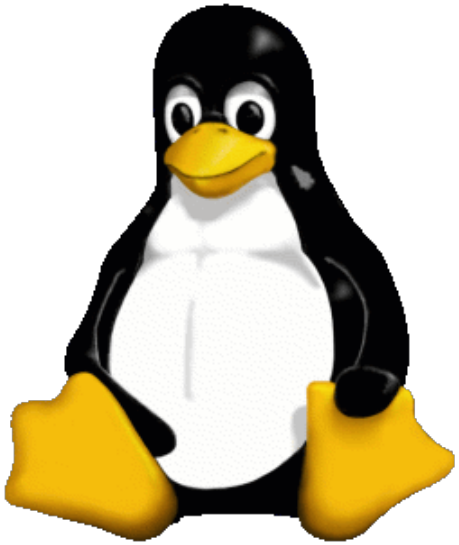
Assignment 1

Building and Installing the Linux Kernel and Developing a Loadable Kernel Module



Compiling and Installing the Linux Kernel

Why compile your own kernel?

- **Customization** - Tailor the kernel to include only the features and drivers you need.
- **Optimization** - Improve performance by enabling or disabling features based on your hardware.
- **Testing** - Experiment with new features or patches before they are officially released.

Boot Process of a Linux System

1. **Power-On Self-Test (POST)**
   - The system powers on and the BIOS (Basic Input/Output System) or UEFI (Unified Extensible Firmware Interface) firmware performs the POST.
   - Checks hardware components (CPU, RAM, disk drives).
   - Initializes system settings and performs basic hardware configuration.
2. **Bootloader Stage**
   - BIOS uses the Master Boot Record (MBR) whereas UEFI uses the EFI System Partition (ESP) to load the bootloader.
   - Displays boot menu if multiple kernels or OSes are available.
   - Loads the kernel image into memory.
   - Passes control to the kernel.

Boot Process of a Linux System

3. Kernel Initialization

- The bootloader loads the Linux kernel from disk into memory.
- The kernel initializes hardware and mounts the root filesystem.
- Initial RAM Filesystem (initramfs):
  - A temporary root filesystem loaded into RAM.
  - Contains necessary drivers and tools to prepare the actual root filesystem.
  - Used for tasks such as detecting hardware, decrypting disks, and setting up filesystems.
  - Mounted by the kernel as the initial root filesystem before switching to the real root filesystem.

4. Init System Initialization

- The kernel switches from initramfs to the real root filesystem (e.g., /).
- The kernel executes the init process, which is the first user-space process (PID 1).
- The init system starts system services and daemons as specified in configuration files.

Compiling and Installing the Kernel

1. **Get the kernel image and install dependencies**
   - Download the kernel image from https://www.kernel.org. Unzip the tarball.
   - Install the required dependencies for compilation

```
sudo apt-get update
sudo apt-get install build-essential libncurses-dev bison flex libssl-dev
```

1. **Configure the kernel**

```
make menuconfig
```

1. **Compile**

```
make -j <no. of threads>
```

1. **Install**

```
make modules_install -j <no. of threads>
make install -j <no. of threads>
```

To check the kernel version you are using you can use

uname -r

If on rebooting the system does not boot into the new kernel, load into grub and manually choose the kernel

Base Kernel Module vs Loadable Kernel Module

- You often have a choice between putting a module into the kernel **by loading it as a Loadable Kernel Module (LKM)** or **binding it into the base kernel**.
- You can add code to the Linux kernel while it is running
  - Called a loadable kernel module.
  - With LKM, you don't have to rebuild your kernel.
- Sometimes it is important to build modules into the **base kernel** instead of making it an LKM.
  - Anything that is necessary to get the system up must obviously be built into the base kernel.
  - For example, the driver for the disk drive that contains the root filesystem must be built into the base kernel.

