream of bytes
  - Examples
    - Text console (`/dev/console`)
    - Serial ports (`/dev/ttyS0`)
  - Usually supports **open**, **close**, **read**, **write**
  - Accessed sequentially (in most cases)
  - Might not support file seeks
  - Exception: frame grabbers
    - Can access acquired image using **mmap** or **lseek**



---

# Block Devices

- Abstraction: array of storage blocks
  - However, applications can access a block device in bytes
    - Block and char devices differ only at the kernel level
    - A block device can host a file system
-

---

# Network Devices

- Abstraction: data packets
  - Send and receive packets
    - Do not know about individual connections
  - Have unique names (e.g., **eth0**)
    - Not in the file system
    - Support protocols and streams related to packet transmission (i.e., no **read** and **write**)
-

---

# Other Classes of Devices

- Examples that do not fit to previous categories:
    - ❑ USB
    - ❑ SCSI
    - ❑ FireWire
    - ❑ MTD
-

---

# File System Modules

- Software drivers, not device drivers
  - Serve as a layer between user API and block devices
  - Intended to be device-independent
-

---

# Security Issues

- Deliberate vs. incidental damage
  - Kernel modules present possibilities for both
  - System does only rudimentary checks at module load time
  - Relies on limiting privilege to load modules
    - And trusts the driver writers
  - Driver writer must be on guard for security problems
-

# Security Issues

- Do not define security policies
  - Provide mechanisms to enforce policies
- Be aware of operations that affect global resources
  - Setting up an interrupt line
    - Could damage hardware
  - Setting up a default block size
    - Could affect other users

# Security Issues

- Beware of bugs
  - Buffer overrun
    - Overwriting unrelated data
  - Treat input/parameters with utmost suspicion
  - Uninitialized memory
    - Kernel memory should be zeroed before being made available to a user
    - Otherwise, information leakage could result
      - Passwords

---

# Security Issues

- Avoid running kernels compiled by an untrusted friend
  - Modified kernel could allow anyone to load a module



# Version Numbering

- Every software package used in Linux has a release number
  - You need a particular version of one package to run a particular version of another package
  - Prepackaged distribution contains matching versions of various packages

---

# Version Numbering

- Different throughout the years
  - After version 1.0 but before 3.0
    - <major>.<minor>.<release>.<bugfix>
    - Time based releases (after two to three months)
  - 3.x
    - Moved to 3.0 to commemorate 20<sup>th</sup> anniversary of Linux
    - <version>.<release>.<bugfix>
    - <https://lkml.org/lkml/2011/5/29/204>
-

---

# License Terms

- GNU General Public License (GPL2)
    - GPL allows anybody to redistribute and sell a product covered by GPL
      - As long as the recipient has access to the source
      - And is able to exercise the same rights
    - Any software product derived from a product covered by the GPL be released under GPL
-

---

# License Terms

- If you want your code to go into the mainline kernel
  - Must use a GPL-compatible license

---

# Joining the Kernel Development Community

- The central gathering point
  - Linux-kernel mailing list
  - <http://www.tux.org/lkml>
- Chapter 20 of LKM further discusses the community and accepted coding style

---

# Device Driver Labs

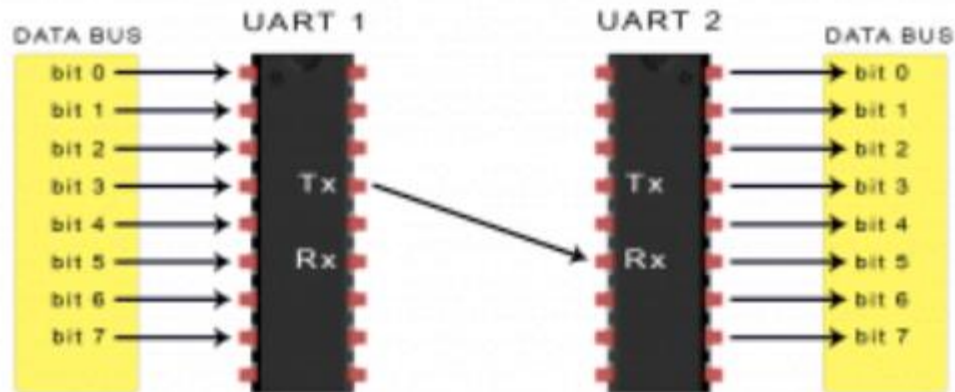
- <http://www.microsoft.com/downloads/details.aspx?FamilyID=486E8250-D311-4F67-9FB3-23E8B8944F3E&displaylang=en>
  - Download lab tools from the course website
  - Create a simple Internet Appliance Windows CE project
  - Create a device driver workspace and build the workspace
  - Build OS image
  - Attach the device
  - Open prompt through build OS | Open Release Directory
    - `dumpbin -exports StreamDrv.dll`
-

# XINU

- Xinu Is Not Unix, is an operating system for embedded systems, originally developed by Douglas Comer for educational use at Purdue University in the 1980s. The name is both recursive, and is Unix spelled backwards.
- <https://xinu.cs.purdue.edu/>
- <https://en.wikipedia.org/wiki/Xinu>
- <https://github.com/xinu-os/xinu>

# UART

- UART stands for Universal Asynchronous Receiver/Transmitter.
- It's not a communication protocol like SPI and I2C, but a physical circuit in a microcontroller, or a stand-alone IC.
- A UART's main purpose is to transmit and receive serial data.





# Examples from XINU

- ✗ Take a look at files in the include directory:
- ✗ device.h
- ✗ tty.h
- ✗ uart.h
- ✗ Also in the system directory devtable.c, initialize.c
- ✗ Bottom line is this, for a device xyz:
  1. Include a file in include directory: xyz.h
    - define the operations/functions for the device
  2. Add a directory xyz
    - implement all functions each in its own file
  3. Add an entry in the devtable.c for the device (note that this has the “minor” device number along with other things)

# Lets Analyze the XINU UART Driver

- Starting point: uart.h in include directory
- uart directory functions
- system directory devtable.c, initialize.c
- Usage of the devices is through device table:
- Ex:  

```
pdev = &devtab[i];  
(pdev→init)(pdev);
```

# UART Driver in EXINU

1. General device driver related files: device.h, devtable.c
2. Uart files: uart.h defining the physical features of the uart
3. All the files in the uart directory that implement the operations related to the uart.
  - + uartControl.c    uartInit.c    uartIntr.c
  - + uartPutChar.c    uartWrite.c    uartGetChar.c    uartRead.c

# Device Drivers

- On board devices are called internal peripherals and one outside are called external peripherals
  - UART Chip (internal)
  - TTY (external)
- UART → transceiver → RS232 → D-9 connector → laptop serial socket
  - WRT54GL board and modifications

# Device drivers (contd.)

- Embedded processor interacts with a peripheral device through a set of control and status registers.
- Registers are part of the peripheral device.
- Registers within a serial controller are different from those in a timer.
- These devices are located in the memory space of the processor or I/O space of the processor-- two types: memory-mapped or I/O mapped respectively.

---

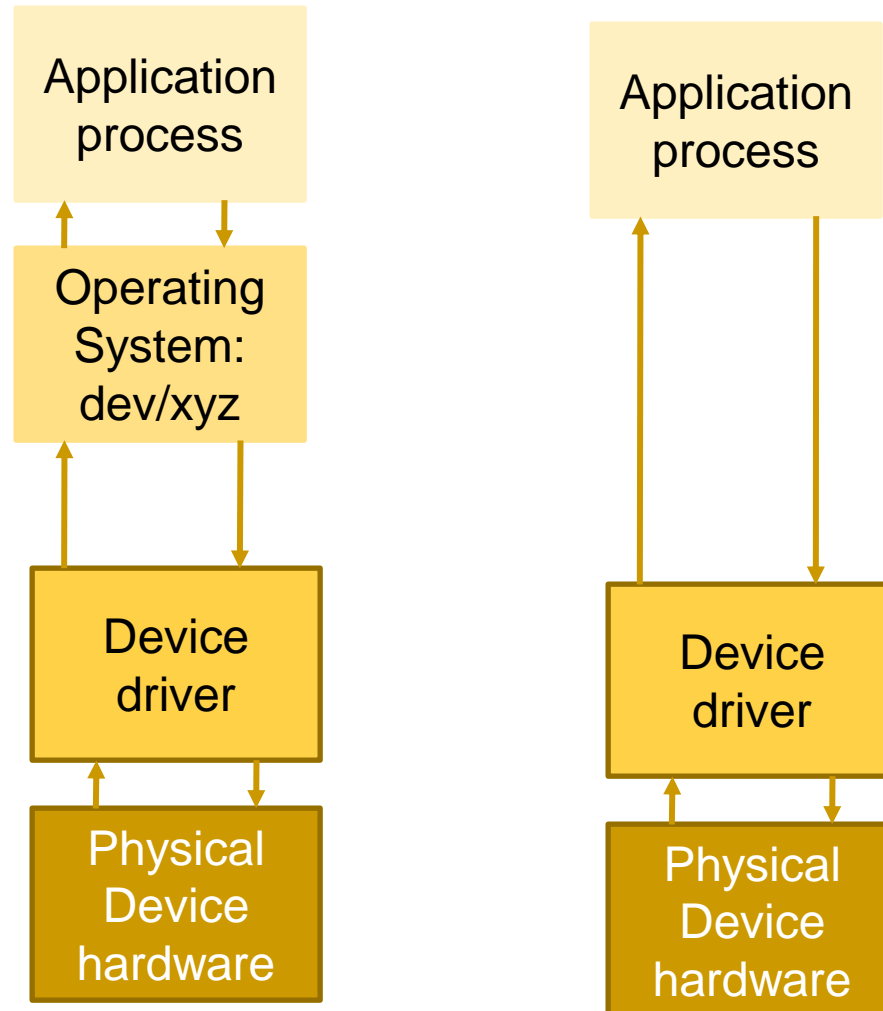
# Device driver (contd.)

