Embedded Linux

Why need an OS?

- Multitasking
- Networking
- Graphics
- File management

OS for Embedded systems

- Linux
- Windows
  - IOT
  - CE
- RTOS
  - Eg. - vxworks, FreeRTOS, Chibi, Nuttx

Why Linux?

- Free software
- Open source
- Supports several embedded hardware

SMEClabs

Hardware supported by Linux

- x86 and x86-64, as found on PC platforms, but also embedded systems (multimedia, industrial)
- ARM, with hundreds of different SoC (multimedia, industrial)
- PowerPC (automotive, industrial applications)
- MIPS (mainly networking applications)
- SuperH (mainly set top box and multimedia applications)
- Blackfin (DSP architecture)
- Microblaze (soft-core for Xilinx FPGA)
- Coldfire, SCore, Tile, Xtensa, Cris, FRV, AVR32, M32R

History of Linux

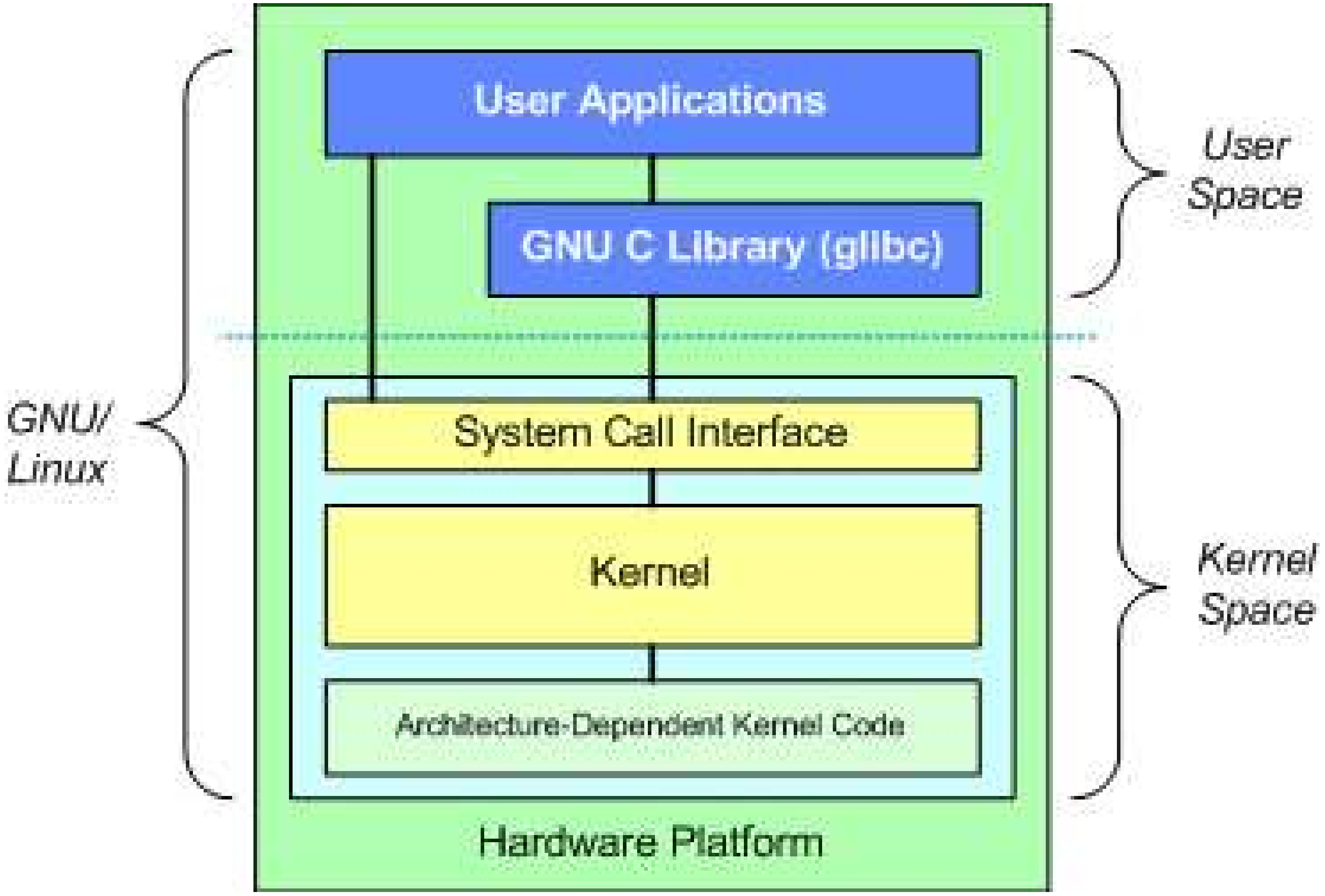
- 1960s, Massachusetts Institute of Technology (MIT) and a host of companies developed an experimental operating system called **Multics** for GE-645.
- One of the developers of this operating system, AT&T, dropped out of Multics and developed their own operating system in 1970 called **Unics**. (C was developed for this.)
- Twenty years later, Andrew Tanenbaum created a microkernel version of UNIX®, called **MINIX** (for minimal UNIX), that ran on small personal computers.
- This open source operating system inspired Linus Torvalds' initial development of **Linux** in the early 1990s.

