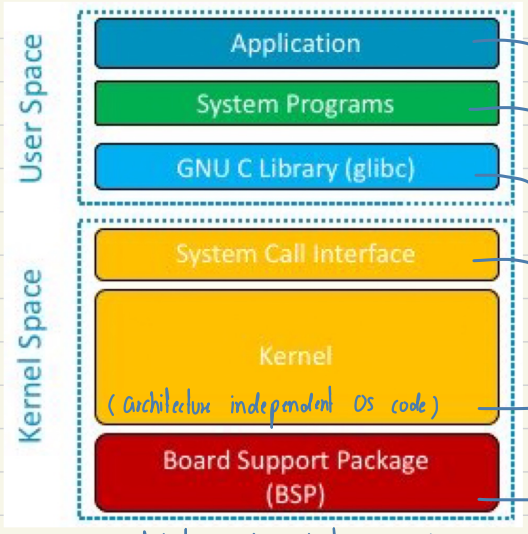
## Linux Architecture

- user space and kernel space · แยกส่วนแล: เป็น independent ต่อกัน
- user and kernel communicate ผ่าน system calls




software delivered to embedded system user

access OS service

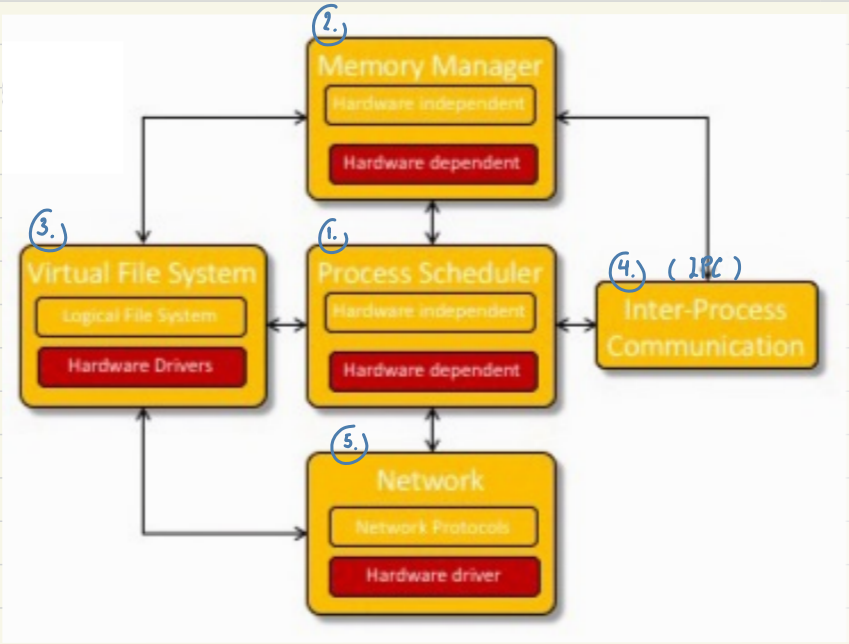
interface between User and Kernel

entry point to access the service

hardware agnostic service of OS ex. process scheduler

hardware specific service of OS ex. context switch

## Conceptual View of Kernel



1. เป็น response ของการตัดสินใจว่า task ไหนจะได้ run

## Main functions:

- Allows processes to create new copies of themselves
- Implements CPU scheduling policy and context switch
- Receives, interrupts, and routes them to the appropriate Kernel subsystem
- Sends signals to user processes
- Manages the hardware timer
- Cleans up process resources when a processes finishes executing
- Provides support for loadable Kernel modules

## External interface:

- System calls interface towards the user space (e.g. `fork()`)
- Intra-Kernel interface towards the kernel space (e.g. `create_module()`)

## Scheduler tick:

- Directly from system calls (e.g. `sleep()`)
- Indirectly after every system call
- After every slow interrupt

## Interrupt type:

- Slow: traditional interrupt (e.g. coming from a disk driver)
- Fast: interrupt corresponding to very fast operations (e.g. processing a keyboard input)

2. Use MMU ( memory management unit ) to map virtual address to physical address

## It is responsible for handling:

- Large address space: user processes can reference more RAM memory than what exists physically
- Protection: the memory for a process is private and cannot be read or modified by another process; also, the memory manager prevents processes from overwriting code and read-only-data.
- Memory mapping: processes can map a file into an area of virtual memory and access the file as memory.
- Fair access to physical memory: it ensures that processes all have fair access to the memory resources, ensuring reasonable system performance.
- Shared memory: it allows processes to share some portion of their memory (e.g. executable code is usually shared amongst processes).

