NYCU Operating System - Assignment I

111550129, 林彥亨

I. Compiling a custom Linux Kernel

1) Compiled the kernel output

```
~ # uname -a
Linux (none) 6.1.0-os-111550129 #2 SMP Fri Oct 17 18:42:11 UTC 2025 riscv64 GNU/Linux
~ # cat /proc/version
Linux version 6.1.0-os-111550129 (root@fe09809c084d) (riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0, GNU ld (GNU Binutils for
Ubuntu) 2.42) #2 SMP Fri Oct 17 18:42:11 UTC 2025
```

2) Compile kernel Steps

- 下載 Linux Kernel source v6.1

  ```
  git clone https://github.com/torvalds/linux --branch v6.1 --depth 1
  ```

- 建置前清理、產生 RISC-V 預設組態

  ```
  export ARCH=riscv
  export CROSS_COMPILE=riscv64-linux-gnu-
  make defconfig
  ```

- 修改 local version

  ```
  cd /home/ubuntu/linux/
  vim Makefile
  make -j$(nproc)
  ```

- 進入Makefile找到EXTRAVERSION，更改”-os-111550129”
- 檢查

  ```
  make kernelrelease
  ```

```
root@fe09809c084d:/home/ubuntu/linux# make kernelrelease
6.1.0-os-111550129
```

3) Q&As

a) What are the main differences between the RISC-V and x86 architectures?

Ans:

RISC-V 是簡單指令集; x86 是複雜指令集

RISC	CISC
所有指令都一樣長	指令長短不一
只支援幾種 addressing mode	支援多種 addressing mode
只支援幾種 instruction type	支援多種 instruction type

b) Why do the architecture differences matter when building the kernel? What happens if you build the kernel without the correct RISC-V cross-compilation flag?

Ans:

(1) 因為不同構造的CPU有自己不同的ISA，所以當kernel的程式碼進行編譯，再經過組譯後得到的machine code必須要符合該CPU的ISA才能夠讓CPU讀懂指令，並且執行。

(2) 系統會預設用gcc來編譯成x86的machine code，並且產生的一個x86 kernel，然後丟到QEMU就會無法開啟。

c) Why is Docker used in this assignment, and what advantages does it provide? Please list at least two of them.

Ans:

優點	原因
一致性	確保不同的使用者可以使用相同的環境去建構kernel
獨立性	確保本機的系統不被影響

II. Implementing new System Calls

1) Execution Result of New System Calls

a) sys_revstr

```
/home # ./test_revstr
Ori: hello
Rev: olleh
Ori: Operating System
Rev: metsyS gnitarep0
```

```
[   14.286897] The origin string: hello
[   14.286945] The reversed string: olleh
[   14.287438] The origin string: Operating System
[   14.287458] The reversed string: metsyS gnitarep0
```

b) sys_tempbuf

```
/home # ./test_tempbuf
Hello Operating Systems
Operating Systems
```

```
[   89.653021] [tempbuf] Added: Hello
[   89.653114] [tempbuf] Added: Operating Systems
[   89.653246] [tempbuf] Hello Operating Systems
[   89.654773] [tempbuf] Removed: Hello
[   89.654836] [tempbuf] Operating Systems
```

2) How to Add System Calls to Linux kernel

a) 本次作業沒有去修改 init/Kconfig

b) 在 /kernel/Makefile 中加入

```
# Adding New System call
obj-y += revstr_syscall.o
obj-y += tempbuf_syscall.o
```

c) 在 /include/linux/syscalls.h 新增新的system call 函式的prototype宣告

```
/* New system call */
asmlinkage long sys_revstr(char __user *str, size_t len);
asmlinkage long sys_tempbuf(int mode, void __user *data, size_t size);
```

d) 在 /include/uapi/asm-generic/unistd.h header 定義新 syscall 編號去新增號碼，並且把 syscalls 的號碼調大。

```
/* Adding new system call*/
#define __NR_revstr   451
__SYSCALL(__NR_revstr, sys_revstr)
#define __NR_tempbuf  452
__SYSCALL(__NR_tempbuf, sys_tempbuf)

#undef __NR_syscalls
#define __NR_syscalls 453
```

e) 在 kernel/sys_ni.c 中加入 stub，如果新增的system calls沒啟用，kernel 仍能正常編譯與回傳 ENOSYS

```
/* new system call */
COND_SYSCALL(revstr);
COND_SYSCALL(tempbuf);
```

f) 實作 sys_revstr 和 sys_tempbuf 並且把檔案放在 /kernel/ 底下

i) sys_revstr

(1) 先宣告用來存從user space copy進來的資料的k_buf以及用來存反轉後結果的temp

```
char *k_buf;
char *temp;
k_buf = kmalloc(len + 1, GFP_KERNEL);
temp = kmalloc(len + 1, GFP_KERNEL);
```

(2) 把會用到的資料從user space copy進來

```
// copy len bytes from user space
if (copy_from_user(k_buf, str, len)) {
    kfree(k_buf);
    return -EFAULT;
}
k_buf[len] = '\0';
printk(KERN_INFO "The origin string: %s\n", k_buf);
```

(3) 進行reverse

```
// reverse
for(int i = 0; i < len; i++) {
    temp[len - 1 - i] = k_buf[i];
}
temp[len] = '\0';
printk(KERN_INFO "The reversed string: %s\n", temp);
```

