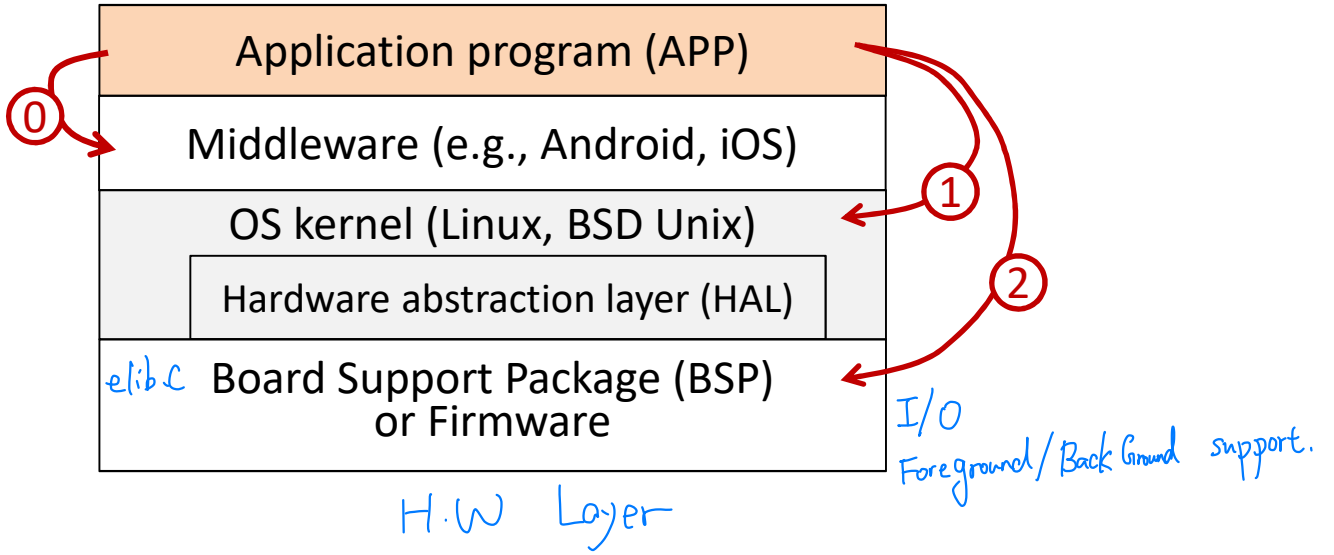
Microprocessor Support for Operating Systems

Chun-Jen Tsai
NYCU
11/25/2022

Software Stack of Computing Systems

- Layer structure of software on a computing system:

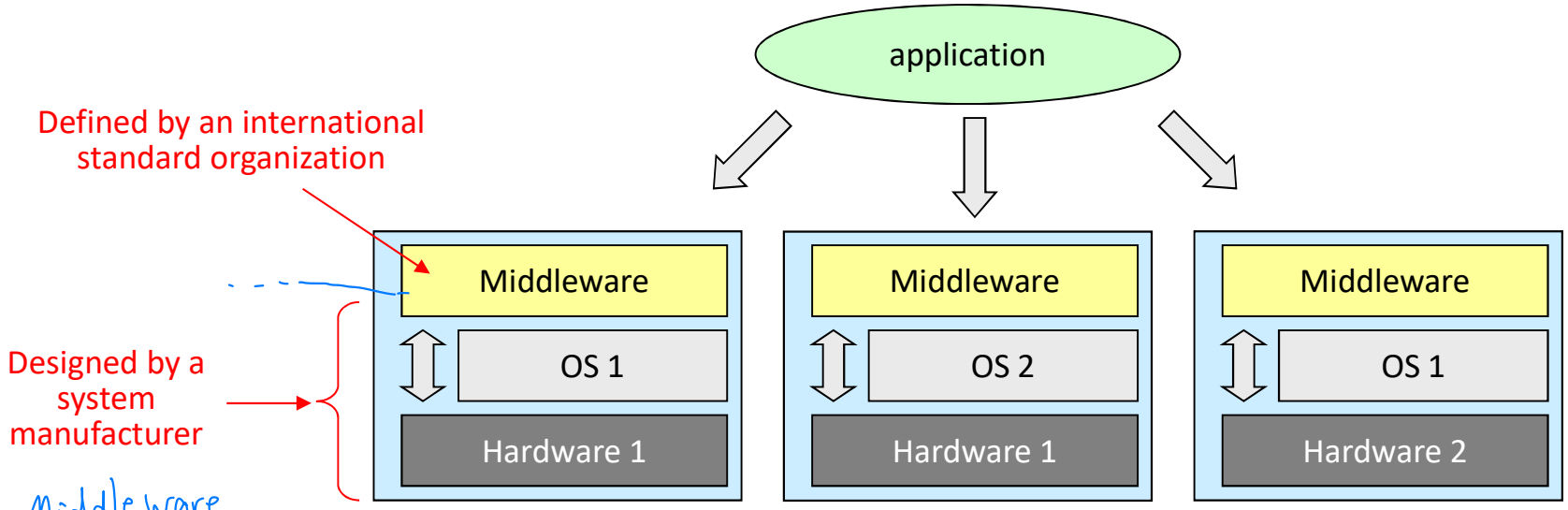


- Ideally, an application shall only call middleware APIs to maintain portability

FreeRTOS we use only support single core. There are other RTOS with multi-core support.

Middleware: Runtime Environment

- The purpose of middleware is to provide a portable runtime environment for applications
  - An "universal" OS could handle portability, but due to the variety of μPs and OSs, we need something at a higher level



About Middleware

Android (Google) use Java language to write OS, for its portability and VM. (as an Middleware)

IOS (Apple) uses binary translation (compiler)

Classification of Runtime Systems

- Runtime software environment can be classified as:
  - Foreground-background (F/B) systems
    - One thread plus event-driven tasks
    - One memory space, single execution level
  - Real-Time OS (RTOS)
    - Multiple threads and event-driven tasks
    - One memory space, single execution level
  - Full-featured OS
    - Multiple processes, each with its own memory space
    - Each process has multiple threads
    - Multiple execution levels

no virtual address.

