Unit - VI

Linux Kernel Module Programming -

What are kernel modules ?

Kernel modules are piece of code, that can be loaded & unloaded from kernel on demand.

Kernel modules offers an easy way to extend the functionality of base kernel without having to rebuild or recompile the kernel again. Most of the drivers are implemented as a Linux kernel modules. When those drivers are not needed, we can unload only that specific driver, which will reduce the kernel image size.

The kernel module will have a .ko extension. On a normal linux system, the kernel modules will reside inside /lib/modules/<kernel_version>/kernel/directory.

I] Utilities to Manipulate Kernel Modules -

1] lsmod -

List Modules that loaded already.

lsmod command will list modules that are already loaded in the kernel as shown below:

2] insmod -

Invest module into kernel

insmod command will invest a new module into the kernel as shown below.

```
# insmod /lib/modules/3.5.0-19-generic/kernel/fs/squashfs/squashfs.ko
# lsmod | grep "squash"
squashfs        85884  0
```

3] modinfo -

Display module info.

modinfo command will display information about a kernel module as shown below.

```
# modinfo /lib/modules/3.5.0-19-generic/kernel/fs/squashfs/squashfs.ko
```

4] rmmod -

Remove module from kernel.

rmmod command will remove a module from the kernel. You cannot remove a module which is already used by any program.

```
# rmmod squshfs.ko
```

5) modprobe -

Add or remove modules from the kernel.

modprobe is an intelligent command which will load/unload modules based on the dependancy between modules. Refer to modprobe commands for more detailed examples.

II) Write a Simple Hello World Kernel Module.

1] Installing the linux headers -

You need to install the linux headers first as shown below. Depending on your distro, use apt-get or yum.

```
# apt-get install build-essential linux-headers-$(uname -r)
```

2) Hello World Module Source Code -

Next, create the following hello c module in C programming language.

```
#include <linux/module.h> //included for all kernel modules.
# include <linux/kernel.h> // included for KERN-INFO.
# include <linux/init.h> //included for -init and -exit
```

DATE :

```
MODULE_LICENSE ("GPL")
MODULE_AUTHOR ("Lakshmanan");
MODULE_DESCRIPTION ("A simple Hello World module");
static int __init hello_init (void)
{
    printk (KERN_INFO "Hello world!\n");
    return 0;
  // Non-zero return means that the module couldn't be loaded.
}
static void __exit hello_cleanup (void)
{
    printk (KERN_INFO "cleaning up module.\n");
}
module_init (hello_init);
module_exit (hello_cleanup);
```

Warning: All kernel modules will operate on kernel space, a highly privileged mode. So be careful with what you write in a kernel module.

3] Create Makefile to compile kernel Module.

The following makefile can be used to compile the above basic hello world kernel module.

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

Use the make command to compile hello world kernel module as shown below.

```
# make
make -C /lib/modules/3.5.0-19-generic/build M=/home/
lakshmanan/a modules
make [1]: Entering directory /usr/src/linux-headers-
3.5.0-19-generic'
  CC [M] /home/lakshmanan/a/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC     /home/lakshmanan/a/hello.mod.o
  LD [M] /home/lakshamanan/a/hello.ko
make [1]: Leaving directory /usr/src/linux-headers-
3.5.0-19-generic'
```

The above will create hello.ko file, which is our sample kernel module.

4] Insert or Remove the Sample Kernel Module –

Now that we have our hello.ko file we can insert this module to the kernel by using insmod command as shown below.

```
# insmod hello.ko
# dmesg | tail -1
[ 8394.731865] Hello world!
# rmmod hello.ko
# dmesg | tail -1
[ 8707.989319] Cleaning up module.
```

When a module is inserted into the kernel, the module_init macro will be invoked, which will call the function hello_init. Similarly, when the module is removed with rmmod, module_exit macro will be invoked, which will call the hello_exit. Using dmesg command, we can see the output from the sample kernel module.

DATE :

Please note that printk is a function which is defined in kernel, & it behaves similar to the printf in the IO library. Remember that you cannot use any of the library functions from the kernel module.

Embedded systems –

The term embedded system refers to the use of electronics & software within a product, as opposed to a general-purpose computer, such as a laptop or desktop system.

