#LEARN IN DEPTH

#Be professional in embedded system

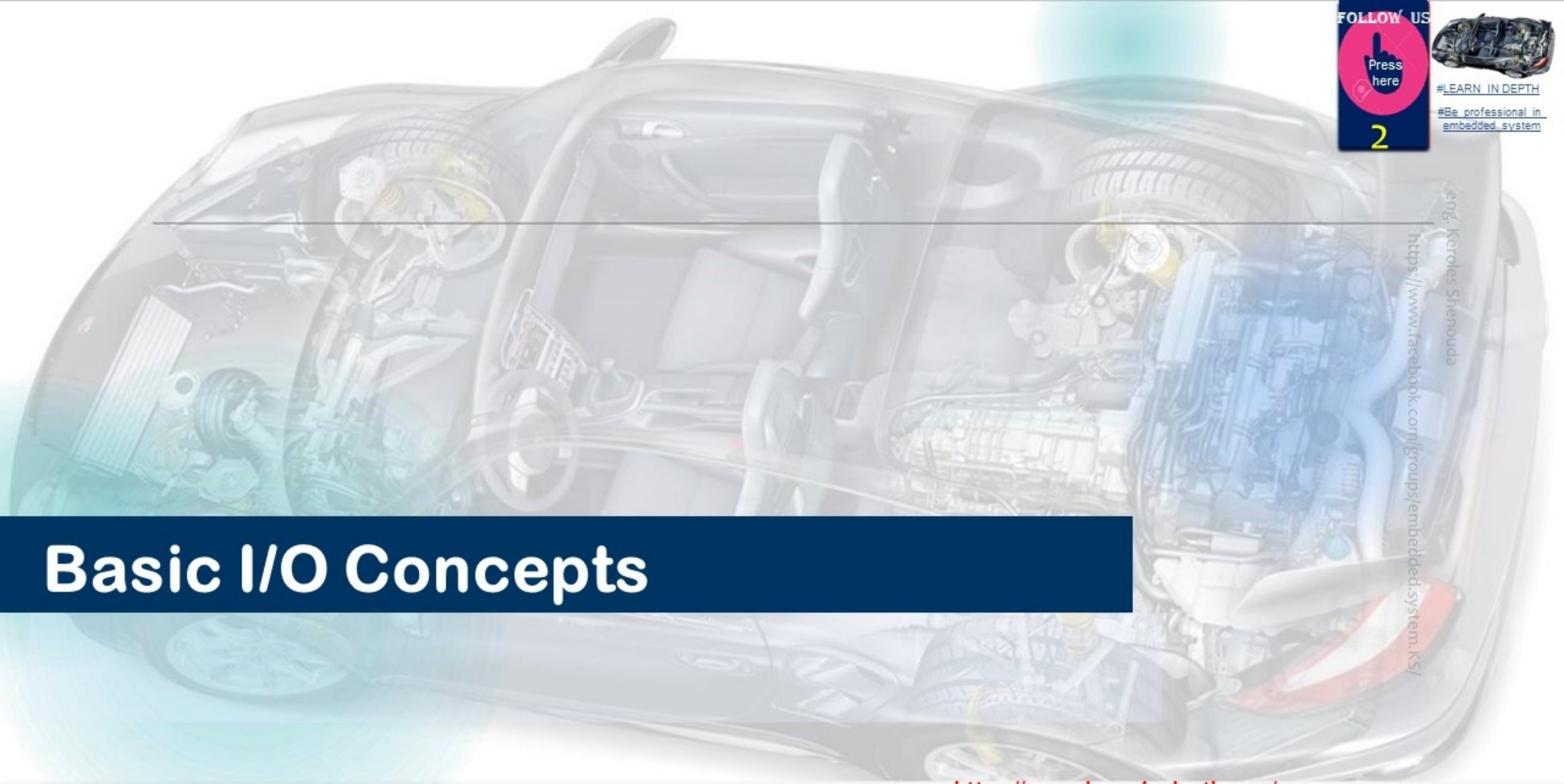


# Embedded Linux (PART 3)

- BASIC I/O CONCEPTS
- KERNEL MODULES/PROCESS MANAGEMENT IN LINUX
- OS: ABSTRACTION PROVIDER
- PROCESSES
- SIGNALS

ENG.KEROLES SHENOUDA

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

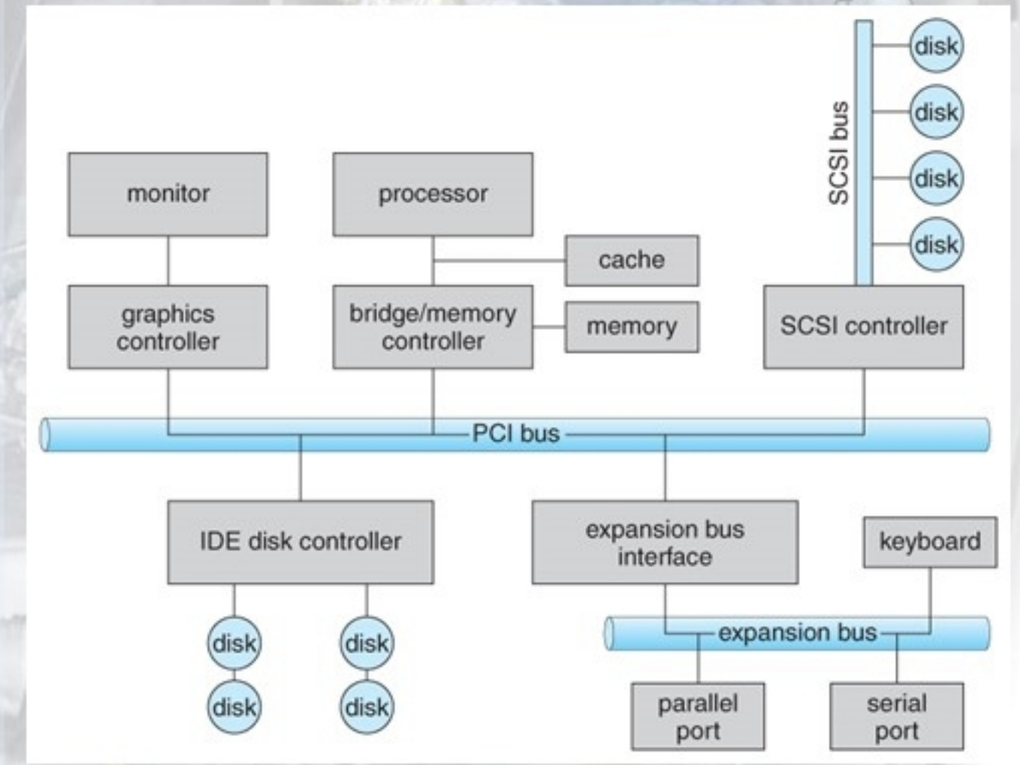



# Basic I/O Concepts

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

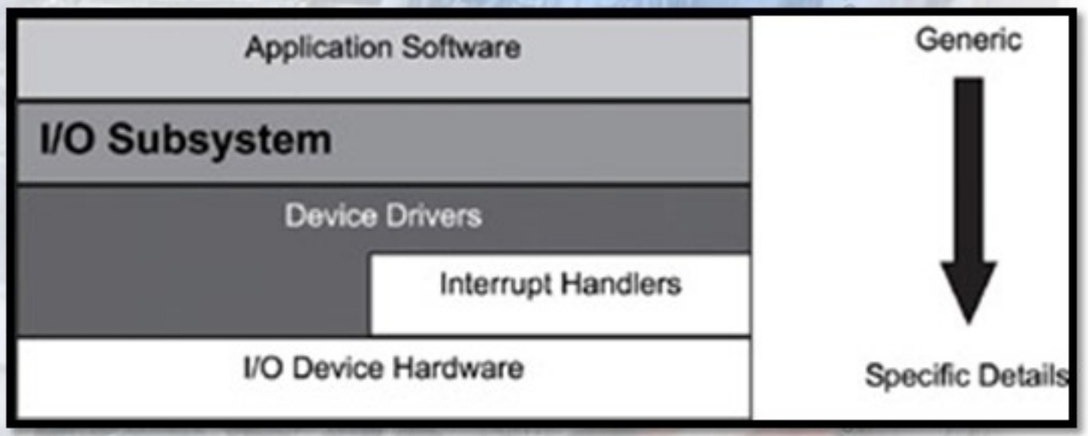
# Modern I/O Systems

- For example in computer systems many kinds of IO devices are connected via bus.
- The CPU and memory are connected via PCI bus and slow devices such as keyboards are connected by expansion bus.
- In addition to this disks are connected via SCSI Bus you interact with IO devices via device controllers
- device controllers contains a set of registers including control registers and status registers to control **IO devices.**



# Basic I/O Concepts

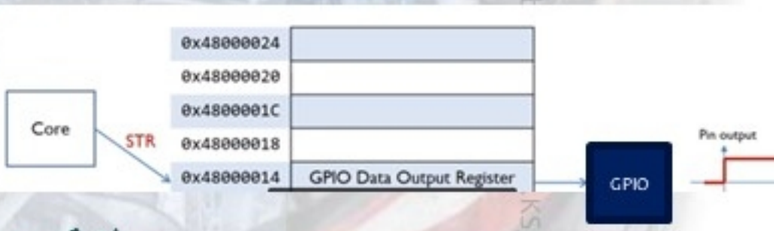
- The purpose of the I/O subsystem is to **hide** the device-specific information from the kernel as well as from the application developer and to **provide** a uniform access method to the peripheral I/O devices of the system.
- Peripheral devices are generally controlled by writing and reading its registers
- Common mechanisms to access registers
  - Operations on **memory address space**
  - Separate **I/O address space** for I/O ports (different than memory)
    - Special instructions on I/O address space
- To access those devices, **it is necessary for the developer to determine** if the device **is port mapped or memory mapped**.
- This information determines which of two methods, port-mapped I/O or memory-mapped I/O, is deployed to access an I/O device.





# The CPU control the IO devices via registers in device controllers by two ways:

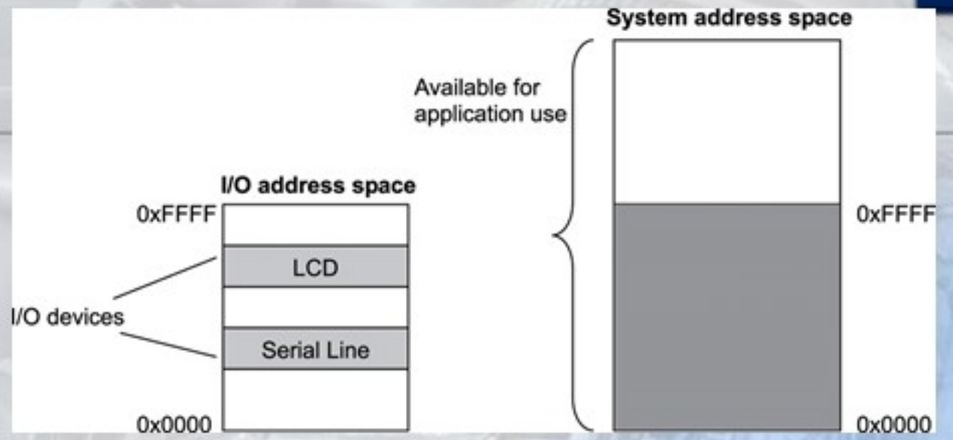

- the controller has one or more registers for data and control signals. The processor communicates with the controller by reading and writing bit patterns in these registers.
- PORT I/O:
  - One way in which this communication can occur is through the use of **special I/O instructions** that specify the transfer of a byte or word to an **I/O port** address. The I/O instruction triggers bus lines to select the proper device and to move bits into or out of a device register
  - Each device uses a different I/O port. (port number)
- Memory-mapped I/O
  - The second way the device controller can support **Memory-mapped I/O**
  - **Use native CPU load/store instructions LDR/STR Reg, [Reg, #imm]**
  - In this case, the device-control registers are mapped into the address space of the processor. The CPU executes I/O requests using the standard data-transfer instructions to read and write the device-control registers.

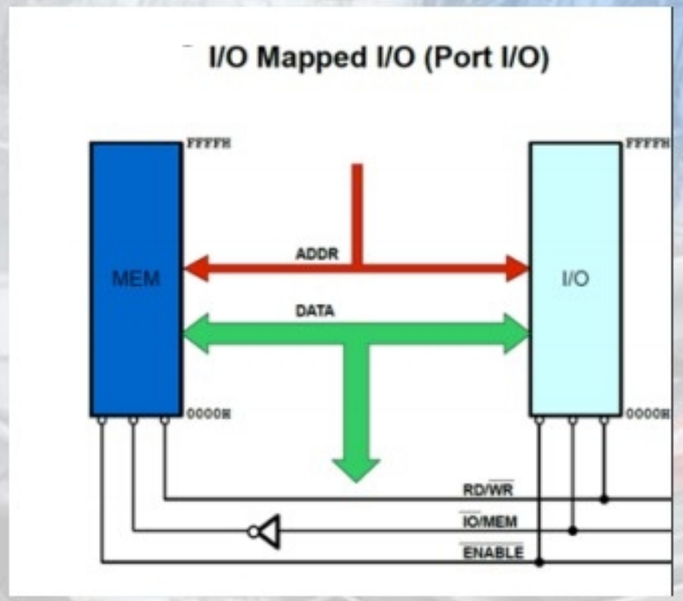



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

# port-mapped I/O

- When the **I/O device address space** is **separate** from **the system memory address space**, special processor instructions, such as the IN and OUT instructions offered by the Intel processor, are used to transfer data between the I/O device and a microprocessor register or memory.
- The **I/O device address** is **referred** to **as the port number** when specified for these special instructions.




https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

# Using I/O Ports for intel CPU

- See /proc/ioports to see the current allocation

```
0000-001f : dma1
0020-0021 : pic1
0040-0043 : timer0
0050-0053 : timer1
0060-006f : keyboard
0070-0077 : rtc
0080-008f : dma page reg
00a0-00a1 : pic2
00c0-00df : dma2
00f0-00ff : fpu
0170-0177 : ide1
```

# cat /proc/ioports for ARM

```
[  OK  ] Reached target Timers.
[  OK  ] Started triggerhappy global hotkey daemon.
[  OK  ] Started Disable WiFi if country not set.
[  OK  ] Started Check for v3d driver.
[  OK  ] Started System Logging Service.
[  OK  ] Started Save/Restore Sound Card State.
[  OK  ] Started dhcpcd on all interfaces.
[  OK  ] Started Avahi mDNS/DNS-SD Stack.
[  OK  ] Started Login Service.
[  OK  ] Started LSB: Switch to ondemand cpu…nor (unless shift key is pressed).
[  OK  ] Started LSB: Autogenerate and use a swap file.
[  OK  ] Started Raise network interfaces.
[  OK  ] Reached target Network.
         Starting OpenBSD Secure Shell server...
         Starting Permit User Sessions...
[  OK  ] Reached target Network is Online.
         Starting /etc/rc.local Compatibility...
         Starting Daily apt download activities...
         out waiting for device dev-serial1.device.
         Started BluezALSA proxy.
[  OK  ] Started Permit User Sessions.
[  OK  ] Started /etc/rc.local Compatibility.
         Starting Hold until boot process finishes up...
         Starting Light Display Manager...
         Starting Terminate Plymouth Boot Screen...

Raspbian GNU/Linux 9 raspberrypi ttyAMA0
raspberrypi login: pi
Password:
Last login: Fri Nov 22 13:47:41 UTC 2019 on tty1
Linux raspberrypi 5.4.0-rc8 #1 SMP Tue Nov 19 06:34:25 EET 2019 armv7l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.



SSH is enabled and the default password for the 'pi' user has not been changed.
This is a security risk - please login as the 'pi' user and type 'passwd' to set a new password.

pi@raspberrypi:~$
pi@raspberrypi:~$
pi@raspberrypi:~$ uname -a
Linux raspberrypi 5.4.0-rc8 #1 SMP Tue Nov 19 06:34:25 EET 2019 armv7l GNU/Linux
pi@raspberrypi:~$
```

Ubuntu Software



# cat /proc/ioports for ARM Cont.

```
pi@raspberrypi:~$ cat /proc/ioports
pi@raspberrypi:~$
```

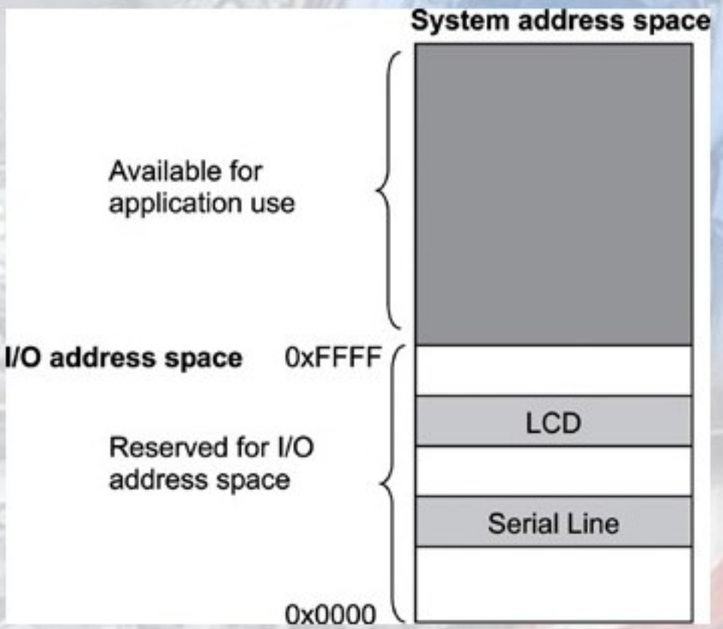
Nothing found ☺ Learn-in-depth

Embedded Linux

ENG.KEROLES SHENOUDA

# Memory-mapped I/O

- the device address is part of the system memory address space.
- Any machine instruction that is encoded to transfer data between a memory location and the processor or between two memory locations can potentially be used to access the I/O device.
- The I/O device is treated as if it were another memory location. Because the I/O address space occupies a range in the system memory address space
- **The registers in device controller are mapped into a physical address space.**
- **IO is accomplished with load and store instructions, which the** CPU generally uses for memory access.



# I/O Memory Allocation and Mapping on intel CPU

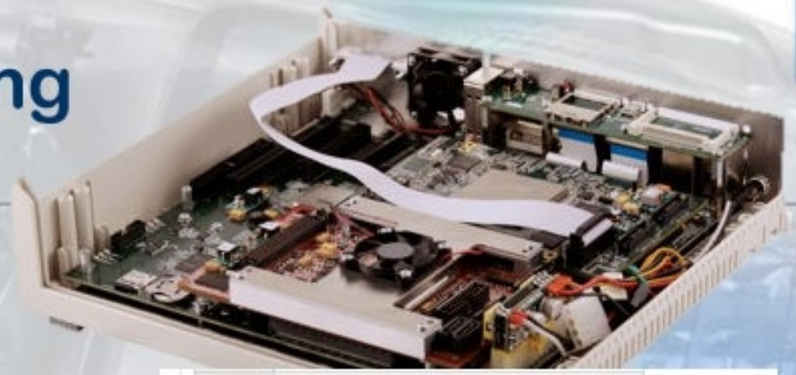
- more /proc/iomem

```
00000000-0009b7ff : System RAM
0009b800-0009ffff : reserved
000a0000-000bffff : Video RAM area
000c0000-000c7fff : Video ROM
000c8000-000c8fff : Adapter ROM
000f0000-000fffff : System ROM
00100000-7ff6ffff : System RAM
  00100000-002c7f2f : Kernel code
  002c7f30-003822ff : Kernel data
7ff70000-7ff77fff : ACPI Tables
7ff78000-7ff7ffff : ACPI Non-volatile Storage
```

# IO on Intel Desktop

```
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/images_prebuilt
/VexpressA9$ sudo cat /proc/iomem
[sudo] password for embedded_system_ks:
00000000-00000fff : reserved
00001000-0009fbff : System RAM
0009fc00-0009ffff : reserved
000a0000-000bffff : PCI Bus 0000:00
  000a0000-000bffff : Video RAM area
000c0000-000c7fff : Video ROM
000e2000-000e2fff : Adapter ROM
000f0000-000fffff : reserved
  000f0000-000fffff : System ROM
00100000-dffeffff : System RAM
  05000000-05842e9e : Kernel code
  05842e9f-05c3e93f : Kernel data
  05d44000-05e28fff : Kernel bss
dfff0000-dfffffff : ACPI Tables
e0000000-fdffffff : PCI Bus 0000:00
  e0000000-e7ffffff : 0000:00:02.0
  f0000000-f001ffff : 0000:00:03.0
    f0000000-f001ffff : e1000
  f0400000-f07fffff : 0000:00:04.0
    f0400000-f07fffff : vboxguest
  f0800000-f0803fff : 0000:00:04.0
  f0804000-f0804fff : 0000:00:06.0
    f0804000-f0804fff : ohci_hcd
  f0820000-f083ffff : 0000:00:08.0
    f0820000-f083ffff : e1000
  f0840000-f0841fff : 0000:00:0d.0
    f0840000-f0841fff : ahci
fec00000-fec00fff : reserved
  fec00000-fec003ff : IOAPIC 0
fee00000-fee00fff : Local APIC
  fee00000-fee00fff : reserved
fffc0000-ffffffff : reserved
100000000-8e21fffff : System RAM
8e2200000-8e3ffffff : RAM buffer
```

# I/O Memory Allocation and Mapping on ARM SoC based



```
pi@raspberrypi:~$ sudo cat /proc/iomem
10000008-1000000b : basic-mmio-gpio.1.auto
10000048-1000004b : basic-mmio-gpio.2.auto
1000004c-1000004f : basic-mmio-gpio.3.auto
100000a0-100000ab : vexpress-syscfg.6.auto
10002000-10002fff : 10002000.i2c
10004000-10004fff : aaci@4000
  10004000-10004fff : aaci-pl041
10005000-10005fff : mmci@5000
  10005000-10005fff : 10005000.mmci
10006000-10006fff : kmi@6000
  10006000-10006fff : kmi-pl050
10007000-10007fff : kmi@7000
  10007000-10007fff : kmi-pl050
10009000-10009fff : uart@9000
  10009000-10009fff : 10009000.uart
1000a000-1000afff : uart@a000
  1000a000-1000afff : 1000a000.uart
1000b000-1000bfff : uart@b000
  1000b000-1000bfff : 1000b000.uart
1000c000-1000cfff : uart@c000
  1000c000-1000cfff : 1000c000.uart
10011000-10011fff : timer@11000
10012000-10012fff : timer@12000
10013600-100137ff : 10013600.virtio_mmio
10016000-10016fff : 10016000.i2c
10017000-10017fff : rtc@17000
  10017000-10017fff : rtc-pl031
1001f000-1001ffff : clcd@1f000
10020000-10020fff : clcd@10020000
40000000-43ffffff : 40000000.flash
44000000-47ffffff : 40000000.flash
48000000-49ffffff : 48000000.psram
4e000000-4e00ffff : smsc911x
60000000-7fffffff : System RAM
  60008000-609fffff : Kernel code
  60b00000-60b91a6f : Kernel data
```

```
86  [VE_SYSREGS] = 0x10000000,
87  [VE_SP810] = 0x10001000,
88  [VE_SERIALPCI] = 0x10002000,
89  [VE_PL041] = 0x10004000,
90  [VE_MMCI] = 0x10005000,
91  [VE_KMI0] = 0x10006000,
92  [VE_KMI1] = 0x10007000,
93  [VE_UART0] = 0x10009000,
94  [VE_UART1] = 0x1000a000,
95  [VE_UART2] = 0x1000b000,
96  [VE_UART3] = 0x1000c000,
97  [VE_WDT] = 0x1000f000,
98  [VE_TIMER01] = 0x10011000,
99  [VE_TIMER23] = 0x10012000,
100 [VE_VIRTIO] = 0x10013000,
101 [VE_SERIALDVI] = 0x10016000,
102 [VE_RTC] = 0x10017000,
103 [VE_COMPACTFLASH] = 0x1001a000,
104 [VE_CLCD] = 0x1001f000,
105 /* CS0: 0x40000000 .. 0x44000000 */
106 [VE_NORFLASH0] = 0x40000000,
107 /* CS1: 0x44000000 .. 0x48000000 */
```

