CE5045
Embedded System Design

Embedded Software Architecture

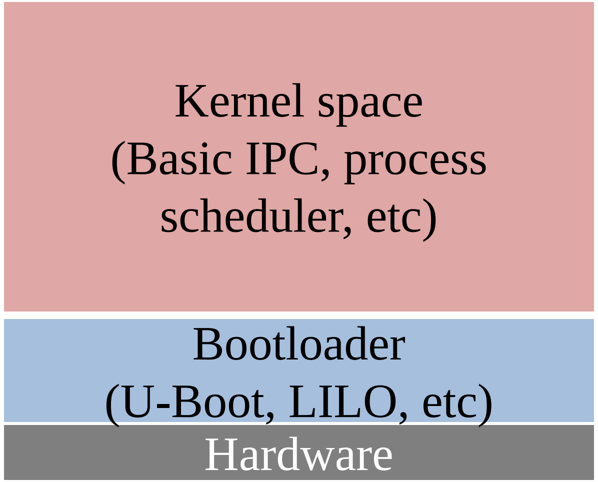
https://github.com/tychen-NCU/EMBS-NCU

Instructor: Dr. Chen, Tseng-Yi

Computer Science & Information Engineering

Outline

- How to Boot an Embedded System (ES)
    - ✓ Grub for x86 Architecture
    - ✓ U-boot for ARM Architecture
    - ✓ How to Implement a Bootloader in an Embedded System.
- Microkernel for Embedded System
    - ✓ What is Process?
    - ✓ The Process Concept in an Embedded System
    - ✓ The Practical Knowledge of Process Management

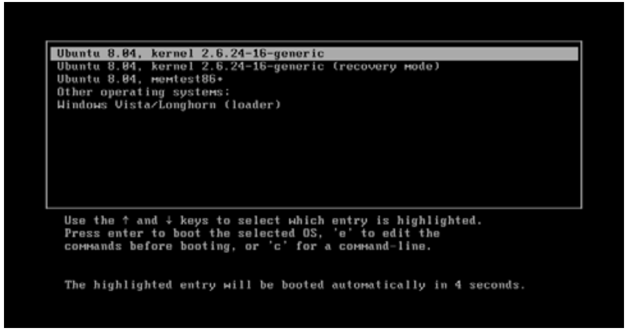


Outline

- How to Boot an Embedded System (ES)
  - ✓ Grub for x86 Architecture
  - ✓ U-boot for ARM Architecture
  - ✓ How to Implement a Bootloader in an Embedded System.
- Microkernel for Embedded System
  - ✓ What is Process?
  - ✓ The Process Concept in an Embedded System
  - ✓ The Practical Knowledge of Process Management

What is GRUB?

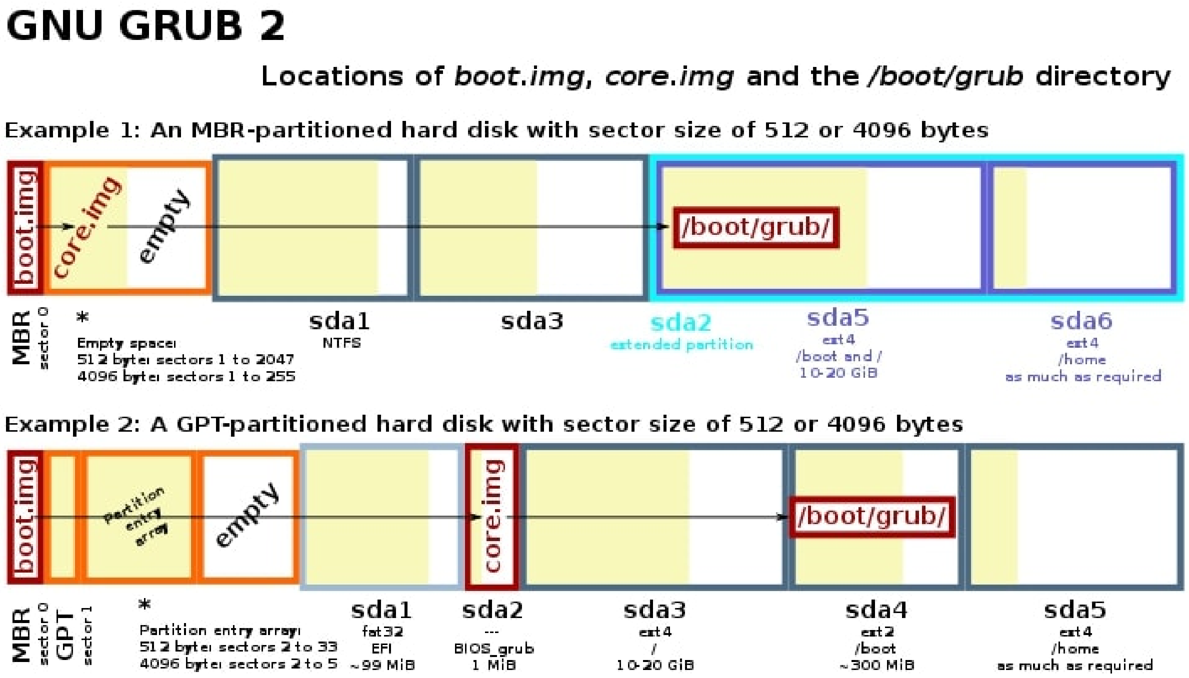
- Before Grub
  - ✓ What is boot sector?
    - A boot sector is generally the first sector of the hard drive that is accessed when the computer is turned on.
    - Master boot record (MBR) or GUID partition table (GPT)



- So Grub …
  - ✓ Is a piece of software that exists in the MBR or GPT
  - ✓ Allows a user to opt between multiple Operating systems that are installed on one or more drives existing on the computer

How to Boot with Grub

- Depending upon the boot sector that is either the master boot record or the GUID partition table the physical allocation of the GRUB change.



The Image Files for Grub

- boot.img
  - Its size is 446 bytes
  - Is written to the MBR (sector 0)
  - Contains utilities, Operating system, kernel files, diagnostics and various other drivers for the hardware to initiate
- core.img
  - Is written to the empty sectors between the MBR and the first partition
  - Default RAW disk image loaded on the hard disk by the manufacturer
- File system
  - Takes care of the addressing of the data onto a physical sector. Ex: NTFS, FAT, Ex-FAT, HFS etc.

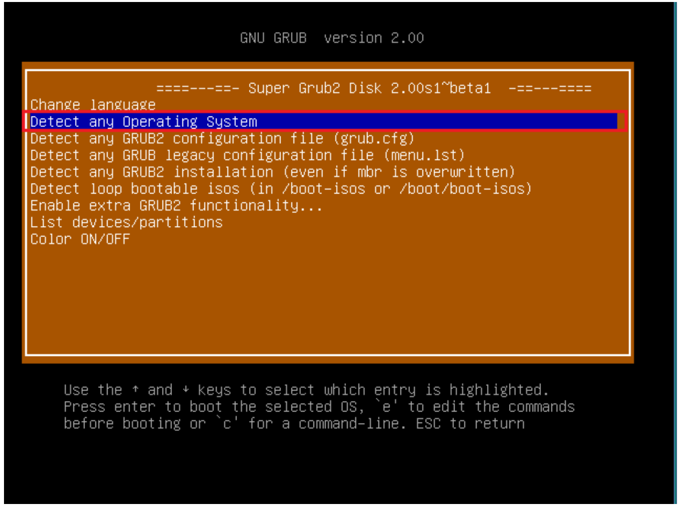
The Functionality of GRUB (I/II)

- Stage 1
  - ✓ Find the boot.img in Master Boot Record (MBR) or GUID partition table (GPT)
  - ✓ At this stage GRUB directs the next stage using an address of the kernel files of various Operating systems

- Stage 1.5
  - ✓ The core.img will load the file needed for configuration with various other modules like file system drivers acquired from boot.img

The Functionality of GRUB (II/II)

- Stage 2
  - ✓ In this final stage a Text-based User Interface is displayed which will enable the user to select between the Operating systems.
  - ✓ You can specify a default Operating system to load after a user define timeout.

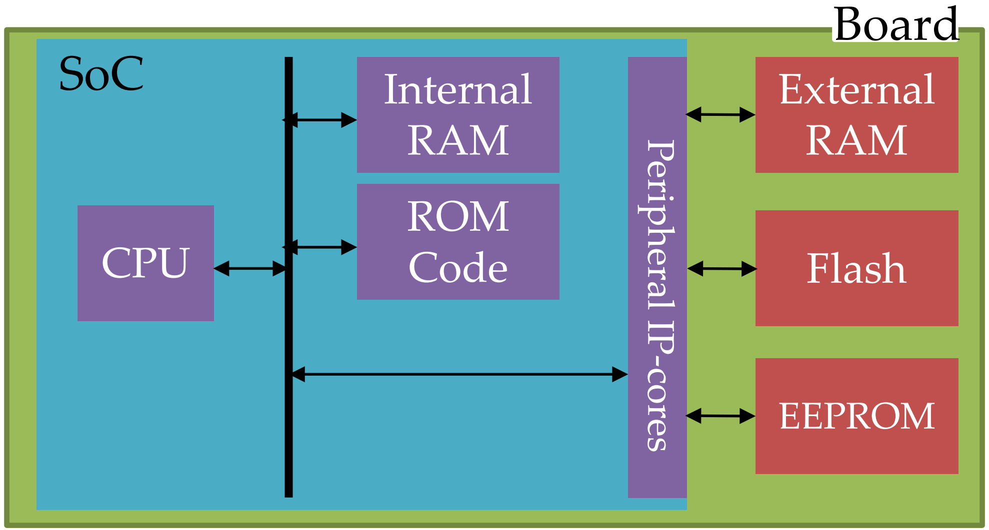


The Differences Between PC and ES

- The bootloader of embedded system is more complicated than that of PC
  - ✓ Embedded systems do not have a BIOS to perform the initial system configuration.

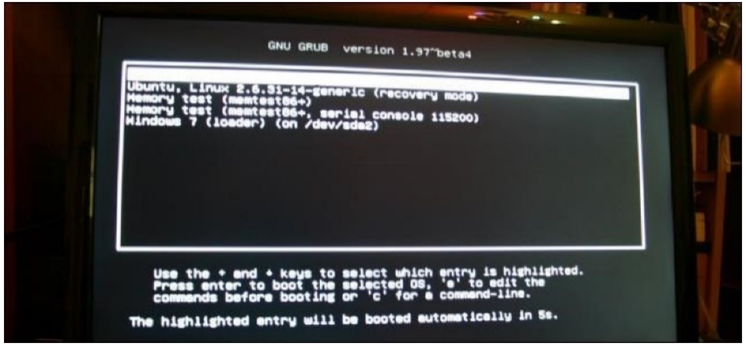
- Bootloader in x86 architecture consists of two parts
  - ✓ BIOS (Basic Input/Output System)
  - ✓ OS loader (located in MBR of hard disk)
    - e.g., LInux LOader(LILO) and GRUB

Embedded Board



What is Bootloader?

- Somebody will say …
  - ✓ The first section of code to be executed after the embedded system is powered on or reset on any platform.
  - ✓ A program that starts whenever a device is powered on to activate the right operating system.

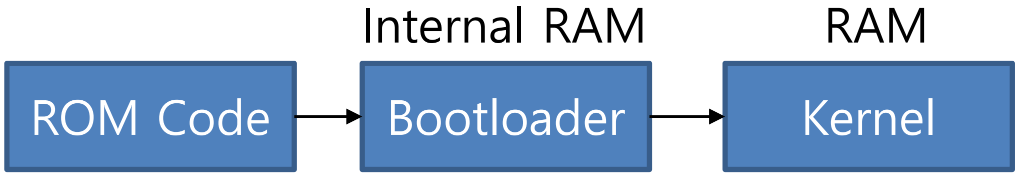


The Purpose of Bootloader

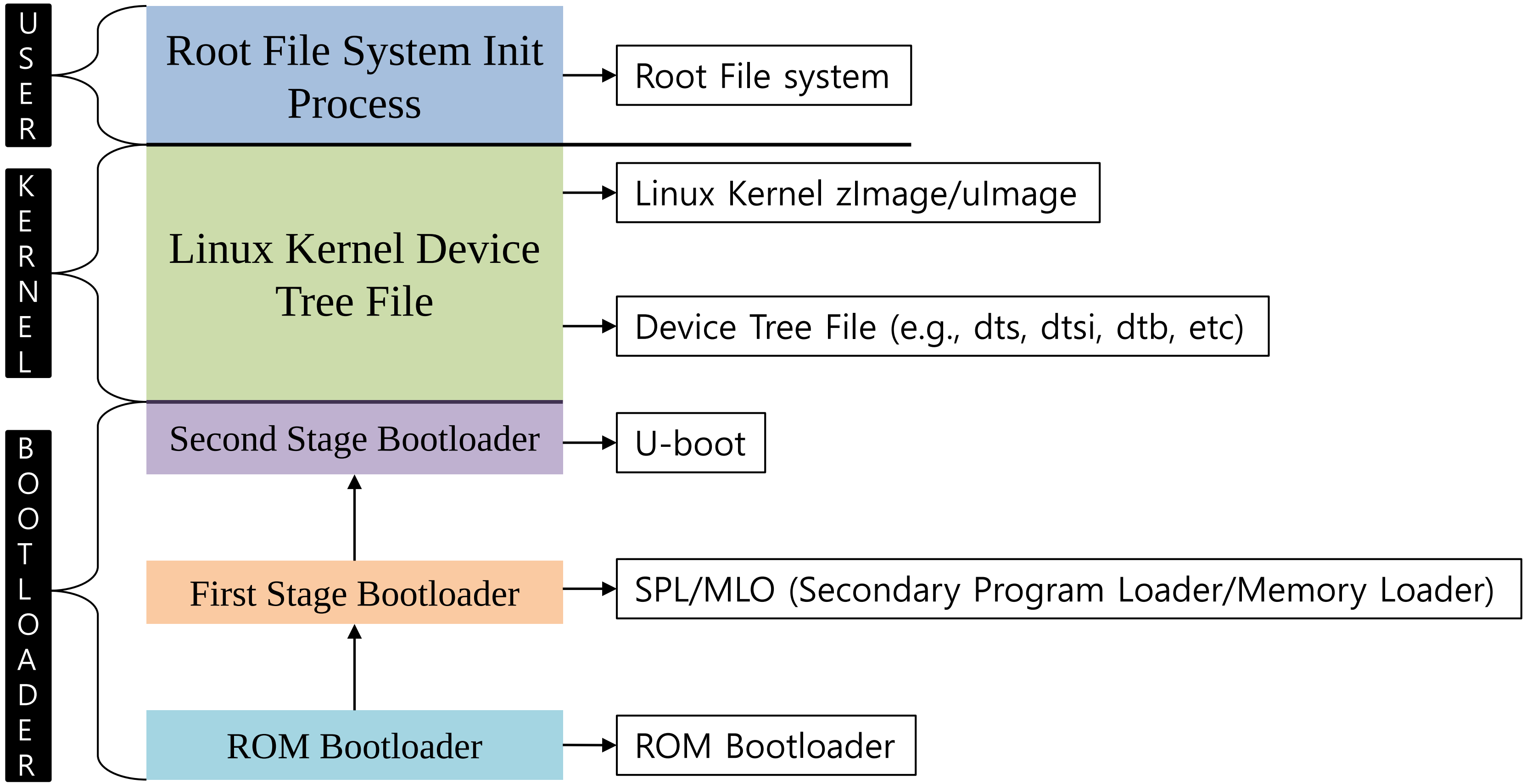
- Bootloader in (embedded) computing systems
  - ROM code has limitations:
    - Doesn’t know about RAM address
    - Doesn’t know about board name
    - Not flexible enough