Embedded system – A combination of computer hardware & software, & perhaps additional mechanical or other parts, designed to perform a dedicated function. In many cases, embedded systems are part of a larger system or product, as in the case of an antilock braking system in a car.

* Possible Organization of an Embedded System.

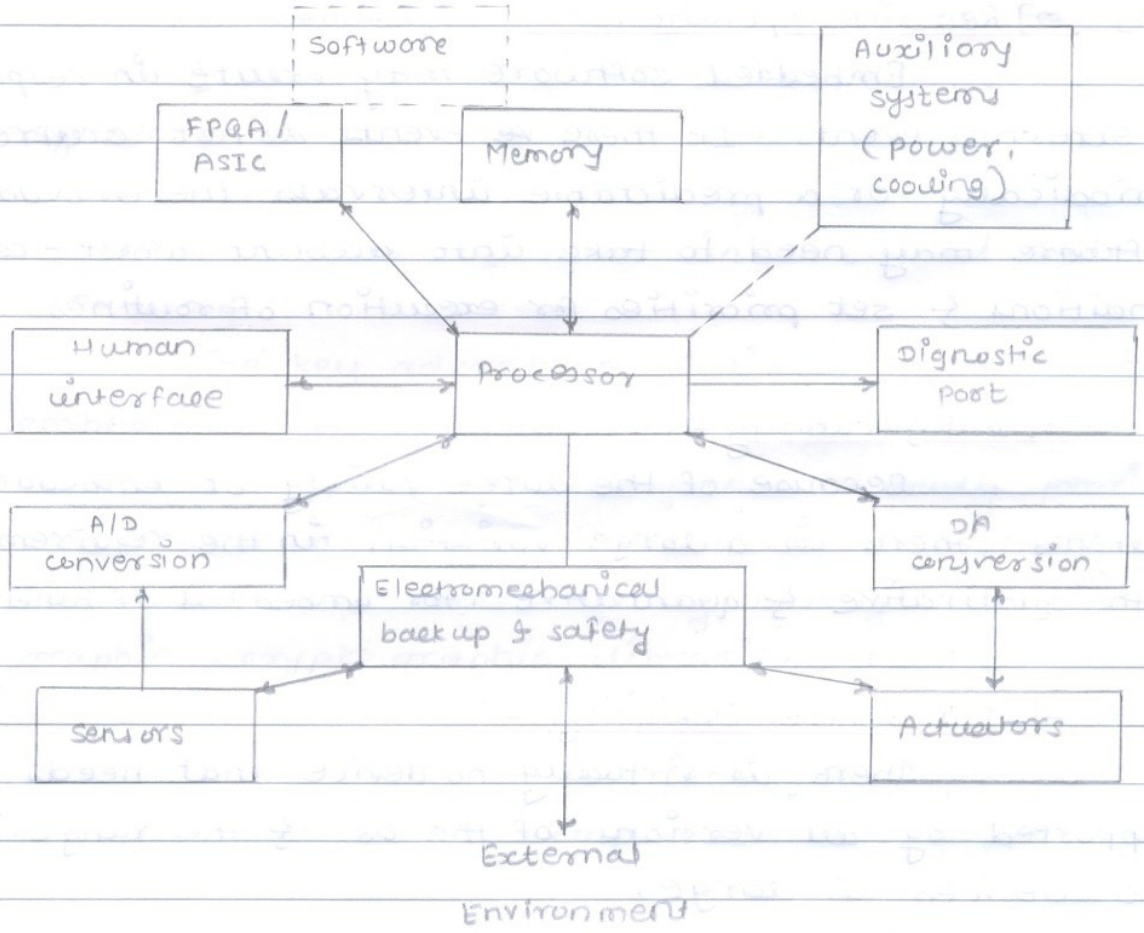


Fig- Possible organization of an Embedded system.

Characteristics of Embedded Operating Systems -

A simple embedded system, with simple functionality may be controlled by a special purpose program or set of programs with no other software. Typically, more complex embedded systems include an OS. Although it is possible in principle to use a general-purpose OS, such as Linux, for an embedded system, constraints of memory space, power consumption & real time requirements typically dictate the use of a special-purpose OS designed for the embedded system environment.

The following are some of the unique characteristics & design requirements for embedded operating systems.

