Fundamental Concepts of Operating Systems

Minsoo Ryu

Department of Computer Science and Engineering
Hanyang University

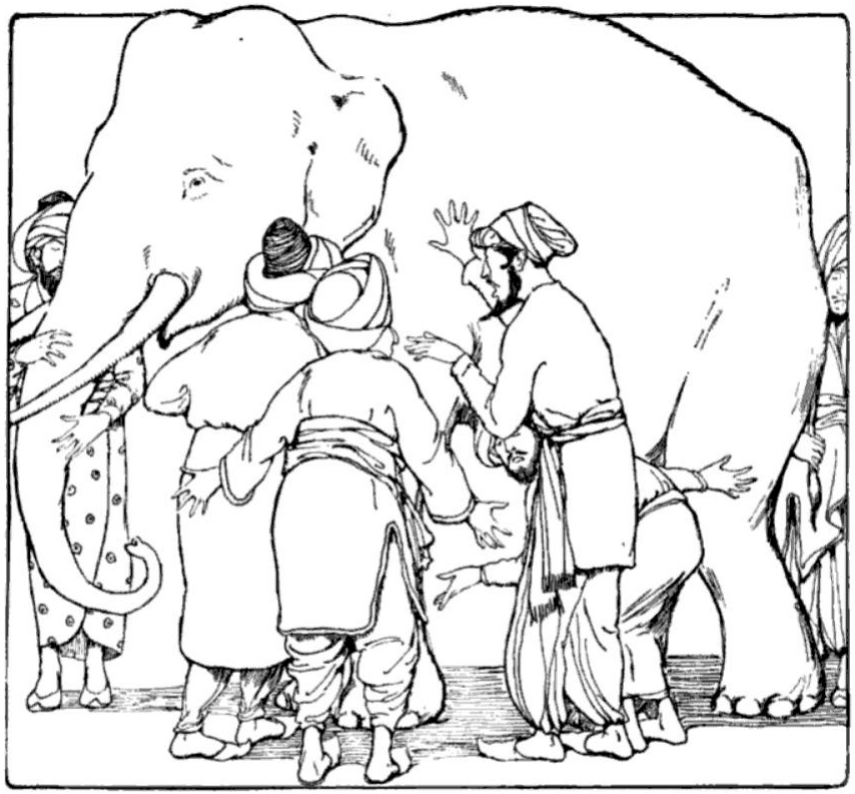
Outline

OS, What and Why?

What is the OS?

App developers
(set of APIs and services)

OS developers
(?)



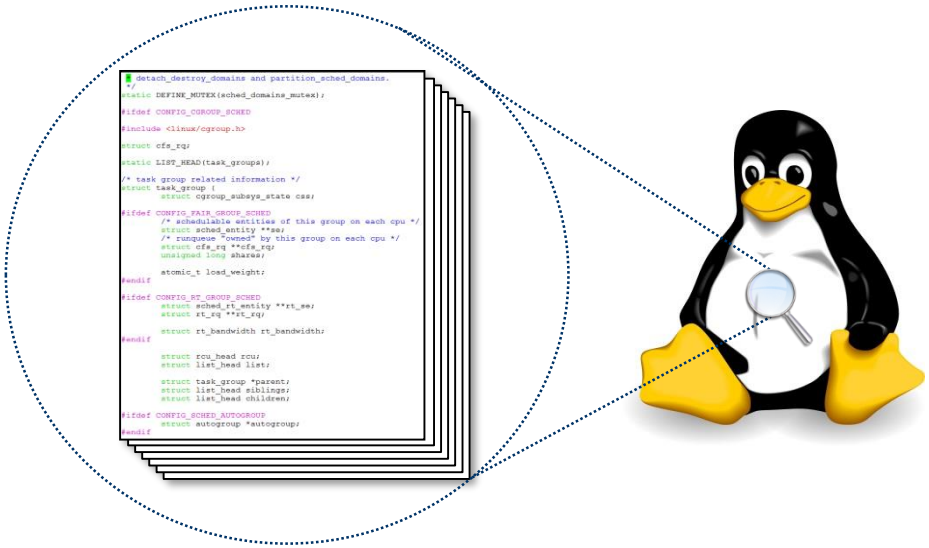
Users
(GUI and windows, file systems)

Device Manufacturers
(device drivers)

CPU, I/O hardware vendors
(responsible for hardware management)

Huge and Complex

- Red Hat Linux version 7.1 (released April 2001) contained over 30 million lines
  - About 8,000 man-years of development effort
  - Over $1 billion (in year 2000 U.S. dollars)



Linux has 18M lines of code

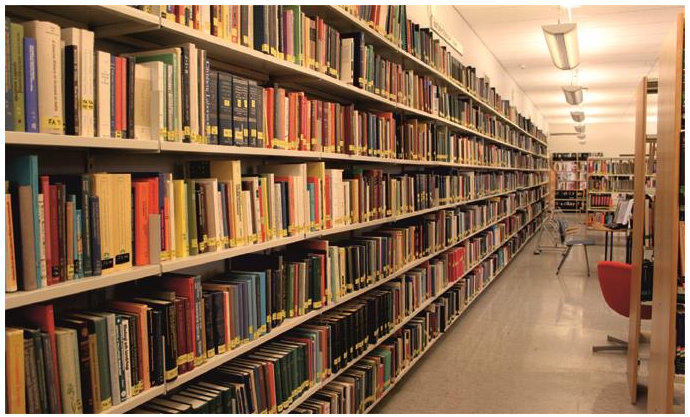
45M lines of code

Difficult to Build and Own

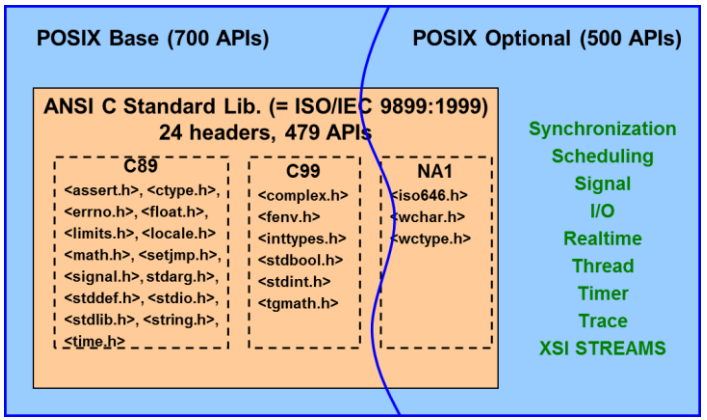
- OS interfaces a variety of apps and HW components

Simply, OS is a Big Library

- An organized collection of reusable program code
  - Bootstrapping, device drivers, scheduling, GUI …
  - POSIX Standard API implementations

