Check point Linux kernel Assignment

Task1:

1. A kernel module is code that can be dynamically loaded and unloaded into the kernel of the (linux) operating system without requiring a complete system reboot or recompile.
2. Dynamic Loading and Unloading: Kernel modules can be loaded and unloaded when we want, providing the flexibility to add or remove functionalities without restarting the whole system.
   Hardware Support: Kernel modules can be used to provide device drivers, enabling support for various hardware components without bloating the kernel with all possible drivers.
   Reduced Kernel Size: With kernel modules, less essential functionalities can be offloaded from the main kernel image, ant it is leading to a smaller and more streamlined core kernel.
3. Security Concerns: Kernel modules can have direct access to kernel internals, and it can lead to potential security risks when malicious modules are loaded into the kernel.
   Performance Overhead**:** Loading and unloading kernel modules can cause some performance overhead due to the additional process of loading code and resolving dependencies, so it can make your kernel slower.
4. A kernel module is an extension of the kernel itself. The kernel provides core functionalities, including process management, memory management, I/O, and system calls. It exposes APIs that can be accessed by kernel modules to interact with its internal services.

Check point Linux kernel Assignment

When a kernel module is loaded, it becomes part of the running kernel and can use these interfaces to extend the kernel's capabilities or modify its behavior.

5. The trivial example is device drivers. device drivers provides support for various hardware devices such as graphics cards, network cards, sound cards, and peripherals.
   One more example is Filesystems. Many filesystems, such as ext4, NTFS, and VFAT, are implemented as kernel modules to enable the operating system to interact with different file storage formats.
   We can talk also about Security Modules**:** Kernel modules can enhance system security by implementing access control policies, SELinux being one example.

Task2:

The code:

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

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Asher&Shai");
MODULE_DESCRIPTION("A simple 'Hello worLd' Linux module.");
MODULE_VERSION("0.01");

static int __init lkm_hello_init(void) {
    printk(KERN_INFO "Hello, World!\n");
    return 0;
}

static void __exit lkm_by_exit(void) {
    printk(KERN_INFO "Goodbye, World!\n");
}

module_init(lkm_hello_init);
module_exit(lkm_by_exit);
```

Check point Linux kernel Assignment

The
dmesg:

```
[12331.192352] asher_shai_module: loading out-of-tree module taints kernel.
[12331.194925] asher_shai_module: module verification failed: signature and/or required key missing - tainting kernel
[12331.204210] Hello, World!
[12368.302074] Goodbye, World!
```

Task3:

The code:

me > shai-axelrod > cp_part2 > C asher_shai_module.c

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

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Asher&Shai");
MODULE_DESCRIPTION("A simple 'Hello worLd' Linux module.");
MODULE_VERSION("0.01");

static char* input_var = "World";
module_param(input_var, charp, 0);
MODULE_PARM_DESC(input_var, "A character replacing the default 'world'");
static int __init lkm_hello_init(void) {
    printk(KERN_INFO "Hello, %s!\n", input_var);
    return 0;
}

static void __exit lkm_by_exit(void) {
    printk(KERN_INFO "Goodbye, %s!\n", input_var);
}

module_init(lkm_hello_init);
module_exit(lkm_by_exit);
```

The dmesg:

```
[15865.043311] Hello, shai axel!
[15950.536949] Goodbye, shai axel!
```