- The keyword `volatile` should be used when declaring pointers to device drivers.
  - ❑ Bit patterns for testing, setting, clearing, toggling, shifting bits, bitmasks, and bitfields.
- Struct overlays:
  - ❑ In embedded systems featuring memory mapped IO devices, it is common to overlay a C struct on to each peripheral's control and status registers.
  - ❑ This will provide the offsets for the various registers from the base address of the device.

# Device Driver Philosophy

- ✗ Hide the hardware completely: hardware abstraction
- ✗ If the device generates any interrupts include interrupt controllers.
- ✗ Device driver presents a generic interface for applications at higher level to access the devices: `device.h`
- ✗ Device drivers in embedded systems are different from general purpose operating systems: See diagram in slide #14
  - + Applications in general purpose systems accesses OS (Operating Systems) which in turn accesses device drivers.
  - + Applications in embedded systems can directly access device drivers.

# General Purpose OS vs. Embedded System





# Device Driver development steps

1. An interface to the control and status registers.
2. Variables to track the current state of the physical and logical devices
  - Major and minor device number, device name
3. A routine to initialize the hardware to known state
4. An API for users of the device driver
  - Read, write, seek
5. Interrupts service routines

---

## Example: A serial device driver

- Read the text for explanation and general example of a timer
- Now lets look at the UARTdriver of the embedded xinu and WRT54GL.
- Study the tty driver that is a logical device that is layered on top of the UART driver.
- Discuss how you would develop a device driver for a framebuffer.. This was a lab2 last year.

# Shift Operators

- << left shift
- >> right shift

Usage:

```
unsigned int x = 70707;
```

```
//x = 00000000 00000001 00010100 00110011
```

```
unsigned int y, z;
```

```
y = x << 2;
```

```
// y = 00000000 00000100 01010000 11001100
```

```
z = x >> 2;
```

```
//z = 00000000 00000000 01000101 00001100
```

# Logic Operators

- Bitwise & (AND)
- Bitwise inclusive | (OR)
- Bitwise exclusive ^ (XOR)
- Bitwise negation ~

Usage:

```
unsigned exp1 = 1;  
unsigned exp2 = 4;  
printf (" %d\n", exp1 | exp2);  
printf (" %d\n", exp1 & exp2);  
printf (" %d\n", exp1 ^ exp2);  
printf (" %d\n", ~exp1);
```

# Relevance of shift and logic operators

- ✖ Bitwise operations are necessary for much low-level programming, such as writing to device drivers, low-level graphics, communications protocol packet assembly and decoding.
- ✖ Device drivers use these operators to test the presence or absence of a bit in a serial port or a device input, for example. (checking for on or off)

---

# Summary

- We studied the design and development of device drivers.
- We analyzed the code for a sample UART driver.

# Linux Device Drivers & Project3 preview

---

CSC345

---

# Project 3 Preview

- Write a device driver for a pseudo stack device
  - Idea from <http://www.cs.swarthmore.edu/~newhall/cs45/f01/proj5.html>
  - Linux character device type supports the following operations
    - Open: only one is allowed.
    - Write: writes an char string to top of the device stack. Error if stack is empty
    - Read: reads an item from top of the device stack. Error if stack is empty
    - Release: release the device
  - Install with LKM.
  - Test: It will be a dedicated standalone machine in the lab. Root password may be given out. If you mess up, you will re-install the
-

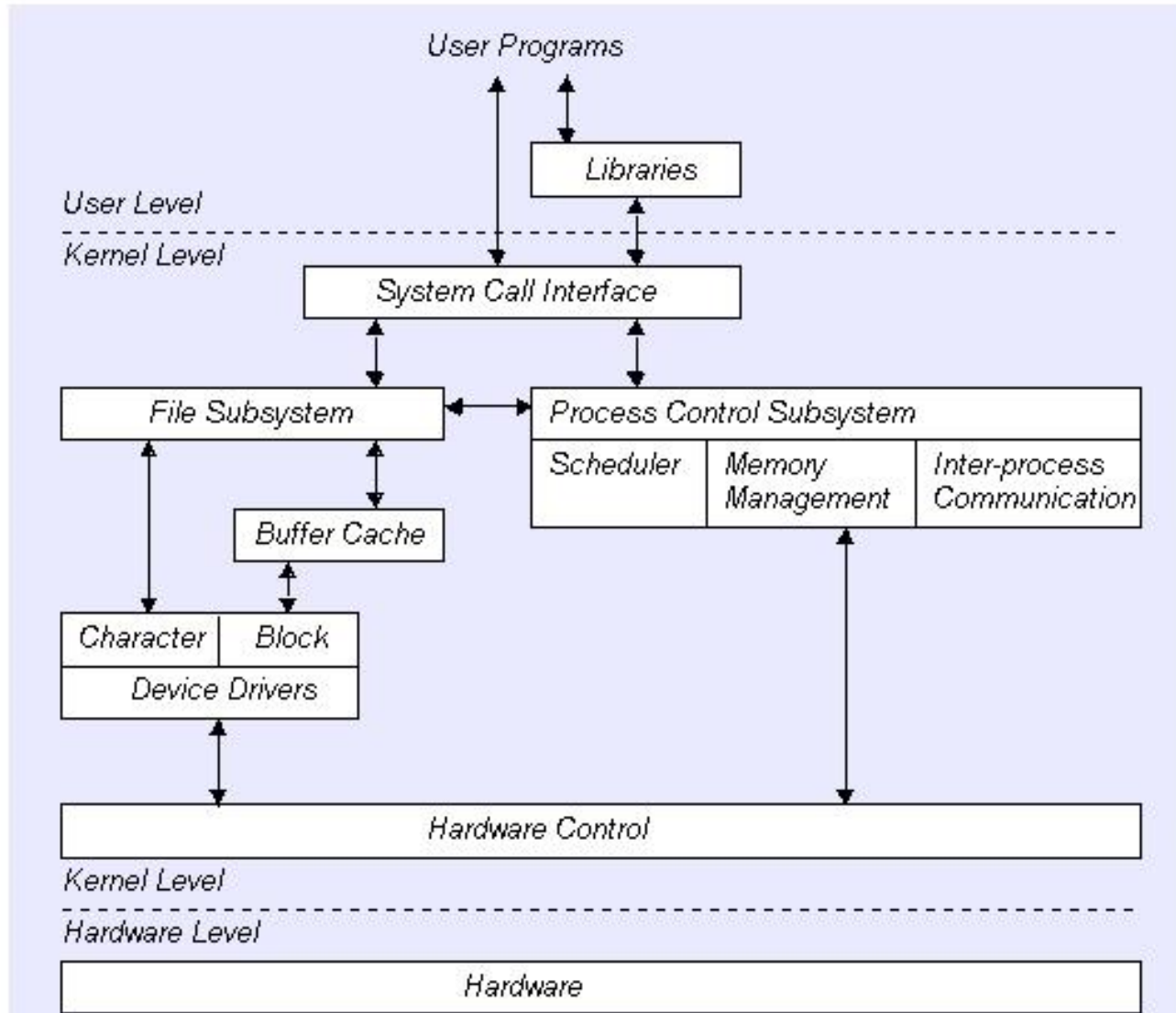


---

# What is a device driver?

- A programming module with interfaces
    - Communication Medium between application/user and hardware
  - In Unix,
    - Kernel module
    - device driver interface = file interface
    - What are normal operations?
    - Block vs. character
-