The Purpose of Bootloader

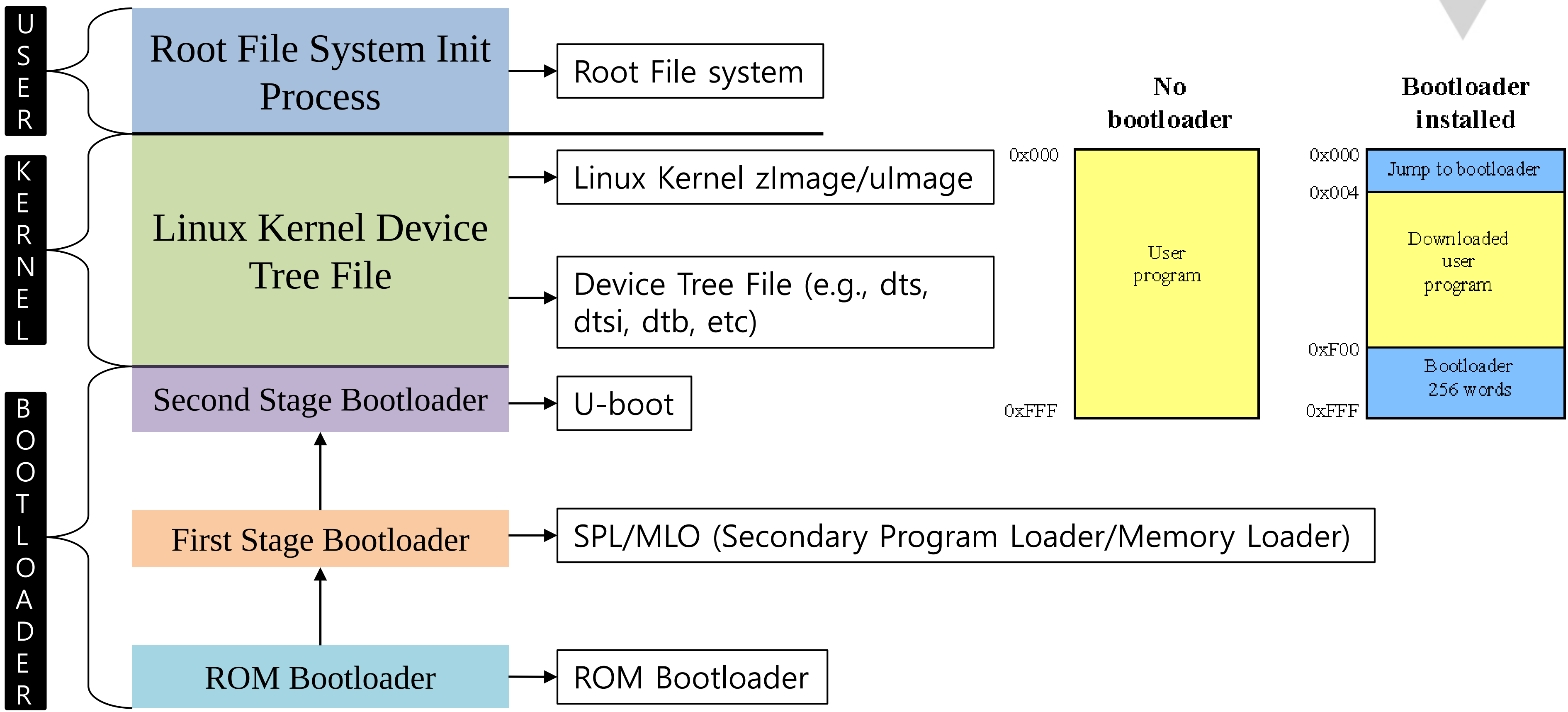
- Bootloader in (embedded) computing systems
  - ✓ ROM code has limitations:
    - Doesn’t know about RAM address
    - Doesn’t know about board name
    - Not flexible enough
  - ✓ Bootloader to the rescue:
    - Reside in flash. But why?
    - Able to configure RAM
    - Knows boot procedure
    - Convenient features



How Does Bootloader Work?



How Does Bootloader Work?



What is U-Boot

- Das U-Boot
  - A GPL'ed cross-platform boot loader
  - Created by Wolfgang Denk

- U-Boot provides out-of-the-box support for hundreds of embedded boards and a wide variety of CPUs including PowerPC, ARM, XScale, MIPS, Coldfire, NIOS, Microblaze, and x86

- Official website
  - https://www.denx.de/wiki/U-Boot/WebHome

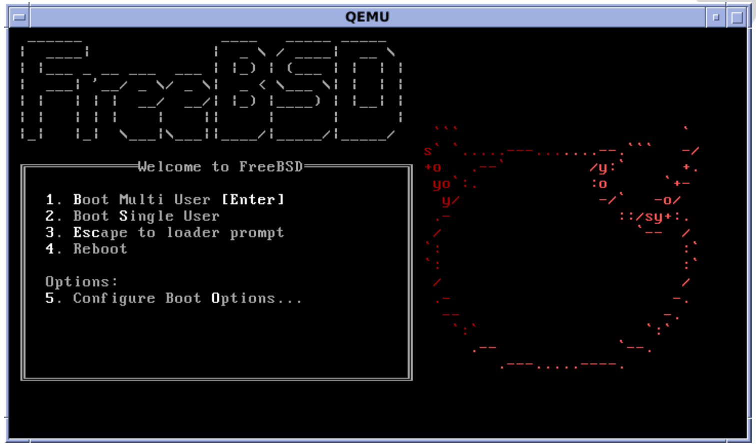
Why U-Boot?

- Bootloader for embedded boards
  - ✓ Popular for Android device
  - ✓ Adoption in automotive
- 13 architectures (including ARM, x86, MIPS, etc.)
- ~300 boards
- Device drivers and lib routines
- Resembles Linux kernel a lot
- Scripting, extensive command set

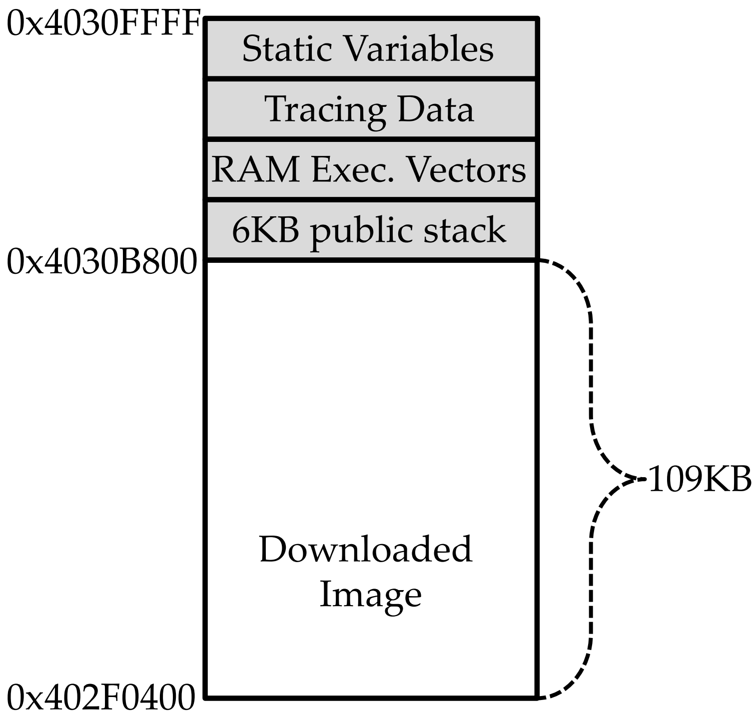
U-Boot

U-Boot Features

- ➢ Boots from various source
  - ✓ Network source
  - ✓ External storage
- ➢ Boots various O.S.s
  - ✓ OpenBSD, NetBSD, FreeBSD,4.4BSD, Linux, SVR4, Esix, Solaris, Irix, SCO, Dell, NCR, VxWorks, LynxOS, pSOS, QNX, RTEMS, ARTOSDevice drivers and lib routines
- ➢ 2 Stage boot (SPL + U-Boot)
- ➢ Falcon mode (SPL only)



Why Two-stage Boot



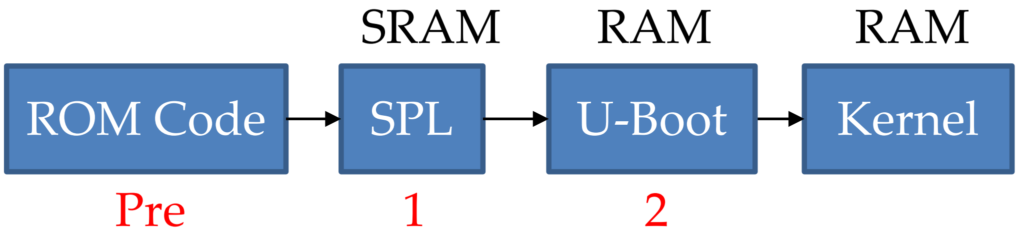
SRAM layout (From AM335x TRM)

But bootloader is so big!

For BeagleBone Black, u-boot.img is 391KB

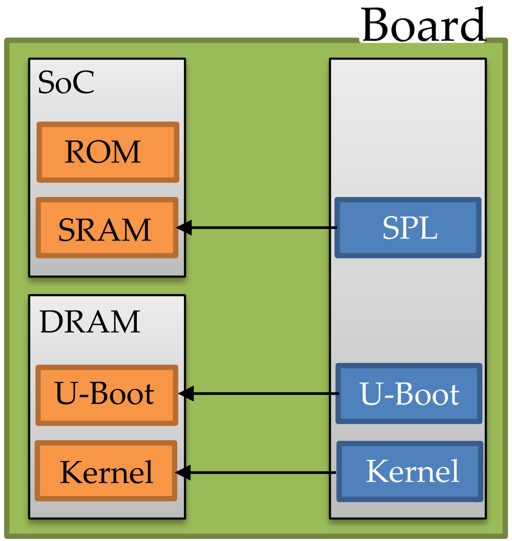
What is Two-stage Boot

An intermediate stage (SPL):



Example: BeagleBone Black

Stage	Size
SPL	75 KB
U-Boot	391 KB



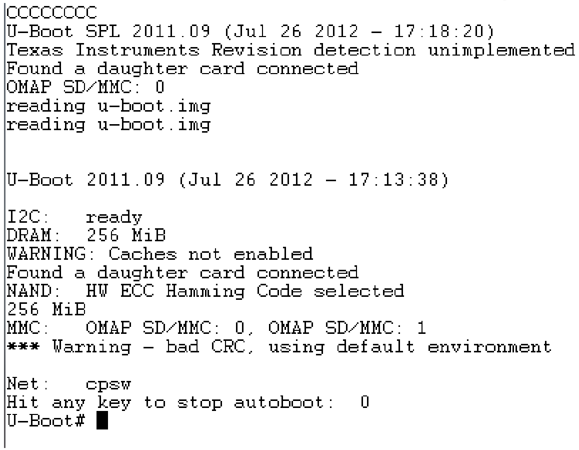
Start-up an Embedded System (I/II)

- Step 1: Load the first instruction from a base address (ROM Boot).
  - ✓ 0x00000000 for ARM
  - ✓ 0xBFC00000 for MIPS
  - ✓ Two main functions
    - Configuration of the device and initialization of primary peripherals
    - Ready device for next bootloader

- Step 2: Secondary program loader (SPL) also referred as to MLO
  - ✓ The first stage of U-boot
  - ✓ To set-up the boot process for the next bootloader stage

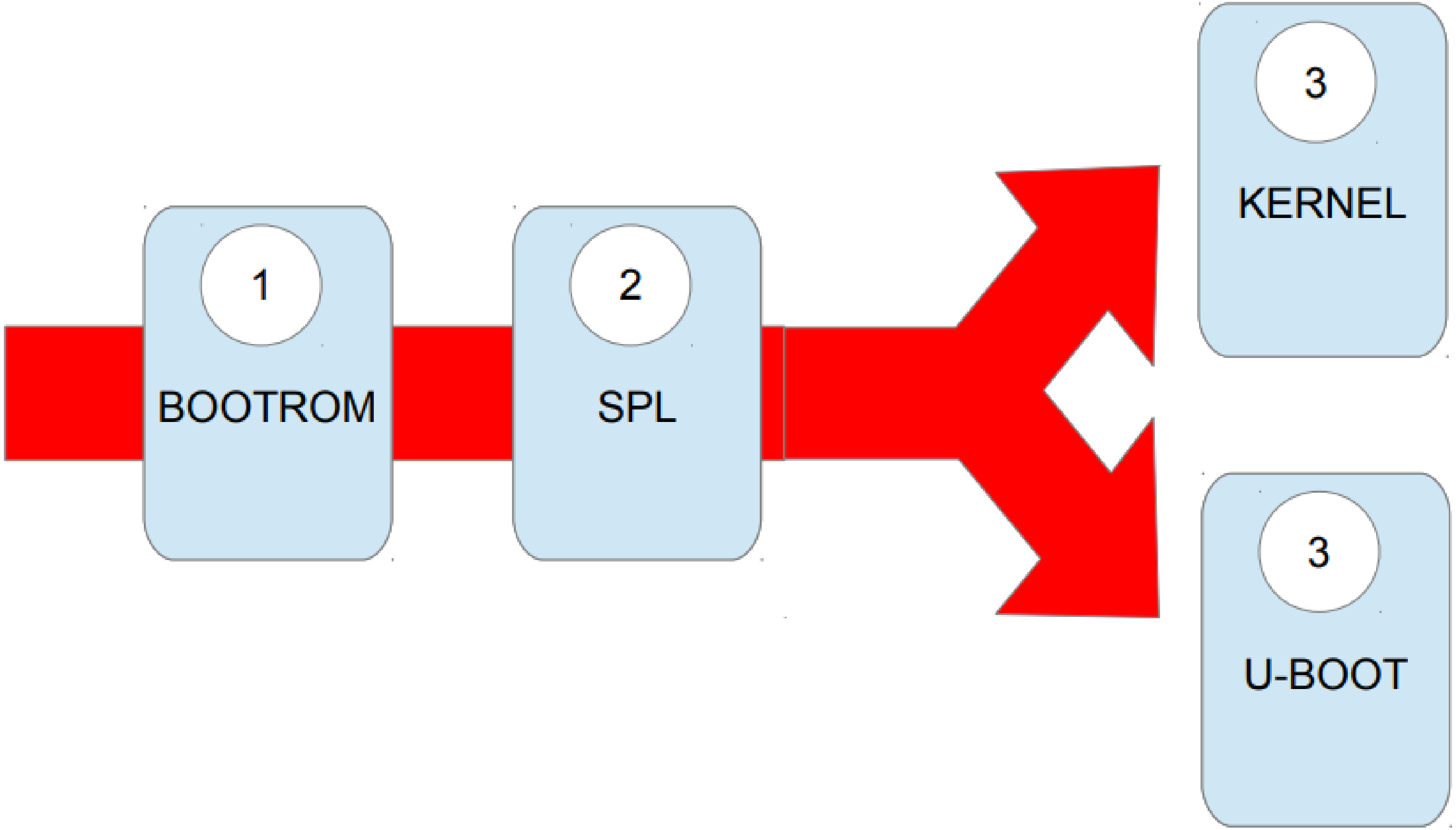
Start-up an Embedded System (II/II)

- Step 3: U-Boot
  - ✓ Powerful command-based control over the kernel boot environment via a serial terminal
  - ✓ Environment variables in the uEnv.txt file
  - ✓ These environment variables can be viewed, modified, and saved using the `printenv`, `setenv`, and `saveenv` commands, respectively.



