reSilient coMputer archItectures and LIfE Sciences

Politecnico di Torino

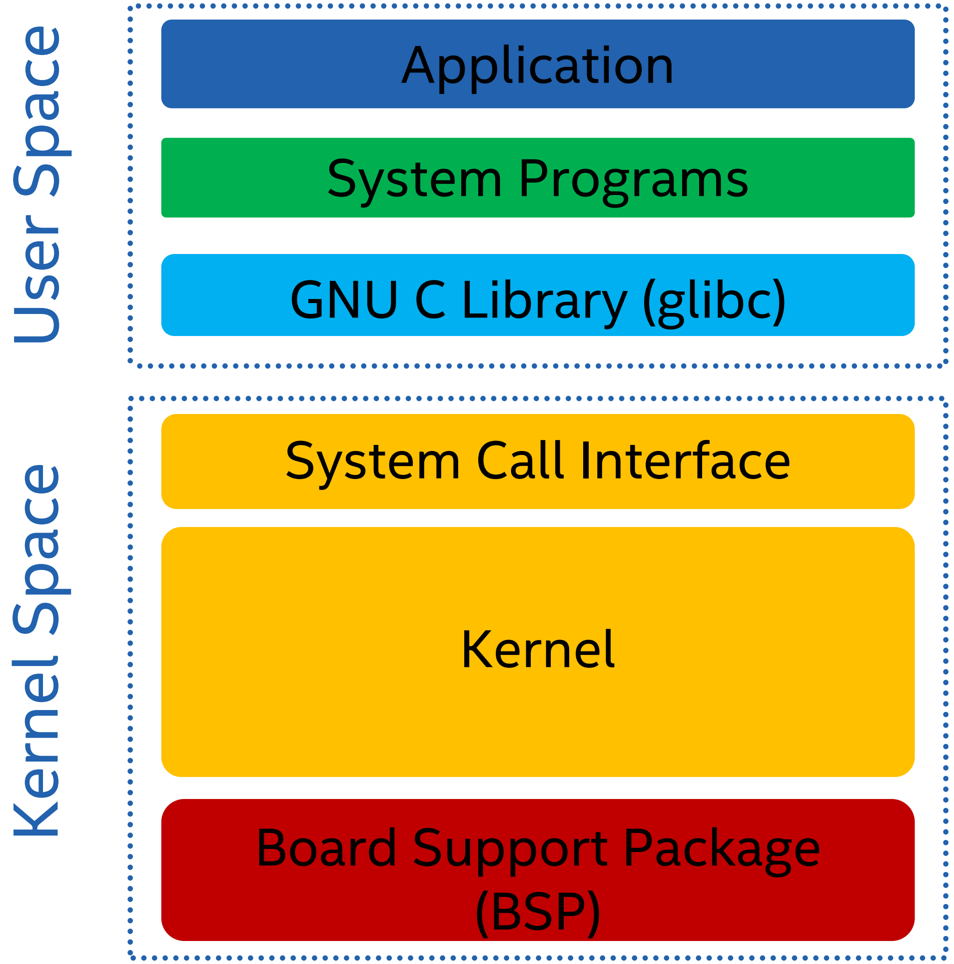
Department of Control and Computer Engineering

ANATOMY OF A LINUX-BASED SYSTEM

STEFANO DI CARLO

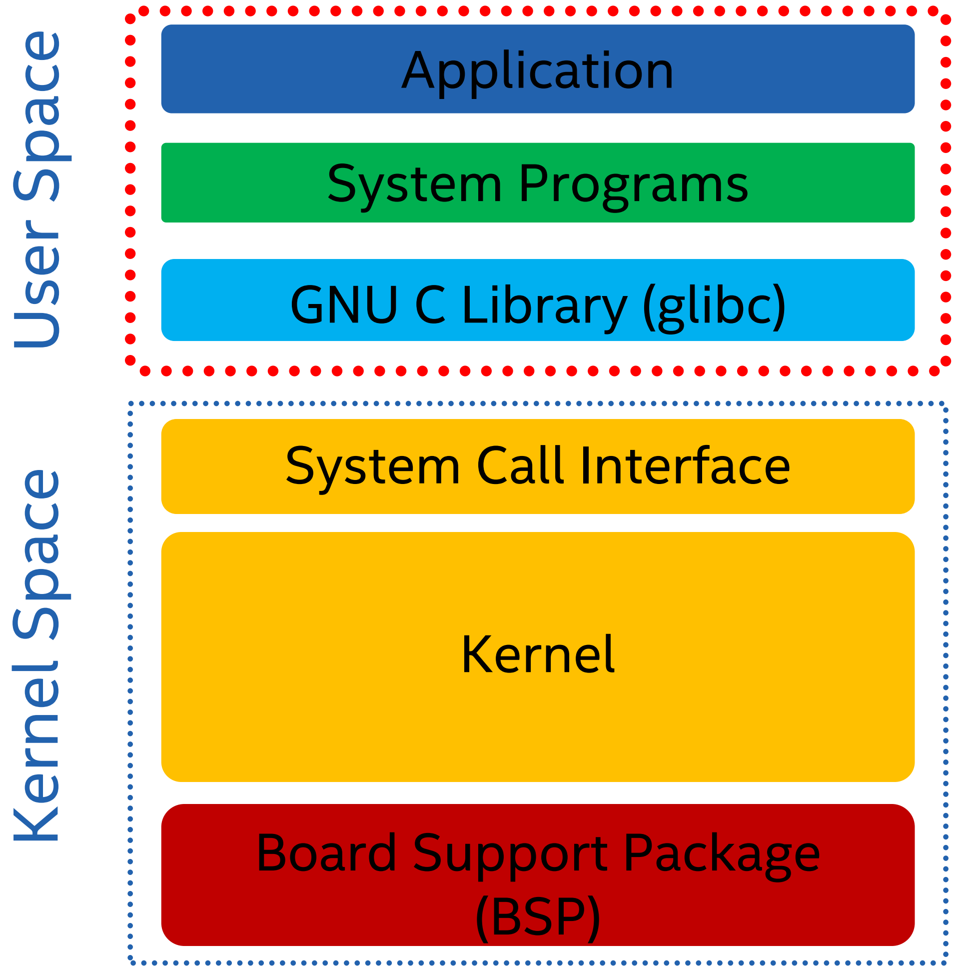
LINUX ARCHITECTURE

- Layered architecture based on two levels:
  - User space
  - Kernel space
- User space and kernel space are independent and isolated
- User space and kernel space communicate through special purpose functions known as system calls



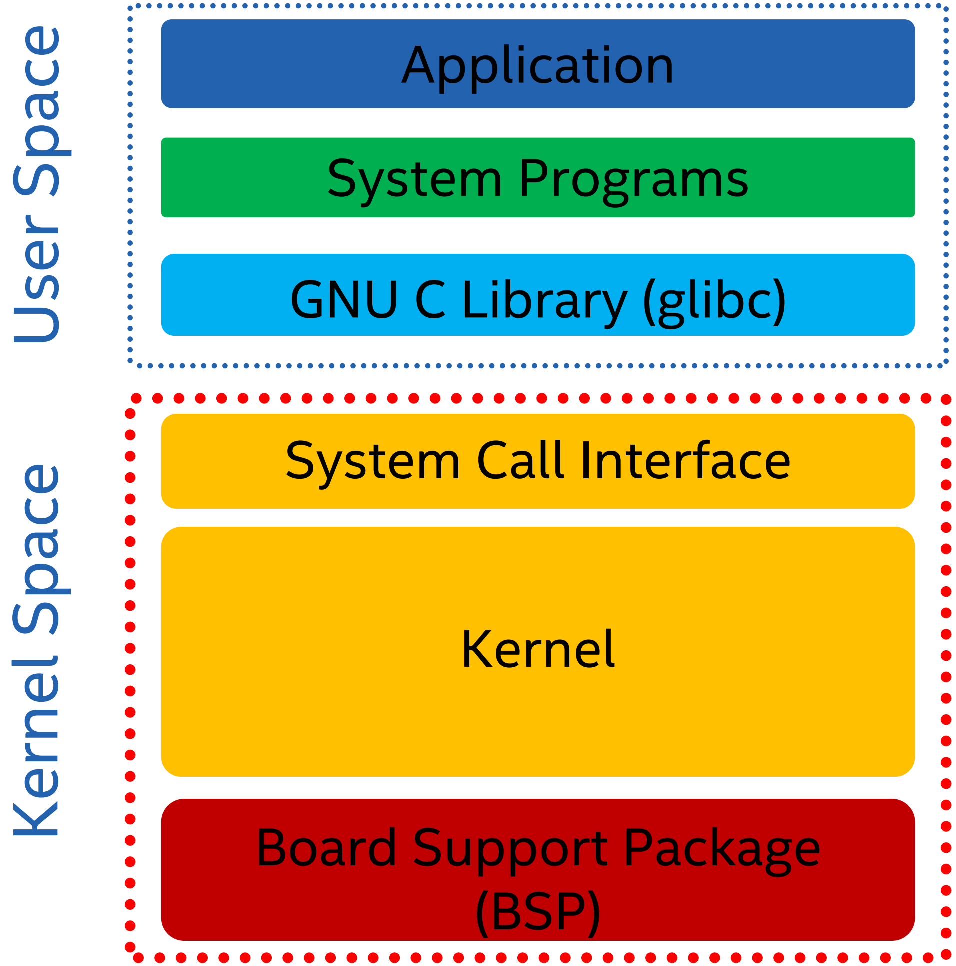
LINUX ARCHITECTURE

- Application
  - Software implementing the functionalities to be delivered to the embedded system user
- System programs
  - User-friendly utilities to access operating system services
- GNU C Library (glibc)
  - Interface between the User Space and the Kernel Space