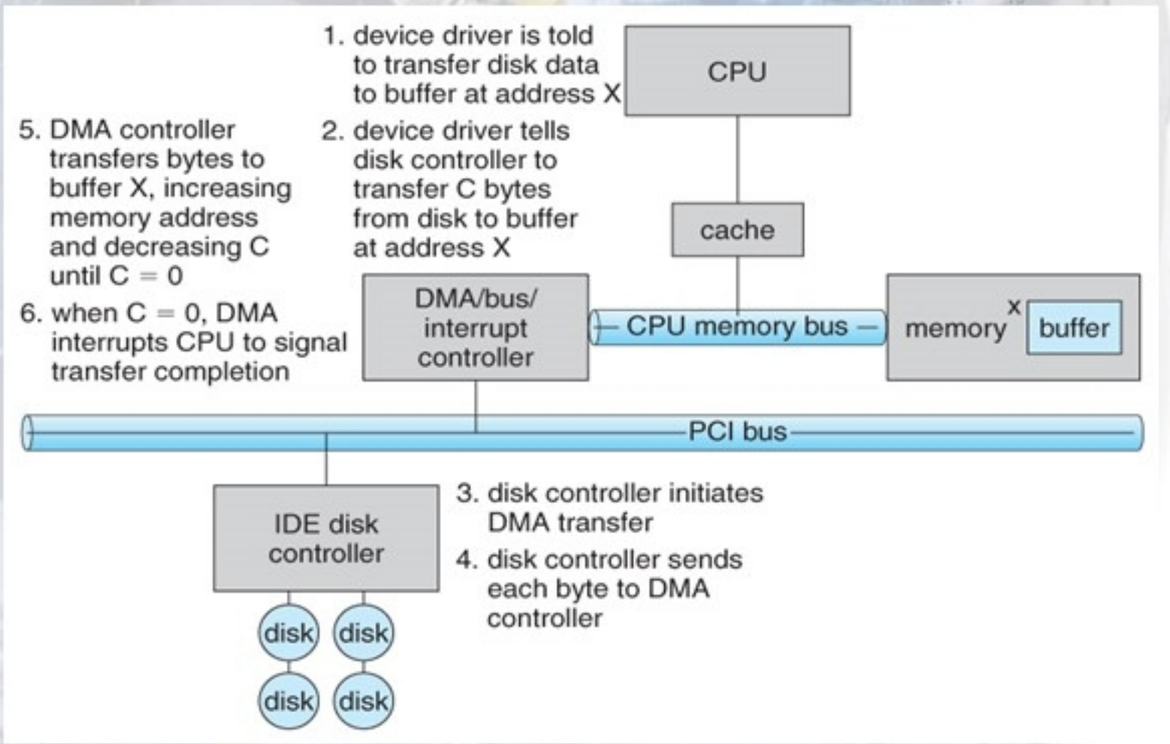


# Comparison - Memory-mapped vs port-mapped

| Memory-mapped IO | Port-mapped IO |
|---|---|
| **Same address bus** to address memory and I/O devices | **Different address spaces** for memory and I/O devices |
| Access to the I/O devices using **regular instructions** | Uses **a special class of CPU instructions** to access I/O devices |
| **Most widely used I/O** method | **x86 Intel** microprocessors - IN and OUT instructions |

# DMA Direct Memory Access

- Used to avoid programmed I/O for large data movement.
- whether it is done by using **I O ports** or **memory mapped IO** if IO devices should transfer huge data with disks.
  - The CPU will become much heavier.
  - The DMA technique is introduced to solve this issue.
- In the DMA technique it provides a special DMA controller that can access memory bus directly and bypasses CPU to transfer data between IO device and memory.




https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/



# Polling Vs. Interrupt

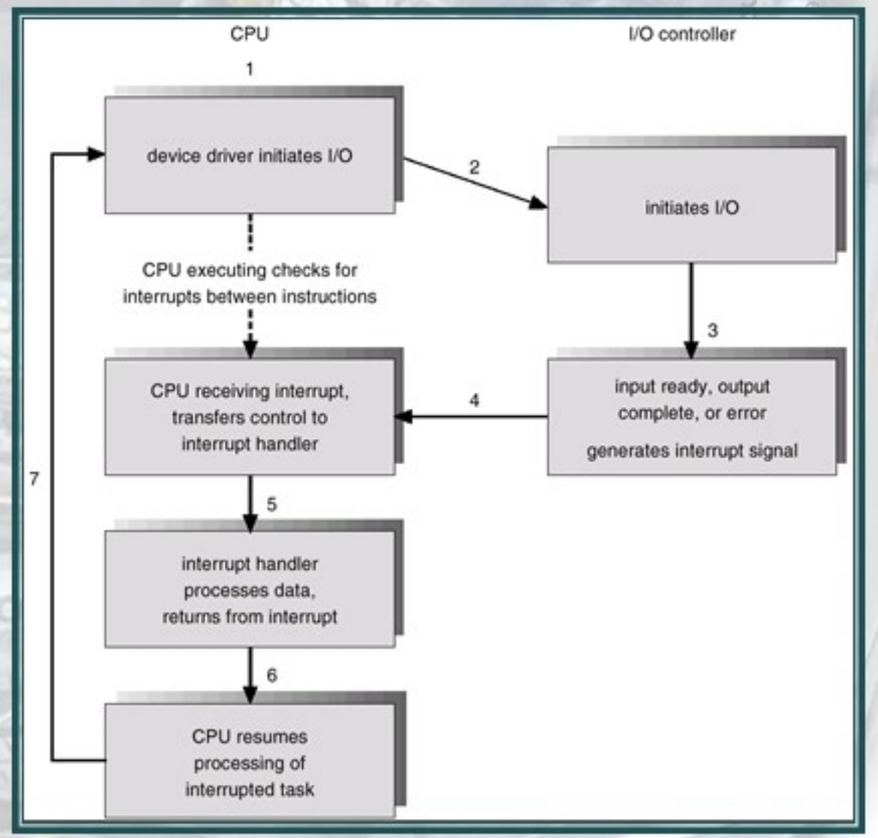
- Operating systems need to know when IO devices complete IO operation.
- There are two ways of doing this interrupting and polling
- Polling
  - IO devices put completion information in status registers.
  - OS periodically checks a device-specific status register.
  - It may waste many cycles if infrequent or unpredictable I/O operations.
- There are many devices that use the **combination** of the two methods interrupt and polling
- Example: High bandwidth network devices
  - Check the first incoming packet with interrupt method and the following packets with polling method.

#LEARN IN DEPTH

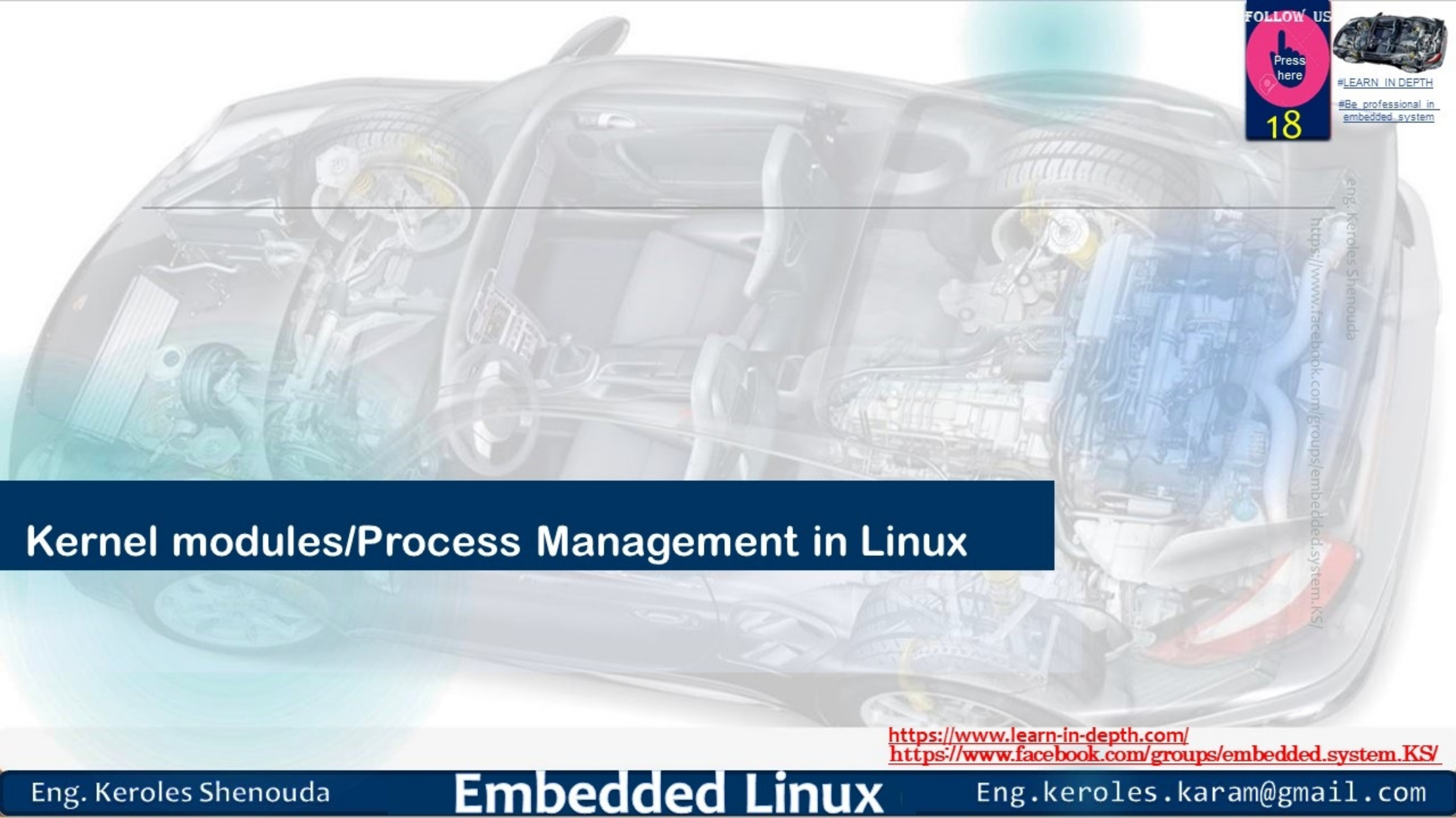
#Be professional in embedded system

# Interrupt

- CPU Interrupt request line triggered by I/O device
- Interrupt handler receives interrupts
- Maskable to ignore or delay some interrupts
- Interrupt vector to dispatch interrupt to correct handler
  - Based on priority
  - Some unmaskable
- Interrupt mechanism also used for exceptions and paging.
- **In general the interrupt handler is implemented in a device driver.**




https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/




# Kernel modules/Process Management in Linux

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/



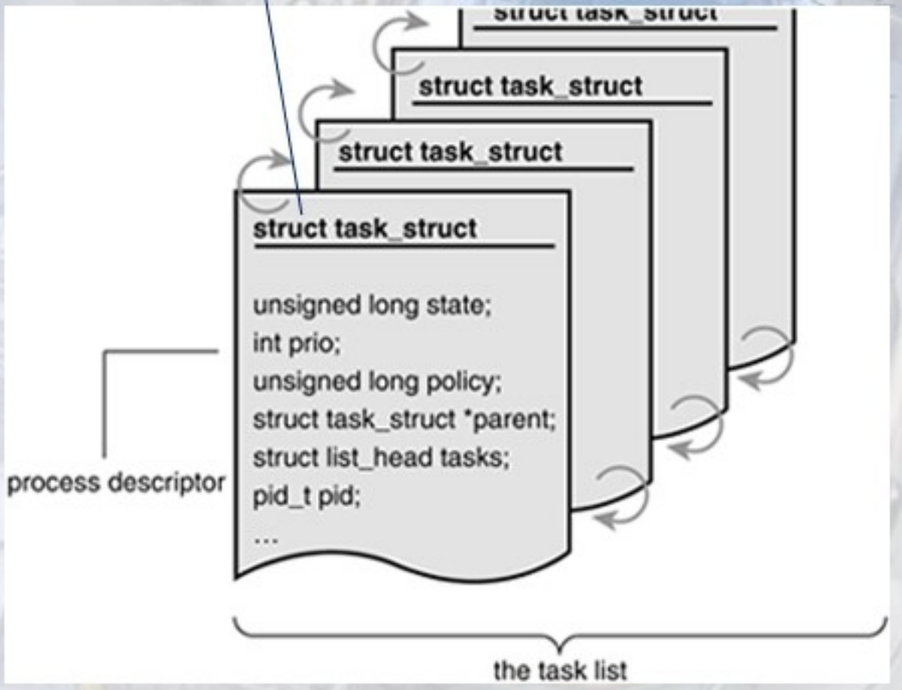
# Kernel modules

- Modules **are pieces of code** that **can be loaded** and **unloaded** into the **kernel** **upon demand**.
- They extend the functionality of the kernel without the need to **reboot** the system.
- Without modules, we would have to build monolithic kernels **and add new functionality directly into the kernel image**.
  - Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the kernel every time we want new functionality
- Modules **can be compiled and dynamically linked** into kernel address space.
- Kernel code that is loaded after the kernel has booted
  - Advantages
    - Load drivers on demand (e.g. for USB devices)
    - Load drivers later – speed up initial boot
  - Disadvantages
    - Adds kernel version dependency to root file system
    - More files to manage

# Process Management in Linux process descriptor

- The kernel stores the list of processes in a circular doubly linked list called the task list
- Each element in the task list is a process descriptor of the type struct task_struct, which is defined in <linux/sched.h>.
- The process descriptor contains all the information about a specific process.
- The process descriptor contains the data that describes the executing program, open files, the process's address space, pending signals, the process's state, and much more
- Task_struct
  - PCB (process control BLOCK) in linux

Process descriptor
- Large data structure (about 1.7 KB)
- Contains the process related data.
- Process ID
- Process state
- Priority
- Open files
- The process's address space
- Pending signals
- Etc.




https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

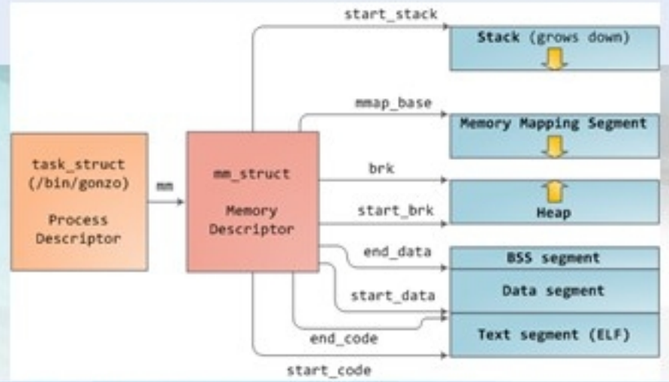
# task_struct data structure, that describes the process and all its details.




#LEARN IN DEPTH

#Be professional in embedded system

| FIELD IN TASK_STRUCT DATA STRUCTURE | DESCRIPTION |
|---|---|
| state | You can easily guess this one when we talk about processes. Process can be in different states. Namely interruptible, uninterruptible, zombie, stopped etc. |
| ptrace | this is related to debugging a process. To trace system calls carried out by a process. |
| static_prio | Nice value for specifying **process priority** |
| cpus_allowed | The programmer can specify the cpu core where he wants the process to run - or allowed to run - this is where cpu affinity is set |
| ptrace_children | Child process beneath that is being traced (actually this is yet again pointing to another struct - another structure) |
| ptrace_list | Other parent processes that is tracing this process (another structure) |
| mm | As we know that task_struct fields can point to other structures (ie: similar to task_struct), this mm field points to another structure called mm_struct. This is the memory description for a particular process. |

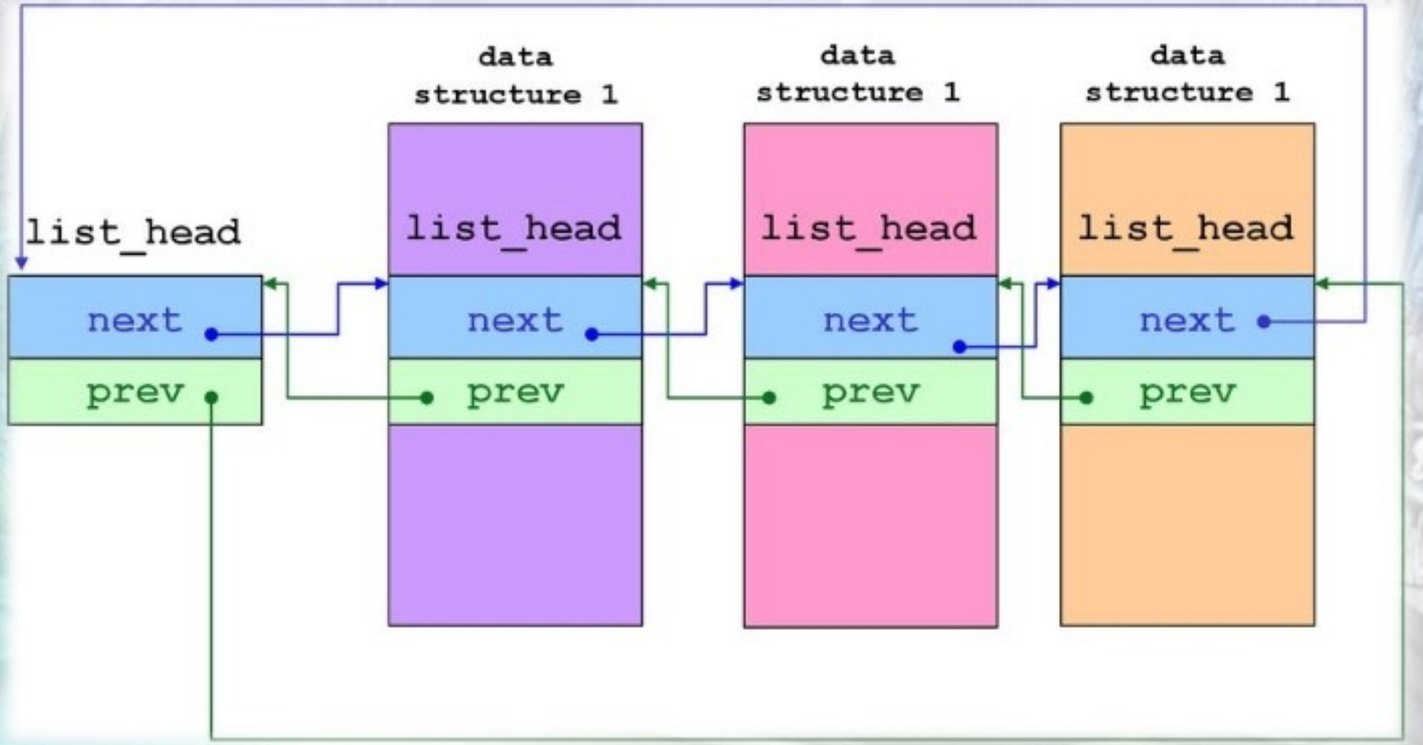



| | |
|---|---|
| exit_code, exit_signal | These fields are used to hold signals and exit codes when the process exits. This will let the parent process know how the child died |
| pdeath_signal | The signal that is sent when the parent dies |
| pid | **process identifier** |
| parent, children | This is again another structure pointing to parent process, and any children this process has |
| utime,stime,cutime,cstime | user time, system time, collective user time spent by process and its children, total system time spent by process and children |
| gid,uid,environment | group id, user id and environment variables available to the process |
| files | open file information - again another structure |
| signals | handlers related to certain signals, signals that are pending etc. |
| tgid | Thread group identifier |



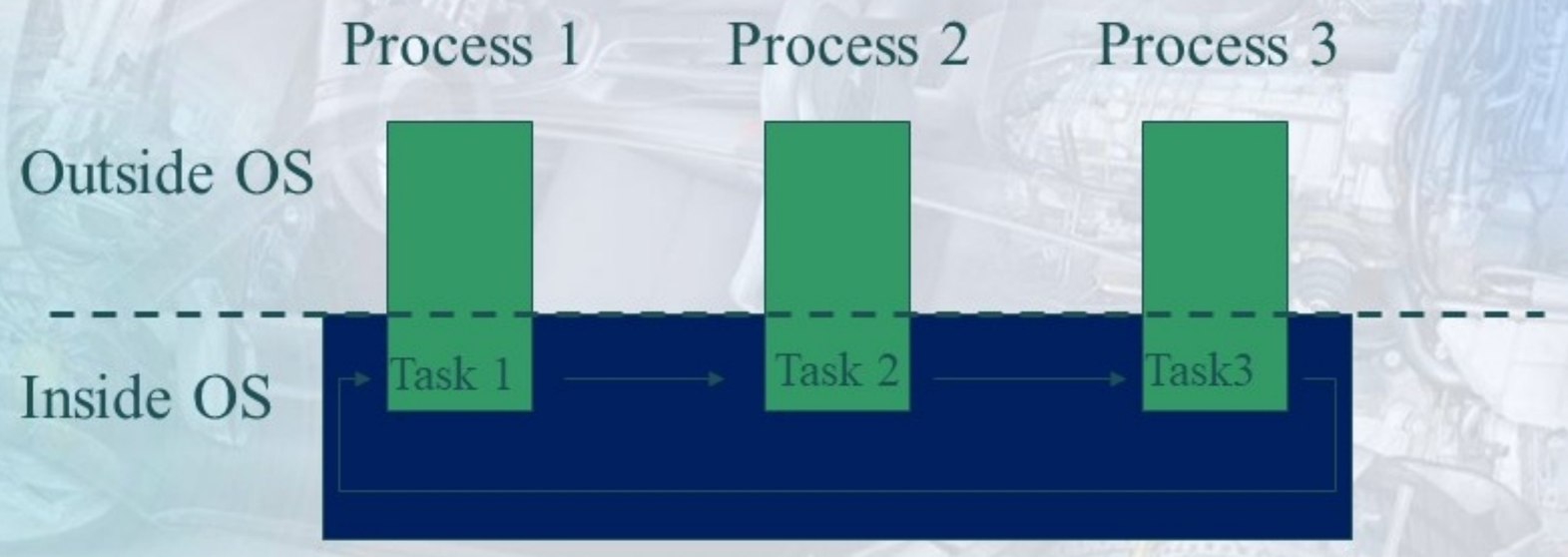
https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