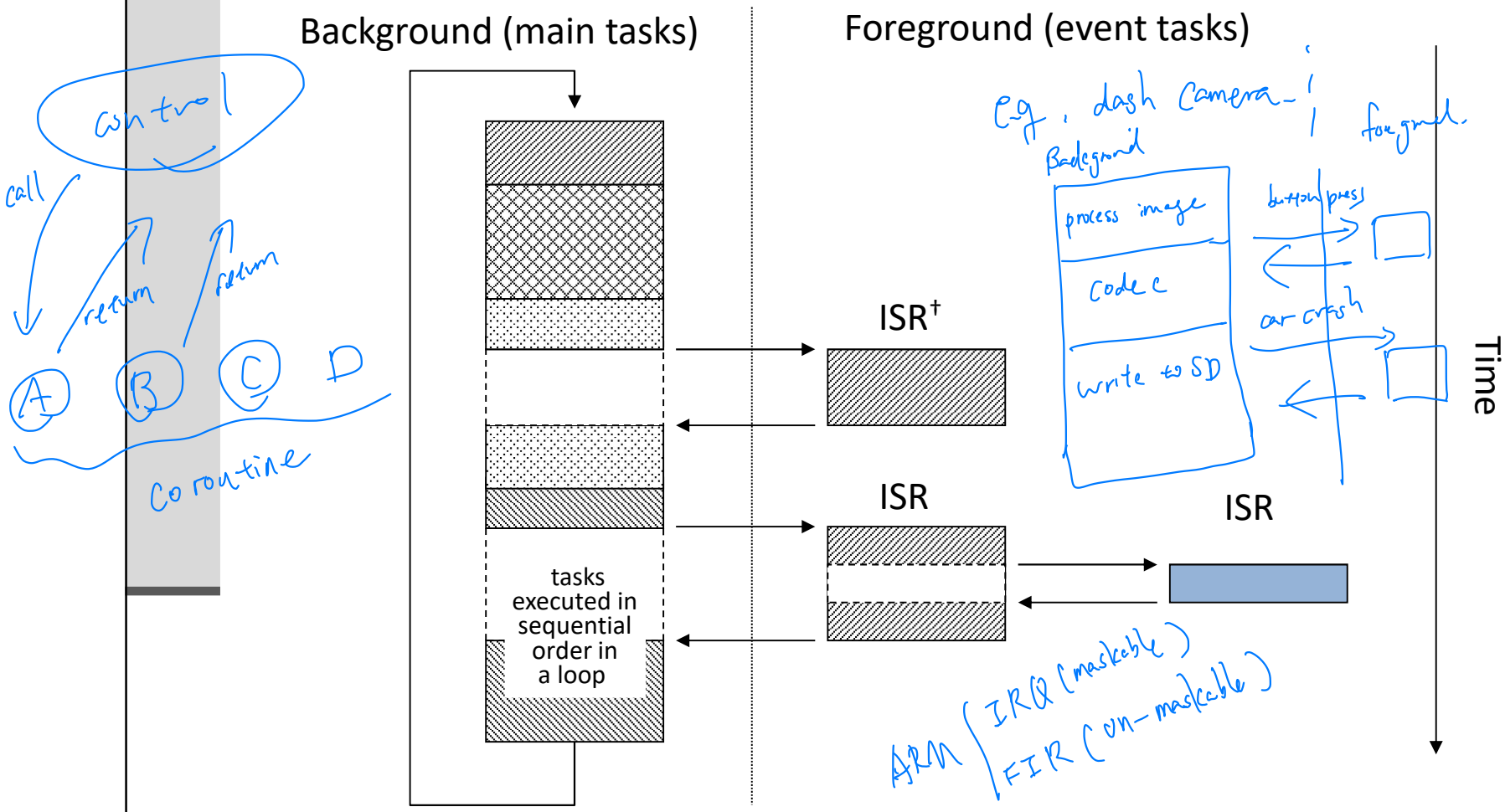
Multiple threads is made possible by the capability of timer interrupt.

- No virtual address (swap memory)