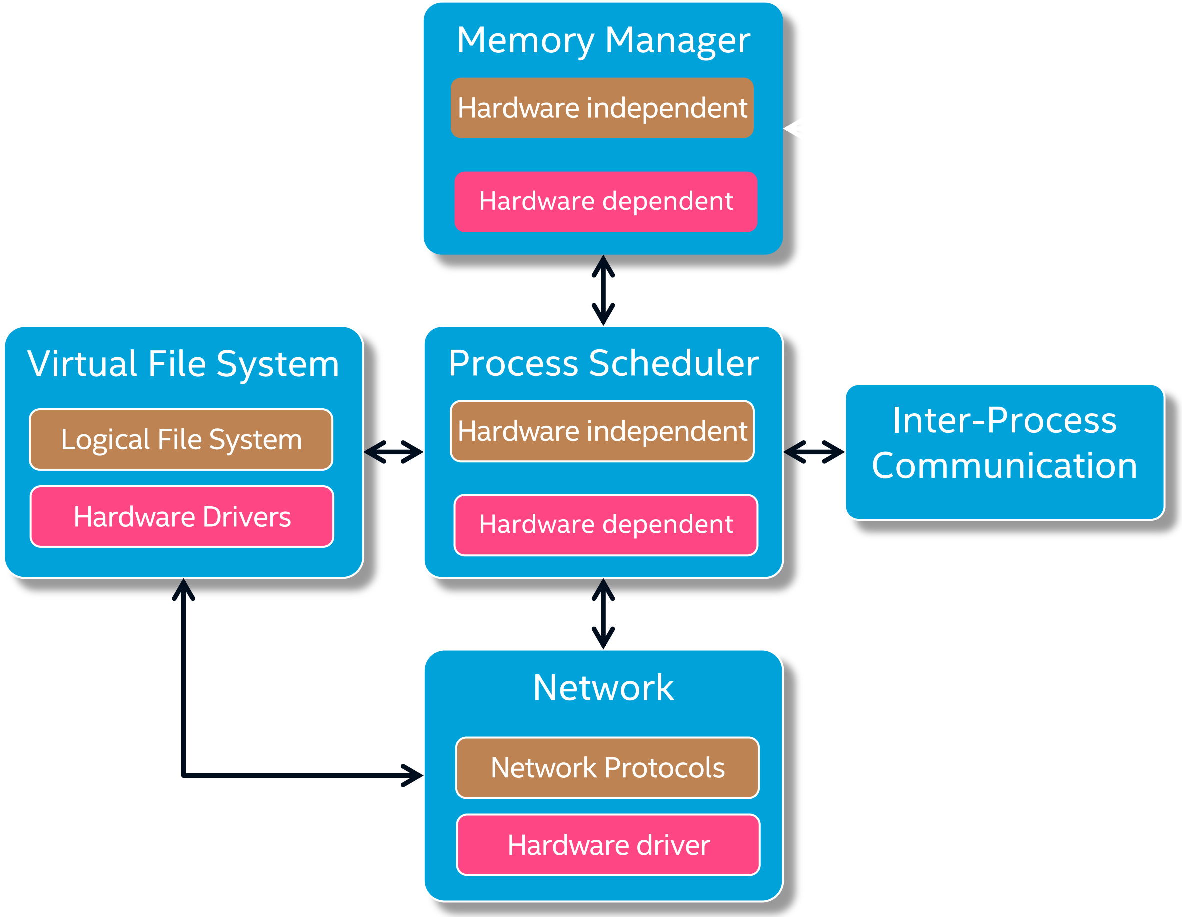
LINUX ARCHITECTURE

- System call interface
  - Entry points to access the services provided by the Kernel (process management, memory management)
- Kernel
  - Architecture-independent operating system code
  - It implements the hardware-agnostic services of the operating system (e.g. the process scheduler).
- Board Support Package (BSP)
  - Architecure-dependant operating system code
  - It implements the hardware specific services of the operating system (e.g. the context switch).



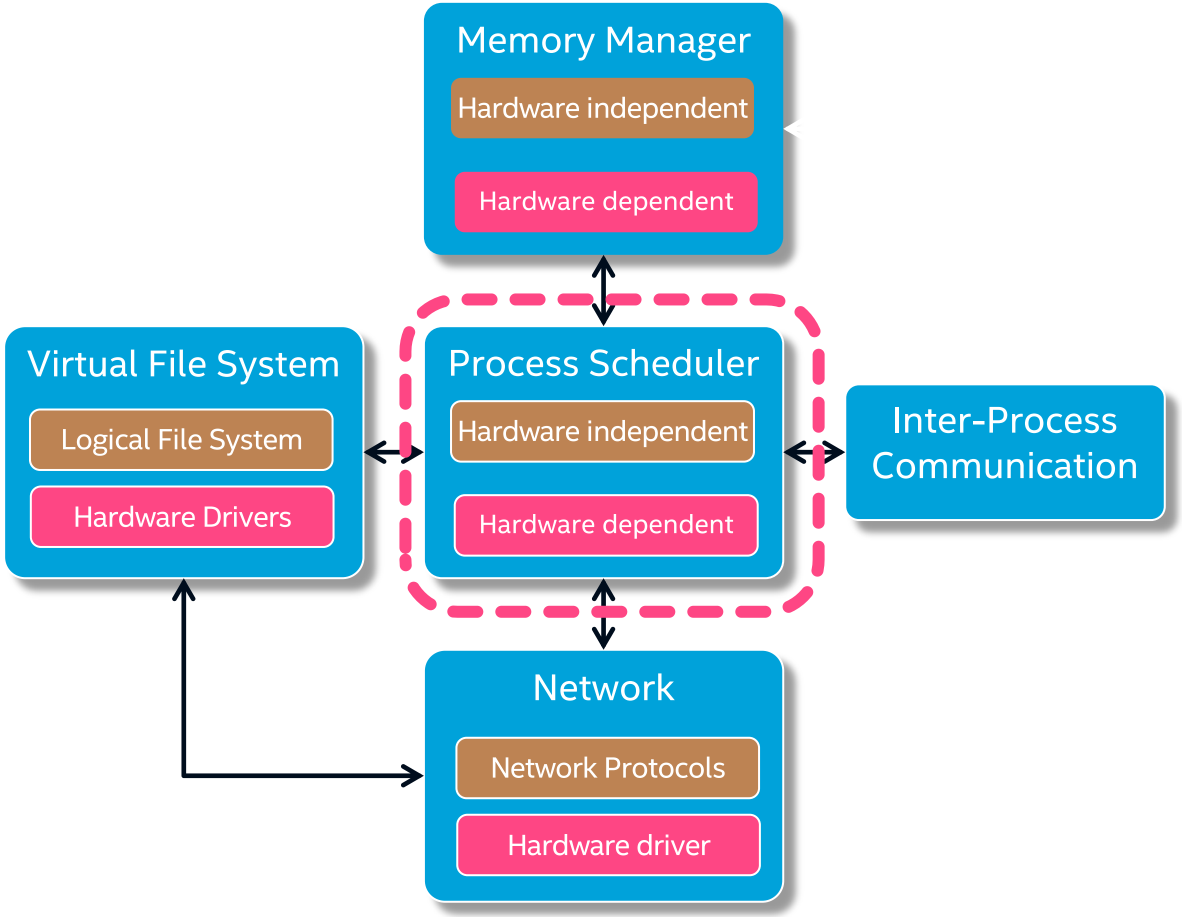
CONCEPTUAL VIEW OF THE KERNEL

- The Kernel can be divided in five subsystems:
  - Process scheduler
  - Memory manager
  - Virtual file system
  - Inter-process communication
  - Network
- Most of them are composed of:
  - Hardware-independent code
  - Hardware-dependent code