# User program & Kernel interface



Note: This picture is excerpted from Write a Linux Hardware Device Driver, Andrew O'Shaughnessy, Unix world

---

# Loadable Kernel Module (LKM)

- A new kernel module can be added on the fly (while OS is still running)
  - LKMs are often called “kernel modules”
  - They are not user program
-

---

# Types of LKM

- Device drivers
  - Filesystem driver (one for ext2, MSDOS FAT16, 32, NFS)
  - System calls
  - Network Drivers
  - TTY line disciplines. special terminal devices.
  - Executable interpreters.
-

# Basic LKM (program)

- Every LKM consist of two basic functions (minimum) :

```
int init_module(void) /*used for all initialition stuff*/
```

```
{
```

```
...
```

```
}
```

```
void cleanup_module(void) /*used for a clean shutdown*/
```

```
{
```

```
...
```

```
}
```

- Loading a module - normally retriected to root - is managed by issuing the follwing command: # insmod module.o

# LKM Utilities cmd

- insmod
  - Insert an LKM into the kernel.
- rmmod
  - Remove an LKM from the kernel.
- depmod
  - Determine interdependencies between LKMs.
- kerneld
  - Kernel daemon program
- ksyms
  - Display symbols that are exported by the kernel for use by new LKMs.
- lsmod
  - List currently loaded LKMs.
- modinfo
  - Display contents of .modinfo section in an LKM object file.
- modprobe
  - Insert or remove an LKM or set of LKMs intelligently. For example, if you must load A before loading B, Modprobe will automatically load A when you tell it to load B.

# Common LKM util cmd

- Create a special device file

% mknode /dev/driver c 40 0

- Insert a new module

% insmod modname

- Remove a module

- %rmmod modname

- List module

% lsmod

Or % more /proc/modules

audio	37840	0	
cmpci	24544	0	
soundcore	4208	4	[audio cmpci]
nfsd	70464	8	(autoclean)

# Linux Device Drivers

---

- A set of API subroutines (typically system calls) interface to hardware
  - Hide implementation and hardware-specific details from a user program
  - Typically use a file interface metaphor
  - Device is a special file
-



---

## Linux Device Drivers (continued)

- Manage data flow between a user program and devices
  - A self-contained component (add/remove from kernel)
  - A user can access the device via file name in `/dev` , e.g. `/dev/lp0`
-

# General implementation steps

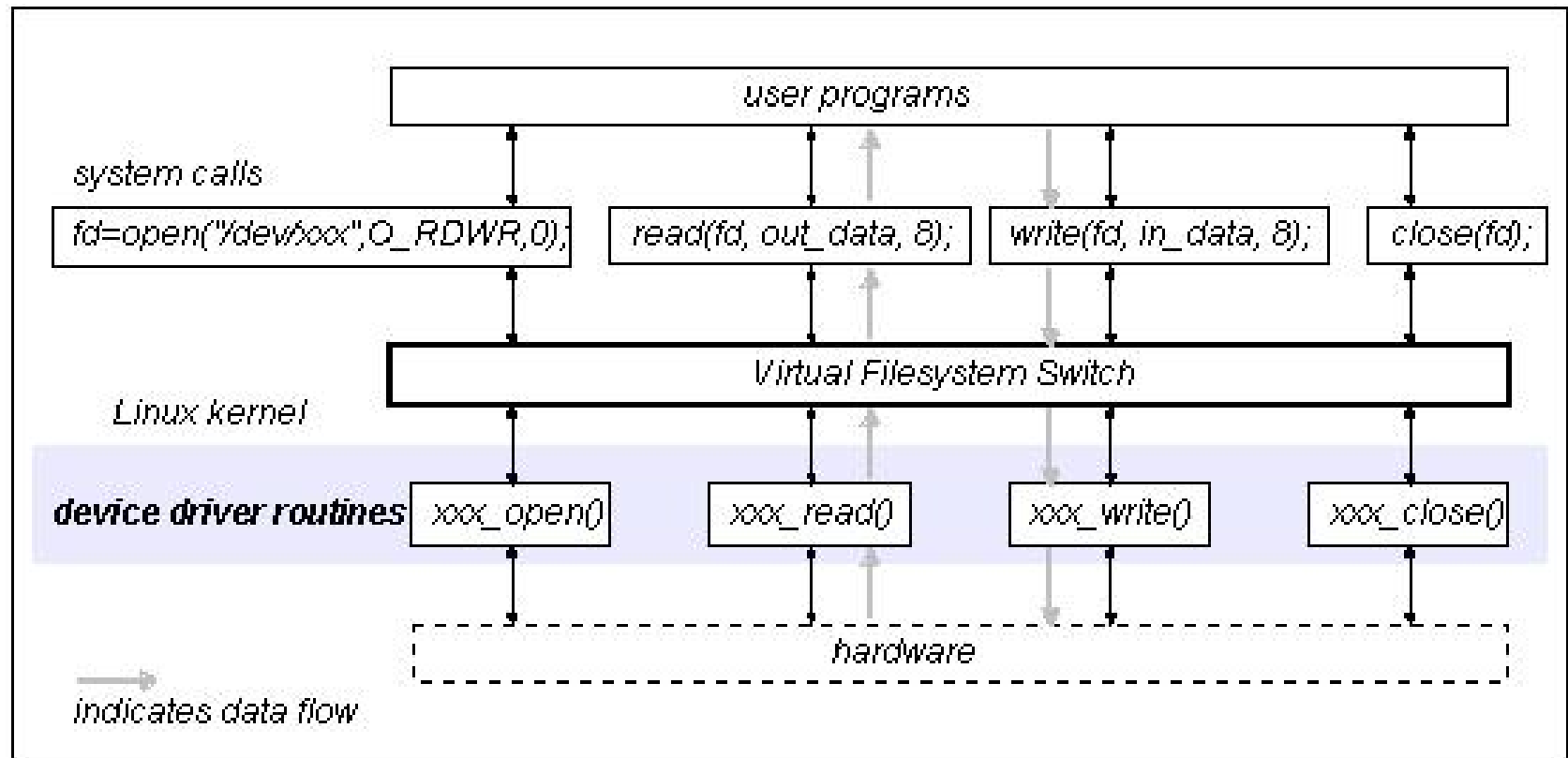
1. Understand the device characteristic and supported commands.
2. Map device specific operations to unix file operation
3. Select the device name (user interface)
  - ❑ Namespace (2-3 characters, /dev/lp0)
4. (optional) select a major number and minor (a device special file creation) for VFS interface
  - ❑ Mapping the number to right device sub-routines
5. Implement file interface subroutines
6. Compile the device driver
7. Install the device driver module with loadable kernel module (LKM)
8. or Rebuild (compile) the kernel

---

# Read/write (I/O)

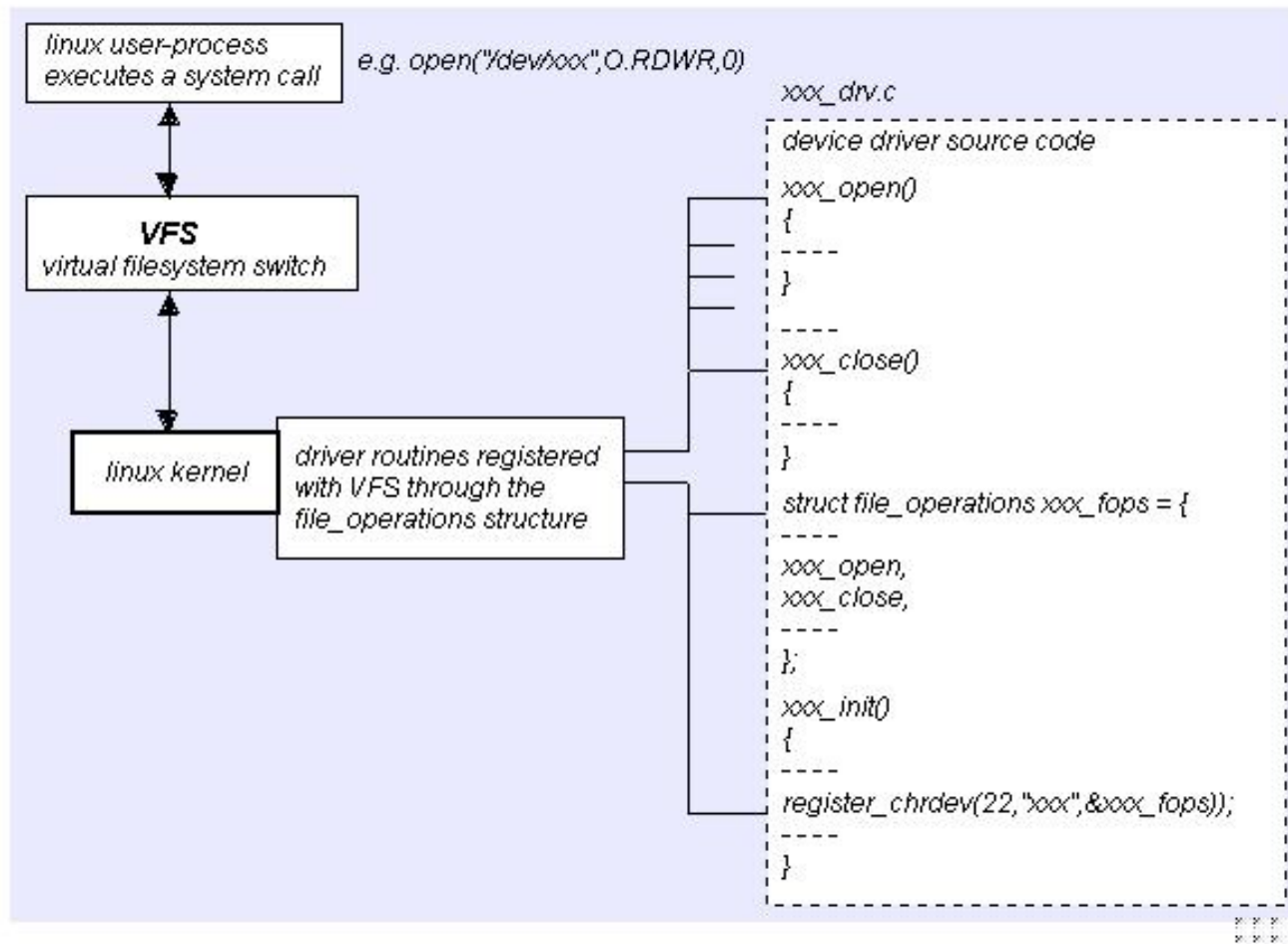
- Pooling (or synchronous)
  - Interrupt based
-