SMEClabs

Linux based Distributions

- Android
- Angstrom
- Debian (GNU/Linux)
- Fedora
- ArchLinux
- Buildroot
- Gentoo
- Nerves Erlang/OTP
- Sabayon
- Ubuntu
- Yocto

Architecture of GNU/Linux



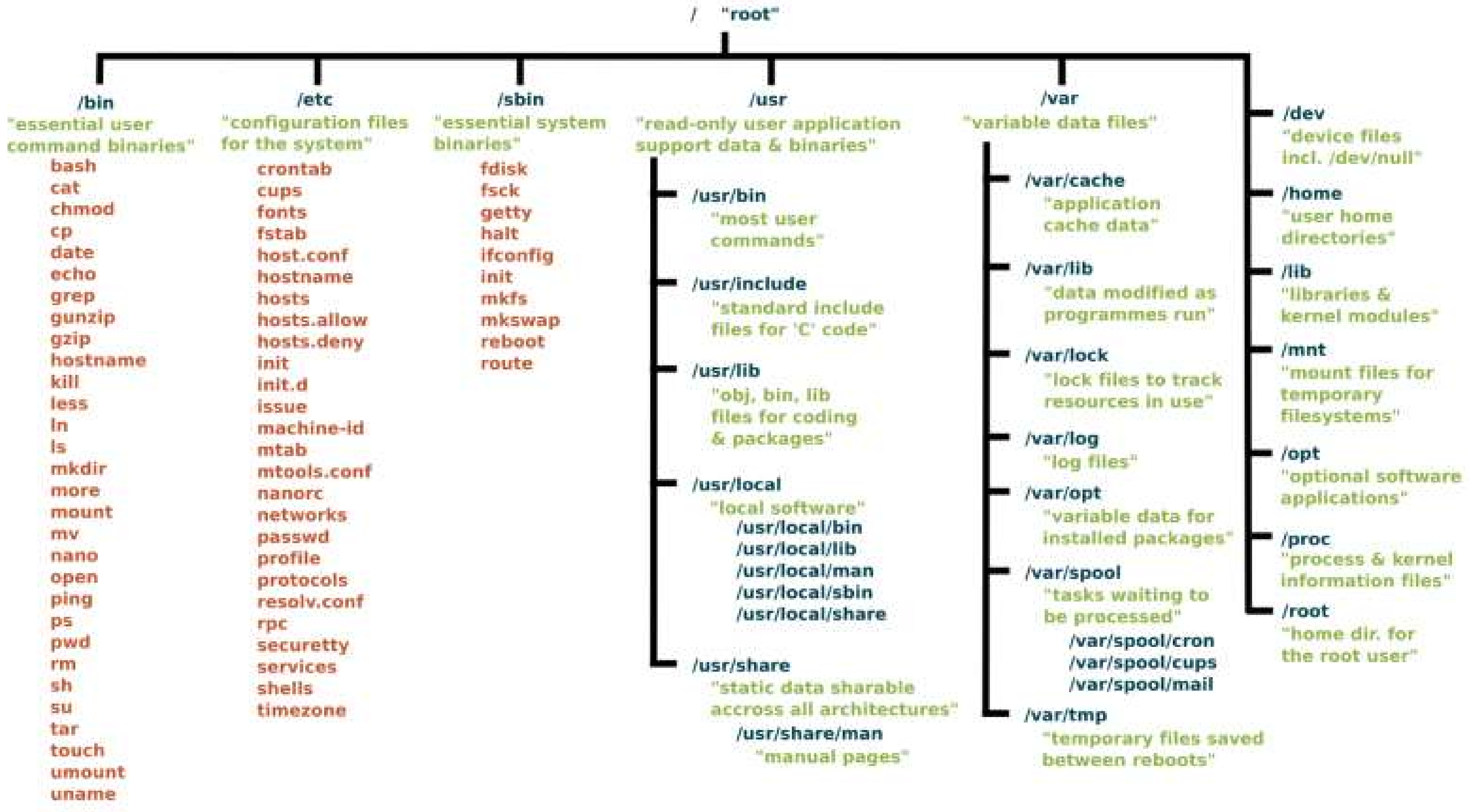
Architecture of GNU/Linux

- User applications – Eg. A program to print hello world
- C library - Implements functions like printf
- System call – eg. Open(), read(), write() ..
- Kernel – architecture independent code
- Hardware dependent code - BSP

SMEClabs

Interacting with Linux

Directory Structure



Linux shell

- A shell is a user interface for access to an operating system's services.
- Shells are actually special applications that use the kernel API in just the same way as it is used by other application programs.
- In general, operating system shells use either a command-line interface (CLI) or graphical user interface (GUI), depending on a computer's role and particular operation. It is named a shell because it is the outermost layer around the operating system kernel.
- Graphical shells are typically build on top of a windowing system. In the case of X Window System or Wayland, the shell consists of an X window manager or a Wayland compositor, respectively, as well as of one or multiple programs providing the functionality to start installed applications, to manage open windows and virtual desktops, and often to support a widget engine.
- CLI shells require the user to be familiar with commands and their calling syntax, and to understand concepts about the shell-specific scripting language (for example bash script).

SMEClabs

Linux CLI shell

- There are two major types of shells −
  - Bourne shell −($ prompt) – Eg. Bourne shell (sh), Almquist shell (ash), Bourne-Again shell (bash), Korn shell (ksh), Z shell (zsh)
  - C shell − ($ prompt) – Eg. C shell (csh), TENEX/TOPS C shell (tcsh)
- On most Linux systems bash (an enhanced version of the original Unix shell program, sh) acts as the shell program.
- Shell is often accessed using a terminal emulator. This is a program that opens a window and lets you interact with the shell. There are a bunch of different terminal emulators you can use. Most Linux distributions supply several, such as: gnome-terminal, konsole, xterm, rxvt, kvt, nxterm, and eterm.