eg. BSD Unix / MacOS / Windows

Machine mode
Supervisor mode
) privileged level
User mode

Foreground/Background Systems

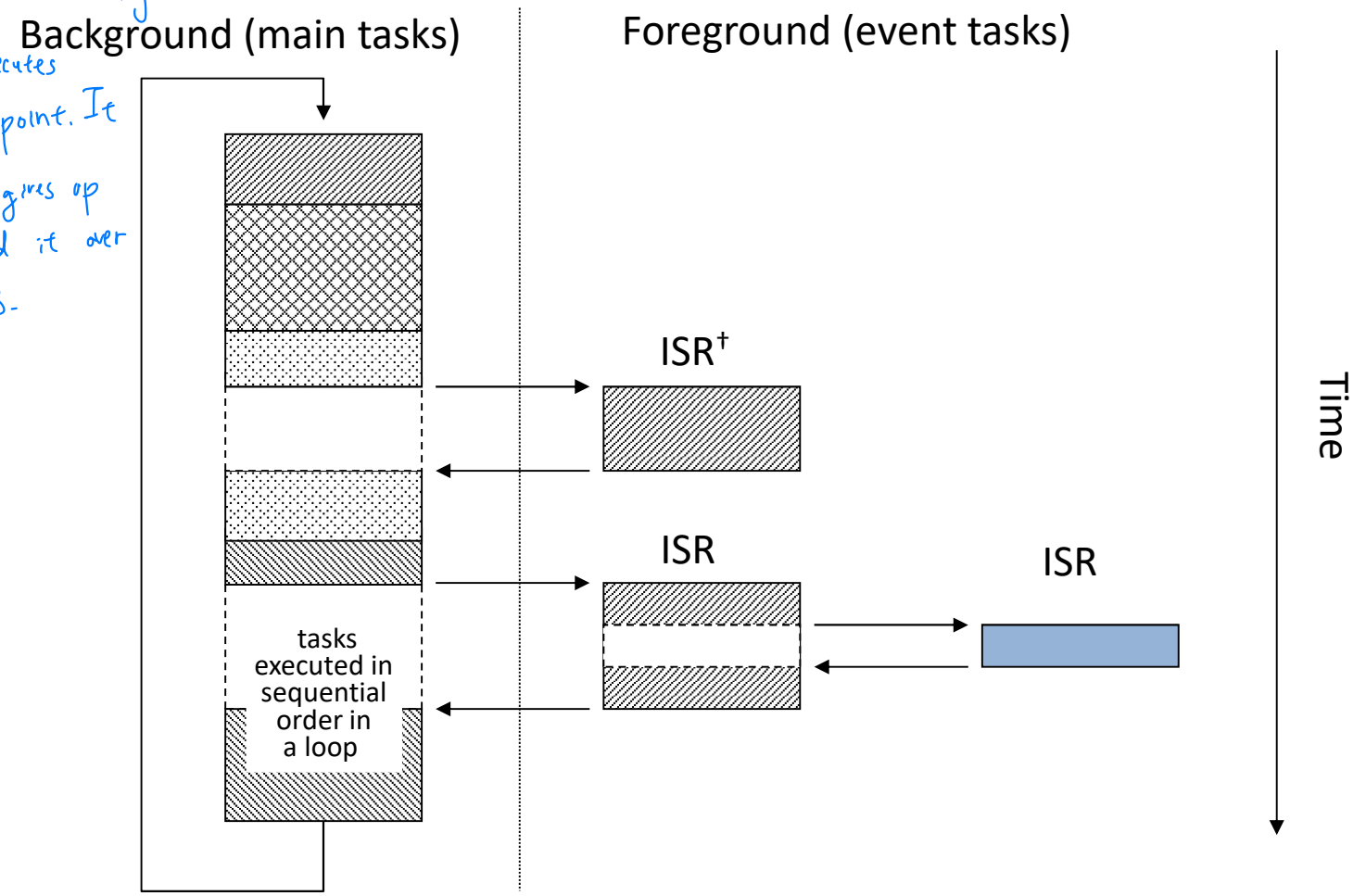


† ISR stands for Interrupt Service Routine

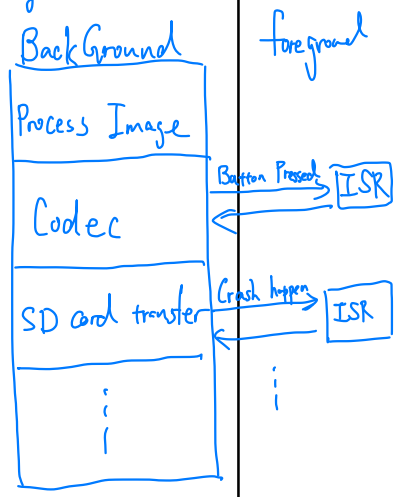
Foreground/Background Systems

process run in backgrounds are called co-routine.

process A executes to a checkpoint. It "voluntarily" gives up execution, hand it over to process B.



e.g dash camera



† ISR stands for Interrupt Service Routine

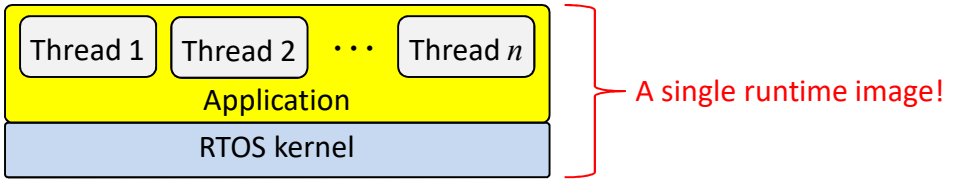
ISR could belong to different levels to signify different importance

eg. ARM { IRQ (maskable IRQ)
FIR (un-maskable IRQ)

RTOS

RTOS is similar to a library for the process

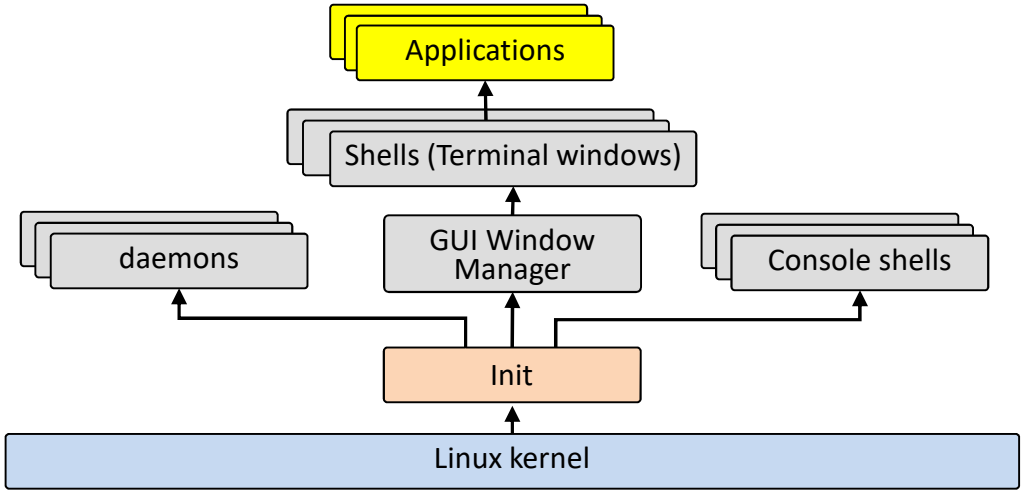
- Popular RTOSs
  - ITRON, μC/OS-III, Nucleus, FreeRTOS, …, etc.
    (ITRON: pioneer; Nucleus: Xilinx uses)
- RTOS Characteristics
  - Often linked with the application into a single binary image
    - One application, multiple threads, plus event-triggered ISRs
  - All threads sharing the same address space
  - Low jitter for periodic scheduling of each thread
  - Minimal interrupt latency
  - Minimal context-switching overhead