# a circular doubly linked list called the task list for three elements



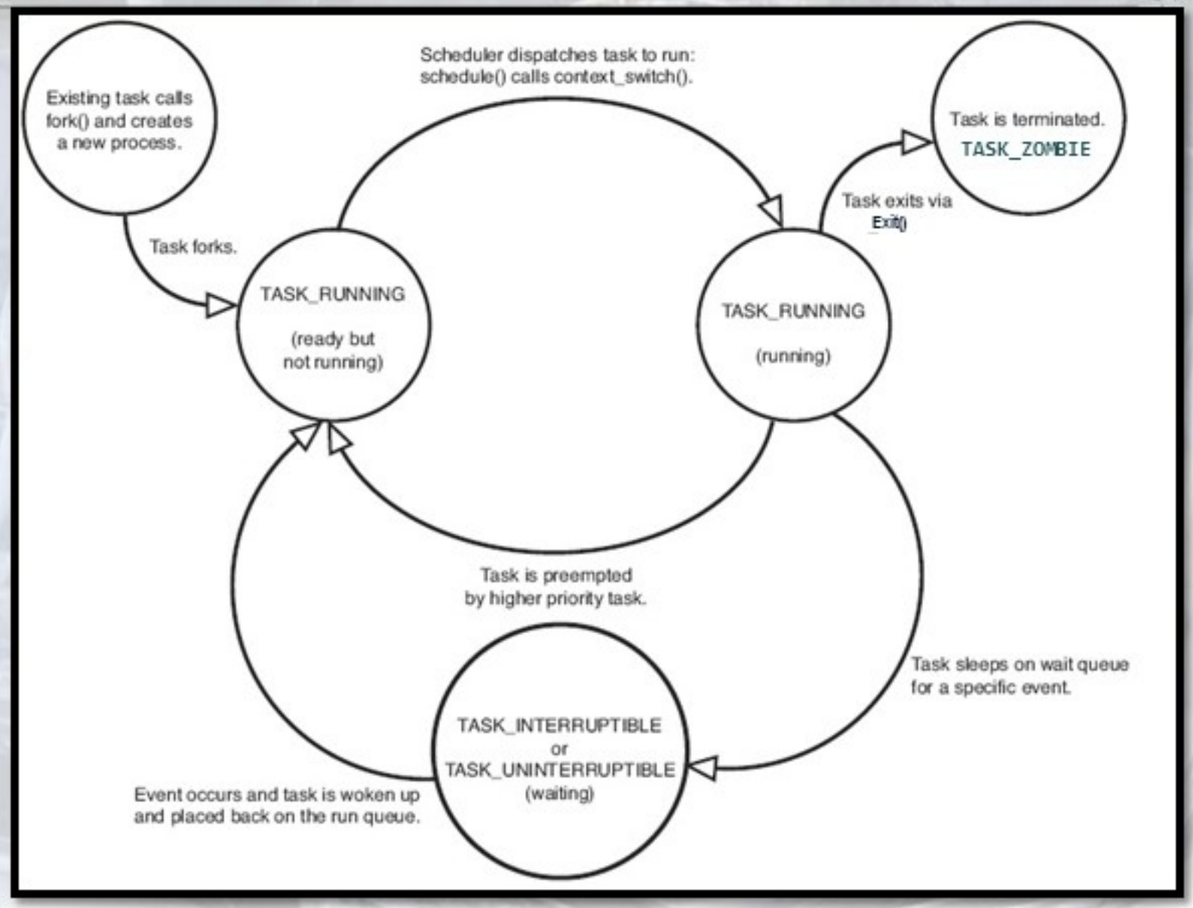
# Processes vs. Tasks

- Linux views all work as a series of tasks imposed on the OS



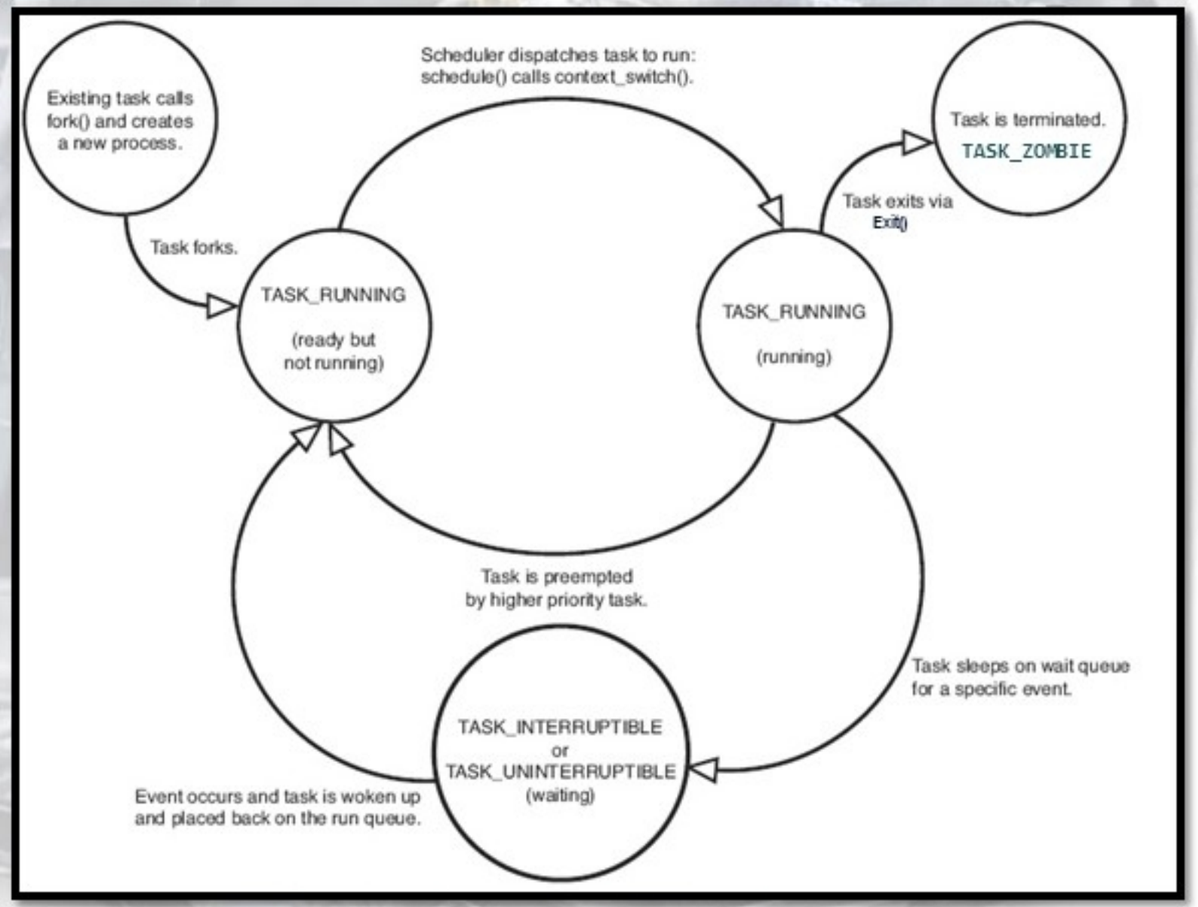
# Process State

- The state field of the process descriptor describes the current condition of the process.
- Each process on the system is in exactly one of five different states. This value is represented by one of five flags:
- TASK_RUNNING
- TASK_INTERRUPTIBLE
- TASK_UNINTERRUPTIBLE
- TASK_STOPPED
- TASK_ZOMBIE



# Process State

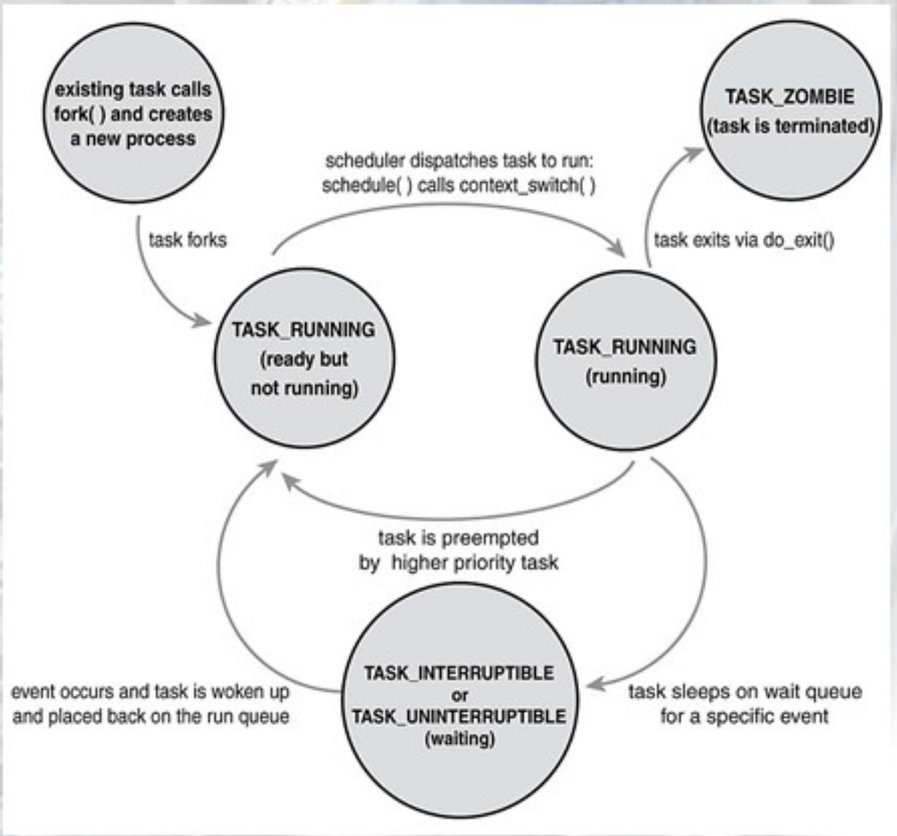
- TASK_RUNNING
  - The process is runnable; it is either currently running or on a runqueue waiting to run
- TASK_INTERRUPTIBLE
  - The process is sleeping (it is blocked), waiting for some condition to exist. When this condition exists, the kernel sets the process's state to TASK_RUNNING.
  - The process becomes runnable if it receives a signal.
- TASK_UNINTERRUPTIBLE
  - This state is identical to TASK_INTERRUPTIBLE except that it does not wake up and become runnable if it receives a signal.
- TASK_STOPPED
  - the process has been stopped by a signal or by a debugger
- TASK_ZOMBIE
  - The task has terminated, The task's process descriptor must remain in case the parent wants to access it.
- TASK_TRACED
  - -the process is being traced via the ptrace system call




https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

# Creating a New Task

- When a parent process invokes the ***fork()*** system call, it creates a new process by performing the following steps:
  - Allocate new process descriptor to hold info needed to manage task (struct task_struct) and link it to the task list
  - Creates new kernel space stack
  - Duplicate info from parent's process descriptor into the child's.
  - Modify the fields in child's process descriptor
    - New PID
    - Link to parents task and siblings
    - Initialize task specific timers (e.g. creation time)
  - Create file table and file descriptor for each open file in the parent's file table.
  - Create new user data segment for child and copy from parent
  - Copy info regarding signals and handlers
  - Copy virtual memory tables
  - Change child state to TASK_RUNNING




https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

# Find process info: /proc/<pid>

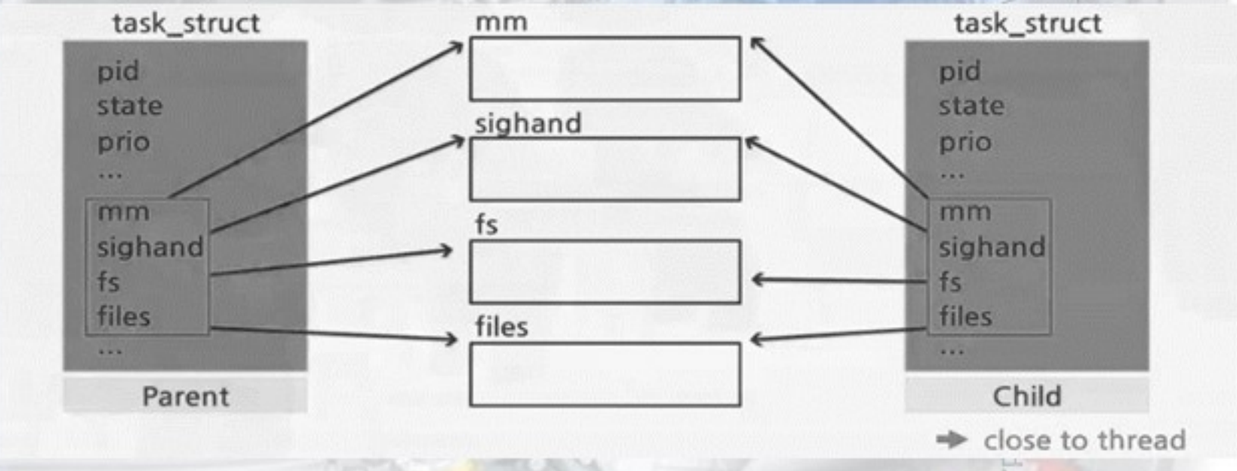
- ps to get process id
- For each process, there is a corresponding directory /proc/<pid> to store this process information in the /proc pseudo file system

# Process-related files

- **Header files**
  - include/linux/sched.h – declarations for most task data structures
  - include/linux/wait.h – declarations for wait queues
  - include/asm-i386/system.h – architecture-dependent declarations
- **Source files**
  - kernel/sched.c – task scheduling routines
  - kernel/signal.c – signal handling routines
  - kernel/fork.c – process/thread creation routines
  - kernel/exit.c – process exit routines
  - fs/exec.c – executing program
  - arch/i386/kernel/entry.S – kernel entry points
  - arch/i386/kernel/process.c – architecture-dependent process routines
- http://lxr.linux.no/

# Linux Threads

- the Linux kernel, there is *no* concept of a thread.
- Linux implements all threads as standard processes.
  - The Linux kernel does not provide any special scheduling semantics or data structures to represent threads.
  - Instead, a thread is merely a process that shares certain resources with other processes
- **Clone()** system call is used for create threads.
- However Linux doesn't distinguish between processes and threads clearly inline as it determines the **level of resource share through flags for argument.**
- **Flag** set f clone determines how much sharing between parent and child.
- Ex. Threads are created like normal tasks, with the exception that the clone() system call is passed flags corresponding to specific resources to be shared:
  - clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
  - The previous code results in behavior identical to a normal fork(), except that **the address space, filesystem resources, file descriptors, and signal handlers** are shared. In other words, the new task and its parent are what are popularly called threads.




https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/



# The Linux Scheduler

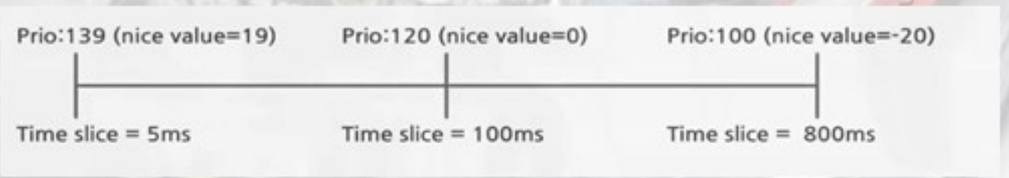
- The scheduler is that part of the kernel code that makes it possible to execute multiple programs at the same time, thus sharing the CPU among a number of processes.
- Deciding which process to run, when and for how long is the scheduler's primary responsibility.
- Linux scheduling is based on the ***time-sharing technique***
  - the CPU time is roughly divided into "slices," one for each runnable process
  - If a currently running **process is not terminated** when its time slice or *quantum expires*, ***the process is preempted.***
  - Time-sharing relies on timer interrupts and is thus transparent to processes.
  - No additional code needs to be inserted in the programs in order to ensure CPU time-sharing.
- The scheduling policy *is also based on* ***ranking processes*** according to their **priority**.
  - Highest priority process always run fast.
  - In Linux, process priority is dynamic.
  - The scheduler keeps track of what processes are doing and adjusts their priorities periodically;
    - Processes that have been denied the use of the CPU for a long time interval are boosted by dynamically increasing their priority.
    - Processes running for a long time are penalized by decreasing their priority.

# Static task prioritization

- All tasks have a static priority, often called a nice value.
- On Linux, nice values range from -20 to 19, with higher values being lower priority
  - The lower a process' nice value, the higher its priority and the larger it's timeslice.
- It divides the ready queue into **multiple queues** according to priority
  - 0~99 : for real time processes
  - 100~139 : User task Prio for non real time processes
- Tasks at the same prio are round-robined.

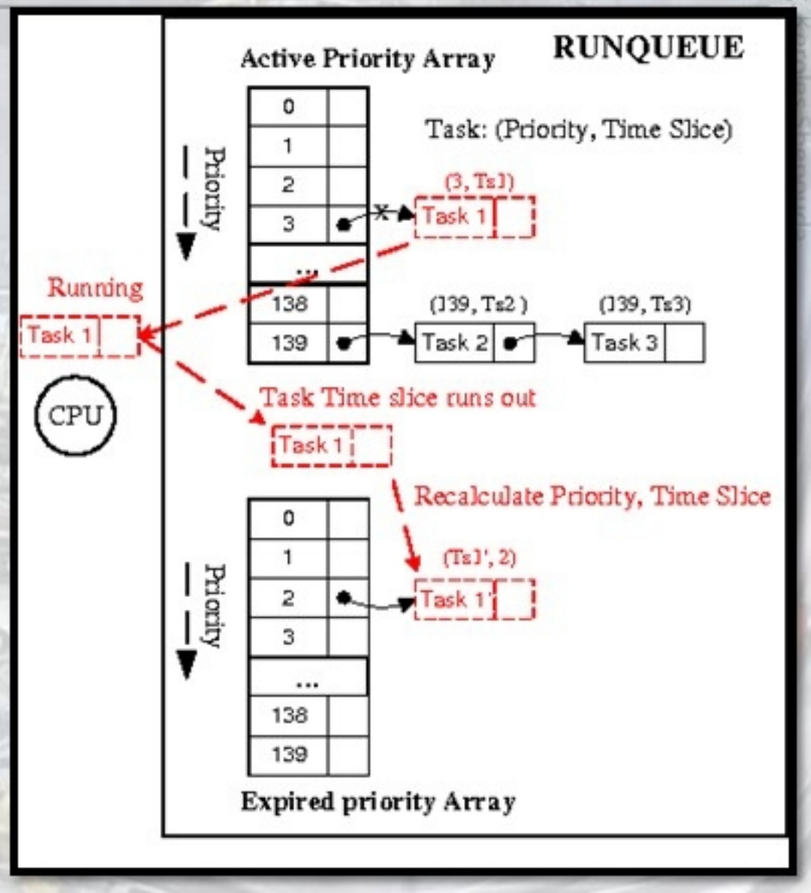
| | Static Priority | NICE | Quantum |
|---|---|---|---|
| High Priority | 100 | -20 | 800 ms |
| | 110 | -10 | 600 ms |
| | 120 | 0 | 100 ms |
| | 130 | +10 | 50 ms |
| Low Priority | 139 | +19 | 5 ms |

$$\text{Quantum} = \begin{cases} (140 - \text{SP}) \times 20 & \text{if SP} < 120 \\ (140 - \text{SP}) \times 5 & \text{if SP} \geq 120 \end{cases}$$



# O(1) scheduler

- Active Array: list of processes with time remaining in their time slices.
- Expired array: list of expired processes
- The scheduler chooses the **process with the highest priority from the active array.**




https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/



# Completely Fair scheduling (CFS) scheduler

- Since kernel 2.6.23, O(1) algorithm is replaced by CFS algorithm
- In O(1) algorithm time slice is assigned based on priority.

However there are some problems in this method.

- For example there are two processes with Priority 120 and 121 respectively.
  - Then time slice 100 m s millisecond and 95 s are assigned respectively based on priority 5
  - the difference in time slice between the two processes and their priority difference is 1.

| process 1 (prio=120) | process 2 (prio=121) | → Time slice = 100ms, 95ms |
|---|---|---|

  - In this case time slices of the two processes are 10 and 5 respectively and the difference in the time slice is 5 .

| process 3 (prio=138) | process 4 (prio=139) | → Time slice = 10ms, 5ms |
|---|---|---|

  - the difference 5 m s is relatively two times of the other.

| | Static Priority | NICE | Quantum |
|---|---|---|---|
| High Priority | 100 | -20 | 800 ms |
| | 110 | -10 | 600 ms |
| | 120 | 0 | 100 ms |
| | 130 | +10 | 50 ms |
| Low Priority | 139 | +19 | 5 ms |

$$\text{Quantum} = \begin{cases} (140 - \text{SP}) \times 20 & \text{if SP} < 120 \\ (140 - \text{SP}) \times 5 & \text{if SP} \geq 120 \end{cases}$$

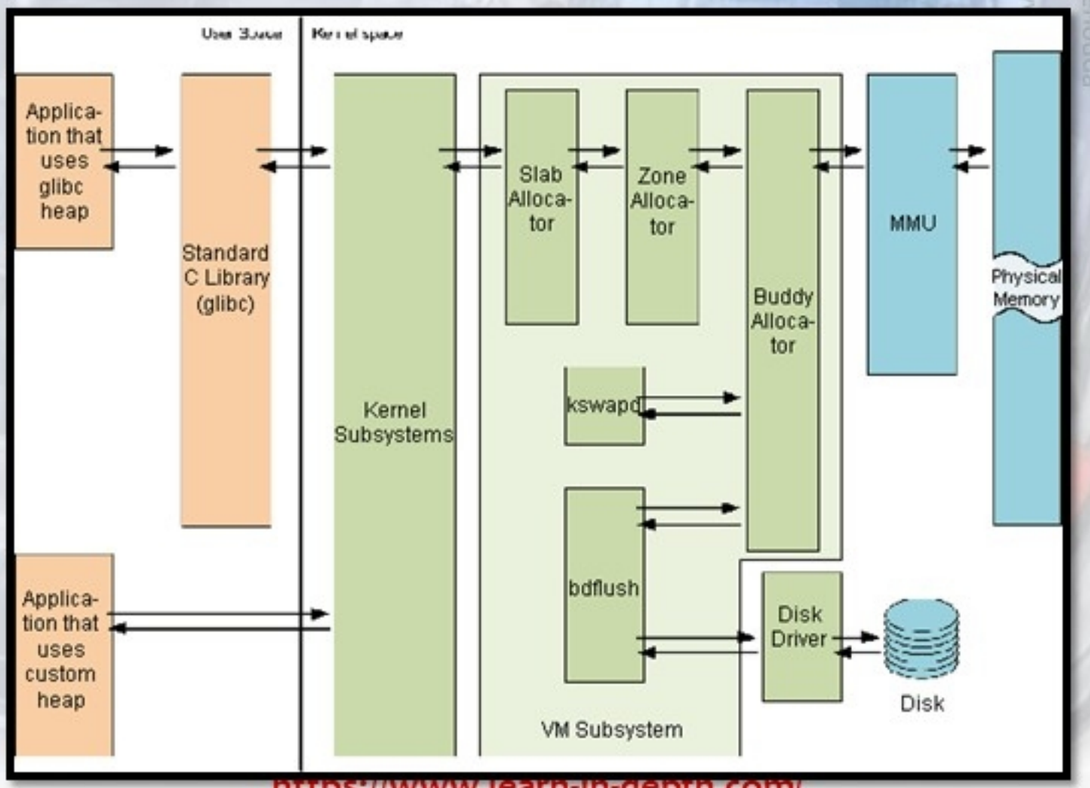


# Completely Fair scheduling (CFS) scheduler

- In CFS:
  - It assigns time slice based on **weight** to solve this problem.
  - it assigns **relative time slice** based on **predefined weight by priority** not predefined time slice based on priority.

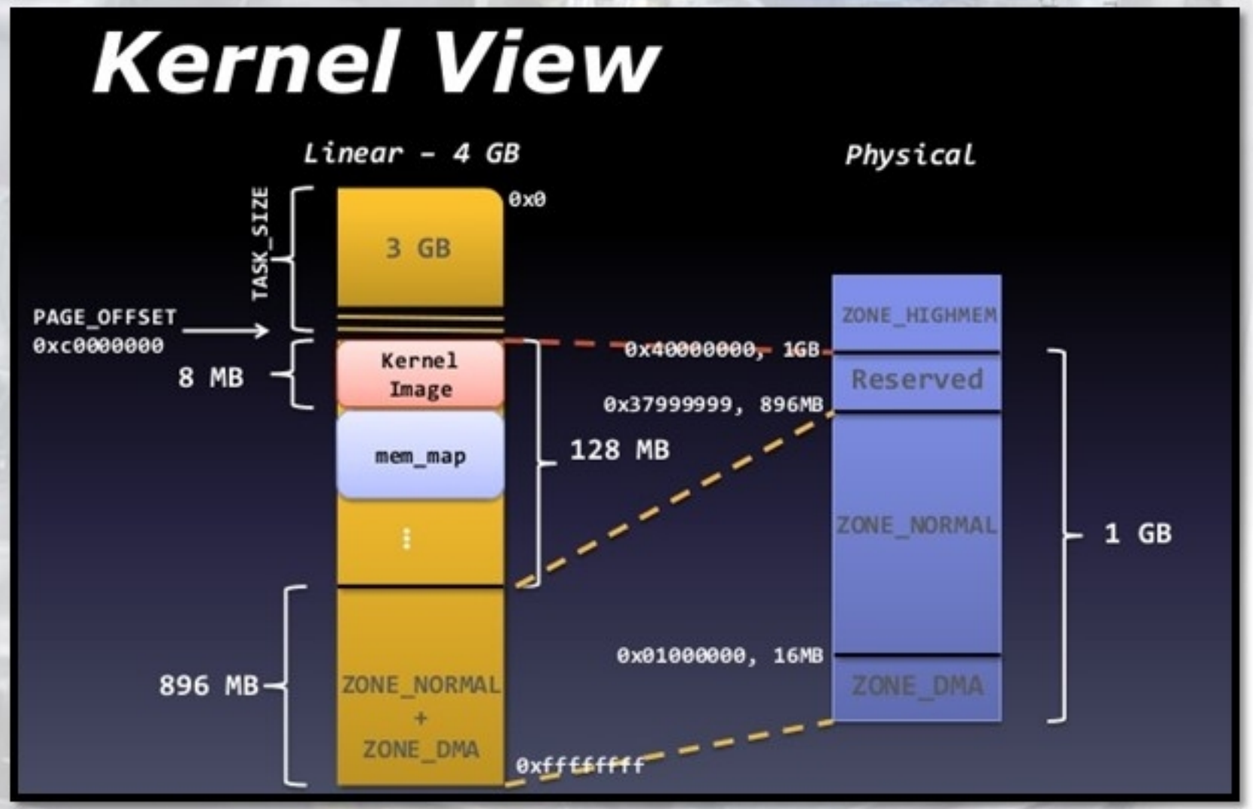
# Linux Memory Management

- In Linux memory management is divided into
  - physical memory management
  - virtual memory management.