SMEClabs

Shell Commands

- |
- &
- >
- >>
- man
- info
- pwd
- cd
- cd ..
- ls
- more
- less
- cat
- cp

- cp
- mv
- mkdir
- rmdir
- locate
- find
- ps
- clear
- more
- less
- grep
- su
- sudo
- chmod

- Refer printout

Environment Variable

- Environmental variables are variables that are defined for the current shell and are inherited by any child shells or processes.
- Shell variables are variables that are contained exclusively within the shell in which they were set or defined.
- VAR=value - to create a shell variable
- export VAR   - create and environmental variable
- export –n VAR - downgrade to shell variable
- unset VAR   - delete variable from shell and environment
- $ - print value of variable (try echo $VAR)
- printenv   - print only environmental variables
- set   - prints all environmental and shell variables

- Refer printout

SMEClabs

Text Editors

- Vi/Vim
- Gedit
- Emacs
- Nano
  - To open/create -- sudo nano filename
  - To save -- ctrl+s
  - To exit -- ctrl+x

Shell Scripts

- The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments, preceded by # sign, describing the steps.
- There are conditional tests, such as value A is greater than value B, loops allowing us to go through massive amounts of data, files to read and store data, and variables to read and store data, and the script may include functions.
- Shell scripts are interpreted. This means they are not compiled.
- Assume we create a test.sh script. Note all the scripts would have the .sh extension. Before you add anything else to your script, you need to alert the system that a shell script is being started. This is done using the shebang construct. For example −
- #!/bin/sh This tells the system that the commands that follow are to be executed by the Bourne shell. It's called a shebang because the # symbol is called a hash, and the ! symbol is called a bang.

```
#!/bin/bash
pwd
ls
```

Shell Script Example

```
#!/bin/bash
# Author : Paul Xavier
# Script follows here:
echo "What is your name?"
read PERSON
echo "Hello", $PERSON
```

- Save the above content and make the script executable
- ls –l filename to check file permissions
- sudo chmod +x test.sh The shell script is now ready to be executed
- ls –l filename see whats the difference
- ./test.sh

Task

- Write a shell script to print the name entered 10 times using for loop and while loop

User application simple

```
#include<stdio.h>
void main()
{
printf(“hiiii”);
 }
```

SMEClabs

Compiling simple application

- Save the usr file as for example test.c
- Install gcc if not done
  - sudo apt-get install build-essential
- cc test.c for compiling
- a.out executable will be generated
- ./a.out to see the output
- To generate executable in another name use

  cc –o test test.c
- ./test

User application complex

hellomake.c

```
#include<hellomake.h>
int main()
{
//call a function in another file
myPrintHelloMake();
return(0);
 }
```

hellofunc.c

```
#include<stdio.h>
#include<hellomake.h>
void
   myPrintHelloMake(void)
{
printf("Hello makefiles!\n");
return;
}
```

hellomake.h

```
/* example include file */
void myPrintHelloMake(void);
```

Compiling complex application

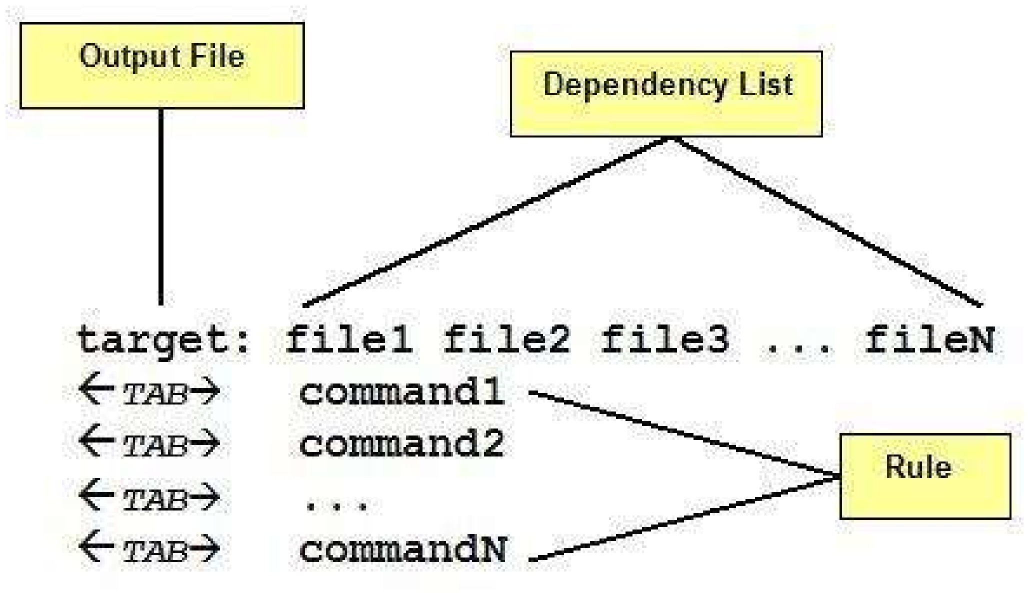
```
#create object file. –I. is to search for include file in .
cc –I. -c -o hellomake.o hellomake.c

#create object file  -c only compiles no linking
cc –I. -c –o hellofunc.o hellofunc.c

#link object files
cc –I. -o hellomake hellomake.o  hellofunc.o
```