Full OS Kernel

- Full OS Characteristics
  - Requires virtual memory support (MMU hardware unit)
  - Runs multiple processes, each has its own address space
    - Each processes can have multiple threads in the same space
  - The first process (`init` under a Linux system) spawns system processes, a shell, and often the entire GUI processes

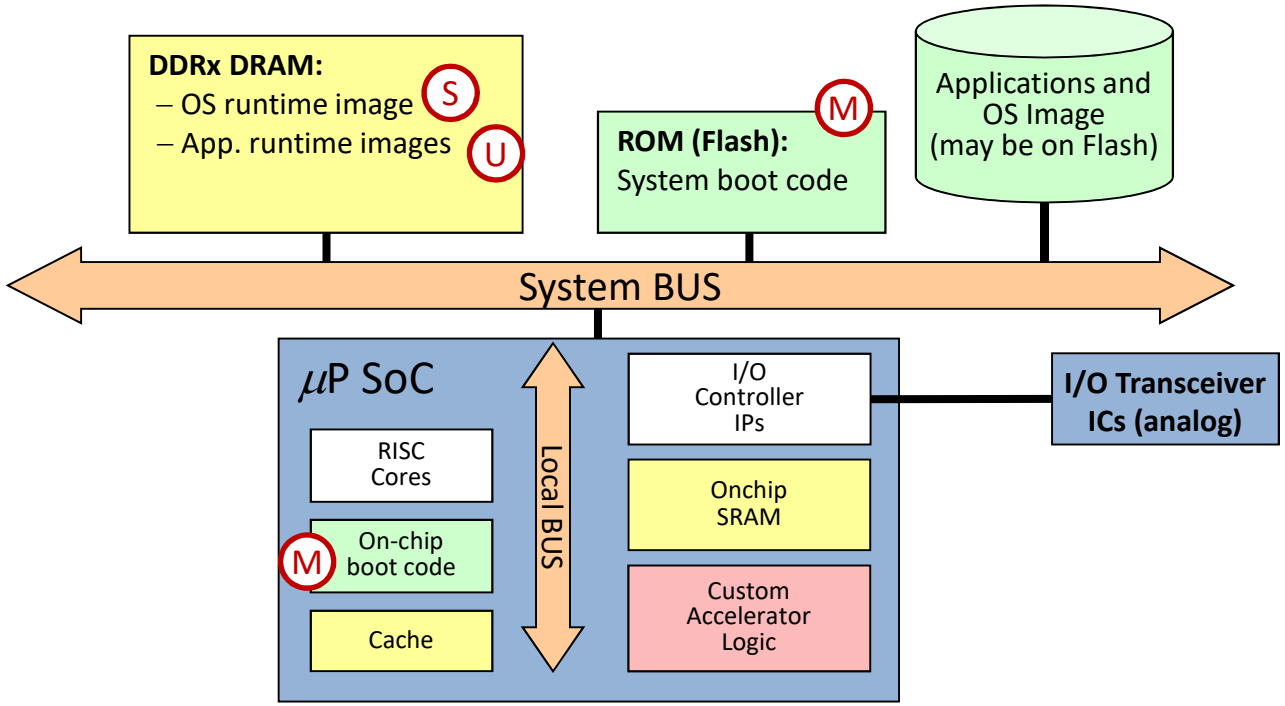
daemon process
eg. logging process.
network daemon



Software Privilege Levels

- RISC-V supports three execution privilege levels:
  - M – machine mode
  - S – supervisor mode
  - U – user mode

Some processors do not distinguish between these two.

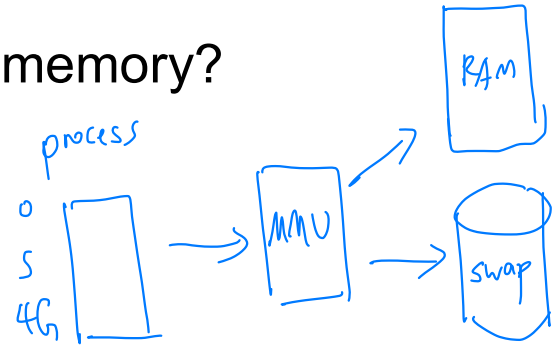


OS Kernel Components

OS is designed to allocate precious H.W resources to process.

- Process Management
  - Determines who gets to use the processor cores?
- Memory Management (Management of heap + stack)
  - Determines who gets to use the runtime memory?
- File System
  - Determines how to retrieve/store data?
- I/O (Sub)-system
  - Determines how to talk to the peripherals?
  - This can be part of the file system (e.g. Unix)
- *Application Runtime (GUI, Events, Comm, … etc.)?*

