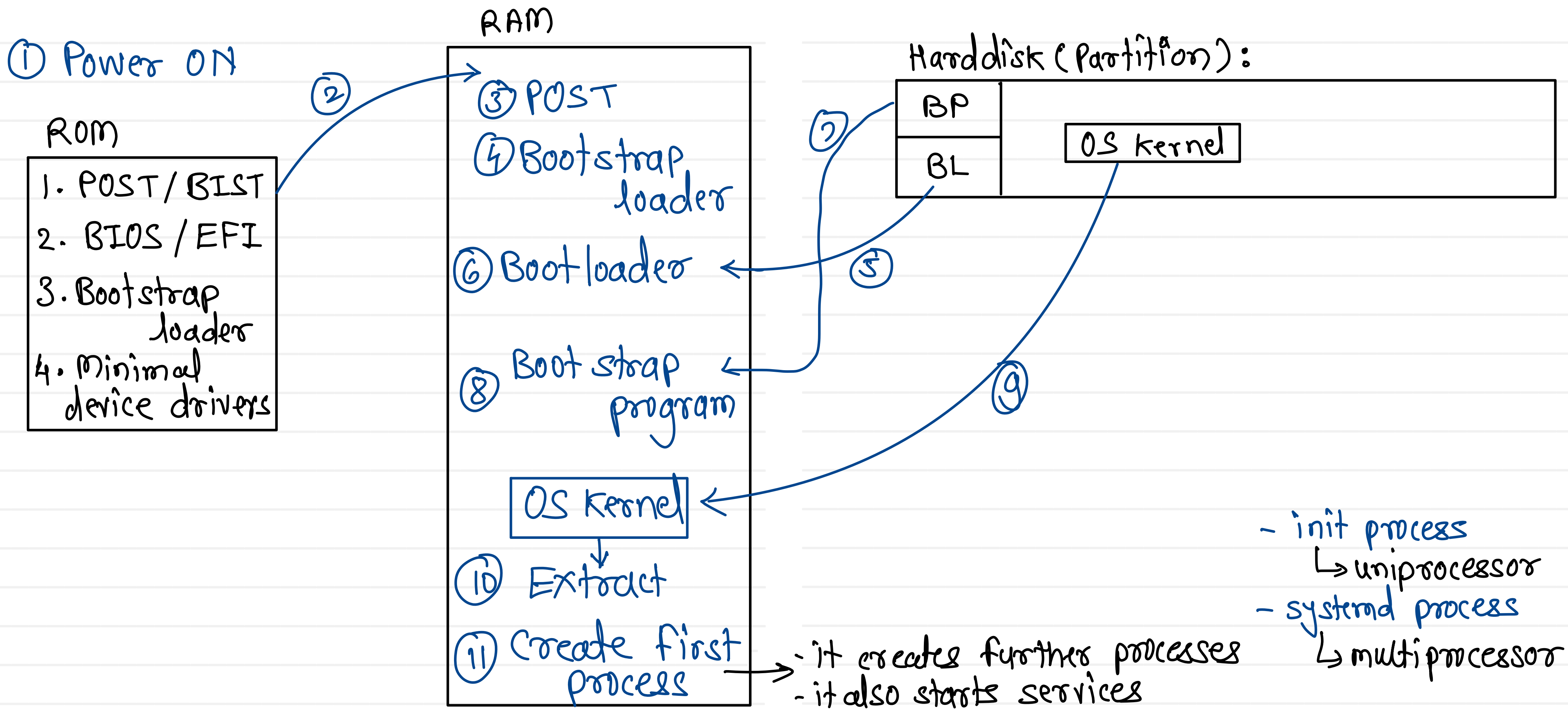
Sunbeam Institute of Information Technology
Pune and Karad