- If one of the file changes this procedure has to be repeated

Makefiles



- When a project gets too complex with many source files it becomes necessary to have a tool that allows the developer to manage the project.

- A make file consists of a set of **targets**, **dependencies** and **rules**. A **target** most of time is a file to be created/updated. **target** depends upon a set of source files or even others **targets** described in **Dependency List**. **Rules** are the necessary commands to create the **target** file by using **Dependency List**.
- The **makefile** is read by **make** command which determines **target** files to be built by comparing the dates and times (timestamp) of source files in **Dependency List**. If any dependency has a changed timestamp since the last build **make** command will execute the rule associated with the **target**.

Makefiles

- Only those files included in the dependency list will be checked for changes when make is called.
- The dependency list should also contain header files, otherwise even if header files changes it wont be recompiled.
- The name of the makefile should be either **makefile** or **Makefile**. If any other name is used use **make –f** ***filename***
- The main target(final target) should be the first target.
- Sometimes a **target** does not mean a **file** but it might represent an action to be performed. When a **target** is not related to a file it is called **phony target. Eg. all, install, clean**

Makefiles

- We can define a **MACRO** by writing:

  MACRONAME=value

  and access the value of MACRONAME by writing either

  $(MACRONAME) or ${MACRONAME}.

  Eg. CC=gcc
- Sometimes special characters (**-** and **@**) preceding the commands are used.
- **-** tells **make** to continue processing even an error occurs
- **@** tells **make** not to print **command** before executing it.

Makefiles – example makefile

```
# makefile sample that uses phony targets and macros

# *** MACROS(cc is compiler and –I. is to search for include file in .)
CC=cc
CFLAGS=-I.

# *** Targets
hellomake: hellomake.o hellofunc.o
	$(CC) $(CFLAGS) -o hellomake hellomake.o  hellofunc.o

#-c will only compile not link and –o is for specifying output file other than a.out
hellomake.o: hellomake.c hellomake.h
	$(CC) $(CFLAGS) -c -o hellomake.o hellomake.c

hellofunc.o: hellofunc.c hellomake.h
	$(CC) $(CFLAGS) -c -o hellofunc.o hellofunc.c

# This one is used for housekepping(-f is to remove forcefully)
clean:
	rm -f hellomake
	-@rm -f ./*.o
```

Task

- Create a phony target “run” which executes the generated file

Loadable Kernel Module

LKM

- A loadable kernel module (LKM) is a mechanism for adding code to, or removing code from, the Linux kernel at run time. They are ideal for device drivers, enabling the kernel to communicate with the hardware without it having to know how the hardware works. The alternative to LKMs would be to build the code for each and every driver into the Linux kernel.
- Without this modular capability, the Linux kernel would be very large, as it would have to support every driver that would ever be needed on the BBB. You would also have to rebuild the kernel every time you wanted to add new hardware or update a device driver. The downside of LKMs is that driver files have to be maintained for each device. LKMs are loaded at run time, but they do not execute in user space — they are essentially part of the kernel.