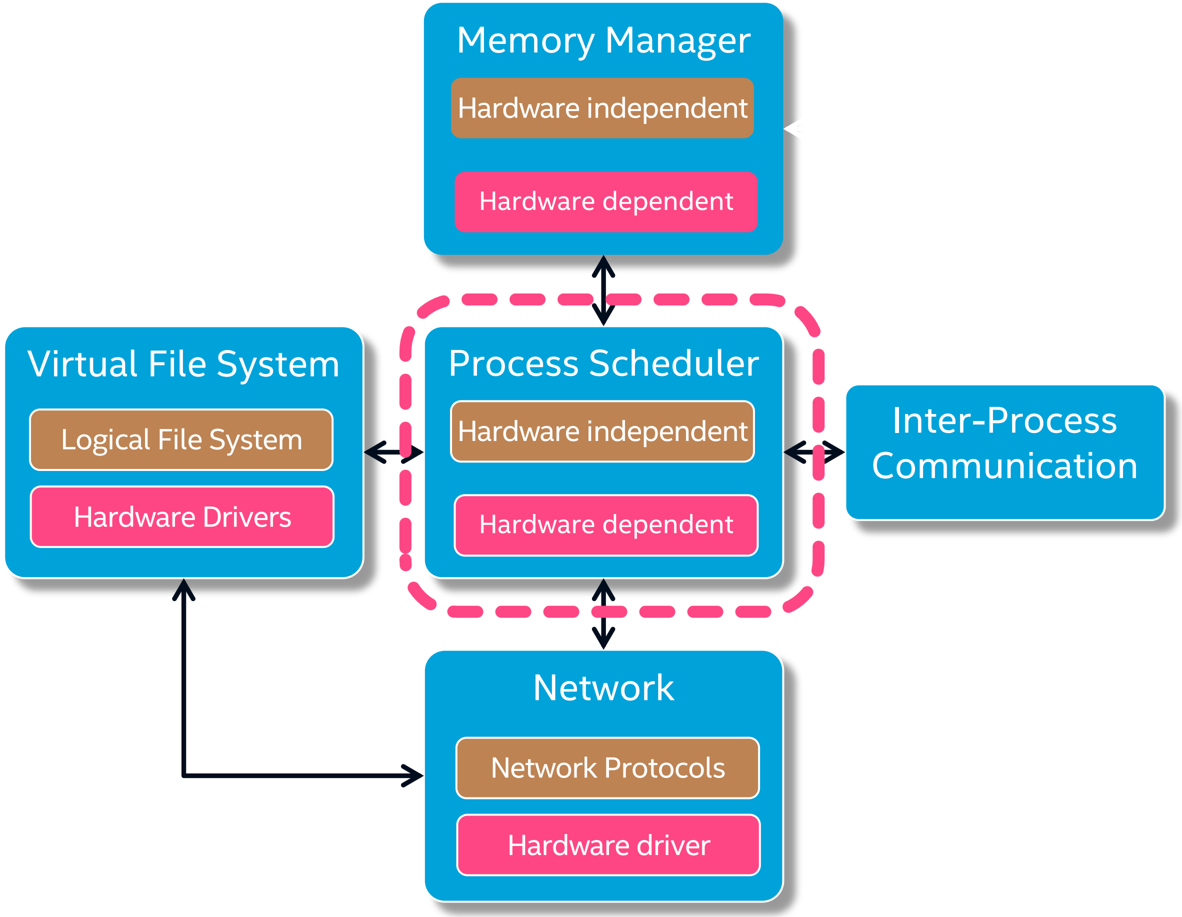
PROCESS SCHEDULER

- 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



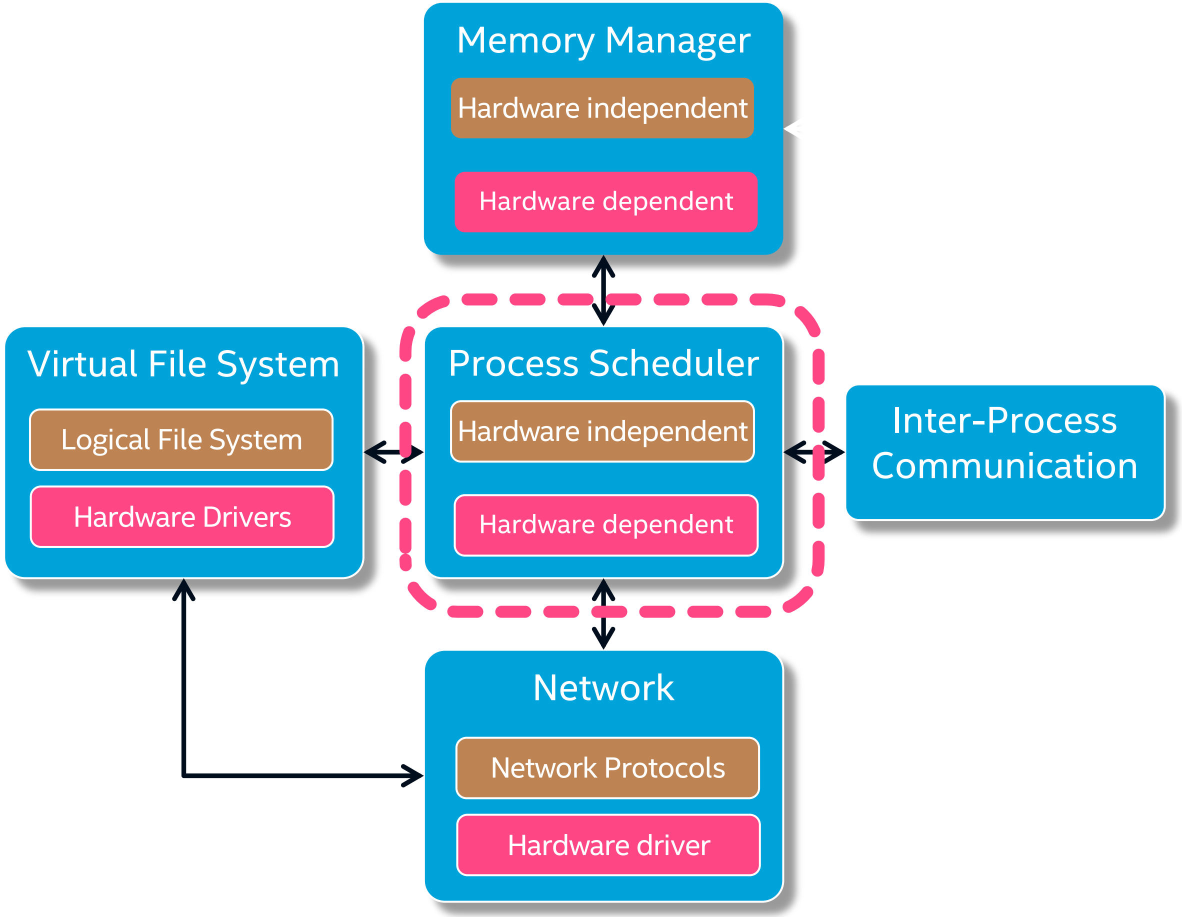
PROCESS SCHEDULER

- 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



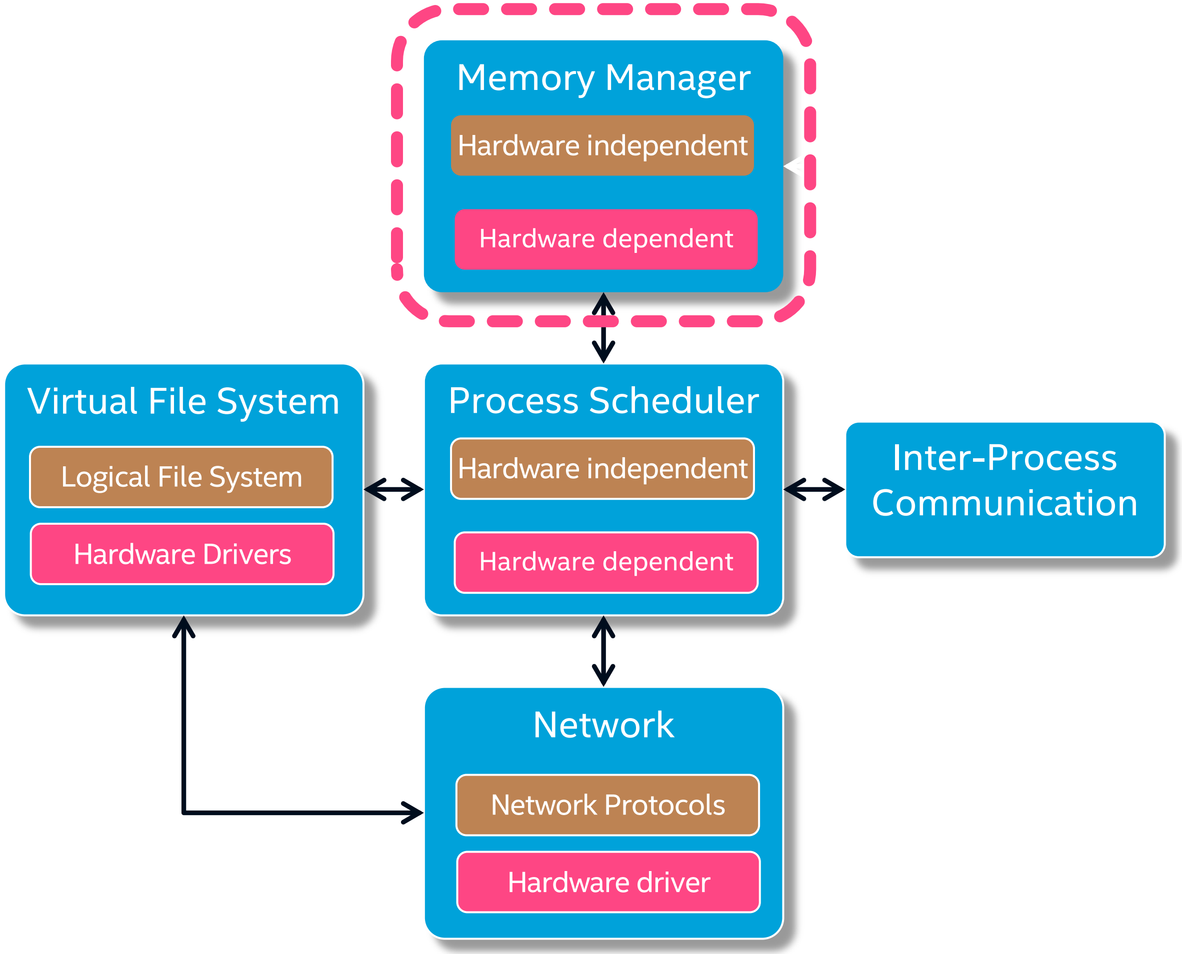
PROCESS SCHEDULER

- Interrupts:
  - Fast interrupt requests (FIQ) can be generated for events that need to be handled as they occur.
  - Standard interrupt requests (IRQ) can be generated for more general interrupt events.
  - As such – FIQs have a higher priority.
    - FIQs can interrupt code servicing an IRQ, but not vice versa



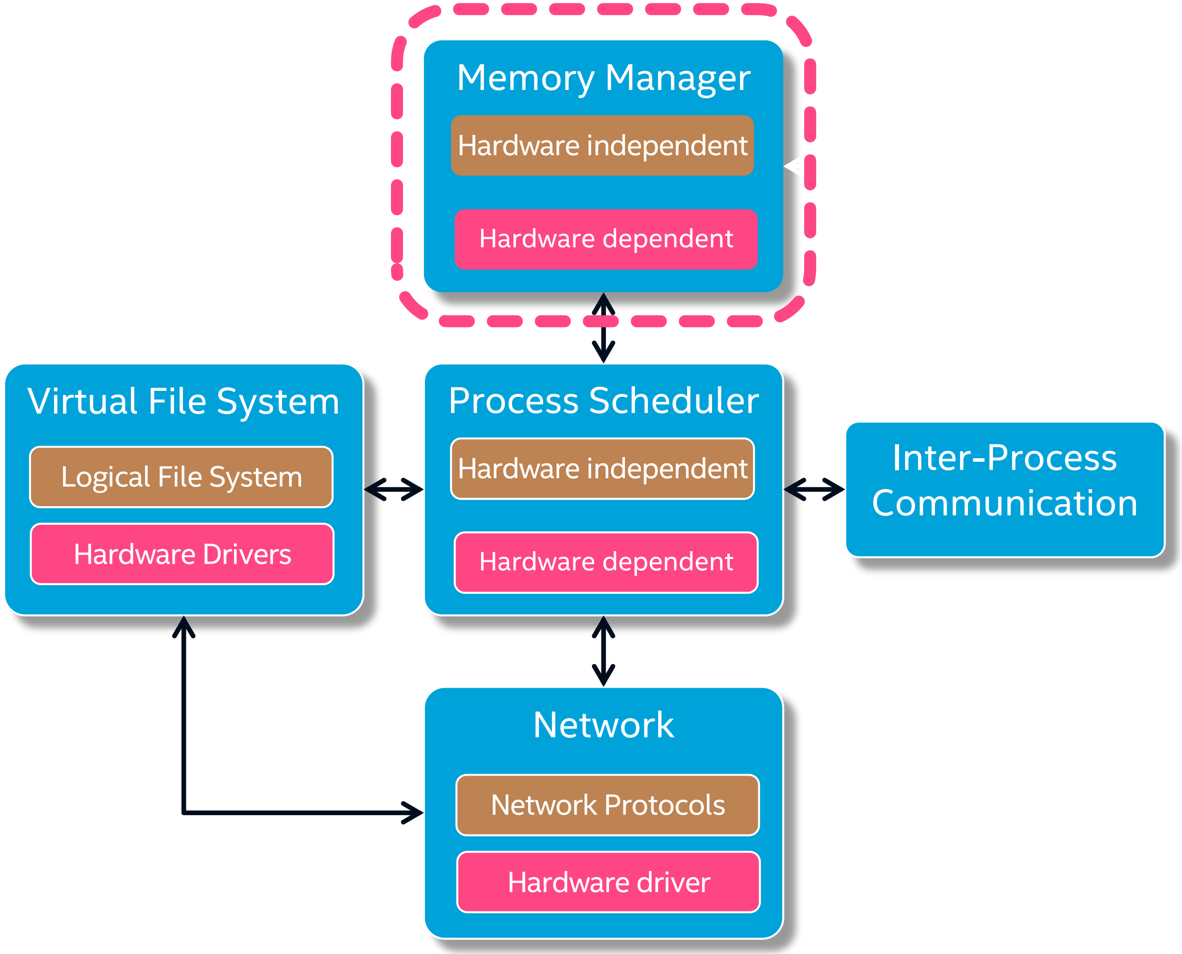
MEMORY MANAGER

- 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.