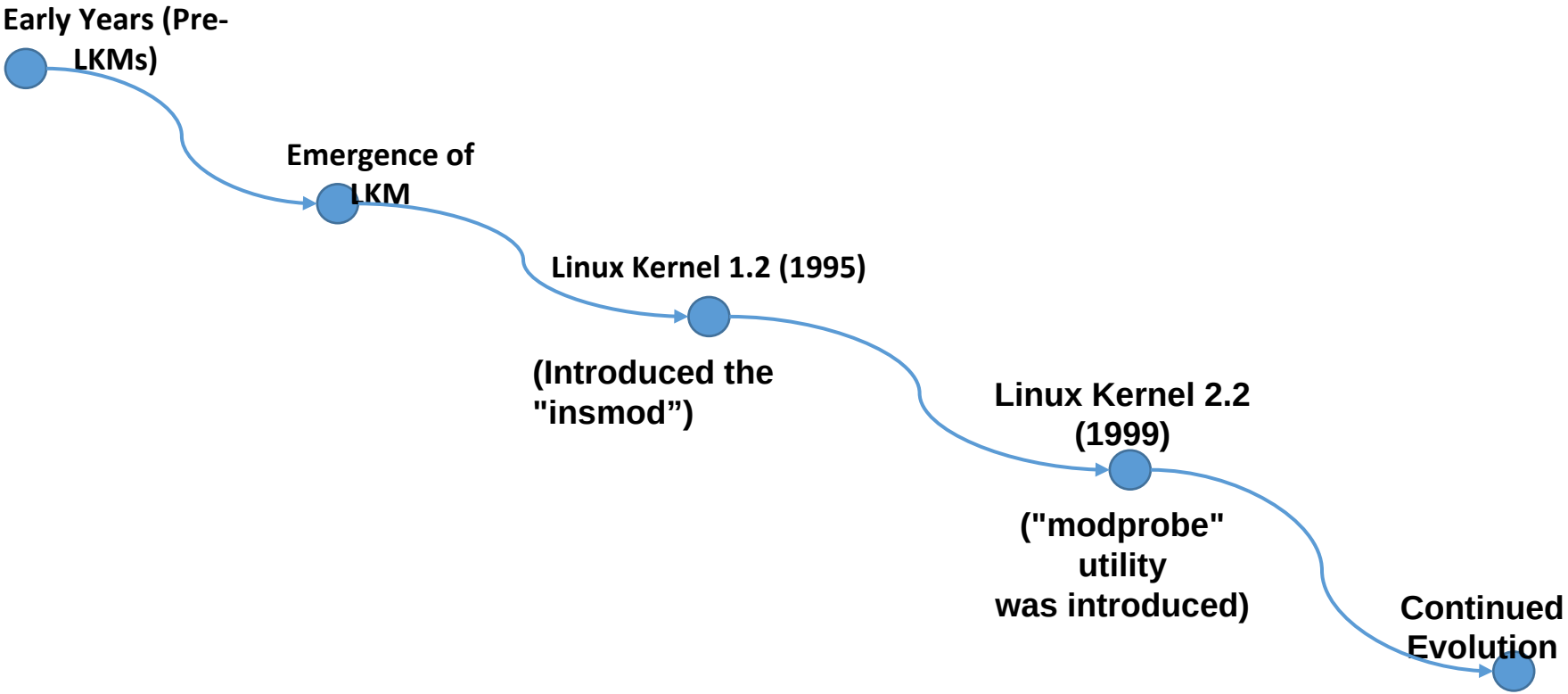
What is LKM?

- A loadable kernel module (LKM) is an object file that contains code to extend the running kernel, or so-called base kernel, of an operating system.
- LKM's are critical to the Linux administrator as they provide them the capability to add functionality to the kernel without having to recompile the kernel.
- Examples: Video and other device drivers can be added to the Linux kernel without shutting down the system, recompiling, and rebooting.

Why LKM?


Application Developer:
I want this new feature to observe my app
Hey kernel developer! Please add this new feature to the Linux kernel
OK! Just give me a year to convince the entire community that this is good for everyone.
1 year later...
I'm done. The upstream kernel now supports this.
But I need this in my Linux distro
5 year later...
Good news. Our Linux distribution now ships a kernel with your required feature
OK but my requirements have changed since...

Evolution of LKM Development?



Use Cases

- These modules can help in different ways --
- **Device drivers** - The kernel uses it to communicate with that piece of hardware without having to know any details of how the hardware works.
- **Filesystem drivers** - A filesystem driver interprets the contents of a filesystem as files and directories and such.
- **System calls** - Most system calls are built into the base kernel. But you can invent a system call of your own and install it as an LKM. Or you can override an existing system call with an LKM of your own.
- **Network drivers** - A network driver interprets a network protocol.

