INVESTIGATION REPORT ON LINUX DEVICE DRIVER

Trainee: Hy Nhat La - ID: TR2029 – hy.la.xc@renesas.com – phone number: 0836476472

Table of Contents

I. **SW Architecture** [1] [2]

The Linux architecture can be devided in to 2 parts: User space and Kernel space.

1. User space

This is where user applications are executed.

- There is also the GNU C library which provides: the system call interface that connects to the kernel.
- The mechanism to transition between the user space application and the kernel.

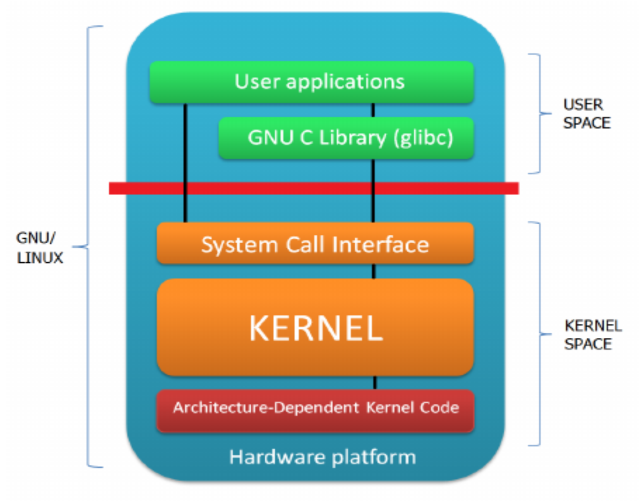


Figure 1 Linux OS architecture

2. Kernel space

The kernel space includes:

- System call interface.
- Kernel code: architecture-independent kernel code. This code is common to all the processor architectures supported bu Linux.
- Architecture-dependent code: this code serves as the processor and platform-specific code for the given architecture.

3. Privilege level

Linux cpu run in 2 modes:

- Kernel mode: a privileged and powerful mode used by the operating system kernel, has full access to the hardware and system state.
- User mode: is restricted.

The kernel also devides the virtual address space into two parts:

- Kernel space: contains kernel code, core data structure identical to all process.
- User space: contains process code, data, and memory-mapped files.

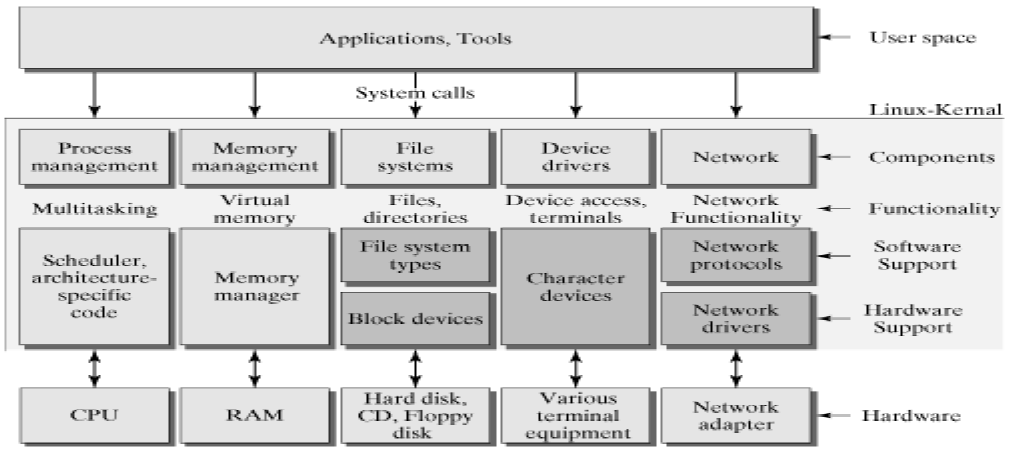


Figure 2 Linux kernel architecture

II. System call [2]

A system call is a way for programs to interact with the OS. A computer program makes a system call when it makes a request to the OS's kernel.

System call provides the services of the OS to the user programs via Application Program Interface. It provides an interface between a process and operating system to allow user-level processes to request services of the OS.

System call are the only entry points into kernel system. All programs needing resources must use system calls.