# Device Driver interface



# VFS & Major number

- principal interface between a device driver and Linux kernel



# File operation structure

- ```
struct file_operations
Fops = {
    NULL,      /* seek */
    xxx_read,
    xxx_write,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,      /*
xxx_open,
NULL,      /* flush */
xxx_release /* a.k.a.
close */
};
```



- ```
struct file_operations
Fops = {
    read: xxx_read,
    write: xxx_write,
    open: xxx_open,
    release:
        xxx_release, /*
a.k.a. close */
};
```

Watch out compatibility issue with Linux version

---

# Device special file

- Device number

- Major (used to VFS mapping to right functions)
- Minor (sub-devices)

- `mknod /dev/stk c 38 0`

- `ls -l /dev/tty`

- `crw-rw-rw- 1 root root 5, 0 Apr 21 18:33 /dev/tty`

# Register and unregister device

```
int init_module(void) /*used for all initialition stuff*/
{
    /* Register the character device (atleast try) */
    Major = register_chrdev(0,
                           DEVICE_NAME,
                           &Fops);
    :

}
void cleanup_module(void) /*used for a clean shutdown*/

{ret = unregister_chrdev(Major, DEVICE_NAME);

...
}
```



# Register and unregister device

- compile

`-Wall -DMODULE -D__KERNEL__ -DLINUX -DDEBUG -I  
/usr/include/linux/version.h -I/lib/modules/`uname -r`/build/include`

- Install the module

`%insmod module.o`

- List the module

`%lsmod`

- If you let the system pick Major number, you can find the major number (for special creation) by

`% more /proc/devices`

- Make a special file

`% mknod /dev/device_name c major minor`

# Device Driver Types

- A character device driver ( c )
  - Most devices are this type (e.g.Modem, Ip, USB)
  - No buffer.
- A block device driver (b)
  - through a system buffer that acts as a data cache.
  - Hard drive controller and HDs

---

# Implementation

- Assuming that your device name is Xxx
  - Xxx\_init() initialize the device when OS is booted
  - Xxx\_open() open a device
  - Xxx\_read() read from kernel memory
  - Xxx\_write() write
  - Xxx\_release() clean-up (close)
  - init\_module()
  - cleanup\_module()
-

# kernel functions

- **add\_timer()**
  - Causes a function to be executed when a given amount of time has passed
- **cli()**
  - Prevents interrupts from being acknowledged
- **end\_request()**
  - Called when a request has been satisfied or aborted
- **free\_irq()**
  - Frees an IRQ previously acquired with request\_irq() or irqaction()
- **get\_user\*()**
  - Allows a driver to access data in user space, a memory area distinct from the kernel
- **inb(), inb\_p()**
  - Reads a byte from a port. Here, inb() goes as fast as it can, while inb\_p() pauses before returning.
- **irqaction()**
  - Registers an interrupt like a signal.
- **IS\_\*(inode)**
  - Tests if inode is on a file system mounted with the corresponding flag.
- **kfree\*()**
  - Frees memory previously allocated with kmalloc()
- **kmalloc()**
  - Allocates a chunk of memory no larger than 4096 bytes.
- **MAJOR()**
  - Reports the major device number for a device.
- **MINOR()**
  - Reports the minor device number for a device.

# kernel functions

- `memcpy_*fs()`
  - Copies chunks of memory between user space and kernel space
- `outb(), outb_p()`
  - Writes a byte to a port. Here, `outb()` goes as fast as it can, while `outb_p()` pauses before returning.
- `printk()`
  - A version of `printf()` for the kernel.
- `put_user*()`
  - Allows a driver to write data in user space.
- `register_*dev()`
  - Registers a device with the kernel.
- `request_irq()`
  - Requests an IRQ from the kernel, and, if successful, installs an IRQ interrupt handler.
- `select_wait()`
  - Adds a process to the proper `select_wait` queue.
- `*sleep_on()`
  - Sleeps on an event, puts a `wait_queue` entry in the list so that the process can be awakened on that event.
- `sti()`
  - Allows interrupts to be acknowledged.
- `sys_get*()`
  - System calls used to get information regarding the process, user, or group.
- `wake_up*()`
  - Wakes up a process that has been put to sleep by the matching `*sleep_on()` function.

---

# Pitfalls

1. **Using standard libraries:** can only use kernel functions, which are the functions you can see in `/proc/ksyms`.
  2. **Disabling interrupts** You might need to do this for a short time and that is OK, but if you don't enable them afterwards, your system will be stuck
  3. Changes from version to version
-

---

# Resources

- Linux Kernel API: <http://kernelnewbies.org/documents/kdoc/kernel-api/linuxkernelapi.html>
  - Kernel development tool <http://www.jungo.com/products.html>
  - Linux Device Drivers 2nd Edition by Rubini & Corbet, O'Reilly Pub, ISBN 0-596-00008-1
-

---

## **LINUX Shell Programming**

# X Windows

---

Jyoti Gajrani

[jyotigairani@gmail.com](mailto:jyotigairani@gmail.com)