- Step 4: Kernel Image
  - ✓ uImage is the kernel image wrapped with header info that describes the kernel.

Falcon Boot



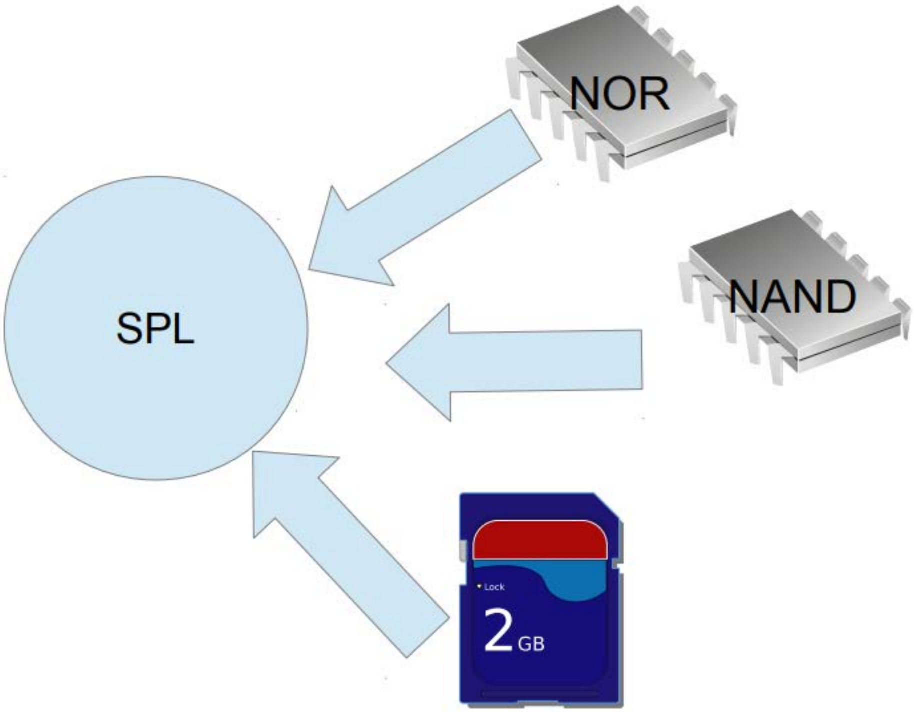
Why Falcon

- Saves time to load U-BOOT
- Saves U-BOOT execution time
- Save time to prepare Boot Parameter Area (legacy kernel)

Supported Boards

- A3m071 (PowerPC MPC 5200)
- Lwmon5 (PowerPC 440 EPX)
- Ipam390 (TI davinci)
- TI OMAP5 boards (dra7xx, uevm) NAND only
- Twister, devkit8000 (TI AM3517)
- Am335_evm (TI AM335x)

Supported Storage



Brief Summary: U-Boot

- The first execution is the fixed ROM Code

Start address = 0x00000000

↓

ROM Code

1. Performs minimal clocks and peripheral configuration.
2. Searches the booting devices for a valid booting image
3. Load the x-loader (or SPL) into SRAM and execute it

- A two-stage boot loader is employed

Brief Summary: U-Boot

- The first execution is the fixed ROM Code

Start address = 0x00000000

ROM Code

1. Performs minimal clocks, memories, and peripheral configuration.
2. Searches the booting devices for a valid booting image
3. Loads the x-loader (or SPL) into SRAM and execute it

Second Program Loader

1. Sets up the pin muxing
2. Initializes clocks and memory
3. Loads the U-Boot into SDRAM and executes it

- In Stage 1
  - ✓ Codes are implemented in assembly language. (For efficiency)
  - ✓ Detect the machine type
  - ✓ Mask all interrupt request
  - ✓ Initialize the indicators (i.e., LED)
  - ✓ Disable data and instruction Cache
  - ✓ Load preparing
    - Prepare RAM space for Stage 2
    - Setup stack
    - Jump to the entry-point of Stage 2

Brief Summary: U-Boot

➢ The first execution is the fixed ROM Code

Start address = 0x00000000

ROM Code

1. Performs minimal clocks, memories, and peripheral configuration.
2. Searches the booting devices for a valid booting image
3. Loads the x-loader (or SPL) into SRAM and execute it

Second Program Loader

1. Sets up the pin muxing
2. Initializes clocks and memory
3. Loads the U-Boot into SDRAM and executes it

U-Boot

1. Performs some additional platform initialization
2. Sets the boot arguments
3. Passes control to the kernel image

Brief Summary: U-Boot

➢ The first execution is the fixed ROM Code

Start address = 0x00000000

ROM Code

1. Performs minimal clocks, memories, and peripheral configuration.
2. Searches the booting devices for a valid booting image
3. Loads the x-loader (or SPL) into SRAM and execute it

Second Program Loader

1. Sets up the pin muxing
2. Initializes clocks and memory
3. Loads the U-Boot into SDRAM and executes it

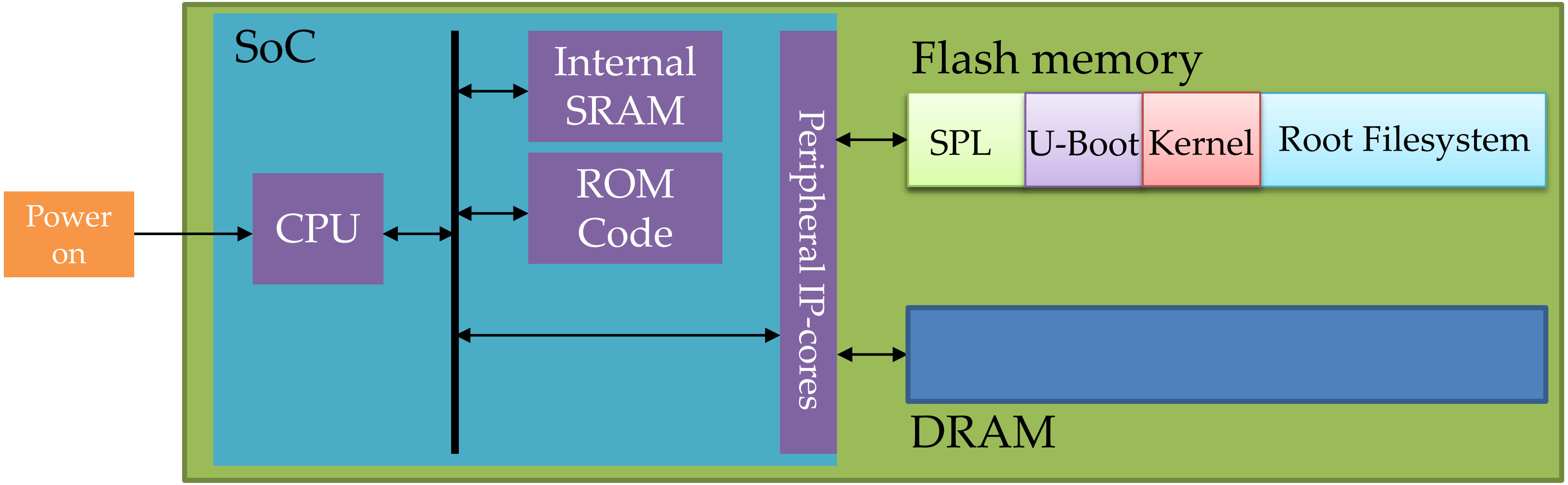
U-Boot

1. Performs some additional platform initialization
2. Sets the boot arguments
3. Passes control to the kernel image

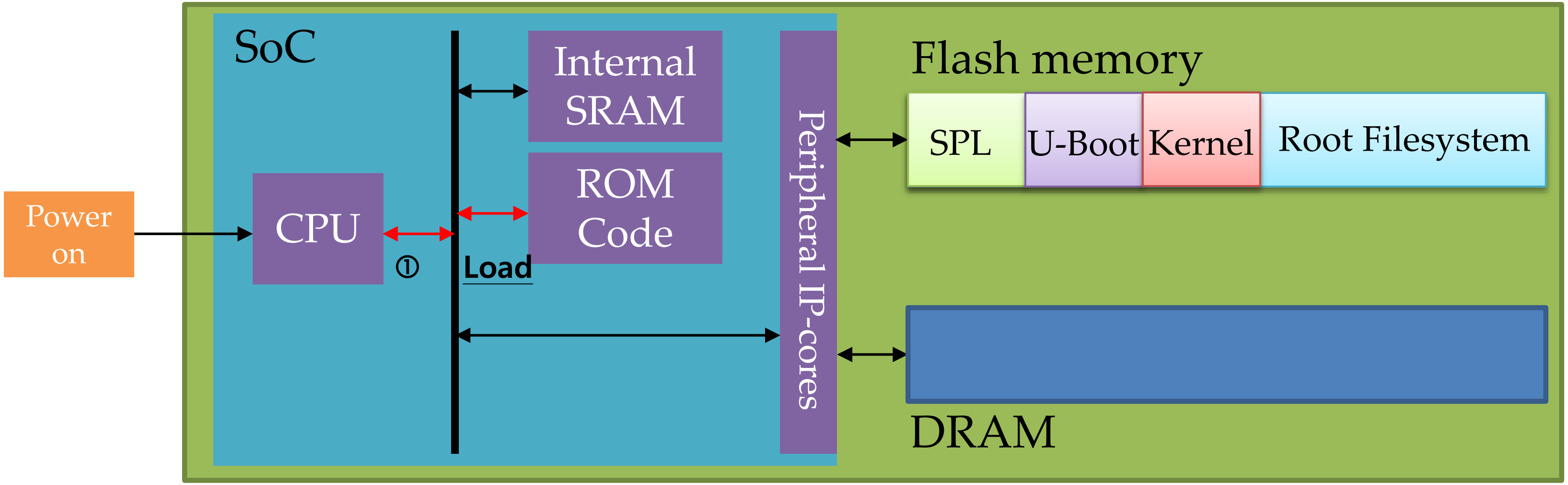
U-Boot

1. Decompresses the kernel into SDRAM
2. Sets up peripherals, e.g., LCD, HDMI, audio, etc.
3. Mounts the Linux filesystem that contains all userspace libraries/ applications

Flow for Boot Process



Flow for Boot Process



- Load bootrom code from 0x00000000

ROM Code for Execution

- You can refer to U-Boot linker script (e.g., \board\samsung\smdk6400\u-boot-nand.ld)

```
SECTIONS
{
    . = 0x00000000; /* Start address 0x0 */
    . = ALIGN(4);
    .text       :   /* start.o first */
    {
      arch/arm/cpu/arm1176/start.o    (.text)
      *(.text)
    }
```

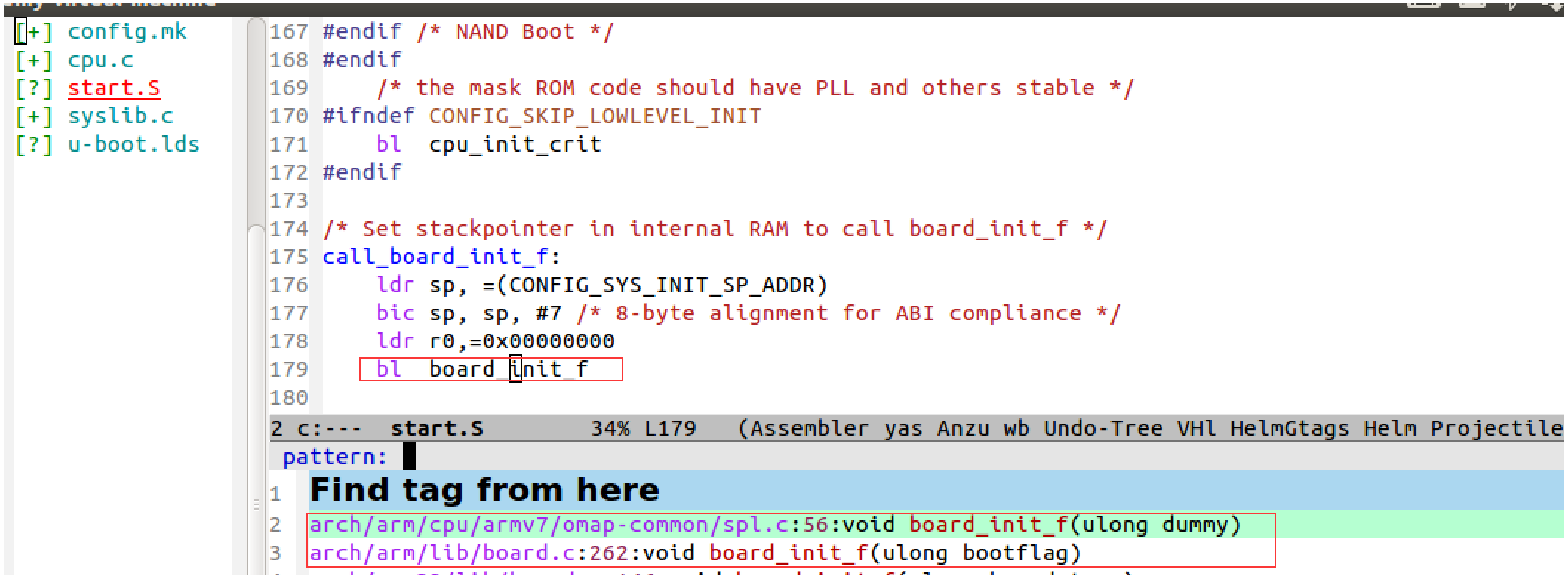
ROM Code for Execution (cont'd)

```
~/uboot_orignal/u-boot/
<+> mx5
<+> omap-common
<+> omap3
<+> omap4
<+> s5p-common
<+> s5pc1xx
<+> s5pc2xx
<+> tegra2
<+> u8500
[+] Makefile
[+] cache_v7.c
[+] config.mk
[+] cpu.c
[?] start.S
[+] syslib.c
[?] u-boot.lds
```

```
LIB = $(obj)lib$(CPU).o

START   := start.o

ifndef CONFIG_SPL_BUILD
COBJS    += cache_v7.o
COBJS    += cpu.o
endif

COBJS   += syslib.o

SRCS     := $(START:.o=.S) $(COBJS:.o=.c)
OBJS     := $(addprefix $(obj),$(COBJS))
START    := $(addprefix $(obj),$(START))

all:     $(obj).depend $(START) $(LIB)

$(LIB): $(OBJS)
    $(call cmd_link_o_target, $(OBJS))

#####################################################
```

