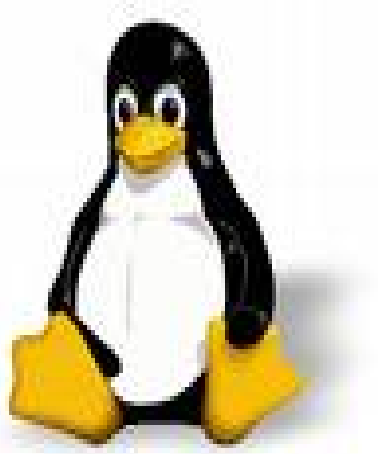


Linux Kernel Programming

Santosh Sam Koshy

JD, C-DAC Hyderabad

santoshk@cdac.in

Agenda

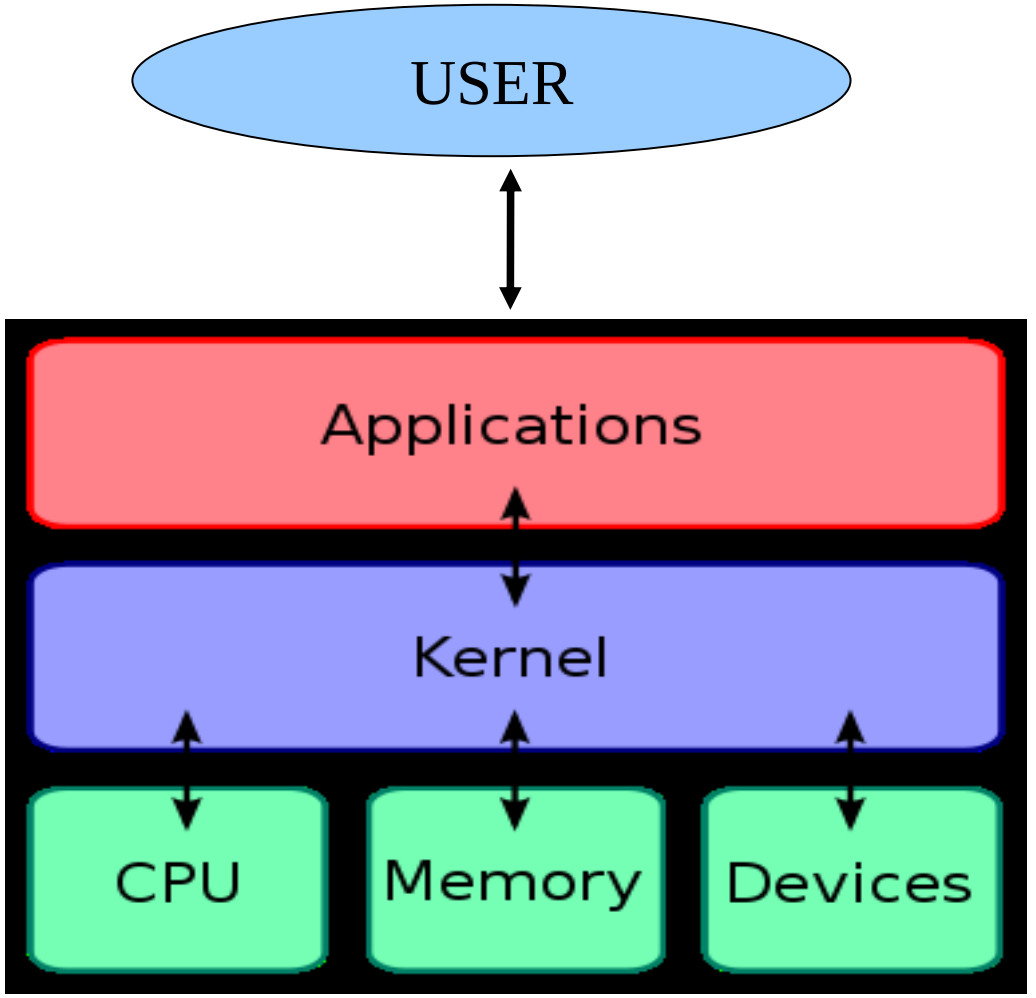
- Kernel Classifications
- The Linux Kernel
- Module Programming
- References

Kernels & Classifications

What is the KERNEL?

- Central component of most computer operating systems
- Primary Purpose is to
  - Manage Resources - Allow multiple process to use the resources
  - Provide services to various processes
- Resources consist of:
  - Central processing unit (CPU)
  - Computer's memory
  - Input / Output devices

Kernel Layout



Implementation detail

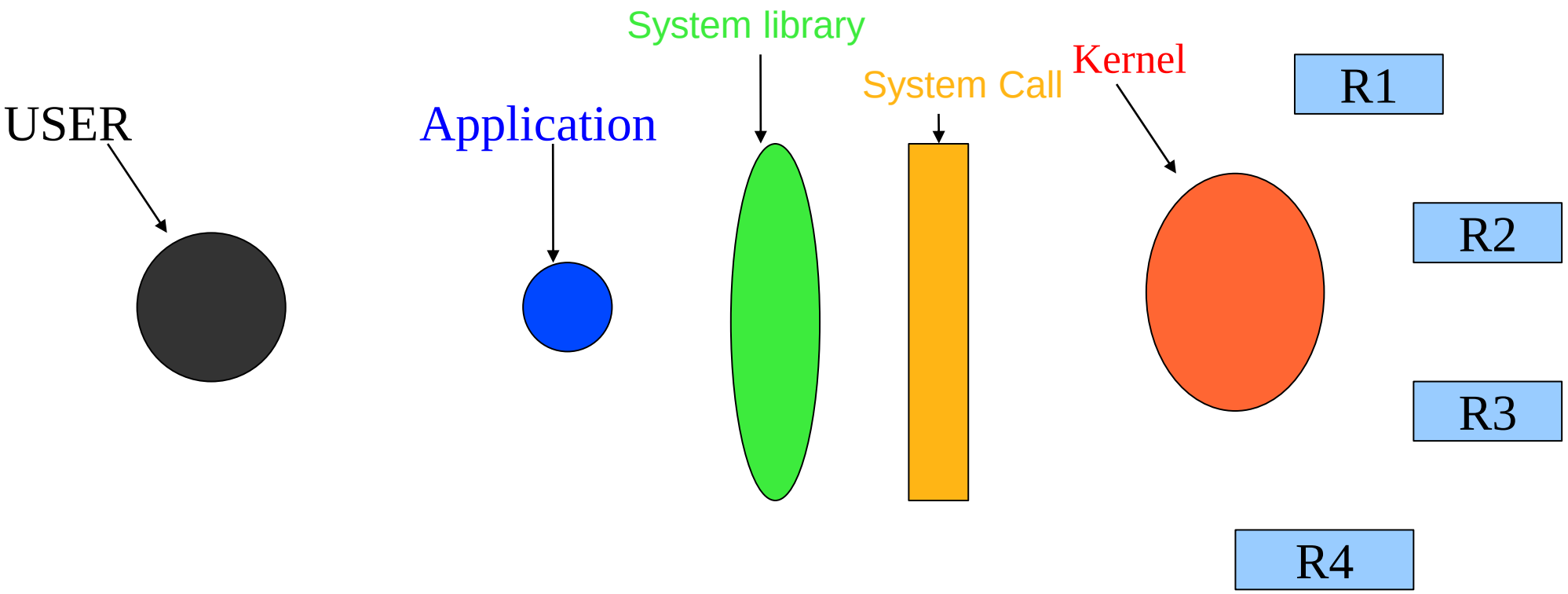
Abstraction

*reference: wikipedia

Kernel Services

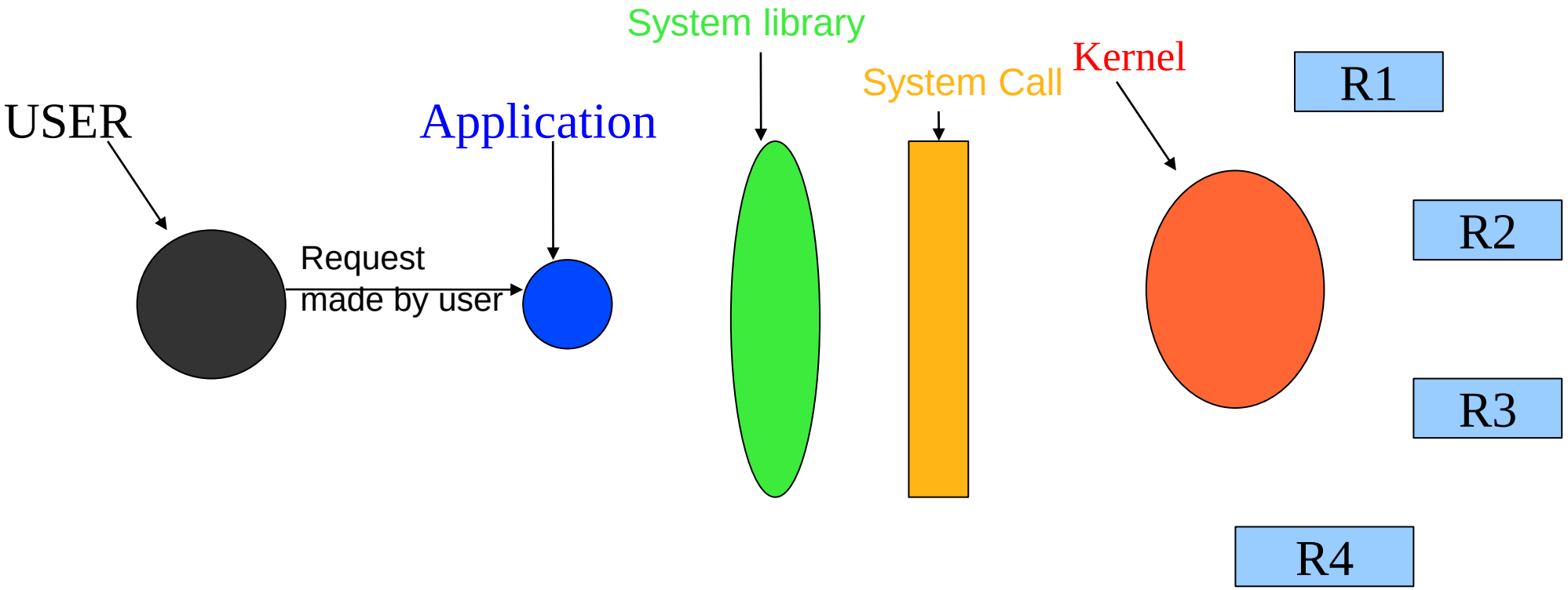
- The whole kernel entity can be split into following different units according to their roles:
  - Process management
  - Memory management
  - File systems
  - Device control
  - Networking
- Seperate Entities, working together.....

Kernel Explanation



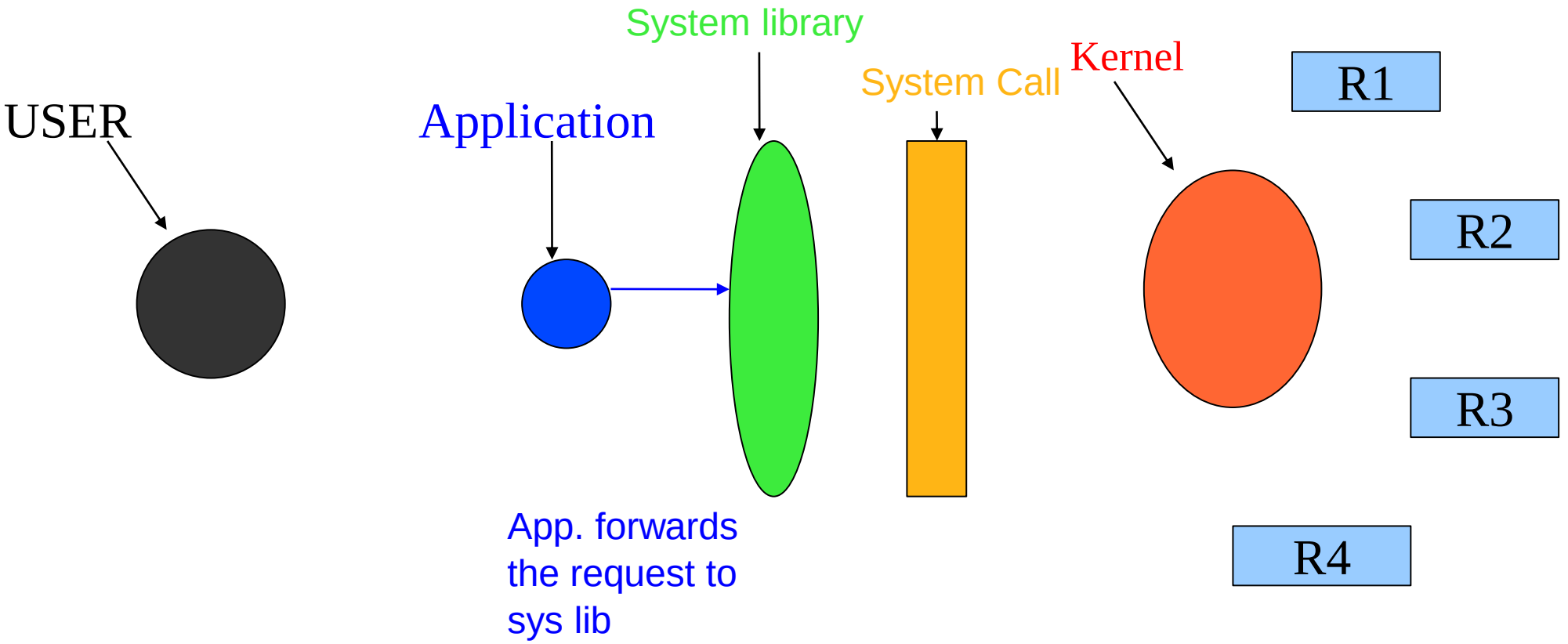
R1, R2, R3 and R4 are 4 different resources

Kernel Explanation contd..