SMEClabs

example module

```
//Filename: /home/paul/paul.driverdemo/pauldriver.c
#include<linux/kernel.h>
#include<linux/init.h>
#include<linux/module.h>

static int __init hello_world_init(void)
{
   printk(KERN_INFO "Welcome to SMEClabs\n");
      printk(KERN_INFO "This is the Simple Module\n");
      printk(KERN_INFO "Kernel Module Inserted Successfully...\n");
   return 0;
}

void __exit hello_world_exit(void)
{
   printk(KERN_INFO "Kernel Module Removed Successfully...\n");
}

module_init(hello_world_init);
module_exit(hello_world_exit);

MODULE_LICENSE("GPL");
MODULE_LICENSE("GPL v2");
MODULE_LICENSE("Dual BSD/GPL");
MODULE_AUTHOR("Paul xavier");
MODULE_DESCRIPTION("A simple hello world driver");
MODULE_VERSION("2:1.0");
```

example Makefile

```
//Filename: /home/paul/paul.driverdemo/Makefile

obj-m += pauldriver.o
KDIR = /lib/modules/$(shell uname -r)/build

all:
        make -C $(KDIR)  M=$(shell pwd) modules
clean:
        make -C $(KDIR)  M=$(shell pwd) clean
```

SMEClabs

LKM Makefile Explanation

- The syntax **obj-m** is used to tell the makefile that the driver needs to be compiled as module using the specified object file.
- When you just run command **make** then the control comes to the **all:** section of the Makefile and if you run command **make clean** then the control goes to the **clean:** section of Makefile.
- From this Makefile we are actually running make inside the kernel source directory using option -C. Please make sure you have kernel source directory installed in your system. we use command **uname -r** to find the current version of your system's linux kernel.
- We have used option M=$(PWD) to indicate in the kernel makefile that the source for driver is in present working directory and we are specifying the word **modules** to tell kernel makefile to just build modules and not to build the complete kernel source code.
- In **clean:** section of Makefile we are telling kernel makefile to clean the object files generated to build this module.

SMEClabs

compiling steps

- sudo apt-get update
- sudo apt-get install linux-headers-$(uname -r)
- sudo apt install libelf-dev
- sudo make
- sudo insmod pauldriver.ko
- lsmod
- dmesg
- modinfo pauldriver.ko
- sudo rmmod pauldriver.ko
- dmesg

Task

- Modify the Makefile to include a phony target “load” to insert the module and another target “unload” to remove the module

SMEClabs

Device Driver

Device Driver

- One class of kernal module is the device driver, which provides functionality for external hardware like a usb device or a serial port.
- On Linux, each piece of hardware is represented by a file located in /dev named a device file which provides the means to communicate with the hardware.
- The device driver provides the communication on behalf of a user program.
- A user space program like mp3blaster can use /dev/sound without ever knowing what kind of sound card is installed