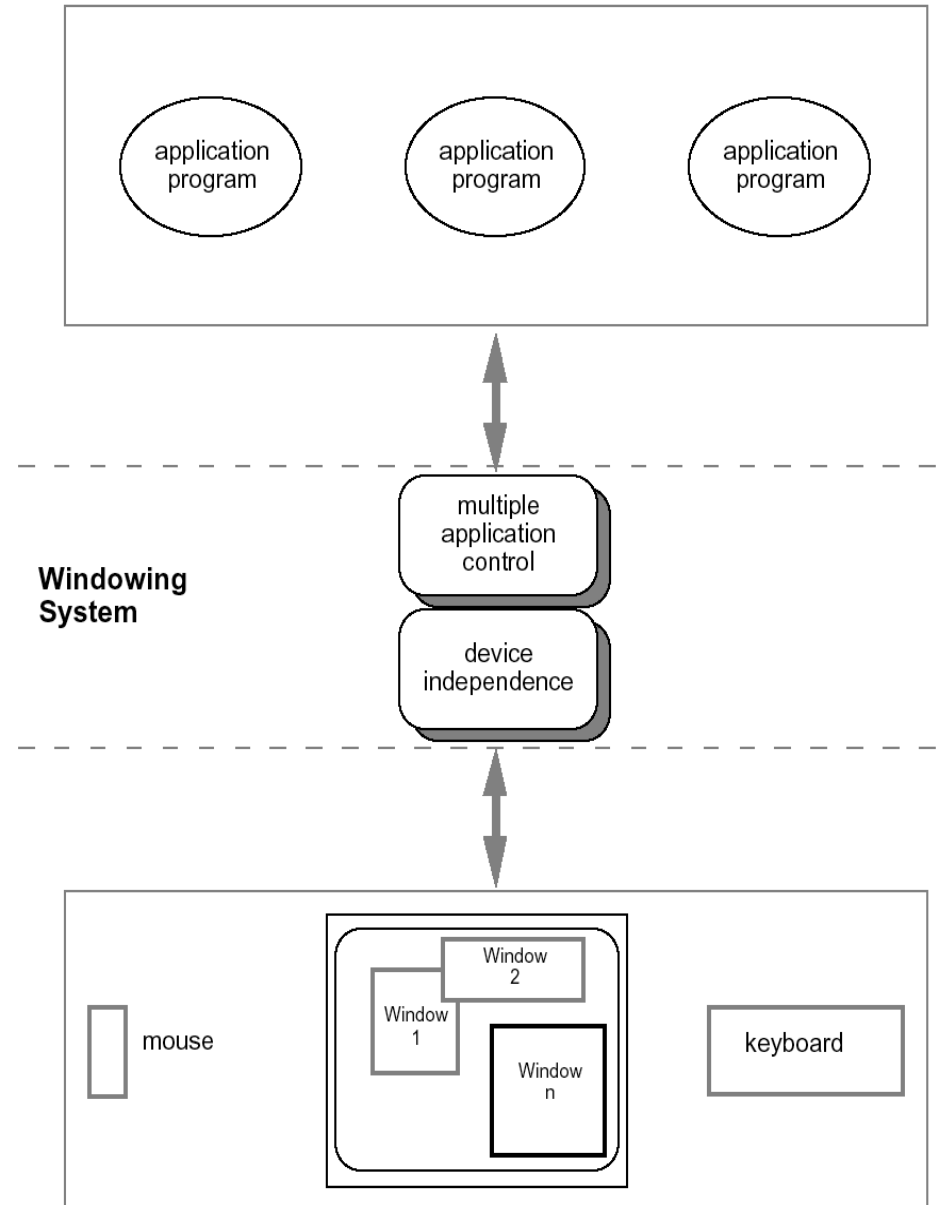
---

# Course topics

- X Windows
  - x-window as client/ server system
  - Concept of window manager, remote computing & local displays,
  - xinitrc file,
  - Customize X work environment and applications
  - Customizing the fvwm window manager.
-

# What is a windowing system ?

A windowing system is a system for sharing a computer's graphical display presentation resources among multiple applications at the same time.



# Architectures of windowing systems

Three possible software architectures

- all assume device driver is separate
- differ in how multiple application management is implemented

1. each application manages all processes

- everyone worries about synchronization
- reduces portability of applications

2. management role within kernel of operating system

- applications tied to operating system

3. management role as separate application

---

maximum portability

---

# X Windows

The [X Windows System](#), also referred to as 'X' or "X11", is the standard graphical engine for Unix and Linux. The X Window System is the basis for graphical user interfaces.

Linux also often confusingly is used to refer to systems like Debian, Ubuntu, Redhat, CentOS, Suse, and many more. These systems are better described as X11+Gnu+Linux.

It is largely OS and hardware independent, it is network-transparent, and it supports many different desktops.

---

---

# X Windows cont'd

X was developed in the mid 80's to provide a standard GUI for Unix systems, similar to Microsoft Windows.

Windows only runs locally on the machine, but X Windows uses the [X Protocol](#) to separate the processing and display for an application. It provides remote display feature.

Unlike Microsoft Windows, X is not part of the operating system. Although the X server used to have extraordinary privileges in order to utilize the graphics hardware

---

# Features of X

---

- **Not part of OS**

It has been said “MS-Windows is a windowing system with an OS stuck on as an after thought, and Unix/Linux is an OS with a windowing system stuck on as an after thought.”

- **Architecture Independent**

Gnu (including Gnu+Linux), Bsd, Solaris, Hp-Ux, etc.

- **Network Transparent**

Run application on a remote (possibly more powerful) machine, and display application locally. This is done on a per application, or per window basis, unlike VNC or remote desktop, that do it a desktop at a time.

- **Policy Free**

X11 has no policy as to what things look like or how things are done. All the changes of look and feel are done by changing or replacing window managers and other helper apps.

---

# Example

X model is really powerful; the classical example of this is running a processor-intensive application on a Cray computer, a database monitor on a Solaris server, an e-mail application on a small BSD mail server, and a visualization program on an SGI server, and then displaying all those on my Linux workstation's screen.

---

# Does X-Window complete GUI?

*X-Window* is a piece of software that allows to draw windows on hardware displays. It only creates, moves and closes windows as well as interprets the mouse events like cursor moving and buttons click. *X-Window* does not provide all that necessary features like nice windows frames, color schemes, graphics effects, sounds etc.

- it is a job for ***Window Manager*** which works on the top of *X-Window*.



---

# Window managers

The [window manager](#) is a special X client that controls the

- placement and movement of applications,
- provides title bars and control buttons etc.

Classic window managers include: kwm, twm, mwm, olwm, fvwm

---

---

# Desktop Environment

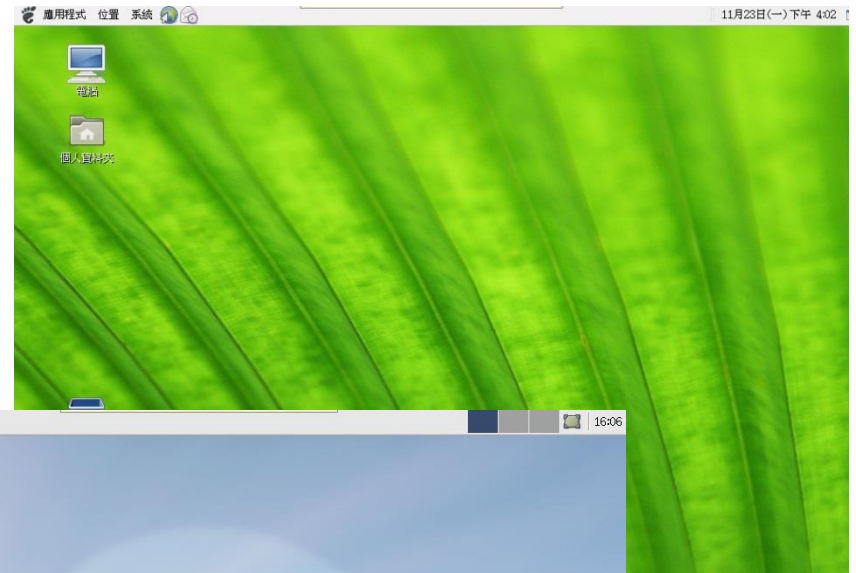
(Some times merged with WM)

- Yet another level running on top of window manager
  - Complete the desktop with:
    - Icon based access to files and directories
    - Overall system menus and toolbars
    - Etc.
  - GNOME, KDE are examples.
-

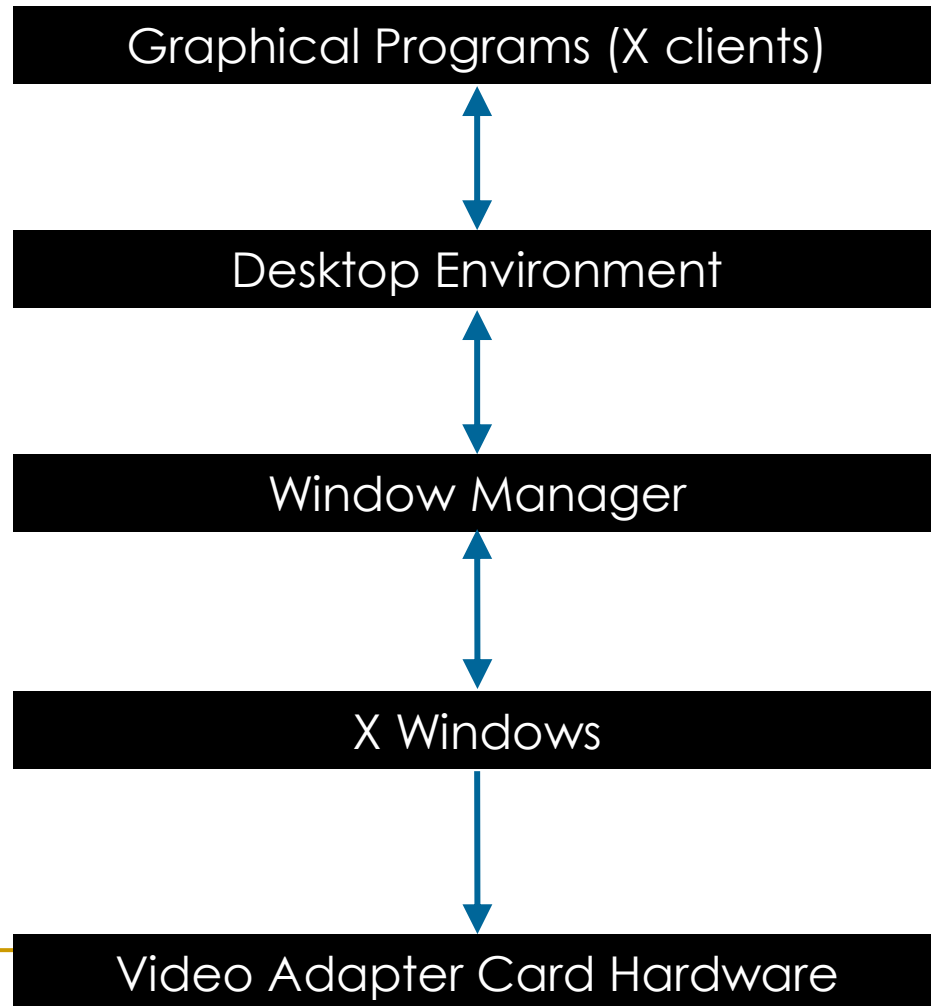
# The Window Manager (2)

## Some Popular Window Managers (+ DE)

- Gnome
- KDE
- Lxde
- Xfce
- Afterstep
- etc..