1) Real-time operation -

In many embedded systems, the correctness of a computation depends, in part, on the time at which it is delivered.

2] Reactive operation -

Embedded software may execute in response to external events. If these events do not occur periodically or a predictable intervals, the embedded software may need to take into account worst-case conditions & set priorities for execution of routines.

3] Configurability -

Because of the large variety of embedded systems, there is a large variation in the requirements, both qualitative & quantative, for embedded OS functionality

4] I/O device flexibility -

There is virtually no device that needs to be supported by all versions of the OS, & the range of I/O devices is large.

5] Streamlined protection mechanism -

Embedded systems are typically designed for a limited, well-defined functionality. Untested programs are rarely added to the software. After the software has been configured & tested, it can be assumed to be reliable.

6] Direct use of interrupts -

General-purpose operating systems typically do not permit any user process to use interrupts directly. [MARW06] lists three reasons why it is possible to let interrupts start or stop tasks (e.g. by storing the task's start address in the interrupt vector address table) rather than going through OS interrupt service routines.

Embedded Linux —

Embedded Linux is the usage of the Linux kernel & various open-source components in embeded systems.

Advantages of using Linux in embedded systems.

1) Re-using components -

- The key advantage of Linux & open-source in embedded systems is the ability to re-use components.
- The open source ecosystem already provides many components for standard features, from hardware support to network protocols, going through multimedia, graphic, cryptographic libraries, etc.
- As soon as a hardware device, or a protocol, or a feature is wide-spread enough, high chance of having open-source components that support it.

2] Low Cost -

-Free software can be duplicated on as many devices as you want, free of charge.

-If your embedded system uses only free software you can reduce the cost of software license to zero. Even the development tools are free, unless you choose a commercial embedded Linux edition.

3] Full control -

-With open source, you have the source code for all components in your system.

- Allows unlimited modifications, changes, tuning debugging, optimization, for an unlimited period of time.

- Allows to have full control over the software part of your system.

4] Quality -

-Many open-source components are widely used, on millions of systems.

-Usually higher quality than what an in-house development can produce, or even proprietary vendors.

- Allows to design your system with high-quality components at the foundations.

5] Eases testing of new features -

-Open-source being freely available, it is easy to get a piece of software & evaluate it.

-Allows to easily study several options while making a choice.

6] Community support -

- Open-source software components are developed by communities of developers & users.