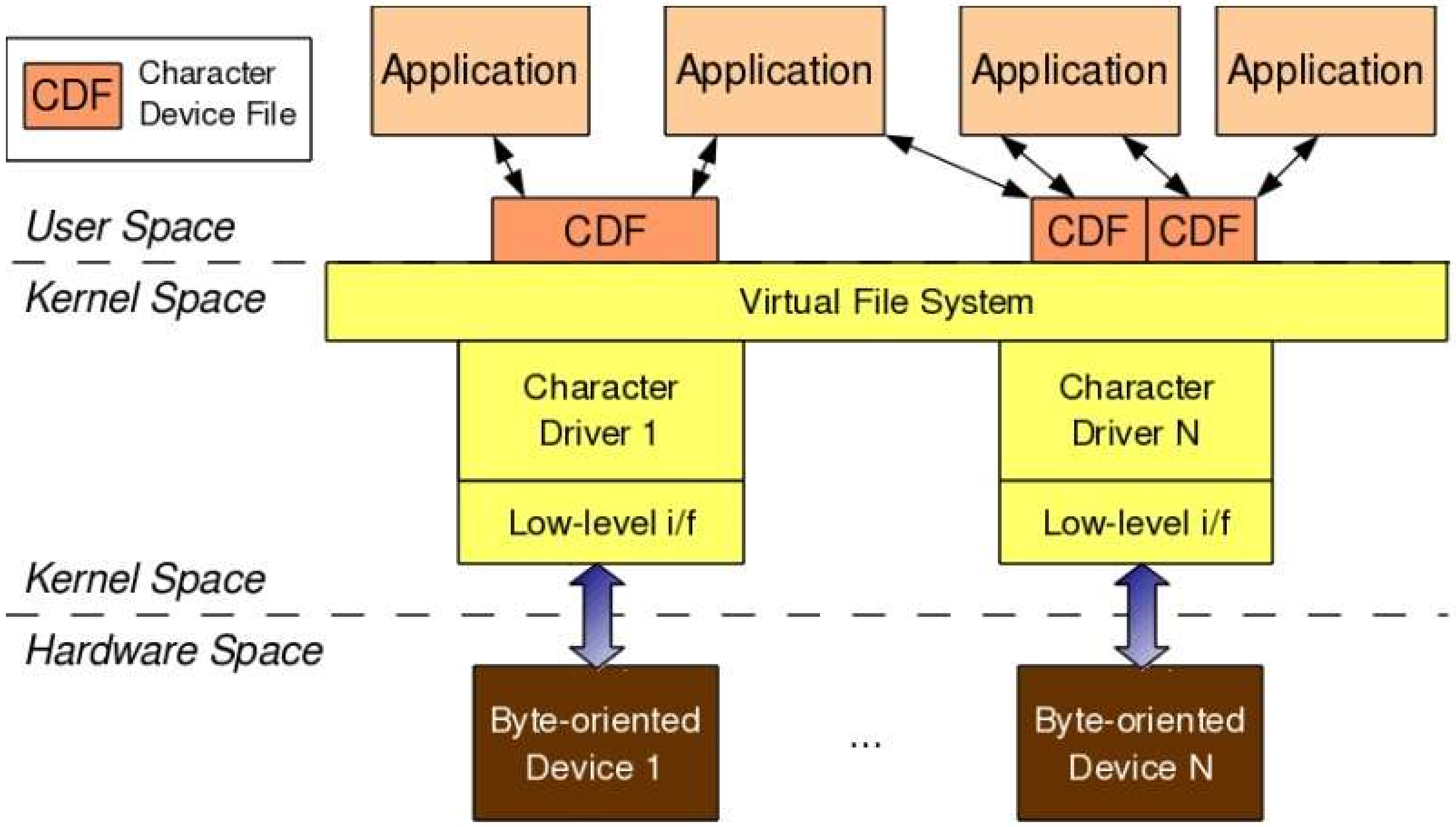
Device(Driver) types

- A ***character device*** typically transfers data to and from a user application — they behave like pipes or serial ports, instantly reading or writing the byte data in a character-by-character stream. They provide the framework for many typical drivers, such as those that are required for interfacing to serial communications, video capture, and audio devices.
- The main alternative to a character device is a ***block device***. Block devices behave in a similar fashion to regular files, allowing a buffered array of cached data to be viewed or manipulated with operations such as reads, writes, and seeks. Both device types can be accessed through device files that are attached to the file system tree.