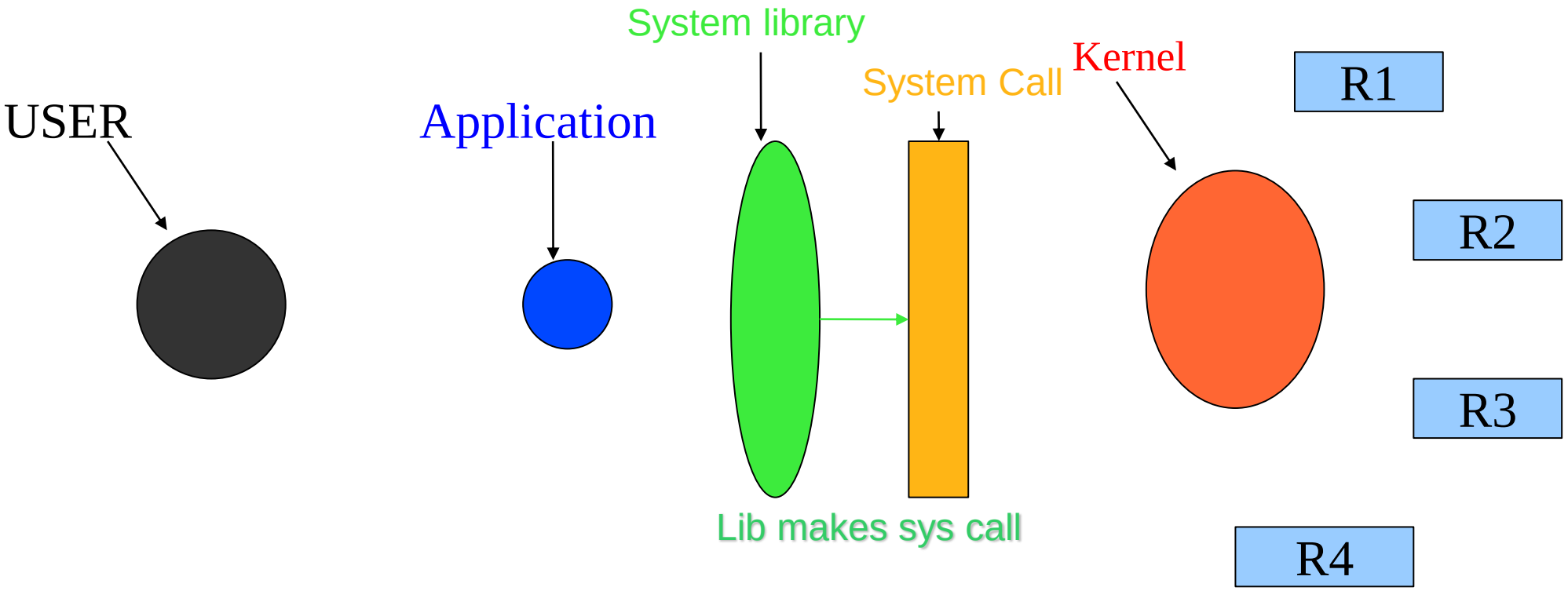
R1, R2, R3 and R4 are 4 different resources

Kernel Explanation contd..



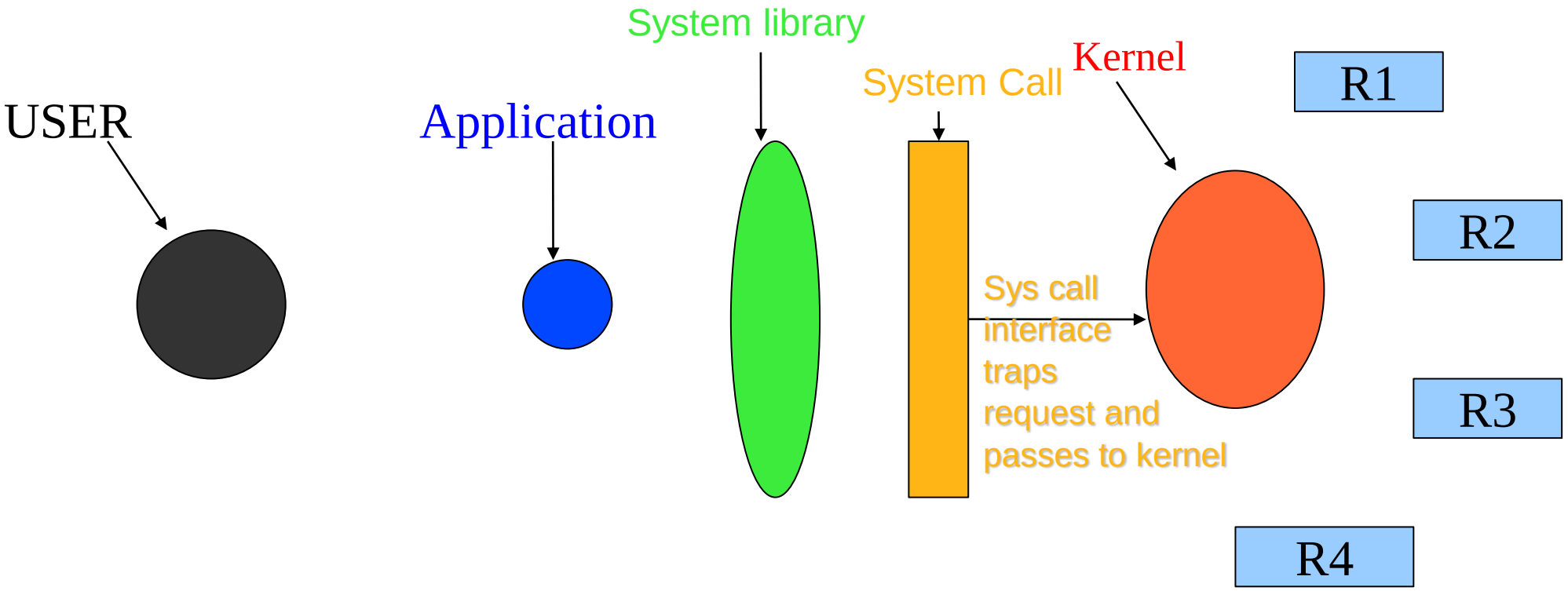
R1, R2, R3 and R4 are 4 different resources

Kernel Explanation contd..



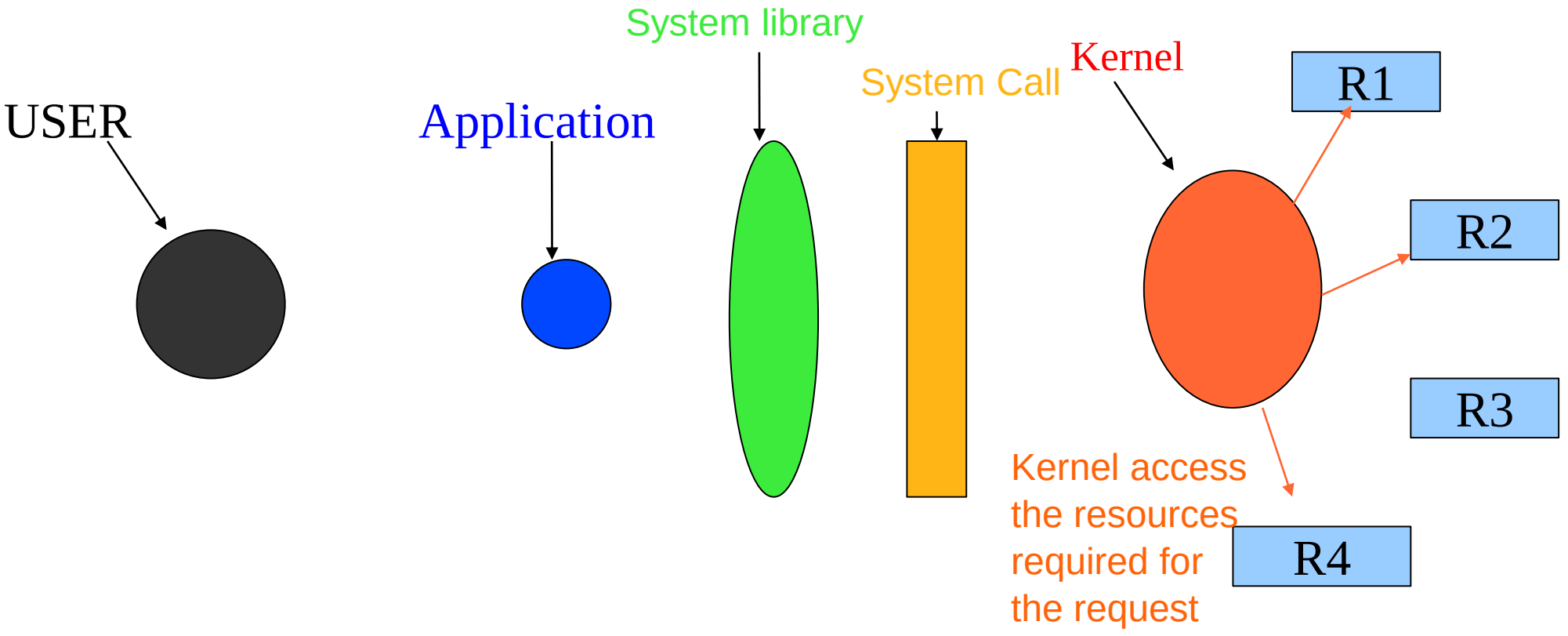
R1, R2, R3 and R4 are 4 different resources

Kernel Explanation contd..



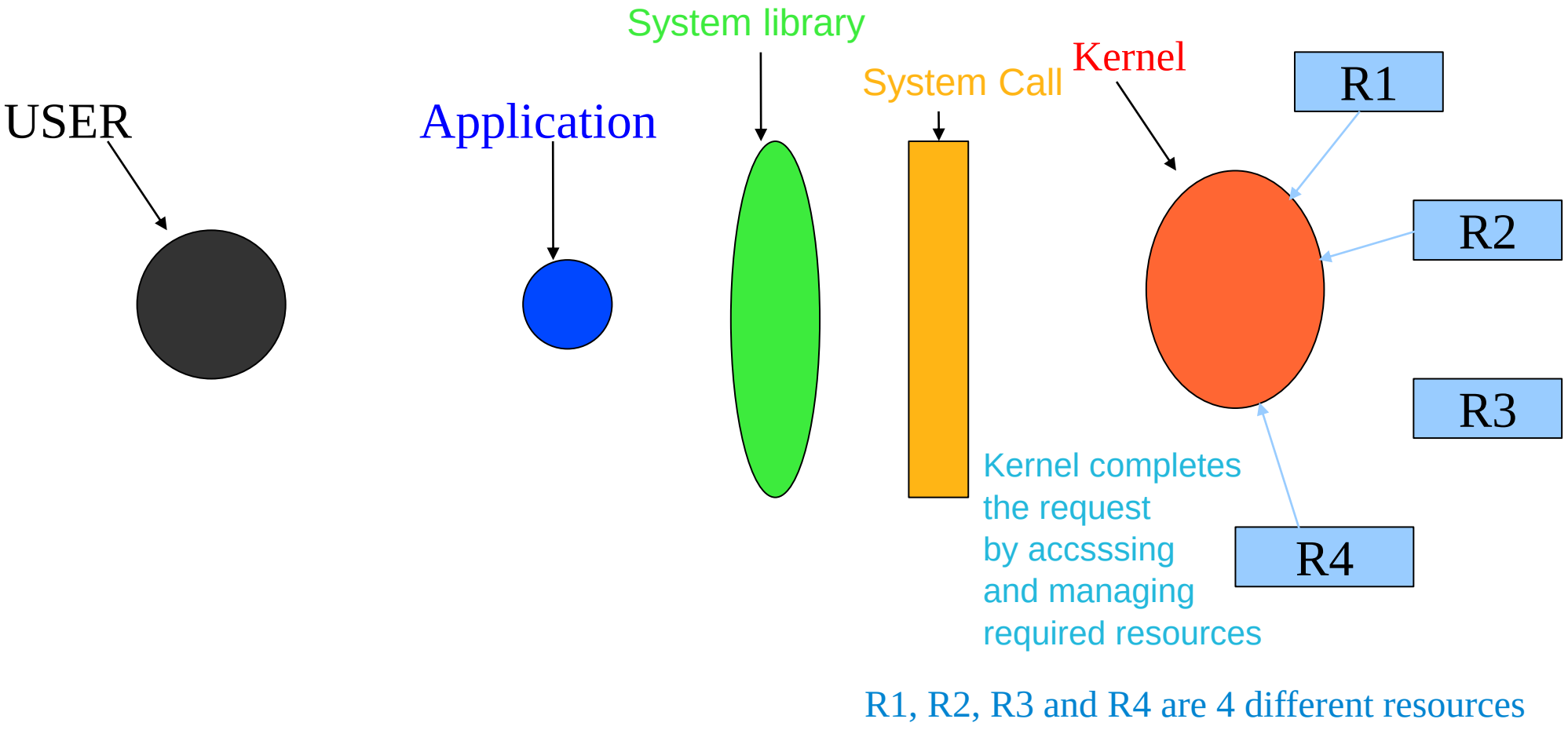
R1, R2, R3 and R4 are 4 different resources

Kernel Explanation contd..