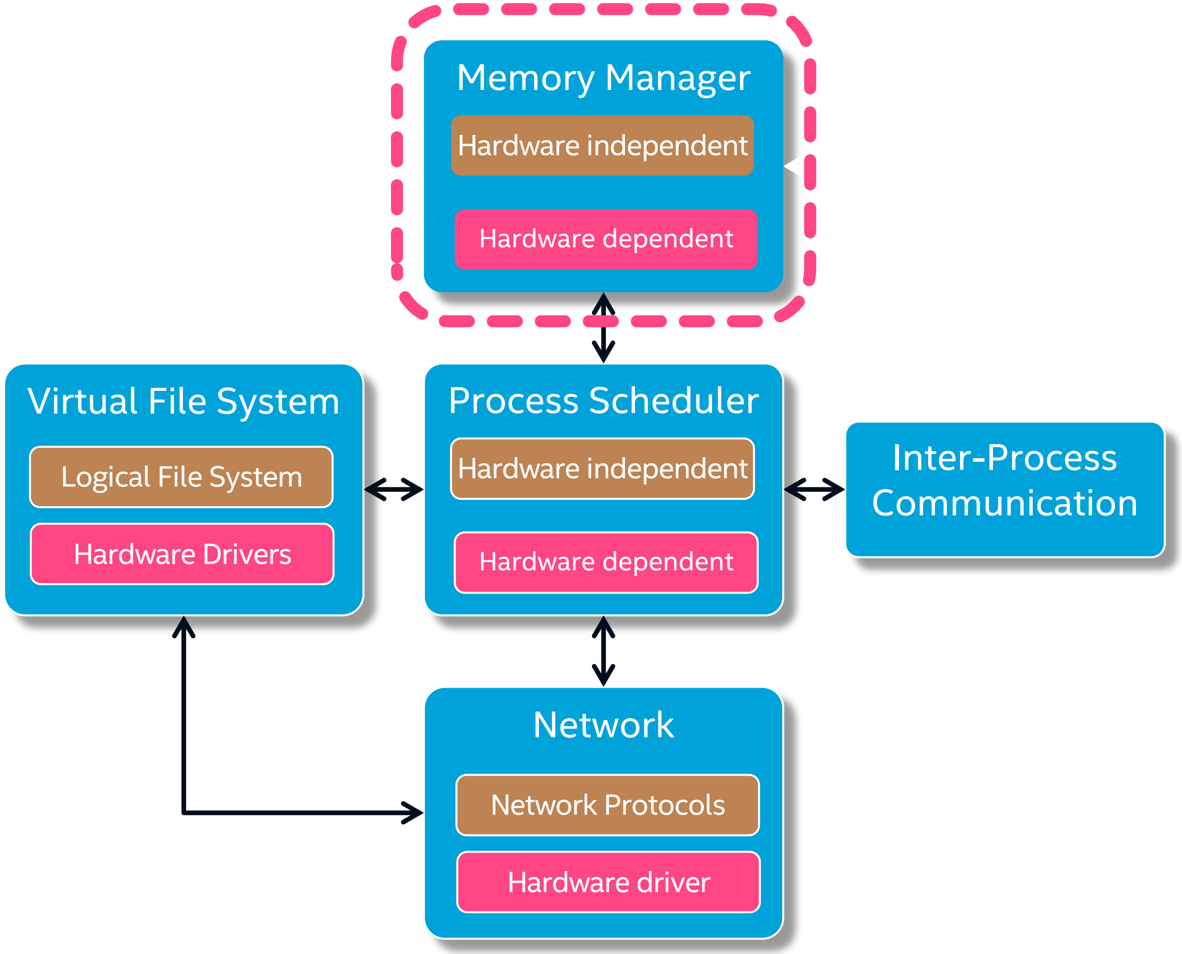
MEMORY MANAGER

- It is responsible for handling:
  - 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).



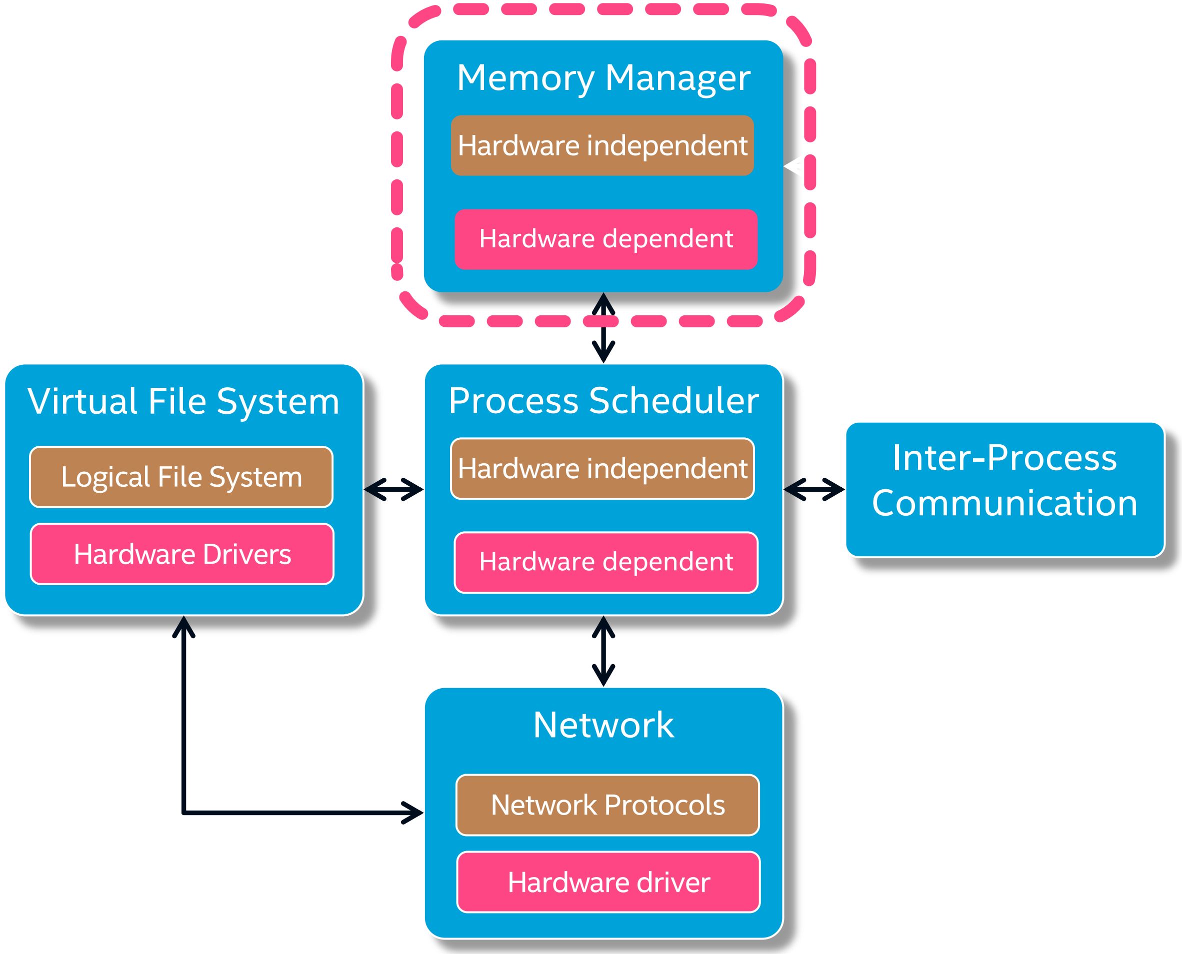
MEMORY MANAGER

- It uses the Memory Management Unit (MMU) to map virtual addresses to physical addresses.
  - It is conventional for a Linux system to have a form of MMU support.
- Advantages:
  - Processes can be moved among physical memory maintaining the same virtual addresses.
  - The same physical memory may be shared among different processes.