ROM Code for Execution (cont’d)

- Board initialization in assembly



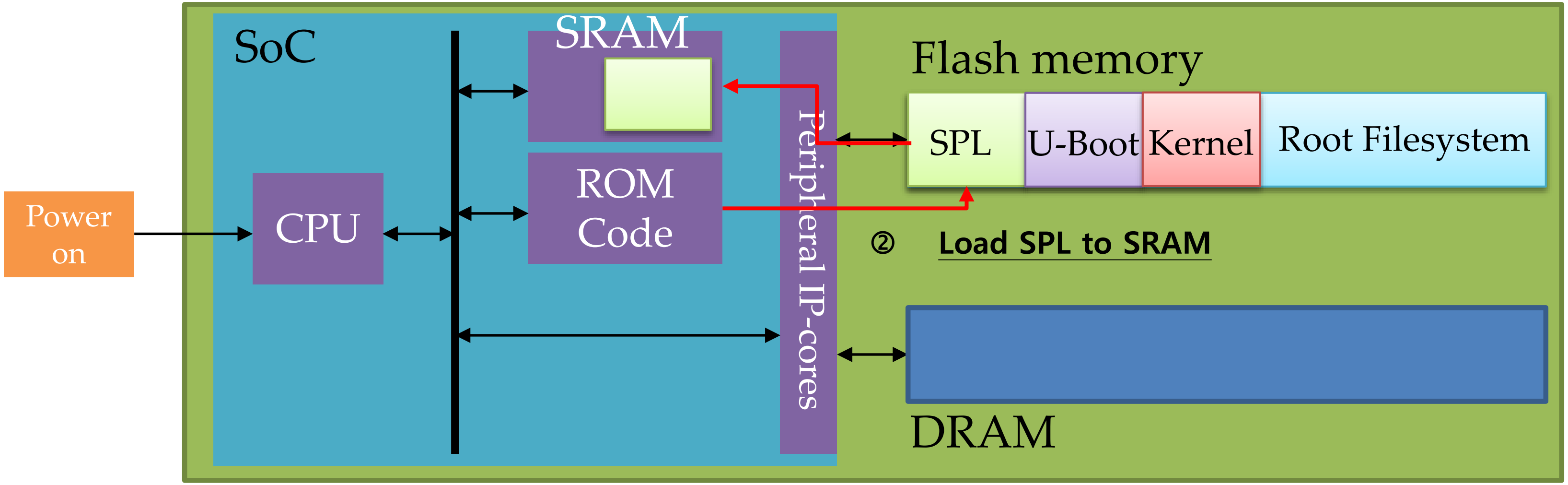
ROM Code for Execution (cont'd)

- Board initialization in C language

```c
void board_init_f(ulong bootflag)
{
    bd_t *bd;
    init_fnc_t **init_fnc_ptr;
    gd_t *id;
    ulong addr, addr_sp;

    /* Pointer is writable since we allocated a register for it */
    gd = (gd_t *) ((CONFIG_SYS_INIT_SP_ADDR) & ~0x07);
    /* compiler optimization barrier needed for GCC >= 3.4 */
    __asm__ __volatile__("": : :"memory");

    memset((void *)gd, 0, sizeof(gd_t));

    gd->mon_len = _bss_end_ofs;

    for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
        if ((*init_fnc_ptr)() != 0) {
            hang ();
        }
    }

    debug("monitor len: %08lX\n", gd->mon_len);
    /*
     * Ram is setup, size stored in gd !!
     */
    debug("ramsize: %08lX\n", gd->ram_size);
```

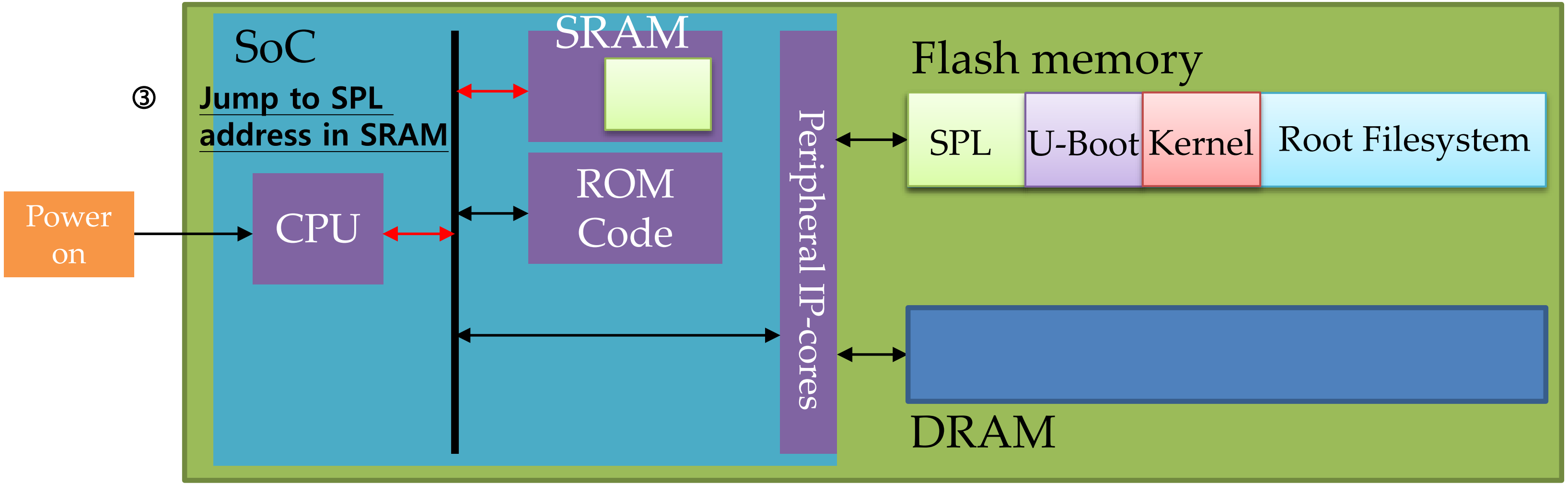
```c
}
void dram_init_banksize(void)
    __attribute__((weak, alias("__dram_init_banksize")));

init_fnc_t *init_sequence[] = {
#if defined(CONFIG_ARCH_CPU_INIT)
    arch_cpu_init,        /* basic arch cpu dependent setup *
#endif
#if defined(CONFIG_BOARD_EARLY_INIT_F)
    board_early_init_f,
#endif
    timer_init,       /* initialize timer */
#ifdef CONFIG_FSL_ESDHC
    get_clocks,
#endif
    env_init,         /* initialize environment */
    init_baudrate,        /* initialze baudrate settings */
    serial_init,          /* serial communications setup */
    console_init_f,       /* stage 1 init of console */
    display_banner,       /* say that we are here */
#if defined(CONFIG_DISPLAY_CPUINFO)
    print_cpuinfo,        /* display cpu info (and speed) */
#endif
#if defined(CONFIG_DISPLAY_BOARDINFO)
    checkboard,       /* display board info */
#endif
#if defined(CONFIG_HARD_I2C) || defined(CONFIG_SOFT_I2C)
```

Flow for Boot Process



- SRAM does not need to be initialized and it is ready after power-on
- Load SPL to SRAM

Flow for Boot Process



- CPU executes the SPL code in SRAM to initialize DRAM and clock.

Set CPU to SVC

- In ./arch/arm/cpu/<HW>/<Model>/start.S

```
    /*
        31  30  29  28 ----- 7   6   -   4   3   2   1   0
        N   Z   C   V        I   F       M4  M3  M2  M1 M0
                                         1   0   0   0   0     User Mode
                                         1   0   0   0   1     FIQ Mode
                                         1   0   0   1   0     IRQ Mode
                                         1   0   0   1   1     SVC Mode
                                         1   0   1   1   1     ABT Mode
                                         1   1   0   1   1     UND Mode
                                         1   1   1   1   1     SYS Mode
        */
        mrs r0, cpsr
        bic r0, r0, #0x1f
        orr r0, r0, #0xd3
```

Load board_init_r and Enter Stage 2

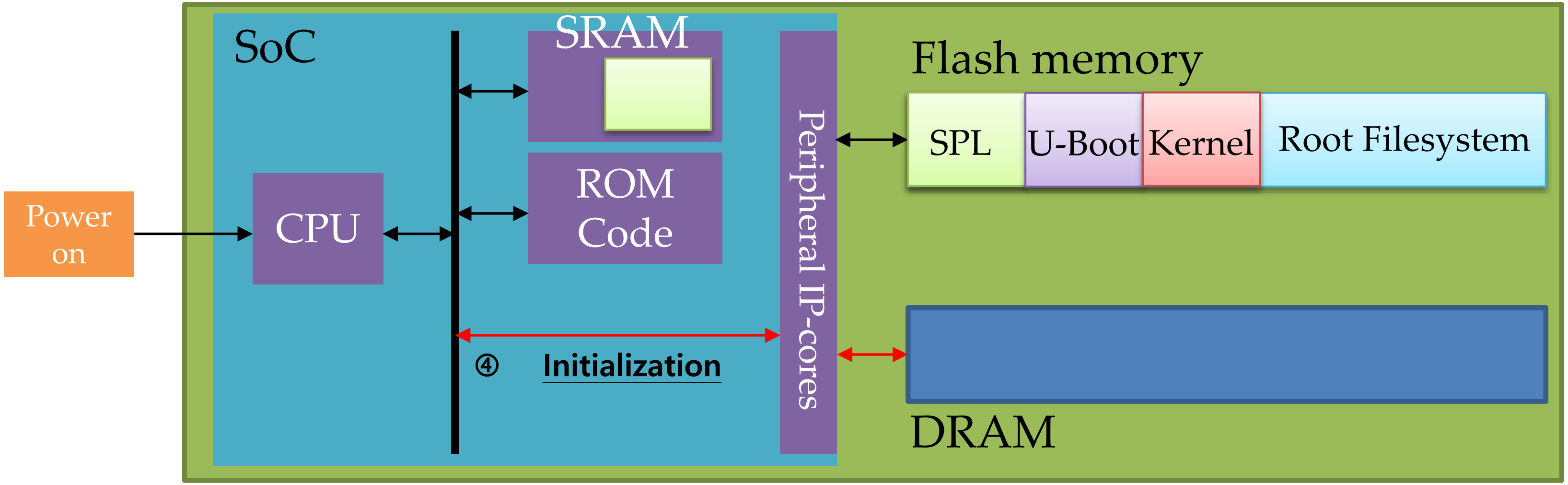
- In ./arch/arm/cpu/<HW>/<Model>/start.S

```
mov r0, r9                          /* gd_t */
ldr r1,  =(CONFIG_SYS_MALLOC_END)   /* dest_addr for malloc heap end */
/* call board_init_r */
ldr pc, =board_init_r
```

- In /u-boot/arch/arm/lib/board.c

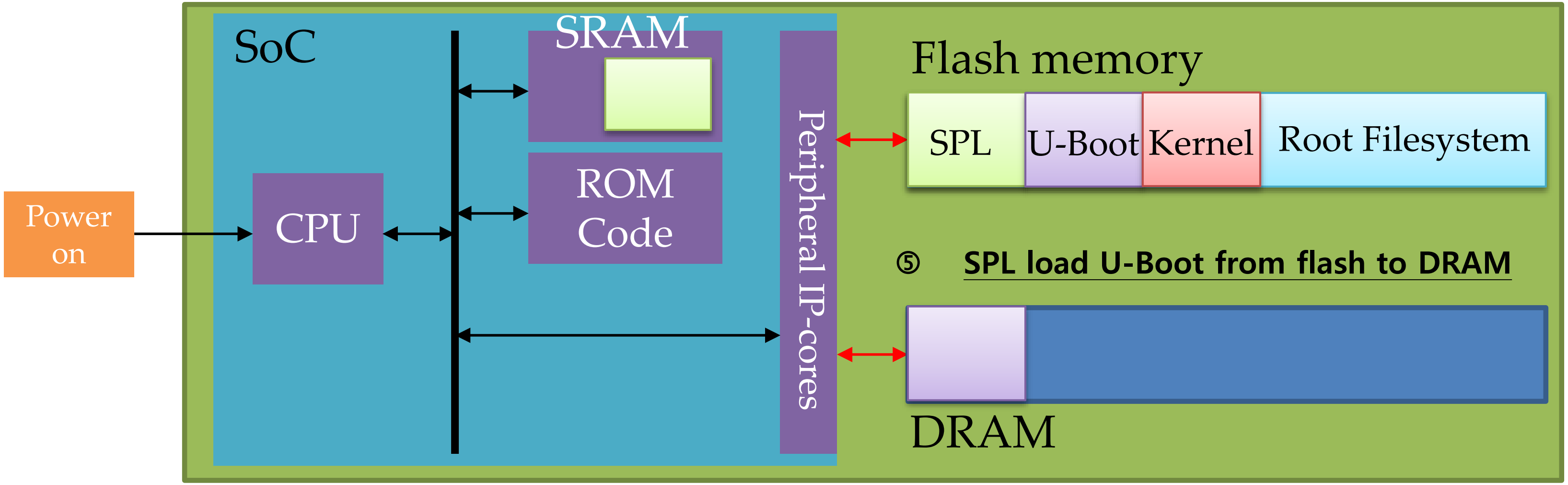
```
init_fnc_t init_sequence_r[] = {
#if defined(CONFIG_SYS_INIT_RAM_LOCK) && defined(CONFIG_E500)
    initr_unlock_ram_in_cache,
#endif
    initr_barrier,
    initr_malloc,
    console_init_m,
```