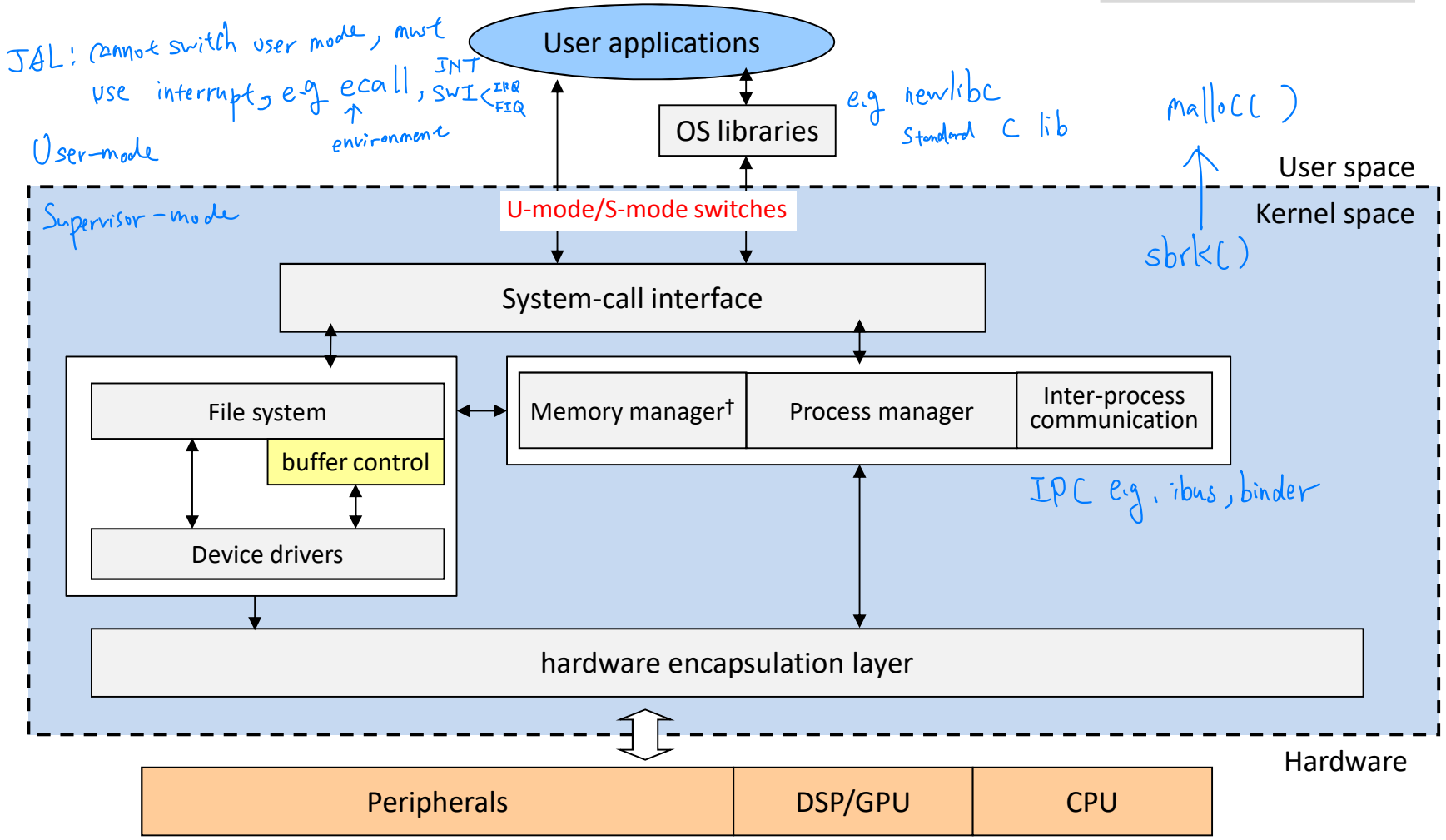
i-node (Linux) → EXT1 ---
FAT (Windows) → NTFS ---



I/O system & file system I/O shall be manage separately, but to linux, they can have unified Interface.
→ by virtual file system : open( )
(system call) write( )
⋮
ioctl( )

Java VM (JVM) : virtualization of CPU + OS

Unix-Like OS Architecture



† Memory manager may includes virtual memory (VM), and dynamic linking-loading (DLL) support usually needs VM.

Microprocessor Support for OS

- To execute a full OS kernel, we need the following support from the microprocessors:
  - Interrupt support for event-driven tasks → Only need this for F/B systems
  - Timer interrupt for multi-threading† CLINT (Core local Interrupt)
  - Software trap for system call
  - Atomic ops for concurrent sync.
  - Privilege level support for protection
  - MMU and virtual memory for
    - multi-processing†
    - dynamic linking loading library calls (DLL)
    - memory protection

Need these for RTOS

μClinux (MicroController Linux): is a linux version without MMU.

† In this context, multi-threading and multi-processing has different meaning. Different threads shares the same address space while different processes do not.

RISC-V Control and Status Registers

- CSRs store the system state of a RISC-V machine
  - Each privilege mode has its own set of CSRs
  - CSRs have their own 12-bit address space, accessible only through CSR read/write instructions:

```
csrrw rd, csr, rs     ; Read-and-write CSR. rd ← csr ← rs
csrrs rd, csr, rs     ; Read-and-set CSR.   rd ← csr ← csr | rs
csrrc rd, csr, rs     ; Read-and-clear CSR. rd ← csr ← csr ^ rs

csrrwi rd, csr, imm   ; Read-and-write CSR. rd ← csr ← imm
csrrsi rd, csr, imm   ; Read-and-set CSR.   rd ← csr ← csr | imm
csrrci rd, csr, imm   ; Read-and-clear CSR. rd ← csr ← csr ^ imm
```

  - These instructions are atomic and zero-extend the `csr` or `imm` values to XLEN bits
- You can define user CSRs using unused addresses

Important M-mode CSRs (1/3)

- Identification CSRs
  - `misa`, `mhartid`, `mvendorid`, `marched`, `mimpid`
- Machine status CSR similar to traditional machine state register
  - `mstatus` is the most important CSR. For RV32, it has the following fields:

Bits	Field Name	Description	Bits	Field Name	Description
0	UIE	U-mode Interrupt Enable	[14:13]	FS	Floating Point State
1	SIE	S-mode Interrupt Enable	[16:15]	XS	U-mode Extension State
2	Reserved		17	MPRIV	Modify Privilege
3	MIE	M-mode Interrupt Enable	18	SUM	Permit S-mode User Mem. Access
4	UPIE	U-mode Previous Interrupt Enable	19	MXR	Make Executable Readable
5	SPIE	S-mode Previous Interrupt Enable	20	TVM	Trap Virtual Memory
6	Reserved		21	TW	Timeout Wait
7	MPIE	M-mode Previous Interrupt Enable	22	TSR	Trap SRET
8	SPP	S-mode Previous Privilege	[23:30]	Reserved	
[10:9]	Reserved		31	SD	State Dirty
[12:00]	MPP	M-mode Previous Privilege			

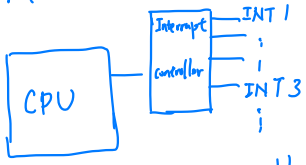
Important M-mode CSRs (2/3)

- Timer CSRs (both are of 64 bits)
  - `mtime`
    - A counter register that increments periodically
    - The counter frequency can be freely defined by the system designer, but exposed through a system software header file
  - `mtimecmp`
    - The threshold for generating interrupt for the OS kernel
    - A timer interrupt will be triggered whenever `mtime` ≥ `mtimecmp`
    - To generate periodic timer interrupt, in the ISR, you must set `mtime` to `0` or `mtimecmp` to `mtime + time_quantum`

mtime & mtimecmp are in clint.v

Important M-mode CSRs (3/3)

CPU does not have many pins for interrupt
e.g ARM : 2 ports IRQ, FIR
Aquila : 3
An interrupt controller is needed.
ARM:



or an interrupt jump table, x86

- Interrupt CSRs
  - `mepc` – stores interrupted program counter value
  - `mcause` – stores the cause of the interrupt
  - `mtvec` – stores the ISR entry point information
    - `mtvec` is composed of `{BASE[31:2], MODE[1:0]}`
    - In direct-mode, `BASE` is the ISR address
    - In vector-mode, the `BASE+4×mcause` is the ISR address
- Memory protection CSRs
  - For systems that only needs memory protection, not virtual address translation, use `pmpcfg0` ~ `pmpcfg15` to do the job
  - Sets up to 16 memory regions with `RWX` attributes for each
  - S- and U-mode programs are restricted by these attributes

CSRs in Supervisor Mode

- S-Mode has the least number of CSRs

Setting medeleg/mideleg will delegate a S-/U- mode trap to S-mode.
Setting sedeleg/sideleg will delegate a U-mode trap to U-mode.

Number	Privilege	Name	Description
Supervisor Trap Setup			
0x100	SRW	sstatus	Supervisor status register.
0x102	SRW	sedeleg	Supervisor exception delegation register.
0x103	SRW	sideleg	Supervisor interrupt delegation register.
0x104	SRW	sie	Supervisor interrupt-enable register.
0x105	SRW	stvec	Supervisor trap handler base address.
0x106	SRW	scounteren	Supervisor counter enable.
Supervisor Trap Handling			
0x140	SRW	sscratch	Scratch register for supervisor trap handlers.
0x141	SRW	sepc	Supervisor exception program counter.
0x142	SRW	scause	Supervisor trap cause.
0x143	SRW	stval	Supervisor bad address or instruction.
0x144	SRW	sip	Supervisor interrupt pending.
Supervisor Protection and Translation			
0x180	SRW	satp	Supervisor address translation and protection.

Pointer to the S-mode context of the current hart.

↳ points to MMU table

Preemptive vs. Non-Preemptive OS

- There are two types of multi-tasking:
  - Non-preemptive: a task gives up CPU voluntarily, the OS kernel cannot stop it by force (a.k.a. cooperative multi-tasking)
  - Preemptive: the OS can force a task to pause and put it in a queue waiting for CPU usage
- For preemptive multi-tasking, timer interrupts are used to switch from one task to anther
  - The behavior is called context-switching

Context Switching (CS)

- The system hardware periodically generates timer interrupts, forcing the current task to go into the ISR
  - For RISC-V systems, a Core Local Interrupt (CLINT) module is used to generate the timer interrupts for multi-tasking
- The ISR performs the following things to switch from one task to another
  - Store the current CPU state† into a task control block (TCB)
  - Loading the next task state from its TCB into CPU state
  - Set the program counter to resume the newly loaded task
- Whether a CS will be carried out and the selection of the next task is the decision of the task manager

† CPU states means the content of user registers, the program counter, and any other special purpose registers (such as the page table base register `sptbr`).

OS Kernel System Call

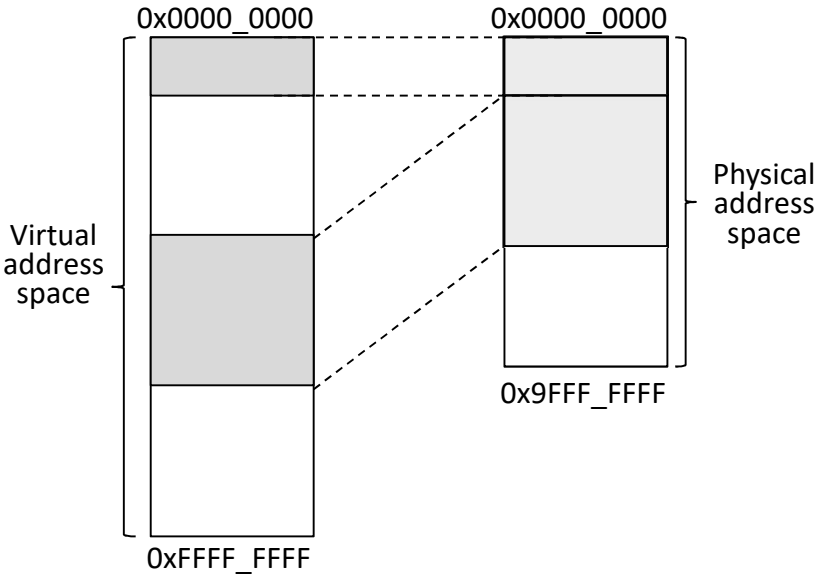
- If an application needs to call an OS kernel for some services, can we simply use `jal` or `jalr`?
  - For security reasons, user applications and OS kernel functions should be running at different privilege levels
  - A system call should allows changing of execution modes
  - The behavior of a high-privilege function shall be fixed and can not be modified at runtime
- Software trap instructions are used to provide system call interface since 1980's