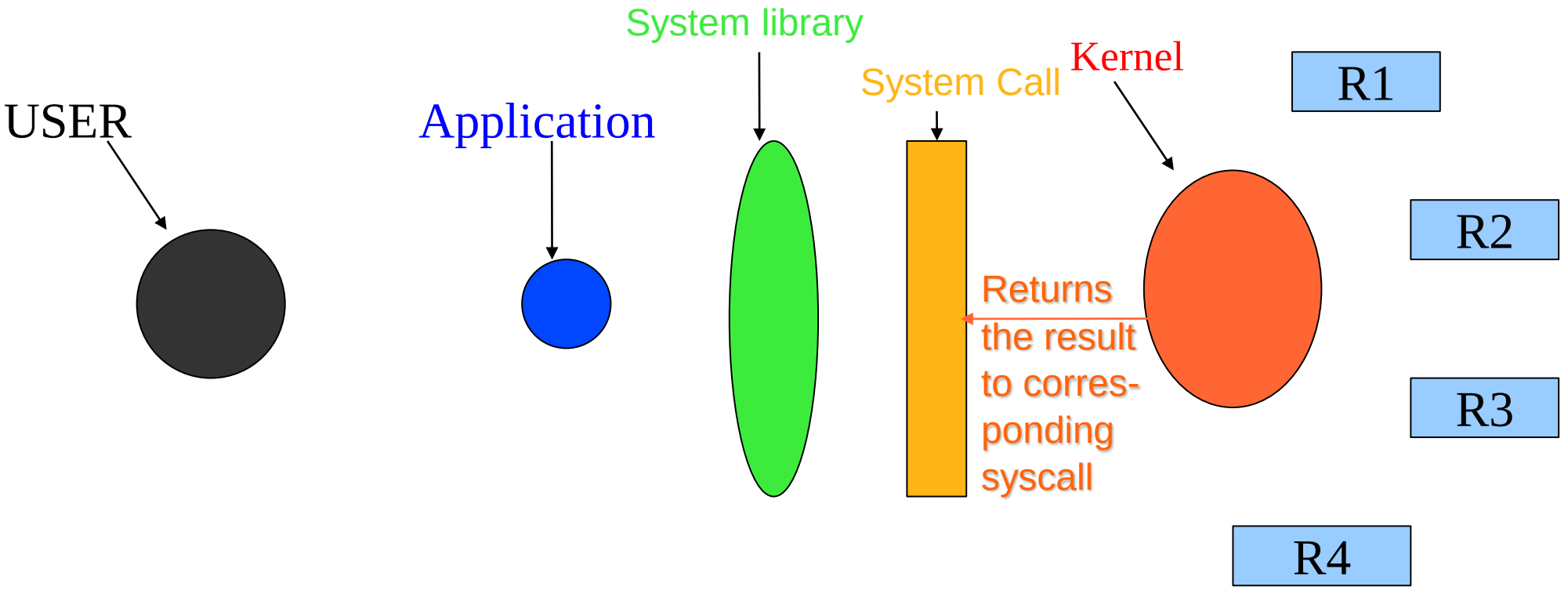


R1, R2, R3 and R4 are 4 different resources

Kernel Explanation contd..

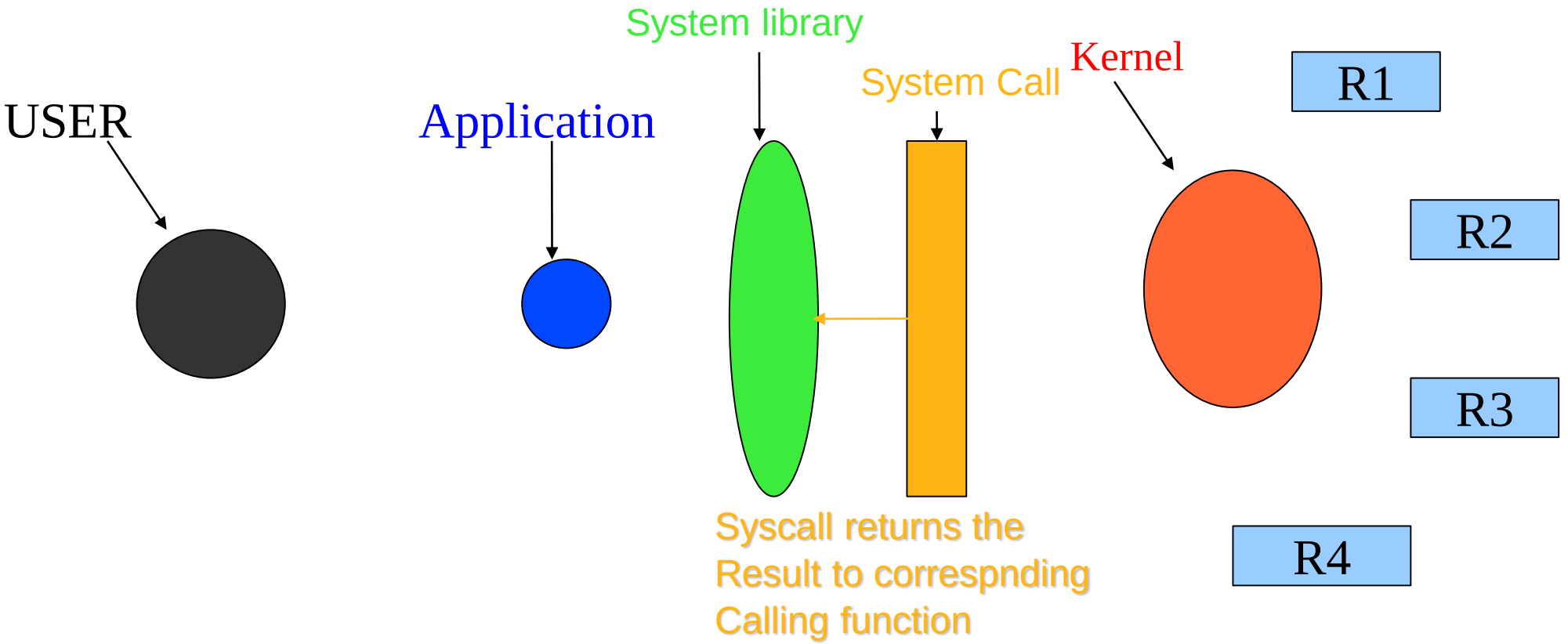


Kernel Explanation contd..



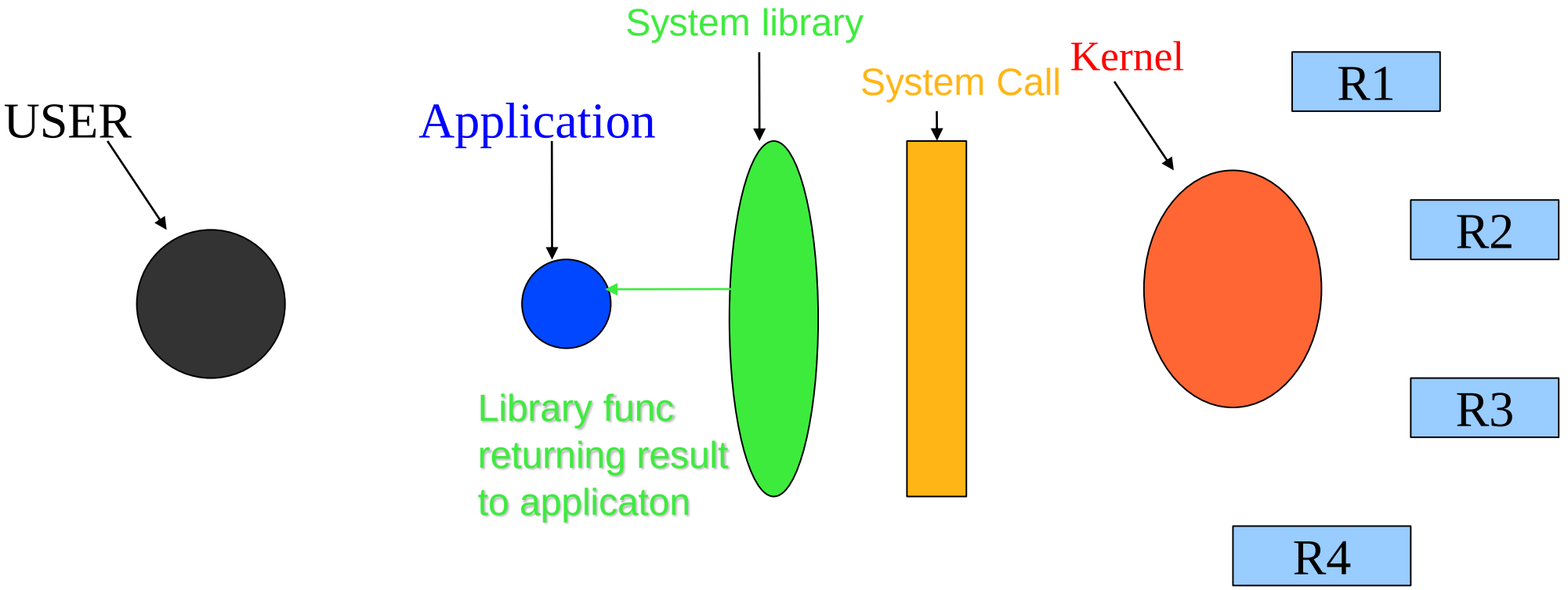
R1, R2, R3 and R4 are 4 different resources

Kernel Explanation contd..