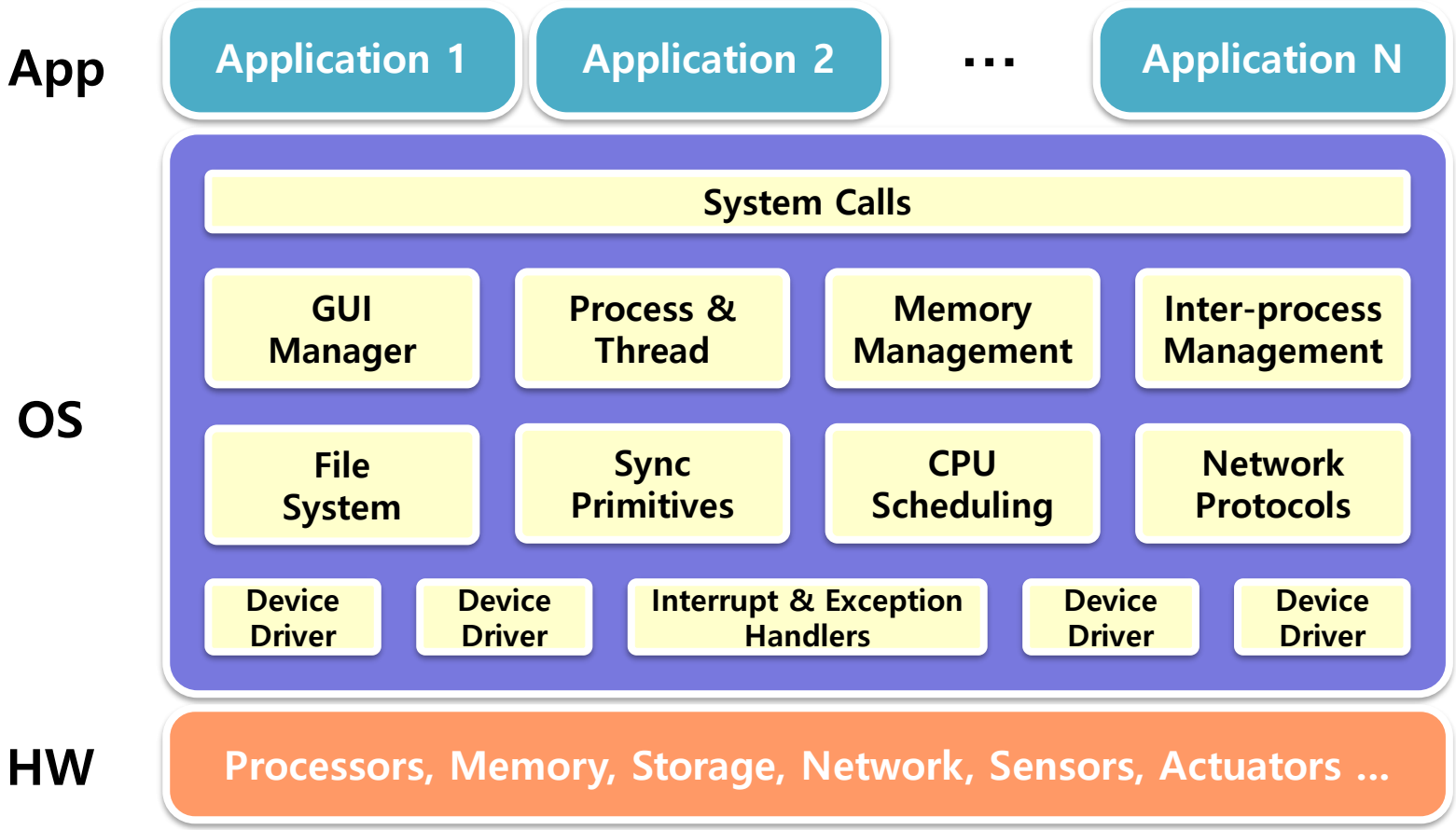


University Library

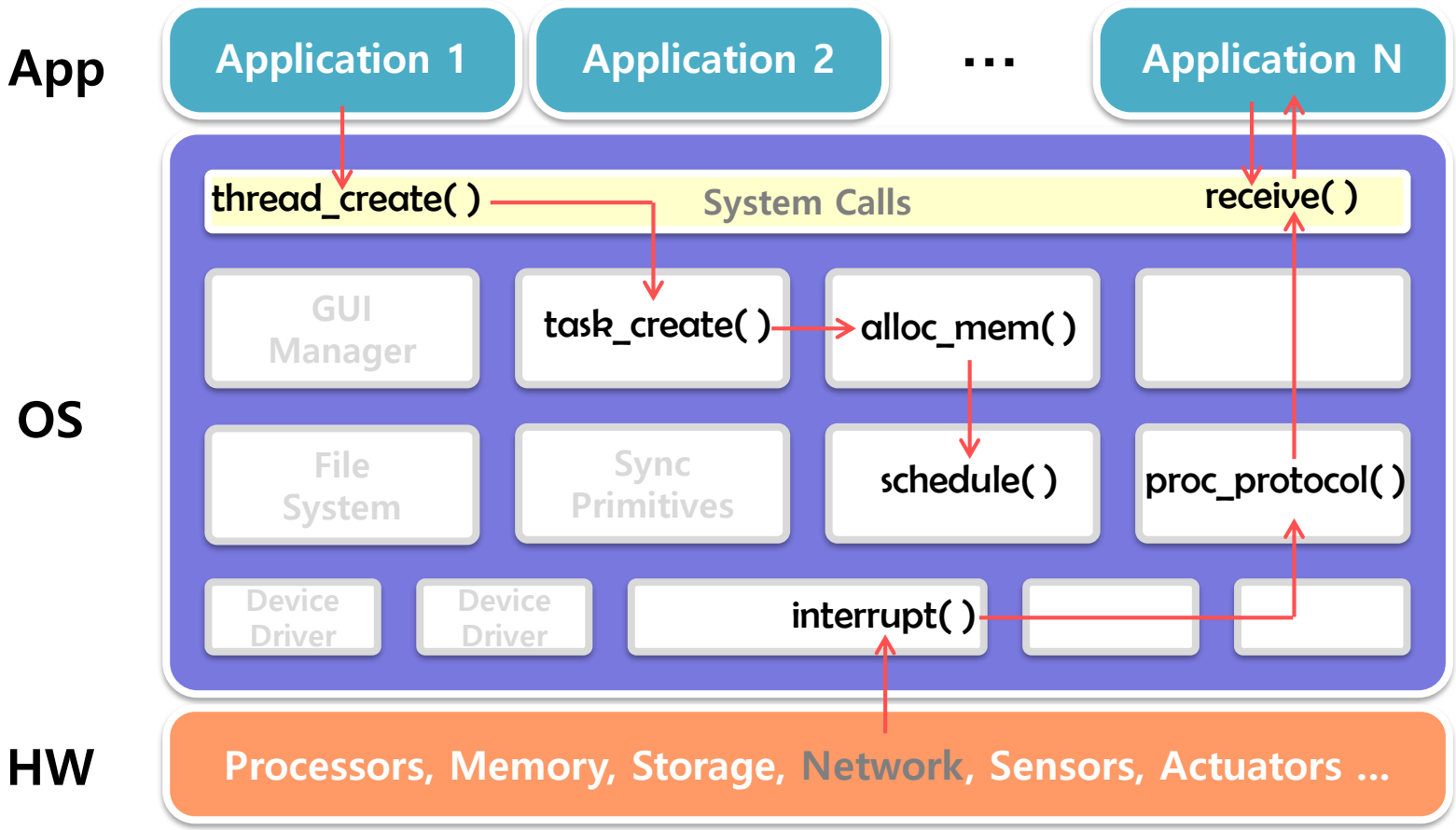


POSIX Standard 1003.1

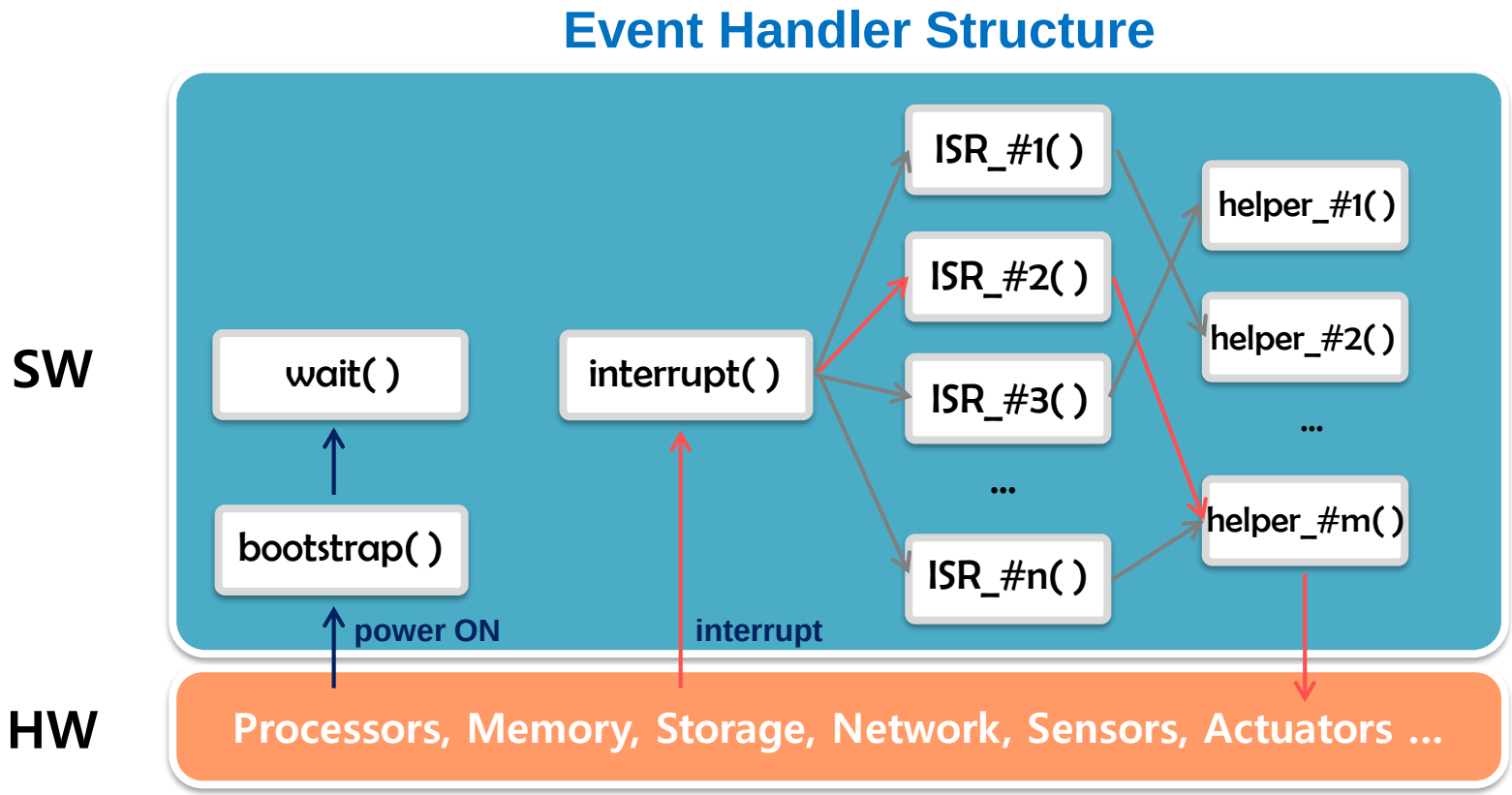
Common OS Structure



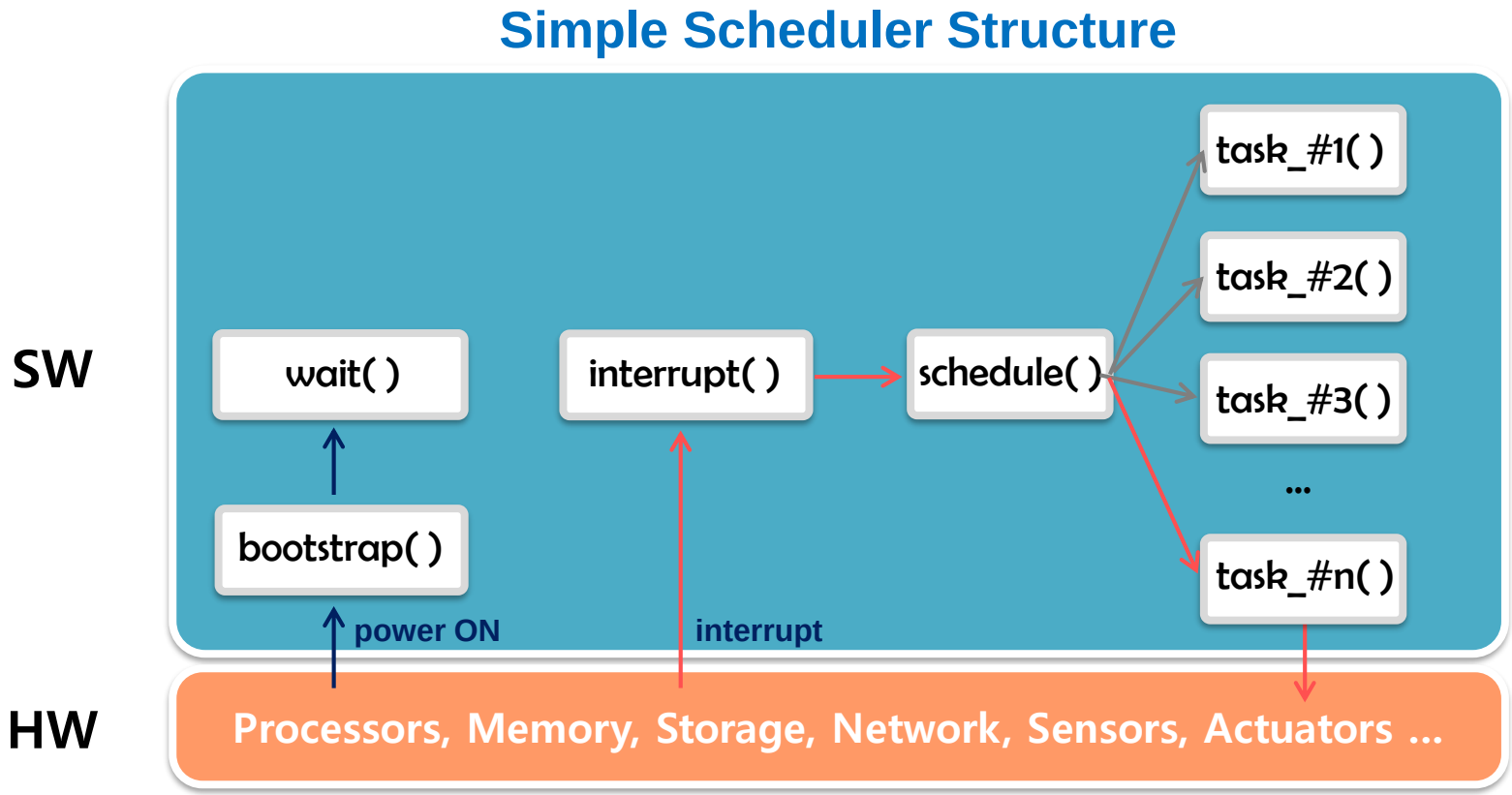
Execution Flow with OS



Execution Flow without OS



Execution Flow without OS

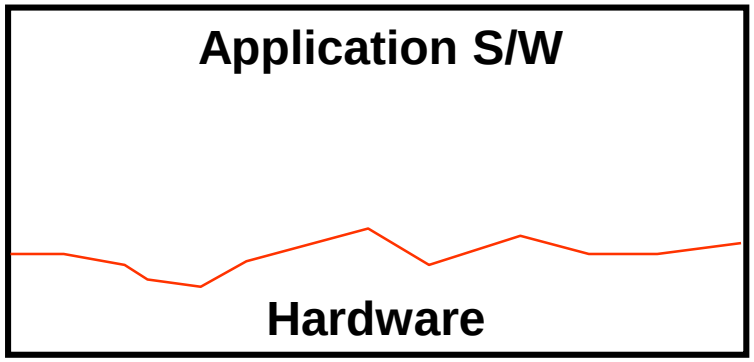


OS is more than a library

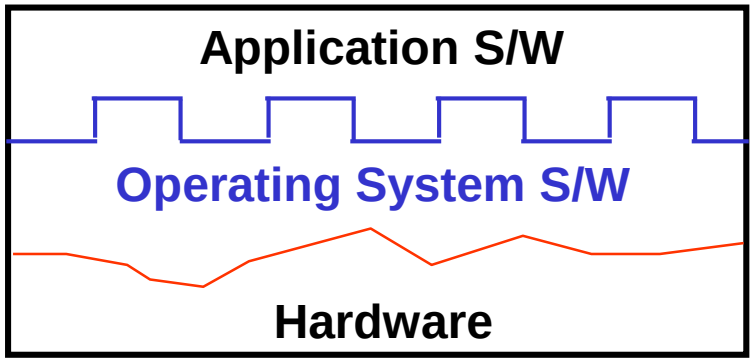
1. Abstraction
2. Protection
3. Illusion

Abstraction

- Pick out only key features to reduce complexity
- Provide a clean, uniform, and standard interface to users hiding hardware complexity
  - POSIX Standard APIs



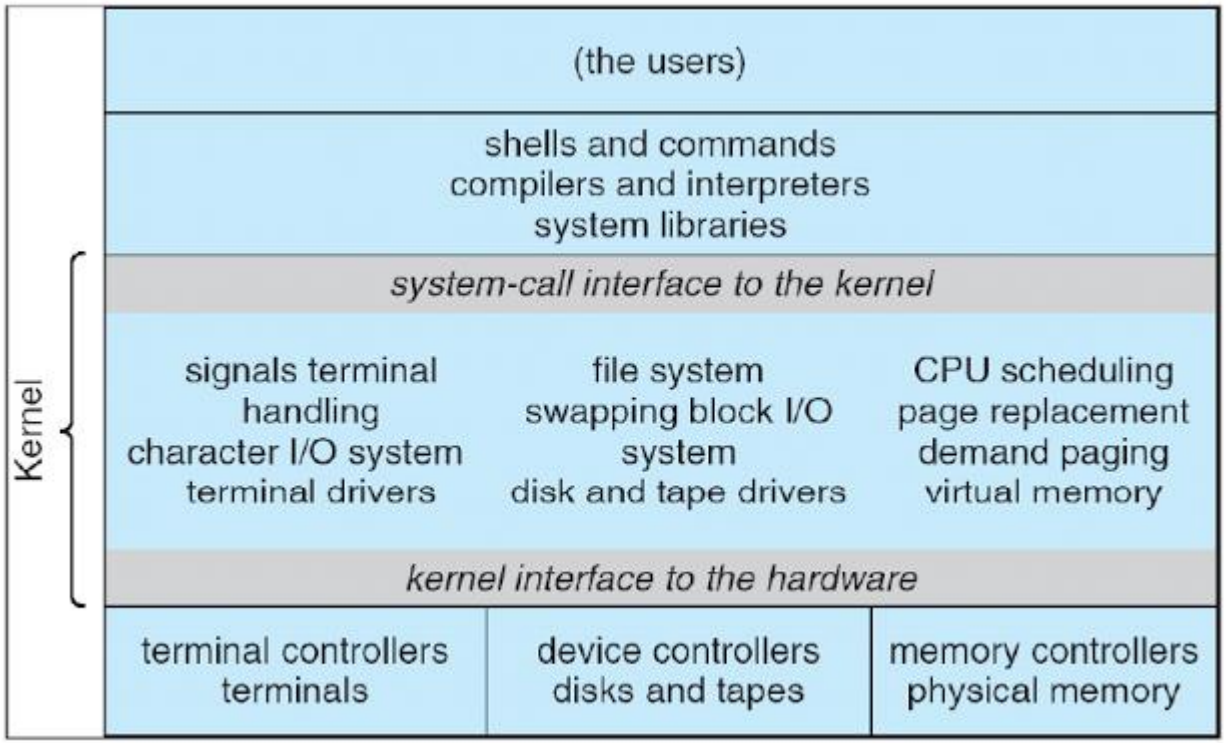
Without OS



With OS

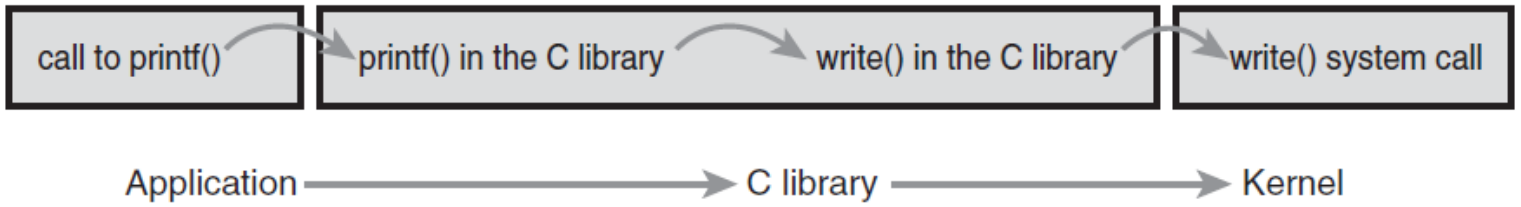
System Call Interface

- System calls provide the interface between application SW and the operating system