System Call in RISC-V

- RISC-V `ecall` instruction triggers an exception to the system call interface
  - The CSR `xcause` is set to 8 (U-mode), 9 (S-mode), or 11 (M-mode) for a “environmental call”
  - An interrupt is generated to promote the privilege level (or the same for M mode)
  - The CSR `xtvec` is used to compute the system call handling routine
- Register convention for `ecall`:
  - System call number in register a7
  - Arguments in a0 ~ a6
  - Return results in a0, a1, or stack
  - No other data registers will be modified by the system call

RISC-V Virtual Memory Support

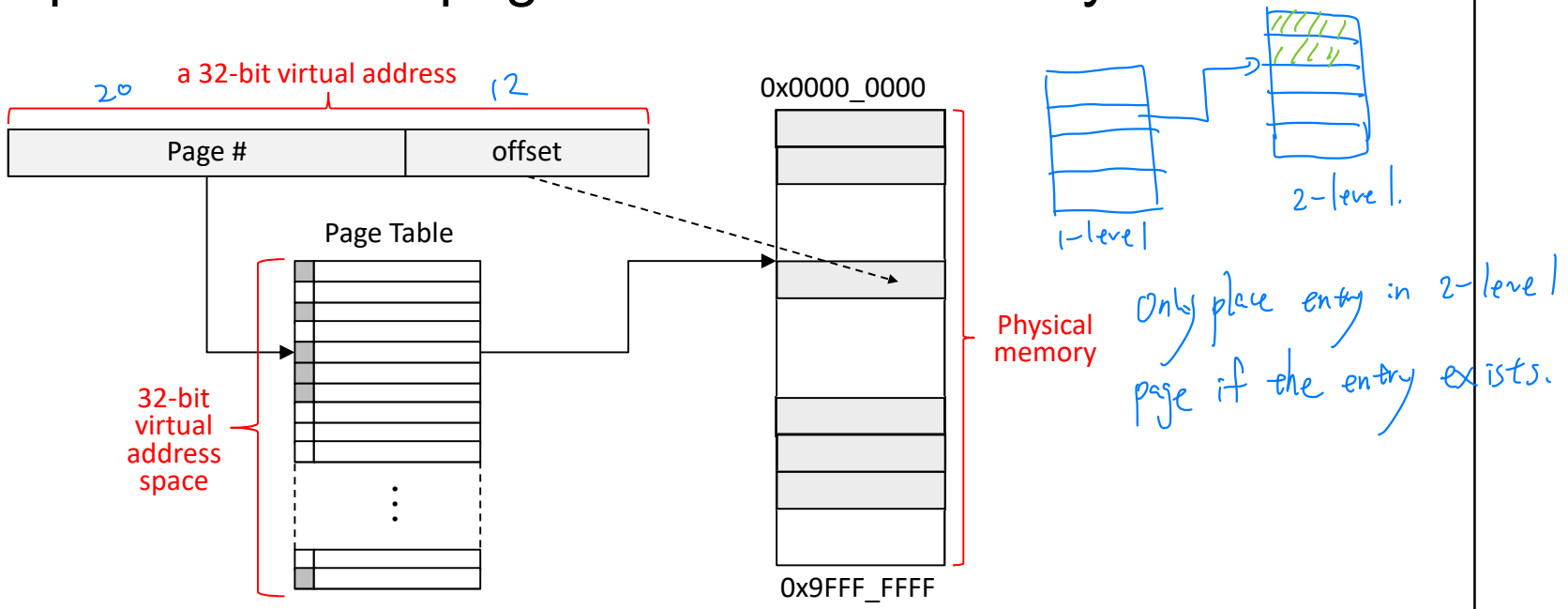
- RISC-V supports paged virtual memory in S-mode
  - Minimal page size is 4KB
  - Base address of the page table is stored in `sptbr`
- Virtual memory type (stored in `satp`)
  - SV32 (for RV32)
    - 32-bit address, 2-level
  - SV39 (for RV64)
    - 39-bit address, 3-level
  - SV48 (for RV64)
    - 48-bit address, 4-level



Memory Management Unit (MMU)

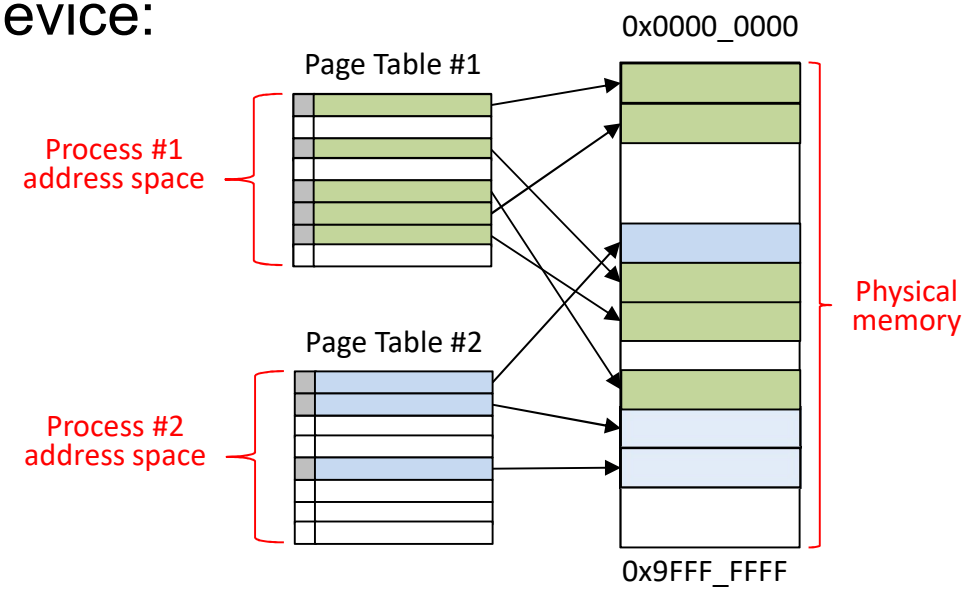
With MMU, Each Process could have a virtual memory space.