(4) 把處理好的資料丟回去user space 並且印出資訊

```
// return to user space
if (copy_to_user(str, temp, len)) {
    kfree(k_buf);
    kfree(temp);
    return -EFAULT;
}
```

ii) sys_tempbuf

(1) 先宣告list的結構和 list 以及避免race condition的 mutex lock

```
// node
struct Node{
    struct list_head node;
    size_t len;
    char str[];
};

// initial a list & mutex
static LIST_HEAD(list);
static DEFINE_MUTEX(lock);
```

(2) Add部份，先檢查輸入，再配置記憶體空間用來存user space的內容，用list_add_tail()把資料插入到尾巴。

```
//ADD
static int Add(const void __user *data, size_t size)
{
    struct Node *rec;

    if(data == NULL || size == 0)
    {
        return -EFAULT;
    }

    rec = kmalloc(sizeof(*rec) + size + 1, GFP_KERNEL);

    // copy from user space
    if(copy_from_user(rec->str, data, size))
    {
        kfree(rec);
        return -EFAULT;
    }
    rec->str[size] = '\0';
    rec->len = size;

    mutex_lock(&lock);
    // Add new record
    list_add_tail(&rec->node, &list);
    mutex_unlock(&lock);

    printk(KERN_INFO "[tempbuf] Added: %s\n", rec->str);
    return 0;
}
```

(3) Remove的部份，先檢查輸入，再配置記憶體空間用來存user space的內容，list_for_each_entry_safe()來traverse整個list看有沒有第一個符合條件的，如果有符合條件的就用list_del()刪除。

```c
// Remove
static int Remove(const void __user *data, size_t size)
{
    struct Node *rec, *temp;
    char *k_buf;

    if(data == NULL || size == 0)
    {
        return -EFAULT;
    }

    k_buf = kmalloc(size, GFP_KERNEL);

    if(copy_from_user(k_buf, data, size))
    {
        kfree(k_buf);
        return -EFAULT;
    }

    mutex_lock(&lock);
    // traverse whole list
    list_for_each_entry_safe(rec, temp, &list, node)
    {
        // find the first same
        if(rec->len == size && !memcmp(rec->str, k_buf, size))
        {
            // remove
            list_del(&rec->node);
            mutex_unlock(&lock);

            printk(KERN_INFO "[tempbuf] Removed: %s\n", rec->str);
            kfree(rec);
            kfree(k_buf);
            return 0;
        }
    }
    mutex_unlock(&lock);
    kfree(k_buf);
    return -ENOENT;
}
```

(4) Print的部份，先檢查輸入，設定輸出上限512，再配置記憶體空間用來存user space的內容，用list_for_each_entry_safe()來traverse整個list。非第一個節點先嘗試放入一個空白，計算剩餘空間就copy目前的字串，若空間用盡則break。

```
if(data == NULL || size == 0)
{
    return -EFAULT;
}
k_buf = kmalloc(buf_cap + 1, GFP_KERNEL);

p = k_buf;

mutex_lock(&lock);
list_for_each_entry(rec, &list, node)
{
    size_t need;
    size_t remain;
    need = rec->len + (first ? 0 : 1);
    remain = (buf_cap > used) ? (buf_cap -used) : 0;
    if(remain == 0)
    {
        break;
    }

    if(!first)
    {
        if(remain >= 1)
        {
            *p++ = ' ';
            used += 1;
            remain -= 1;
        }
        else
        {
            break;
        }
    }

    if(remain > 0)
    {
        size_t copy = (rec->len <= remain) ? rec->len : remain;
        memcpy(p, rec->str, copy);
        p += copy;
        used += copy;
    }
    first = false;
}
mutex_unlock(&lock);
```

copy_to_user()丟回去給user space。

```
*p = '\0';
printk(KERN_INFO "[tempbuf] %s\n", k_buf);

if(copy_to_user(data, k_buf, used))
{
    kfree(k_buf);
    return -EFAULT;
}
if (used < size) {
    char nul = '\0';
    if (copy_to_user((char __user *)data + used, &nul, 1))
        pr_warn("[tempbuf] Failed to copy null terminator to user\n");
}


kfree(k_buf);
return used;
```

3) Q&As

a) What does the include/linux/syscalls.h do?

Ans:

用來宣告 Linux Kernel 的 system call 實做的 prototype 以及用來定義 system call 的.c檔裡 SYSCALL_DEFINEn 的使用方式。

b) Explain the difference between a system call and a glibc library call, then give one example that maps a glibc function to the specific system call it ultimately invokes.

Ans:

	System call	glibc library call
level	Kernel space	User space
功能	執行一些需要權限的 operation	提供程式設計者方便使用的 C 語言介面，通常會包裝對應的系統呼叫
implementation	Linux kernel 去寫的	GNU C Library寫的

例如 glibc function read()是從 user program 去呼叫這個 funciton，這個 glibc function 會對應到 SYS_read 的 system call，這個system call會呼叫 kernel 讓 kernel 去執行 sys_read() 的 function。

c) Explain the difference between static linking and dynamic linking.

Ans:

static linking 在產生執行檔之前把 user program 和會用到的 library整合進由 assembler 組譯的目的檔變成最終可執行檔；dynamic linking 是在執行的時候才讓 loader 去載入會用到的 library 的共享的 .so檔。

d) In this assignment environment, why do we have to compile the test programs with the -static flag? What would happen if we omitted it?

Ans:

QEMU 裡的 initramfs 根檔系統是極度精簡的，沒有任何共享函式庫，所以不加 -static 會被預設成 dynamic linking 的模式進行，進而產生執行時找不到檔案的狀況。