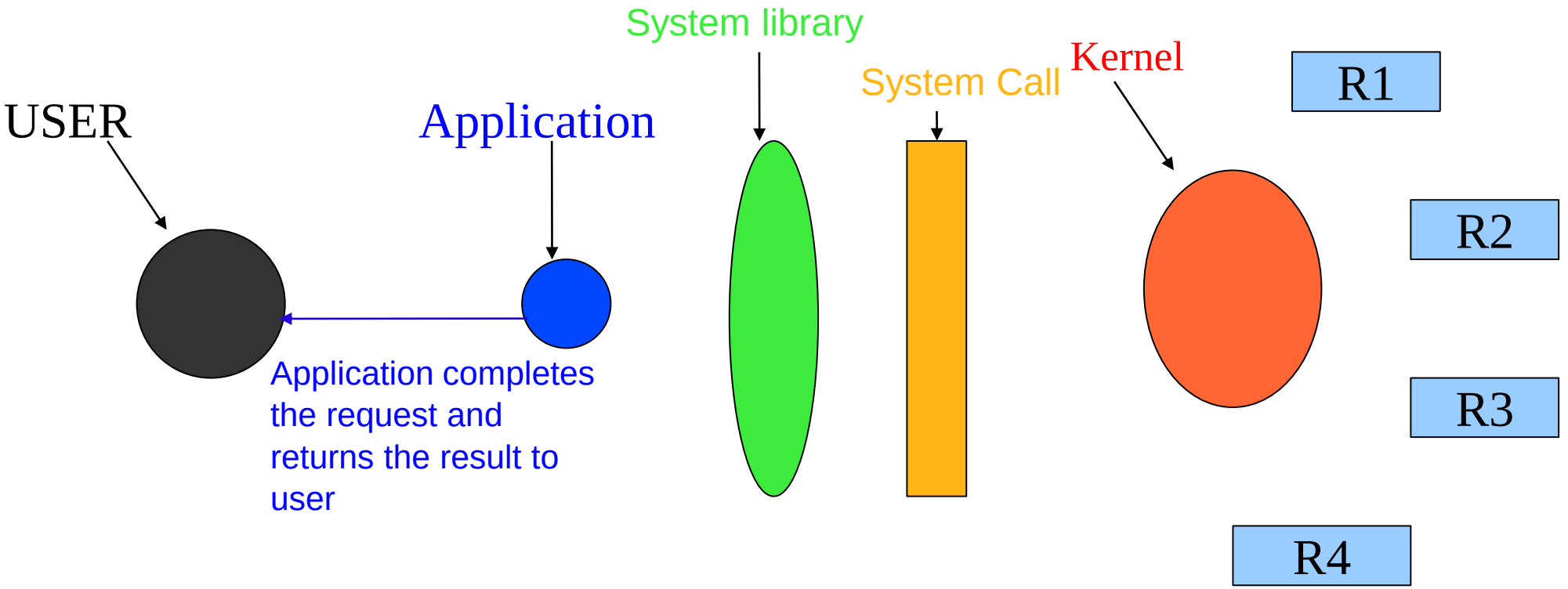
R1, R2, R3 and R4 are 4 different resources

Kernel Explanation contd..



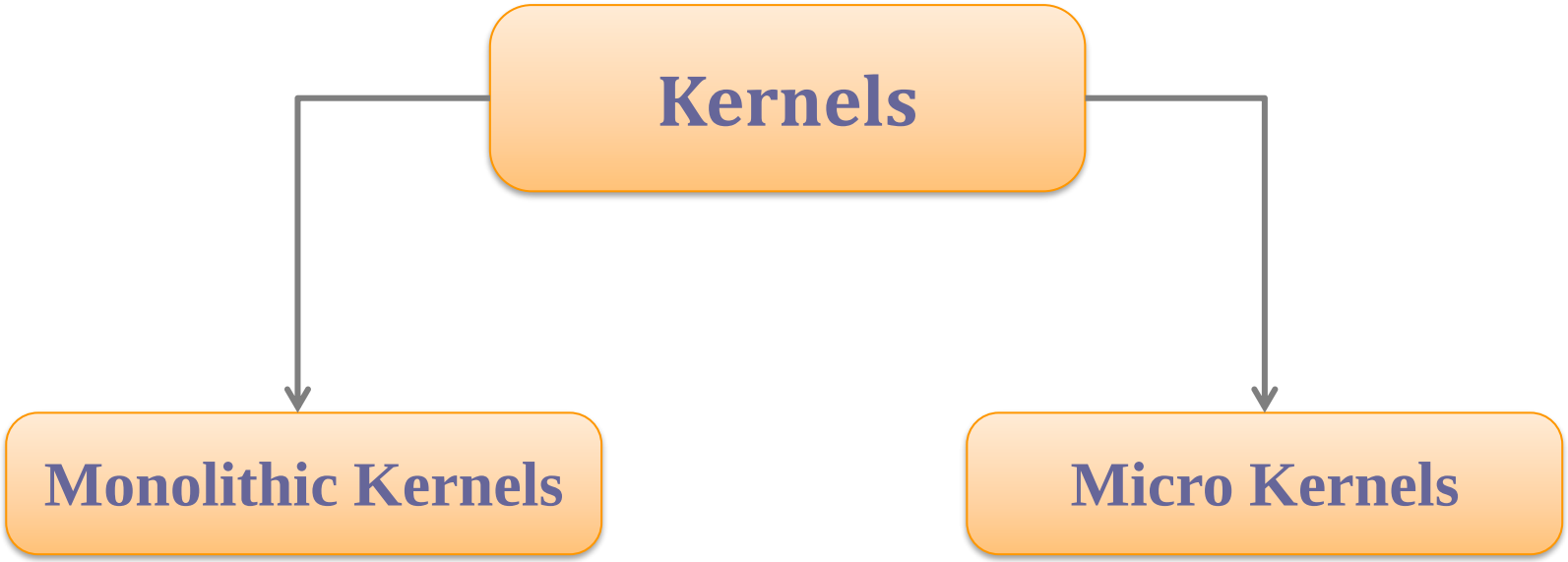
R1, R2, R3 and R4 are 4 different resources

Kernel Explanation contd..

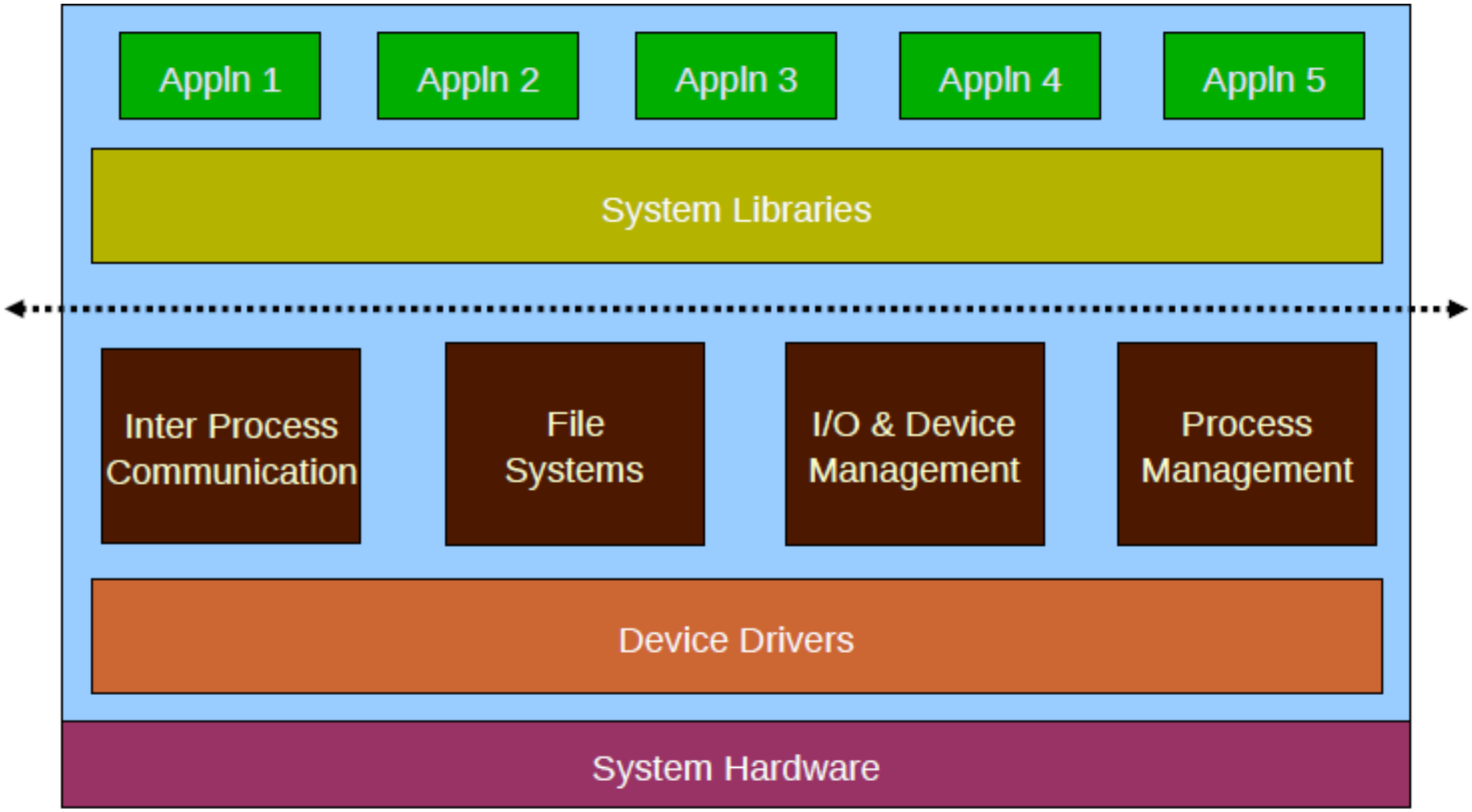


R1, R2, R3 and R4 are 4 different resources

Kernel Classifications



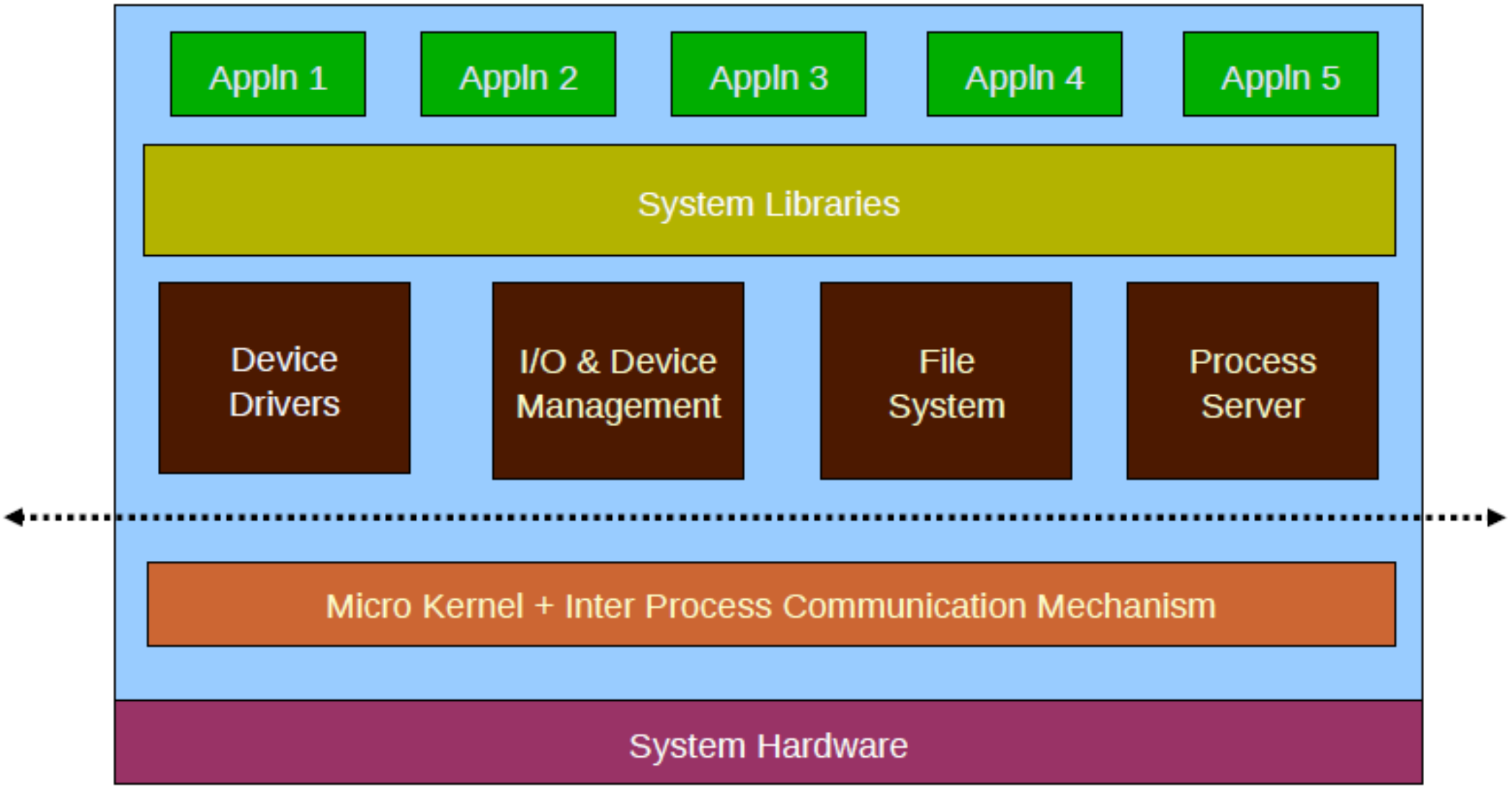
Monolithic Kernels



Monolithic Kernels

- All the parts of a kernel like the Scheduler, File System, Memory Management, Networking Stacks, Device Drivers, etc., are maintained in one unit within the kernel
  - Advantages
    - *Faster processing*
  - Disadvantages
    - *Prone to crash*
    - *Porting Inflexibility*
    - *Kernel Size explosion*
  - Examples
    - *MS-DOS, Unix, Linux, etc...*