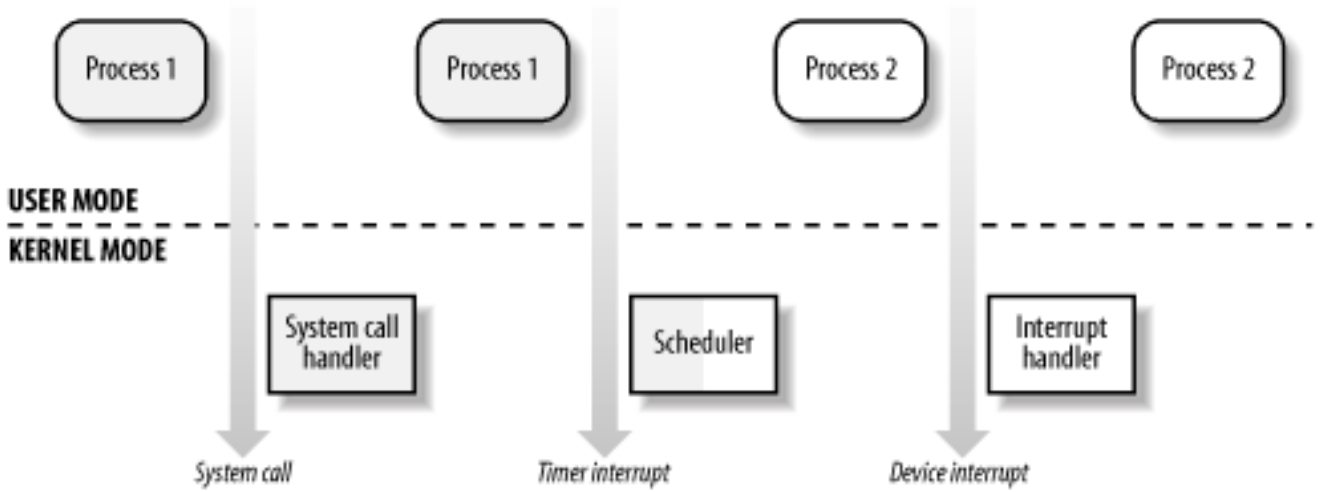
System Calls vs. Library Calls

System calls	Library calls
open	fopen
close	fclose
read	fread, getchar, scanf, fscanf, getc, fgetc, gets, fgets
write	fwrite, putchar, printf, fprintf putc, fputc, puts, fputs
lseek	fseek



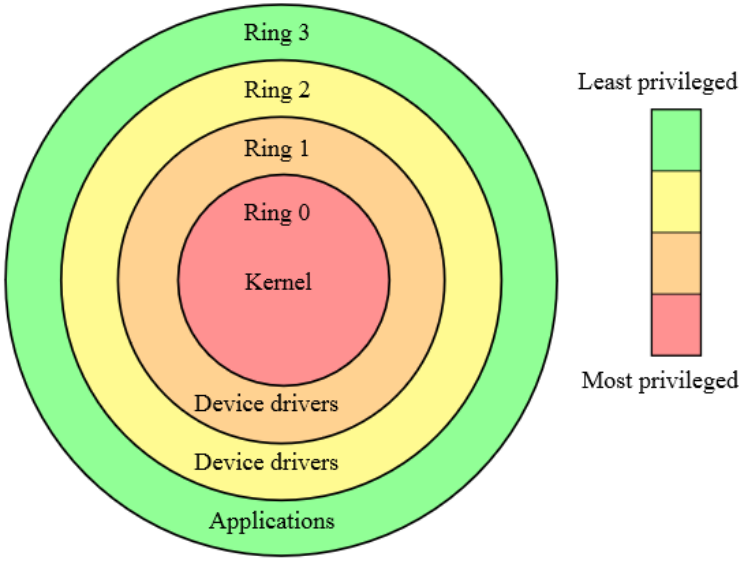
Protection

- **OS has a dual mode mechanism to protect the system from many types of application faults**
  - **User mode: application code runs**
    - Using only non-privileged instructions and/or registers
  - **Kernel mode: kernel code runs**
    - Using privileged I/O instructions and/or registers



Processor Modes

- Processors support different privilege modes
  - Intel architectures have four privilege modes
  - ARM architectures have eight privilege modes



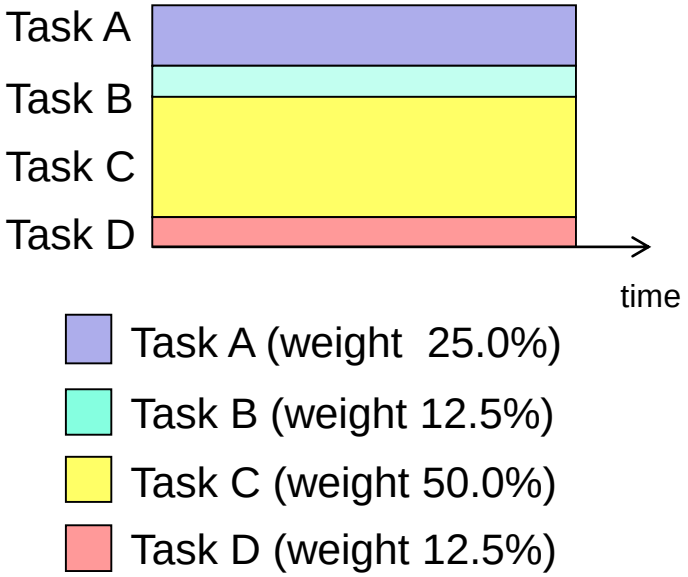
Intel Protection Rings

Mode	Description	
Supervisor (SVC)	Entered on reset and when a Supervisor call instruction (SVC) is executed	Privileged modes
FIQ	Entered when a high priority (fast) interrupt is raised	
IRQ	Entered when a normal priority interrupt is raised	
Abort	Used to handle memory access violations	
Undef	Used to handle undefined instructions	
System	Privileged mode using the same registers as User mode	
User	Mode under which most Applications / OS tasks run	Unprivileged mode

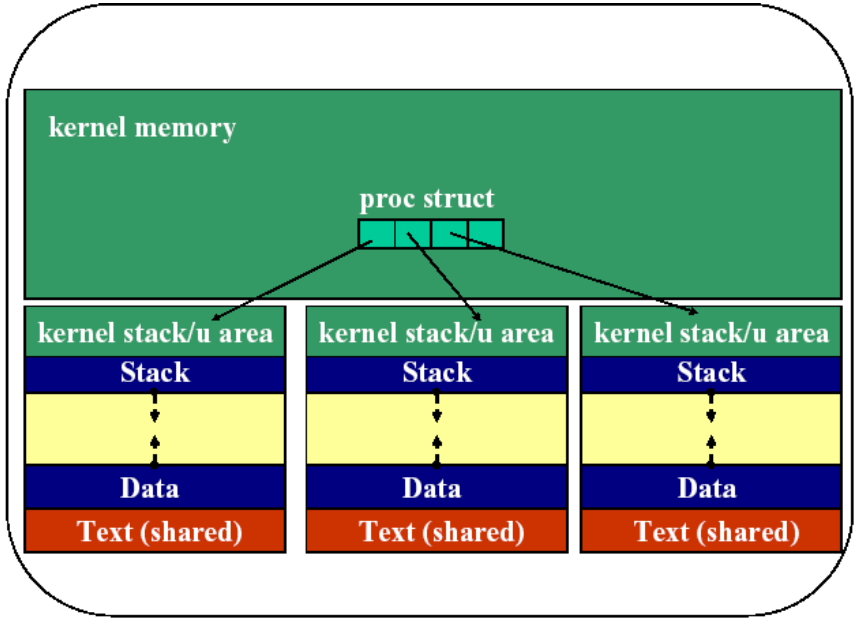
ARM Processor Modes

Illusion

- **Make hardware limitations go away**
  - As if there are infinite number of processors (time sharing)
  - As if a process has large separate memory (virtual mem.)



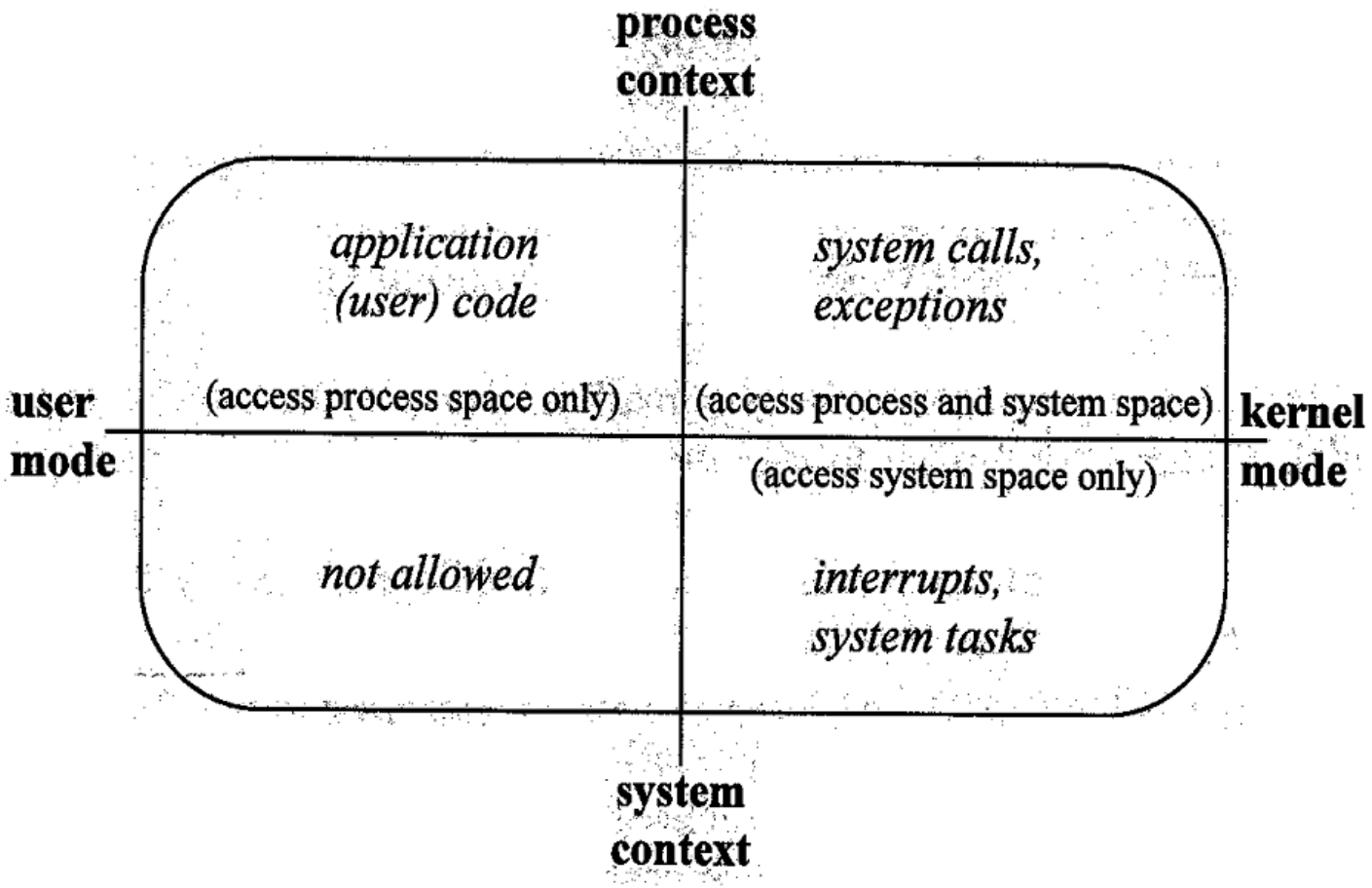
Time Shared Multitasking



Virtual Memory

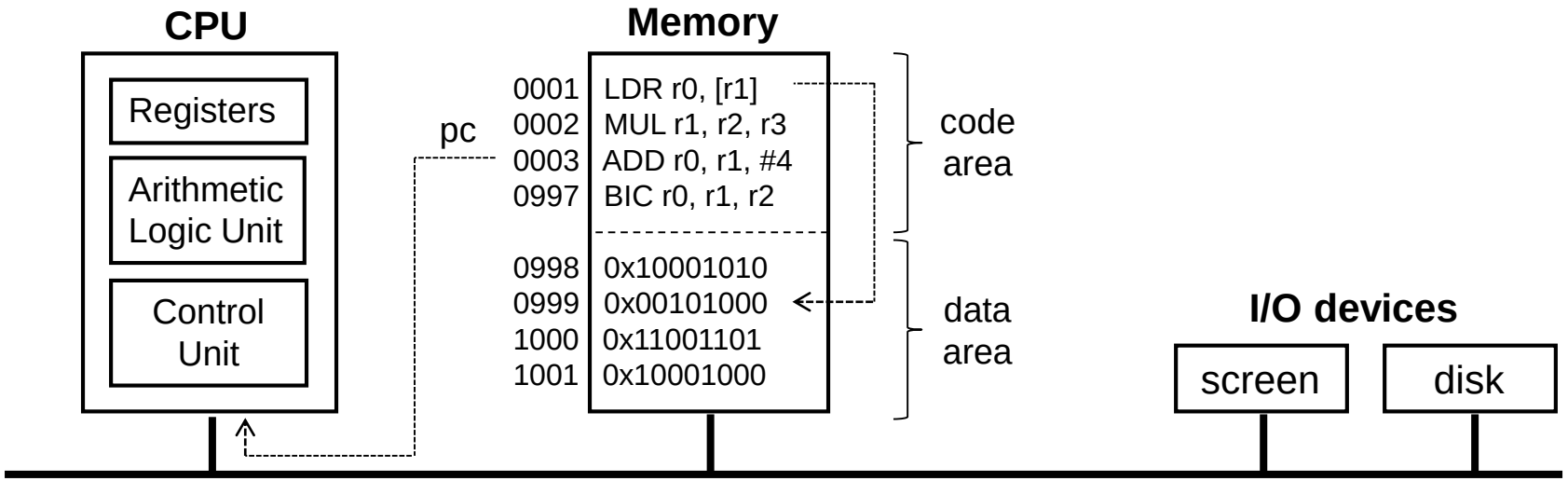
Dual Mode Operation

Execution Mode and Context



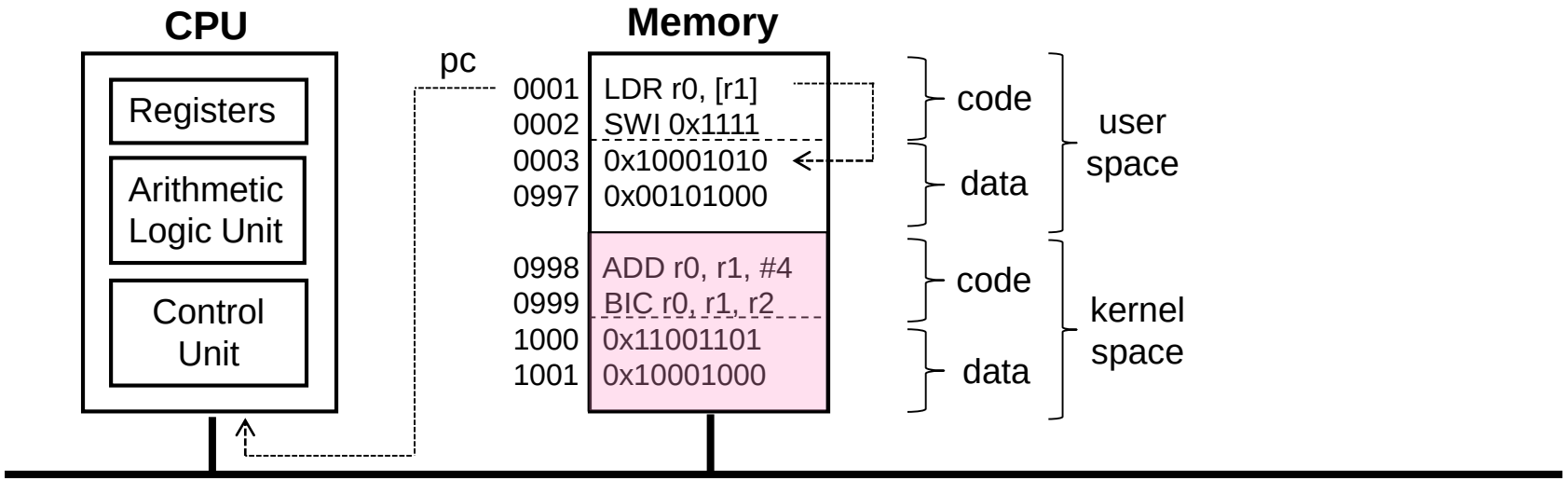
Program Execution without OS

- **von Neumann machine works by**
  - **Fetching instructions from memory**
    - pointed by program counter
  - **Executing them on registers**