# Linux GUI Components



---

# X Protocol

The X Protocol provides a client-server architecture at the application level:

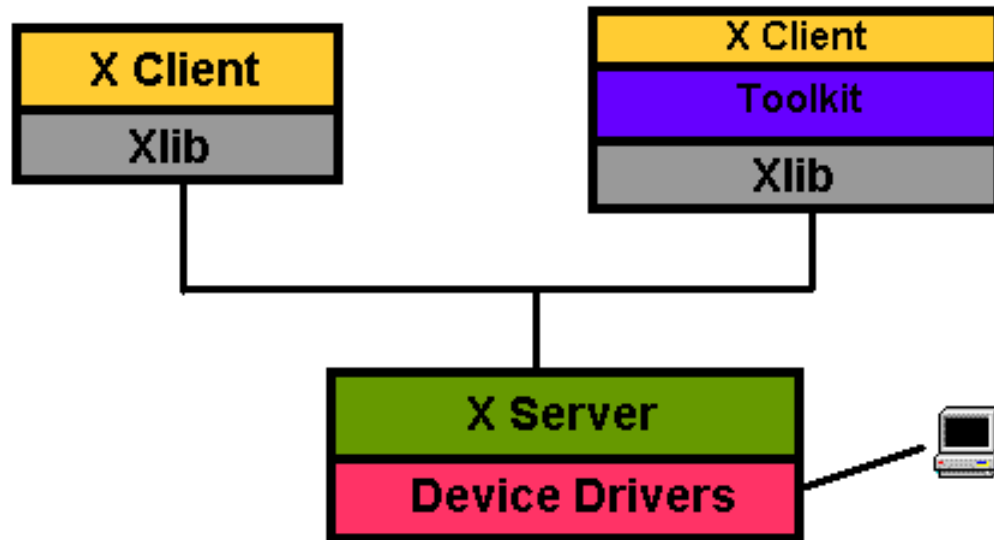
The **X client** is the processing part of the application and may run on a remote machine.

The **X server** is the display and interaction system.

---

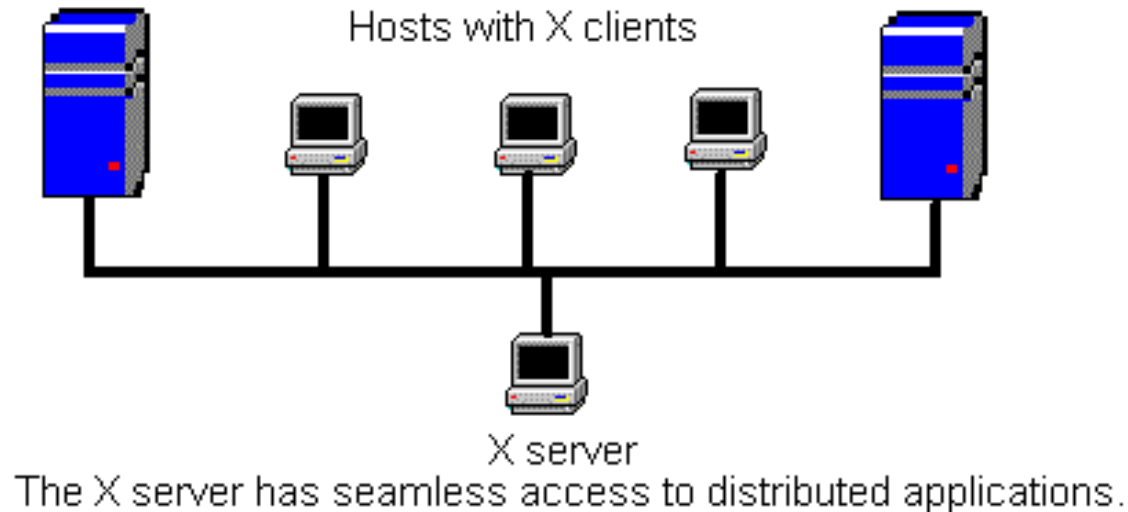
# X Protocol cont'd

The X Protocol is also divided into device dependent and device independent layers.

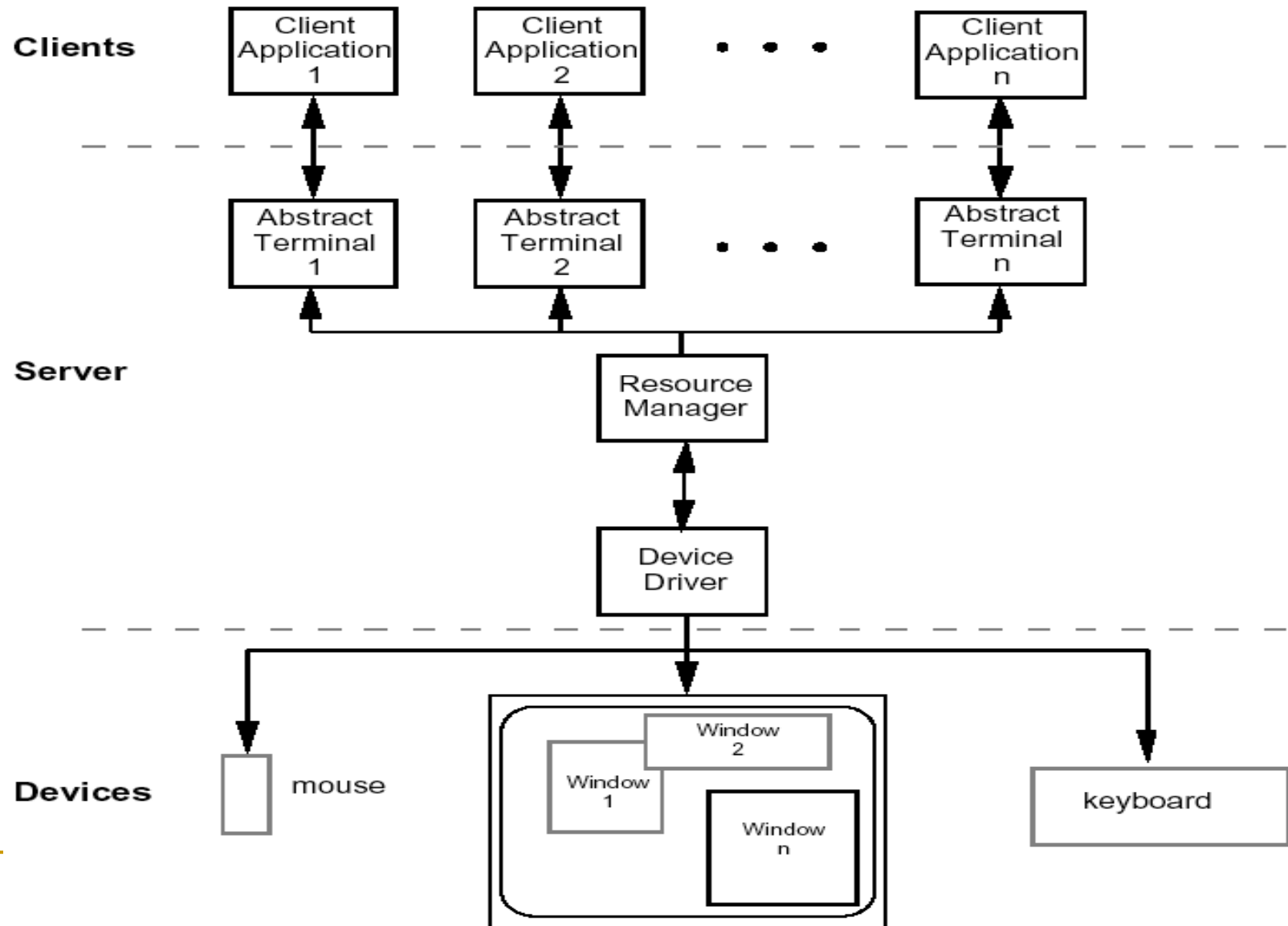


# X architecture

The client-server nature of the X Protocol allows a single X server to support many clients (applications) on several hosts.