µ-Kernels

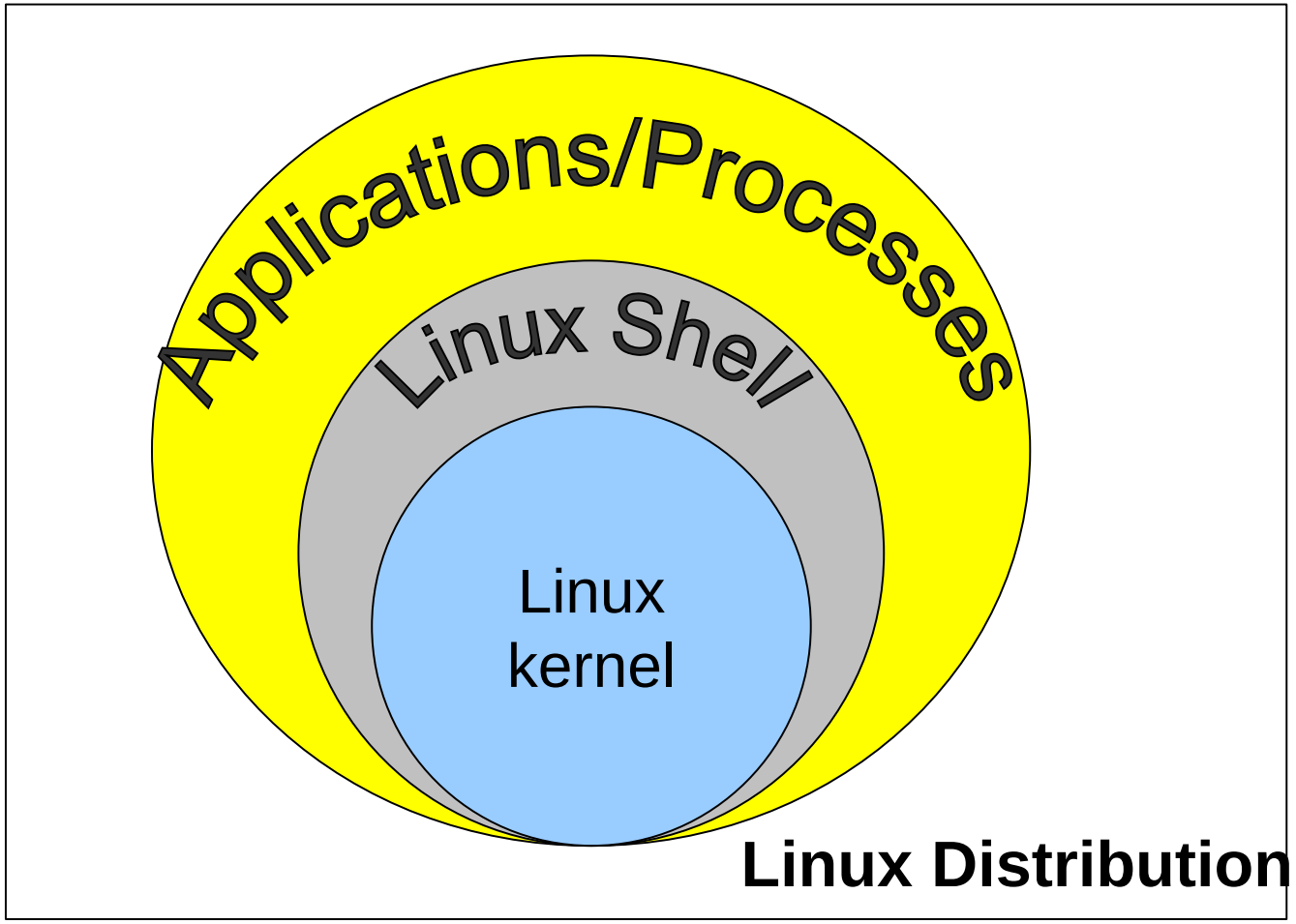

Appln 1
Appln 2
Appln 3
Appln 4
Appln 5
System Libraries
Device Drivers
I/O & Device Management
File System
Process Server
Micro Kernel + Inter Process Communication Mechanism
System Hardware

µ-Kernels

- Only the very important parts like IPC, basic scheduler, basic memory handling, basic I/O primitives etc., are put into the kernel. Others are maintained as server processes in User Space
  - Advantages
    - *Crash Resistant, Portable, Smaller Size*
  - Disadvantages
    - *Slower Processing due to additional Message Passing*
  - Examples
    - *QNX, Minix 3, EROS, KeyKOS, Windows NT*

The Linux Operating System

Linux Overview

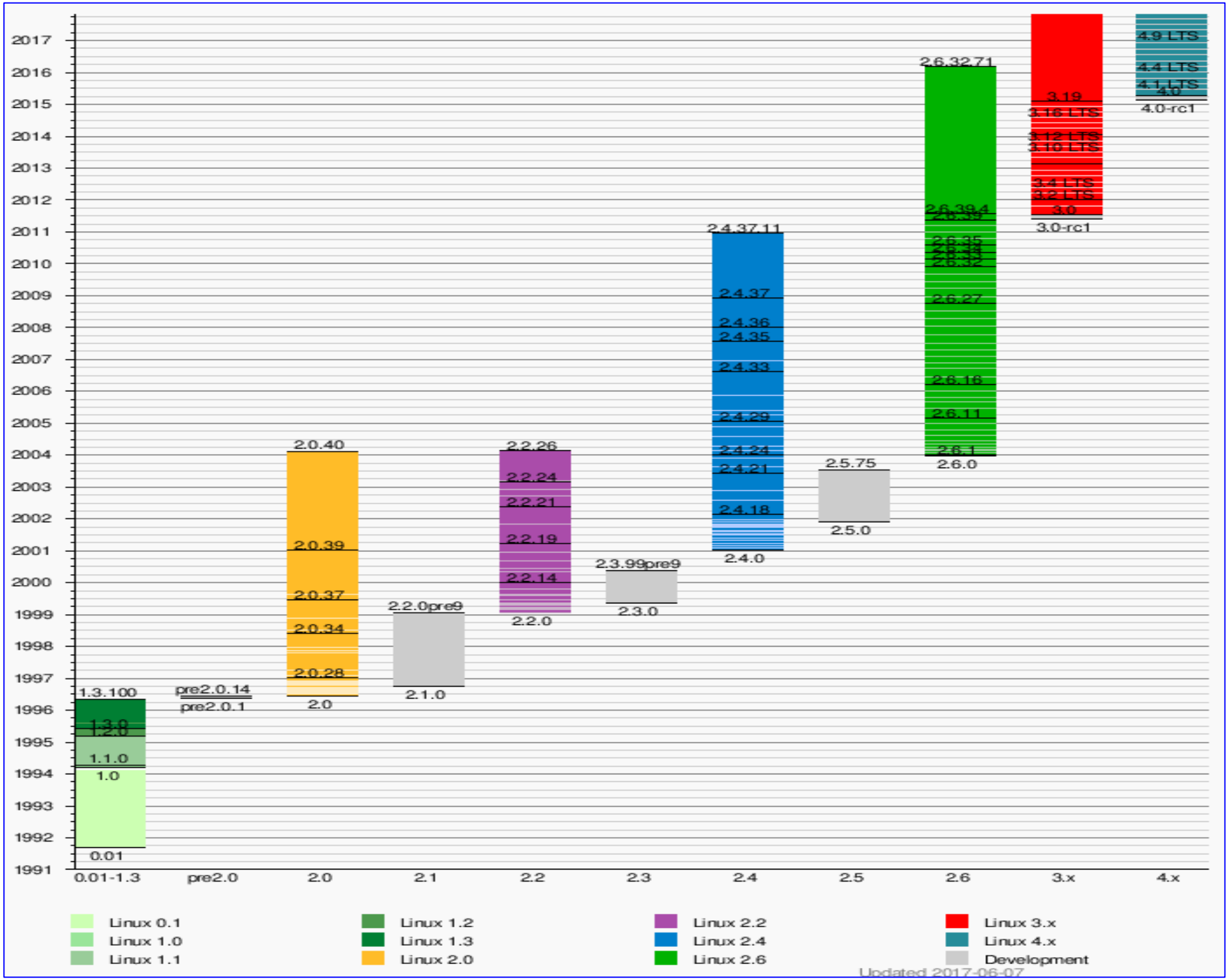


Linux Distributions

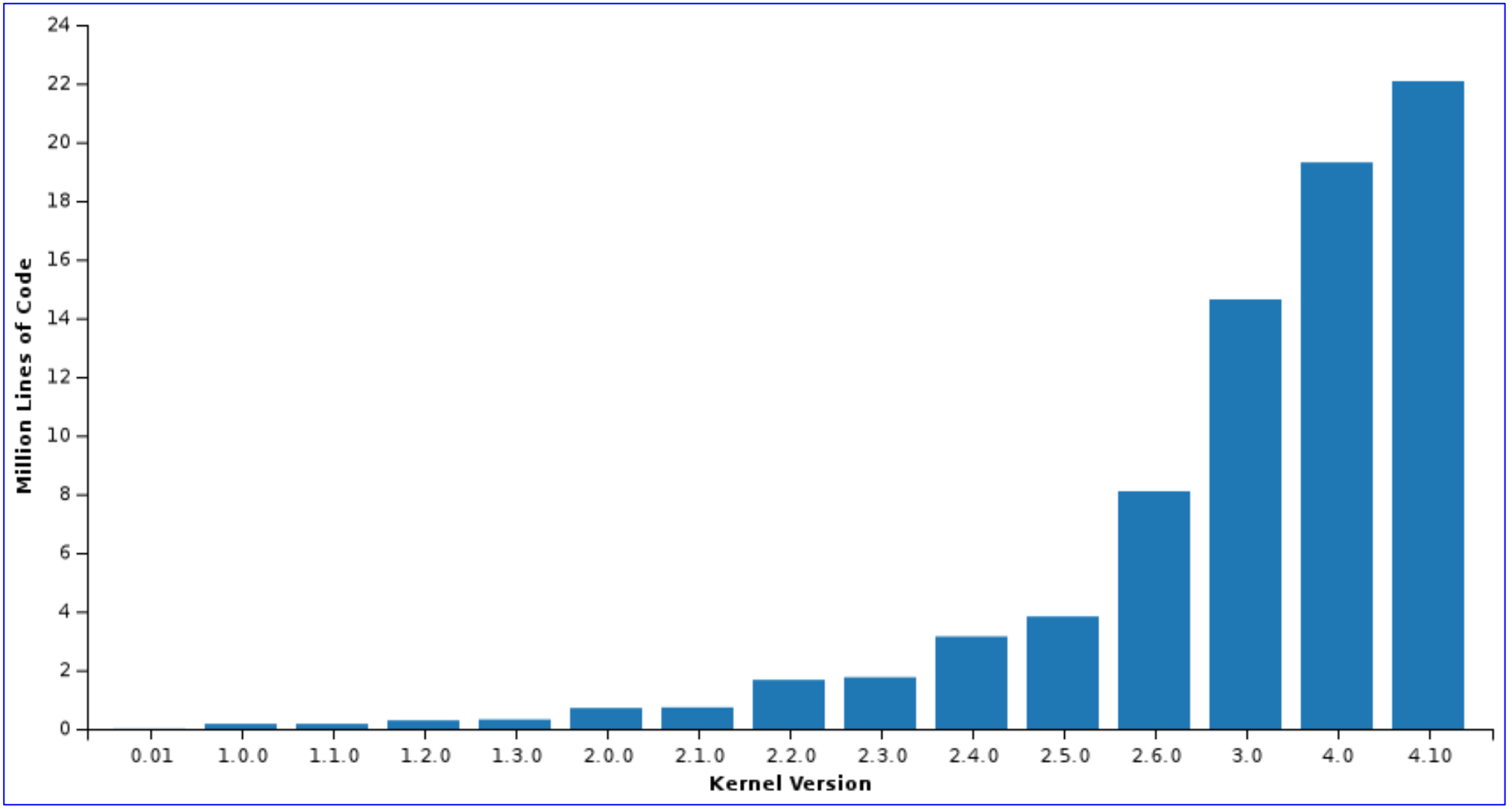
- In addition to the Linux kernel, it includes:
  - Extra system-installation utilities
  - Pre-compiled and ready to use packages of many tools
  - Like Web browsers, games, editors, music players etc.