Flow for Boot Process



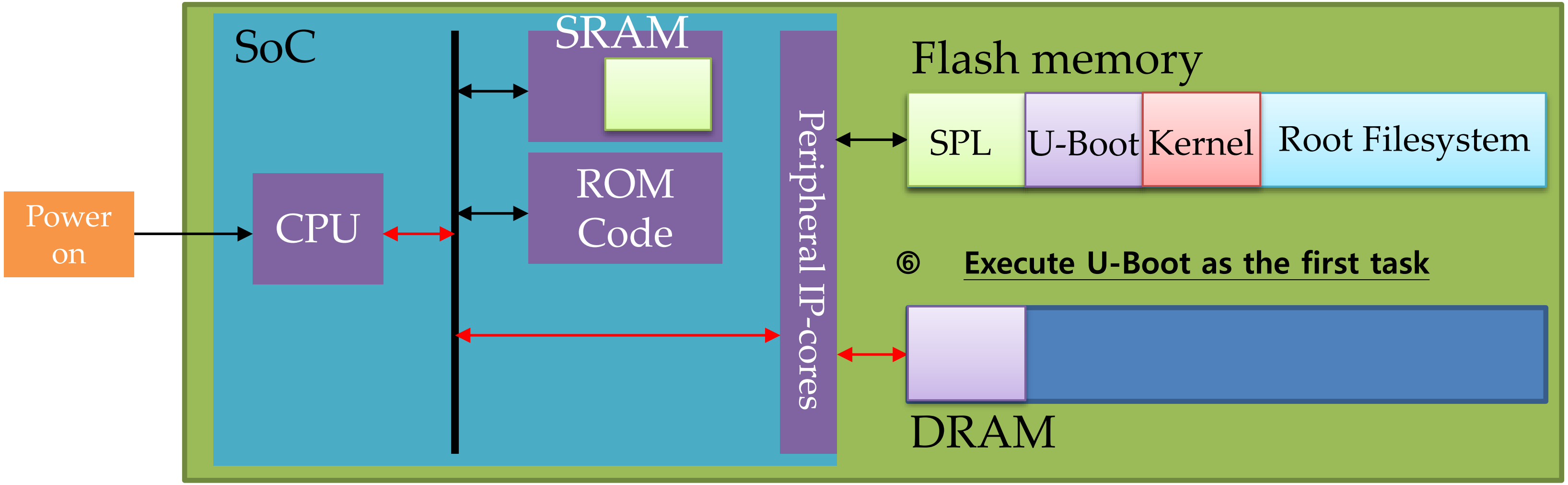
- DRAM space initialization

Flow for Boot Process



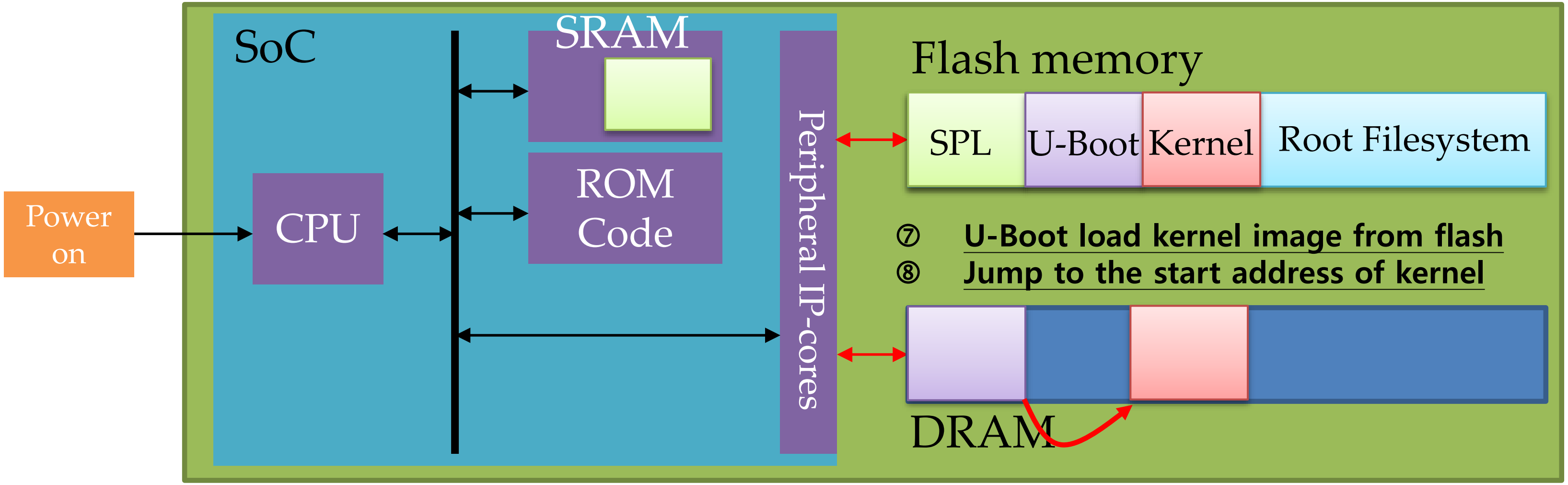
- Execute SPL to load U-Boot from flash memory to DRAM

Flow for Boot Process



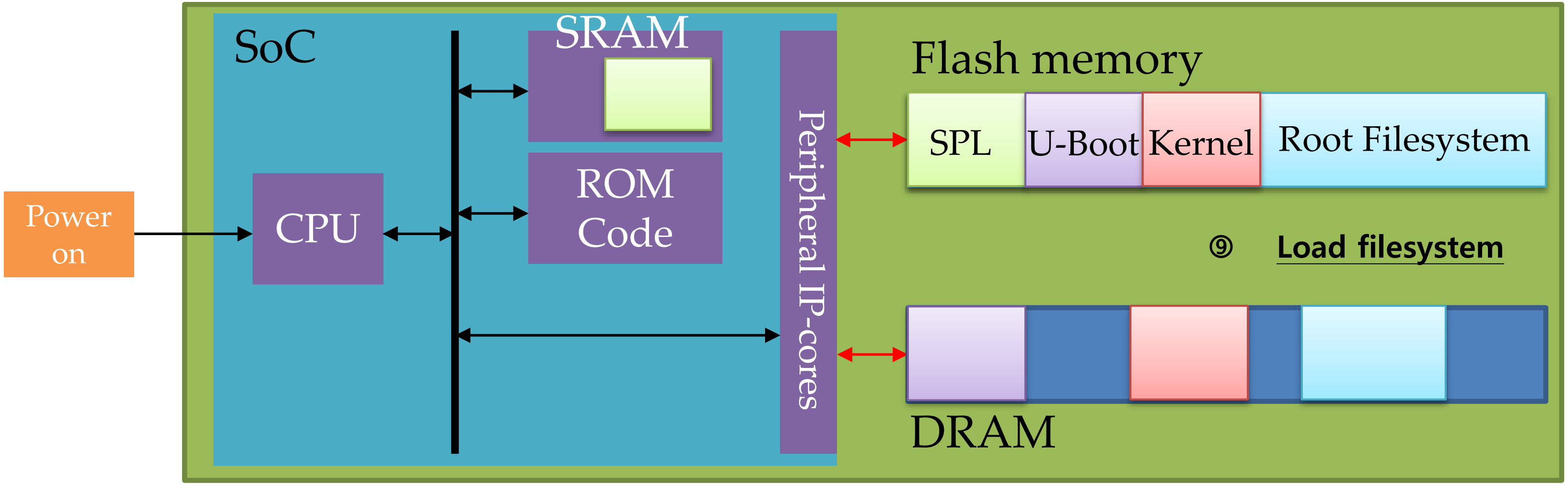
- Jump to the start address of U-Boot and execute its code

Flow for Boot Process



- Load kernel image from flash memory
- Decompress the kernel image in DRAM

Flow for Boot Process



- Load filesystem image from flash memory to DRAM
- Mount the root filesystem

How to Run U-Boot

- ➢ Step 1: What do we need?
  - ✓ Platform information: QEMU to emulate vexpress-a9
  - ✓ U-Boot source code from official site
    - https://www.denx.de/wiki/U-Boot/WebChanges
  - ✓ Linux source code from linux official site
    - https://www.kernel.org/

- ➢ Step 2: Compile U-Boot

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- vexpress_ca9x4_defconfig
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- menuconfig
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j8
```

How to Run U-Boot (cont'd.)

- Step 3: Copy zImage and dtb to SD card

```
sudo cp linux-4.14.13/arch/arm/boot/zImage p1/
sudo cp linux-4.14.13/arch/arm/boot/dts/vexpress-v2*.dtb p1/
sudo cp -raf ../rootfs/rootfs/* ./p2
```

- Step 4.a: Launch U-Boot on QEMU
  - ✓ uboot_image=./u-boot-2019.10/u-boot
  - ✓ qemu_path=/home/pengdl/work/Qemu/qemu-4.1.0/build/arm-softmmu

```
${qemu_path}/qemu-system-arm -M vexpress-a9 \
    -m 1024M \
    -smp 1 \
    -nographic \
    -kernel ${uboot_image} \
    -sd ./uboot.disk
```

How to Run U-Boot (cont'd.)

- Step 4.b: In U-Boot, mmc command

```
=> mmc dev 0
switch to partitions #0, OK
mmc0 is current device
=> mmc info
Device: MMC
Manufacturer ID: aa
OEM: 5859
Name: QEMU!
Bus Speed: 6250000
Mode: SD Legacy
Rd Block Len: 512
SD version 2.0
High Capacity: No
Capacity: 1 GiB
Bus Width: 1-bit
Erase Group Size: 512 Bytes
```

```
=> part list mmc 0
……
=> ls mmc 0:1  or ext4ls mmc 0:1
<DIR> 1024 .
<DIR> 1024 ..
<DIR> 12288 lost+found
        7680720 zImage
          19161 vexpress-v2p-ca15_a7.dtb
          13384 vexpress-v2p-ca15-tc1.dtb
          12994 vexpress-v2p-ca5s.dtb
          14736 vexpress-v2p-ca9.dtb
=> ls mmc 0:2  or ext4ls mmc 0:2
<DIR> 4096 .
<DIR> 4096 ..
<DIR> 16384 lost+found
<DIR> 4096 bin
<DIR> 4096 dev …
```

How to Run U-Boot (cont’d.)

- Step 5: Load kernel and device tree

```
=> load mmc 0:1 0x60008000 zImage or ext4load mmc 0:1 0x60008000 zImage
7680720 bytes read in 1034 ms (7.1 MiB/s)
=> load mmc 0:1 0x61000000 vexpress-v2p-ca9.dtb  or
=> ext4load mmc 0:1 0x61000000 vexpress-v2p-ca9.dtb
14736 bytes read in 67 ms (213.9 KiB/s)
```

- Step 6: Set-up Bootargs

```
=> setenv bootargs 'root=/dev/mmcblk0p2 rw rootfstype=ext4 rootwait earlycon
console=tty0 console=ttyAMA0 init=/linuxrc ignore_loglevel'
```

- Step 7: Jump to kernel

```
=> bootz 0x60008000 - 0x61000000
```

Device Tree

- ➢ What is device tree?
  - ✓ Device Tree (DT) was created by **<u>Open Firmware</u>** to allow an operating system at runtime to run on **<u>various hardware</u>** without **<u>hard coding any information</u>**
  - ✓ It is a data structure and language for describing hardware.
- ➢ Pros
  - ✓ Attempting to eliminate board specific code from drives
  - ✓ Ability to cleanly support multiple boards with a single kernel
  - ✓ Replaces complex and massive board file
- ➢ Cons
  - ✓ Kernel size increase
  - ✓ Increase in boot time

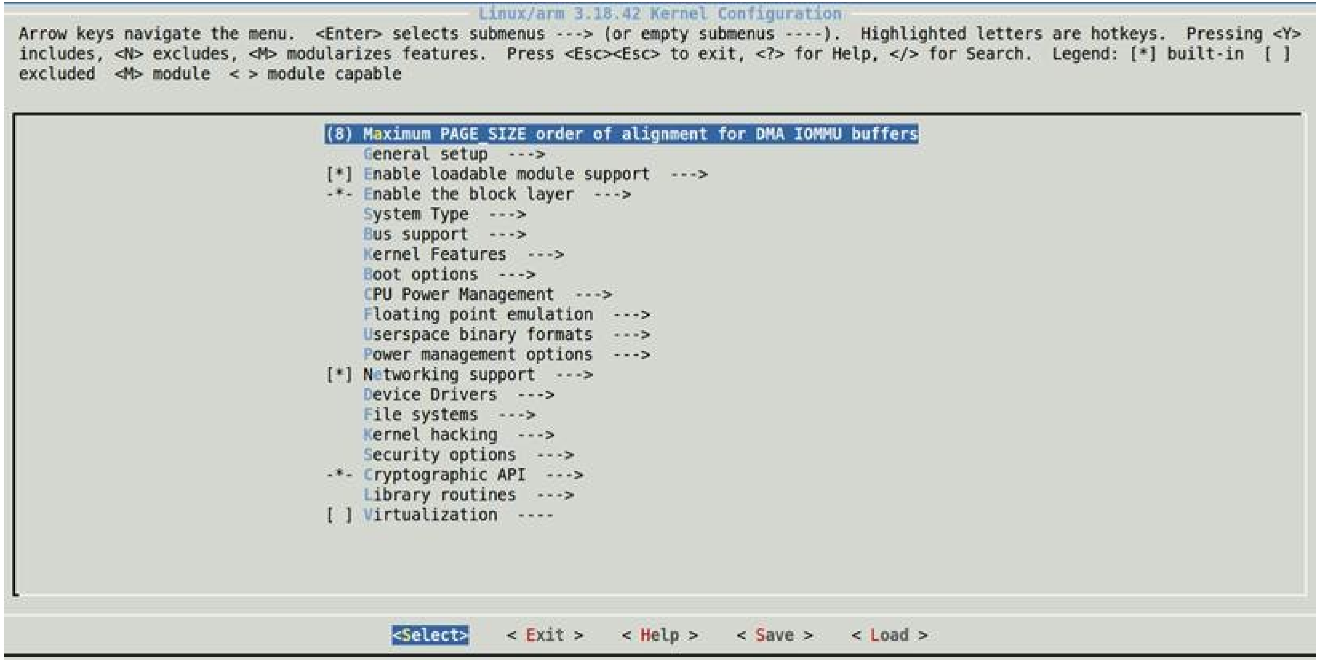
Generate Device Tree

- Need a compiler: Device tree compile (DTC)