# The client-server architecture





---

# X Protocol messages

Requests – client sends requests to the server (e.g. create window)

Replies – server response to client requests

Events – server forwards events (such as mouse clicks or keyboard entry) to the client

Errors – server reports errors to the client

---

---

# Using X Windows

First, an X server must be running:

On a local PC or workstation, you can usually start the X server with the “startx or xinit” command or it may start automatically, presenting a graphical login display.

Historical note: An **X terminal** is a dumb terminal that only runs the X server locally, and always connects to X clients on a remote host.

---

# X Startup Commands

xinit - X Window System initializer

xinit [ [ client ] options ] [ -- [ server ] [ display ] options ]

- Files

- Default client script:

- »~/ .xinitrc

- »/usr/local/lib/X11/xinit/xinitrc

- (run xterm if .xinitrc does not exist)

- Default server script:

- »~/ .xserverrc

- »/usr/local/lib/X11/xinit/xserverrc

- (run X if .xserverrc does not exist)

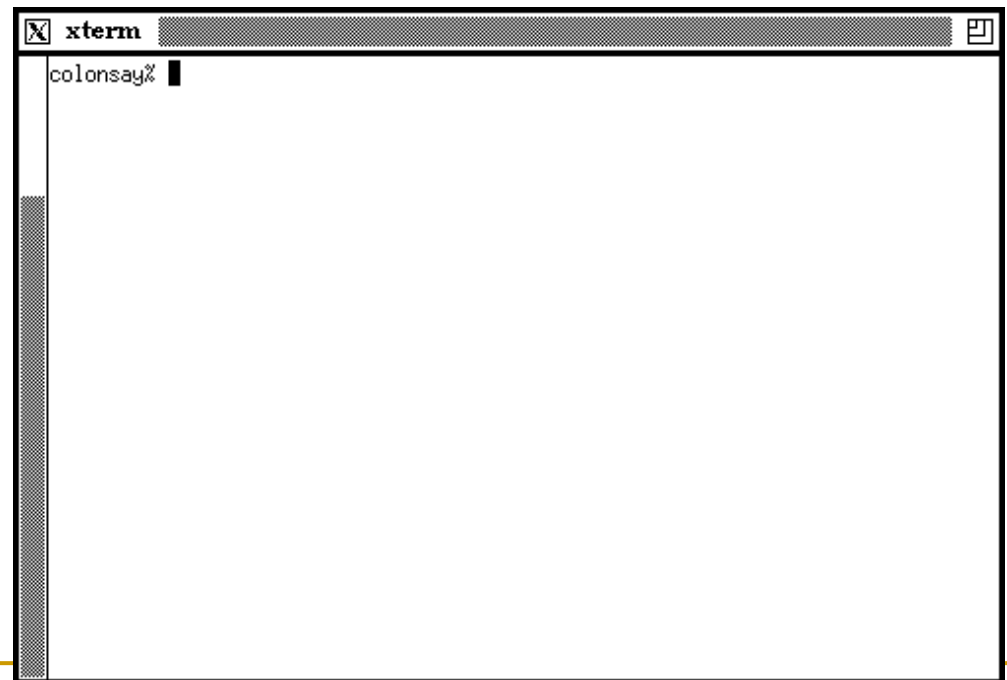
startx:

script to initiate an X session

# xterm

The most important X application is, rather ironically, the terminal program [xterm](#).

**Old school xterm:**



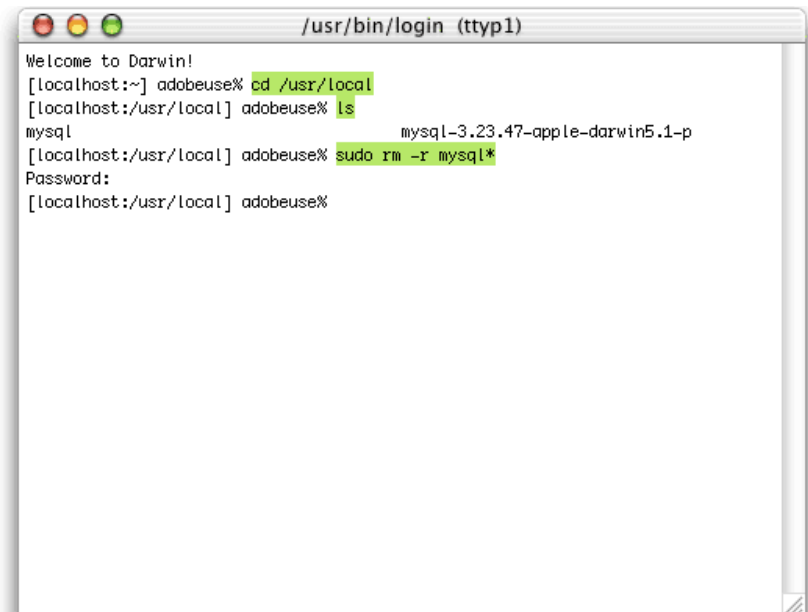
# Xterm cont'd

## Fancy xterm:



```
xterm
autom4te-2.5x.cache  wmcc.sup
autom4te.cache       wmccc.glade
autoscan-2.5x.log    wmccc.glade.bak
cachegrind.out.32386 wmccc.glade~
config.guess         wmcoincoin-2.3.8dtc.tar.gz
config.h             wmcoincoin-2.3.8kikoo.tar.gz
config.h.in          wmcoincoin-2.4.0a.tar.gz
config.h.in.old      wmcoincoin-2.4.0b.tar.gz
config.h.in~         wmcoincoin-2.4.0c.tar.gz
config.log           wmcoincoin-2.4.0cvs.tar.gz
config.rpath         wmcoincoin-2.4.0limule.tar.gz
config.status        wmcoincoin-2.4.1a.tar.gz
config.sub           wmcoincoin.spec
configure            wmpanpan
configure.ac         xpms
configure.ac~
12:03 AM | ~/wmcoincoin/wmcc >
```

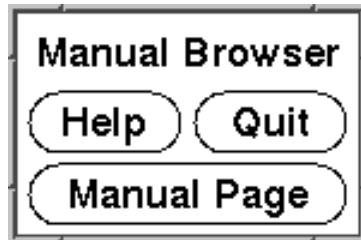
## OS X xterm:



```
/usr/bin/login (ttty1)
Welcome to Darwin!
[localhost:~] adobeuse% cd /usr/local
[localhost:/usr/local] adobeuse% ls
mysql                               mysql-3.23.47-apple-darwin5.1-p
[localhost:/usr/local] adobeuse% sudo rm -r mysql*
Password:
[localhost:/usr/local] adobeuse%
```

# Some classic X apps

xman – manual pages app

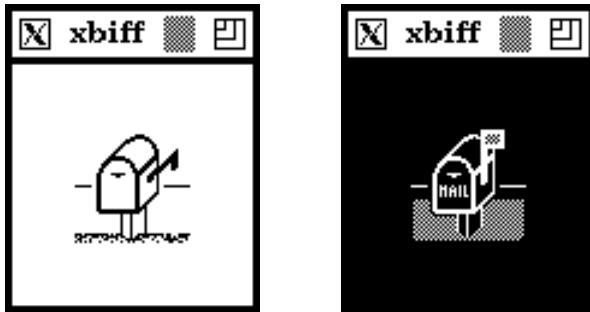


Not this



# Some classic X apps

xbiff – mail notification

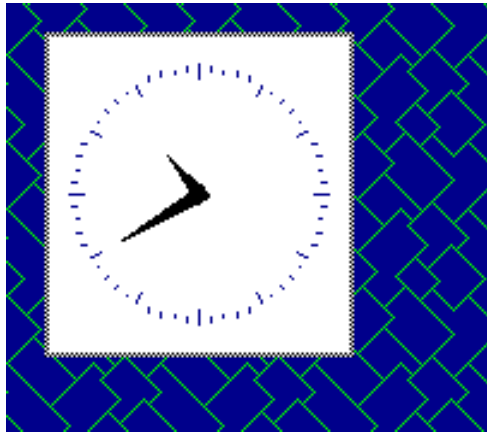


Back when we didn't have mail servers (and we liked it!), mail was stored in local spool files and a process notified the user of new mail. `xbiff` is named after `biff`, which is named after Biff the dog, who barked at the postman. Seriously.