Linux Device Drivers – as a LKM

- Most of the Linux Device drivers are available as LKM. But Why as LKM?
- Device drivers are set of API sub-routines interface to hardware. It mainly abstracts the implementation of hardware-specific details from a user program.
- Another important aspect is that every Device is a special file in all Unix like systems i.e. Typically a user/user-program can access the device via file name in **`/dev`** , e.g. **`/dev/dv1`**.
- We have more than 70% of Linux kernel code specific to device drivers for thousands of devices.
  - Only very few of these devices are needed at any point of time in running instance.
  - As memory is costly and having all of them at once and having drivers of devices, which are useless is just a waste of memory.
- Hence, implementation of device drivers as LKM makes sense.

LKM Utilities

- The following is the list of basic LKM commands (utilities)

Utility	Description
insmod	Insert a LKM into the kernel
rmmod	Remove a LKM from the kernel
depmod	Lists dependencies between LKMs
ksyms	Lists symbols that are exported by the kernel for use by given LKMs
lsmod	Lists currently loaded LKMs
modinfo	Display contents of .modinfo section in an LKM object file
modprobe	Loads given module after loading/unloads modules required for the given module. For example, if you must load A before loading B, 'modprobe' will automatically load A when you tell it to load B

How to Write a LKM?

- Special programs and have structures entirely different from user program.
- They don’t have main functions but must have at-least 2 functions **`init_XXXXXX()`** and **`cleanup_XXXXXX()`** — here **`XXXXXX`** represent the module name.

Loadable Kernel Modules

These functions will match the following LKM Utilities, while they are plugged into kernel:

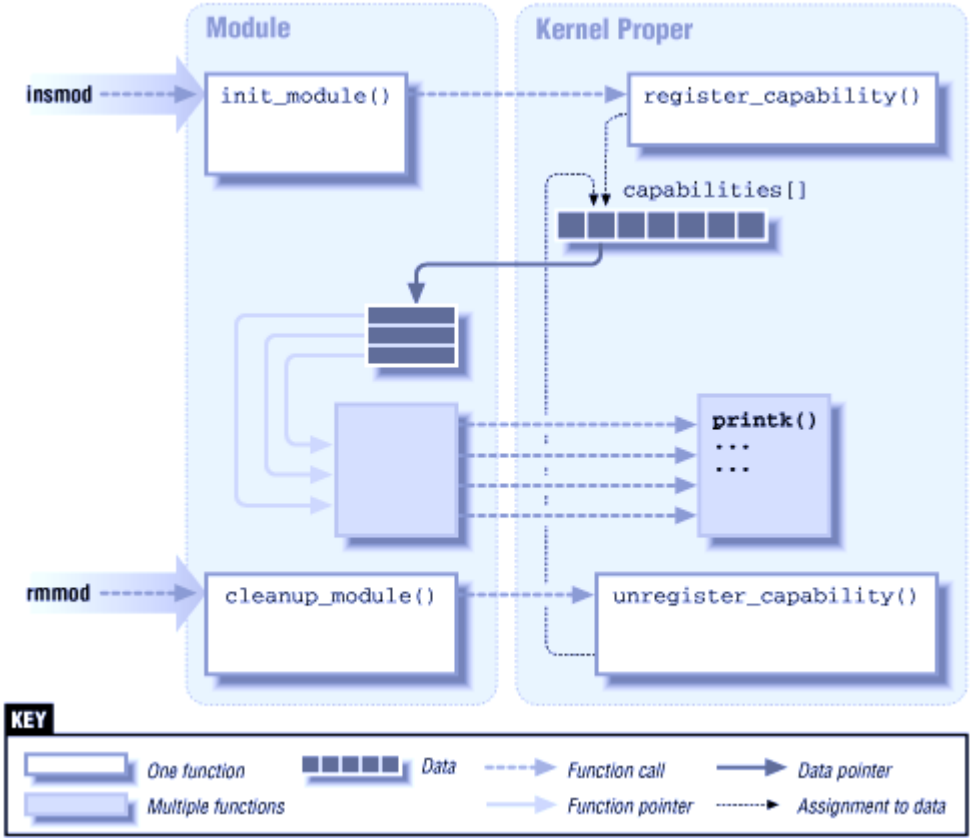
- **insmod**: Insert an LKM into the kernel.
- **rmmod**: Remove an LKM from the kernel.

```
int init_modulename (void)
{
/*initialition code*/
}

void cleanup_modulename (void)
{
/*cleanup code*/
}
```

LKM Architecture

- **init_module():**
  - Executed during loading.
  - It supposed to initialize the module and register capabilities.
  - It informs kernel that it is there and it can do this task.
- **cleanup_module():**
  - Need to free up allocated memory and resources.
  - It informs kernel that it is no longer there.
- **module_init() & module_exit():**
  - Allow any name for initialization and cleanup function.