There are types of system calls:

- Process Control: fork(), exit(), wait(), …
- File Manipulation: open(), read(), write(), close(), …
- Device Manipulation: ioctl(), read(), write(), …
- Information Maintenance: getpid(), alarm(), sleep(), …
- Communication: pipe(), …
- Protection: chmod(), …

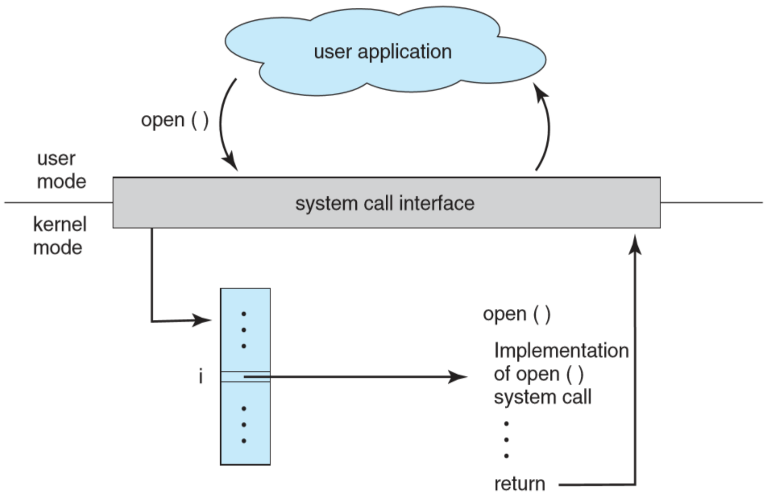


Figure 3 Linux system calls

III. Process [3] [2]

A process is the execution of a program. They can be launched when opening an application or when issuing a command through the command-line terminal.

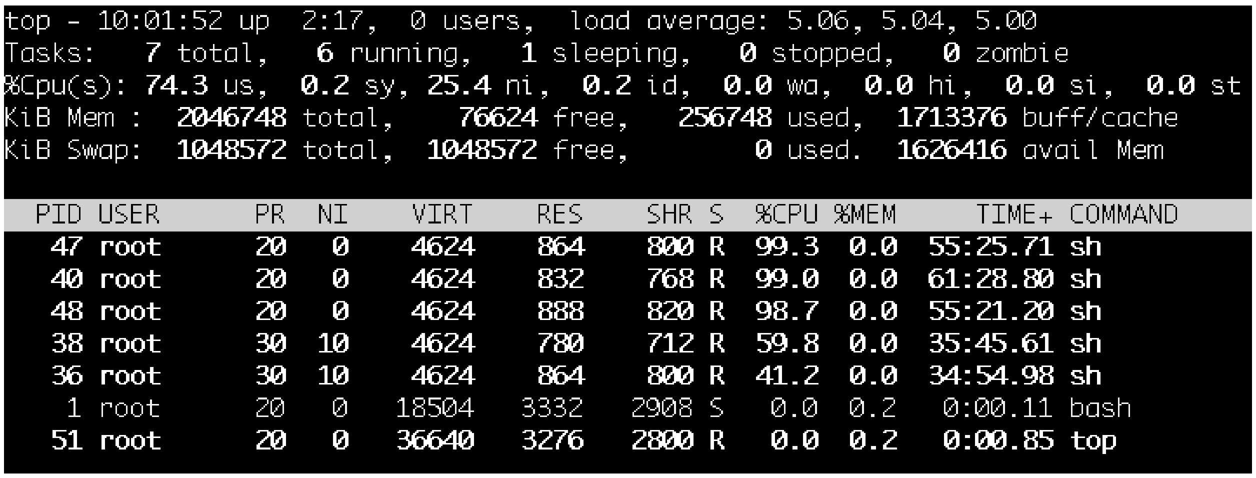
A process is defined as “an address space with one or more threads executing within that address space, and the required system resources for those threads”. Process has its own stack space, used for local variables in functions and for controlling function calls and returns.

Each Linux process is assigned a unique PID (process identification number). If there are no possible combinations left, the system can reuse old PIDs for newer processes.

IV. Scheduling [4] [2]

The Linux kernel uses a process scheduler to decide which process will receive the next time slice. It does this using the process priority. Process with a high priority get to run more often, while others such as low-priority background taks, run less frequently.