- This community can provide a high-quality

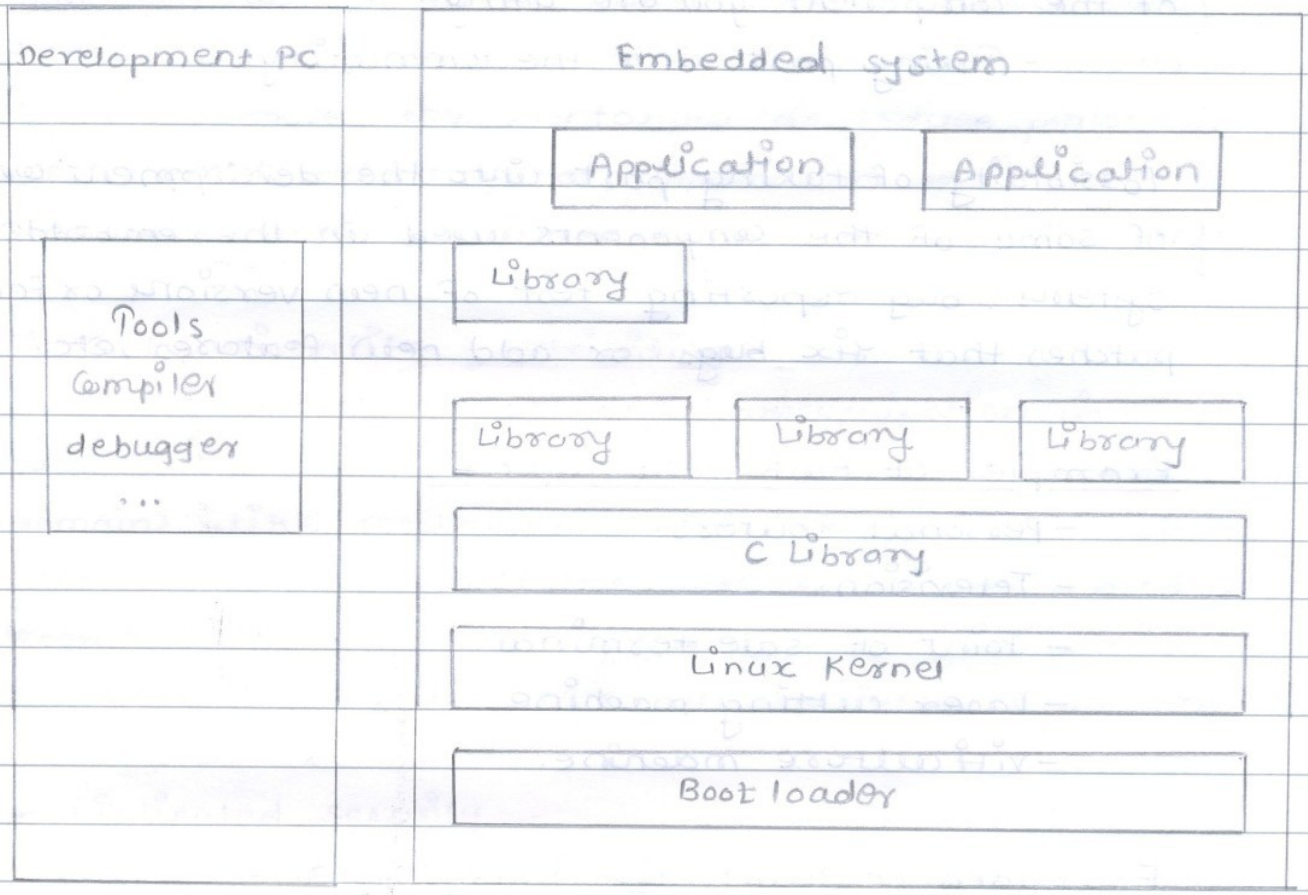
support : you can directly contact the main developers of the component you are using.
- Taking part into the community.

Possibility of taking part into the development community of some of the components used in the embedded systems : bug reporting, test of new versions or features, patches that fix bugs or add new features, etc.

Examples of Embedded Linux –
- Personal routers.
- Television.
- Point of sale terminal
- Laser cutting machine
- Viticulture machine.

Embedded hardware for Linux system -
- The Linux kernel & most other architecture-dependent component support a wide range of 32 & 64 bits architectures.
- x86 and x86-64, as found on PC platforms, but also embedded systems (multimedia, industrial).
- ARM, with hundreds of different SoC (multimedia, industrial)
- PowerPC (mainly real-time, industrial applications)
- MIPS (mainly networking applications)
- Super H (mainly set top box & multimedia applications.
- Blackfin (DSP architecture)
- Microblaze (soft-core for Xilinx FPGA)
- Coldfire, SCore, Tile, Xtensa, Cris, FRV, AVR32, M32R.

Embedded Linux system architecture –



Software components –

Cross-compilation toolchain –

Compiler that runs on the development machine, but generates code for the target.

Bootloader –

Started by the hardware, responsible for the basic initialization, loading & executing the kernel.

Linux kernel –

Contains the process & memory management, network stack, device drivers & provides services to user space applications.

C library –

The interface between the kernel & the user space applications.

Libraries & applications – Third party or in-house.

Embedded Linux Work –

Several distinct tasks are needed when deploying embedded Linux in a product:

Board support Package development

- A BSP contains a bootloader & kernel with the suitable device drivers for the targeted hardware.
- Purpose of our kernel development training.

System integration –

- Integrate all the components, bootloader, kernel, third party libraries & applications & inhouse applications into a working system.
- Purpose of this training.

Development of applications –

- Normal Linux applications, but using specifically chosen libraries.

Application specific operating systems –

Application specific customization became all the more important with the advent of application domains such as multimedia, database, parallel computing etc. which demand high performance & high functionality. Resource management & communication / sharing between applications is to be done by the operating system code running as library routines in each application. Since, these library routines can be made application-specific, the programmer has the flexibility to easily modify them whenever that is necessary for performance.

Need for customized operating systems –

Greater performance –

One of the primary motivation for such a system is greater performance. Problems with traditional operating systems is that they hide information behind high-level abstractions like processes, page table structures, IPC etc.