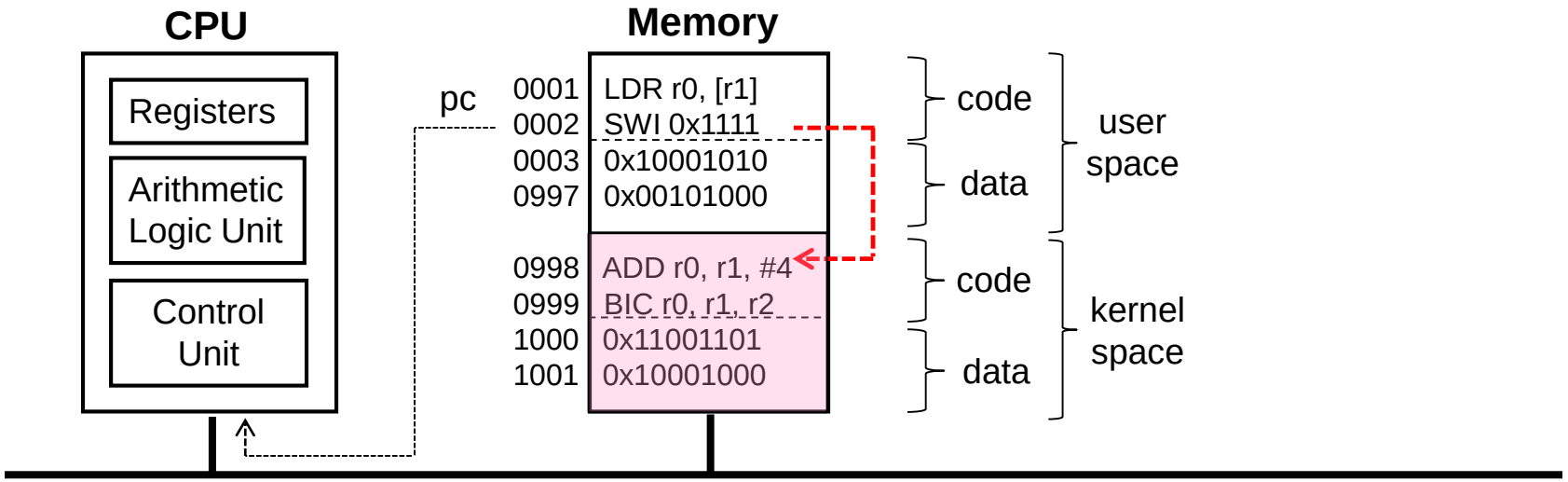
Program Execution with OS

- Memory is divided into two spaces
  - User space
  - Kernel space

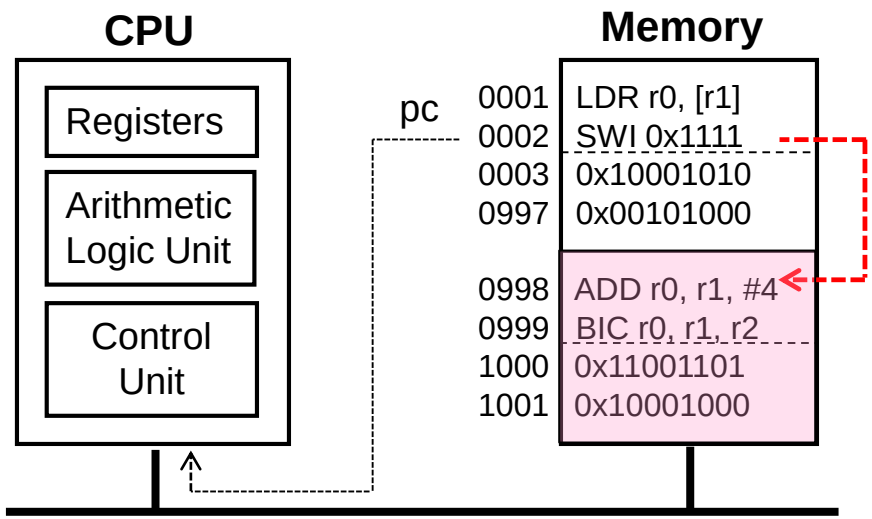


Mode Switching via System Call

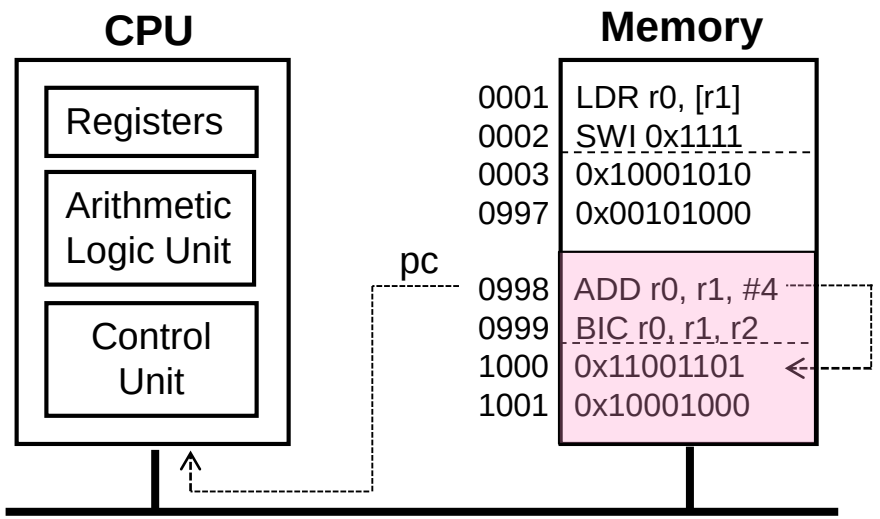
- System calls are similar to function calls, but differ
  - The implementations are in the kernel space
  - They are executed in the kernel mode



Mode Switching via System Call



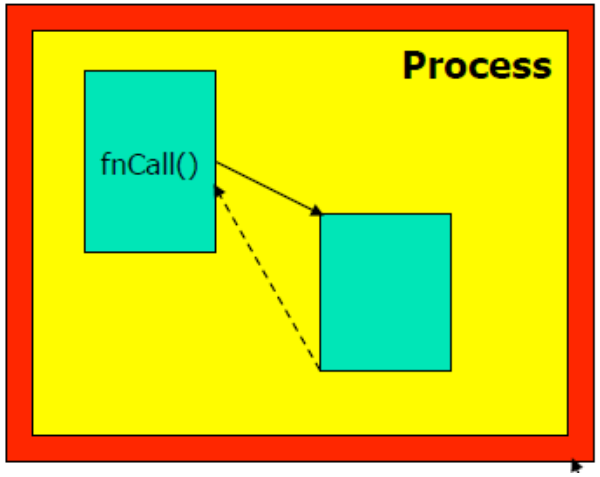
<before invocation>



<after invocation>

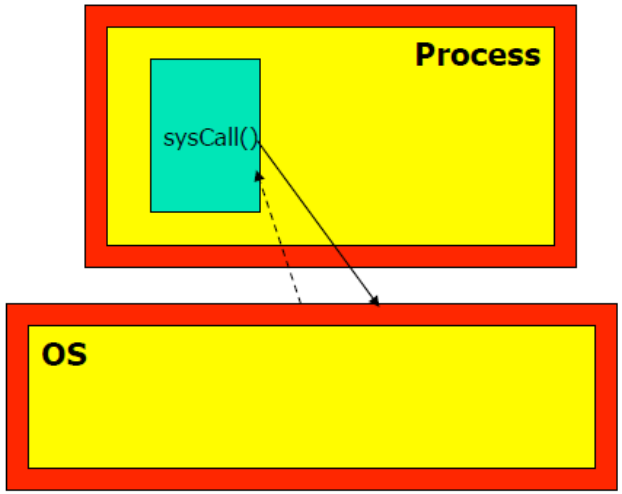
System Calls vs. Function Calls

Function Call



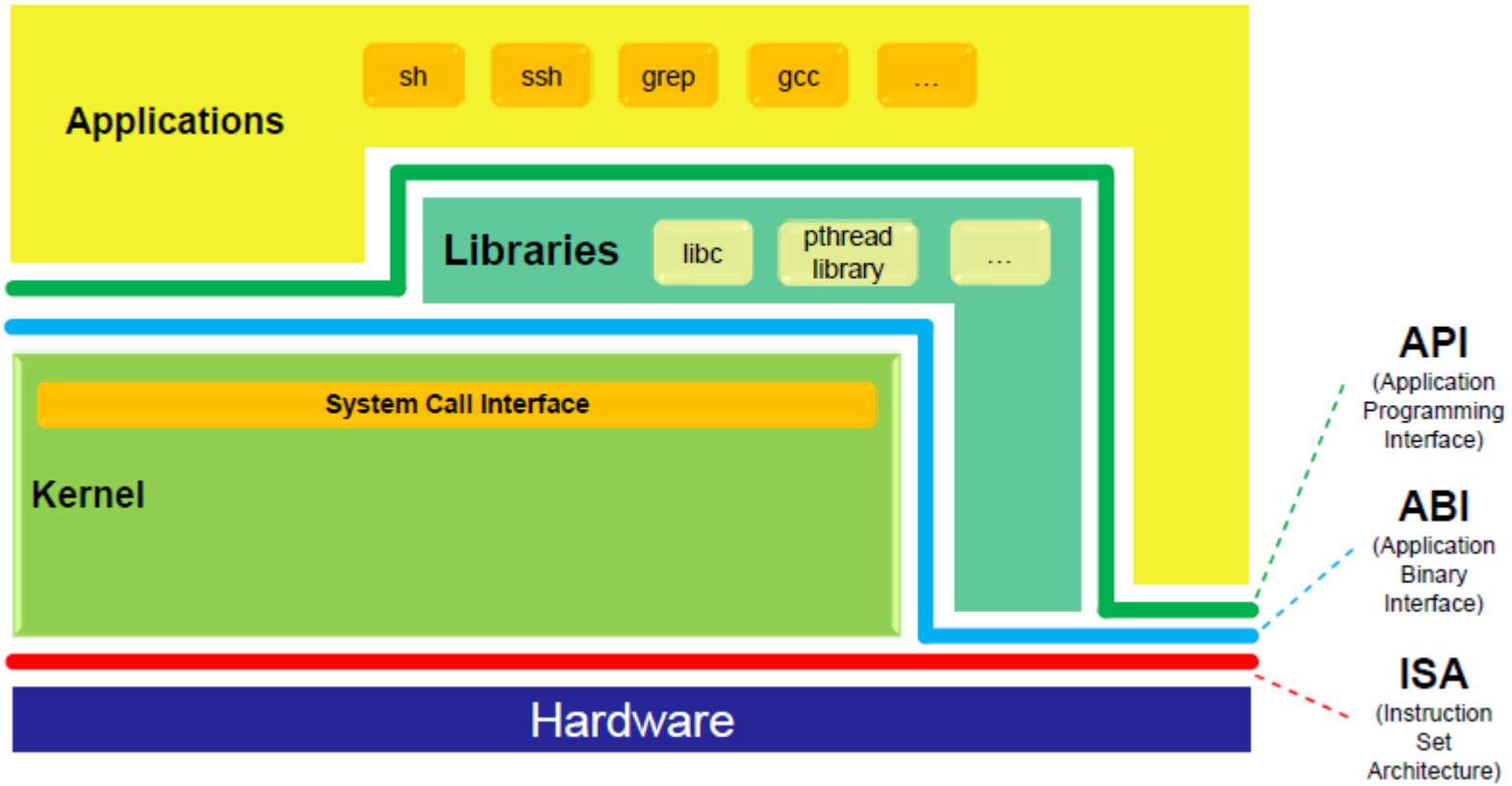
- Caller and callee are in the same process
- User space, user mode