# physical memory management in Linux

- Linux kernel treats physical pages "physical frames" as the basic unit of memory management.
  - Every physical page is represented with a struct page.
  - **Buddy system** for allocating /deallocating memory in page unit.
  - **Slab allocator** for allocating/deallocating memory in small blocks.
- ZONE: The kernel divides pages into different Zone
  - ZONE_DMA (<16MB)
    - Contains Pages that are capable of undergoing DMA
    - is used for I O devices.
  - ZONE_Normal(16MB ~ 896MB)
    - Contains normal, regularly mapped, pages.
  - ZONE_HIGHMEM (>896MB)
    - is to prepare for more than one gigabyte.
    - It is an area which maps virtual address space of kernel dynamically.



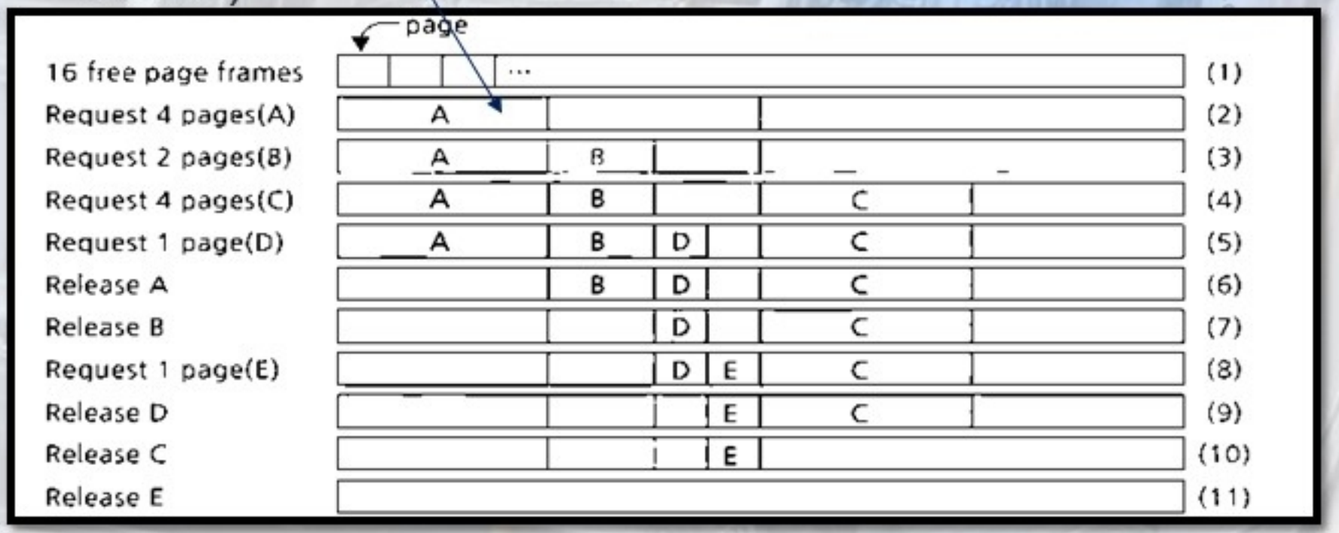

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

# Buddy system

- Linux uses to allocate physical memory as a page unit.
- It allocates ranges of **physically contiguous pages** on **request** to lower management load and external fragmentation
- when two adjacent pages with the same size are freed the two spaces are combined into one.
- If there is no small space available for a memory request it divides one large free region into two to handle the request.

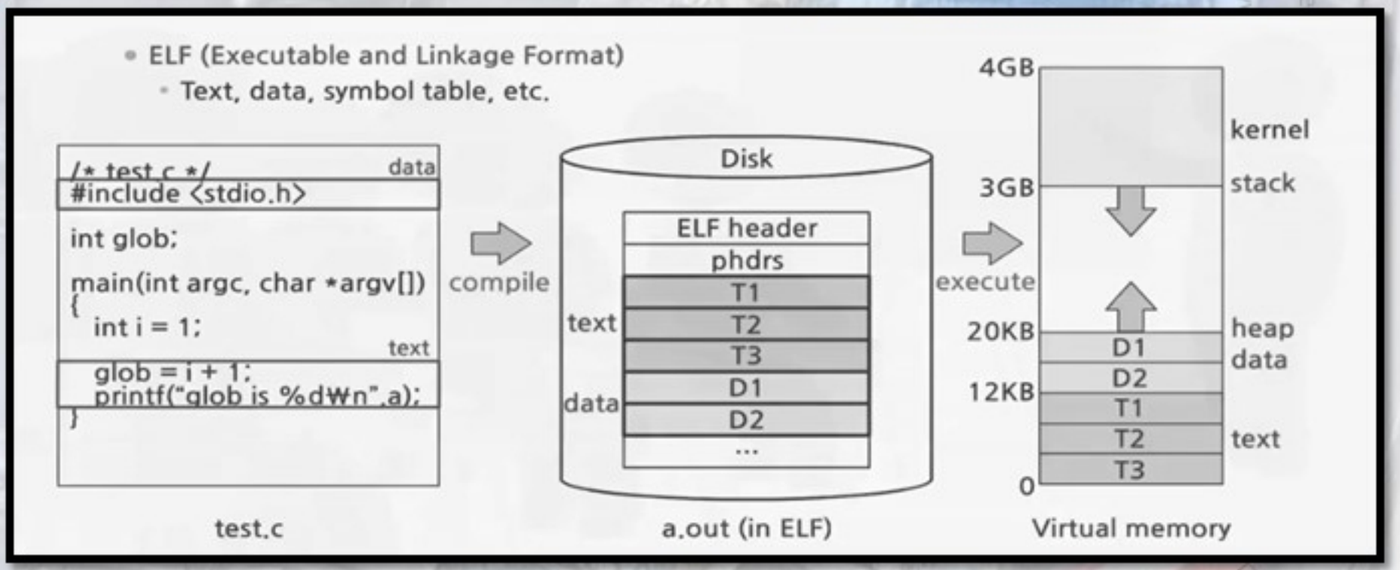
Let us suppose 16 pages that are physically continuous one when it gets a request for 4 pages.

It divides the total 16 pages into two and secures to eight page spaces then it divides one page space to two parts(each part 4 blocks)



# Program Image and Virtual Memory

- In Linux it creates executable file called a.out when it compiles source file the executable file
- built in a format called E L F (Executable and Linkage format) program image as a segment unit such
  - Text, data, symbol table, etc…




https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/



# Elf image

## Dissected file

```
~$uname -p
i686
~$./simple.elf
Hello World!
```

simple.elf

- header 1/2 — Technical details for identification and execution
  - ELF header — identify as an ELF type, specify the architecture
  - Program Header table — Execution information
- sections — Contents of the executable
  - Code — Executable information
  - Data — Information used by the code
  - Sections' names
- header 2/2 — Technical details for linking (ignored for execution)
  - Section Header table — Linking (connecting program objects) information

1

| Fields | Values | Explanation |
|---|---|---|
| e_ident | | |
| EI_MAG | 0x7F, "ELF" | constant signature |
| EI_CLASS, EI_DATA | | 32 bits, Little-Endian |
| EI_VERSION | | Always 1 |
| e_type | | Executable |
| e_machine | | Intel 386 (and later) |
| e_version | | Always 1 |
| e_entry | | Address where execution starts |
| e_phoff | 0x40 | Program Headers' offset |
| e_shoff | 0xC0 | Section Headers' offset |
| e_ehsize | 0x34 | Elf header's size |
| e_phentsize | 0x20 | Size of a single Program Header |
| e_phnum | 1 | Count of Program Headers |
| e_shentsize | 0x28 | Size of a single Section Header |
| e_shnum | 4 | Count of Section Headers |
| e_shstrndx | | Index of the names' section in the table |

2

| | | |
|---|---|---|
| p_type | | The segment should be loaded in memory |
| p_offset | 0 | Offset where it should be read |
| p_vaddr | 0x08000000 | Virtual address where it should be loaded |
| p_paddr | 0x08000000 | Physical address where it should be loaded |
| p_filesz | 0xA0 | Size on file |
| p_memsz | 0xA0 | Size in memory |
| p_flags | | Readable and eXecutable |

3

x86 assembly / Equivalent C code

```
mov ecx, 0x8000090
mov edx, 0xD
mov ebx, 1
mov eax, 4
int 0x80
                    write(STDOUT_FILENO, "Hello World!\n", len("Hello World!\n"));

mov ebx, 1
mov eax, 1
int 0x80
                    exit(1);
```

Strings

```
"Hello World!\n", 0
```

Section names

```
"" .shrtrtab .text .rodata
```

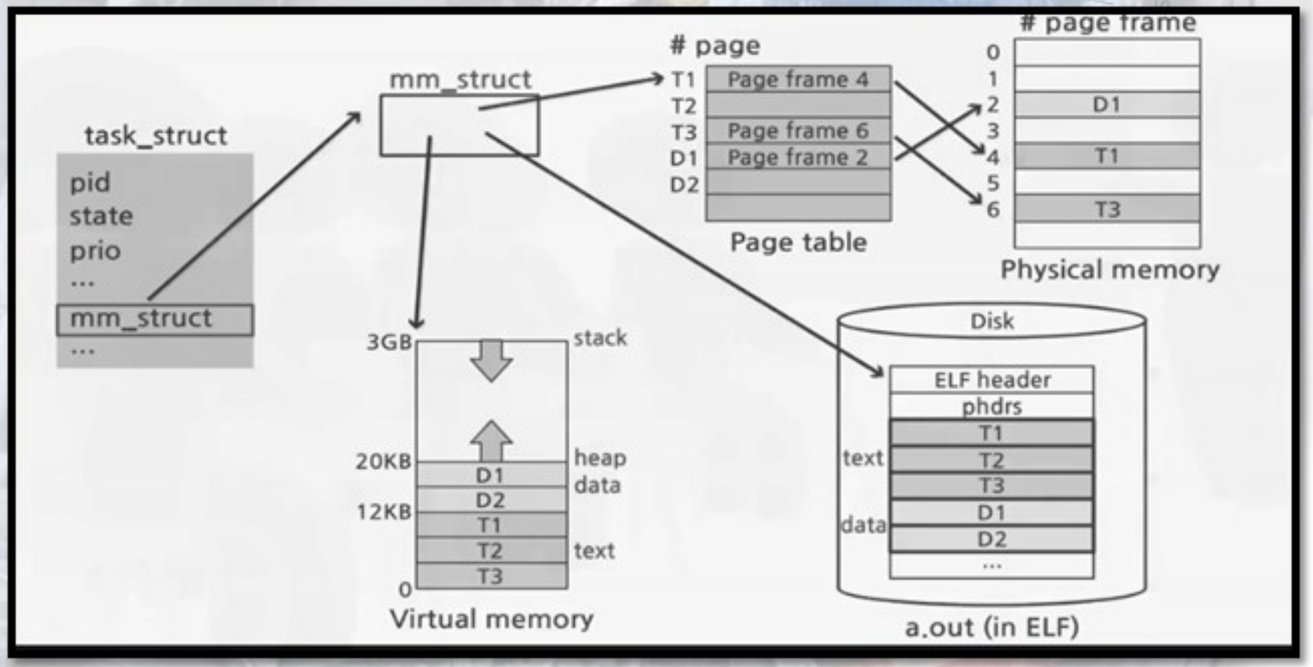
Section Header table

| Index | Name | Type | Flags | Address | offset | Size |
|---|---|---|---|---|---|---|
| 0 | <null> | | | | | |
| 1 | .text | | | 0x8000060 | 0x60 | 0x22 |
| 2 | .rodata | | | 0x8000090 | 0x90 | 0x00 |
| 3 | .shrtrtab | | | | 0xA0 | 0x19 |

This is the whole file, however, most ELF files contain many more elements.
Explanations are simplified, for conciseness.

# mm_struct for virtual memory

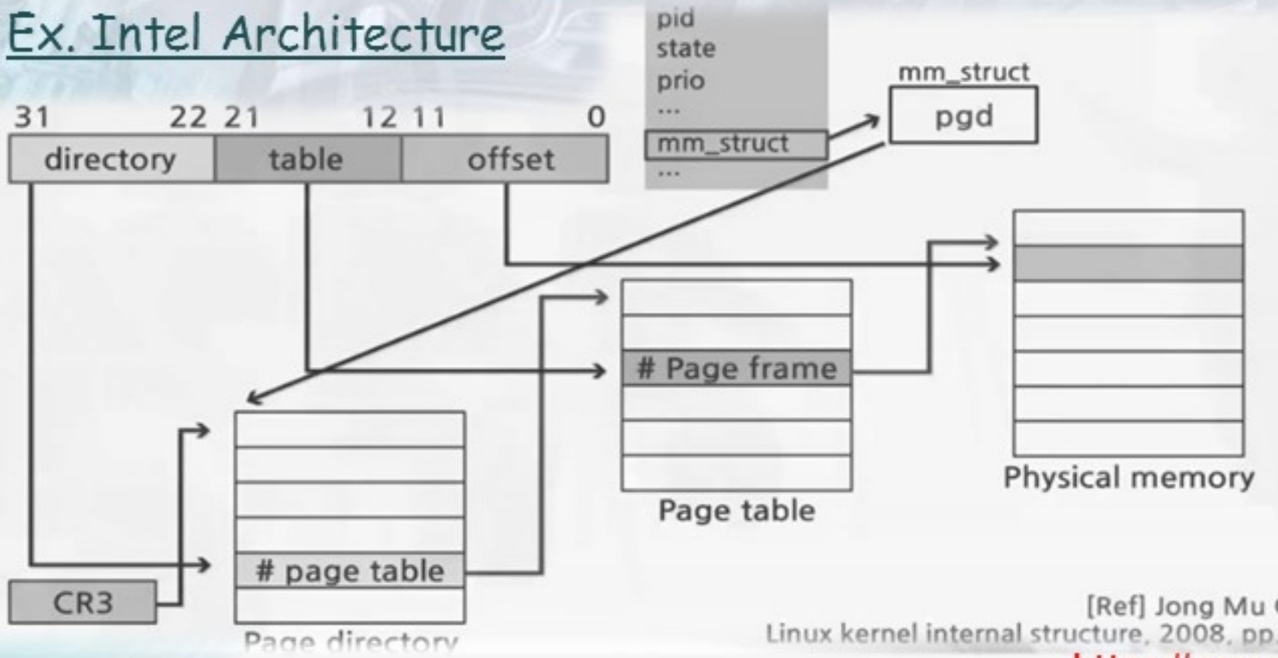
- The **task descriptor** is a data structure to manage each process task descriptor
  - points to ***mm*** descriptor which includes **memory related information of the process and descriptor contains**
    - information about how **virtual memory space** is build and identifies actual location of each segment **on disk**.
    - Also it contains the **location of page table** so **it can identify page frames of physical memory.**




https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

# Address Translation

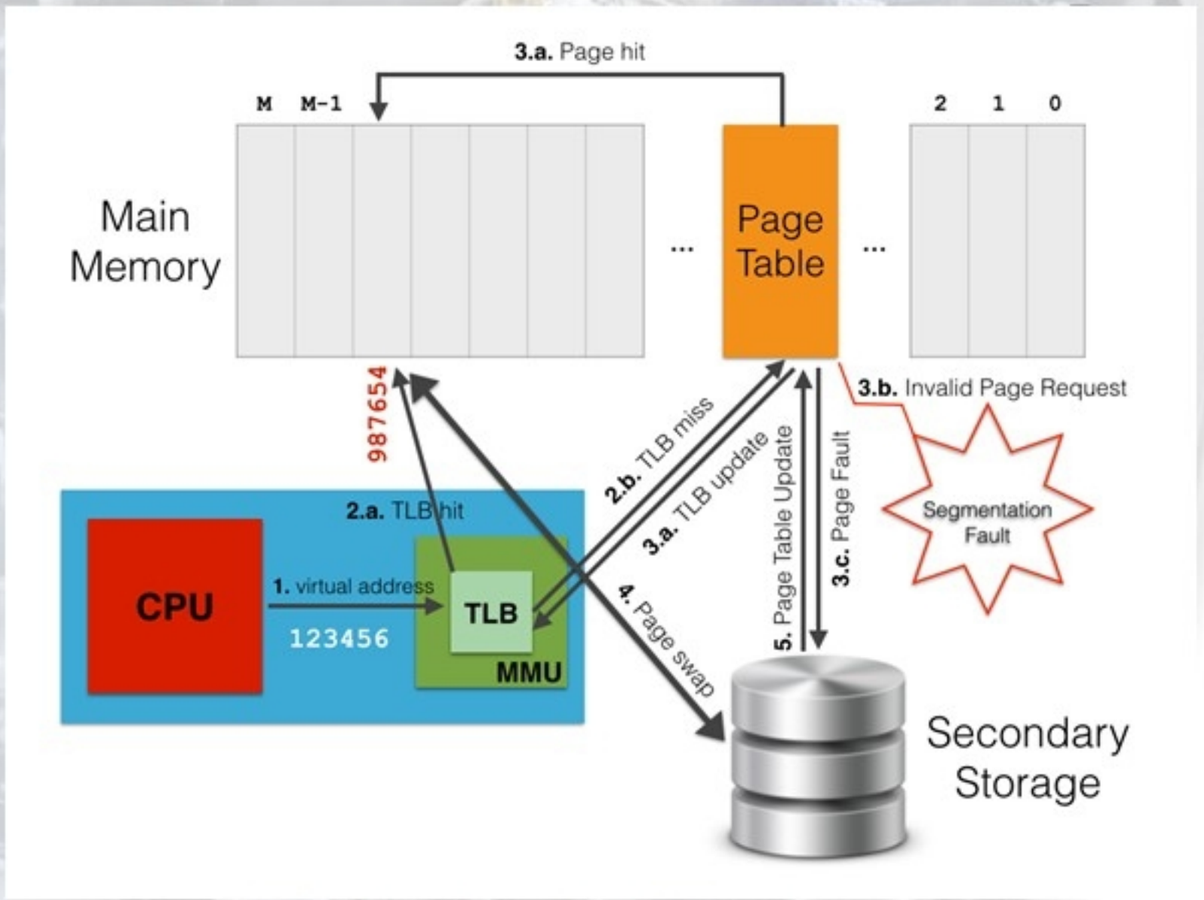
- In Linux it needs to convert virtual address created to physical address to access memory since it also uses paging techniques.
- it supports **multi level page** technique.
  - Ex. Intel Architecture




[Ref] Jong Mu Choi, Linux kernel internal structure, 2008, pp.115

# The modern processor has hardware to assist to translate virtual address to physical address.

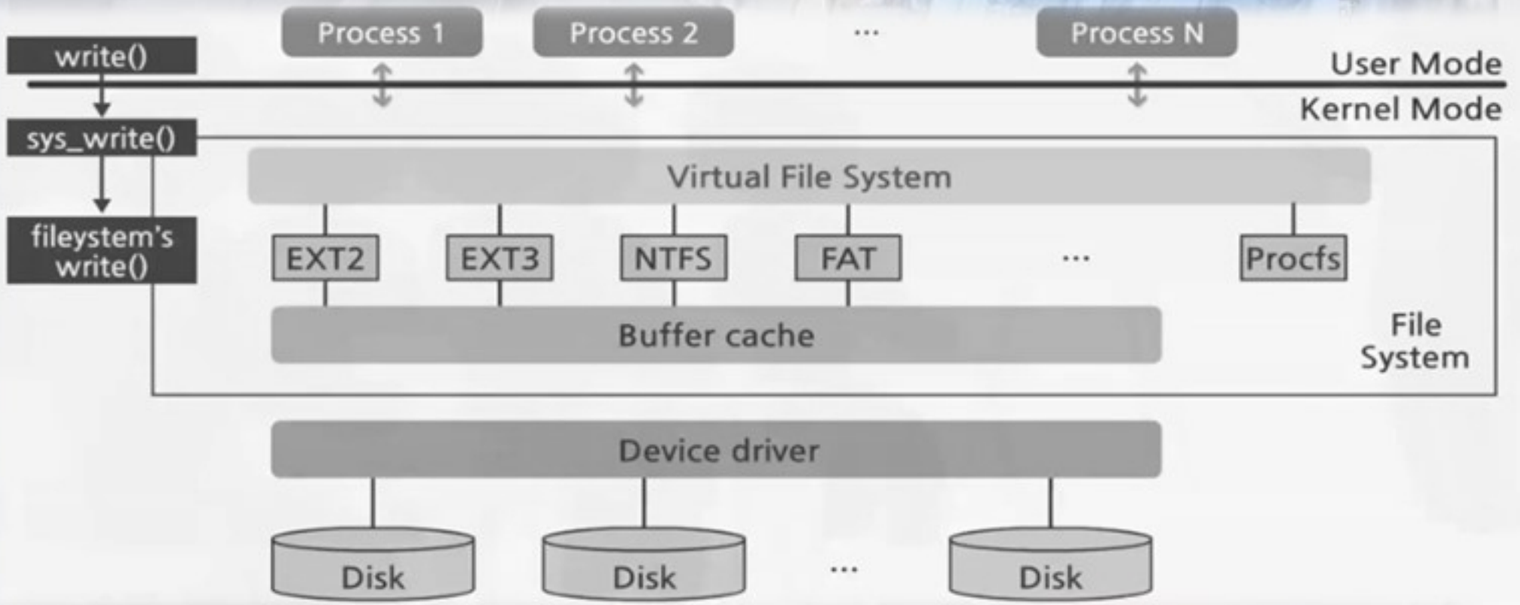
- When a virtual address needs to be translated into a physical address, the MMU first searches for it in the TLB cache (step 1. in the picture above). If a match is found (i.e., TLB hit) then the physical address is returned and the computation simply goes on (2.a.).
- Conversely, if there is no match for the virtual address in the TLB cache (i.e., TLB miss), the MMU searches for a match on the whole page table, i.e., page walk (2.b.). If this match exists on the page table, this is accordingly written to the TLB cache (3.a.). Thus, the address translation is restarted so that the MMU is able find a match on the updated TLB (1 & 2.a.).




https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

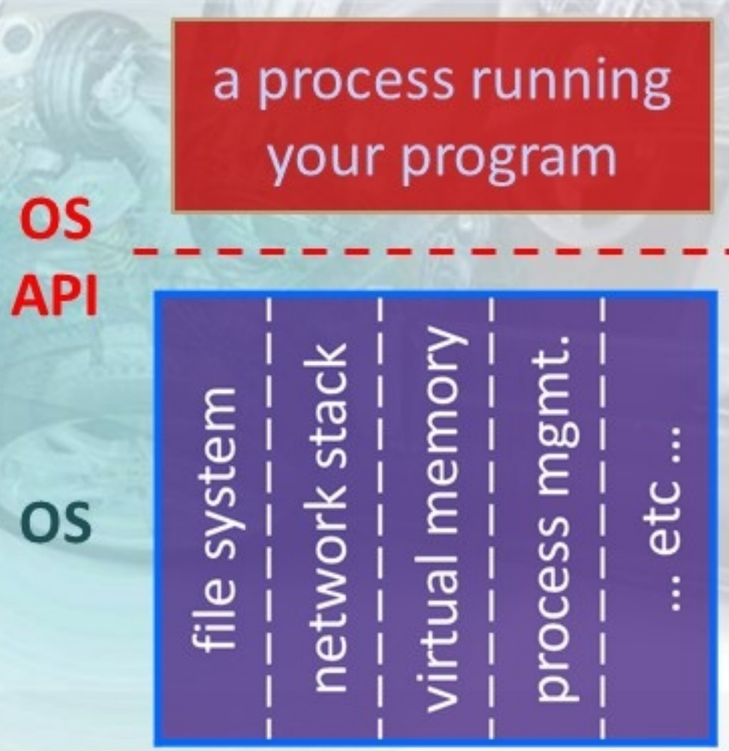
# Linux filesystem structure

- Linux file system provides **virtual file system** to support various file systems
- Kernel internally hides implementation details and manages the multiple different file systems **via an abstraction layer called Virtual File System (VFS)**



# OS: Abstraction Provider

- The OS is the "layer below"
  - A module that your program can call (with system calls)
  - Provides a powerful OS API - POSIX, Windows, etc.