And hence they provide a virtual machine to the user. Thus they give a fixed implementation to all application thereby making domain specific optimizations impossible General purpose implementations do reduce performance to a great extent & that application specific virtual memory policies can increase application performance.

More Functionality & Flexibility -

Further more towards customization is motivated by a need for more functionality. New application domains like multimedia need different sets of services. Incorporating all the services is not a good idea because one may not use number of those service but still have to pay for them in the form of OS overhead

Sophisticated security -

Modern application technologies such as Java require sophisticated security in the form of access controls, that default operating systems may not provide.

Simplicity -

Simple rather than a complex kernel is easy to implement & is efficient. One may draw an analogy to RISC systems for this aspect.

Issues & problems -

- Protection & sharing
- Too much of flexibility.
- Application programmers aren't operating system designers.
- Backward compatibility & portability.

Extensible Application specific operating systems -

SPIN -

SPIN investigates kernels that allow applications to make policy decisions. All management policies are defined by embedded implementations called spindles. These spindles are dynamically loaded into the kernel.

Merits -

- Secure because of the use of type safe language & compiler checks.
- Good amount of extensibility, relatively with less burden on user.

Demerits -

- High overhead due to cross domain calls.
- Fair sharing across applications is not possible.
- Not reflective i.e. it cannot monitor & modify its own behaviours as it executes.

Aegis - an Exokernel -

Aegis uses the familiar "end-to-end" argument that applies to low-level communications protocols. Maximum opportunity for application-level resource management is assured by designing a thin exokernel. The job of exokernel is to a. track ownership of resources b. perform access control.

Merits -

- Truly extensible & flexible as it exposes the hardware.
- Very efficient.

Demerits -

- Too flexible & hence may lead to portability &

compatibility problems.
- Less secure - it expects a high degree of trust.
- Highly complex software support is needed.

SPACE -

SPACE uses processors, address spaces & a generalized exception mechanism as the basic abstractions. The processor has more than one privilege mode & thus arbitrary number of domains can exist in an address space.

Merits -
- Virtualized resources as the fundamental abstraction & hence more customizable.

Demerits -
- Building applications in terms of spaces, domains, & portals is highly complex & lacks portability.

Synthetix -

This work is a follow on from Synthesis project. It makes use of feedback from the system & use it to dynamically modify the code. The make use of the information from dynamically changing conditions. They use this technique not only for designing OS but also for seamless mobility & few others.

Merits -
- Absolutely no burden on user.
- There are no problems of portability, security etc. in fact it is implemented on HP-UX.

Demerits -
- No Full support for customization.

Kea -

In Kea kernel can be reconfigured through RPC mechanism. RPC in Kea is very general unlike others & eliminate unnecessary steps often made in a general RPC system (like in LRPC). Kea has domains, threads, inter-domain calls & portals as abstractions.

Merits -

- Provides a neat interface through user. User can use usual semantics to configure kernel.

Demerits -

- Huge penalty is incurred because of number of cross domain calls.

Vino -

VINO is a downloadable system, that uses sand-boxing [] to enforce trust & security. Sandboxing is a technique developed to allow embedding of untrusted application code inside trusted programs. Extensions are written in c++ & are compiler by a trusted compiler. This trusted code is then downloaded into the kernel. Each extension in VINO is run in the context of a transaction.

Merits -

- Performance is only slightly worse than unprotected c++.
- Extensions can be implemented in conventional languages.

Demerits -

- Security of the system is heavily dependent on compiler. There is no guard if a malicious user tricks the compiler.

Spring –

Spring is also another micro-kernel based operating system developed to build operating system out of replaceable parts.

Table - Comparison of different application specific operating systems.