System Call



- Caller and callee are in different spaces
- Kernel space, kernel mode

Common SW Structure

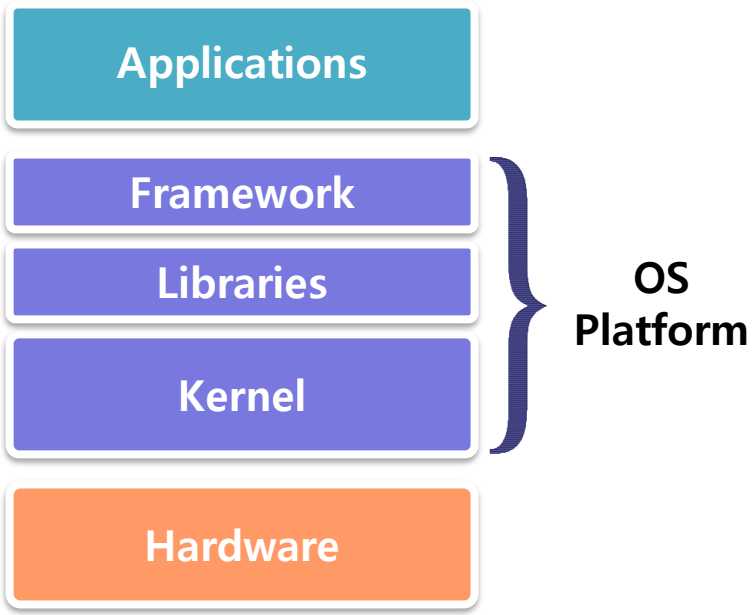


OS is a Reusable SW Platform

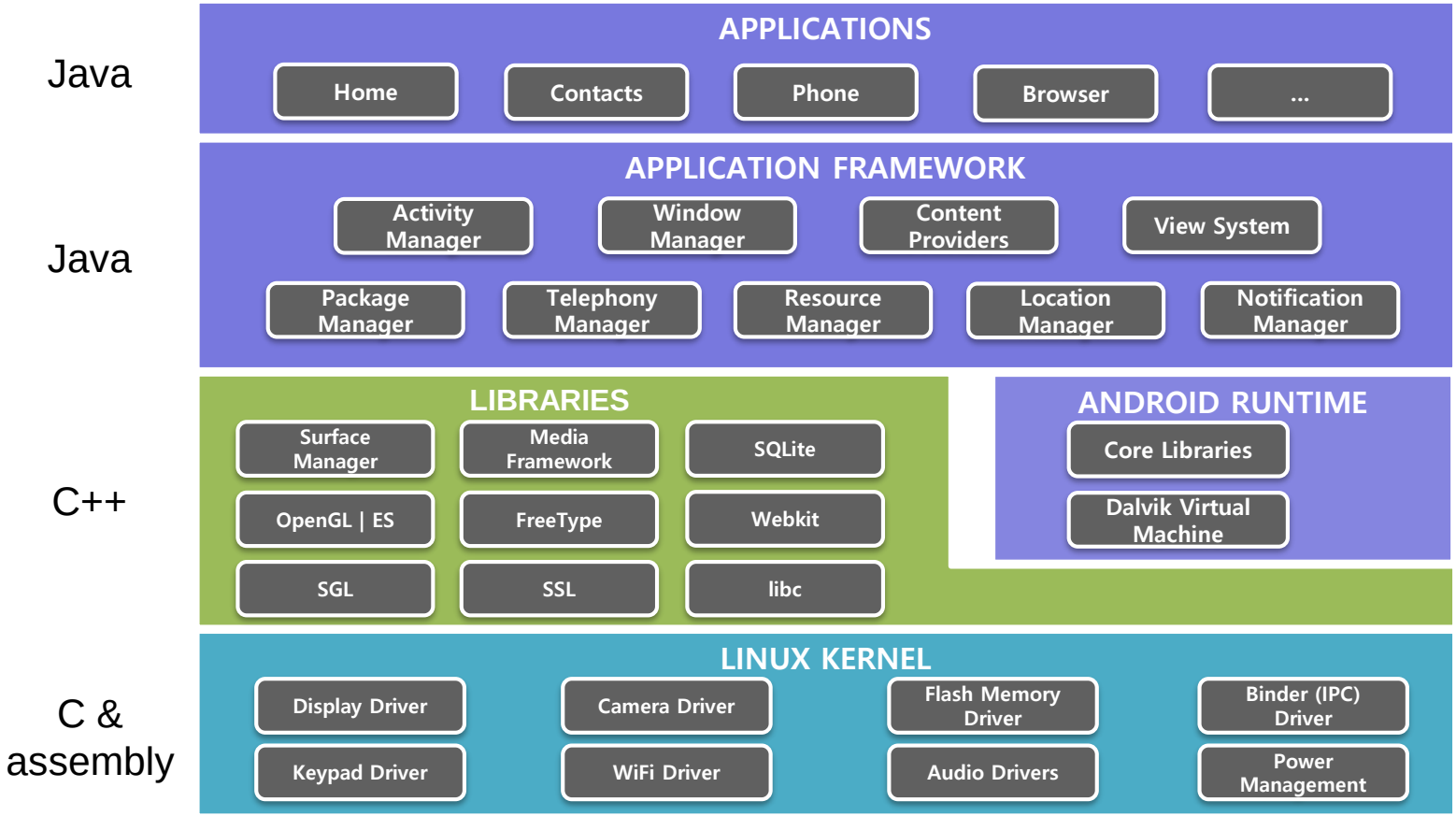
OS is a Reusable Platform

- **OS provides a reusable SW platform**
  - **Reduced development cost and better software quality**

- **Main components**
  - **Framework**
    - Design reuse
    - Inversion of control
  - **Libraries (toolkit)**
    - Code reuse
  - **Kernel**
    - HW abstraction
    - Resource management

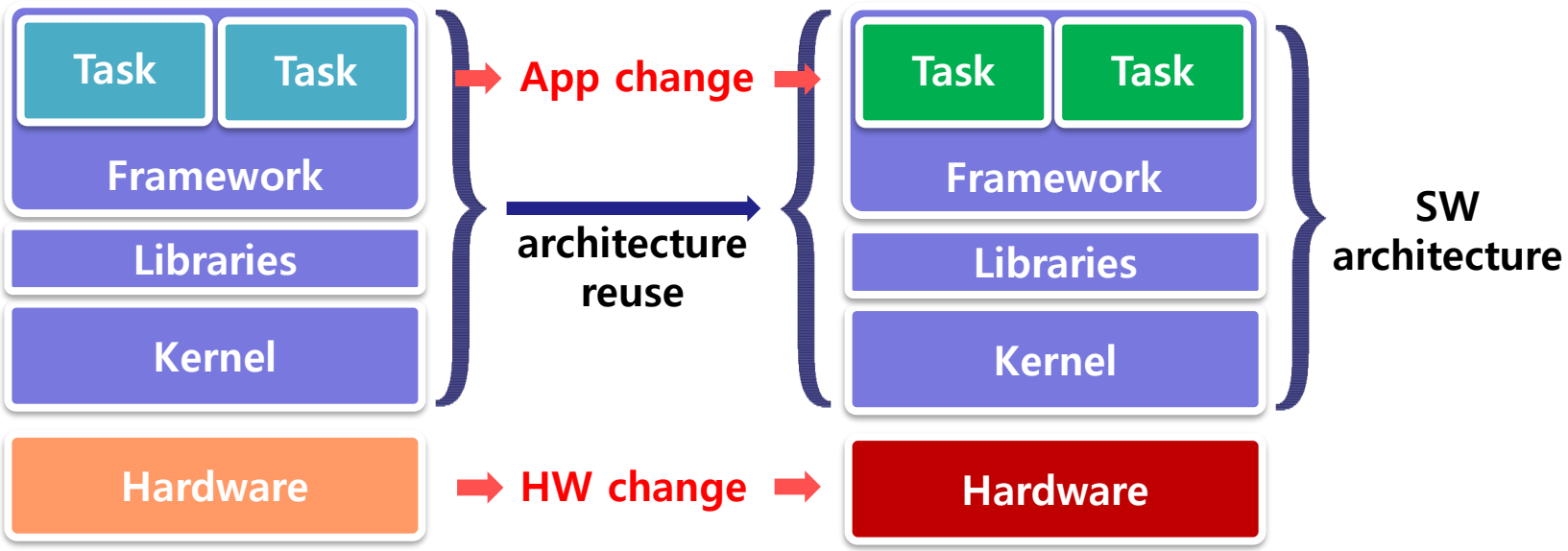


Android Platform

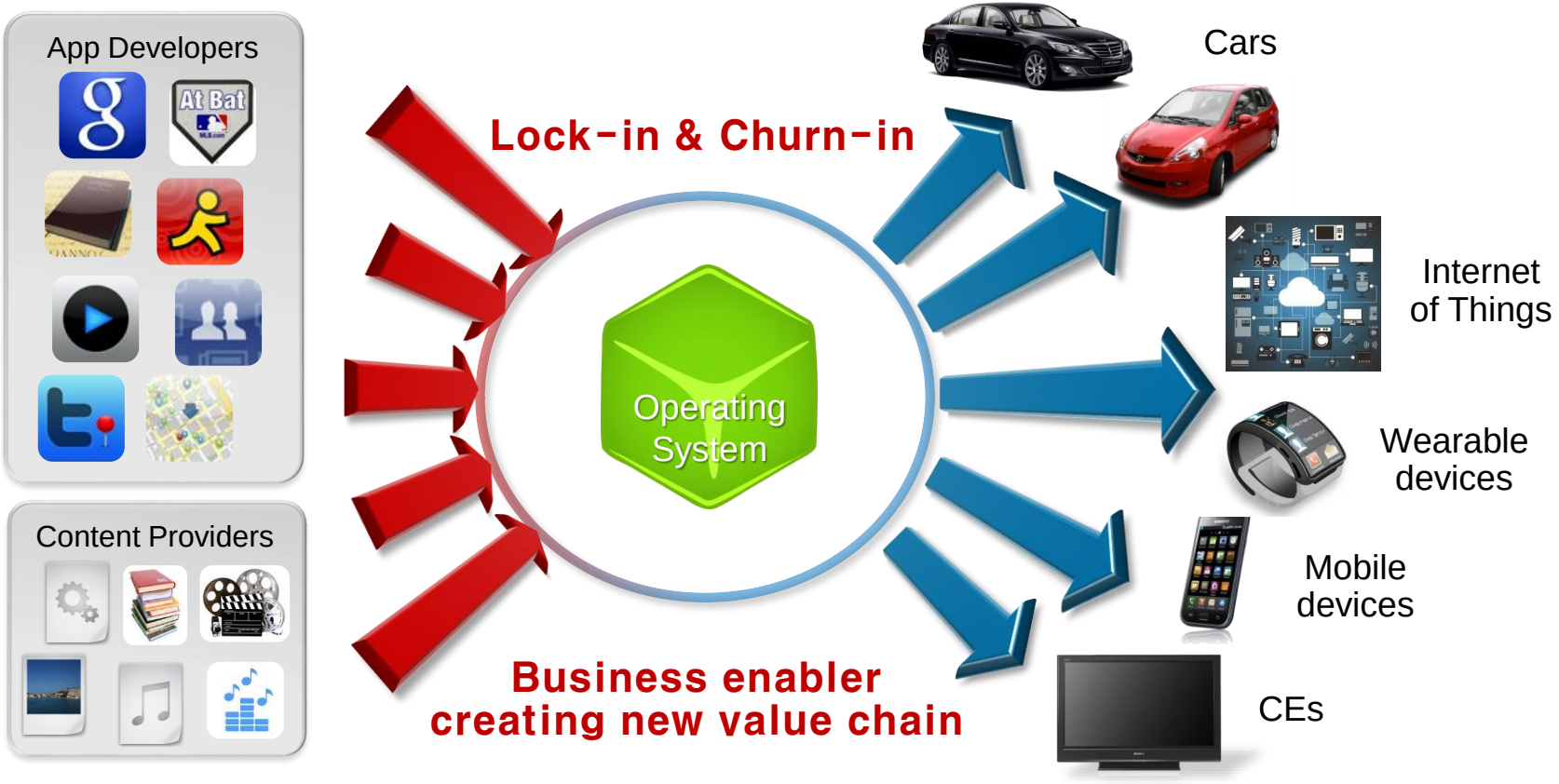


OS defines the Architecture

- OS defines the whole SW architecture
  - Ease of SW design and implementation



OS and the Ecosystem



What is RTOS?

RTOS does not mean “fast”

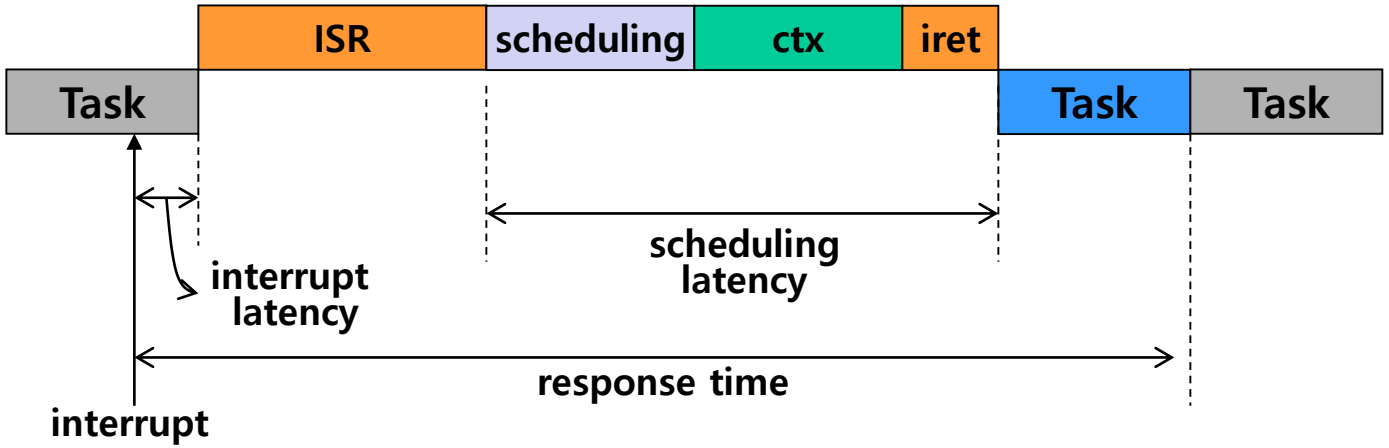
- RTOS adds runtime overheads
- Slightly slower performance than OS-less systems

RTOS does mean “deterministic”