Well behaved programs are termed nice programs and in a sense this “niceness” can be measured. The operating system determines the priority of a process based on a “nice” value, which default to 0, and on the behavior of the program. The nice value is in NI column.



```
top - 10:01:52 up  2:17,  0 users,  load average: 5.06, 5.04, 5.00
Tasks:   7 total,   6 running,   1 sleeping,   0 stopped,   0 zombie
%Cpu(s): 74.3 us,  0.2 sy, 25.4 ni,  0.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  2046748 total,    76624 free,   256748 used,  1713376 buff/cache
KiB Swap:  1048572 total,  1048572 free,        0 used.  1626416 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
   47 root      20   0    4624    864    800 R  99.3  0.0  55:25.71 sh
   40 root      20   0    4624    832    768 R  99.0  0.0  61:28.80 sh
   48 root      20   0    4624    888    820 R  98.7  0.0  55:21.20 sh
   38 root      30  10    4624    780    712 R  59.8  0.0  35:45.61 sh
   36 root      30  10    4624    864    800 R  41.2  0.0  34:54.98 sh
    1 root      20   0   18504   3332   2908 S   0.0  0.2   0:00.11 bash
   51 root      20   0   36640   3276   2800 R   0.0  0.2   0:00.85 top
```

Figure 4 Running processes

We can set the process nice value using nice and adjust it using renice:

```
$ ps -1

$ nice oclock &

$ renice value_nice PID
```

V. **Thread** [5] [2]

Thread are mutiple strands of execution in a single program. It is a sequence of control within a process. All the processes have at least one thread of execution.

Thread shares the same address space of the process. Therefore, spawning a new thread within a process becomes cheap (in terms of the system resources) compared to starting a new process. Threads also can switch faster (since they have shared address space with the process) compared to the processes in the CPU. Internally, the thread has only a stack in the memory, and they share the heap (process memory) with the parent process.

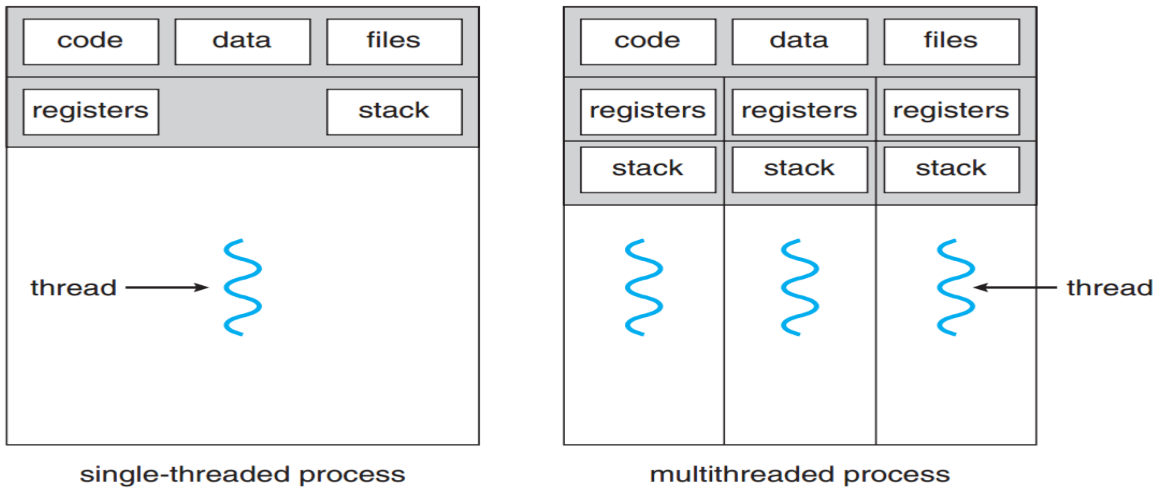


Figure 5 Threads in a process

Drawback of thread:

- Fault cased by the unintentional sharing of variables in a multi thread program.
- Debugging a multithreaded program is much harder than with a single-threaded one.
- A program that splits a large calculation into two and runs the two parts as different threads will not necessarily run more quickly on a single processor machine.