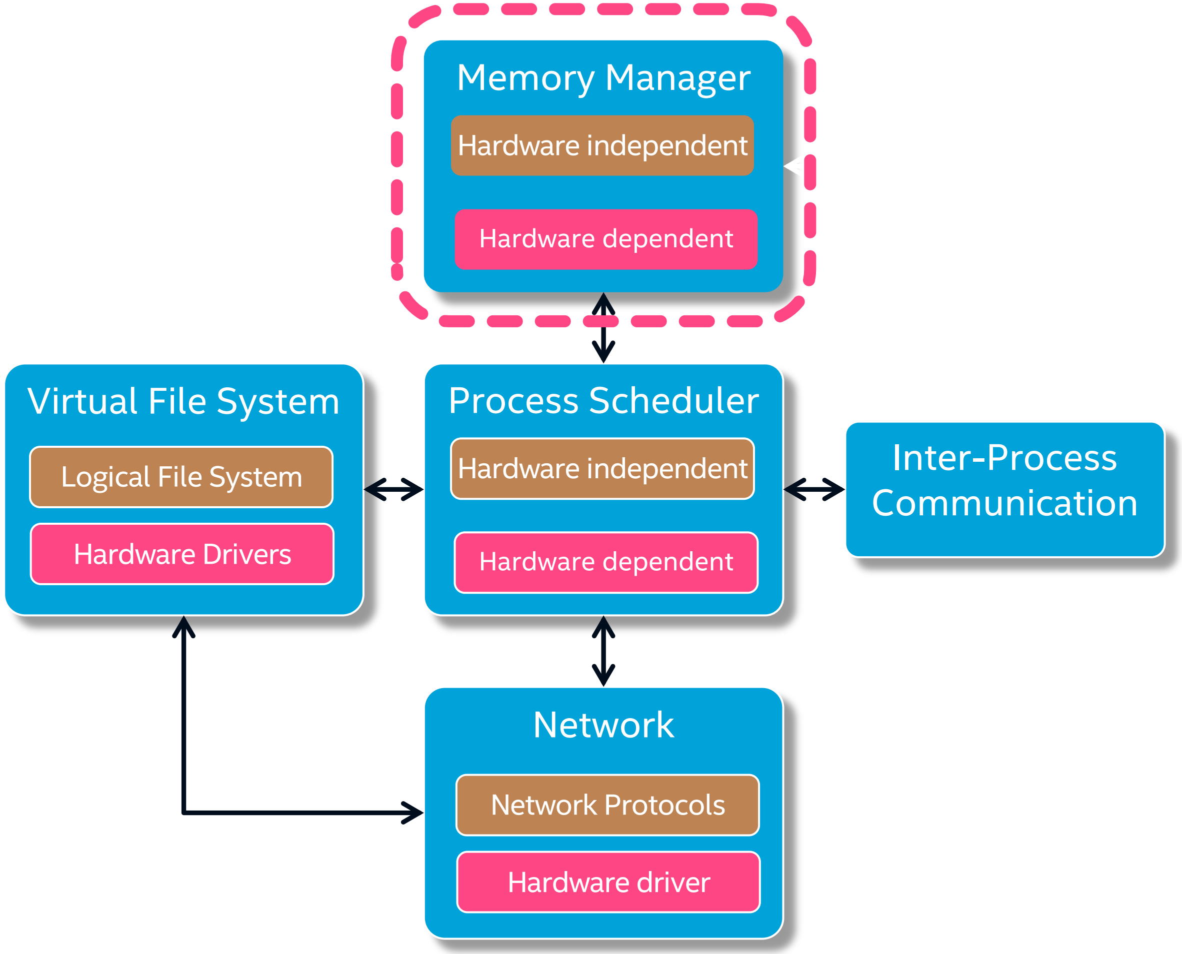
MEMORY MANAGER

- It swaps process memory out to a paging file when it is not in use:
  - Processes using more memory than physically available can be executed.
- The kswapd Kernel-space process (also known as daemon) is used for this purpose.
  - It checks if there are any physical memory pages that haven't been referenced recently .
  - These pages are evicted from physical memory and stored in a paging file.



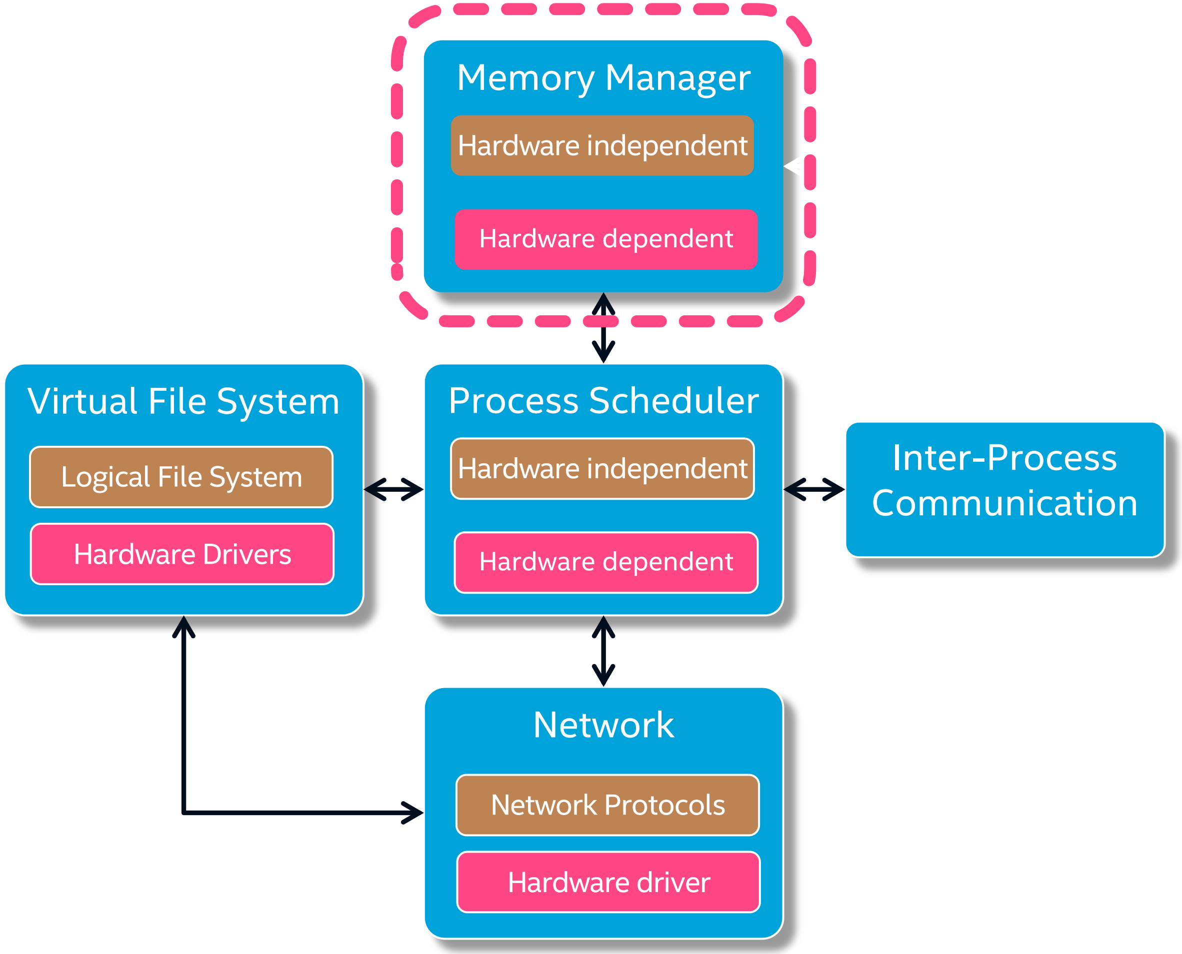
MEMORY MANAGER

- 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.



MEMORY MANAGER

- 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.