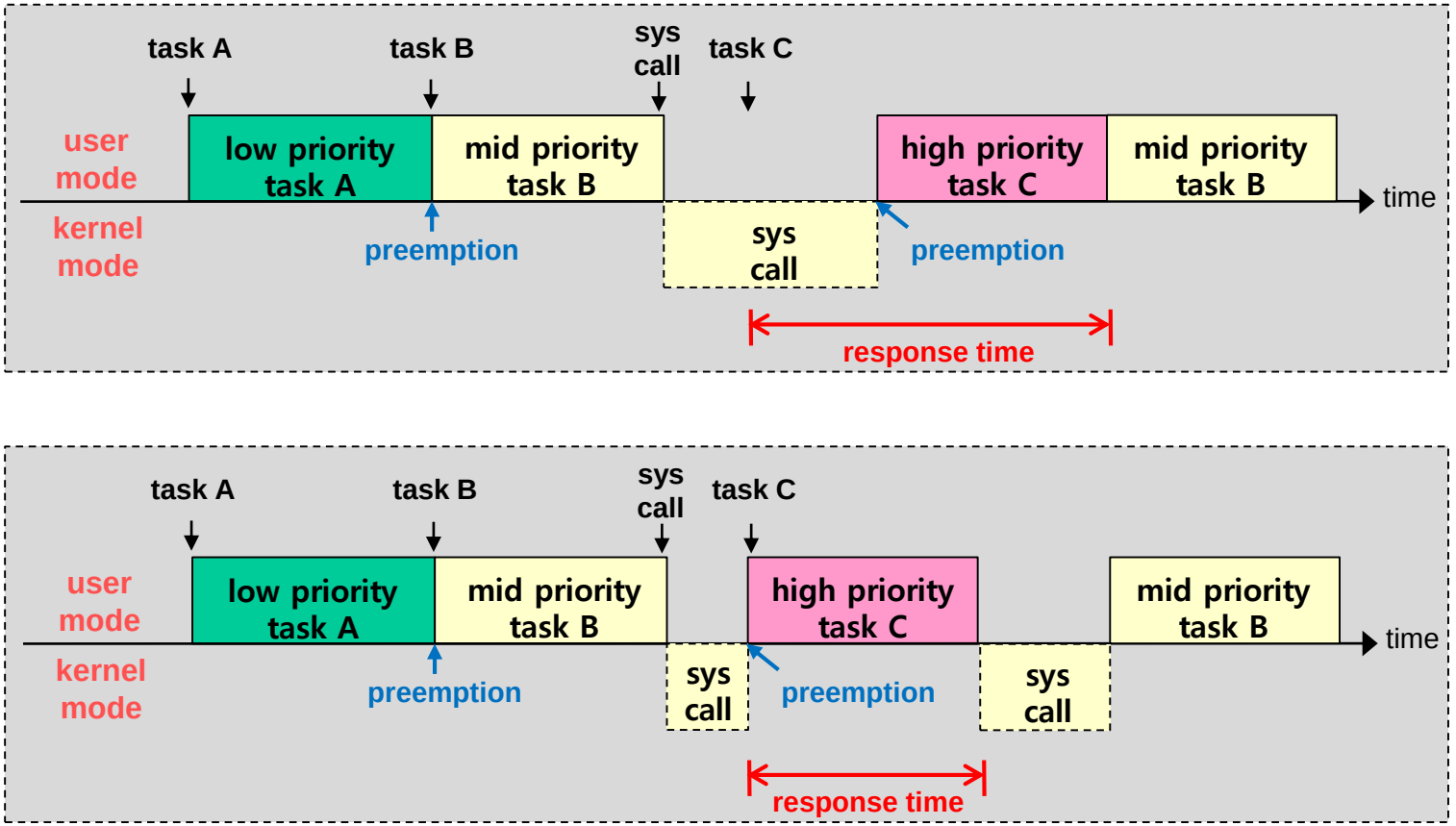
- Guarantees “worst case” times
- Better performance than GPOS

Timing Guarantees from RTOS

- **Key requirements**
  - **Fully preemptive priority-based scheduling**
    - High priority tasks preempt low ones
  - **Bounded latency**
    - eg.) interrupt off time < α and scheduling latency < β



Partially Preemptive vs. Fully Preemptive



Commercial RTOSes

Commercial RTOSes

- **QNX, VxWorks, pSOS, VRTX, Nucleus**

Features

- **Multithreading, real-time synchronization and POSIX APIs**
- **Lightweight kernel**
  - Even scaled down to fit in the ROM of the system
- **Modularized structure**
  - Minimum kernel + service components
- **Responsiveness**
  - Minimal interrupt and switching latency

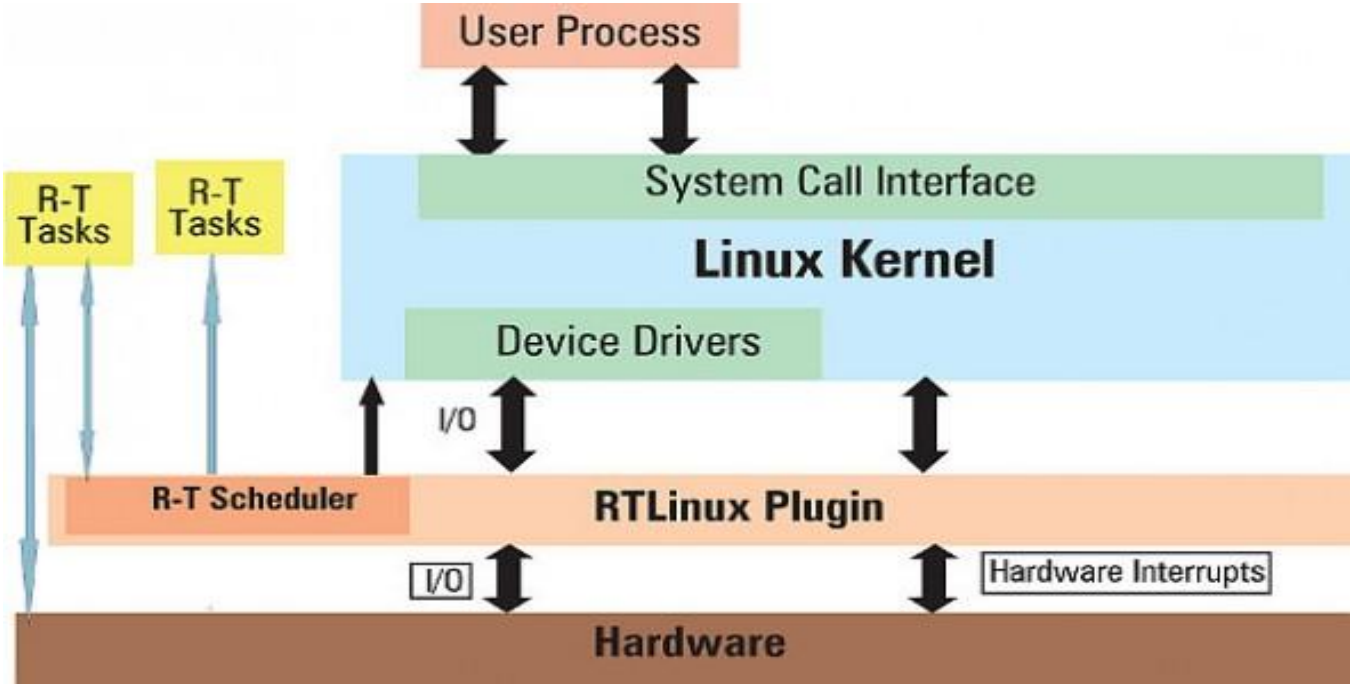
Linux for Real-Time?

- Linux is a general purpose OS
  - Aimed at high performance and stability
- Old Linux did not support
  - Full preemption and bounded latency
- Current Linux supports (PREEMPT_RT)
  - ✓ Deterministic scheduler (SCHED_FIFO and EDF)
  - ✓ Priority inheritance mutexes
  - ✓ Lockable memory (disabling demand paging)
  - ✓ High resolution timers
  - ? Bounded latency inside the kernel
  - ? Boot-up time

Linux Variants for RT

- Extensions for real-time systems
  - RTLinux
  - PREEMP_RT patch
  - Xenomai

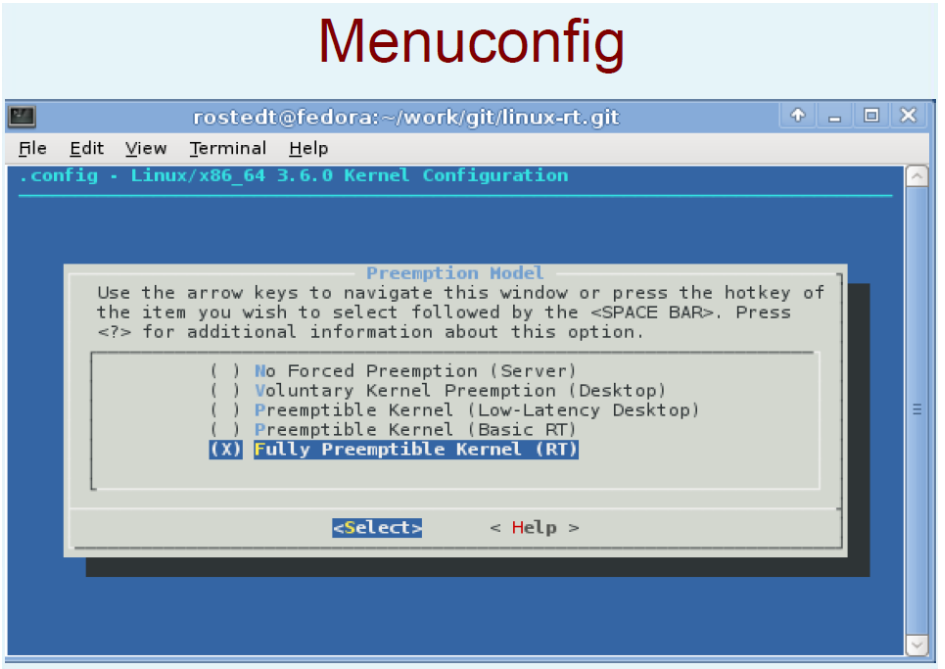
RTLinux



- Developed by Victor Yodaiken, Michael Barabanov, Cort Dougan as a commercial product at FSMLabs
- Wind River Systems acquired FSMLabs in February 2007, but ended it in 2011

PREEMPT_RT

- PREEMPT_RT has merged into the mainline kernel
  - Since August 2006 by Ingo Molnar *et al.*
  - https://www.kernel.org/pub/linux/kernel/projects/rt/



PREEMPT_RT

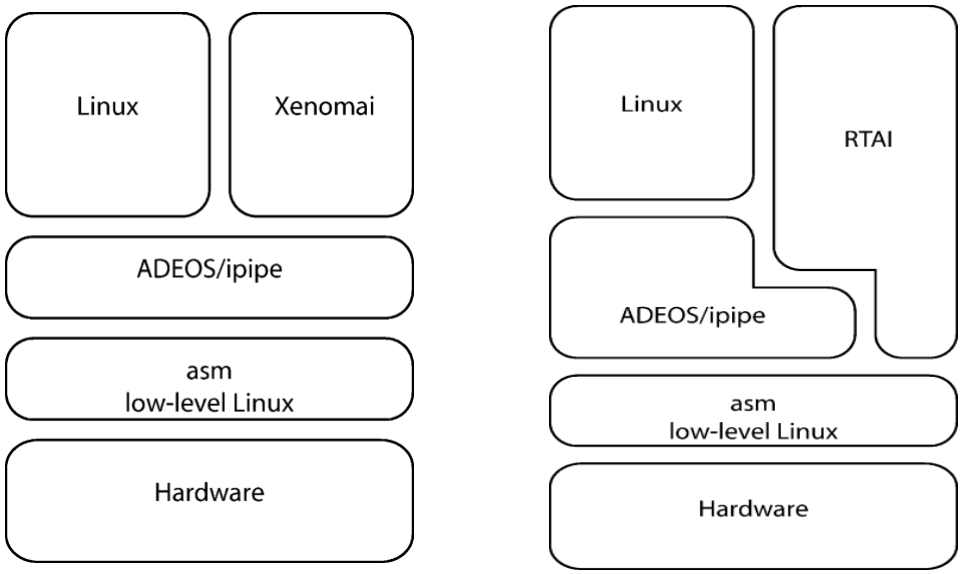
- **Features**
  - Preemptible critical sections
  - Preemptible interrupt handlers
  - Priority inheritance for in-kernel spinlocks and semaphores
  - Deferred operations
  - Some latency-reduction measures

- **Work is still going on**
  - 80 percent of PREEMPT.RT is already in mainline

Xenomai

Xenomai 1.0

- Announced in 2001 as portability framework for RTOS applications
- Development of ADEOS layer for Linux and RTAI
- Merged with RTAI => RTAI/fusion

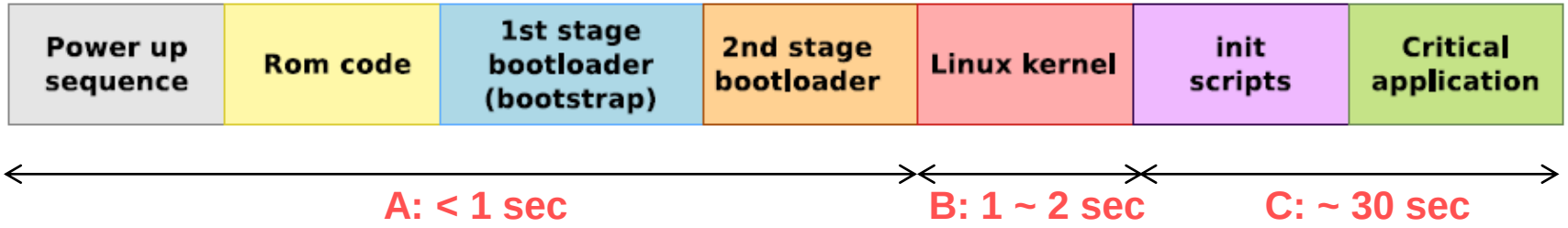


Xenomai

- Xenomai 2.0
  - Departed from RTAI in 2005 – incompatible design goals
  - Evolved ADEOS to I-pipe layer (also used by RTAI)
  - Ported to 6 architectures
- Xenomai 3.0
  - Released in 2015 after >5 years of development
  - Rework of in-kernel core (now POSIX-centric)
  - Support for native Linux
- It now comes in two flavors
  - As co-kernel extension for (patched) Linux
  - As libraries for native Linux (including PREEMPT-RT)

Linux Boot-up Time

- Generic boot sequence
  A. Firmware initialization phase
  B. Kernel initialization phase
  C. User Space initialization phase



- Typical boot time
  - < 60 seconds: desktops, set top boxes, GPS devices, …
  - < 30 seconds: smartphones
  - < 5 seconds: smart TVs