**File System**
- open(), read(), write(), close(), …

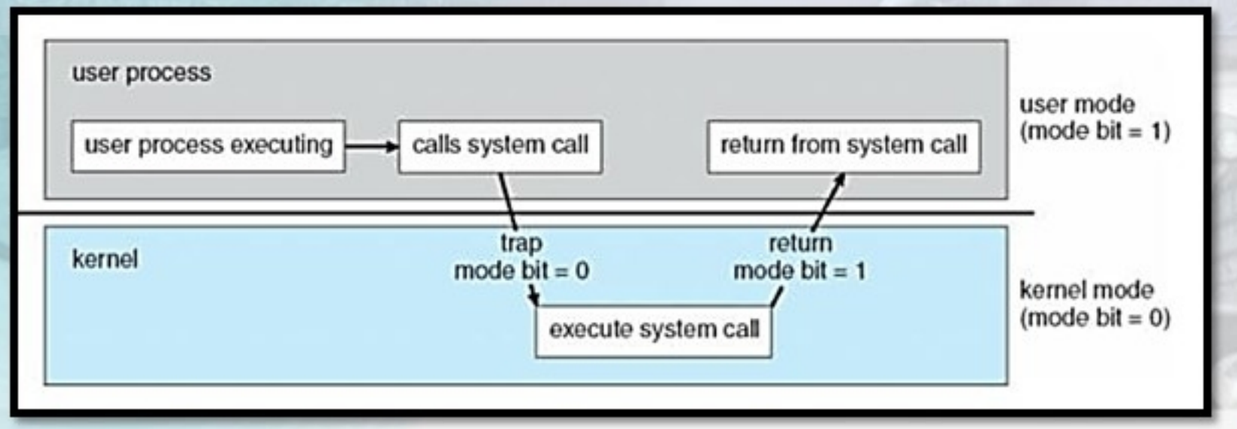
**Network Stack**
- connect(), listen(), read(), write(), …

**Virtual Memory**
- brk(), shm_open(), …

**Process Management**
- fork(), wait(), nice(), …

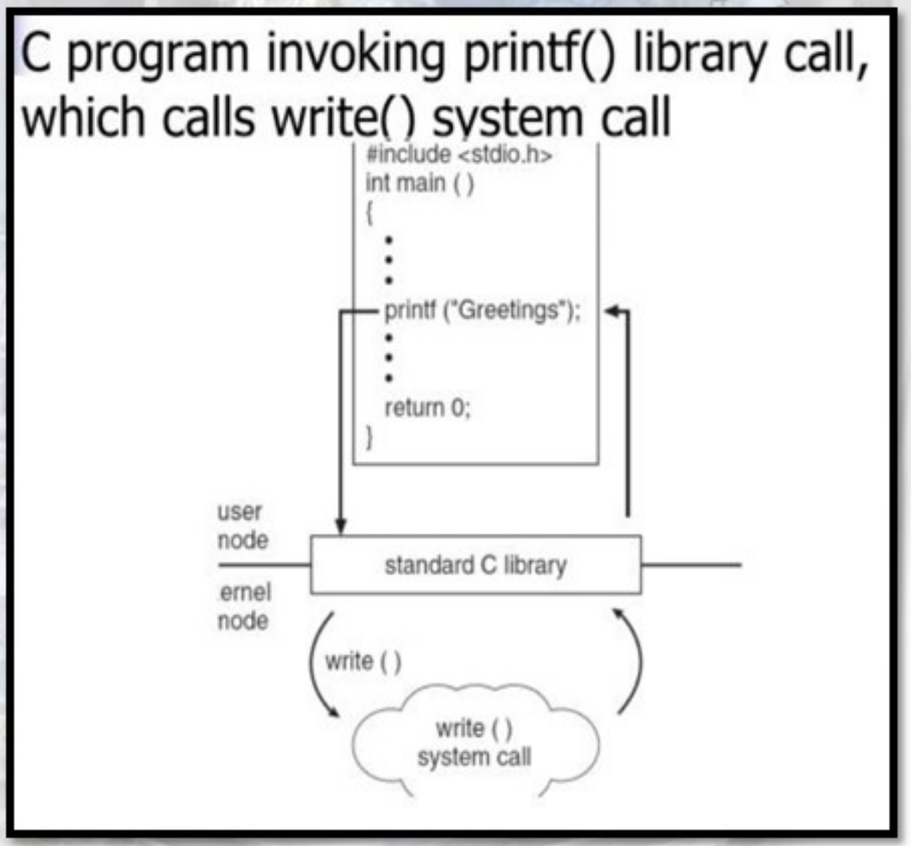
# System Calls

- System programming starts and ends **with system calls.**
- System calls (often shortened to *syscalls*) are function invocations made from user space—your text editor, favorite game, and so on—into the kernel (the core internals of the system) **in order to request some service or resource from the operating system**
- such as read() and write(), to the exotic, such as get_thread_area() and set_tid_address().
- **It is not possible to directly link user-space applications with kernel space**
- the kernel must provide a mechanism by which a user-space application can "**signal**" the kernel that it wishes to invoke a system call.
  - The application can then ***trap*** into the kernel through this well-defined mechanism and execute only code that the kernel allows it to execute. The exact mechanism varies from architecture to architecture.



# The C Library

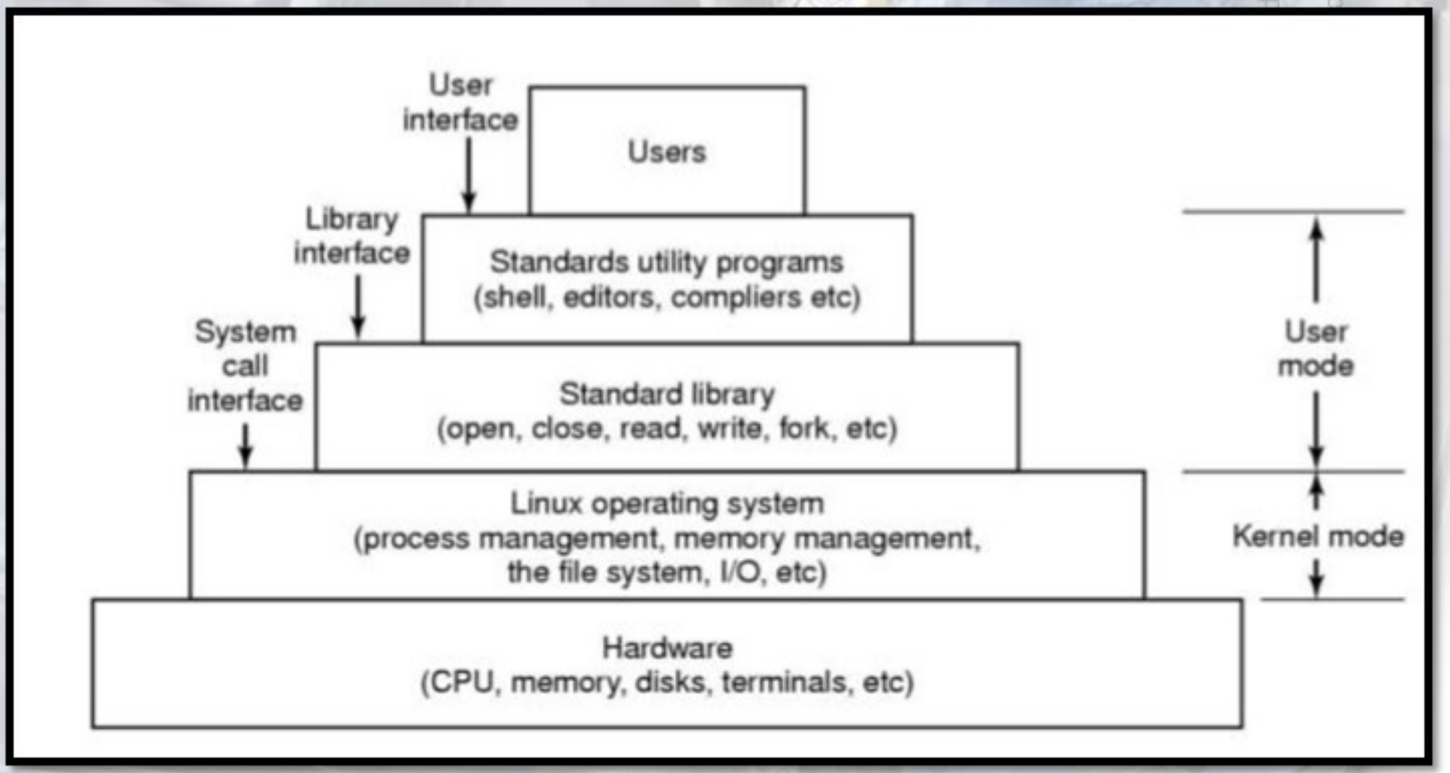
- The C library (*libc*) is at the **heart** of Linux applications. Even when you're programming in another language, the C library is most likely in play, wrapped by the higher-level libraries, **providing core services, and facilitating system call invocation**.
- On modern Linux systems, the C library is provided by *GNU libc*, abbreviated *glibc*, and pronounced
- The GNU C library provides more than its name suggests. In addition to implementing the standard C library, *glibc* provides wrappers for system calls, threading support, and basic application facilities.




https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

# POSIX History

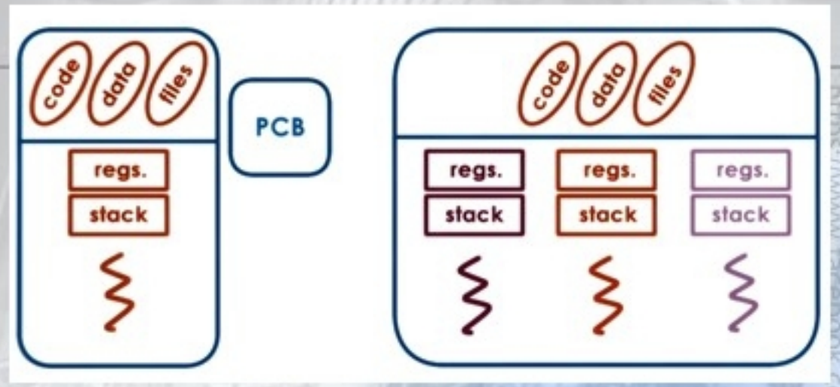
- In the mid-1980s, the Institute of Electrical and Electronics Engineers (IEEE) spearheaded an effort **to standardize system-level interfaces** on Unix systems.
- Richard Stallman, founder of the Free Software movement, suggested the standard be named
  - *POSIX* (pronounced *pahz-icks*), which now stands for *Portable Operating System Interface*.

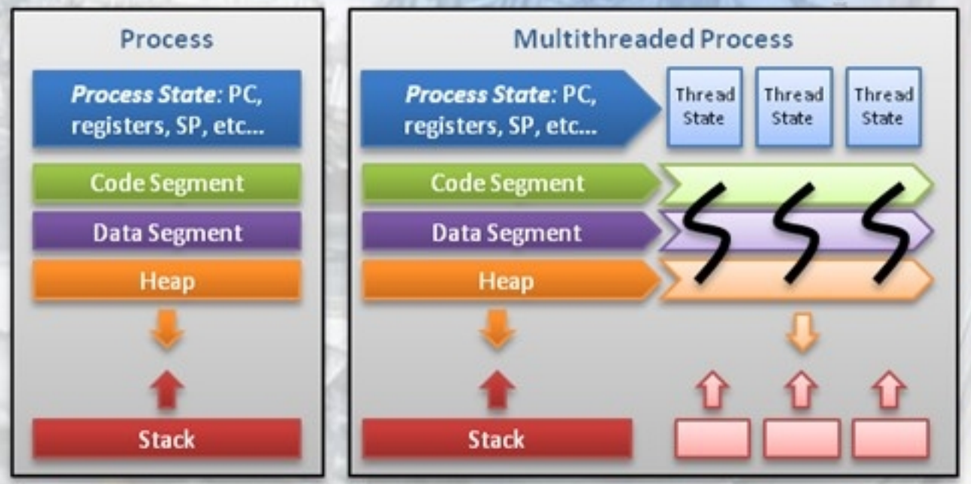




# Processes Vs Threads

- Processes begin life as executable object code, which is machine-runnable code in an executable format
- Each process consists of one or more *threads of execution*
- Most processes consist of only a single thread; they are called ***single-threaded.***
- Processes that contain multiple threads are said to be ***multithreaded***.




Threads contain only necessary information, such as a stack (for local variables, function arguments, return values), a copy of the registers, program counter and any thread-specific data to allow them to be scheduled individually. Other data is shared within the process between all threads.

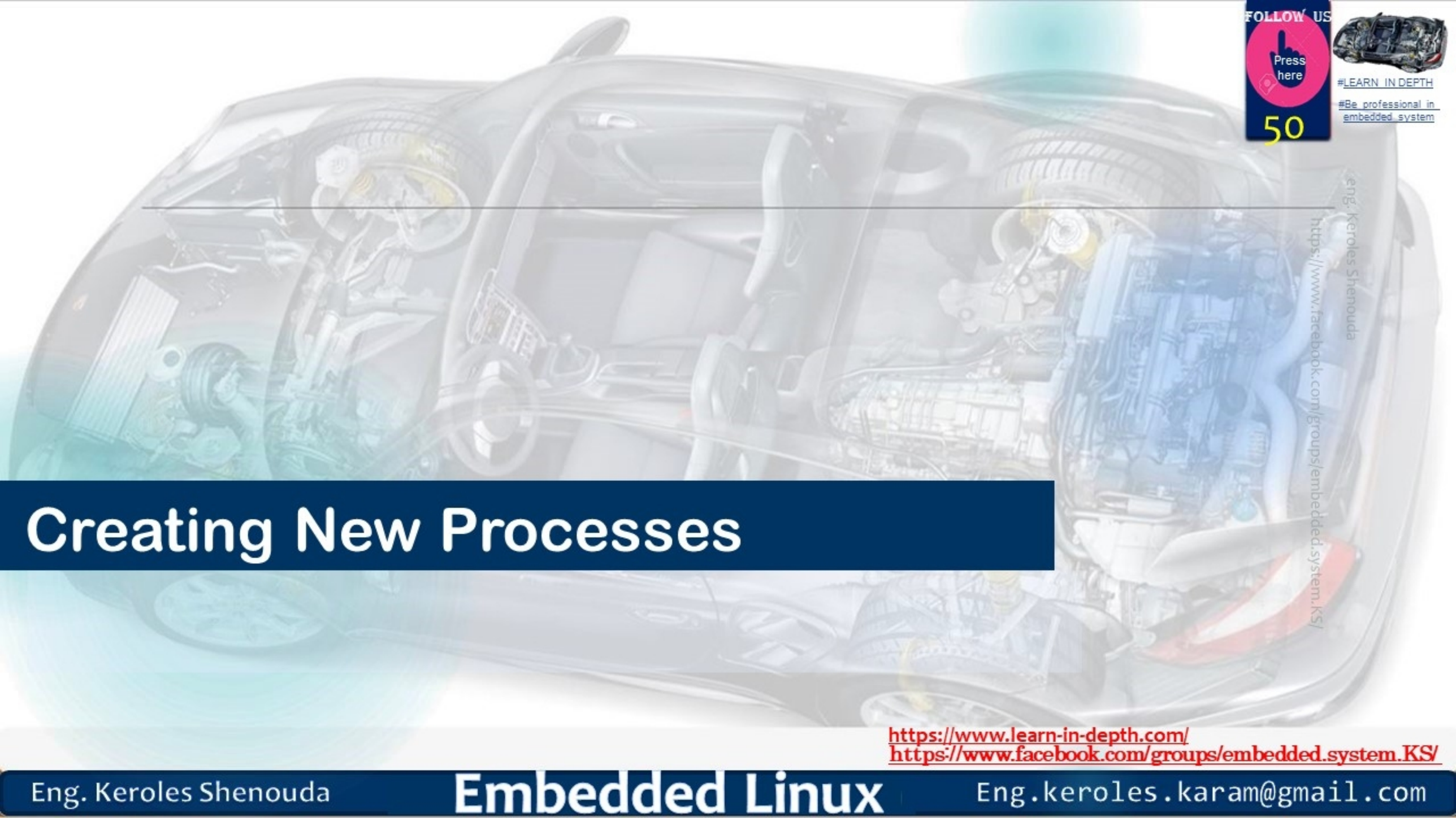


https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/



# Process hierarchy

- Each process is identified by a unique positive integer called the *process ID* (pid)
- The pid of the first process is 1
- The process tree is rooted at the first process, known as the ***init process***
- New processes are created via the fork() system call
- Programmatically, the process ID is represented by the pid_t type, which is defined in the header file <sys/types.h>.




# Creating New Processes

# Creating a New Process

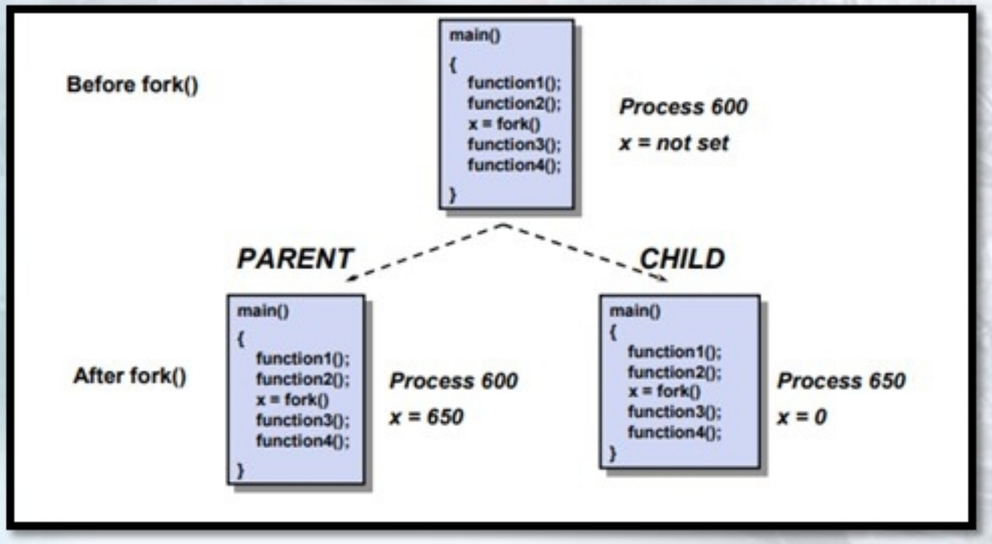
- The fork() system call is used to create a new process.
  - fork() causes the calling process to clone itself, creating a new process that is identical to the original except for pid.
  - The new process is known as the child process;
  - the calling process is known as the parent.
  - fork() takes no parameters.
  - Since the child process is a clone of the parent, it will be executing the same instructions. Therefore, fork() returns twice; once in the child and once in the parent. The return values are:
- fork() returns twice, once in parent process and once in child

```
pid_t fork(void)
```

| return | meaning |
|---|---|
| 0 | return in the child |
| child pid | return in the parent |
| -1 | if an error occurs |

# fork()

- The fork() system call replicates a process in memory. Before fork() is called, only a parent process exists in memory.
- After fork() returns, two almost identical processes exist in memory: the original parent and the newly created child.





# Differences between child and parent process

- Child has its own process id and parent process id
- Alarm signal timers reset to 0 in child

#LEARN IN DEPTH

#Be professional in embedded system

# Example

```
embedded_system_ks@embedded-KS: /media/embedded_system_ks/Embedded_Linux/POSIX/ws/Posix_LABS/
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/POSIX/ws/Posix_
LABS/Debug$ ./Posix_LABS
This is printed from the context of Parent Process  pid=11749
This is printed from the context of child Process pid=0
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/POSIX/ws/Posix_
LABS/Debug$
```

Embedded Linux

ENG.KEROLES SHENOUDA

main.c | dirent.h | dirent.h

```c
/*
 * main.c
 *
 *  Created on: Dec 10, 2019
 *      Author: Keroles Shenouda
 *      www.Learn-in-depth.com
 */
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <dirent.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>


void Fork_lab();

int main (int argc, char *argv[])
{
    Fork_lab();
    return 0 ;
}

void Fork_lab()
{
    pid_t pid;

    //Clone

    if( (pid = fork()) > 0)  //Parent process
    {
        printf("This is printed from the context of Parent Process  pid=%d\n",pid);
    }
    else    //Child Process
    {
        printf("This is printed from the context of child Process pid=%d\n",pid);
    }

}
```



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

# Obtaining the Process ID and Parent Process ID





main.c

```c
/*
 * main.c
 *
 *  Created on: Nov 10, 2019
 *      Author: Keroles Shenouda
 */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int  main(int argc, char **argv)
{
//The getpid() system call returns the process ID of the invoking process:
//  The getppid() system call returns the process ID of the invoking process's parent:

    printf ("My pid=%d\n", getpid ());
    printf ("Parent's pid=%d\n",  getppid ());
    while (1);

}
```

```
embedded_system_ks@embedded-KS:/media/e
first_term/ws/example_1$ gcc main.c ;
embedded_system_ks@embedded-KS:/media/e
first_term/ws/example_1$ ./a.out &
[1] 6039
My pid=6039
Parent's pid=5177
embedded_system_ks@embedded-KS:/media/e
first_term/ws/example_1$ ps
  PID TTY          TIME CMD
 5177 pts/1    00:00:00 bash
 6039 pts/1    00:00:09 a.out
 6042 pts/1    00:00:00 ps
embedded_system_ks@embedded-KS:/media/e
first_term/ws/example_1$
```

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

# Running a New Process

- One system call loads a binary program into memory, replacing the previous contents of the address space, and begins execution of the new program. This is called *executing* a new program, and the functionality is provided
  - by the ***exec*** family of calls.

```
#include <unistd.h>
int execl (const char *path,
const char *arg,
...);
```

```c
/*
 * main.c
 *
 *  Created on: Nov 10, 2019
 *      Author: Keroles Shenouda
 */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int  main(int argc, char **argv)
{

    //  For example, the following code replaces the curren
    int ret;
    ret = execl ("/bin/notjingggggggg", "ls", NULL);
    if ((int)ret <0)
    {
        //print error according to errorno
        perror ("execl");
    }
    while (1);

}
```

```
embedded_system_ks@embedded-KS:/media/er
first_term/ws/example_1$ gcc main.c
embedded_system_ks@embedded-KS:/media/er
first_term/ws/example_1$ ./a.out
execl: No such file or directory
```

# Error Handling

- It goes without saying that checking for and handling errors are of paramount importance.
- In system programming, an error is signified via a function's return value and described via a special variable, **errno**.
- *glibc* transparently provides **errno** support for both library and system calls.
- This variable is declared in <errno.h> as follows:
  - **extern int** errno;

*Errors and their descriptions*

| Preprocessor define | Description |
|---|---|
| E2BIG | Argument list too long |
| EACCES | Permission denied |
| EAGAIN | Try again |
| EBADF | Bad file number |
| EBUSY | Device or resource busy |
| ECHILD | No child processes |
| EDOM | Math argument outside of domain of function |
| EEXIST | File already exists |

| | |
|---|---|
| EFAULT | Bad address |
| EFBIG | File too large |
| EINTR | System call was interrupted |
| EINVAL | Invalid argument |
| EIO | I/O error |
| EISDIR | Is a directory |
| EMFILE | Too many open files |
| EMLINK | Too many links |
| ENFILE | File table overflow |
| ENODEV | No such device |
| ENOENT | No such file or directory |
| ENOEXEC | Exec format error |
| ENOMEM | Out of memory |
| ENOSPC | No space left on device |
| ENOTDIR | Not a directory |
| ENOTTY | Inappropriate I/O control operation |
| ENXIO | No such device or address |
| EPERM | Operation not permitted |
| EPIPE | Broken pipe |
| ERANGE | Result too large |
| EROFS | Read-only filesystem |
| ESPIPE | Invalid seek |
| ESRCH | No such process |
| ETXTBSY | Text file busy |
| EXDEV | Improper link |

# perror()

- The C library provides a handful of functions for translating an errno value to the corresponding textual representation.
- This is **needed only for error reporting**
- The =function is perror():

```
_term/ws/example_1/Debug$ ./example_1
hello
close: Success
```

main.c

```c
/*
 * main.c
 *
 *  Created on: Nov 10, 2019
 *      Author: Keroles Shenouda
 */

#include <stdio.h>
void perror (const char *str);

int main(int argc, char **argv)
{
    printf ("hello \n");
    perror ("close");
    while (1);

}
```



# Running a New Process

- One system call loads a binary program into memory, replacing the previous contents of th address space, and begins execution of the new program. This is called *executing* a new program, and the functionality is provided
  - by the ***exec*** family of calls.

```
#include <unistd.h>
int execl (const char *path,
const char *arg,
...);
```

main.c

```
/*
 * main.c
 *
 *  Created on: Nov 10, 2019
 *      Author: Keroles Shenouda
 */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int  main(int argc, char **argv)
{

    //  For example, the following code replaces the currently executing program with /bin/ls:
    int ret;
    ret = execl ("/bin/ls", "ls", NULL);
    if ((int)ret <0)
    {
        //print error according to errorno
        perror ("execl");
    }
    while (1);

}
```

```
embedded_system_ks@embedded-KS:/media
first_term/ws/example_1$ ./a.out
a.out  Debug  main.c
embedded_system_ks@embedded-KS:/media
first_term/ws/example_1$
```