```
make ARCH=arm vexpress_ca9x4_defconfig
make ARCH=arm menuconfig
```

- Generate dtb
  - scripts/dtc/dtc -I dts -O dtb -o <devicetree name>.dtb <devicetree name>.dts

Make Menuconfig



How to Load DTB

- ➢ Device tree blob (dtb) is passed to the kernel via U-boot

- ➢ U-boot loads the appropriate dtb in to memory
  - ✓ Currently stored at 0x80F80000 (fdtaddr) which is a free region of memory that doesn’t overlap with any other memory location

- ➢ U-boot boots the kernel using the below command
  - ✓ bootm/bootz ${loadaddr} - ${fdtaddr};

DT: Driver Instantiation Example

- AM335x generic configuration for i2c0

Driver node name

DT device id. Determines which driver This DT binding is meant for.

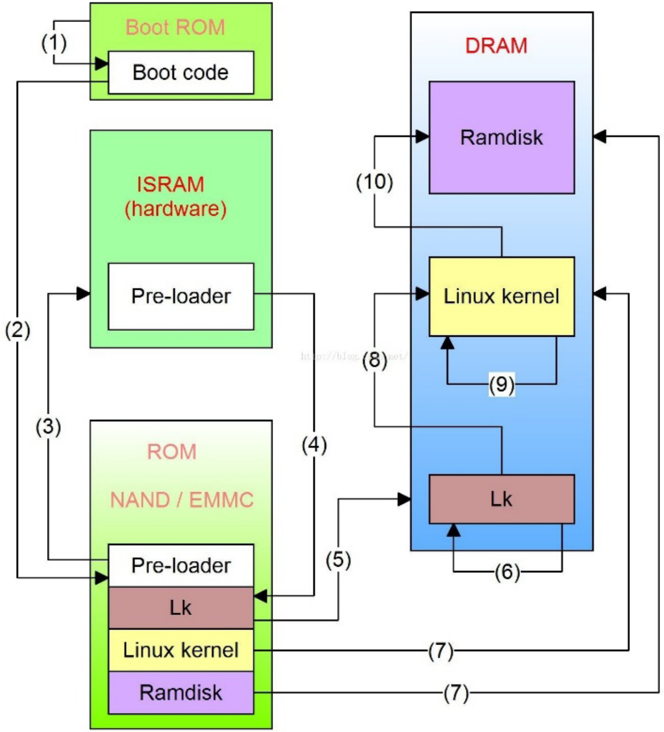
```
i2c0: i2c@44e0b000 {
        compatible = "ti,omap4-i2c";
        #address-cells = <1>;
        #size-cells = <0>;
        ti,hwmods = "i2c1";
        reg = <0x44e0b000 0x1000>;
        interrupts = <70>;
        status = "disabled";
};
```

Driver register information. For i2c this determines memory mapped register for i2c0

Determines which hardware interrupt is required

New Bootloader: Little Kernel (lk)

- MTK MT6580



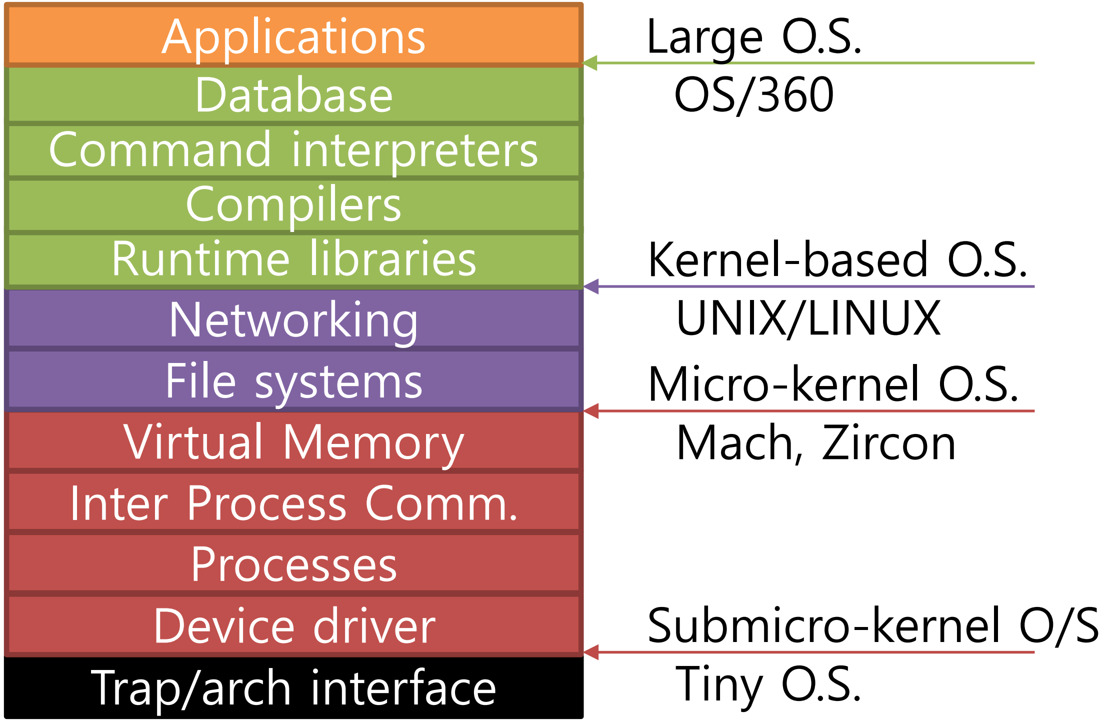
Little Kernel (lk)

- What is little kernel?
  - Open Source software
  - A tiny operating system
  - Suited for small embedded devices, bootloaders, and other environments
  - For Android system
  - Only 15-20 KB
  - Available from https://github.com/littlekernel/lk
  - As a bootloader
    - A few ARM SoC manufacturers
  - As a microkernel
    - Zircon

Outline

- How to Boot an Embedded System (ES)
    - ✓ Grub for x86 Architecture
    - ✓ U-boot for ARM Architecture
    - ✓ How to Implement a Bootloader in an Embedded System.
- Microkernel for Embedded System
    - ✓ What is Process?
    - ✓ The Process Concept in an Embedded System
    - ✓ The Practical Knowledge of Process Management

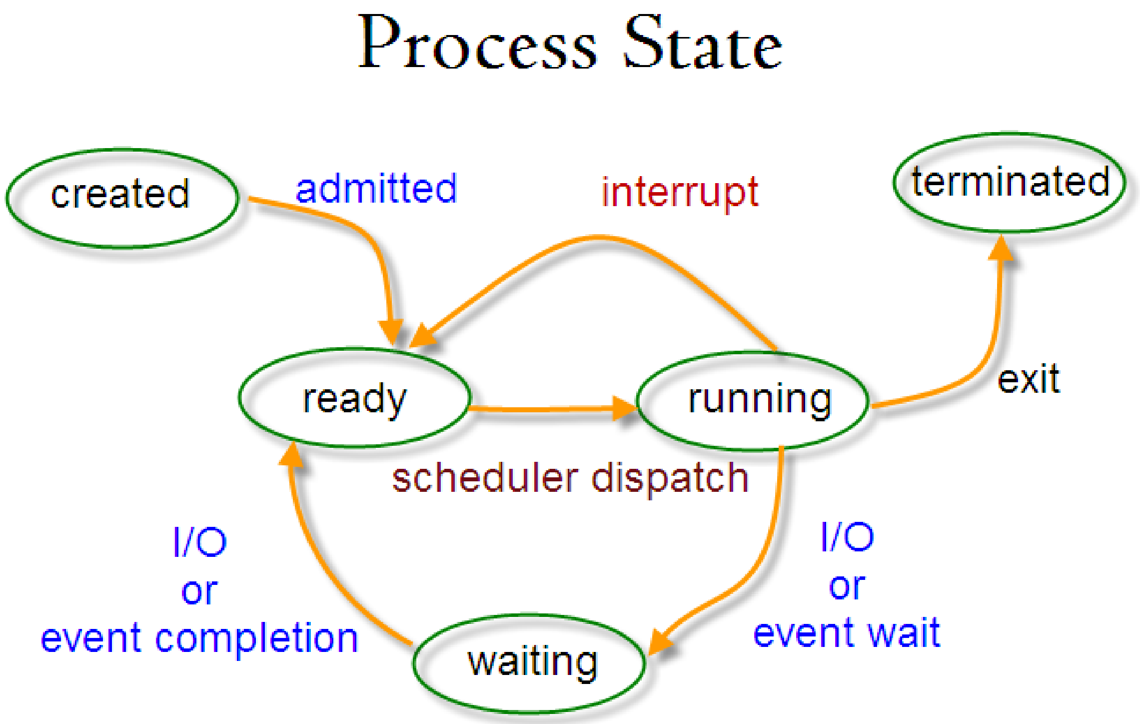
Various O.S. Abstractions



What is Process?

- Definition of process
    - ✓ A program in execution.
    - ✓ A process is a series of steps and decisions involved in the way work is completed.
    - ✓ Each individual process runs in its own memory space and is not capable of interacting with another process except through secure kernel managed mechanisms.

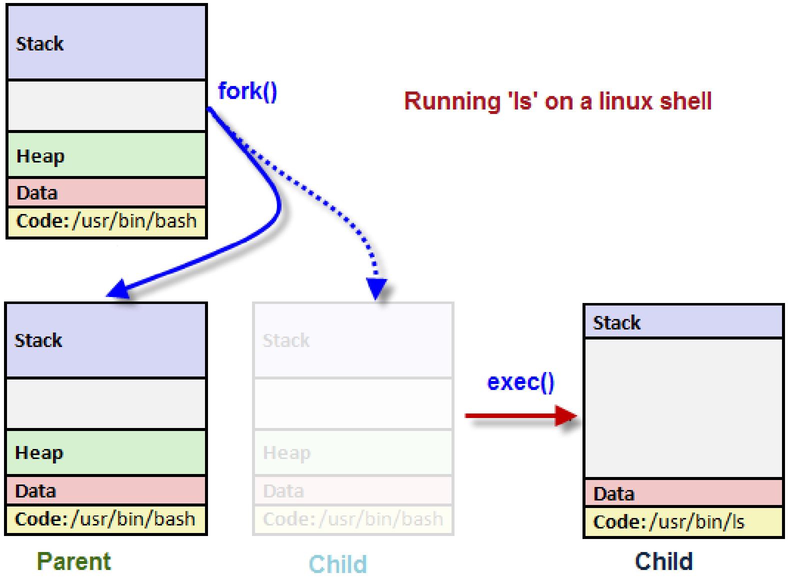
The Process State



How to Create a Process?

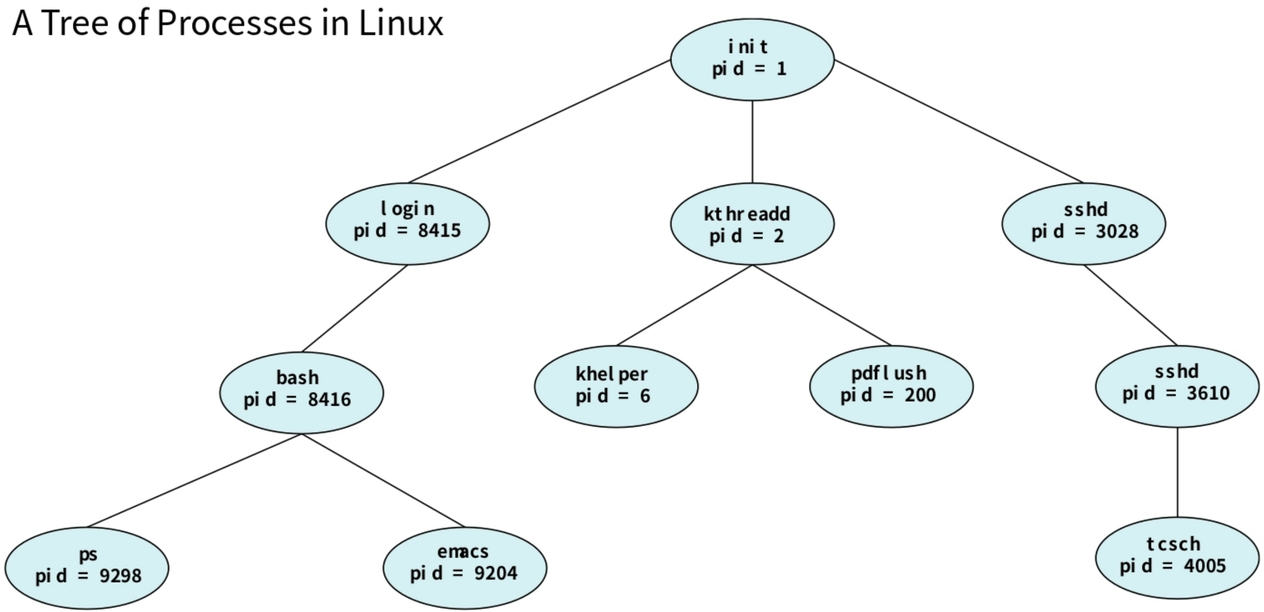
- `fork()` a process

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

int main()
{
        pid_t pid;

        /*fork a child process*/
        pid = fork();
        …
}
```



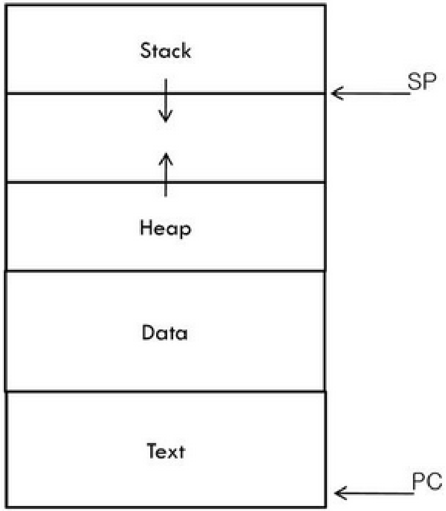
Process Structure in Linux

- A tree structure for all processes in Linux



Process in Memory

- A process should includes:
  - Program counter
    - Determines which process instruction is executed next
  - Stack
    - Used for return addresses
  - Data section
    - Initialized static variables, that is, global variables and static local variables
  - Heap
    - Managed by malloc, calloc, realloc, and free
  - Text
    - Contains executable instructions



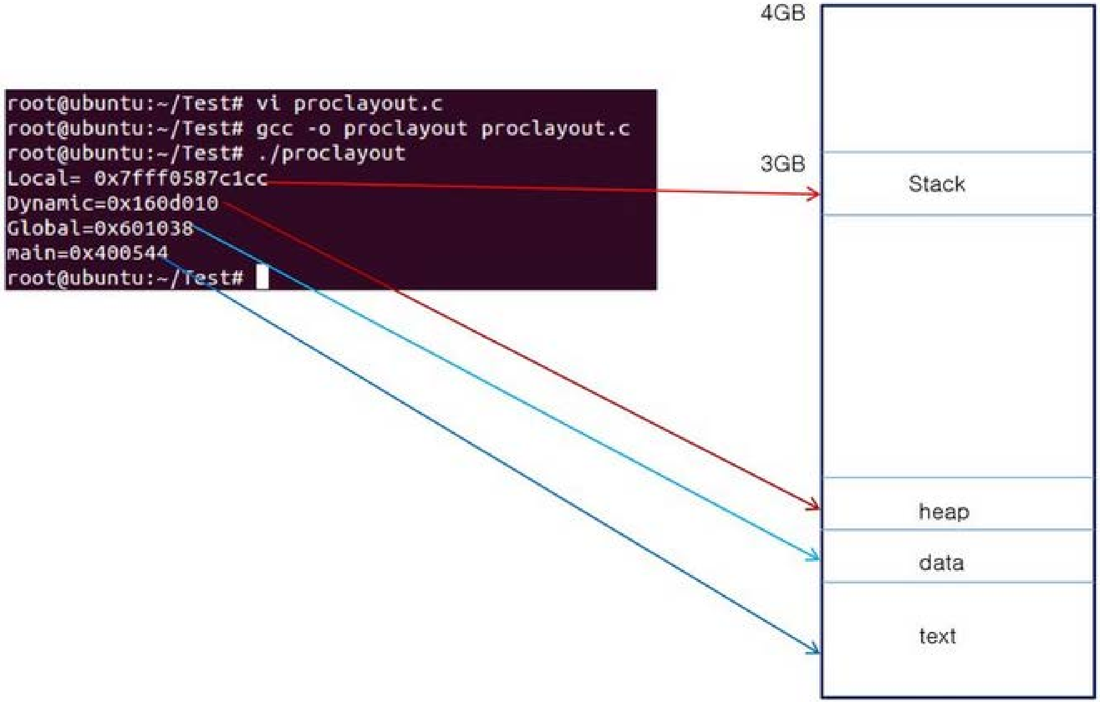
Process in Memory: Example (I/II)

- Write a program to allocate memory and set parameters

```
#include <stdio.h>
#include <stdlib.h>

Int global;

int main(void)
{
        int local, *dynamic;
        dynamic = malloc(1000);
        printf(“Local=%p\n”,&local);
        printf(“Dynamic=%p\n”,dynamic);
        printf(“Global=%p\n”,&global);
        printf(“main=%p\n”,main);
        return 0;
}
```