Cautions

- **No libc** modules. Invoke a function only if it is available in the kernel.
- Use static function to avoid namespace pollution.
- Be careful about kernel space and user space.
- Concurrency in the kernel!!

Building and Running Modules

- **Note:** Vendor kernels can be heavily patched and divergent from the mainline; at times, vendor patches can change the kernel API as seen by device drivers.
  - If you are writing a driver that must work on a particular distribution, you will certainly want to build and test against the relevant kernels

- **Warning:** Faults in kernel code can bring about the demise of a user process or, occasionally, the entire system.
  - They do not normally create more serious problems, such as disk corruption.
  - Nonetheless, it is advisable to do your kernel experimentation on a system that does not contain data that you cannot afford to lose, and that does not perform essential services.

The Hello World Module – hello.c

A special macro (`MODULE_LICENSE`) is used to tell the kernel that this module bears a free license; without such a declaration, the kernel complains when the module is loaded.

Invoked when the module is loaded into the kernel **(`hello_init`)**

Invoked when the module is removed **(`hello_exit`)**

`module_init` & **`module_exit`** lines use special kernel macros to indicate the role of these two functions.

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

MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void) {
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}

static void hello_exit(void) {
    printk(KERN_ALERT "Goodbye, cruel
world\n");
}

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

The Hello World Module - hello.c

The **printk** function is defined in the Linux kernel and made available to modules; it behaves similarly to the standard C library function **printf**.

The kernel needs its own printing function because it runs by itself, without the help of the C library.
The module can call **printk** because, after **insmod** has loaded it, the module is linked to the kernel and can access the kernel's public symbols (functions and variables).

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

MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void) {
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}

static void hello_exit(void) {
    printk(KERN_ALERT "Goodbye, cruel
world\n");
}

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

The Hello World Module – hello.c

The string KERN_ALERT is the priority of the message

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

MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void) {
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}

static void hello_exit(void) {
    printk(KERN_ALERT "Goodbye, cruel
world\n");
}

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

Compiling Modules - Makefile

```
obj-m+=hello.o
all:
    make -C /lib/modules/$(shell uname -r)/build/ M=$(shell pwd)
modules
clean:
    make -C /lib/modules/$(shell uname -r)/build/ M=$(shell pwd) clean
```

The assignment above (which takes advantage of the extended syntax provided by GNU *make*) states that there is one module to be built from the object file *hello.o*.
The resulting module is named *hello.ko* after being built from the object file.

-C : Changing to the kernel source directory
M: Location of external module sources informs kernel an external module is being built
`uname –r`: version of currently running kernel

Testing the Module – Loading and Unloading

```
% make
make[1]: Entering directory `/usr/src/linux-x.y.z'
  CC [M] /home/ldd3/src/misc-modules/hello.o
  Building modules, stage 2.
  MODPOST
  CC /home/ldd3/src/misc-modules/hello.mod.o
  LD [M] /home/ldd3/src/misc-modules/hello.ko
make[1]: Leaving directory `/usr/src/linux-x.y.z'

% su

root# insmod ./hello.ko
root# dmesg
Hello, world

root# rmmod hello
root# dmesg
Goodbye cruel world
```

Note: Only the superuser can load and unload a module.

Configuration – for Debian UNIX

1. Update current packages of the system to the latest version.

```
$ sudo apt update && sudo apt upgrade –y
```

2. Download and install the essential packages to compile kernels

```
$ sudo apt install build-essential libncurses-dev libssl-dev libelf-dev bison flex -y

$ sudo apt install linux-headers-$(uname -r)
```

3. Compile LKM

```
$ make
```

Sending Data to LKM

```
// Include the header
    #include <linux/proc_fs.h>

// Declare global file operation variable
    static struct proc_ops file_ops;

// Create a file in /proc file during initialization (don't forget to check for error)
    struct proc_dir_entry *entry = proc_create("hello", 0, NULL, &file_ops);
    if(!entry)
          return -ENOENT;

// Set write function pointer
   file_ops.proc_write = write;

// Define the write function
    static ssize_t write(struct file *file, const char *buf, size_t count, loff_t *pos) {
          printk("%.*s", count, buf); return count;
    }
// Remove the file in /proc during exit
          remove_proc_entry("hello", NULL);
```