# if you wanted to ADD option

```c
/*
 * main.c
 *
 *  Created on: Nov 10, 2019
 *      Author: Keroles Shenouda
 */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int  main(int argc, char **argv)
{

    //  For example, the following code replaces the c
    int ret;
    ret = execl ("/bin/ls", "ls", "-la", NULL);
    if ((int)ret <0)
    {
        //print error according to errorno
        perror ("execl");
    }
    while (1);

}
```

```
^C
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/Embedded_Li
first_term/ws/example_1$ gcc main.c
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/Embedded_Li
first_term/ws/example_1$ ./a.out
a.out  Debug  main.c
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/Embedded_Li
first_term/ws/example_1$ gcc main.c
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/Embedded_Li
first_term/ws/example_1$ ./a.out
total 40
drwxr-xr-x 3 embedded_system_ks embedded_system_ks  4096 فون 10 20:59 .
drwxr-xr-x 4 embedded_system_ks embedded_system_ks  4096 فون 10 20:22 ..
-rwxr-xr-x 1 embedded_system_ks embedded_system_ks  7256 فون 10 20:59 a.out
-rw-r--r-- 1 embedded_system_ks embedded_system_ks 10822 فون 10 20:22 .cproject
drwxr-xr-x 2 embedded_system_ks embedded_system_ks  4096 فون 10 20:59 Debug
-rw-r--r-- 1 embedded_system_ks embedded_system_ks   424 فون 10 20:59 main.c
-rw-r--r-- 1 embedded_system_ks embedded_system_ks   761 فون 10 20:22 .project
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/Embedded_Li
first_term/ws/example_1$
```

# Execvp()

- The following snippet uses execvp() to execute *vi*, as we did previously, relying on the fact that *executable* is in the **user's path**:

main.c

```
/*
 * main.c
 *
 *  Created on: Nov 10, 2019
 *      Author: Keroles Shenouda
 */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int  main(int argc, char **argv)
{
    const char *args[] = {  "ls", "-la", NULL };
    //  For example, the following code replaces the currently
    int ret;
    ret = execvp ("ls", args);
    if ((int)ret <0)
    {
        //print error according to errorno
        perror ("execl");
    }
    while (1);

}
```

```
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/Embedded_Linux/
first_term/ws/example_1$ ./a.out
total 40
drwxr-xr-x 3 embedded_system_ks embedded_system_ks  4096 فون 10 21:04 .
drwxr-xr-x 4 embedded_system_ks embedded_system_ks  4096 فون 10 20:22 ..
-rwxr-xr-x 1 embedded_system_ks embedded_system_ks  7256 فون 10 21:04 a.out
-rw-r--r-- 1 embedded_system_ks embedded_system_ks 10822 فون 10 20:22 .cproject
drwxr-xr-x 2 embedded_system_ks embedded_system_ks  4096 فون 10 21:04 Debug
-rw-r--r-- 1 embedded_system_ks embedded_system_ks   452 فون 10 21:04 main.c
-rw-r--r-- 1 embedded_system_ks embedded_system_ks   761 فون 10 20:22 .project
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/Embedded_Linux/
first_term/ws/example_1$
```



# The fork() System Call

## fork(): System call to create a child process.

- A new process running the same image as the current one can be created via the fork()
- system call:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork (void);
```

- A successful call to fork() creates a new process, identical in almost all aspects to the invoking process.
- The new process is called the "child" of the original process, which in turn is called the "parent."
- In the child, a successful invocation of fork() returns 0.
- In the parent, fork() returns the pid of the child



# The fork() System Call

- On error, a child process is not created, fork() returns -1, and errno is set appropriately.
- There are two possible errno values, with three possible meanings
  - EAGAIN :The kernel failed to allocate certain resources, such as a new pid, or the *RLIMIT_NPROC* resource limit (rlimit) has been reached
  - ENOMEM: Insufficient kernel memory was available to complete the request.

#LEARN IN DEPTH

#Be professional in embedded system

# The fork() System Call

main.c

```c
 *
 *  Created on: Nov 10, 2019
 *      Author: Keroles Shenouda
 */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int  main(int argc, char **argv)
{
    pid_t pid;
    pid = fork ();
    unsigned int i =3 ;

    while (i--)
    {
        if (pid > 0) //Parent Process
        {
            printf ("I am the parent of child process with pid=%d!\n", pid);
            printf ("I am the parent! getpid=%d \n", getpid());
            printf ("I am the parent! getppid=%d \n", getppid());
        }
        else if (!pid) //pid =0 Child  Process
        {
            printf ("I am the child! pid=%d \n", pid);
            printf ("I am the child! getpid=%d \n", getpid());
            printf ("I am the child! getppid=%d \n", getppid());


            while (1);//infinite loop for child
        }
        else if (pid < 0)
            perror ("fork");

    }
    while (1);////infinite loop for parent

}
```

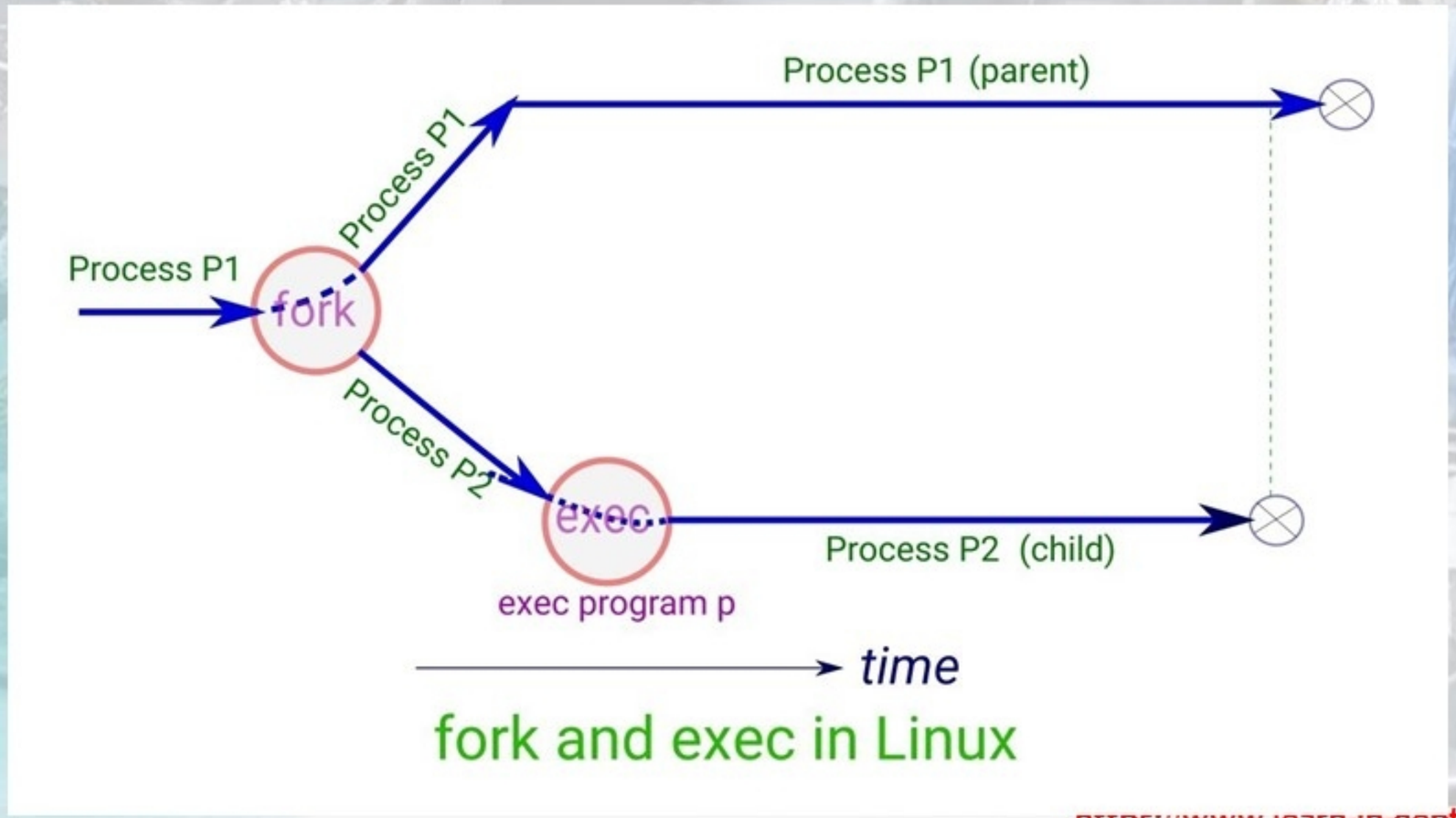
```
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Em
first_term/ws/example_1$ ./a.out &
[1] 7298
I am the parent of child process with pid=7299!
I am the parent! getpid=7298
I am the parent! getppid=5177
I am the parent of child process with pid=7299!
I am the parent! getpid=7298
I am the parent! getppid=5177
I am the parent of child process with pid=7299!
I am the parent! getpid=7298
I am the parent! getppid=5177
I am the child! pid=0
I am the child! getpid=7299
I am the child! getppid=7298
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Em
first_term/ws/example_1$ ps
  PID TTY          TIME CMD
 5177 pts/1    00:00:00 bash
 7298 pts/1    00:00:03 a.out
 7299 pts/1    00:00:03 a.out
 7301 pts/1    00:00:00 ps
```

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

# Note: you have to kill the background process

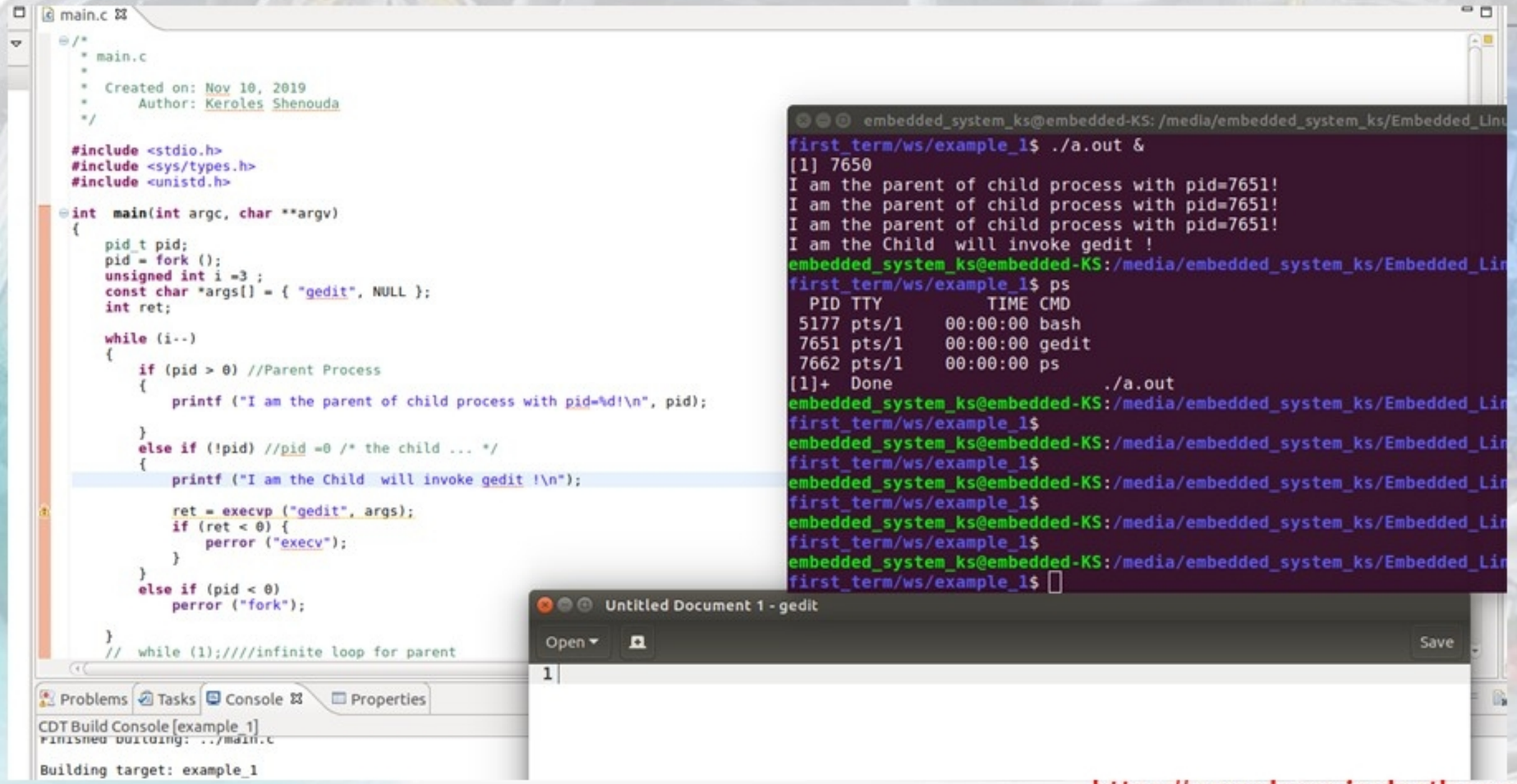
```
first_term/ws/example_1$ ps
  PID TTY          TIME CMD
 5177 pts/1    00:00:00 bash
 7298 pts/1    00:03:04 a.out
 7299 pts/1    00:03:04 a.out
 7376 pts/1    00:00:00 ps
embedded_system_ks@embedded-KS:/media/embedded_system_ks/
first_term/ws/example_1$ kill 7298 7299
```

# Fork and exec




fork and exec in Linux

# execv() inside the Child process




https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

# vfork()

- vfork -> create a child process and **block** parent process.
  - Note:- In vfork, signal handlers are inherited but not shared.

| | fork() | vfork() |
|---|---|---|
| **Address space** | Both the child and parent process will have different address space | Both child and parent process share the same address space |
| **Modification in address space** | Any modification done by the child in its address space is not visible to parent process as both will have separate copies | Any modification by child process is visible to both parent and child as both will have same copies |
| **Execution summary** | Both parent and child executes simultaneously | Parent process will be suspended until child execution is completed. |
| **Outcome of usage** | Behaviour is predictable | Behaviour is not predictable |

#LEARN IN DEPTH

#Be professional in embedded system

# QUIZ: what is the output ?

main.c

```c
 */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int  main(int argc, char **argv)
{
    pid_t pid;
    pid = fork ();
    unsigned int i =1 ;

    while (i--)
    {
        if (pid > 0) //Parent Process
        {
            printf ("I am the parent of child process with pid=%d!\n", pid);
            printf ("I am the parent! getpid=%d \n", getpid());
            printf ("I am the parent! getppid=%d \n", getppid());
        }
        else if (!pid) //pid =0 Child  Process
        {
            printf ("I am the child! pid=%d \n", pid);
            printf ("I am the child! getpid=%d \n", getpid());
            printf ("I am the child! getppid=%d \n", getppid());


//          while (1);//infinite loop for child
        }
        else if (pid < 0)
            perror ("fork");

    }

//  while (1);////infinite loop for parent
    printf ("->>>>>>>>>> the end getpid=%d \n", getpid());

}
////////////////////////////
```



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/


#LEARN IN DEPTH

#Be professional in embedded system

# QUIZ: what is the output ?

main.c

```c
 */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int  main(int argc, char **argv)
{
    pid_t pid;
    pid = fork ();
    unsigned int i =1 ;

    while (i--)
    {
        if (pid > 0) //Parent Process
        {
            printf ("I am the parent of child process with pid=%d!\n", pid);
            printf ("I am the parent! getpid=%d \n", getpid());
            printf ("I am the parent! getppid=%d \n", getppid());
        }
        else if (!pid) //pid =0 Child  Process
        {
            printf ("I am the child! pid=%d \n", pid);
            printf ("I am the child! getpid=%d \n", getpid());
            printf ("I am the child! getppid=%d \n", getppid());


//          while (1);//infinite loop for child
        }
        else if (pid < 0)
            perror ("fork");

    }

//  while (1);////infinite loop for parent
    printf ("->>>>>>>>>> the end getpid=%d \n", getpid());

}
////////////////////////////
```

```
first_term/ws/example_1$ ./a.out
I am the parent of child process with pid=7994!
I am the parent! getpid=7993
I am the parent! getppid=5177
->>>>>>>>>> the end getpid=7993
I am the child! pid=0
I am the child! getpid=7994
I am the child! getppid=7993
->>>>>>>>>> the end getpid=7994
embedded_system_ks@embedded-KS:/media/embedded_sy
first_term/ws/example_1$ ps
  PID TTY          TIME CMD
 5177 pts/1    00:00:00 bash
 7651 pts/1    00:00:01 gedit
 8024 pts/1    00:00:00 ps
embedded_system_ks@embedded-KS:/media/embedded_sy
first_term/ws/example_1$
```



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

#LEARN IN DEPTH

#Be professional in embedded system

# QUIZ: what is the output ?

main.c

```c
 */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int  main(int argc, char **argv)
{
    pid_t pid;
    pid = fork ();
    unsigned int i =1 ;

    while (i--)
    {
        if (pid > 0) //Parent Process
        {
            printf ("I am the parent of child process with pid=%d!\n", pid);
            printf ("I am the parent! getpid=%d \n", getpid());
            printf ("I am the parent! getppid=%d \n", getppid());
        }
        else if (!pid) //pid =0 Child  Process
        {
            printf ("I am the child! pid=%d \n", pid);
            printf ("I am the child! getpid=%d \n", getpid());
            printf ("I am the child! getppid=%d \n", getppid());


            while (1);//infinite loop for child
        }
        else if (pid < 0)
            perror ("fork");

    }

//  while (1);////infinite loop for parent
    printf ("->>>>>>>>>> the end getpid=%d \n", getpid());

}
////////////////////////////
```



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

#LEARN IN DEPTH

#Be professional in embedded system

# QUIZ: what is the output ?

main.c

```c
/*
 * main.c
 *
 *  Created on: Nov 10, 2019
 *      Author: Keroles Shenouda
 */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int  main(int argc, char **argv)
{
    pid_t pid;
    pid = fork ();
    unsigned int i =1 ;

    while (i--)
    {
        if (pid > 0) //Parent Process
        {
            printf ("I am the parent of child process with pid=%d!\n", pid);
            printf ("I am the parent! getpid=%d \n", getpid());
            printf ("I am the parent! getppid=%d \n", getppid());
        }
        else if (!pid) //pid =0 Child  Process
        {
            printf ("I am the child! pid=%d \n", pid);
            printf ("I am the child! getpid=%d \n", getpid());
            printf ("I am the child! getppid=%d \n", getppid());


            while (1);//infinite loop for child
        }
        else if (pid < 0)
            perror ("fork");

    }

//  while (1);////infinite loop for parent
    printf ("->>>>>>>>>> the end getpid=%d \n", getpid());

}
//////////////////////////////
```

```
embedded_system_ks@embedded-KS: /media/embedded_system_ks/
embedded_system_ks@embedded-KS:/media/embedded_system_ks/E
first_term/ws/example_1$ gcc main.c
embedded_system_ks@embedded-KS:/media/embedded_system_ks/E
first_term/ws/example_1$ ./a.out
I am the parent of child process with pid=8071!
I am the parent! getpid=8070
I am the parent! getppid=5177
->>>>>>>>>> the end getpid=8070
I am the child! pid=0
embedded_system_ks@embedded-KS:/media/embedded_system_ks/E
first_term/ws/example_1$ I am the child! getpid=8071
I am the child! getppid=1877
```

```
embedded_system_ks@embedded-KS: /media/embedded_syst
embedded_system_ks@embedded-KS:/media/embedded_syste
first_term/ws/example_1$ ./a.out &
[1] 8107
I am the parent of child process with pid=8108!
I am the parent! getpid=8107
I am the parent! getppid=5177
->>>>>>>>>> the end getpid=8107
I am the child! pid=0
I am the child! getpid=8108
I am the child! getppid=1877
embedded_system_ks@embedded-KS:/media/embedded_syste
first_term/ws/example_1$ ps
  PID TTY          TIME CMD
 5177 pts/1    00:00:00 bash
 8108 pts/1    00:00:02 a.out
 8109 pts/1    00:00:00 ps
[1]+  Done                    ./a.out
```

The Behavior is
Nondeterministic
Why !?
Think in depth

rn-in-depth.com/

https://www.facebook.com/groups/embedded.system.KS/

#LEARN IN DEPTH

#Be professional in embedded system

# QUIZ: what is the output ?

main.c

```c
/*
 * main.c
 *
 *  Created on: Nov 10, 2019
 *      Author: Keroles Shenouda
 */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int  main(int argc, char **argv)
{
    pid_t pid;
    pid = vfork ();
    unsigned int i =1 ;

    while (i--)
    {
        if (pid > 0) //Parent Process
        {
            printf ("I am the parent of child process with pid=%d!\n", pid);
            printf ("I am the parent! getpid=%d \n", getpid());
            printf ("I am the parent! getppid=%d \n", getppid());
        }
        else if (!pid) //pid =0 Child  Process
        {
            printf ("I am the child! pid=%d \n", pid);
            printf ("I am the child! getpid=%d \n", getpid());
            printf ("I am the child! getppid=%d \n", getppid());


            while (1);//infinite loop for child
        }
        else if (pid < 0)
            perror ("fork");

    }

//  while (1);////infinite loop for parent
    printf ("->>>>>>>>>> the end getpid=%d \n", getpid());

}
//////////////////////////////
```



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/



# QUIZ: what is the output ?

main.c

```c
/*
 * main.c
 *
 *  Created on: Nov 10, 2019
 *      Author: Keroles Shenouda
 */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int  main(int argc, char **argv)
{
    pid_t pid;
    pid = vfork ();
    unsigned int i =1 ;

    while (i--)
    {
        if (pid > 0) //Parent Process
        {
            printf ("I am the parent of child process with pid=%d!\n", pid);
            printf ("I am the parent! getpid=%d \n", getpid());
            printf ("I am the parent! getppid=%d \n", getppid());
        }
        else if (!pid) //pid =0 Child  Process
        {
            printf ("I am the child! pid=%d \n", pid);
            printf ("I am the child! getpid=%d \n", getpid());
            printf ("I am the child! getppid=%d \n", getppid());


            while (1);//infinite loop for child
        }
        else if (pid < 0)
            perror ("fork");

    }

//  while (1);////infinite loop for parent
    printf ("->>>>>>>>> the end getpid=%d \n", getpid());

}
//////////////////////////////
```

```
embedded_system_ks@embedded-KS: /media/embedd
embedded_system_ks@embedded-KS:/media/embedded_
first_term/ws/example_1$ gcc main.c
embedded_system_ks@embedded-KS:/media/embedded_
first_term/ws/example_1$ ./a.out
I am the child! pid=0
I am the child! getpid=8193
I am the child! getppid=8192
```

vfork –> create a child process and block parent process.



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

# QUIZ: what is the output ?

main.c

```c
/*
 * main.c
 *
 *  Created on: Nov 10, 2019
 *      Author: Keroles Shenouda
 */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int  main(int argc, char **argv)
{
    pid_t pid;
    pid = fork ();
    unsigned int i =1 ;
    unsigned int g =7 ;


    while (i--)
    {
        if (pid > 0) //Parent Process
        {
            printf ("I am the parent of child process with pid=%d!\n", pid);
            printf ("I am the parent! getpid=%d \n", getpid());
            printf ("I am the parent! getppid=%d \n", getppid());
            printf ("I am the parent! --g=%d \n", --g);

        }
        else if (!pid) //pid =0 Child  Process
        {
            printf ("I am the child! pid=%d \n", pid);
            printf ("I am the child! getpid=%d \n", getpid());
            printf ("I am the child! getppid=%d \n", getppid());
            printf ("I am the child! --g=%d \n", --g);



//          while (1);//infinite loop for child
        }
        else if (pid < 0)
            perror ("fork");

    }

//  while (1);////infinite loop for parent
    printf ("->>>>>>>>> the end getpid=%d \n", getpid());
    printf ("---->>>>>>>>>>>>>>>>! --g=%d \n", --g);
```



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

# QUIZ: what is the output ?


#LEARN IN DEPTH

#Be professional in embedded system

main.c

```c
/*
 * main.c
 *
 *  Created on: Nov 10, 2019
 *      Author: Keroles Shenouda
 */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int  main(int argc, char **argv)
{
    pid_t pid;
    pid = fork ();
    unsigned int i =1 ;
    unsigned int g =7 ;


    while (i--)
    {
        if (pid > 0) //Parent Process
        {
            printf ("I am the parent of child process with pid=%d!\n", pid);
            printf ("I am the parent! getpid=%d \n", getpid());
            printf ("I am the parent! getppid=%d \n", getppid());
            printf ("I am the parent! --g=%d \n", --g);