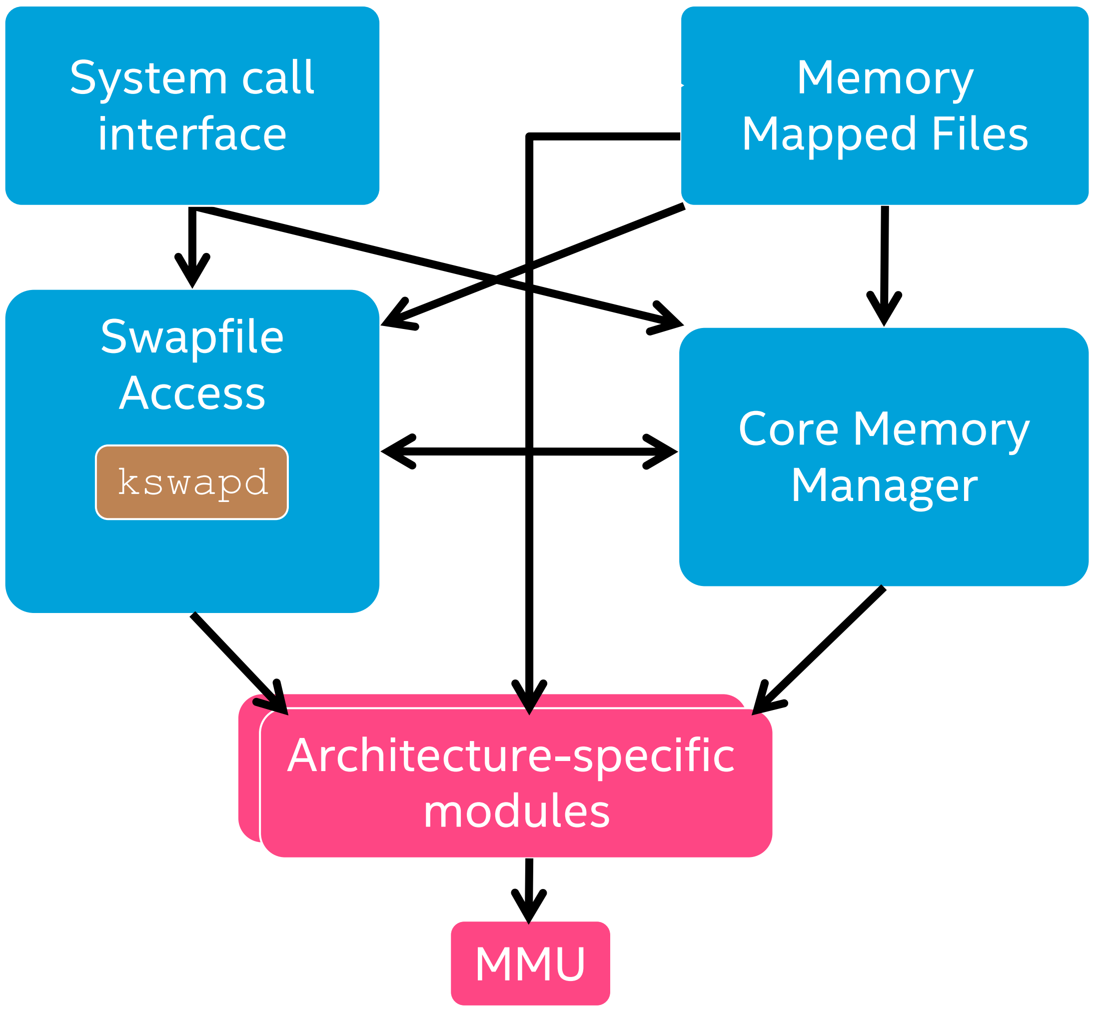


MEMORY MANAGER EXTERNAL INTERFACES

- 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

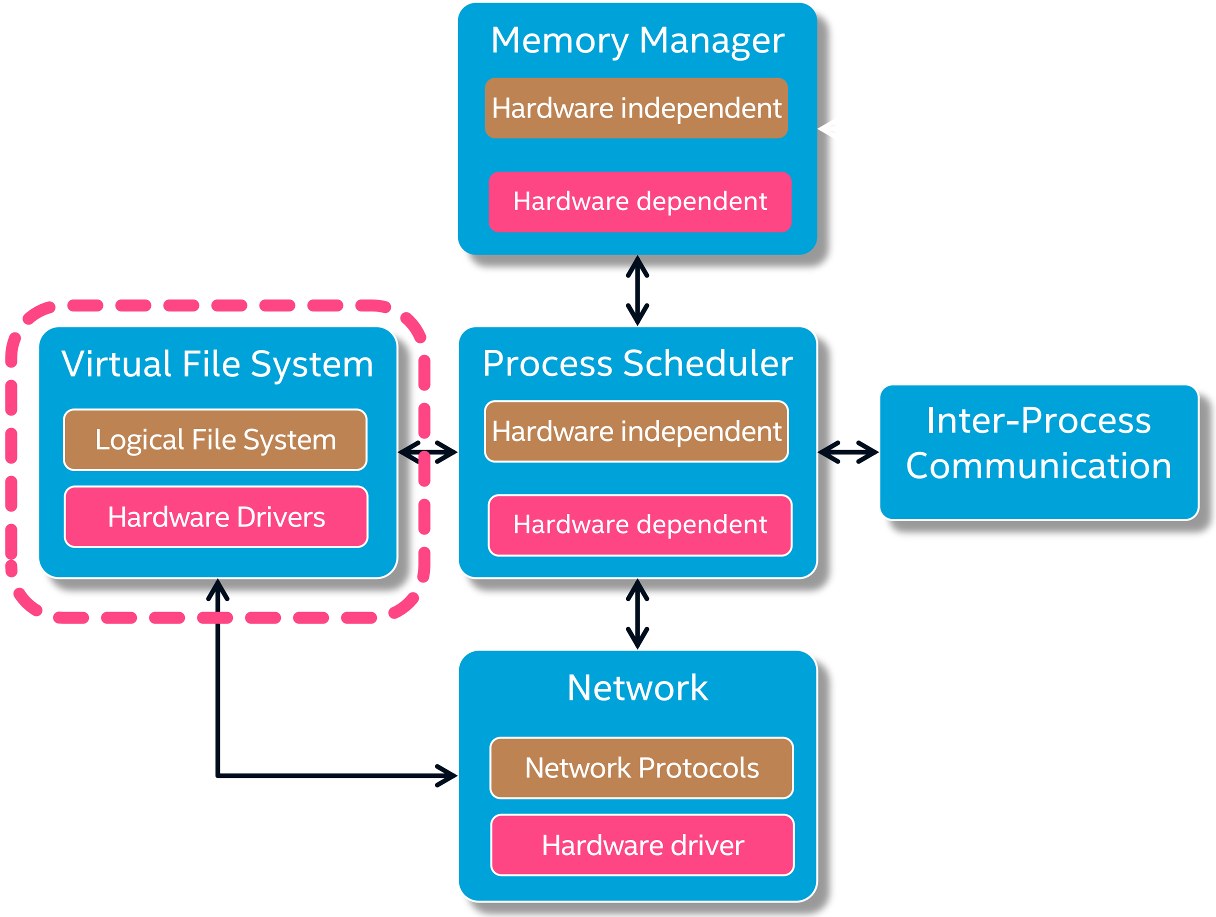
MEMORY MANAGER ARCHITECTURE

- 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).



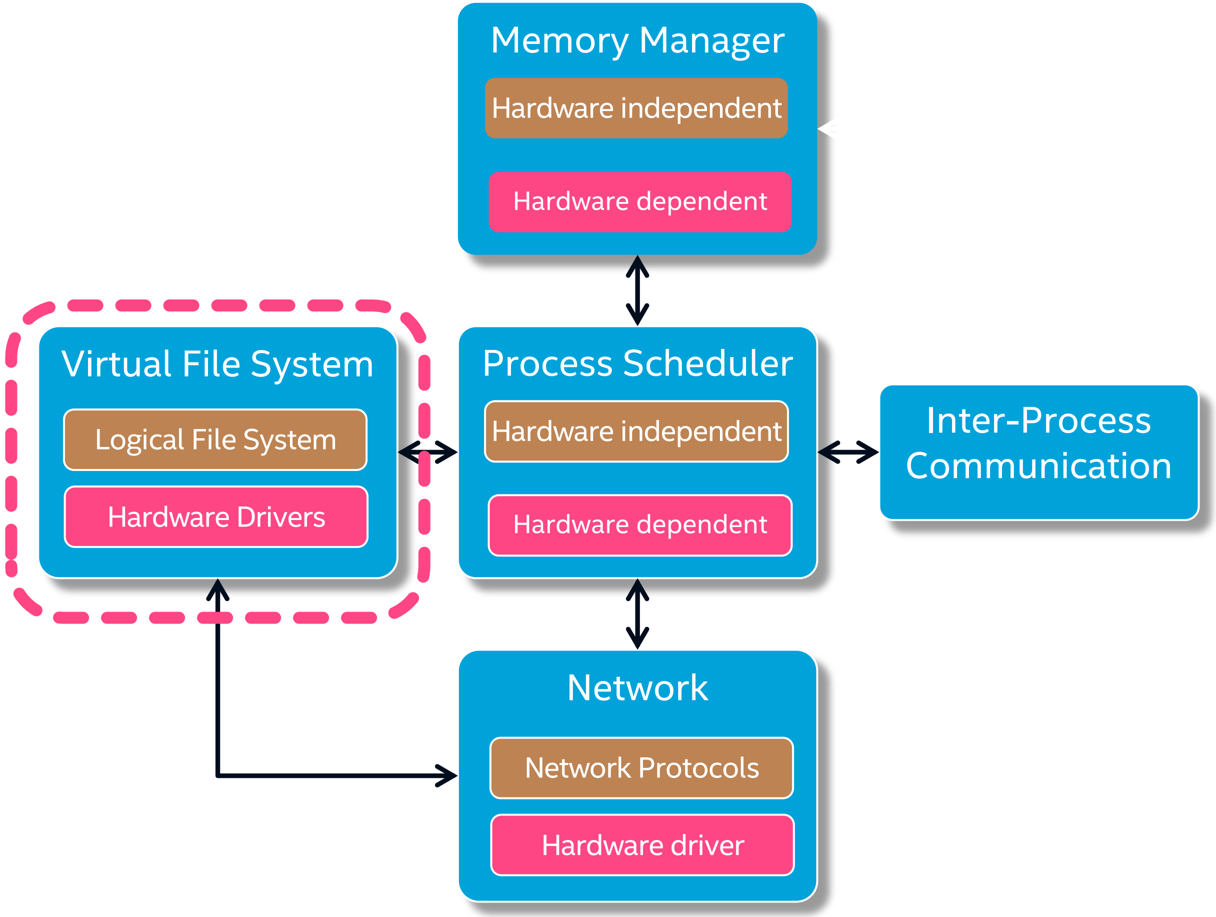
VIRTUAL FILE SYSTEM

- 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.



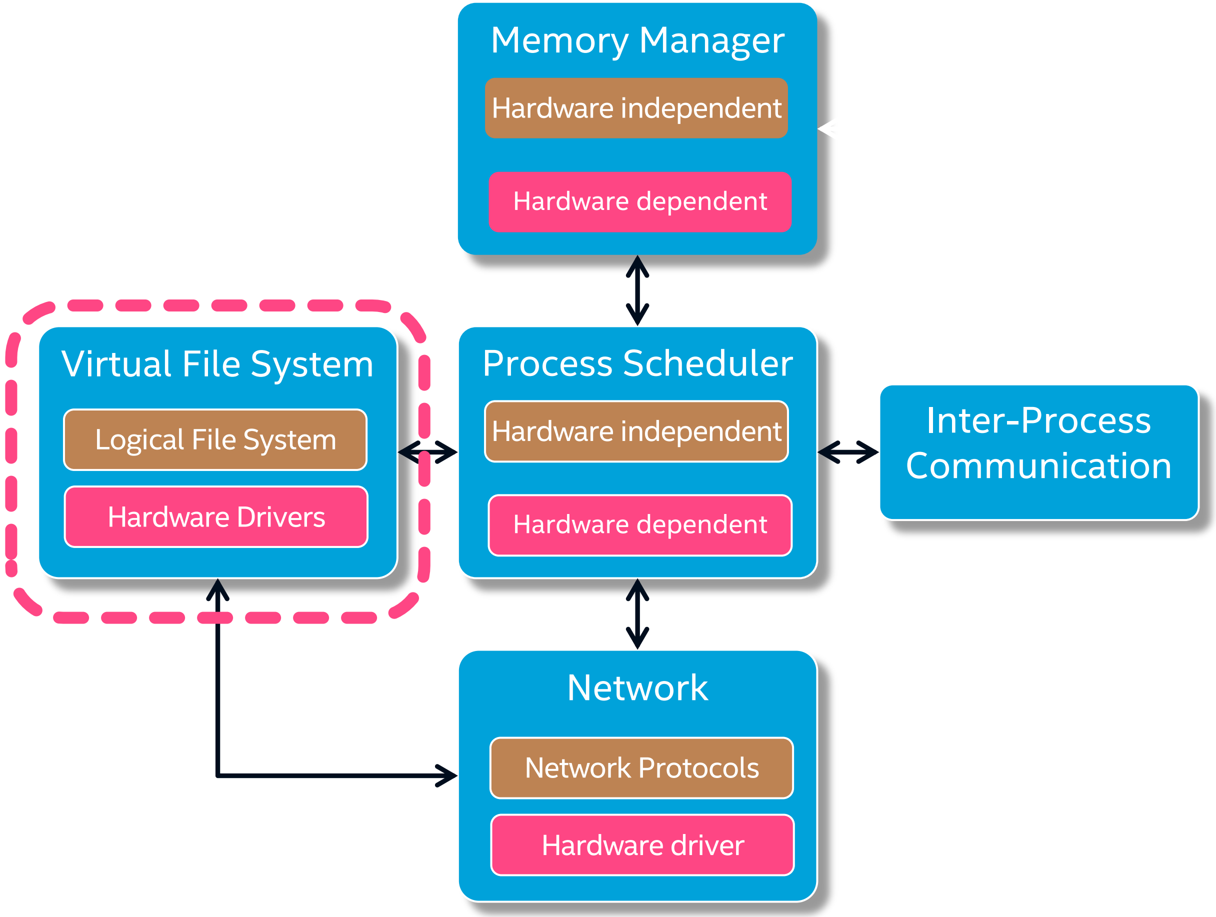
VIRTUAL FILE SYSTEM

- It is responsible for handling:
  - <u>Performance</u>: it provides high-speed access to files
  - <u>Safety</u>: it enforces policies to not lose or corrupt data
  - <u>Security</u>: it enforces policies to grant access to files only to allowed users, and it restricts user total file size with quotas.