- MMU handles virtual-to-physical address translation on a page-by-page basis
  - Segment-based translation are less flexible and obsolete
- Each process has a page table maintained by the OS



Page Table Lookup

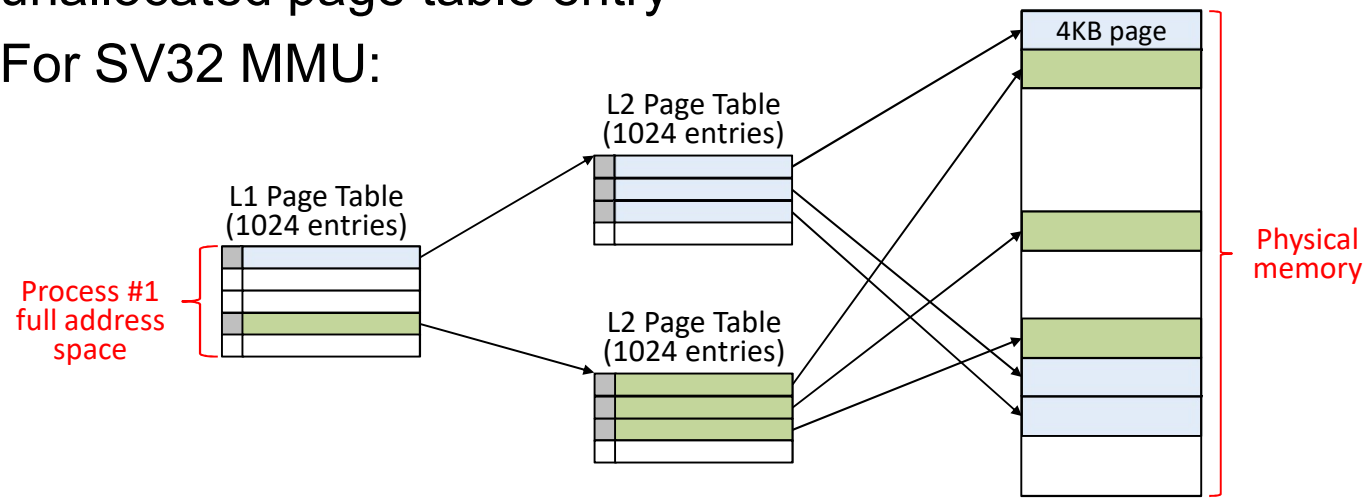
- At runtime, MMU must translate the virtual address of a process to a physical address before sending it to the memory device:



- Page tables can be huge → need a Translation lookaside buffer (TLB) cache in MMU to speed up table lookup

Multi-Level Page Table

- If each page is 4KB, a 32-bit virtual address space requires a 4MB page table → one for each process
- To reduce table size, a multi-level scheme is used:
  - Mandatory creation of level-1 page table
  - On-demand allocation of level-2 (and above) page tables
  - CPU triggers a page-fault exception when accessing an unallocated page table entry
  - For SV32 MMU:



Integration of Cache and MMU (1/2)

- Should caches use virtual or physical address?
  - CPU send memory requests in virtual addresses
- Cache indexing using physical address
  - Extra latency for cache access due to address translation
  - A cached data may be slow to access due to TLB miss
- Cache indexing using virtual address
  - Same address for different processes points to different data $\rightarrow$ the tag must be augmented to differentiate cache blocks
  - Shared physical data memory via different virtual addresses will be cached multiple times $\rightarrow$ bad practice anyway

Integration of Cache and MMU (2/2)

- Virtually Indexed Physically Tagged (VIPT): use virtual address for line index, and physical address for tag
  - Need proper cache-line/MMU-page alignment so that the index bits are the same for both virtual and physical addresses
  - Limited cache size. Each cache way is constrained by the page size, e.g., 32KB maximum for 8-way cache with 4K page
  - Fast, typically used for L1 caches
- Physically Indexed, Physically Tagged (PIPT):
  - Longer latency due to extra memory translation process
  - Typically used for L2/L3 caches

Statistically, designs with L2 & L3 cache performs better than a big L2 cache.

Runtime Loader and MMU

- Virtual address space per process can facilitate dynamic loading of executables/libraries
  - Not all memory accesses can use PC as a base register
- Without an MMU, a system must:
  - Compiler “tags” all absolute addresses in the executable binary w.r.t. base addresses
  - Runtime loader adjusts all the tagged addresses based on the physical loading address of each section
- Major disadvantage:
  - on-demand swapping of code/data sections are inefficient

The Boot Process of a System

- A modern computing system begins its life-cycle from the in-core boot code:
  - All processor cores except the #0 core execute a “wait-and-indirect-jump” boot code
  - Core #0 initializes the shared resources and page tables of all cores, and jump to the 1st-level boot code in ROM
- The first-level boot code supports minimal HW devices
  - Just enough to load the OS kernel or 2nd-level boot code
  - If a 2nd-level boot code is used, the OS kernel image can be stored in remote servers or exotic devices
- Other cores are waken up and initialize their private resources during OS kernel boot