Embedded Linux Device Driver

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

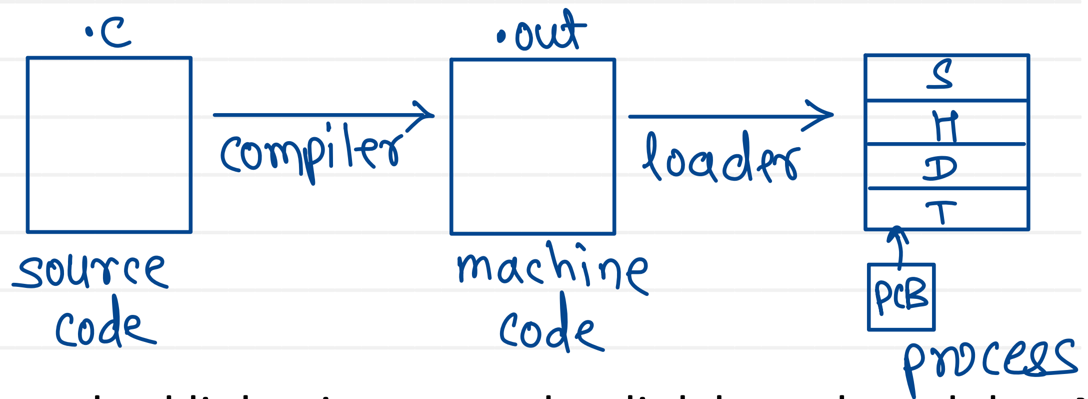


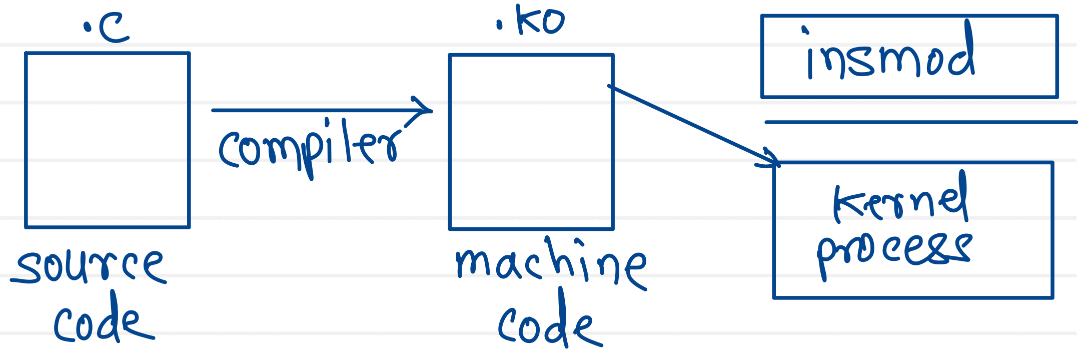
- it creates further processes
- it also starts services

- init process
  ↳ uniprocessor
- systemd process
  ↳ multiprocessor

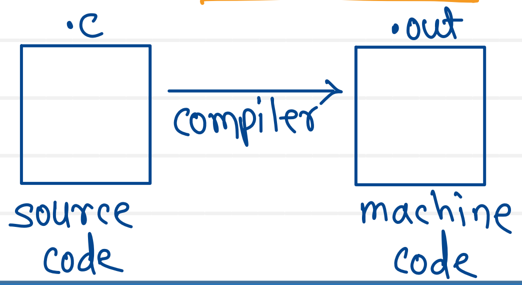
Linux Kernel module programming vs User space programming

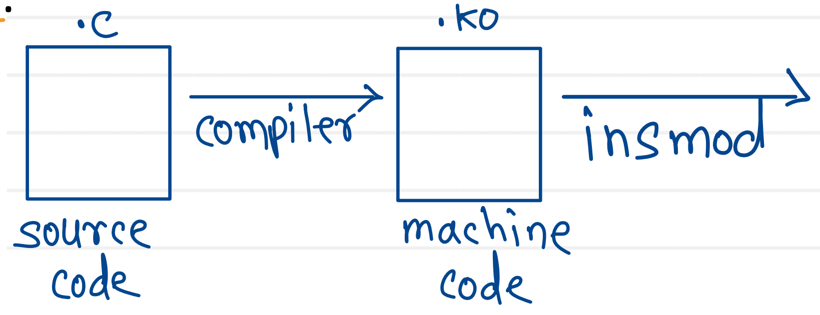
- User programs are self-executable and independent process is created for each program at runtime. However kernel modules are compiled into object files and at runtime loaded into the kernel process.





- Standard linker is not used to link kernel modules. Modules are linked dynamically with kernel using insmod like utilities. Since standard linker is not used to link kernel modules, user-space libraries (including C library) cannot be used in kernel modules. Instead kernel modules can access only functions exported by the kernel, called as Kernel APIs. Note that few commonly needed user-space functions are re-implemented in kernel space e.g. memset, strcpy, etc.





Linux Kernel module programming vs User space programming

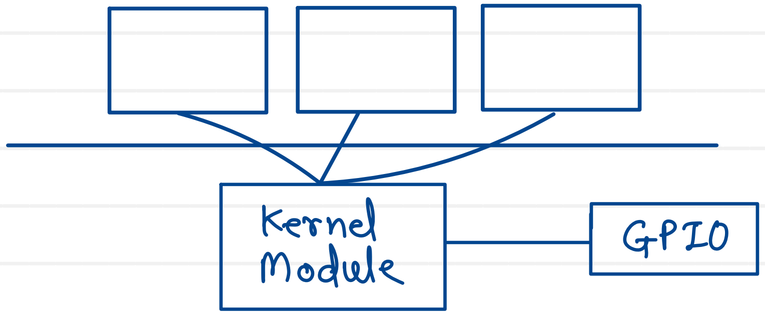
- - Typical user programs have single entry point i.e. main(). Program is terminated when main() is completed. Kernel modules have multiple entry-points. Also kernel modules are not terminated, when entry point function is finished.

```
int main(void)
{
        =
        return 0;
}
```

```
int init_module(void){   <- called at insmod

}
void cleanup_module(void){ <- called at rmmod
}
```

- User space applications are not multi-threaded and hence rarely concurrency aware (synchronization usage). However kernel modules can be used by multiple Linux applications at the same time, so they must be concurrency aware programs.