## Advantages:

- Processes can be moved among physical memory maintaining the same virtual addresses.
- The same physical memory may be shared among different processes.

- swaps process memory ออกไปที่ paging file ตอนที่หน่วยความจำไม่พอใช้ - kswapd

The MMU detects when a user process accesses a memory address that is not currently mapped to a physical memory location.

The MMU notifies the Linux Kernel the event known as page fault.

The memory manager subsystem resolves the page fault.

If the page is currently swapped out to the paging file, it is swapped back in.

If the memory manager detects an invalid memory access, it notifies the event to the user process with a signal.

If the process doesn't handle this signal, it is terminated.

Memory manage external interface

System call interface:

- `malloc()/free()`: allocate or free a region of memory for the process's use
- `mmap()/munmap()/msync()/mremap()`: map files into virtual memory regions
- `mprotect()`: change the protection on a region of virtual memory
- `mlock()/mlockall()/munlock()/munlockall()`: super-user routines to prevent memory being swapped
- `swapon()/swapoff()`: super-user routines to add and remove swap files for the system

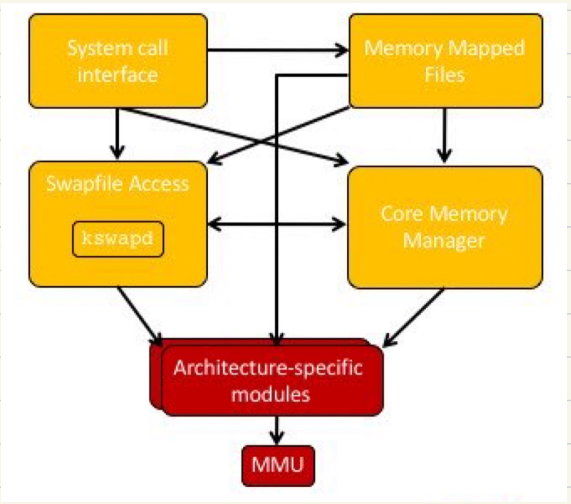
Intra-Kernel interface:

- `kmalloc()/kfree()`: allocate and free memory for use by the kernel's data structures
- `verify_area()`: verify that a region of user memory is mapped with required permissions
- `get_free_page()/free_page()`: allocate and free physical memory pages

component ใน memory manager architecture

## Main components:

- System call interface: it provides memory manager services to the user space.
- Memory mapped files: it implements memory file mapping algorithms.
- Core memory manager: it is responsible for implementing memory allocation algorithms.
- Swapfile access: it controls the paging file access.
- Architecture-specific modules: they handle hardware-specific operations related to memory management (e.g. access to the MMU).




3.)

## It is responsible for handling:

- Multiple hardware devices: it provides uniform access to hardware devices.
- Multiple logical file systems: it supports many different logical organizations of information on storage media.
- Multiple executable formats: it supports different executable file formats (e.g. a.out, ELF).
- Homogeneity: it presents a common interface to all of the logical file systems and all hardware devices.
- Performance: it provides high-speed access to files
- Safety: it enforces policies to not lose or corrupt data
- Security: it enforces policies to grant access to files only to allowed users, and it restricts user total file size with quotas.