        }
        else if (!pid) //pid =0 Child  Process
        {
            printf ("I am the child! pid=%d \n", pid);
            printf ("I am the child! getpid=%d \n", getpid());
            printf ("I am the child! getppid=%d \n", getppid());
            printf ("I am the child! --g=%d \n", --g);



//          while (1);//infinite loop for child
        }
        else if (pid < 0)
            perror ("fork");

    }

//  while (1);////infinite loop for parent
    printf ("->>>>>>>>>> the end getpid=%d \n", getpid());
    printf ("---->>>>>>>>>>>>>>>>>! --g=%d \n", --g);
```

```
embedded_system_ks@embedded-KS: /media/embedded_sys
embedded_system_ks@embedded-KS:/media/embedded_syste
first_term/ws/example_1$ ./a.out
I am the parent of child process with pid=8345!
I am the parent! getpid=8344
I am the parent! getppid=5177
I am the parent! --g=6
->>>>>>>>>> the end getpid=8344
---->>>>>>>>>>>>>>>>>! --g=5
I am the child! pid=0
I am the child! getpid=8345
I am the child! getppid=1877
I am the child! --g=6
->>>>>>>>>> the end getpid=8345
---->>>>>>>>>>>>>>>>>! --g=5
embedded_system_ks@embedded-KS:/media/embedded_syste
first_term/ws/example_1$
```

The global variable is copied to the CHILD but it is not shared

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

#LEARN IN DEPTH

#Be professional in embedded system

# QUIZ: what is the output ?

main.c

```c
/*
 * main.c
 *
 *  Created on: Nov 10, 2019
 *      Author: Keroles Shenouda
 */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
    int n =10;
    pid_t pid = fork(); //creating the child process
    if (pid == 0)          //if this is a child process
    {
        printf("Child process started\n");
        n=20;
    }
    else//parent process execution
    {
        printf("Now i am coming back to parent process\n");
    }
    printf("value of n: %d \n",n); //sample printing to check "n" value
    return 0;
}
////////////////////////////
```



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

# QUIZ: what is the output ?

main.c

```c
/*
 * main.c
 *
 *  Created on: Nov 10, 2019
 *      Author: Keroles Shenouda
 */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
    int n =10;
    pid_t pid = fork(); //creating the child process
    if (pid == 0)          //if this is a child process
    {
        printf("Child process started\n");
        n=20;
    }
    else//parent process execution
    {
        printf("Now i am coming back to parent process\n");
    }
    printf("value of n: %d \n",n); //sample printing to check "n" value
    return 0;
}
////////////////////////////
```

```
embedded_system_ks@embedded-KS:/media/embedded
first_term/ws/example_1$ gcc main.c
embedded_system_ks@embedded-KS:/media/embedded
first_term/ws/example_1$ ./a.out
Now i am coming back to parent process
value of n: 10
Child process started
value of n: 20
embedded_system_ks@embedded-KS:/media/embedded
first_term/ws/example_1$
```

#LEARN IN DEPTH

#Be professional in embedded system

# QUIZ: what is the output ?

main.c

```c
/*
 * main.c
 *
 *  Created on: Nov 10, 2019
 *      Author: Keroles Shenouda
 */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
    int n =10;
    pid_t pid = vfork(); //creating the child process
    if (pid == 0)           //if this is a child process
    {
        printf("Child process started\n");
        n=20;
    }
    else//parent process execution
    {
        printf("Now i am coming back to parent process\n");
    }
    printf("value of n: %d \n",n); //sample printing to check "n" value
    return 0;
}
////////////////////////////
```



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/


# QUIZ: what is the output ?

main.c

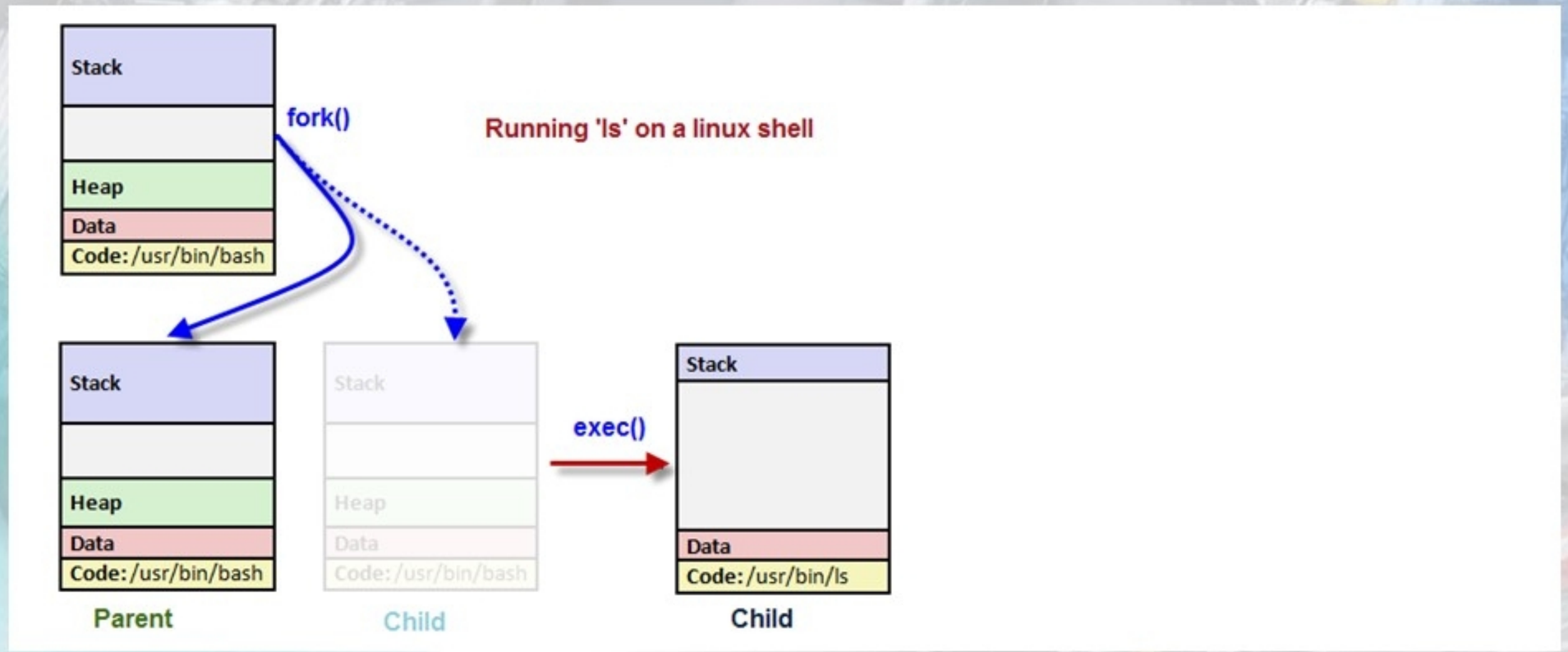
```c
/*
 * main.c
 *
 *  Created on: Nov 10, 2019
 *      Author: Keroles Shenouda
 */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
    int n =10;
    pid_t pid = vfork(); //creating the child process
    if (pid == 0)          //if this is a child process
    {
        printf("Child process started\n");
        n=20;
    }
    else//parent process execution
    {
        printf("Now i am coming back to parent process\n");
    }
    printf("value of n: %d \n",n); //sample printing to check "n" value
    return 0;
}
////////////////////////////
```

```
embedded_system_ks@embedded-KS: /media/embedded_s
embedded_system_ks@embedded-KS:/media/embedded_sys
first_term/ws/example_1$ gcc main.c
embedded_system_ks@embedded-KS:/media/embedded_sys
first_term/ws/example_1$ ./a.out
Child process started
value of n: 20
Now i am coming back to parent process
value of n: 0
Segmentation fault (core dumped)
embedded_system_ks@embedded-KS:/media/embedded_sys
first_term/ws/example_1$
```

the outcome of vfork is not defined. Value of "n" has been printed first time as 20, which is expected. But the next time in the parent process it has printed some garbage or initialized zero value.

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

# Fork() ☺



#LEARN IN DEPTH

#Be professional in embedded system

# QUIZ: what is the output ?

main.c

```c
/*
 * main.c
 *
 *  Created on: Nov 10, 2019
 *      Author: Keroles Shenouda
 */

////////////////////////////
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>


static int idata = 111; /* Allocated in data segment */
int
main(int argc, char *argv[])
{
    int istack = 222; /* Allocated in stack segment */
    pid_t childPid;
    switch (childPid = fork()) {
    case -1:
        perror("fork");
    case 0:
        idata *= 3;
        istack *= 3;
        break;
    default:
        sleep(3); /* Give child a chance to execute */
        break;
    }
    /* Both parent and child come here */
    printf("PID=%ld %s idata=%d istack=%d\n", (long) getpid(),
            (childPid == 0) ? "(child) " : "(parent)", idata, istack);
}
////////////////////////////////
```



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

# QUIZ: what is the output ?

main.c

```c
/*
 * main.c
 *
 *  Created on: Nov 10, 2019
 *      Author: Keroles Shenouda
 */

//////////////////////////////
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>


static int idata = 111; /* Allocated in data segment */
int
main(int argc, char *argv[])
{
    int istack = 222; /* Allocated in stack segment */
    pid_t childPid;
    switch (childPid = fork()) {
    case -1:
        perror("fork");
    case 0:
        idata *= 3;
        istack *= 3;
        break;
    default:
        sleep(3); /* Give child a chance to execute */
        break;
    }
    /* Both parent and child come here */
    printf("PID=%ld %s idata=%d istack=%d\n", (long) getpid(),
            (childPid == 0) ? "(child) " : "(parent)", idata, istack);
}
//////////////////////////////////
```

```
embedded_system_ks@embedded-KS: /media/embedded_system_
embedded_system_ks@embedded-KS:/media/embedded_system_k
first_term/ws/example_1$ gcc main.c
embedded_system_ks@embedded-KS:/media/embedded_system_k
first_term/ws/example_1$ ./a.out
PID=10994 (child)  idata=333 istack=666
PID=10993 (parent) idata=111 istack=222
embedded_system_ks@embedded-KS:/media/embedded_system_k
first_term/ws/example_1$
```

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

# QUIZ: what is the output ?

main.c

```c
/*
 * main.c
 *
 *  Created on: Nov 10, 2019
 *      Author: Keroles Shenouda
 */

//////////////////////////////
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>


static int idata = 111; /* Allocated in data segment */
int
main(int argc, char *argv[])
{
    int istack = 222; /* Allocated in stack segment */
    pid_t childPid;
    switch (childPid = vfork()) {
    case -1:
        perror("fork");
    case 0:
        idata *= 3;
        istack *= 3;
        break;
    default:
        sleep(3); /* Give child a chance to execute */
        break;
    }
    /* Both parent and child come here */
    printf("PID=%ld %s idata=%d istack=%d\n", (long) getpid(),
            (childPid == 0) ? "(child) " : "(parent)", idata, istack);
}
//////////////////////////////////
```


#LEARN IN DEPTH

#Be professional in embedded system

main.c

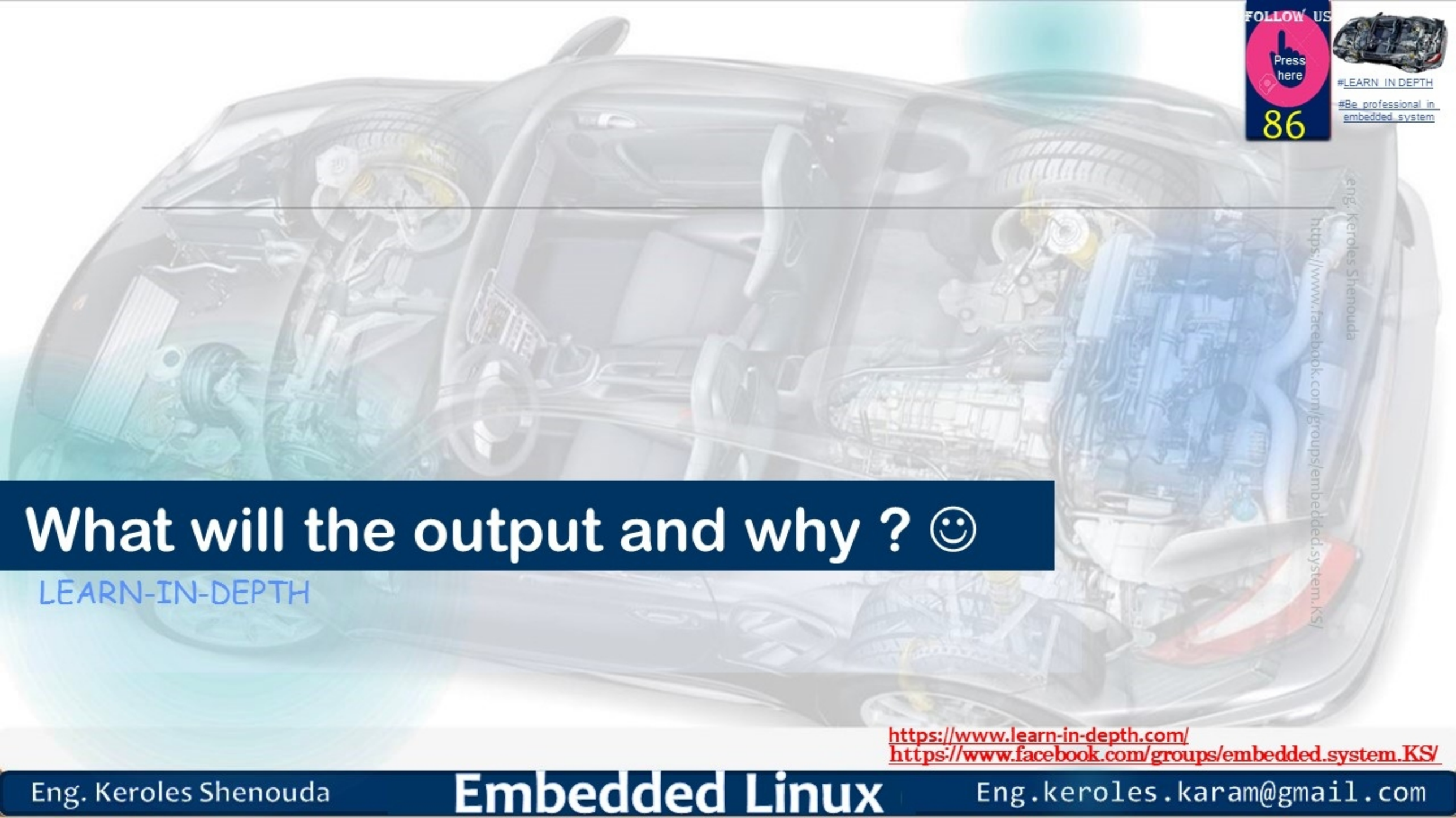
```c
/*
 * main.c
 *
 *  Created on: Nov 10, 2019
 *      Author: Keroles Shenouda
 */

//////////////////////////////
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>


static int idata = 111; /* Allocated in data segment */
int
main(int argc, char *argv[])
{
    int istack = 222; /* Allocated in stack segment */
    pid_t childPid;
    switch (childPid = vfork()) {
    case -1:
        perror("fork");
    case 0:
        idata *= 3;
        istack *= 3;
        break;
    default:
        sleep(3); /* Give child a chance to execute */
        break;
    }
    /* Both parent and child come here */
    printf("PID=%ld %s idata=%d istack=%d\n", (long) getpid(),
            (childPid == 0) ? "(child) " : "(parent)", idata, istack);
}
//////////////////////////////////////
```

```
embedded_system_ks@embedded-KS: /media/embedded_system_ks/Em
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Em
first_term/ws/example_1$ gcc main.c
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Em
first_term/ws/example_1$ ./a.out
PID=10994 (child)  idata=333 istack=666
PID=10993 (parent) idata=111 istack=222
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Em
first_term/ws/example_1$ gcc main.c
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Em
first_term/ws/example_1$ ./a.out
PID=11019 (child)  idata=333 istack=666
PID=11018 (parent) idata=333 istack=-1218393655
Segmentation fault (core dumped)
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Em
first_term/ws/example_1$
```

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/




# What will the output and why ? ☺

LEARN-IN-DEPTH

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

# What will the output and why ? ☺

main.c | dirent.h | dirent.h

```c
#include <errno.h>
#include <dirent.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int GLOBAL1 = 10;
static int GLOBAL2 = 20;
void Fork_lab();

int main (int argc, char *argv[])
{
    Fork_lab();
    return 0 ;
}

void Fork_lab()
{
    int local_process = 30;
    pid_t pid;

    if( (pid = fork()) > 0)  //Parent process
    {
        printf("Parent: initial value of GLOBAL1=%d\n", GLOBAL1);
        printf("Parent: initial value of static GLOBAL2=%d\n", GLOBAL2);
        printf("Parent: initial value of local var=%d\n", local_process);
        GLOBAL1 = 11;
        GLOBAL2 = 21;
        local_process = 31;
        printf("Parent: modified value of GLOBAL1=%d\n", GLOBAL1);
        printf("Parent: modified value of static GLOBAL2=%d\n", GLOBAL2);
        printf("Parent: modified value of local var=%d\n", local_process);
    }
    else    //Child Process
    {
        printf("child: initial value of GLOBAL1=%d\n", GLOBAL1);
        printf("child: initial value of static GLOBAL2=%d\n", GLOBAL2);
        printf("child: initial value of local var=%d\n", local_process);
        GLOBAL1 = 15;
        GLOBAL2 = 25;
        local_process = 35;
        printf("child: modified value of GLOBAL1=%d\n", GLOBAL1);
        printf("child: modified value of static GLOBAL2=%d\n", GLOBAL2);
        printf("child: modified value of local var=%d\n", local_process);
    }

    sleep(1);
    return 0;

}
```

# What will the output and why ? ☺

```
embedded_system_ks@embedded-KS:/media/embedded_system_
LABS/Debug$ ./Posix_LABS
Parent: initial value of GLOBAL1=10
Parent: initial value of static GLOBAL2=20
Parent: initial value of local var=30
Parent: modified value of GLOBAL1=11
Parent: modified value of static GLOBAL2=21
Parent: modified value of local var=31
child: initial value of GLOBAL1=10
child: initial value of static GLOBAL2=20
child: initial value of local var=30
child: modified value of GLOBAL1=15
child: modified value of static GLOBAL2=25
child: modified value of local var=35
embedded_system_ks@embedded-KS:/media/embedded_system_
LABS/Debug$
```

main.c | dirent.h | dirent.h

```c
#include <errno.h>
#include <dirent.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int GLOBAL1 = 10;
static int GLOBAL2 = 20;
void Fork_lab();

int main (int argc, char *argv[])
{
    Fork_lab();
    return 0 ;
}

void Fork_lab()
{
    int local_process = 30;
    pid_t pid;

    if( (pid = fork()) > 0)  //Parent process
    {
        printf("Parent: initial value of GLOBAL1=%d\n", GLOBAL1);
        printf("Parent: initial value of static GLOBAL2=%d\n", GLOBAL2);
        printf("Parent: initial value of local var=%d\n", local_process);
        GLOBAL1 = 11;
        GLOBAL2 = 21;
        local_process = 31;
        printf("Parent: modified value of GLOBAL1=%d\n", GLOBAL1);
        printf("Parent: modified value of static GLOBAL2=%d\n", GLOBAL2);
        printf("Parent: modified value of local var=%d\n", local_process);
    }
    else    //Child Process
    {
        printf("child: initial value of GLOBAL1=%d\n", GLOBAL1);
        printf("child: initial value of static GLOBAL2=%d\n", GLOBAL2);
        printf("child: initial value of local var=%d\n", local_process);
        GLOBAL1 = 15;
        GLOBAL2 = 25;
        local_process = 35;
        printf("child: modified value of GLOBAL1=%d\n", GLOBAL1);
        printf("child: modified value of static GLOBAL2=%d\n", GLOBAL2);
        printf("child: modified value of local var=%d\n", local_process);
    }

    sleep(1);
    return 0;