Linux Kernel module programming vs User space programming

- If resources like memory, file or network connections are not released by user space applications, they are automatically released when process terminates. Any resource leakage in kernel module is never recovered (until reboot), because modules are loaded into system process itself.

memory → dynamically allocated space → kmalloc()
synchronization objects → semaphore, mutex, spinlocks

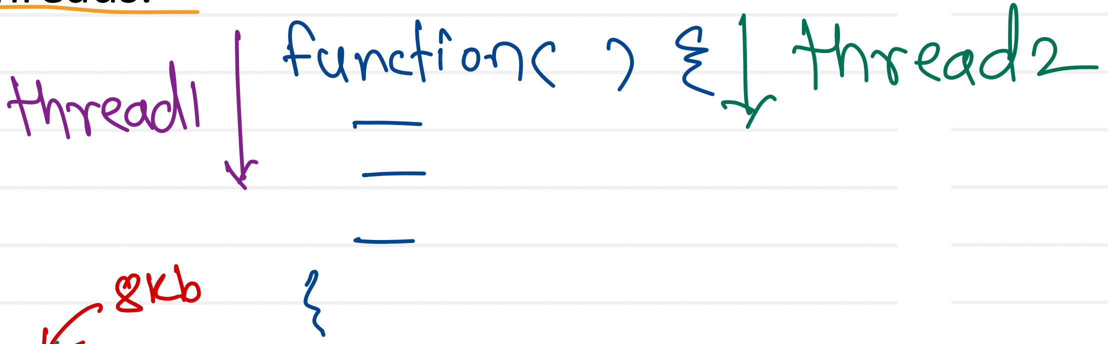
- Programming mistakes in user space programs are ignored, produce exception or terminate process. However mistakes in kernel module may lead to kernel panic and system crash. Errors will be logged under /var/log/messages (as hexdump).

dmesg → display the content of error log file

- User space applications may use FPU heavily. Resetting FPU for each operation doesn't hamper whole system performance. However kernel space code should run in real time, so using FPU in kernel module is not recommended.

Linux Kernel module programming vs User space programming

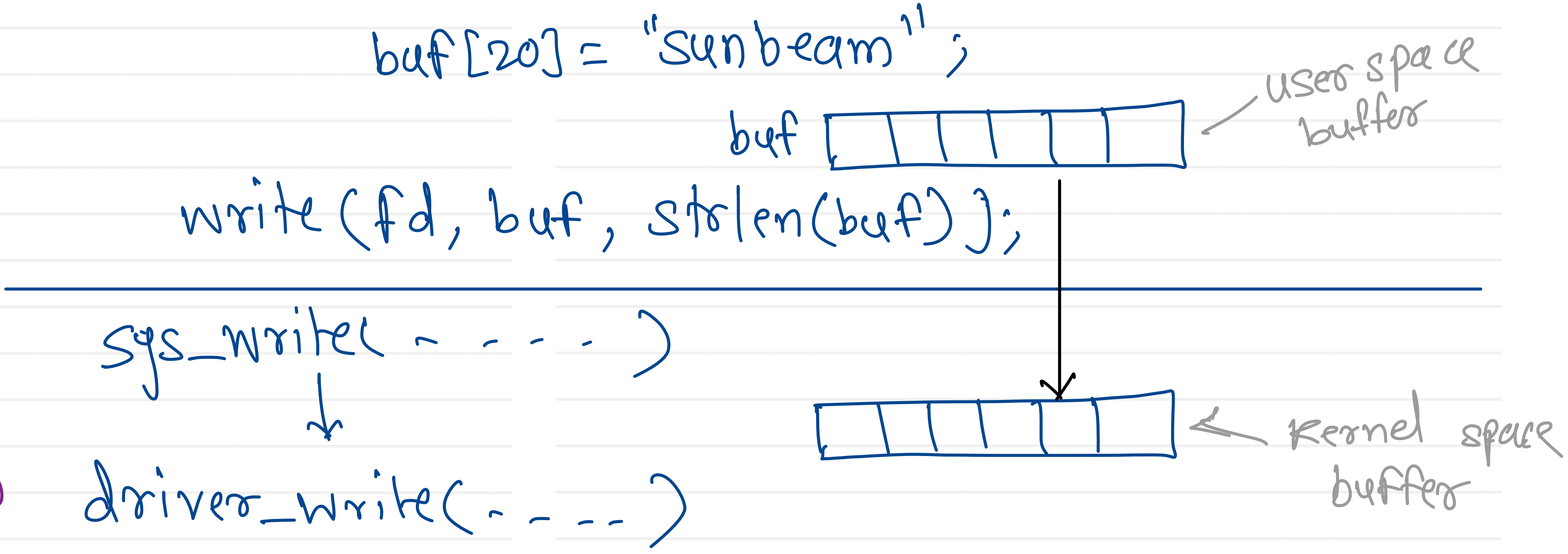
- Kernel module code is re-entrant i.e. while one thread is executing a kernel module function, another thread can also begin execution of the same. Here multiple copies of the variables will be created on kernel stack of threads.