## External interface:

- System-call interface based on normal operations on file from the POSIX standard (e.g. open/close/read/write)
- Intra-kernel interface based on i-node interface and file interface

( i-node : data structure ที่เก็บ information ทั้งหมดเกี่ยวกับ file ยกเว้น its name แต่: data ที่มัน contain ไว้

เมื่อ file ถูกสร้างมันจะถูก assign ด้วย unique i-node number (int) )

## i-node Interface

`create()`: creates a file in a directory

`lookup()`: finds a file by name within a directory

`link()/symlink()/unlink()/readlink()/follow_link()`: manages file system links

`mkdir()/rmdir()`: creates or removes sub-directories

`mknod()`: creates a directory, special file, or regular file

`readpage()/writepage()`: reads or writes a page of physical memory

`truncate()`: sets the length of a file to zero

`permission()`: checks to see if a user process has permission to execute an operation

`smap()`: maps a logical file block to a physical device sector

`bmap()`: maps a logical file block to a physical device block

`rename()`: renames a file or directory

## File Interface

`open()/release()`: opens or closes the file

`read()/write()`: reads or writes the file

`select()`: waits until the file is in a particular state (readable or writeable)

`lseek()`: moves to a particular offset in the file

`mmap()`: maps a region of the file onto the virtual memory of a user process

`fsync()/fasync()`: synchronizes any memory buffers with the physical device

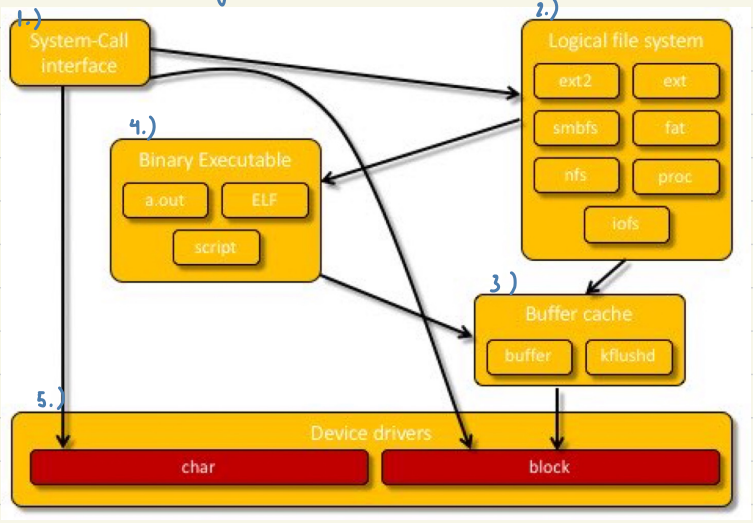
`readdir()`: reads the files that are pointed to by a directory file

`ioctl()`: sets file attributes

`check_media_change()`: checks to see if a removable media has been removed

`revalidate()`: verifies that all cached information is valid

virtual file system architecture




1.) provides virtual file system service to user space

2.) provide logical structure สำหรับการเก็บข้อมูลใน storage medium

3.) provides data caching mechanisms ในการเพิ่มประสิทธิภาพของ storage media access operations

4.) support executable file ใน type ต่างๆ ให้ user

5.) Device drivers provide a uniform interface to access hardware devices:

- Character-based devices are hardware devices accessed sequentially (e.g. serial port).
- Block-based devices are devices that are accessed randomly and whose data is read/written in blocks (e.g. hard disk unit).