The Linux Kernel

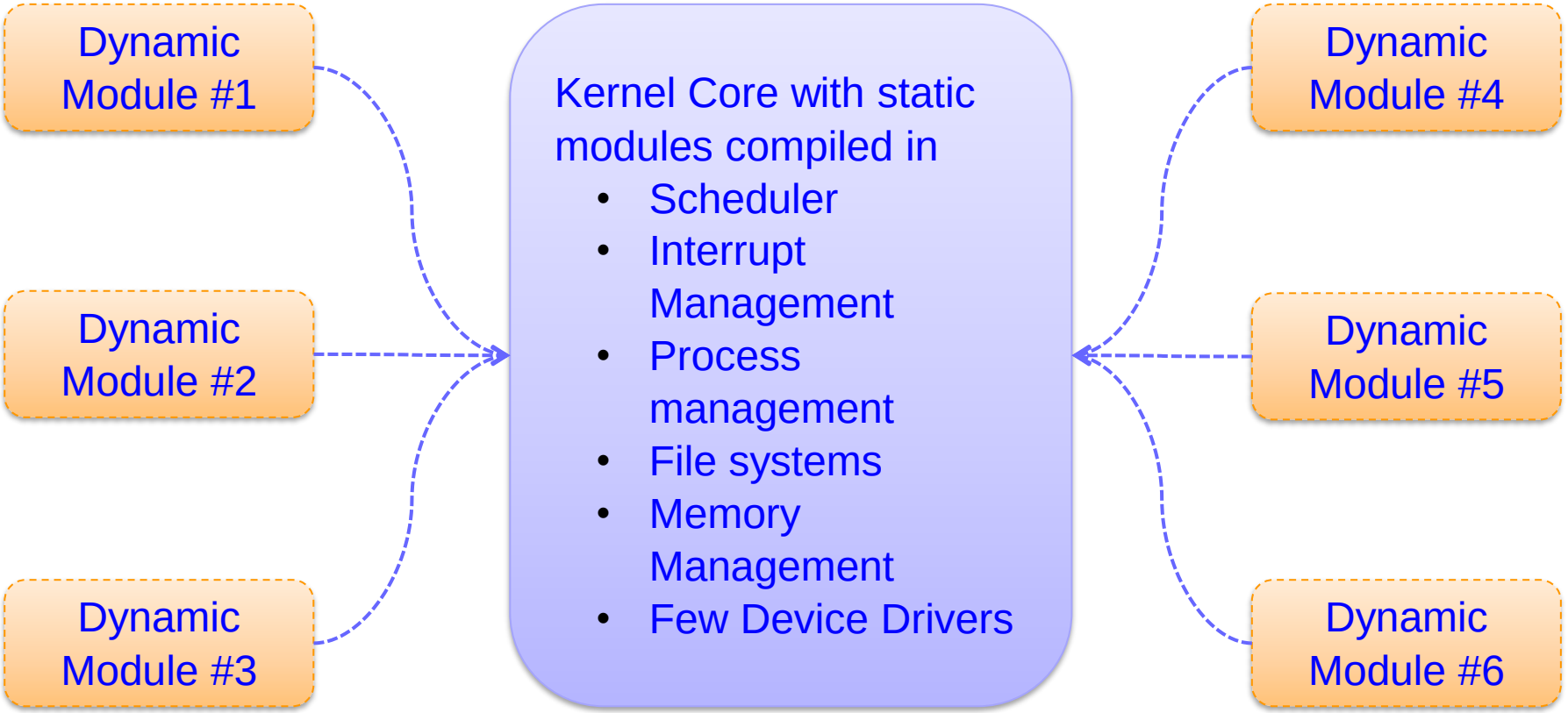
- Started off in 1991 by Linus Torvalds as a simple terminal driver
- Distributed under the GNU General Public License as a Free and Open Source Kernel
- The latest stable version of the kernel is 5.9+
- The kernel can be freely downloaded in source code from www.kernel.org


2017
2016
2015
2014
2013
2012
2011
2010
2009
2008
2007
2006
2005
2004
2003
2002
2001
2000
1999
1998
1997
1996
1995
1994
1993
1992
1991
0.01-1.3
pre2.0
2.0
2.1
2.2
2.3
2.4
2.5
2.6
3.x
4.x
0.01
1.0
1.1.0
1.2.0
1.3.0
1.3.100
pre2.0.1
pre2.0.14
2.0
2.0.28
2.0.34
2.0.37
2.0.39
2.0.40
2.1.0
2.2.0pre9
2.2.0
2.2.14
2.2.19
2.2.21
2.2.24
2.2.26
2.3.0
2.3.99pre9
2.4.0
2.4.18
2.4.21
2.4.24
2.4.29
2.4.33
2.4.35
2.4.36
2.4.37
2.4.37.11
2.5.0
2.5.75
2.6.0
2.6.1
2.6.11
2.6.16
2.6.27
2.6.32
2.6.33
2.6.34
2.6.35
2.6.39.4
2.6.32.71
3.0-rc1
3.0
3.2 LTS
3.4 LTS
3.10 LTS
3.12 LTS
3.16 LTS
3.19
4.0-rc1
4.0
4.1 LTS
4.4 LTS
4.9 LTS
Linux 0.1
Linux 1.0
Linux 1.1
Linux 1.2
Linux 1.3
Linux 2.0
Linux 2.2
Linux 2.4
Linux 2.6
Linux 3.x
Linux 4.x
Development
Updated 2017-06-07

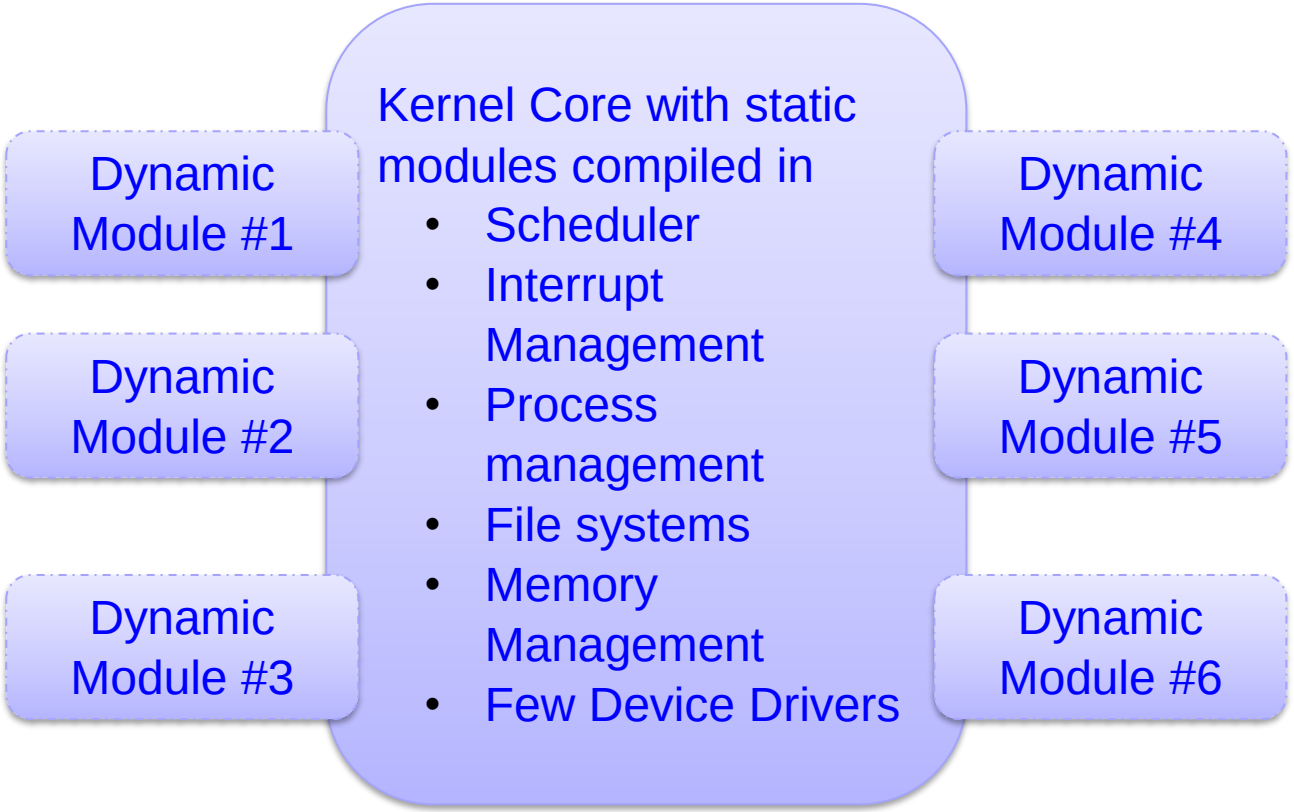
Kernel Development



Supports Dynamic Module Loading



Supports Dynamic Module Loading



Module Programming

What are Modules???

- Simple pieces of C Code.....
- Loaded at Runtime within the kernel ... No Reboot required...
- No main( ) function like normal C programs
- Provides Initialization and Cleanup Functions as entry & exit points to the Module

Applications vs Modules

- Event Driven Execution
- Libraries
- Floating Point Operations
- Segmentation Faults
- Debugging
- Stack Limitations
- Infinite Looping

What composes a module program?

- Module Initialization
- Module De-Initialization
- Compilation & Loading a Module
- Utilities to view/manage/remove a module
  - Insmod, lsmod, rmmod, dmesg
- The Makefile to compile a kernel module
- Module Parameters – Passing command line arguments
- Sharing data between modules – The Kernel Symbol Table

Programming Template of Modules

- Module Initialization
  - Mechanism of registering with the kernel
  - A module always begin the function you define with module_init()
    - This the entry point for the module
    - It tells the kernel what functionality module provides
    - Once it's done, init function returns and module remain in kernel waiting to be called
      - Calls are executed by user space application or system hardware

Module Initialization

```
#include <linux/init.h>
#include <linux/module.h>

static int hello_init(void)
{
  printk(“\nHello, World\n”);
  return 0;
}

module_init (hello_init);
```

Programming Template of Modules

- Module Cleanup
  - A module ends by calling module_exit() function.
    - Exit point for the module
    - It undoes whatever initialization function performed
    - It unregisters the module from the kernel

Module Cleanup

```
#include <linux/init.h>
#include <linux/module.h>

static void hello_exit(void)
{
   printk(“\nGoodye, World\n”);
}

module_exit (hello_exit);
```

Important MACROs

- **module_init (***Any Name for your init function***)**
- **module_exit** (*A suitable name for your exit function)*
  - *What ever name is passed to the above macros should be defined as the name of the init and exit functions*
- Additional Macros - Declared anywhere within the module
  - **MODULE_LICENSE("GPL"):** It's not necessary
    - Without this kernel gives warning message of kernel tainted
    - Few licenses that it understands is GPL, GPL v2 etc
  - **MODULE_AUTHOR("AUTHOR_NAME"):** author name
  - **MODULE_DESCRIPTION("DESCRIPTION ABOUT MODULE"):**
  - **MODULE_ALIAS("ALIAS NAME FOR MODULE")**

So, How does the program look??