}
```

when you create child process using fork all the global variables and local variables and everything gets separated in the memory

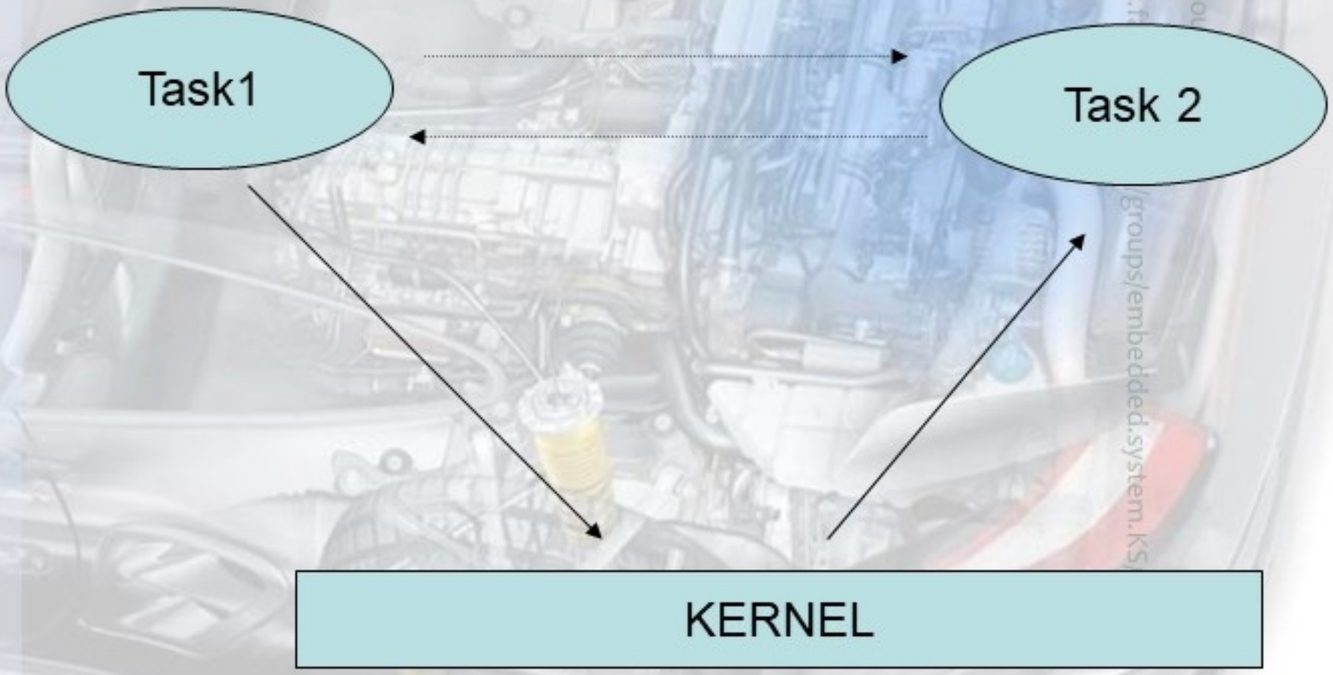


# Signals

- *Signals* are a mechanism for one-way asynchronous notifications
- signal may be sent from the kernel to a process, from a process to another process, or from a process to itself.
- Signals typically alert a process to some event, such as a segmentation fault or the user pressing Ctrl-C.

# Inter-process communication

- All synch and communications are not direct but via the kernel.
- Linux is rich in inter process communication mechanism. These are:
  - Signal
  - Pipe
  - FIFO (named pipe)
  - IPC
    - Message queues
    - Semaphore
    - Shared memory (fastest)





AN IMPORTANT ASPECT OF PROCESS CONTROL IS SIGNAL HANDLING, WHICH DEALS WITH THE INTERACTION BETWEEN A PROCESS AND THE KERNEL IN HANDLING ASYNCHRONOUS EVENT....

Don't know when, but must react once happen

# Signals & Alarm



# What is a Signal?

- A signal is an asynchronous event which is delivered to a process.
- Asynchronous means that the event can occur at any time may be unrelated to the execution of the process.
- Signals are raised by some error conditions, such as memory segment violations, floating point processor errors, or illegal instructions. – e.g. user types ctrl-C, or the modem hangs

# Signals

- Linux supports both POSIX reliable signals "standard signals "and "POSIX real-time signals".
- Each signal has a current disposition, **which determines how the process behaves when it is delivered the signal.**
- By default, **a signal handler** is invoked on the normal process stack. It is possible to arrange that the signal handler uses an alternate stack.
- The signal disposition is a per-process attribute: in a multithreaded

Application

- **A child created via fork inherits a copy of its parent's signal dispositions.**

You have to include #include <signal.h>

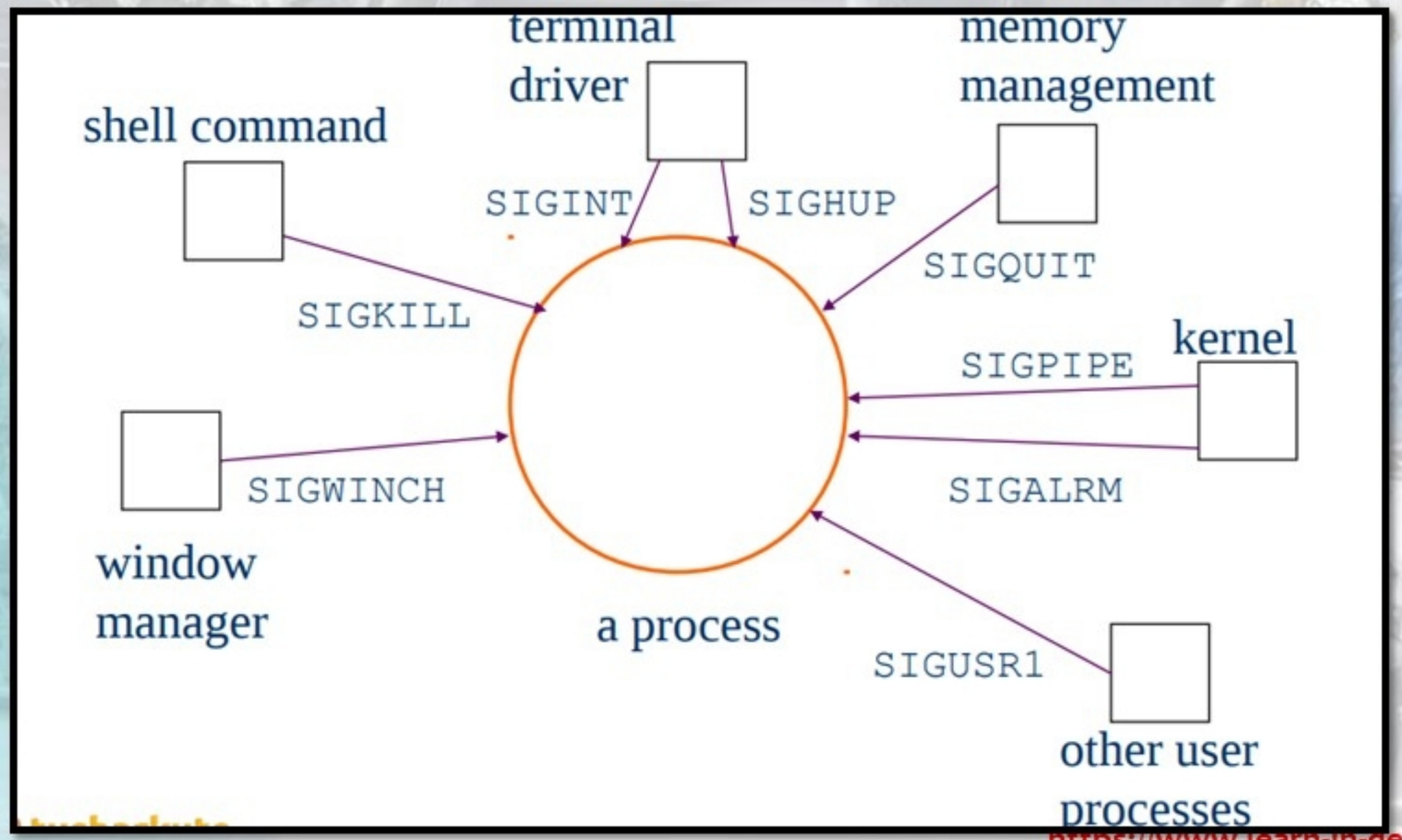
# Signal numbering for standard signals

The numeric value for each signal is given in the table below. As shown in the table, many signals have different numeric values on different architectures

We will focus now on SIGALARM

| Signal | x86/ARM most others | Alpha/ SPARC | MIPS | PARISC | Notes |
|---|---|---|---|---|---|
| SIGHUP | 1 | 1 | 1 | 1 | |
| SIGINT | 2 | 2 | 2 | 2 | |
| SIGQUIT | 3 | 3 | 3 | 3 | |
| SIGILL | 4 | 4 | 4 | 4 | |
| SIGTRAP | 5 | 5 | 5 | 5 | |
| SIGABRT | 6 | 6 | 6 | 6 | |
| SIGIOT | 6 | 6 | 6 | 6 | |
| SIGBUS | 7 | 10 | 10 | 10 | |
| SIGEMT | - | 7 | 7 | - | |
| SIGFPE | 8 | 8 | 8 | 8 | |
| SIGKILL | 9 | 9 | 9 | 9 | |
| SIGUSR1 | 10 | 30 | 16 | 16 | |
| SIGSEGV | 11 | 11 | 11 | 11 | |
| SIGUSR2 | 12 | 31 | 17 | 17 | |
| SIGPIPE | 13 | 13 | 13 | 13 | |
| SIGALRM | 14 | 14 | 14 | 14 | |
| SIGTERM | 15 | 15 | 15 | 15 | |
| SIGSTKFLT | 16 | - | - | 7 | |
| SIGCHLD | 17 | 20 | 18 | 18 | |
| SIGCLD | - | - | 18 | - | |
| SIGCONT | 18 | 19 | 25 | 26 | |
| SIGSTOP | 19 | 17 | 23 | 24 | |
| SIGTSTP | 20 | 18 | 24 | 25 | |
| SIGTTIN | 21 | 21 | 26 | 27 | |
| SIGTTOU | 22 | 22 | 27 | 28 | |
| SIGURG | 23 | 16 | 21 | 29 | |
| SIGXCPU | 24 | 24 | 30 | 12 | |
| SIGXFSZ | 25 | 25 | 31 | 30 | |
| SIGVTALRM | 26 | 26 | 28 | 20 | |
| SIGPROF | 27 | 27 | 29 | 21 | |
| SIGWINCH | 28 | 28 | 20 | 23 | |
| SIGIO | 29 | 23 | 22 | 22 | |
| SIGPOLL | | | | | Same as SIGIO |
| SIGPWR | 30 | 29/- | 19 | 19 | |
| SIGINFO | - | 29/- | - | - | |
| SIGLOST | - | -/29 | - | - | |
| SIGSYS | 31 | 12 | 12 | 31 | |
| SIGUNUSED | 31 | - | - | 31 | |

# Signal Sources





# POSIX predefined signals

- SIGALRM: Alarm timer time-out. Generated by alarm( ) API.
- SIGABRT: Abort process execution. Generated by abort( ) API.
- SIGFPE: Illegal mathematical operation.
- SIGHUP: Controlling terminal hang-up.
- SIGILL: Execution of an illegal machine instruction.
- SIGINT: Process interruption. Can be generated by or keys.
- SIGKILL: Sure kill a process. Can be generated by – “kill -9 “ command.
- SIGPIPE: Illegal write to a pipe.
- SIGQUIT: Process quit. Generated by keys.
- SIGSEGV: Segmentation fault. generated by de-referencing a NULL pointer.



# POSIX predefined signals Cont.

- SIGTERM: process termination. Can be generated by – "kill " command.
- SIGUSR1: Reserved to be defined by user.
- SIGUSR2: Reserved to be defined by user.
- SIGCHLD: Sent to a parent process when its child process has terminated.
- SIGCONT: Resume execution of a stopped process.
- SIGSTOP: Stop a process execution.
- SIGTTIN: Stop a background process when it tries to read from from its controlling terminal.
- SIGTSTP: Stop a process execution by the control_Z keys.
- SIGTTOUT: Stop a background process when it tries to write to its controlling terminal.

# Actions on signals

- Process that receives a signal can take one of three action:


**Perform the system-specified default for the signal**
– notify the parent process that it is terminating;
– generate a core file; (a file containing the current memory image of the process)
– terminate.

**Ignore the signal** – A process can do ignoring with all signal but two special signals: SIGSTOP and SIGKILL.

**Catch the Signal** – When a process catches a signal, except SIGSTOP and SIGKILL, it invokes a special signal handing routine.

# Example of signals

- User types Ctrl-c
  - Event gains attention of OS - OS stops the application process immediately, sending it a 2/SIGINT signal
  - - Signal handler for 2/SIGINT signal executes to completion
  - - Default signal handler for 2/SIGINT signal exits process

| Signal | x86/ARM most others | Alpha/ SPARC | MIPS | PARISC | Notes |
|---|---|---|---|---|---|
| SIGHUP | 1 | 1 | 1 | 1 | |
| SIGINT | 2 | 2 | 2 | 2 | |

# Example of signals

- Process makes illegal memory reference
  - Event gains attention of OS
  - - OS stops application process immediately, sending it a 11/SIGSEGV signal
  - - Signal handler for 11/SIGSEGV signal executes to completion
  - - Default signal handler for 11/SIGSEGV signal prints "segmentation fault" and exits process

| | | | | |
|---|---|---|---|---|
| SIGUSR1 | 10 | 30 | 16 | 16 |
| SIGSEGV | 11 | 11 | 11 | 11 |
| SIGUSR2 | 12 | 31 | 17 | 17 |

# Send signals via commands

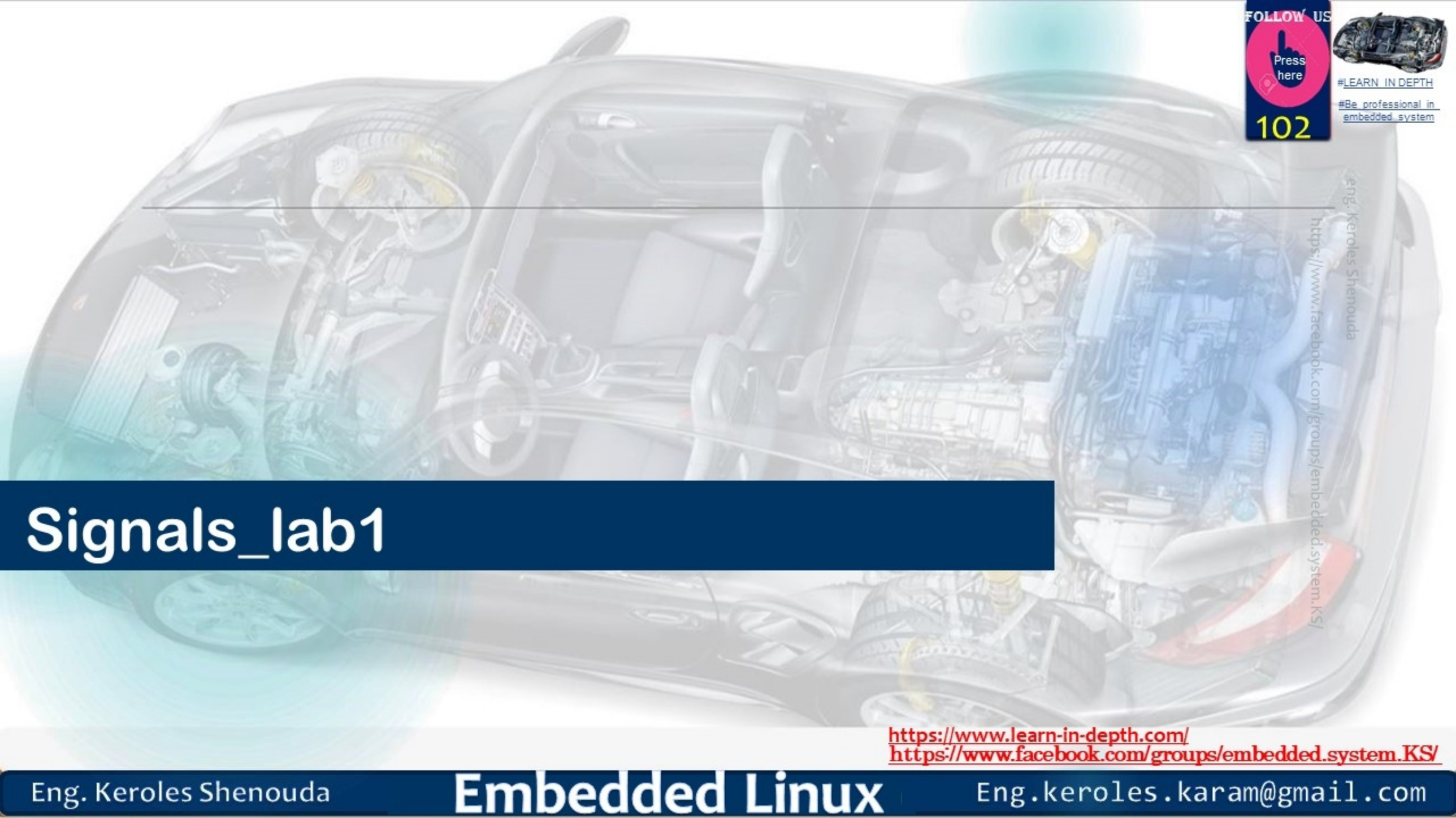
- kill Command
  - -kill signal pid
  - Send a signal of type signal to the process with id **pid**
  - Can specify either signal type name (-SIGINT) or number (-2)

Examples

kill -2 1234

kill SIGINT 1234

- Same as pressing Ctrl-c if process 1234 is running in foreground

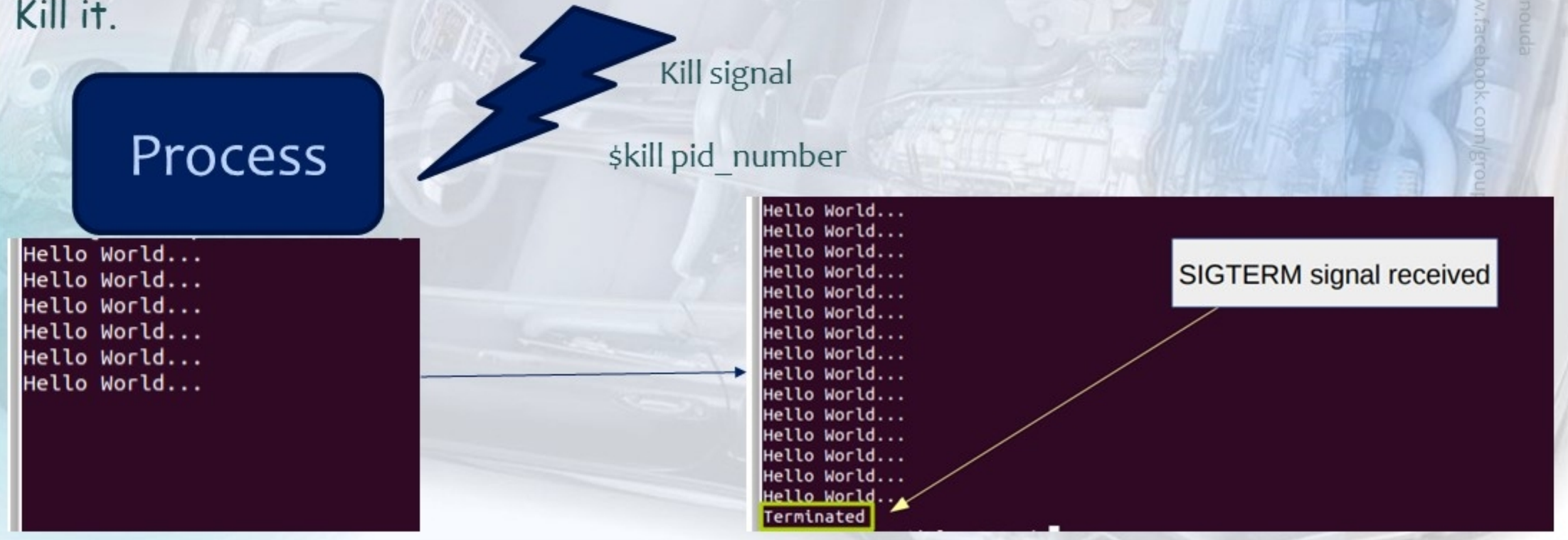



# Signals_lab1

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

# Signals_lab1

- Create an application print "Hello World..." forever and then Kill it.





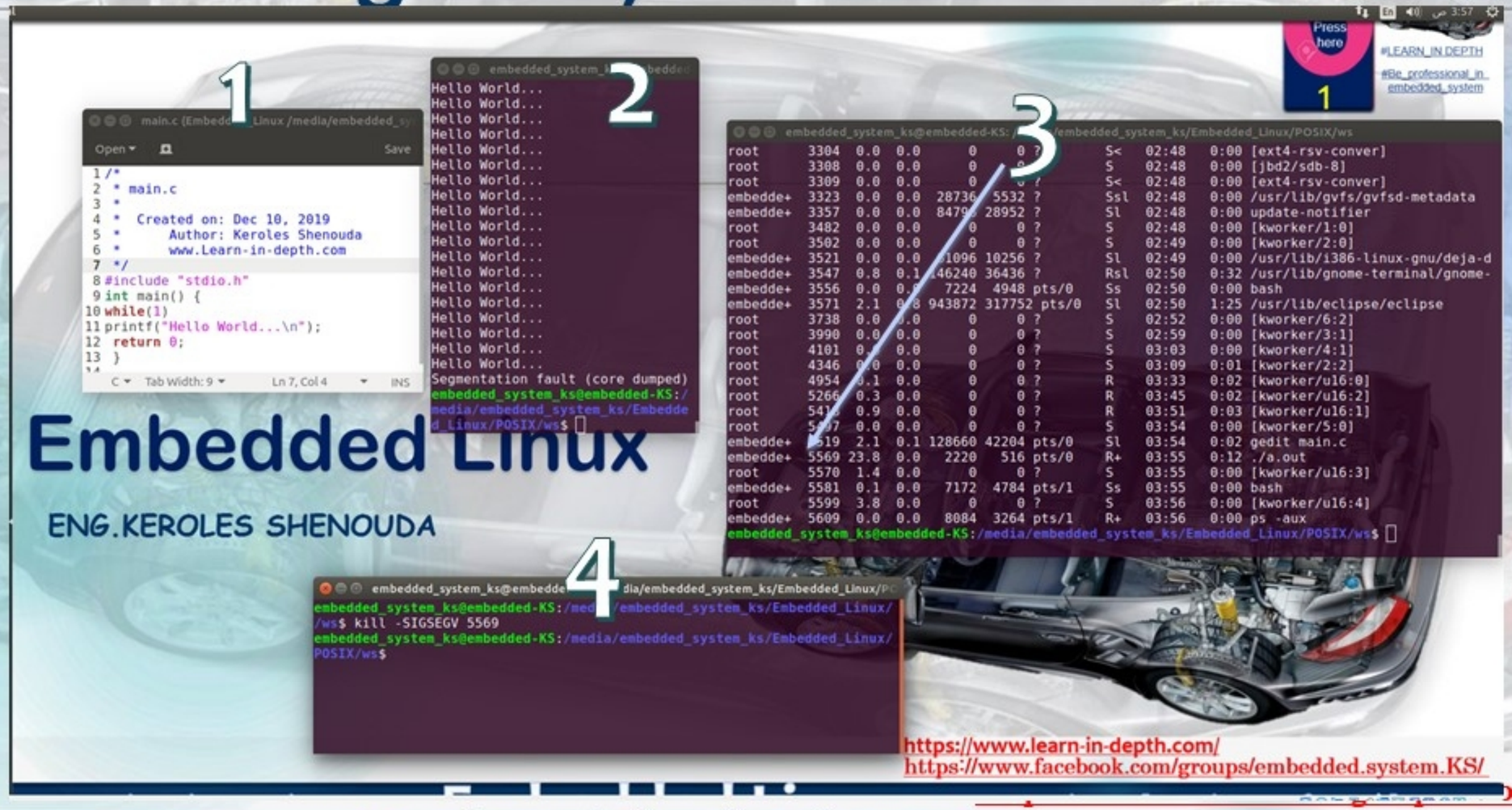
# Signal_lab2 (Killing process by different signals )

- The same application in lab1, we need to sed a segmentation fault signal to this application

- **SIGSEGV: Segmentation fault. generated by de-referencing a NULL pointer.**

- To pass it by this command
- Kill -**SIGSEGV** PID

# Signal_lab2 (Killing process by different signals )



# Signal Function

- Programs can handle signals using the signal library function.
- **void (*signal(int signo, void (*func)(int)))(int);**
  - **signo** is the signal number to handle
  - **func** defines how to handle the signal
    - SIG_IGN (ignore)
    - SIG_DFL (default)
    - - Function pointer of a custom handler
    - Returns previous disposition if ok, or SIG_ERR on error

# Signal_lab3 what will be the output ? Ctr+c

```c
/*
 * main.c
 *
 *  Created on: Dec 10, 2019
 *      Author: Keroles Shenouda
 *      www.Learn-in-depth.com
 */
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void My_Signal_Handler(int sig)
{
        printf("My_Signal_Handler!  I got signal %d\n", sig);
//set the signal default handler
        (void) signal(SIGINT, SIG_DFL);
}
int main()
{
//pass the signal handler for dignal "SIGINT"
        (void) signal(SIGINT, My_Signal_Handler);
        while(1)
        {
                printf("Lern-In-Depth!\n");
                sleep(1);
        }
        return 0;
}
```



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

# Signal_lab3 what will be the output ? Ctr+c


#LEARN_IN DEPTH

#Be_professional in embedded system

```c
/*
 * main.c
 *
 *  Created on: Dec 10, 2019
 *      Author: Keroles Shenouda
 *      www.Learn-in-depth.com
 */
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void My_Signal_Handler(int sig)
{
        printf("My_Signal_Handler!  I got signal %d\n", sig);
//set the signal default handler
        (void) signal(SIGINT, SIG_DFL);
}
int main()
{
//pass the signal handler for dignal "SIGINT"
        (void) signal(SIGINT, My_Signal_Handler);
        while(1)
        {
                printf("Lern-In-Depth!\n");
                sleep(1);
        }
        return 0;
}
```

First Ctr+C

Second Ctr+C

SIG_DFL Handler

```
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/POSIX/ws$ ./a.out
Lern-In-Depth!
Lern-In-Depth!
Lern-In-Depth!
Lern-In-Depth!
Lern-In-Depth!
Lern-In-Depth!
^CMy_Signal_Handler!   I got signal 2
Lern-In-Depth!
Lern-In-Depth!
Lern-In-Depth!
Lern-In-Depth!
Lern-In-Depth!
Lern-In-Depth!
^C
embedded_system_ks@embedded-KS:/media/embedded_system_ks/Embedded_Linux/POSIX/ws$
```

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/


Learn-in-depth.com



# References


- Embedded Linux training
  - https://bootlin.com/training/
- Linux kernel and driver development training
- Yocto Project and OpenEmbedded development training
- Buildroot development training
- Building Embedded Linux Systems, 2nd Edition
- Mastering Embedded Linux Programming - Second Edition
- Christopher Hallinan, **Embedded Linux Primer,** Prentice Hall, 2006.
- **Silberschatz**, Galvin, and Gagne's **Operating System Concepts**, **Eighth Edition**.
- Robert love, linux kernel develoepemnt 3 rd edition 2010
- **Advanced Linux Programming By Mark Mitchell, Jeffrey Oldham, Alex Samuel**
- **Linux OS in Embedded Systems & Linux Kernel Internals(2/2)**
  - Memory Management, Paging, Virtual Memory, File system and its implementation, Secondary Storage(HDD), I/O systems




https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/





- Memory Management article
- Introduction to Operating Systems Class 9 - Memory Management
- Virtual Memory lecture for Introduction to Computer Architecture at Uppsala University.
- OS Lecture Note 13 - Address translation, Caches and TLBs
- CSE 331 Operating Systems Design lectures
- CSE 331 Virtual Memory
- CSE 332 Computer Organization and Architecture
- File Systems: Fundamentals.
- Linux System Programming
- System Calls, POSIX I/O
  CSE 333 Spring 2019, Justin Hsia

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/





- **File Permissions in Linux/Unix with Example**
- Introduction to UNIX / Linux – 4
- **12.2 Basic I/O Concepts**
- Communicating with Hardware
- I/O Systems I/O Hardware Application I/O Interface
- COMPUTER ARCHITECTURE
- **OS**
- Linux Operating System
- W4118 Operating Systems, Instructor: Junfeng Yang
- CSNB334 Advanced Operating Systems 4. Process & Resource Management.
- Using the POSIX API
- **Linux Memory Management**
- Chapter 2: Operating-System Structures
- **POSIX Threads Programming**
- Pthreads: A shared memory programming model https://slideplayer.com/slide/8734550/
- Signal Handling in Linux Tushar B. Kute
- Introduction to Sockets Programming in C using TCP/IP