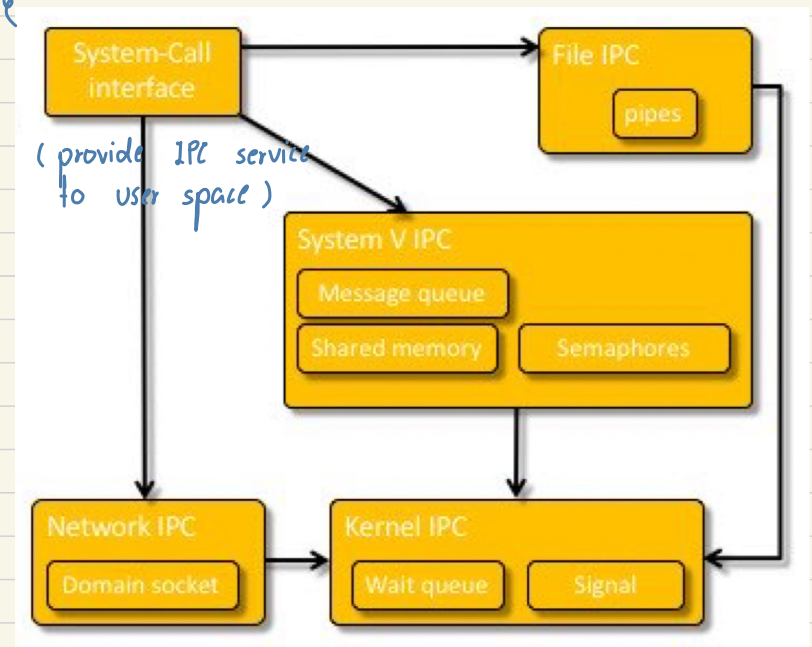
(4.) Machanism to processes

- Resource sharing
- Synchronization
- Data exchange

The following IPCs are supported:

- Pipes
- Message queues
- Shared memory
- Semaphores
- Domain sockets
- Wait queues
- Signals

IPC architecture




( provide IPC service to user space )

( IPC สามารถ take many forms ∴ different process จะ communicate in different method )

(5.) Provides support for network connectivity

- It implements network protocols (e.g. TCP/IP) through hardware-independent code.
- It implements network card drivers through hardware-specific code.

# Device trees

ในการ manage hardware resource , kernel จะต้องรู้ว่า resource ในระบบ embedded system นั้น

1.) hardcode อยู่ใน kernel binary code ( ทุกครั้งที่มีการแก้ไข hardware definition ต้อง compile source code ใหม่ )

2.) provide it to kernel ตอน bootloader ให้ binary file ( device tree blob ) DTB

ซึ่งมาจาก device tree source ( DTS )

- A hardware definition can be changed more easily as only DTS recompilation is needed.
- Kernel recompilation is not needed upon changes to the hardware definition. This is a big time saver.

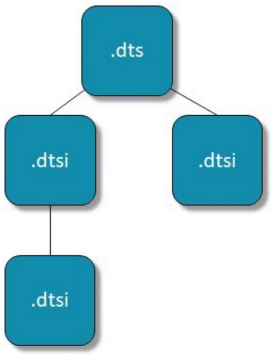
In Arm architecture, all device tree source files are now located in either `arch/arm/boot/dts` or `arch/arm64/boot/dts`.

- `.dts` files for board-level definitions
- `.dtsi` files for included files

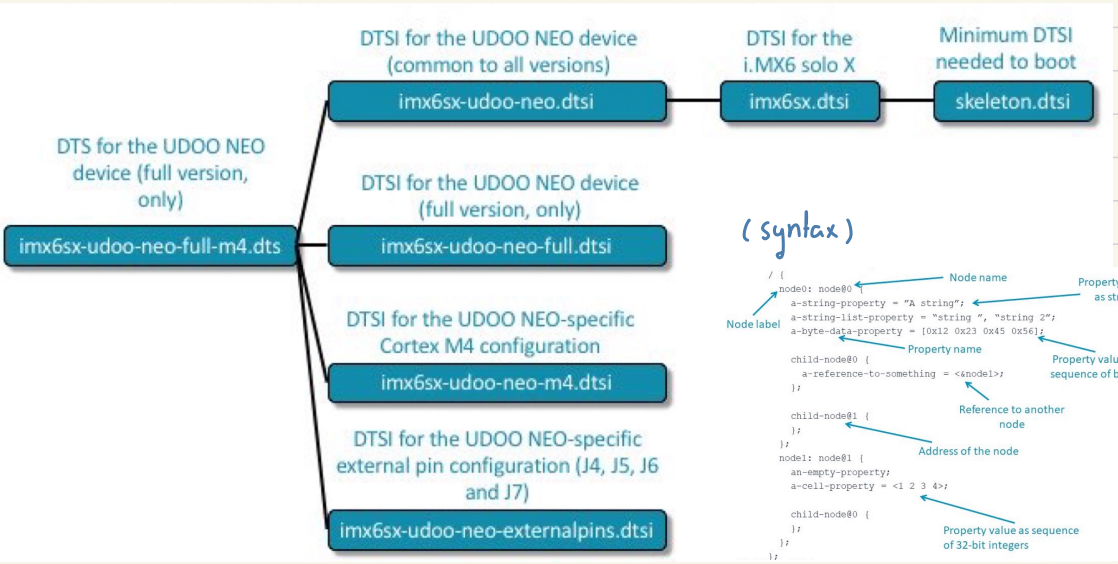
A tool, the device tree compiler, compiles the source into a binary form: the device tree blob (DTB).