Your first C program without main Hello.c

To make it to local to this module only

It says that the given function is used only at initialisation time

It's like printf. Prints value into kernel log message

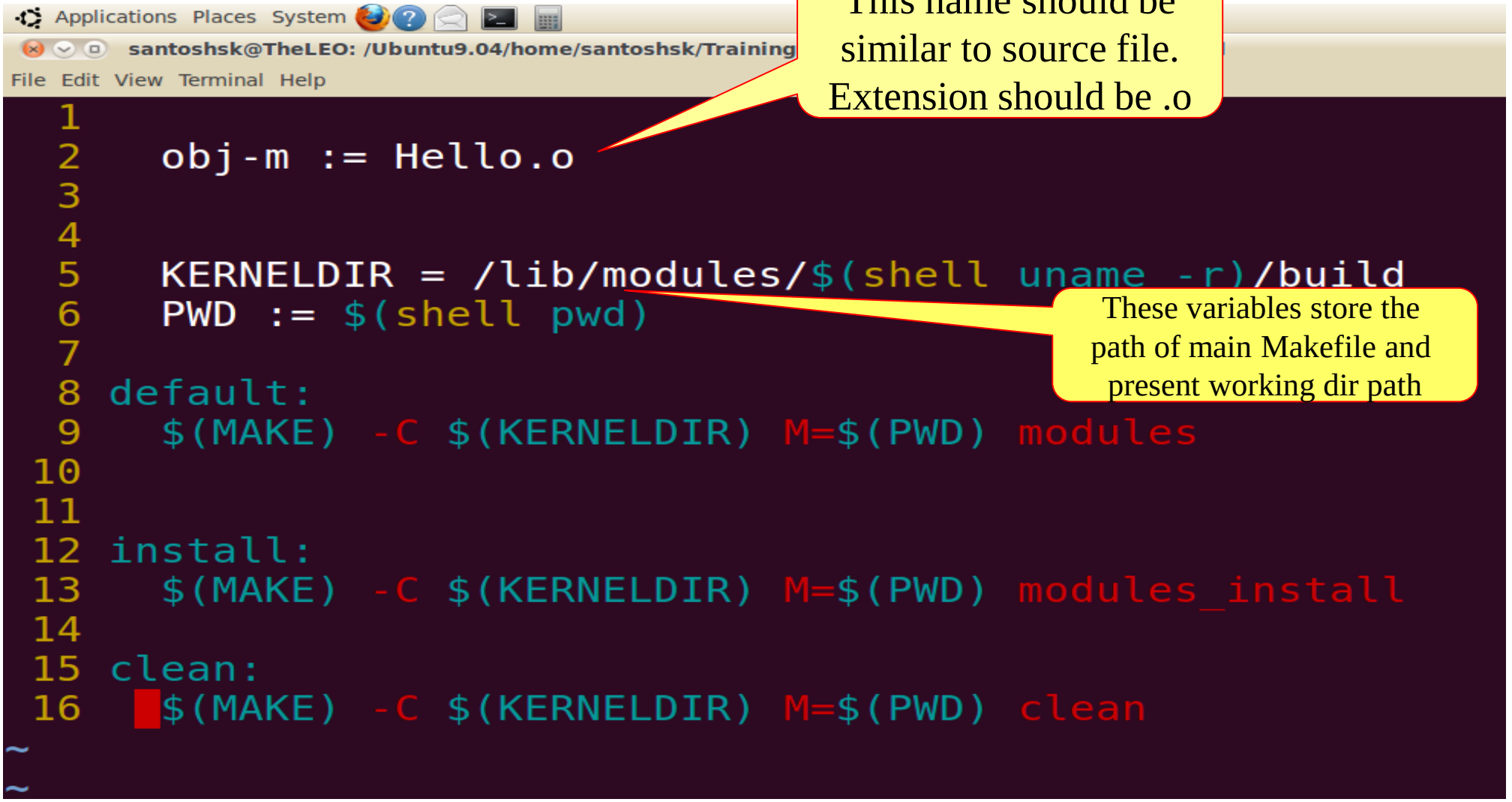
This unloads all the resources loaded by __init function

```
/**** HELLO TO ALL ****/
#include<linux/init.h>
#include<linux/module.h>
#include<linux/kernel.h>

MODULE_LICENSE("GPL");  /* <<-- tells that module bears free license    */
MODULE_AUTHOR("i am");  /* <<-- name of the author                      */

/* To initialise this module and load it into kernel                    */

static int __init hello_init(void)
{
        printk behaves similar to printf but it works without use of C library */
/* KERN_ALERT is the priority message; decides the seriousness of message */
        printk(KERN_ALERT "HELLO TO ALL\n");
        return 0;
}

/* This removes module from kernel                                      */

static void __exit hello_exit(void)
{
        printk(KERN_ALERT "BYE TO ALL\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

The Kernel Makefile

```
obj-m := hello.o

//For Native Compilation Platform
KERNELDIR = /lib/modules/$(shell uname -r)/build
//For Cross Compilation Platform
KERNELDIR = /lib/modules/<YourKernelModulesInstallPath>/build
PWD :=$(shell pwd)

default:
$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
```

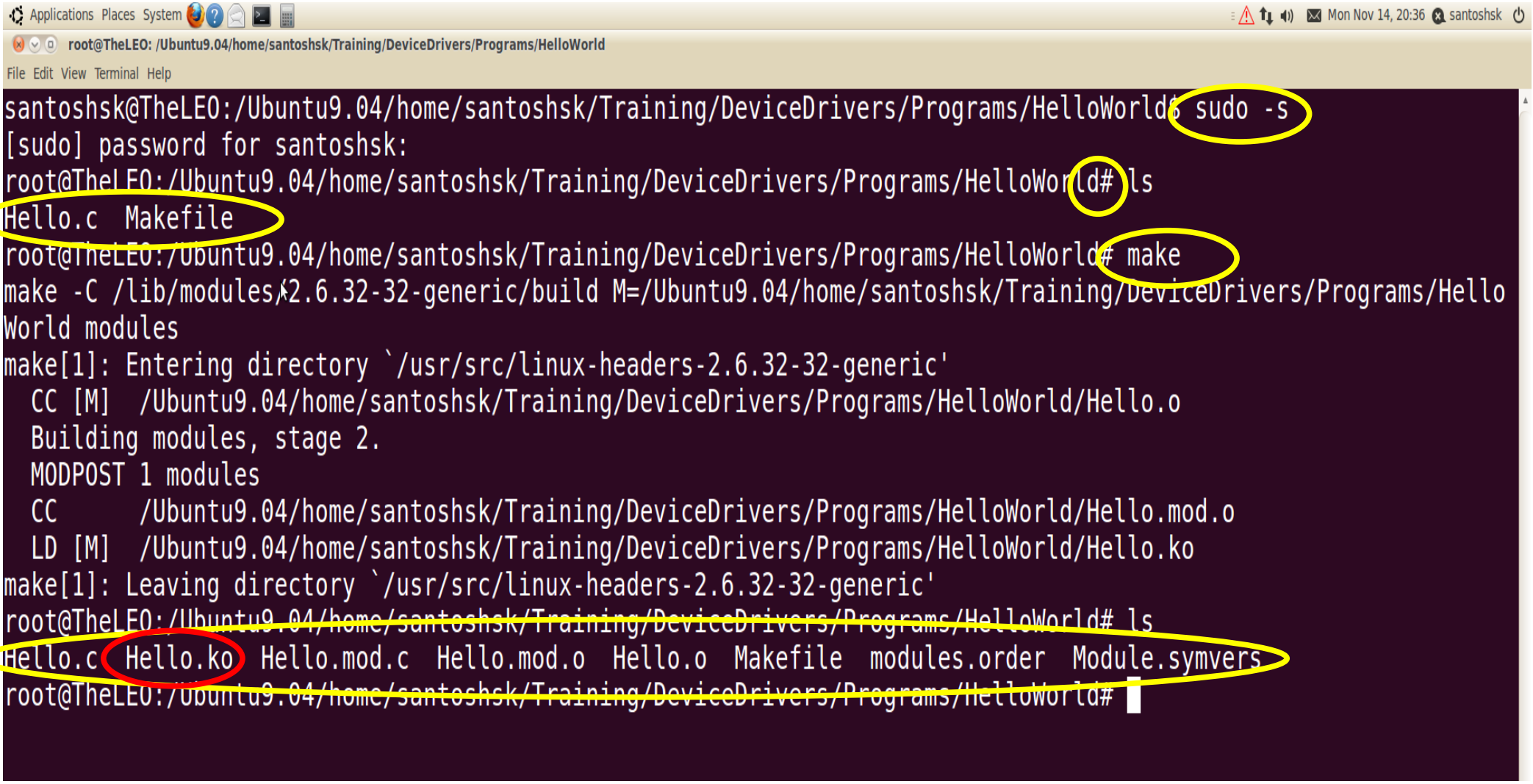
Makefile


Applications Places System
santoshsk@TheLEO: /Ubuntu9.04/home/santoshsk/Training
File Edit View Terminal Help
1
2 obj-m := Hello.o
3
4
5 KERNELDIR = /lib/modules/$(shell uname -r)/build
6 PWD := $(shell pwd)
7
8 default:
9 $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
10
11
12 install:
13 $(MAKE) -C $(KERNELDIR) M=$(PWD) modules_install
14
15 clean:
16 $(MAKE) -C $(KERNELDIR) M=$(PWD) clean
This name should be similar to source file. Extension should be .o
These variables store the path of main Makefile and present working dir path

Compiling & Loading the Module

```
linux:~ $ make

linux:~ $ su

linux:~ # insmod ./hello.ko

linux:~ # dmesg

linux:~ # lsmod

linux:~ #  rmmod hello
```

Compilation



Inserting our “Hello To All” module

- To insert modules commands are : insmod and modprobe

dmesg -c is for clearing the kernel log messages

insmod is used to insert our kernel module (.ko) into kernel space

Output can be seen from kernel log messages which can be seen using dmesg command

lsmod: To see our module is inserted in kernel

```
root@KERNEL:~/kern_mod# ls
kern_mod.c   kern_mod.mod.c  kern_mod.o  Module.markers  Module.symvers
kern_mod.ko  kern_mod.mod.o  Makefile    modules.order
root@KERNEL:~/kern_mod# insmod kern_mod.ko
root@KERNEL:~/kern_mod# dmesg
[  644.857245]
[  644.857246] HELLO TO ALL
[  644.857247]
root@KERNEL:~/kern_mod# lsmod | grep kern_mod
kern_mod                1084  0
root@KERNEL:~/kern_mod#
```

Output message printed by our module

Module unloading

- To remove kernel module : rmmod

```
root@KERNEL:~/kern_mod# dmesg
[  644.857245]
[  644.857246] HELLO TO ALL
[  644.857247]
root@KERNEL:~/kern_mod# rmmod kern_mod
root@KERNEL:~/kern_mod# dmesg
[  644.857245]
[  644.857246] HELLO TO ALL
[  644.857247]
[  772.813630]
[  772.813631] BYE TO ALL
[  772.813632]
root@KERNEL:~/kern_mod#
```

Root permission is required

rmmod and the name of your module, removes the module

In dmesg, we can see the output printed by our exit function

Now when you will do lsmod, our module will be missing from there.

Module Parameters

- Parameters/Command Line Arguments may be passed to the modules during module insertion
- The macro ***"module_param (name, type, perm)"*** is used to initialize the parameter at runtime.
- It takes 3 arguments which are the parameter name, parameter type and the permissions associated with the parameter
- This macro may be defined anywhere in the module

Parameter Passing

This Macro is used to mention that these variables can be initialised from command line

charp = character pointer
bool = boolean
invbool = inverted Boolean
Rest are same data types

```
#include<linux/init.h>
#include<linux/module.h>
#include<linux/kernel.h>

//MODULE_LICENSE("GPL");  /* <<-- tells that module bears free license          */
MODULE_AUTHOR("i am");  /* <<-- name of the author                              */

/* Variables are declared as static to keep their scope local to this module..  *
 * and avoid namespace poolution                                                */

static char* charvar = "module";
static int intvar   = 10;

/* using the following macro, variables are enabled to be modified from command-*
 * line                                                                         */
/* module_param takes three arguments: var name, type of variable, permission   */

module_param(charvar, charp, S_IRUGO);
module_param(intvar, int, S_IRUGO);


static int __init param_init(void)
{
        printk(KERN_ALERT "\n We are in init function\n");
        printk(KERN_ALERT "\n The value of charvar is %s\n", charvar);
        return 0;
}

static void __exit param_exit(void)
{
        printk(KERN_ALERT "\n GoodBye\n");
}

module_init(param_init);
module_exit(param_exit);
~
```

Parameter Passing ...

```
root@KERNEL:~/kern_mod_parm# insmod kern_parm.ko
root@KERNEL:~/kern_mod_parm# dmesg
[  998.301970]
[  998.301972]  We are in init function
[  998.301974]
[  998.301975]  The value of charvar is module
root@KERNEL:~/kern_mod_parm# lsmod | grep kern_parm
kern_parm              1596  0
root@KERNEL:~/kern_mod_parm#
```

No parameter

Printed local value

Passing command line parameters