---

# Some classic X apps

xclock – pretty self-explanatory





---

# Starting X applications from remote Host

To launch an X client from a remote host for display on the local X server, you need to set two things:

1) Permission for the remote host to display X clients on the local machine.

`xhost +remotehost`

2) The target display for the remote application.

`setenv DISPLAY=server.display`

---

---

# .xinitrc file

When `startx` or `xinit` is called, it search for `.xinitrc` file in home directory

The file can be modified to customize the X environment.

- If not present then, default file which gets executed
  - `/etc/X11/xinit/xinitrc`

# Customizing X Environment

```
1  #!/bin/sh
2  # Sample .xinitrc shell script
3
4  # Start xterms
5  xterm -geometry 80x40+10+100 -fg black -bg white &
6  xterm -geometry -20+10 -fn 7x13bold -fg darkslategray -bg white &
7  xterm -geometry -20-30 -fn 7x13bold -fg black -bg white &
8
9  # Other useful X clients
10 oclock -geometry 70x70+5+5 &
11 xload -geometry 85x60+85+5 &
12 xbiff -geometry +200+5 &
13 xsetroot -solid darkslateblue &
14
15 # Start the window manager
16 exec fvwm2
```

# Passing arguments to xinit

- `$ xinit gnome-session # start GNOME`
- `$ xinit startkde # start KDE`
- `$ xinit fvwm # start window manager fvwm`
- `$ xinit xterm # start an xterm without a window manager`
  
- `# Here Xfce is kept as default`
- `session=${1:-xfce}`
  
- `case $session in`
- `awesome ) exec awesome;;`
- `bspwm ) exec bspwm;;`
- `catwm ) exec catwm;;`
- `cinnamon ) exec cinnamon-session;;`
- `dwm ) exec dwm;;`
- `enlightenment ) exec enlightenment_start;;`

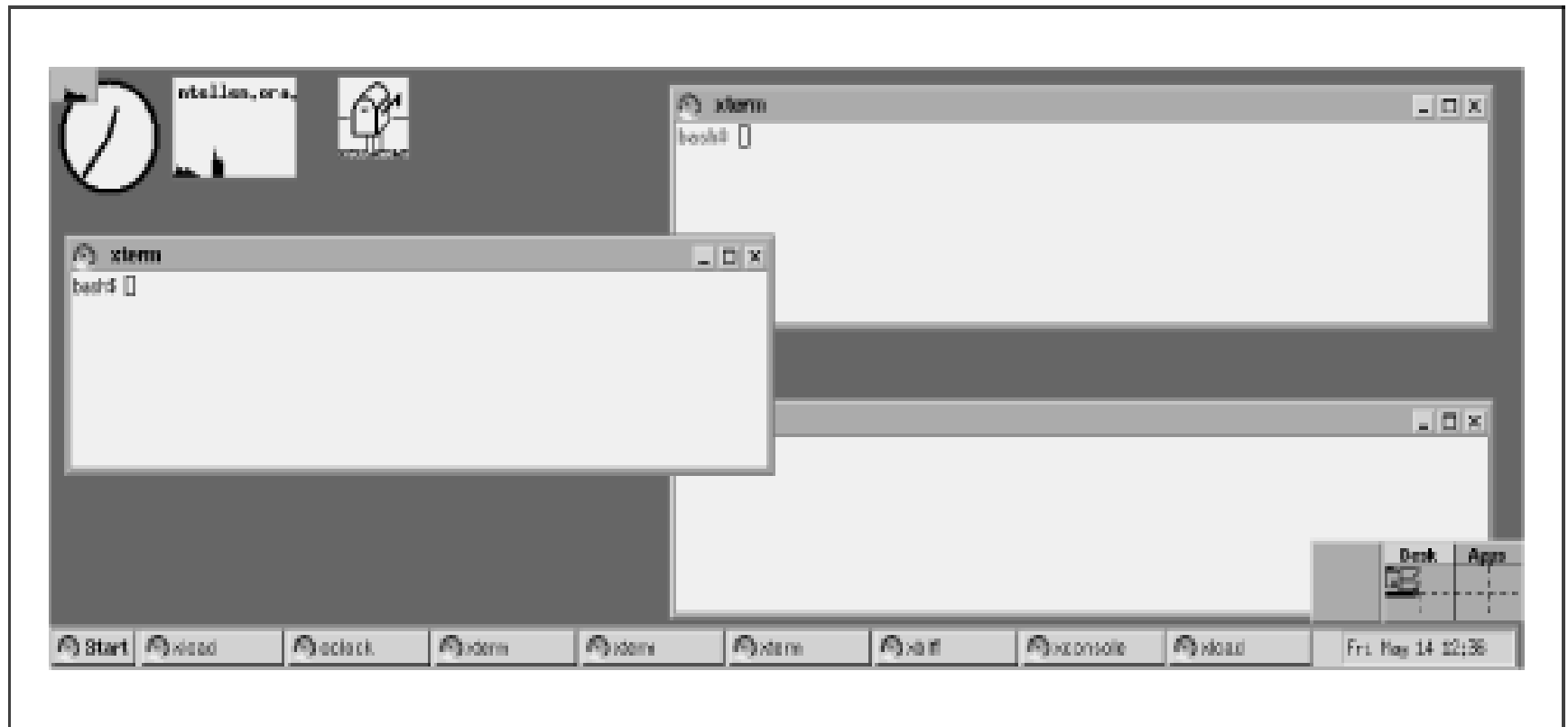
- 
- ede ) exec startede;;
  - fluxbox ) exec startfluxbox;;
  - gnome ) exec gnome-session;;
  - gnome-classic ) exec gnome-session --session=gnome-classic;;
  - i3|i3wm ) exec i3;;
  - icewm ) exec icewm-session;;
  - jwm ) exec jwm;;
  - kde ) exec startkde;;
  - mate ) exec mate-session;;
  - monster|monsterwm ) exec monsterwm;;
  - notion ) exec notion;;
  - openbox ) exec openbox-session;;
  - unity ) exec unity;;
  - xfce|xfce4 ) exec startxfce4;;
  - xmonad ) exec xmonad;;
  - # No known session, try to run it as command
- 
- \*) exec \$1;;
  - esac

---

# FVWM

- F Virtual Windows Manager
  - While KDE and GNOME offer more features, they are also heavy on memory usage.
  - FVWM is light and fast, and you can customize it to meet your needs, and apply these customizations throughout your organization.
  - The default FVWM screen is very basic -- just a simple blue desktop.
  - Clicking anywhere with the left mouse button brings up a menu with a couple of built-in options, including xterm.
-

# FVWM



---

# FVWM FEATURES

- Automatic desktop scrolling when the pointer reaches the screen boundary.
  - 3D look and feel for window frames.
  - Fully configurable desktop menus, which appear when you press the mouse buttons.
  - A keyboard equivalent for almost every mouse-based feature; this is helpful when using X on laptops without a mouse or trackball
-



---

# FVWM Configuration

FVWM uses the file `~/.fvwm/.fvwm2rc` to override its default configuration.

Mouse 0 1 A Iconify

Mouse 0 2 A Maximize 100 100



maximize and minimize  
button for all windows

Style "\*" NoIcon //switch off the icons for minimised applications

EdgeResistance 1000 0 // 1 sec to switch between windows

---

---

# create a task bar at the bottom of the screen without functionality

Style "FvwmTaskBar" NoTitle, BorderWidth 0, HandleWidth 0,  
Sticky

AddToFunc InitFunction I Module FvwmTaskBar

AddToFunc RestartFunction I Module FvwmTaskBar

---

# Adding functionality to taskbar

```
AddToMenu "Internet" "Internet" Title  
+ "Firefox%mini-x.xpm%" Exec firefox &  
+ "Thunderbird%mini-x.xpm%" Exec thunderbird &
```

```
AddToMenu "Main" "Main" Title  
+ "xterm%mini-x.xpm%" Exec xterm &  
+ "Internet%mini-x2.xpm%" Popup Internet  
+ "Restart%mini-turn.xpm%" Restart  
+ "Quit%mini-exclam.xpm%" Quit
```

```
*FvwmTaskBarStartMenu Main
```

# Adding functionality dynamically to taskbar

AddToMenu ListFiles ListFiles Title

```
Piperead `for f in ~/*.html; do echo "+ $f Exec gedit $f"; done`
```

---

These bindings cause Ctrl-arrowkey to scroll the desktop by a full page in the given direction.

Key Up	A	C	Scroll +0	-100
Key Down	A	C	Scroll +0	+100
Key Left	A	C	Scroll -100	0
Key Right	A	C	Scroll +100	+0