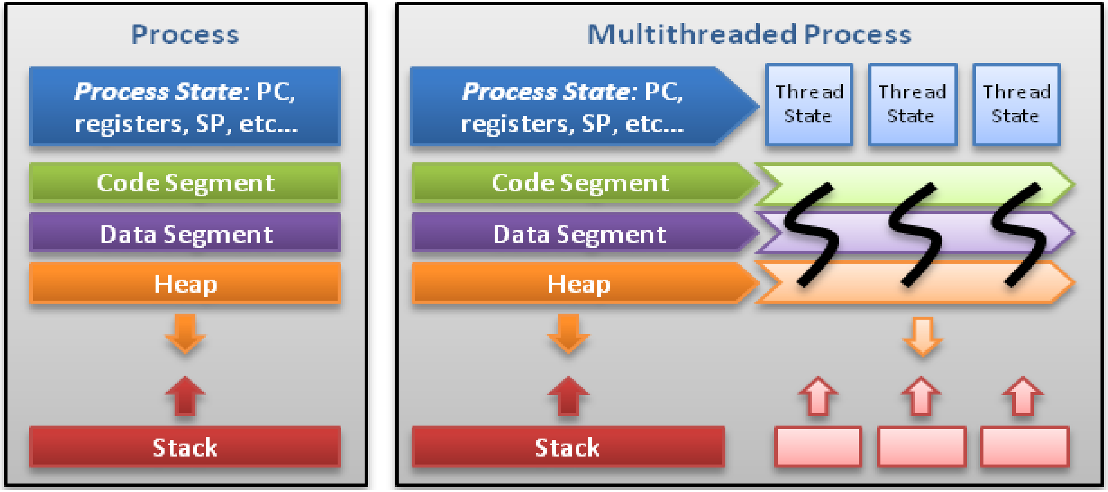
Process in Memory: Example (II/II)

- Write a program to allocate memory and set parameters



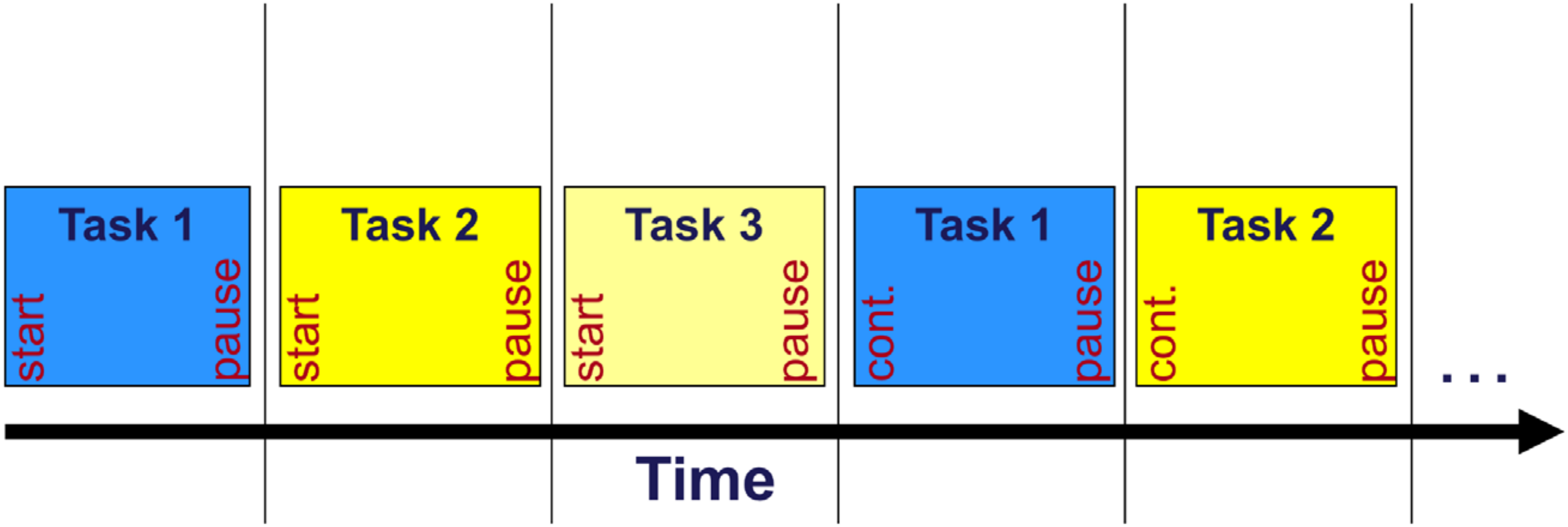
Inside Process

- Threads
  - A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a **scheduler**
  - They can exist within **one process**, **executing concurrently** and **sharing resources** such as memory, while different processes do not share these resources.



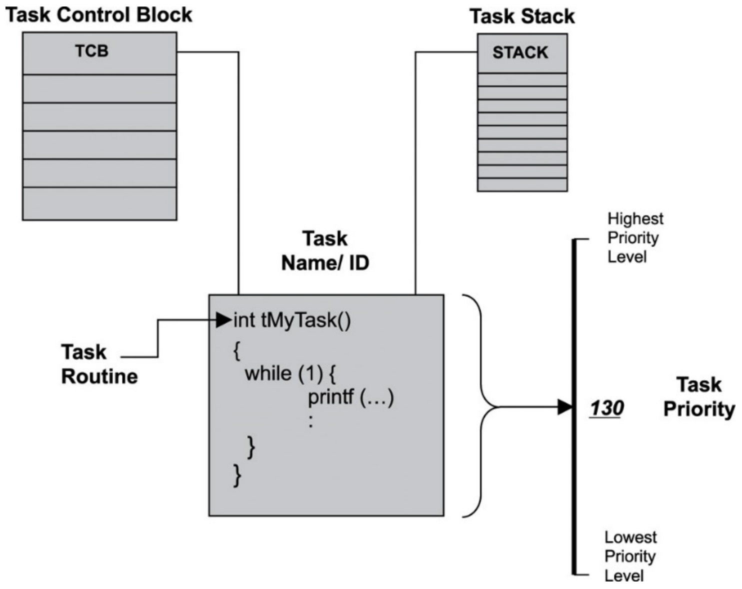
Process and Thread for Scheduler

- Linux **does not distinguish** processes and threads.
- Processes and threads are regarded as **tasks**.



What is a Task?

- A task is an **independent thread** of execution that can compete with other concurrent tasks for processor execution time.
- The task is able to compete for execution time on a system, based on a **predefined scheduling algorithm**.



Why CPU Scheduler

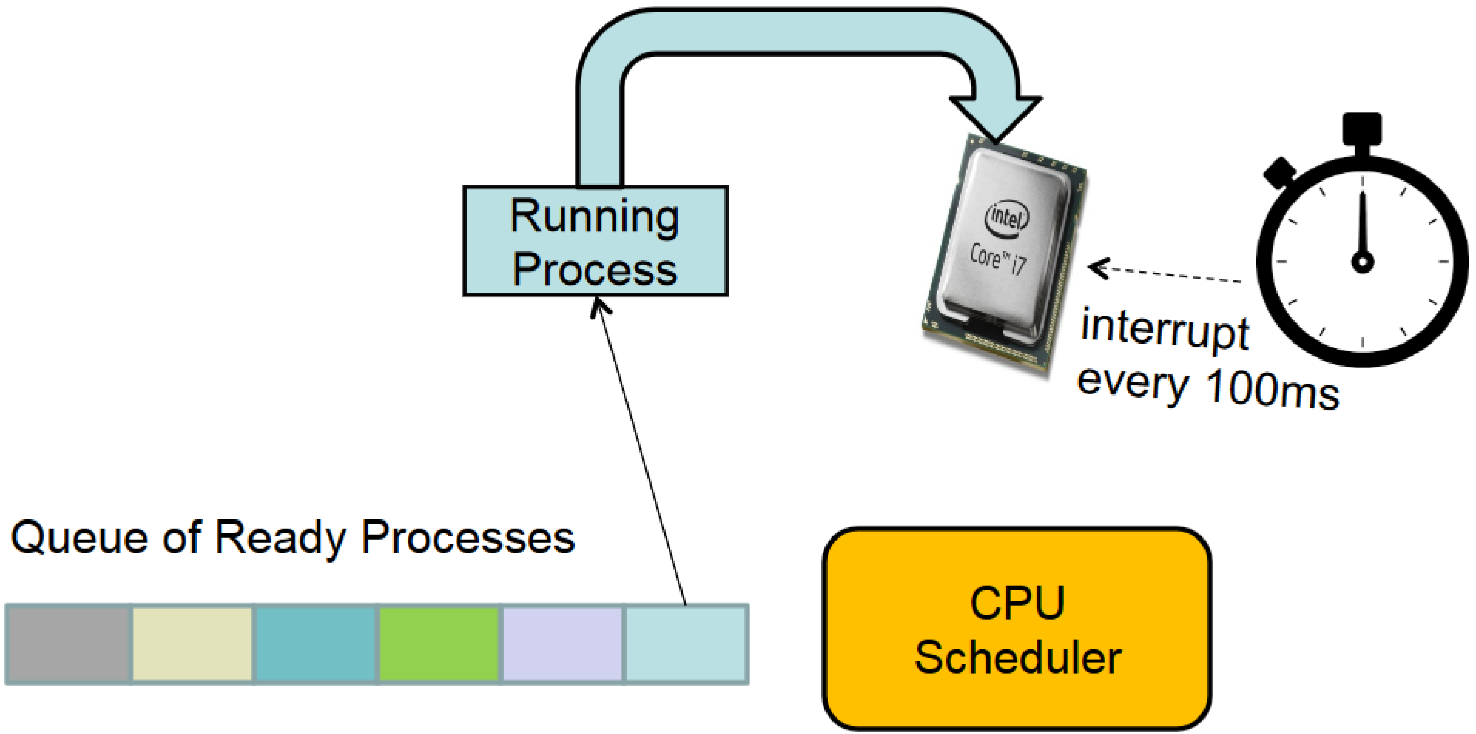
- Types of Process
  - ✓ I/O bound process
    - Has small bursts of CPU activity and then waits for I/O
    - For example: GUI process
    - Affect user interaction

CPU usage

  - ✓ CPU bound process
    - Hardly any I/O, mostly CPU activity (e.g., Scientific computing)

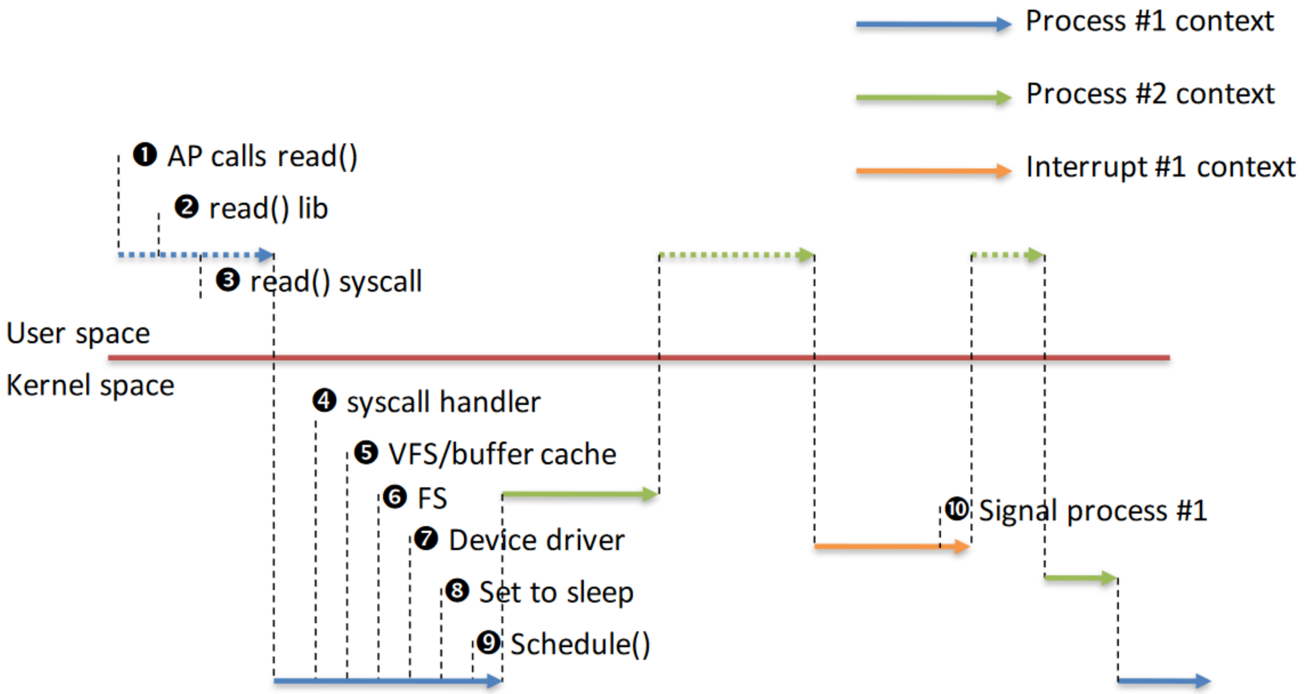
CPU usage

Task Scheduler



When Scheduler is Triggered

- Scheduler triggered to run when timer interrupt occurs or when running process is blocked on I/O



Schedulers

- Decides which process should run next
- Aims …
  - ✓ Minimize waiting time
    - Process should not wait long in the ready queue
  - ✓ Maximize CPU utilization
    - CPU should not be idle
  - ✓ Maximize throughput
    - Complete as many processes as possible per unit time
  - ✓ Minimize response time
    - CPU should respond immediately
  - ✓ Fairness
    - Give each process a fair share of CPU

Some Schedulers in Textbook (I/II)

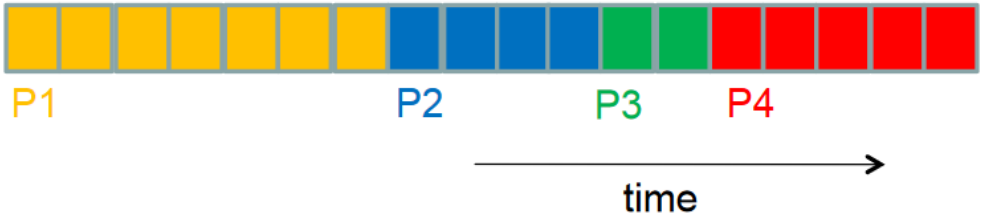
- First come first serve

Process	Arrival Time	Burst Time
P1	0	7
P2	0	4
P3	0	2
P4	0	5

Average Waiting Time
= (0 + 7 + 11 + 13) / 4
= 7.75

Average Response Time
= (0 + 7 + 11 + 13) / 4
= 7.75
(same as Average Waiting Time)

Grantt Chart



Some Schedulers in Textbook (II/II)