Boot-up Time Reduction

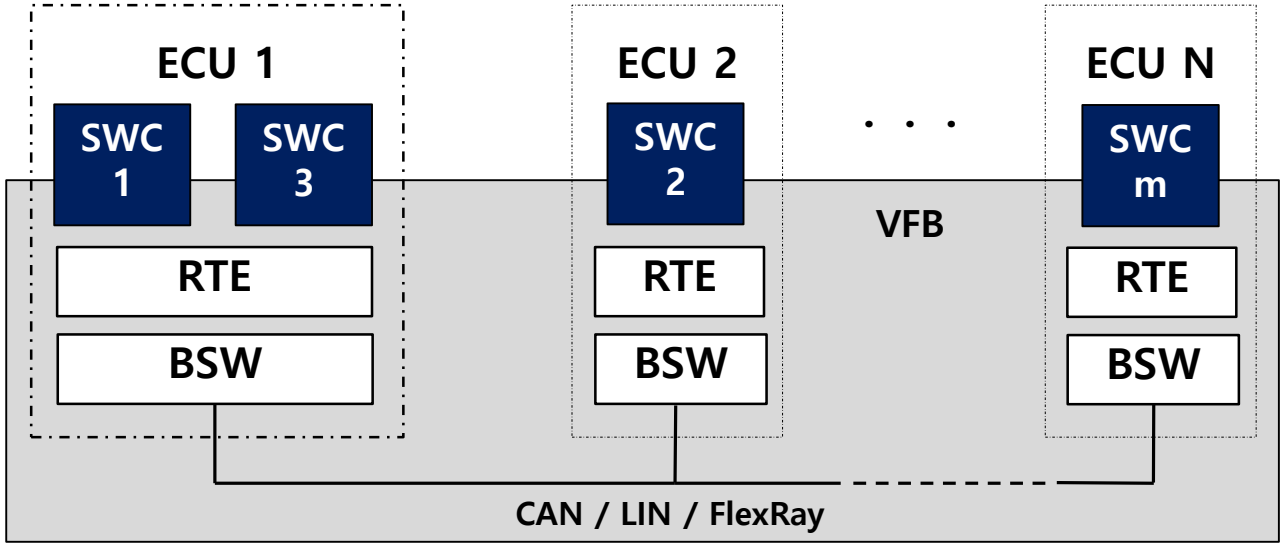
- General techniques for boot-up time reduction
  - Optimize each job of the boot sequence
  - See backup slides for details
- Snapshot-based techniques
  - Save RAM content to nonvolatile storage before power-off
  - Upon power-on, restore RAM state
- Limitations of snapshot approach
  - Stability problem
  - Retake snapshot every time power is turned off
  - Take snapshot only at “stable” state
  - Android adds another SW layer, increasing snapshot size

GPL License Issue

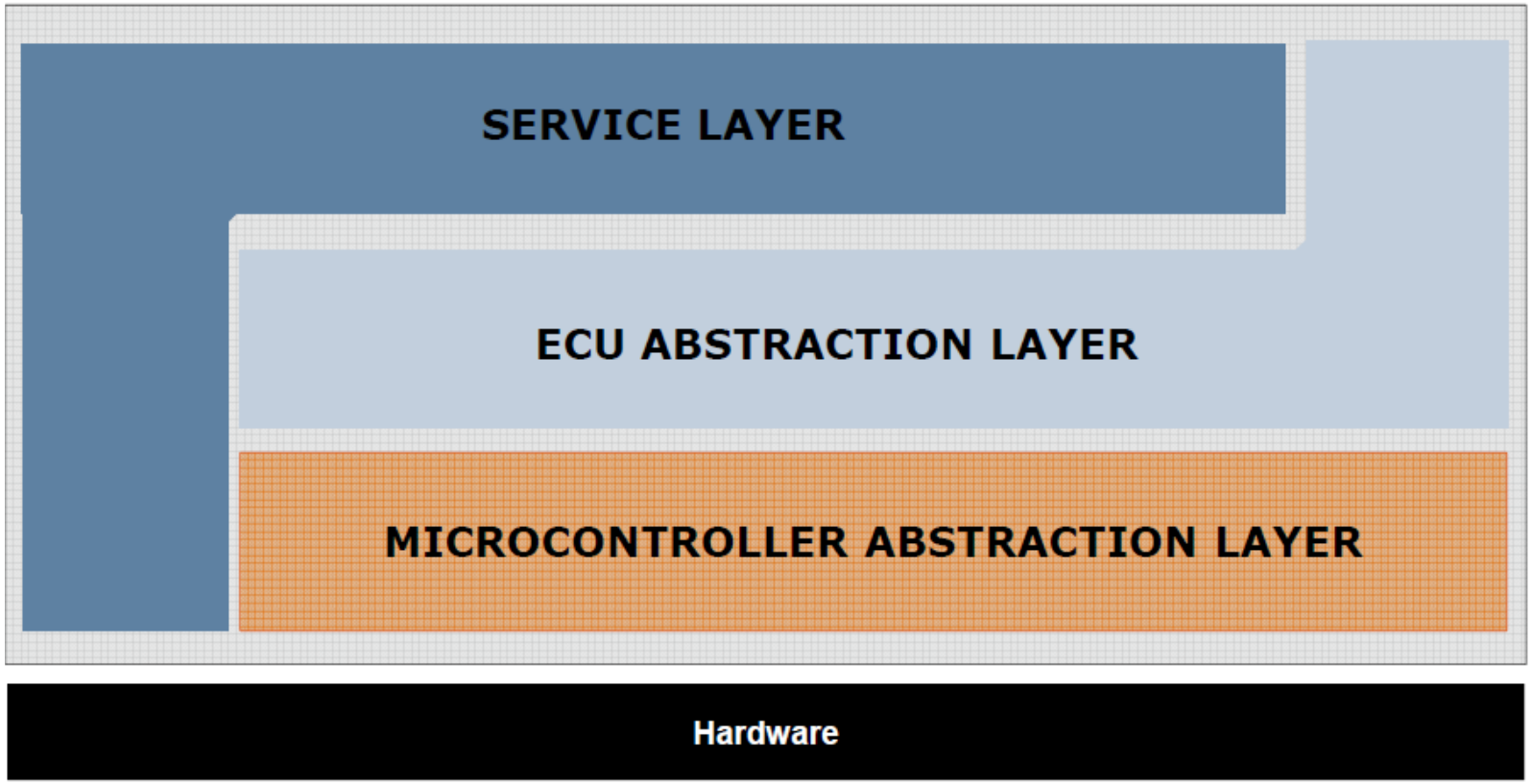
- GPL guarantees end users the freedoms to use, study, share (copy), and modify the software
  - Written by Richard Stallman of the Free Software Foundation (FSF) for the GNU project
- Restriction
  - Modified or derived code must be released under GPL
- LGPL (Lesser GPL)
  - Library linking is allowed without releasing the code
  - Android uses Bionic libc derived from BSD libc to isolate apps from GPL and LGPL

AUTOSAR

- **SWC (Software Component)**
  - Smallest unit of application software composition
- **RTE (Runtime Environment)**
  - Middleware for inter- and intra-ECU communication
- **BSW (Basic Software)**
  - Traditional OS services



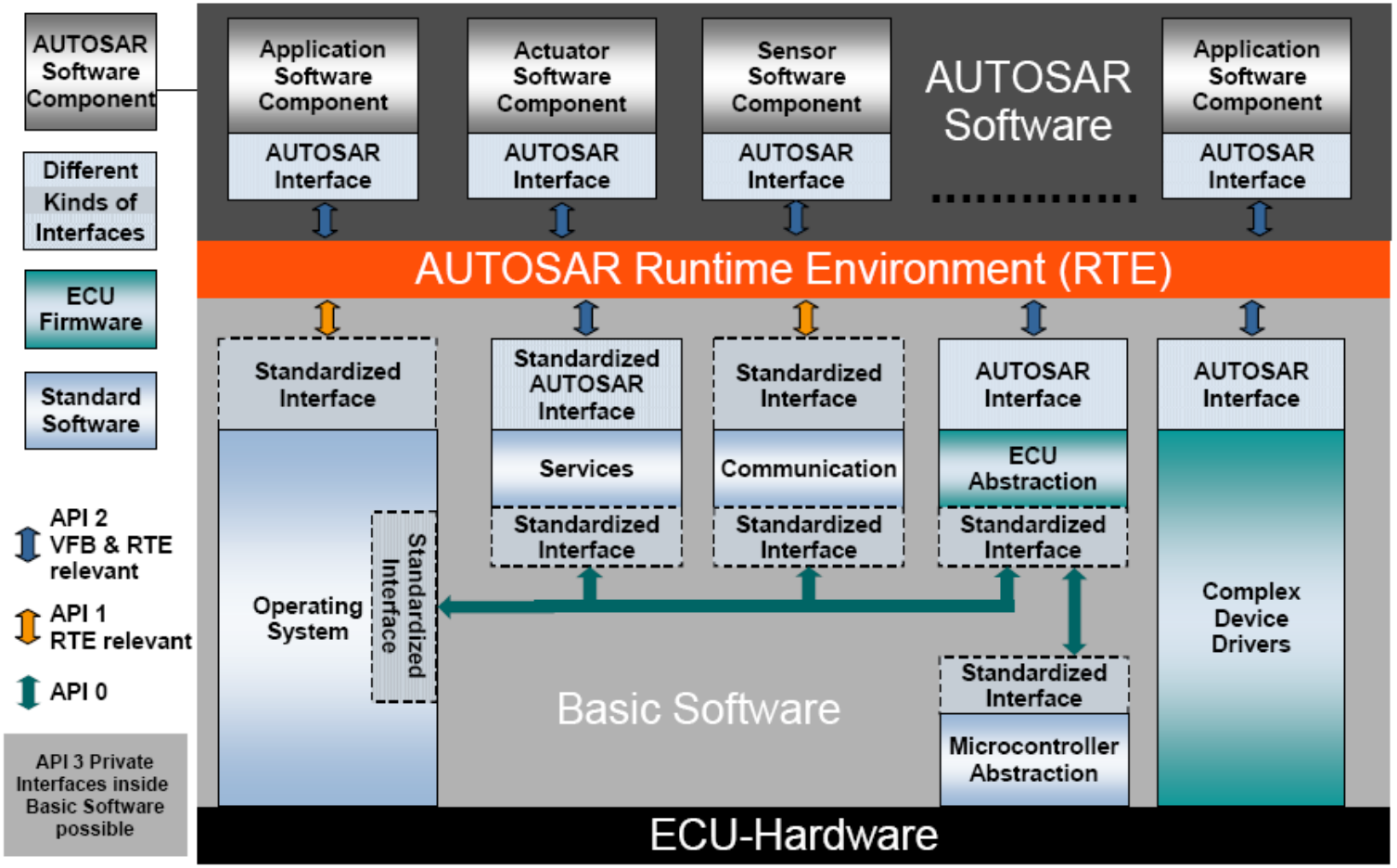
Basic Software (Layered Architecture)



Basic Software

- The Service Layer provides background services such as network services, memory management and bus communication services
  - The operating system is also contained in this layer
  - AUTOSAR OS = OSEK + Protection + Schedule Table + IOC
- The Microcontroller Abstraction Layer (MCAL) offers device drivers for access to memory, communication and input/output (IO) of the microcontroller
- The ECU Abstraction Layer (ECUAL) offers uniform access to all functionalities of an ECU such as communication, memory or IO

AUTOSAR


AUTOSAR Software Component
Different Kinds of Interfaces
ECU Firmware
Standard Software
API 2 VFB & RTE relevant
API 1 RTE relevant
API 0
API 3 Private Interfaces inside Basic Software possible
Application Software Component
AUTOSAR Interface
Actuator Software Component
AUTOSAR Interface
Sensor Software Component
AUTOSAR Interface
AUTOSAR Software
Application Software Component
AUTOSAR Interface
AUTOSAR Runtime Environment (RTE)
Standardized Interface
Operating System
Standardized Interface
Standardized AUTOSAR Interface
Services
Standardized Interface
Standardized Interface
Communication
Standardized Interface
AUTOSAR Interface
ECU Abstraction
Standardized Interface
Standardized Interface
Microcontroller Abstraction
AUTOSAR Interface
Complex Device Drivers
Basic Software
ECU-Hardware


Q&A

Virtualized OS Architectures

Minsoo Ryu

Department of Computer Science and Engineering
Hanyang University

OS Architectures

- Virtualized OS architectures
  - Type 1 virtualization
  - Type 2 virtualization
  - Full virtualization vs. paravirtualization
  - OS-level virtualization

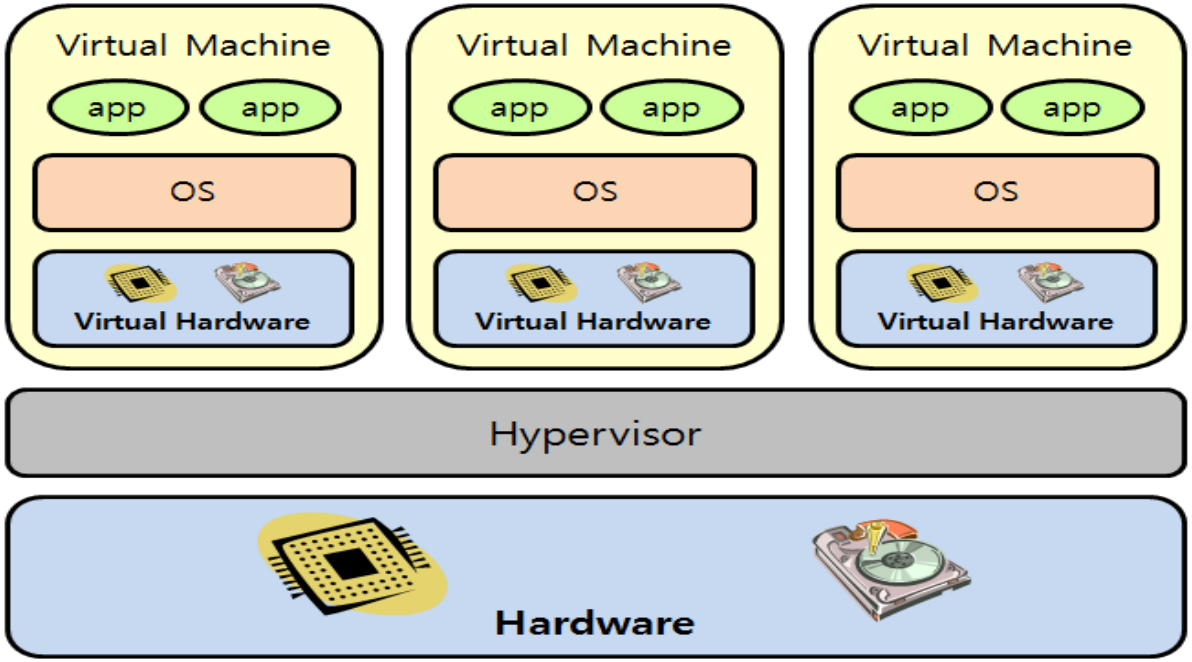
Virtualized Architectures

- **Creation of virtual machines**
  - **Partition a hardware platform into one or more virtual machines**
  - **Each virtual machine acts like a real, separate computer hardware platform**

- **Goals of virtualization**
  - **Isolation and security**
    - Isolate performance
    - Isolate faults and errors (fault containment)
  - **Consolidation**
    - Provide various execution environments on single hardware platform