VIRTUAL FILE SYSTEM

- 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

- It stores all the information about a file excepts its name and the data it contains.
- When a file is created, it is assigned a name and a unique i-node number (a unique integer number).
- When a file is accessed
  - Each file is associated with a unique i-node number.
  - The i-node number is then used for accessing the data structure containing the information about the file being accessed.

```
struct inode {
  struct hlist_node      i_hash;
  struct list_head       i_list;
  struct list_head       i_sb_list;
  struct list_head       i_dentry;
  unsigned long          i_ino;
  atomic_t               i_count;
  umode_t                i_mode;
  unsigned int           i_nlink;
  uid_t                  i_uid;
  gid_t                  i_gid;
  dev_t                  i_rdev;
  loff_t                 i_size;
  struct timespec        i_atime;
  struct timespec        i_mtime;
  struct timespec        i_ctime;
    ...
```

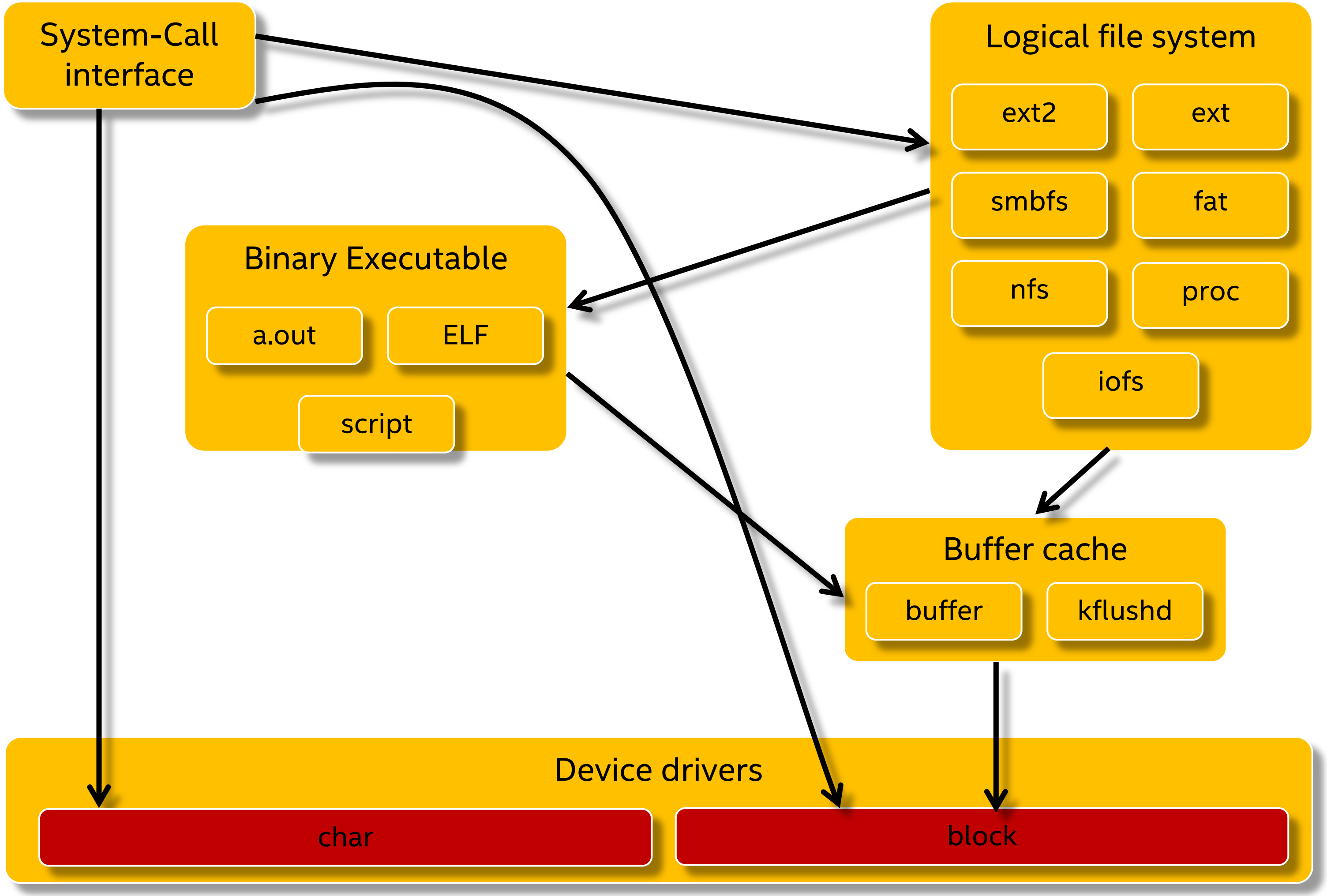
I-NODE INTERFACE

- 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



VIRTUAL FILE SYSTEM ARCHITECTURE

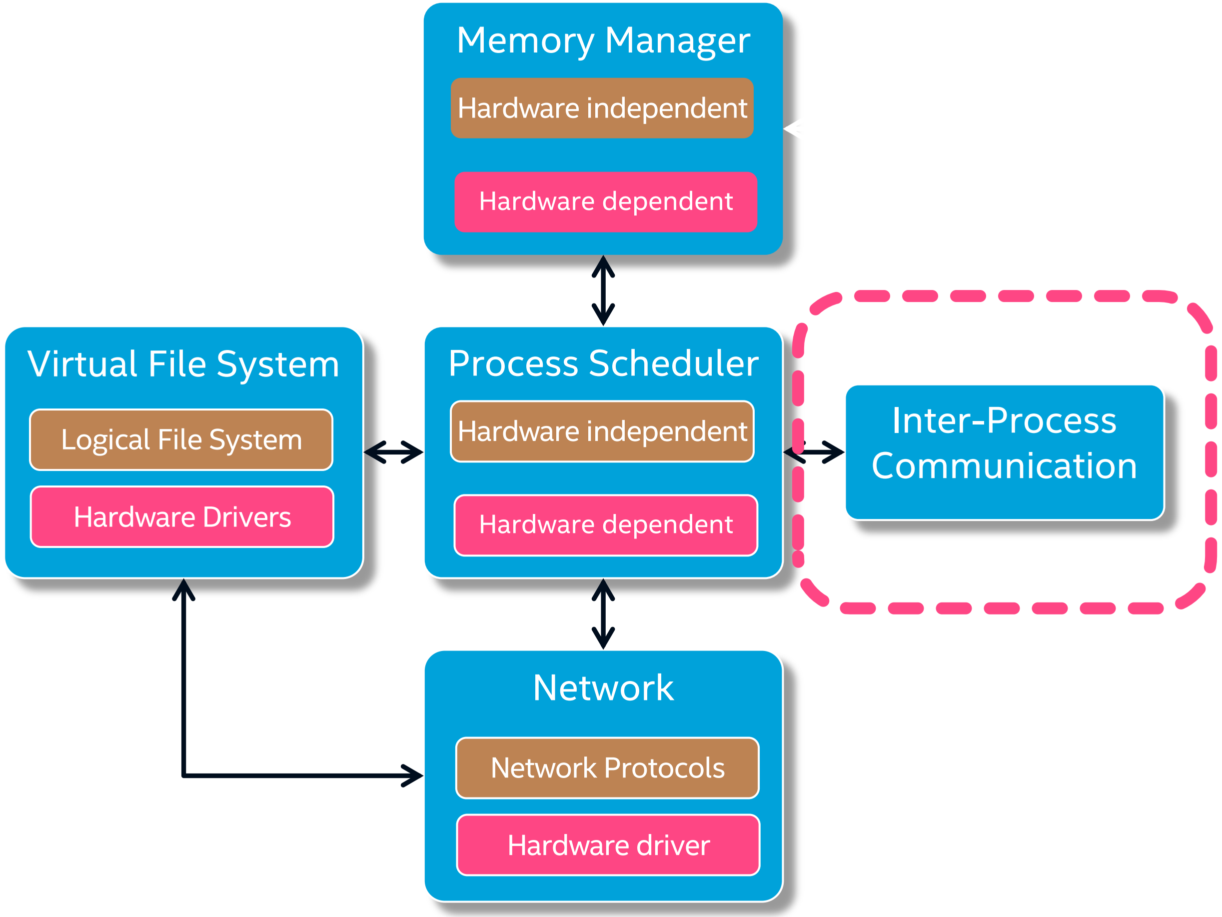
- System call interface: it provides Virtual File System services to the user space
- Logical file system: it provides a logical structure for the information stored in a storage medium.
  - Several logical file systems are supported (e.g. ext2, fat).
  - All files appear the same to the user.
  - The i-node is used to hide logical file system details.
  - For each file, the corresponding logical file system type is stored in the i-node.
  - Depending on the information in the i-node, the proper operations are activated when reading/writing a file in a given logical file system.
- Buffer cache: it provides data caching mechanisms to improve performance of storage media access operations.
- Binary executable: it supports different types of executable files transparently to the user.

VIRTUAL FILE SYSTEM ARCHITECTURE

- 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).
- Device drivers use the file interface abstraction:
  - Each device can be accessed as a file in the file system through a special file, the device file, associated with it.
  - A new device driver is a new implementing of the hardware-specific code to customize the file interface abstraction (more about this later).