Name	Overhead due to cross domain calls	Security	Complexity	Degree of customization & extensibility.	Optimization across applications
SPIN	High	Medium	Medium	High	Low
Aegis	Low	Low	High	High	High
Space	High	Medium	Medium	Medium	Low
Synthetix	Low	High	Low	Low	High
Kea	High	High	Low	Medium	High
Vino	Low	Medium	Low	Medium	High
Spring	High	Medium	Low	Medium	Low
Cache Kernel	Low	High	High	Medium	High
VM	Low	High	High	High	High

Nach OS –

Nach OS is a new teaching operating system & simulation environment. It makes it possible to give assignments that require students to write significant portions of each of the major pieces of a modern operating system: thread management, file systems, multiprogramming, virtual memory & networking. The concepts are necessary to understand the computer systems of today & of the future: concurrency & synchronization, caching & locality,

the trade-off between simplicity & performance, building reliability from unreliable components, dynamic scheduling, the power of a level of translation, distributed computing, layering & virtualization.

Basic services of Nach OS -

1] Thread management -

Basic working thread system & an implementation of semaphores. The assignment is to implement Mesa-style locks & condition variables using semaphores & then to implement solutions to a number of concurrency problems using these synchronization primitives.

2] File systems -

Real file systems can be very complex artifacts. The UNIX file system, for example, has atleast three levels of indirection - the per-process file descriptor table, the system-wide open file table, & the in-core inode table - before one even gets to disk blocks.

3] Multiprogramming -

The code to create a user address space, load a Nachos file containing an executable image into user memory, and then to run the program is provided. Initial code is restricted to running only a single user program at a time.

4] Virtual memory -

It asks students to replace their simple memory management code from the previous assignment with a true virtual memory system, that is, one that presents to each user program the

abstraction of an (almost) unlimited virtual memory size by using main memory as a cache for the disk. No need hardware or operating system components are provided.

5] Networking -

At the hardware level, we simulate the behaviour a network of workstations, each running Nachos, by connecting the UNIX processes running Nachos via sockets. The Nachos operating system & user programs running on it can communicate with other "machines" running Nachos simply by sending messages into the emulated network the transmission is actually accomplished by socket send & receive.

Introduction to service oriented operating system -

The S(o) OS project addressed future distributed systems on the level of a holistic operating system architecture by drawing from service Oriented Architectures & the strength of grids. This decouples the OS from the underlying resource infrastructure, thus making execution across an almost unlimited number of varying devices possible, independant from the actual hardware.

S(o) OS investigated alternative Operating System Architectures that base on a modular principle, where not only the individual modules may be replaced according to the respective resource specifics, but, more importantly, where also the modules may be distributed according to the executing process requirements. The operating system thus becomes a dynamic service oriented infrastructure in which each executing process may specify the required services which are detected, deployed & adapted on the fly.

S(c)OS examined how code can be segmented in a fashion that allows for concurrent execution over a heterogeneous infrastructure by combining the two approaches above. This principle implicitly supports parallelisation of code in a fashion that reduces the communication overhead. With the information about the code behaviour and dependencies, further optimisation steps can be taken related to the infrastructure's communication layout, cache & memory limitations, resource specific capabilities & restrictions etc. Depending on the size & complexity of the segments, on-the-fly adaption steps may furthermore be taken to increase heterogeneity & in particular to improve performance by exploiting the resource features.

Introduction to Ubuntu Edge OS (Ubuntu Touch) -

Ubuntu Touch (also known as Ubuntu Phone) is a mobile version of the Ubuntu operating system developed by Canonical UK Ltd & Ubuntu Community. It is designed primarily for touchscreen mobile devices such as smartphones & tablet computers.

Canocical released Ubuntu Touch 1.0, the first developer / partner version on 17 october 2013, along with Ubuntu 13.10 that "primarily supports the Galaxy Nexus & Nexus 4 phones, touch user interface & various software frameworks originally developed for Maemo & MeeGo such as ofono as telephony stack, accounts-sso for single sign-on, and Maliit for input. Utilizing libhybris the system can often be used with Linux kernels used in Android, which makes it easily ported to most recent Android smartphones.

Ubuntu Touch utilizes the same core technologies as the Ubuntu Desktop, so applications designed for the latter platform run on the former & vice versa. Additionally,

Ubuntu Desktop components come with the Ubuntu Touch system; allowing Ubuntu Touch devices to provide a full desktop experience when connected to an external monitor. Ubuntu Touch devices can be equipped with a full Ubuntu session & may change into a full desktop operating system when plugged into a docking station. If plugged the device can use all the features of Ubuntu & user can perform office work or even play ARM-ready games on such device.

Architecture :– GStreamer Playback Pipeline