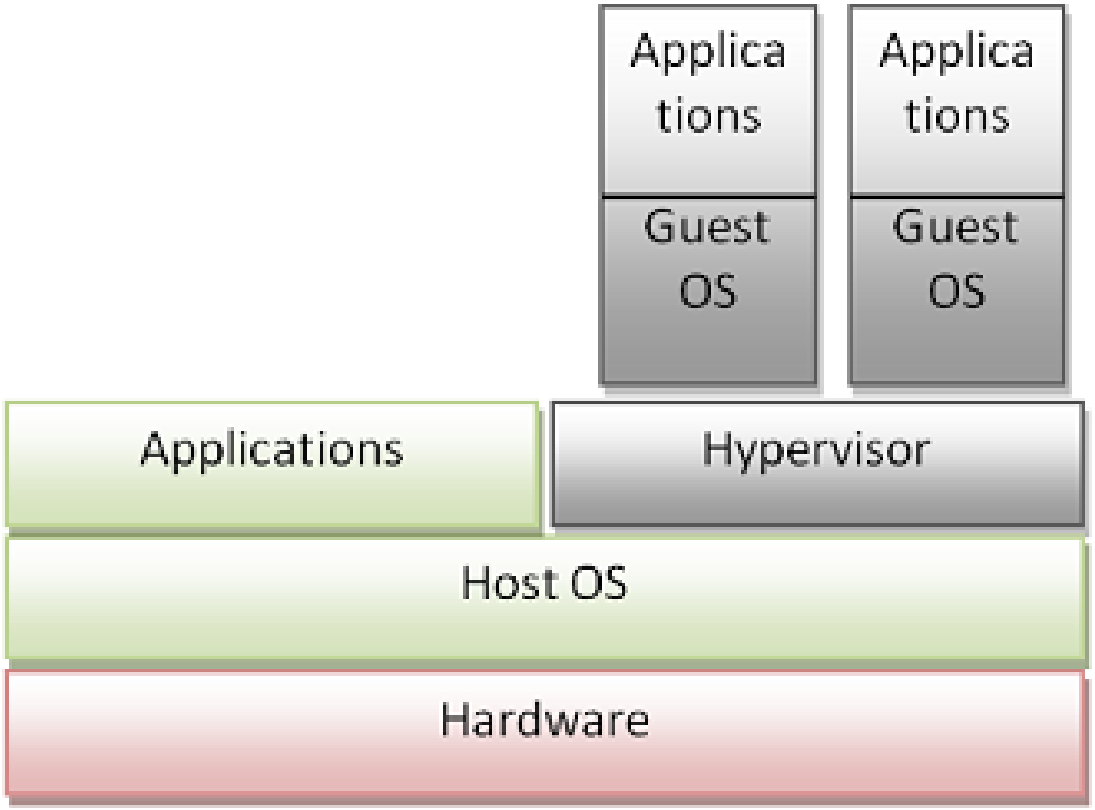
Type 1 Virtualization

- Runs on "bare metal"
  - Known as "native" virtualization
  - Hypervisor or VMM (virtual machine monitor) multiplexes hardware



Type 2 Virtualization

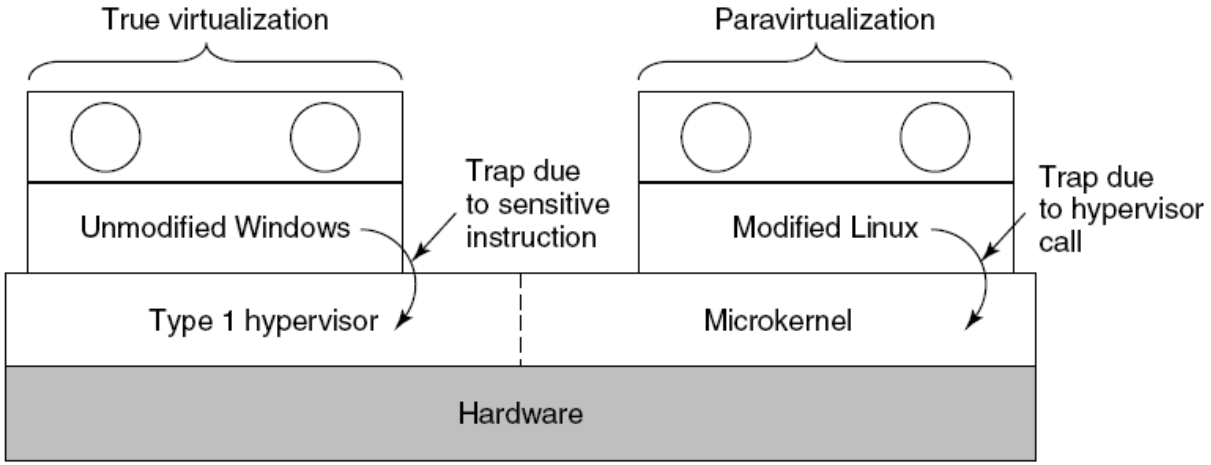
- **Runs within an OS**
  - **Known as "hosted" virtualization**



Full Virtualization vs. Paravirtualization

- **Full virtualization**
  - The virtual machine simulates enough hardware to allow an unmodified "guest" OS to run (binary translation)

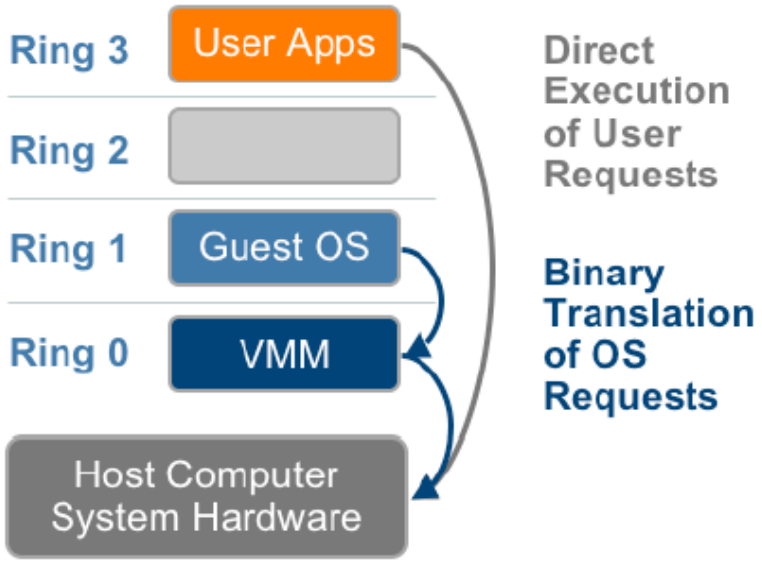
- **Paravirtualization**
  - The virtual machine offers a special API that can only be used by a modified guest OS



Full Virtualization

- **The guest OS is not aware it is being virtualized**
  - **Requires no modification**

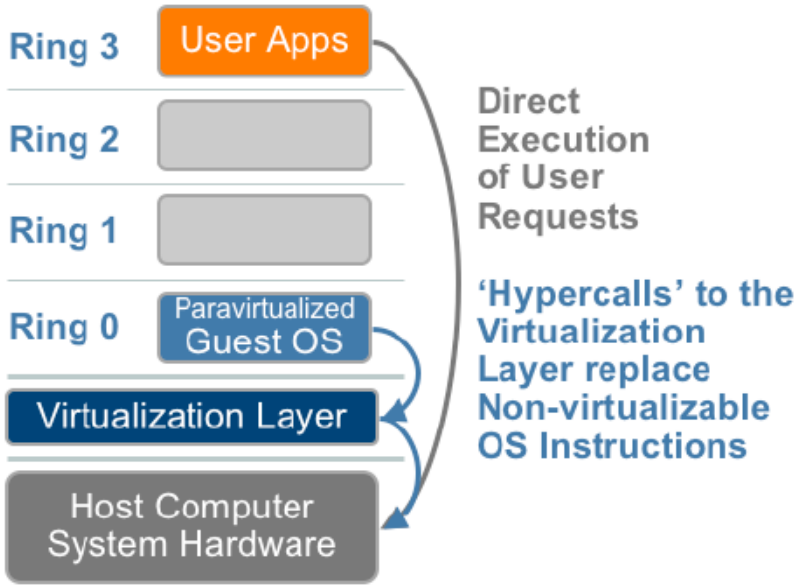
- **Key techniques**
  - **Direct execution**
    - User-level code is directly executed on the processor
  - **Binary translation**
    - The hypervisor translates all OS (privileged) instructions on the fly
    - Caches the results for future use



Paravirtualization

- The guest OS is modified
  - Compatibility and portability is poor
  - Introduces significant support and maintainability issues in production environments

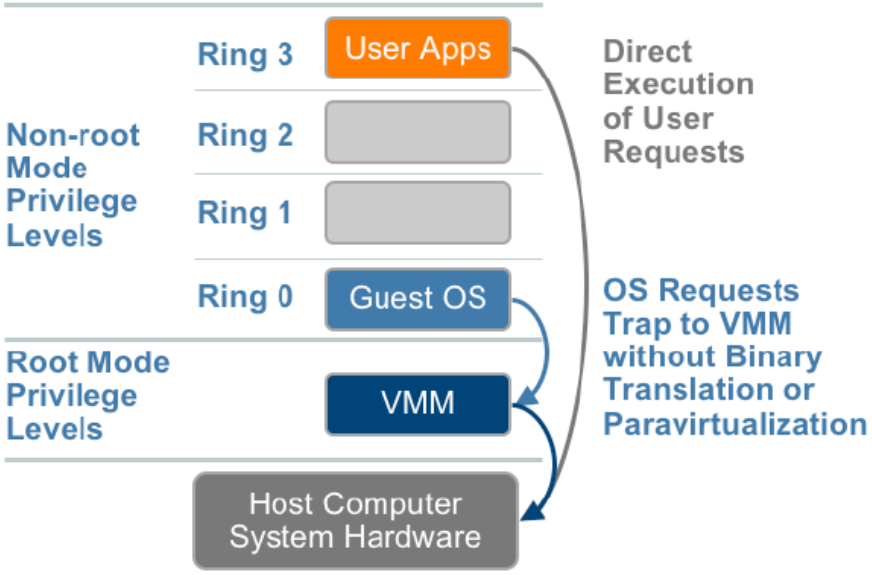
- Key techniques
  - Replace non-virtualizable instructions with hypercalls
  - Hypercalls are special APIs used for communicating directly with the hypervisor



Hardware Assisted Virtualization

- Hardware vendors are rapidly embracing virtualization
  - Provide privileged instructions with a new CPU execution mode feature that allows the VMM to run in a new root mode below

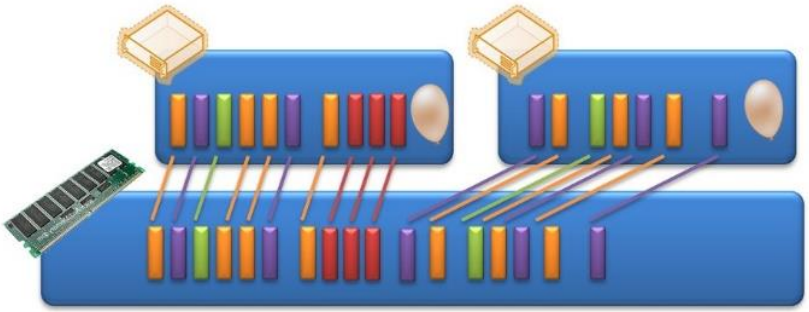
- Examples
  - Intel Virtualization Technology (VT-x)
  - AMD's AMD-V
  - ARM virtualization extension in Cortex-A15

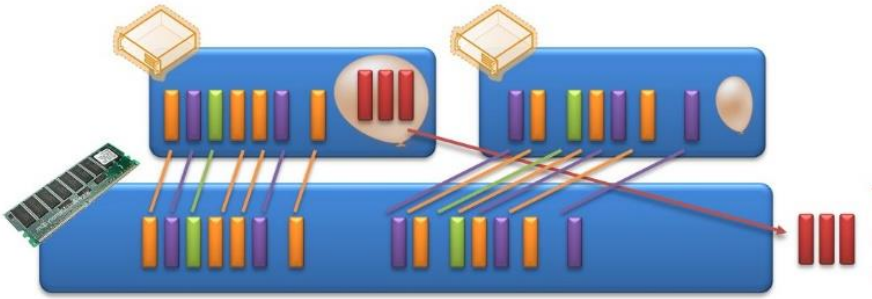


Some Issues on Virtualization

- No communication between a guest OS and VMM
  - No information exchange between guest OSes and VMM
  - This may cause serious performance degradation

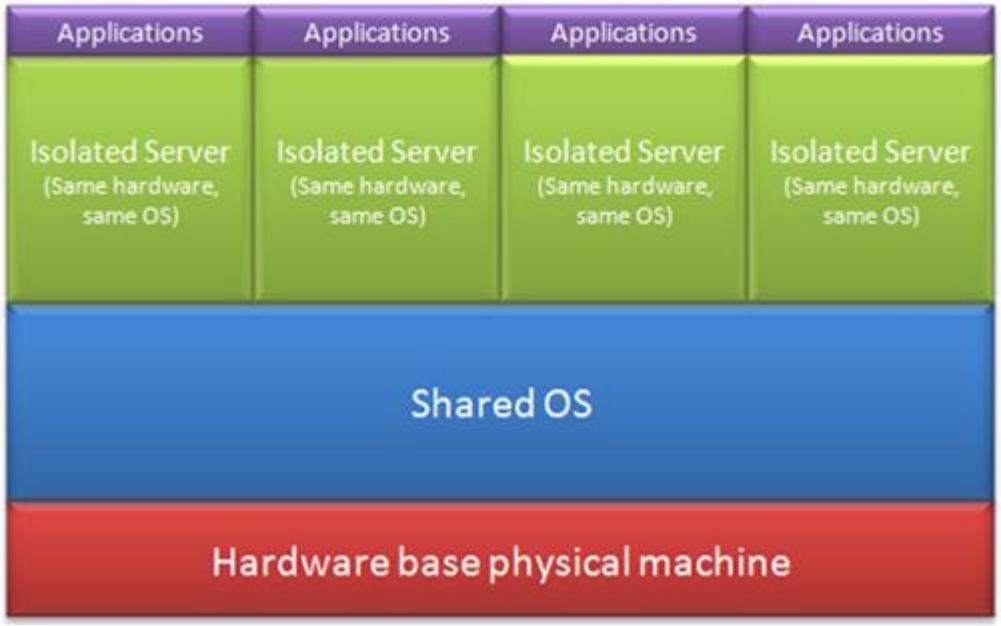
- Page replacement and balloon process
  - The VMM has little knowledge about page replacement
  - Guest OSes has superior knowledge about page replacement





OS-level Virtualization

- A server virtualization method
  - Allows for multiple isolated user space instances
  - Often called containers, virtualization engines (VE), virtual private servers (VPS) or jails




Q&A