SMEClabs

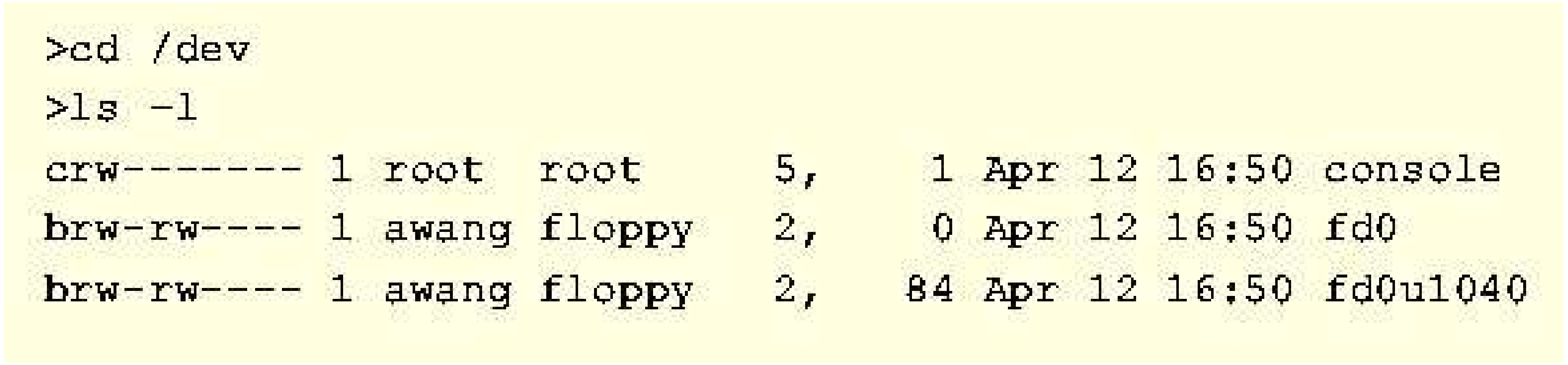
Character Device Driver

Character Device Driver



Major and minor numbers

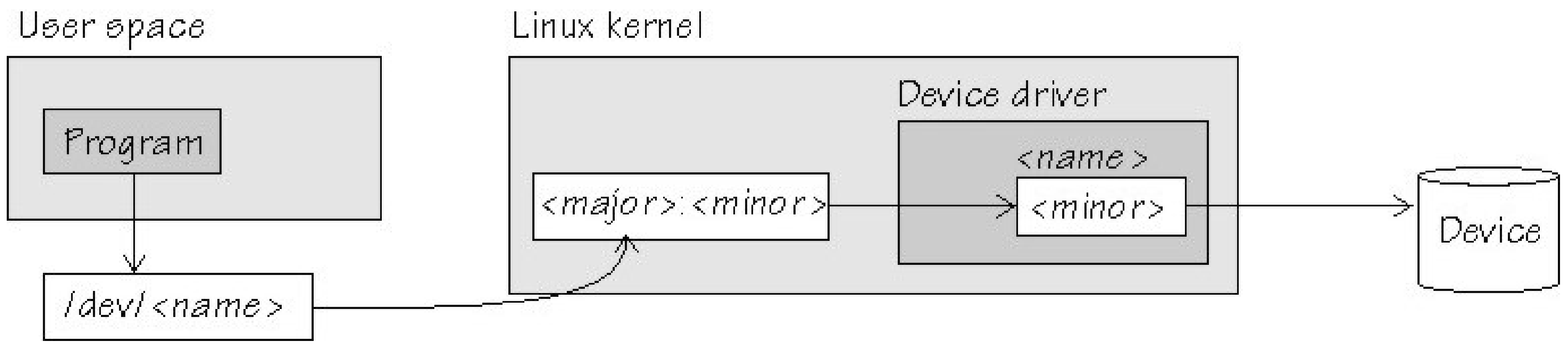
- Access to device driver from user space is through device file
- Kernel needs to know to which driver and which device the device file belongs
- Device files are mapped by the kernel to a major and a minor number



```
>cd /dev
>ls -l
crw------- 1 root  root      5,     1 Apr 12 16:50 console
brw-rw---- 1 awang floppy    2,     0 Apr 12 16:50 fd0
brw-rw---- 1 awang floppy    2,    84 Apr 12 16:50 fd0u1040
```

Major and minor numbers

- Major number refers to the driver, each driver has its own
- Minor number refers to the device which is managed by the driver



SMEClabs

Thank You