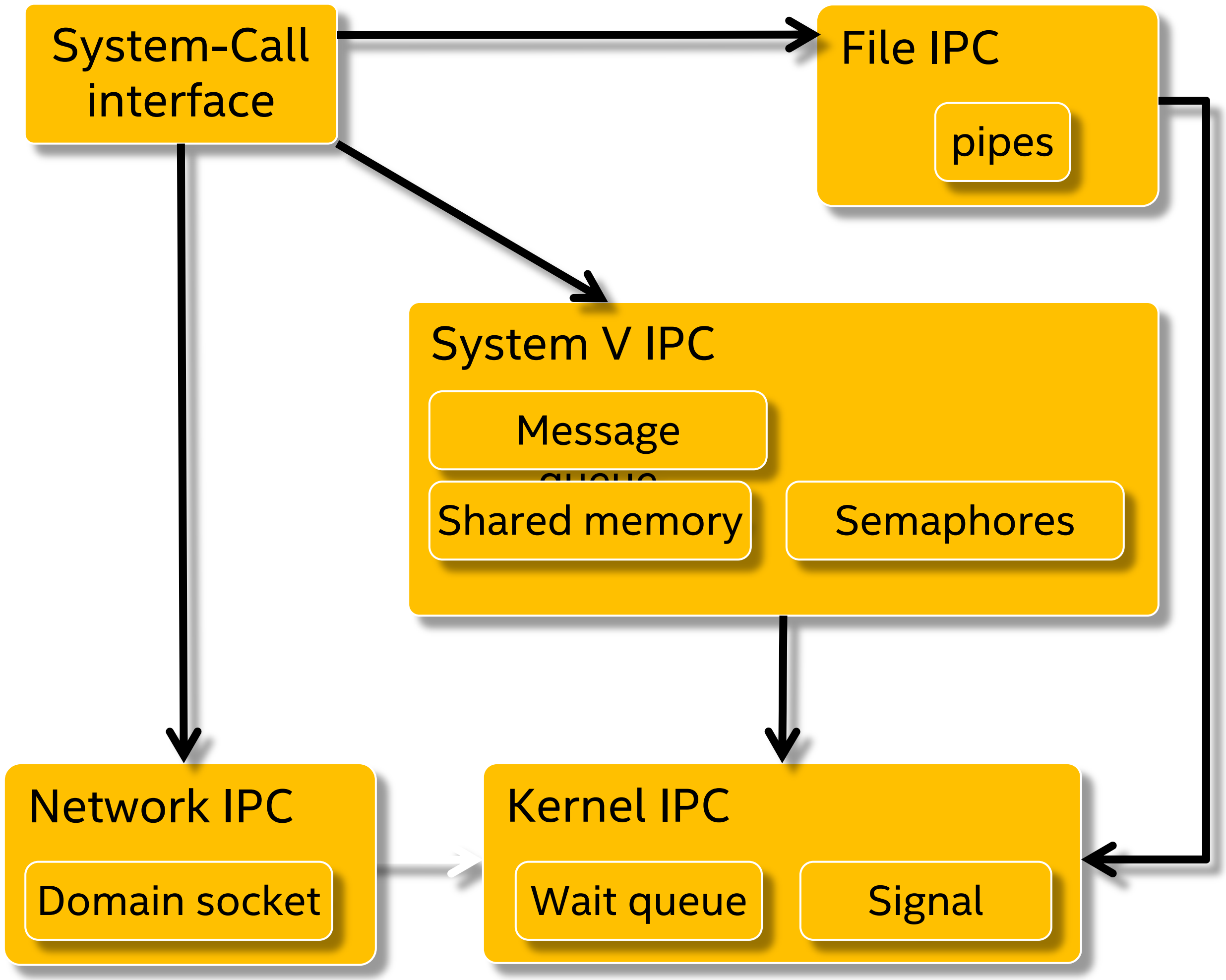
INTER-PROCESS COMMUNICATION

- It provides mechanisms to processes for allowing:
  - Resource sharing
  - Synchronization
  - Data exchange



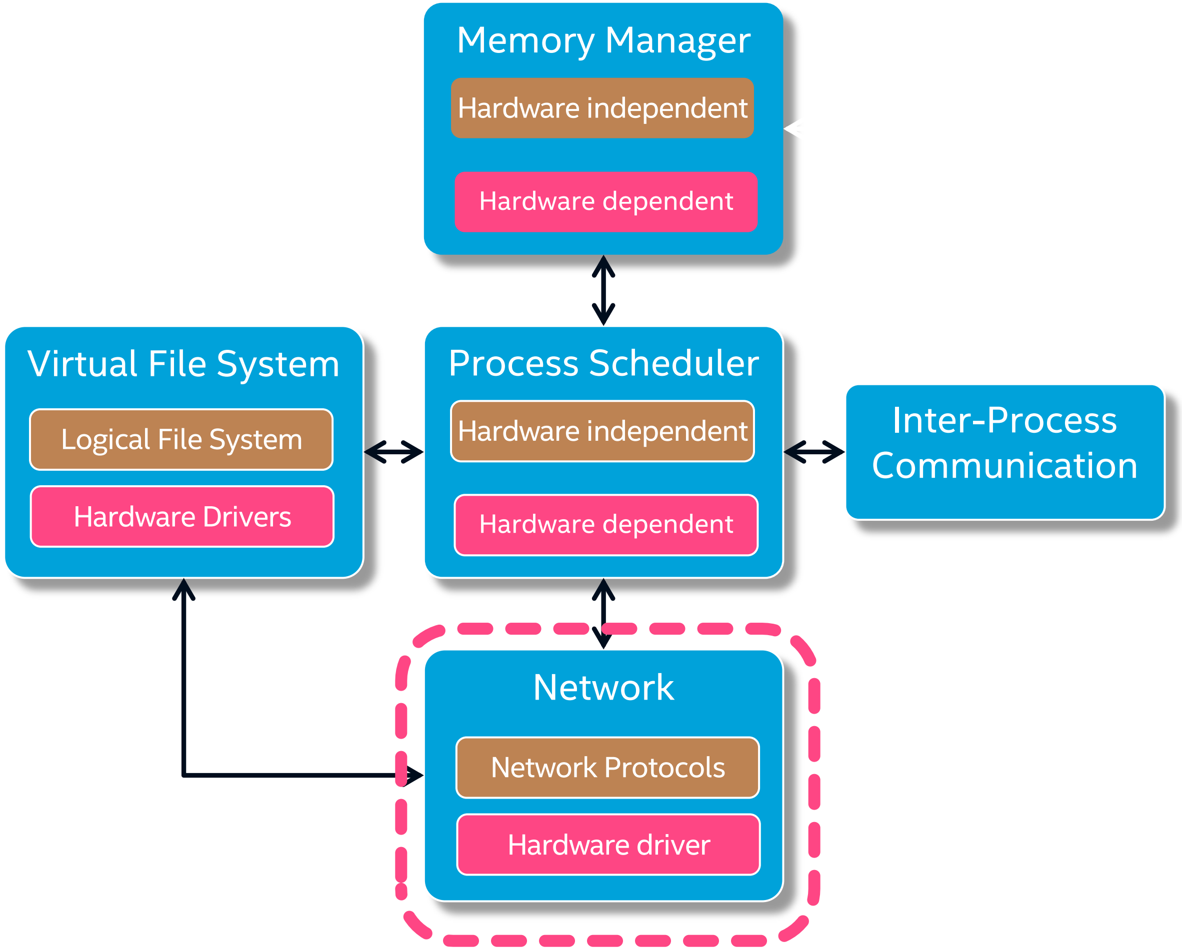
INTER-PROCESS COMMUNICATION ARCHITECTURE

- System call interface: it provides inter-process communication (IPC) services to the user space
- The following IPCs are supported:
  - Pipes
  - Message queues
  - Shared memory
  - Semaphores
  - Domain sockets
  - Wait queues
  - Signals



NETWORK

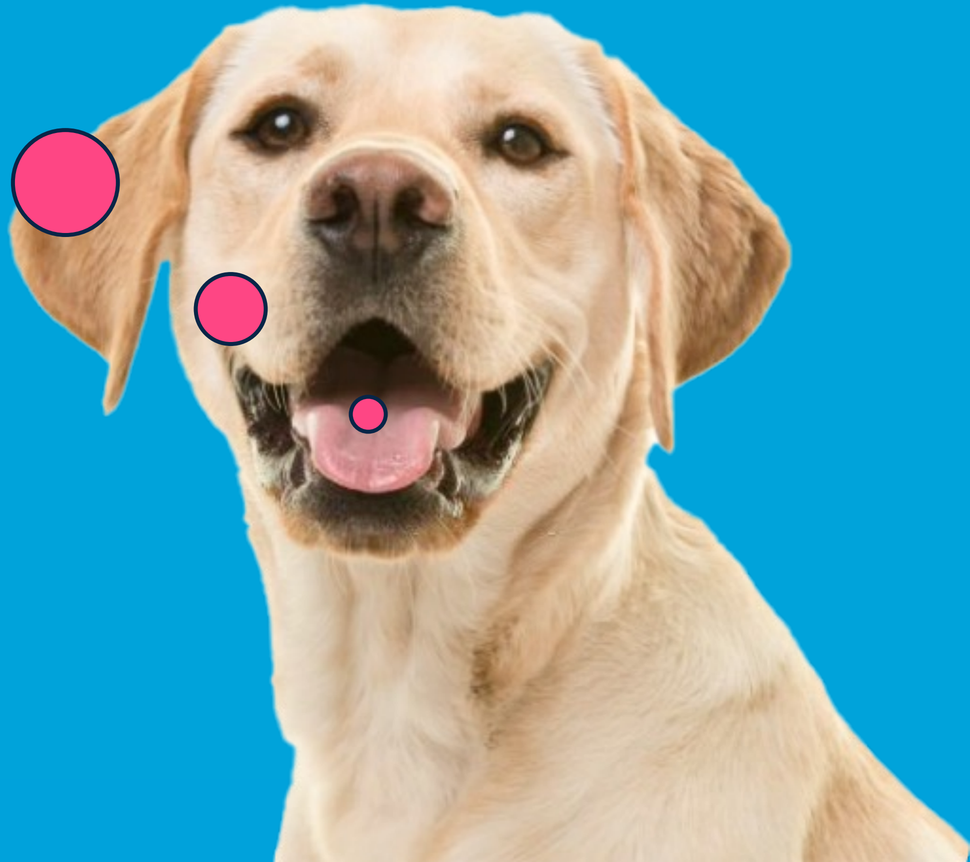
- Provides support for network connectivity
  - It implements network protocols (e.g. TCP/IP) through hardware-independent code.
  - It implements network card drivers through hardware-specific code.



smilies
reSilient coMputer archItectures and LIfE Sciences

Politecnico di Torino
1859
Department of Control and Computer Engineering
DAUIN

QUESTIONS?



THANK YOU!