```
root@KERNEL:~/kern_mod_parm# insmod kern_parm.ko charvar="command"
root@KERNEL:~/kern_mod_parm# dmesg
[ 1131.653534]
[ 1131.653536]  We are in init function
[ 1131.653539]
[ 1131.653539]  The value of charvar is command
root@KERNEL:~/kern_mod_parm#
```

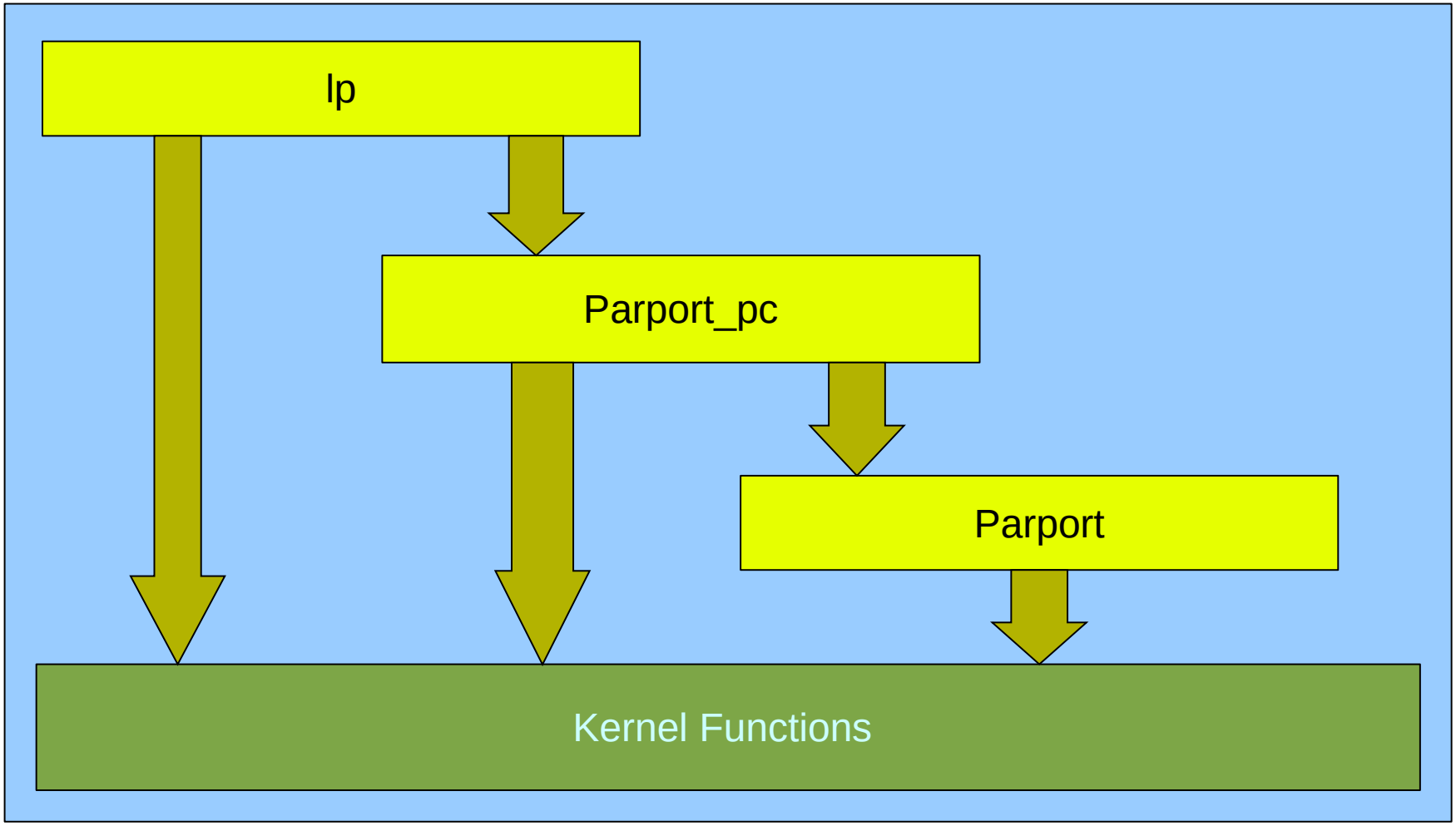
charvar is a charp variable we defined in module

Printed command line value

Library Support

- Standard C library functions cannot be used in module programming
- Functions/Variables shared between modules have to be exported as kernel symbols into a global space
  - To see the symbols exported into the kernel “cat /proc/kallsyms”
  - printk() is one such example
- External symbols used in a module (printk) should be resolved at runtime
- If symbol is not found, module will not be loaded to kernel and return back with an error

The Kernel Symbol Table



The Kernel Symbol Table

- Global Kernel Space
- Modules can export services to other modules through the use of kernel macros ***"EXPORT_SYMBOL(name)"***
  - It takes the name of the parameter or function as argument
- This exported function may be used by other modules that require similar functionality
- For example
  - Module 1 defines a function "**int Add(int a, int b)**" which takes two arguments and returns an integer output
  - This function is exported from Module1 to the Kernel Symbol Table using the macro **"EXPORT_SYMBOL(Add)"**
  - When this Module1 is inserted in the kernel, the exported symbol Add becomes available for other modules to use
  - Now, Module 2 can call the function Add() in its execution, provided Module1 is already inserted
- Please Note
  - The order of insertion and deletion of modules should always be maintained for success

Exporting symbol – Module1

Program name kern_sym.c

This is a simple add function declearation that we are going to export

Command to export symbol

Header file containing declaration of the add function

```
/***** header file*****/
int my_add(int, int);
```

```
#include<linux/init.h>
#include<linux/module.h>
#include<linux/kernel.h>

MODULE_LICENSE("GPL");  /* <<-- tells that module bears free license    */
MODULE_AUTHOR("i am");  /* <<-- name of the author                      */

/* This is addition function that we are going to export as symbol      */
static int my_add(int a, int b)
{
        return (a + b);
}

/* Command to export symbol into kernel symbol table                    */
EXPORT_SYMBOL(my_add);


/* To initialise this module and load it into kernel symbol table       */

static int __init hello_init(void)
{
/* printk behaves similar to printf but it works without use of C library */
/* KERN_ALERT is the priority message; decides the seriousness of message */
        printk(KERN_ALERT "\nHELLO TO ALL\n\n");
        return 0;
}

/* This removes module from kernel symbol table                         */

static void __exit hello_exit(void)
{
        printk(KERN_ALERT "\nBYE TO ALL\n\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Exporting Symbol: Module2

```c
#include<linux/init.h>
#include<linux/module.h>
#include<linux/kernel.h>
#include"kern_add.h"

MODULE_LICENSE("GPL");  /* <<-- tells that module bears free license    */
MODULE_AUTHOR("i am");  /* <<-- name of the author                      */

static int one = 1;
static int two = 2;


static int __init add_init(void)
{
        printk(KERN_ALERT "\nwe are going to add\n");
        printk(KERN_ALERT "\nadd result is: %d\n", my_add(one,two));
        return 0;
}

static void __exit add_exit(void)
{
        printk(KERN_ALERT "\n we are leaving \n");
}

module_init(add_init);
module_exit(add_exit);
```

Header containing the declaration of symbol

We are using the symbol in this module

Resolving Multiple Dependencies

- What if, you had a module, that is dependent on 10 other modules?
  - Using insmod, you have to manually load each module in specific order, before you can load your module
  - Unloading also requires you to follow the same process
- Is there no other mechanism to automate this??
  - Use modprobe
  - modprobe works in similar way as that of insmod, but it also loads any other modules that are required by the module you want to load.

Logically Thinking….

- Questions/Doubts
  - How does the kernel know where these modules are?
  - How does the kernel know what are the symbols that are exported by each module?
- Solutions
  - Provide a standard location from where the kernel will find the module
  - Provide a dependency file which resolves the modules dependencies on other modules that provide those functionalities

How modprobe works?

- The Linux Kernel maintains a module dependency file which stores the dependencies of one module on the other.
- Modprobe checks this file to resolve the dependency of modules and loads those corresponding modules
- The file is called modules.dep and is located in **/lib/modules/`uname -r`/**
- To know the dependency, modprobe looks into modules.dep file

How to update the modules.dep file?

- This file is updated by the command ***'depmod –a'***
- depmod command searches standard locations where the kernel stores its required modules.
  - It opens the modules present in the standard location in the filesystem,
  - It identifies the symbols that are not resolved,
  - It searches for modules that export these symbols
  - It updates the modules.dep file to convey this information.
- Alternately, to make depmod search into a user defined location on the filesystem, the absolute path of the module file should be provided to depmod
  - e.g depmod -a /home/user/test/kern_sym.ko /home/user/test/kern_add.ko

The mechanics of modprobe

- modprobe can now search for the modules that export the unresolved symbols by looking into the modules.dep file and loading the corresponding modules
- modprobe searches for modules in standard directories
  - /lib/modules/(kernel version)/
- To put your modules in standard directories, use the label to execute the modules_install statement in your Makefile.

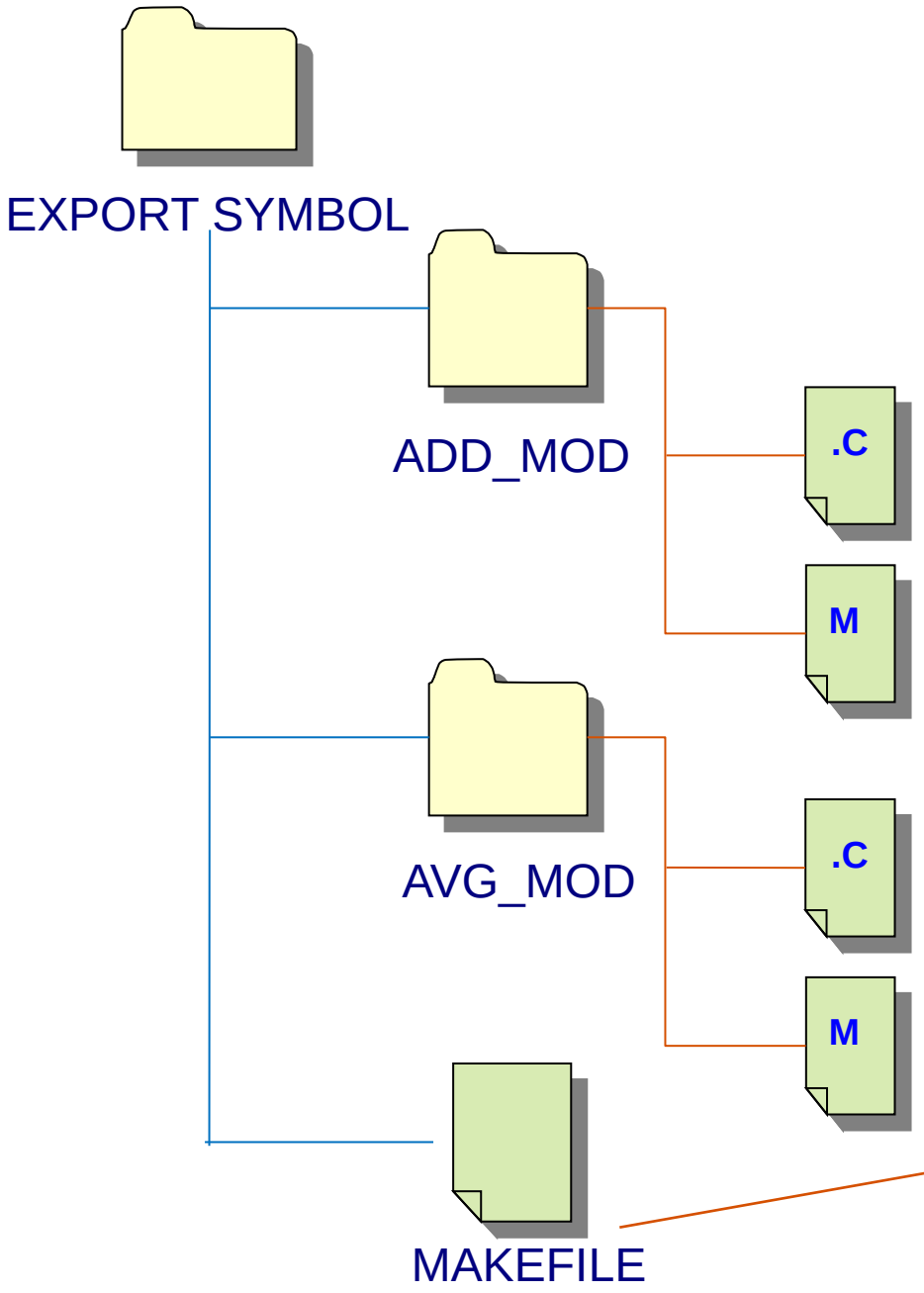
What should we do to use modprobe?

- Modify the Makefile
- Add these lines to Makefile:
  - install:

    <next line><tab>$(MAKE) -C $(KERNELDIR) M=$(PWD) modules_install

    depmod -a
- Doing “make install” will create a folder named “**extra**” in /lib/modules/`uname -r`/ folder and copies all .ko files in the current directory into that folder
- When we do modprobe, it takes .ko files from there and loads into the kernel

Exporting Symbol – A few additions...




```
obj-y := AVG_MOD/ ADD_MOD/

KERNELDIR = /lib/modules/$(shell uname -
                    r)/build
PWD := $(shell pwd)

default:
        $(MAKE) -C $(KERNELDIR) M=$(PWD)
        modules

clean:
        $(MAKE) -C $(KERNELDIR) M=$(PWD)
        clean

install:
        $(MAKE) -C $(KERNELDIR) M=$(PWD)
        modules_install
        depmod -a
```

References

- Jonathan Corbet, Alessandro Rubini and Greg Kroah-Hartman,"*Linux Device Drivers*",3rd Edition, O'Reilly Publications, March 2005
- http://www.senet.com.au/~cpeacock, "*Interfacing the Serial/ RS-232 port,V 5.0*"
- http://www.kernel.org
- http://en.wikipedia.org/wiki/Linux_kernel
- http://lists.ucc.gu.uwa.edu.au/pipermail/ucc/2003-June/009997.html


Thank You