Getting Data to LKM

```
// Add required header file
     #include <linux/uaccess.h>

// Add global variable to store incoming data
     static char buffer[256] = {0};
     static int buffer_len = 0;

// Update write function to store incoming data
          if(!buf || !count)
                              return -EINVAL;
          if(copy_from_user(buffer, buf, count < 256 ? count:256))
                              return -ENOBUFS;
          buffer_len = count < 256 ? count:256;
          printk(KERN_INFO "%.*s", (int)count, buf);    return buffer_len;

// Add read function to send data to user program
     static ssize_t read(struct file *file, char *buf, size_t count, loff_t *pos) {
          int ret = buffer_len;
          if(!buffer_len)
                              return 0;
          if(!buf || !count)
                              return -EINVAL;
          if(copy_to_user(buf, buffer, buffer_len))
                              return -ENOBUFS;
          printk(KERN_INFO "%.*s", (int)buffer_len, buffer);
          buffer_len = 0;
                              return ret;
     }

// Add read function pointer to file_ops
     file_ops.proc_read = read;
```

Further Reading

1. https://www.iitg.ac.in/asahu/cs421/books/LKM2.6.pdf
2. https://www.xml.com/ldd/chapter/book/
3. https://www.oreilly.com/openbook/linuxdrive3/book/
4. https://www.tldp.org/HOWTO/Module-HOWTO/index.html
5. https://lasr.cs.ucla.edu/classes/111_fall16/readings/dynamic_modules.html
6. http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction/

Command line argument passing to a module

```
#include <linux/init.h>
#include <linux/kernel.h> /* for ARRAY_SIZE() */
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/printk.h>
#include <linux/stat.h>

MODULE_LICENSE("GPL");
static short int myshort = 1;
static int myint = 420;
static long int mylong = 9999;
static char *mystring = ”hello";
static int myintarray[2] = { 420, 420 };
static int arr_argc = 0;
```

```
odule_param(myshort, short, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
MODULE_PARM_DESC(myshort, "A short integer");
module_param(myint, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(myint, "An integer");
module_param(mylong, long, S_IRUSR);
MODULE_PARM_DESC(mylong, "A long integer");
module_param(mystring, charp, 0000);
MODULE_PARM_DESC(mystring, "A character string");
module_param_array(myintarray, int, &arr_argc, 0000);
MODULE_PARM_DESC(myintarray, "An array of integers");
```

Command line argument passing to a module

```
static int __init senddata_init(void) {
    int i;
    pr_info("Sending Data\n=============\n");
    pr_info("myshort is a short integer: %hd\n", myshort);
    pr_info("myint is an integer: %d\n", myint);
    pr_info("mylong is a long integer: %ld\n", mylong);
    pr_info("mystring is a string: %s\n", mystring);
    for (i = 0; i < ARRAY_SIZE(myintarray); i++)
        pr_info("myintarray[%d] = %d\n", i, myintarray[i]);
    pr_info("got %d arguments for myintarray.\n", arr_argc);
    return 0;
}
static void __exit senddata_exit(void) {
    pr_info("Removed senddata module!\n");
}

module_init(senddata_init);
module_exit(senddata_exit);
```

```
# insmod senddata.ko
```

```
[503279.562297] Sending Data
                =============
[503279.562307] myshort is a short integer: 1
[503279.562311] myint is an integer: 420
[503279.562316] mylong is a long integer: 9999
[503279.562319] mystring is a string: hello
[503279.562322] myintarray[0] = 420
[503279.562326] myintarray[1] = 420
[503279.562329] got 0 arguments for myintarray.
```

```
# insmod senddata.ko
mystring="world" myintarray=-1
```

```
[503643.772842] Sending Data
                =============
[503643.772850] myshort is a short integer: 1
[503643.772854] myint is an integer: 420
[503643.772858] mylong is a long integer: 9999
[503643.772861] mystring is a string: world
[503643.772865] myintarray[0] = -1
[503643.772868] myintarray[1] = 420
[503643.772871] got 1 arguments for myintarray.
```

```
# rmmod ./senddata.ko
```

```
[503721.112612] Removed senddata module!
```

Thank You!