- The DTB is loaded by the bootloader and parsed by the kernel at boot time.

Device tree files are not monolithic. They can be split in several files, including each other.




## Device tree ex for UDOO NEO




( syntax )

Under the root of the device tree, we can find:

A `cpus` node, which sub nodes describe each CPU in the system

A `memory` node, which defines the location and size of the RAM

A `chosen` node, which is used to pass parameters to kernel (the kernel command line) at boot time

An `aliases` node, to define shortcuts to certain nodes

One or more nodes defining the buses in the SoC

One or mode nodes defining on-board devices

```
/ {
    alias {};

    cpus {};

    apb@80000000 {
      apbh@80000000 {
        /* some devices */
      };
      apbx@80040000 {
        /* some devices */
      };
    };

    chosen {
      bootargs = "root=/dev/nfs";
    };
};
```

The following properties are used:

- `reg = <address1 length1 […] >`, which lists the address sets (each defined as starting address, length) assigned to the node
- `#address-cells = <num of addresses>`, which states the number of address sets for the node
- `#size-cells=<num of size cells>`, which states the number of size for each set

Note:

- Every node in the tree that represents a device is required to have the `compatible` property.
- `compatible` is the key Linux uses to decide which device driver to bind to a device.

```
/ {
    cpus {
      #address-cells = <1>;
      #size-cells = <0>;
      cpu@0 {
        compatible = "arm,cortex-a9";
        reg = <0>;
      };
      cpu@1 {
        compatible = "arm,cortex-a9";
        reg = <1>;
      };
    };
};
```

CPU addressing

- Each CPU is associated with a unique ID only.
- `#size-cells=<0>`, always

Memory mapped devices

- Typically defined by one 32-bit based address, and one 32-bit length
- `#address-cells=<1>`
- `#size-cells=<1>`

```
cpus {
  #address-cells = <1>;
  #size-cells = <0>;
  cpu@0 {
    compatible = "arm,cortex-a9";
    reg = <0>;
  };
};

/{
  #address-cells = <1>;
  #size-cells = <1>;
  gpio1: gpio@0209c000 {
    compatible = "fsl,imx6sx-gpio",
                 "fsl,imx35-gpio";
    reg = <0x0209c000 0x4000>;
  };
};
```

External bus with chip select line

- Typically, `address-cells` uses 2 cells for the address value: one for the chip select number and one for the offset from the base of the chip select.
- `#address-cells=<2>`
- The length field remains as a single cell.
- `#size-cells=<1>`
- The mapping between bus addressing and CPU addressing is defined by the `ranges` property.

```
external-bus {
  #address-cells = <2>
  #size-cells = <1>;

  ranges = <0 0  0x10100000   0x10000
            1 0  0x10160000
0x10000>;
  ethernet@0,0 {
    compatible = "smc,smc91c111";
    reg = <0 0 0x1000>;
  };
  i2c@1,0 {
    compatible = "acme,a1234-i2c-bus";
    reg = <1 0 0x1000>;
     #address-cells = <1>
     #size-cells = <0>;
    rtc@58 {
      compatible = "maxim,ds1338";
      reg = <58>;
    };
  };
};
```