- Size of kernel stack much smaller than user-space process stack. It is recommended not to create huge local arrays, structure variables in kernel module functions.

Linux Kernel module programming vs User space programming

- If user space application need to pass data to/from kernel module, then user space buffer should not be accessed directly from kernel space and vice-versa. The data should be copied using architecture dependent code.



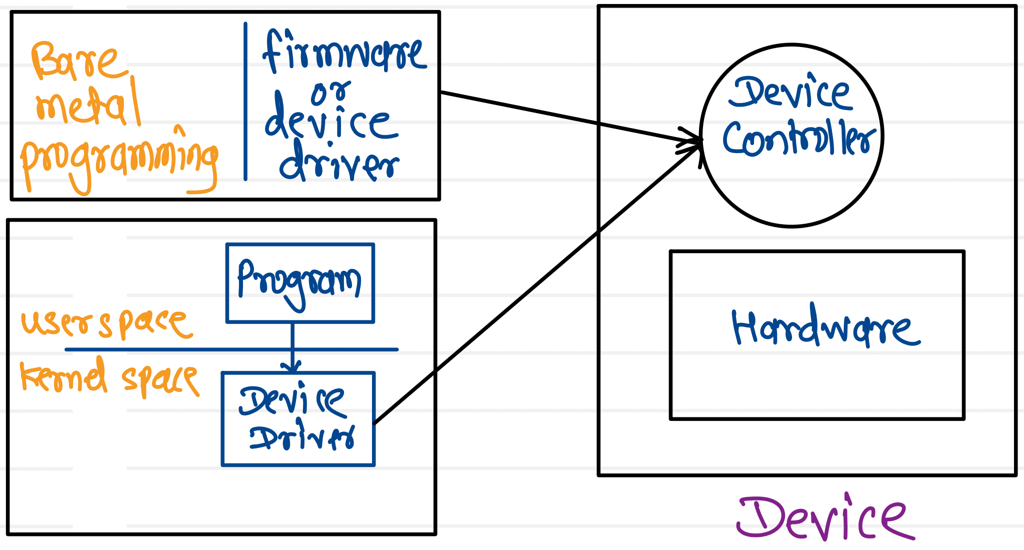
```
buf[20] = "sunbeam";
write(fd, buf, strlen(buf));
```

```
sys_write( - - - - )
    ↓
driver_write( - - - - )
```

```
copy_to_user()
copy_from_user()
```

Linux device drivers

- Device driver is a kernel module that instructs device controllers to perform the operations and also handles interrupts generated from it.



- In linux everything is treated as file.
- IO device are also treated as file and for every device one file is created into dev directory.

(devfs) pseudo file system
↓
/dev/....

Types of devices

1. Character devices ⟶ character special file
   - data is transferred character by character
   e.g. keyboard, mouse

2. Block devices ⟶ block special file.
   - data is transferred block by block.
   e.g. storage devices like harddisk, pendrives ...

Linux device drivers

- **Character device drivers** (raw device drivers)
  - Char devices transfer data in byte by byte manner. So device drivers are implemented to read/writer data as stream of bytes.
  - They support four major operations i.e. open(), close(), read() and write(). Example: Serial port, parallel port, keyboard, tty, etc.
- **Block device drivers** (fine/cooked device driver)
  - Block devices transfer data as bunch of bytes i.e. block by block. Size of block is typically 512 Bytes. Support major operations
  - open(), close(), read(), write() and lseek(). Example: All mass storage devices.
- **Network device drivers**
  - Network drivers are responsible for packets transmit and receive, however network protocols are implemented up in network stack.
  - Unlike character and block devices network device entry is not done under /dev.

Kernel module compilation

- Create Makefile for compiling kernel module.
  - obj-m = hello.o
- Compile the kernel module using kernel Makefile.
  - make -C /lib/modules/`uname -r`/build M=`pwd` modules
- Generated files
  - hello.mod.c, hello.o, hello.mod, hello.mod.o, Module.symvers, modules.order

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com