VI. **Interrupt** [6]

An interrupt is an event that alters the normal execution flow of a program and can be generated by hardware devices or even by the CPU itself. When an interrupt occurs the current flow of execution is suspended, and interrupt handler runs. After the interrupt handler runs the previous execution flow is resumed.

Interrupts can be grouped into some following categories:

- Hardware Interrupt:
  - Generated by an external event
  - Asynchronous

- Non-Maskable Interrupt:
  - Cannot be ignored
  - Signaled via NMI pin
- Maskable Interrupt:
  - Can be ignored
  - Signaled via INT pin
- Software Interrupt:
  - Generated by executing an instruction
  - Synchronous

A device supporting interrupts has an output pin used for signaling an Interrupt ReQuest. IRQ pins are connected to a device named Programmable Interrupt Controller (PIC) which is connected to CPU's INTR pin.

A PIC usually has a set of ports used to exchange information with the CPU. When a device connected to one of the PIC's IRQ lines needs CPU attention the following flow happens:

- Device raises an interrupt on the corresponding IRQn pin
- PIC converts the IRQ into a vector number and writes it to a port for CPU to read
- PIC raises an interrupt on CPU INTR pin
- PIC waits for CPU to acknowledge an interrupt before raising another interrupt
- CPU acknowledges the interrupt then it starts handling the interrupt

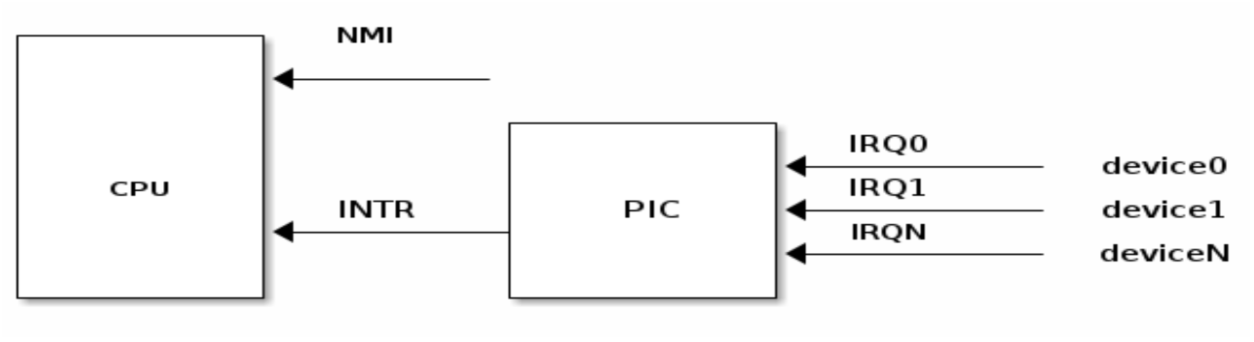


Figure 6 Programmable Interrupt Controller

References

[1] "VSUDO - Phân biệt Kernel Space và User Space," [Online]. Available: https://vsudo.net/blog/kernel-space-va-user-space.html.

[2] B. V. Company, "SW architecture of Linux OS," Jan, 2021.

[3] Hostinger, "Hostinger Tutorial - How to List Running Processes in Linux: A Beginner's Guide," [Online]. Available: https://www.hostinger.com/tutorials/vps/how-to-manage-processes-in-linux-using-command-line#:~:text=Running%20%E2%80%9Chtop%E2%80%9D%20Command-,Introduction%20to%20Linux%20Processes,multiple%20processes%20for%20different%20tasks..

[4] E. Turgeman, "Medium.com - Process Scheduling In Linux," [Online]. Available: https://medium.com/geekculture/process-scheduling-in-linux-592028a5d545.

[5] Bealdung, "Baeldung.com - Linux Process vs. Thread," [Online]. Available: https://www.baeldung.com/linux/process-vs-thread.

[6] T. k. d. community., "The Linux Kernel - Interrupts¶," [Online]. Available: https://linux-kernel-labs.github.io/refs/heads/master/lectures/interrupts.html.

[7] Apoorva, "Apoorva's Blog - Dialogues on linux kernel-1," [Online]. Available: https://rava-dosa.github.io/2018-08-05-Linux-kernel-1/.