Bus address

Corresponding CPU address and range

# U - Bootloader

## ( U - Boot · first and second stage bootloader )

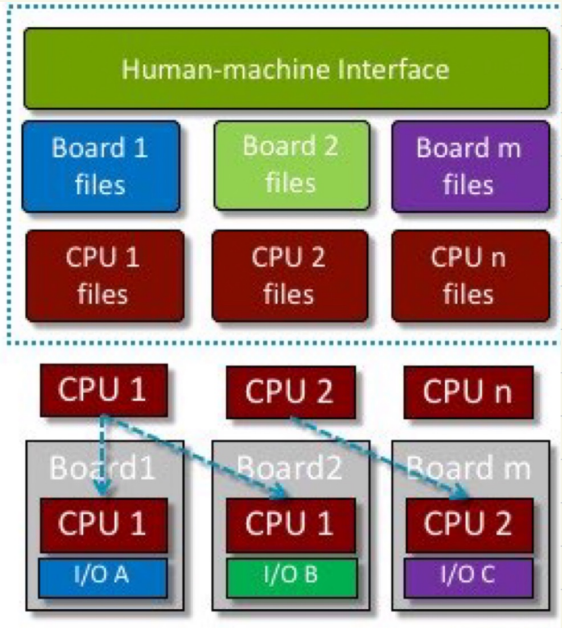
### 1st half

- Written mostly in Assembly code
- It runs from the CPU on-chip memory (e.g., on-chip static RAM).
- It initializes the CPU RAM memory controller and relocates itself in off-chip RAM Memory.

### 2nd half

- Written mostly in C code
- It implements a command-line human-machine interface with scripting capabilities.
- It initializes the minimum set of peripherals to load the device tree Blob, the Linux Kernel, and possibly, the Initial RAM disk to RAM Memory.
- It starts the execution of the Linux Kernel.

## U - Boot source code , U - Boot Bootloader จะแบ่งเป็น group




- Processor - dependent file
  ↳ specific to the CPU ที่จะ run U-Boot
- Board - dependent file
  ↳ specific for board hosting the above CPU (may have different sets of I/O)
- General - purpose file
  ↳ ใช้ร่วมกับ board / CPU

## UDOO NEO Boot Process

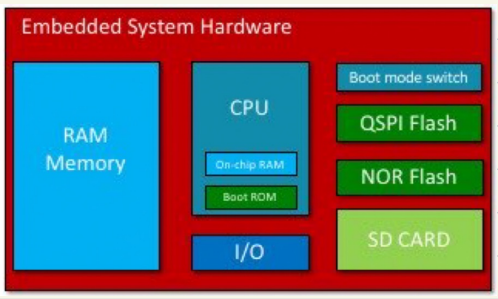
Modern CPUs are capable of booting from multiple sources:

- ROM memory
- Parallel I/O Flash memory (e.g. NOR Flash)
- Serial I/O Flash memory (e.g. QSPI Flash)
- SD Card

An on-chip firmware (stored in on-chip Boot ROM) and I/O configuration (boot mode switch) tell the CPU where to boot from.

This firmware is know as 1st stage bootloader.

Bootloaders such as U-Boot are known as 2nd stage bootloaders.



# Quiz

1. "Which of these components are found in the user space of a Linux architecture?"

- [x] Application
- [ ] Kernel
- [x] System Programs
- [ ] Board Support Package (BSP)

2. "Which of these components are found in the kernel space of a Linux architecture?"

- [ ] GNU C Library (glibc)
- [x] System Call Interface
- [x] Kernel
- [ ] System Programs

3. "What are the main functions of the 'process scheduler'?"

- [x] Implement CPU scheduling policy and context switch
- [x] Managing the hardware timer
- [ ] Handling the address space
- [ ] Mapping files into an area of virtual memory

4. "What are the main functions of the 'memory manager'?"

- [x] Ensures processes have fair access to the memory resources
- [x] Allowing processes to share some portion of their memory
- [x] Enabling processes to map files into an area of virtual memory
- [ ] Enforcing policies to not lose or corrupt data

5. "What is the 'Virtual File-System' responsible for?"

- [ ] Implementing network protocols through hardware-independent code
- [ ] Cleaning up process resources when a process finishes executing
- [x] Providing uniform access to hardware devices
- [x] Enforcing policies to grant access to files only to allowed users

6. "Which of the following is not provided by inter-process communication?"

- ( ) Resource sharing
- (x) System programs access
- ( ) Synchronization
- ( ) Data exchange

7. "Which stages of the booting process does U-Boot handle?"

- (x) U-Boot is a first and second stage bootloader
- ( ) U-Boot is just a first stage bootloader
- ( ) U-Boot is just a second stage bootloader
- ( ) U-Boot is not related to the booting process