- Round robin scheduling: Run process for a time slice then move to FIFO

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	3	2
P4	9	1

Time slice = 2

Average Waiting time
= (7 + 4 + 3 + 3) / 4
= 4.25

Average Response Time
= (0 + 0 + 3 + 3) / 4
= 1.5

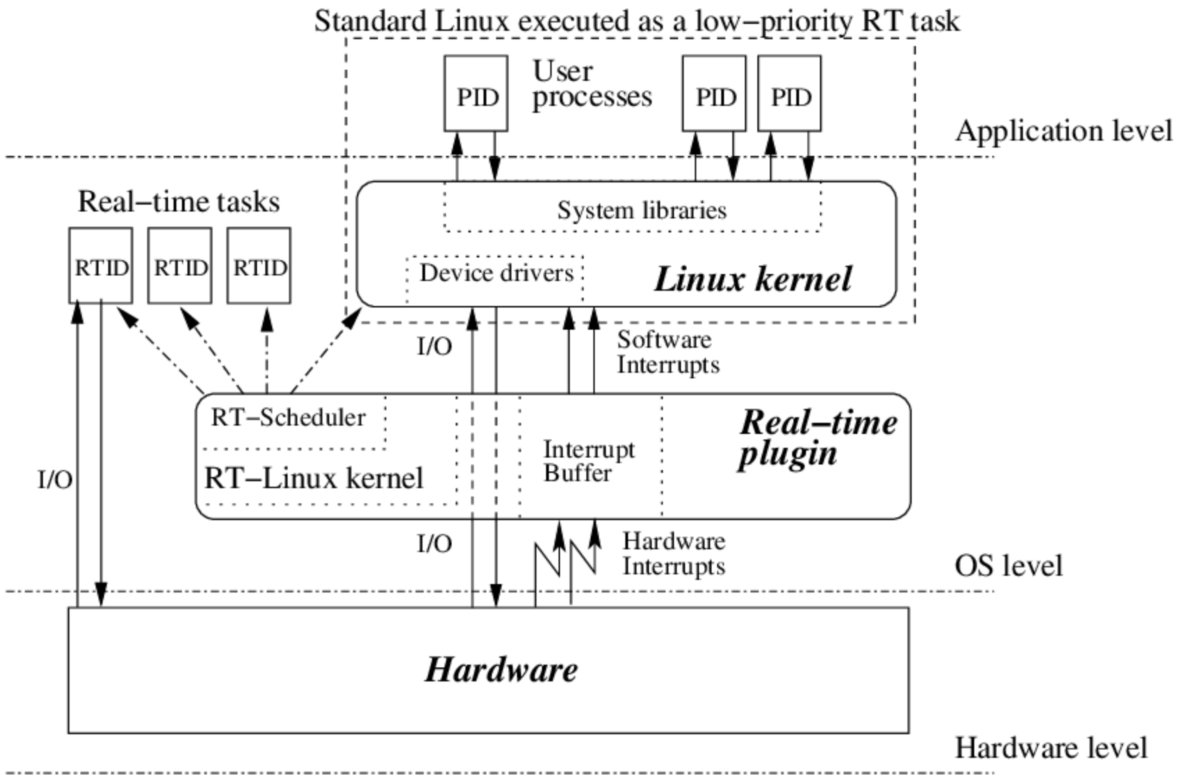
#Context Switches = 7

Arrival	**P1**		**P2**	**P3**						**P4**				
schedule	P1		P2		P1		P3		P2		P1		P4	P1
FIFO			P1	P1 P3	P3 P2		P2 P1		P1	P1 P4	P4 P1		P1	

Recall: Context Switching Overheads

- **Direct Factors** affecting context switching time
  - Timer Interrupt latency
  - Saving/restoring contexts
  - Finding the next process to execute
- Indirect factors
  - TLB needs to be reloaded
  - Loss of cache locality (therefore more cache misses)
  - Processor pipeline flush

Scheduler in Embedded Linux System



Scheduling Algorithms in Linux

- Linux supports the following scheduling algorithms
  - ✓ Normal mode (Completely Fair Scheduler; CFS)
    - The Completely Fair Scheduler (CFS) is a **<u>process scheduler</u>** which was merged into **<u>the 2.6.23</u>** (October 2007) release of the Linux kernel and is the **<u>default scheduler</u>**.
  - ✓ FIFO mode
  - ✓ RR mode
  - ✓ Batch mode
    - For I/O bound and low priority task, it will schedule the tasks while the CPU is idle.

```
/*
 * Scheduling policies
 */
#define SCHED_NORMAL        0
#define SCHED_FIFO          1
#define SCHED_RR            2
#define SCHED_BATCH         3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE          5
#define SCHED_DEADLINE      6
```

In `/include/uapi/linux/sched.h`

Process Types

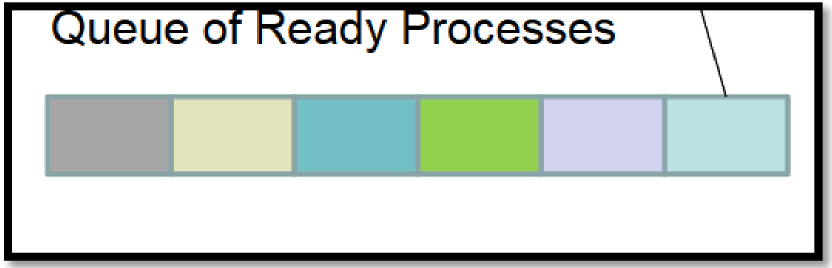
- **Real-time** process/task
  - **Deadlines** that have to be met
  - Should never be **blocked** by **a low priority task**
- **Normal** process/task
  - Interactive
    - Constantly **interact with users**, therefore spend a lot of time waiting for key presses and mouse operations.
    - When input is received, the process must wake up quickly (delay must be between 50 to 150 ms)
  - Background
    - Do not require any user interaction, often **run in the background**.

History

- Schedulers for normal processors
  - O(n) scheduler – Linux 2.4 to 2.6
  - O(1) scheduler – Linux 2.6 to 2.6.22
  - CFS scheduler – Linux 2.6.23 onwards

O(n) Scheduler

- At every context switch
  - ✓ **Scan** the list of runnable processes
  - ✓ **Compute** priorities
  - ✓ **Select** the best process to run



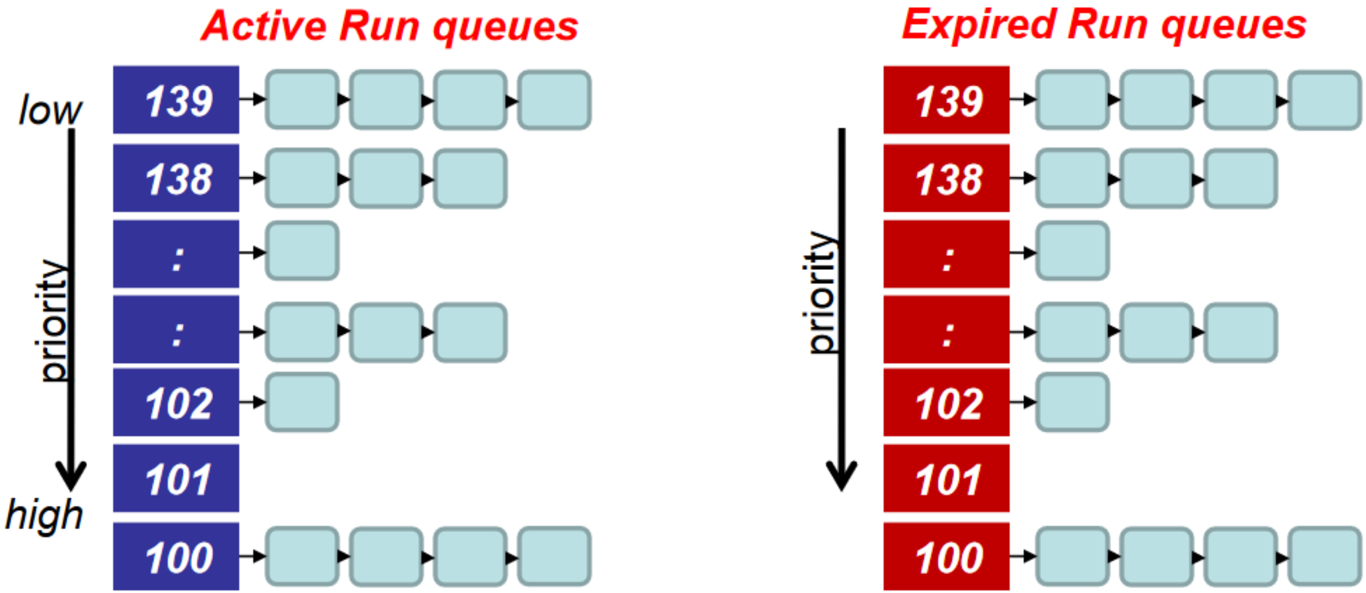
- O(n), when n is the number of runnable processes ... **<u>not scalable</u>**!!
  - ✓ Scalability issues observed when Java was introduced (JVM spawns many tasks)
- Used a global run-queue in SMP systems
  - ✓ Again, **not scalable!!**

O(1) Scheduler

- Constant time required to pick the next process to execute
  - Easily scales to large number of processes
- Processes divided into 2 types
  - Real time
    - Priorities from 0 to 99
  - Normal processes
    - Interactive
    - Batch
    - Priorities from 100 to 139 (100 highest, 139 lowest priority)

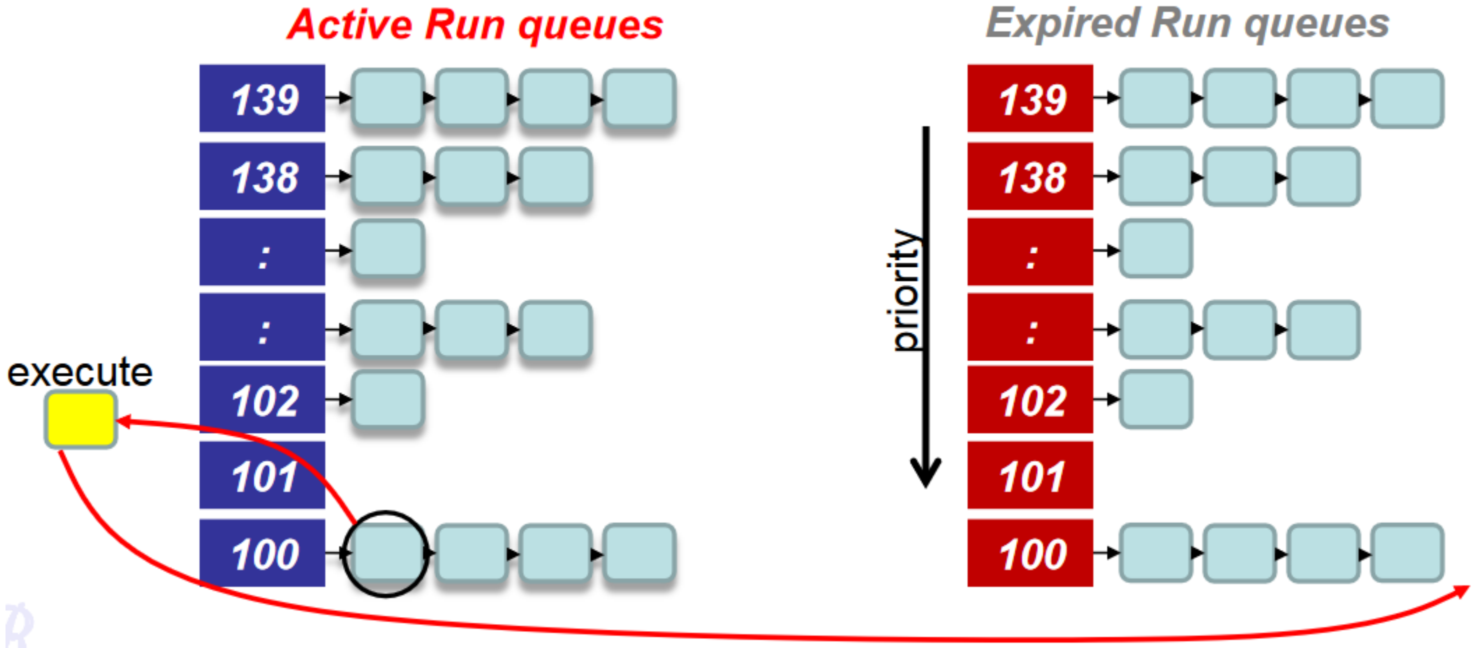
Scheduling Normal Processes

- Two ready queues in each CPU
  - ✓ Each queue has 40 priority classes (100 – 139)
  - ✓ 100 has highest priority, 139 has lowest priority



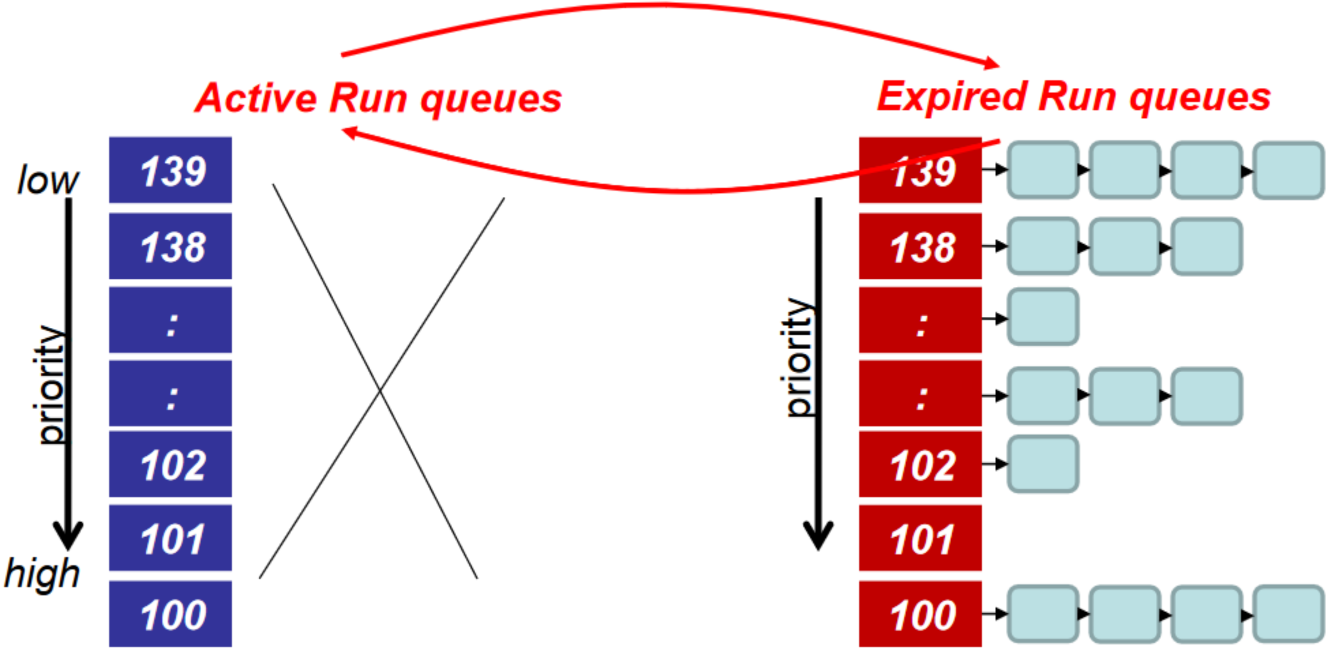
The Scheduling Policy (I/II)

- Pick the first task from the lowest numbered run queue
- When done put task in the appropriate queue in the expired run queue



The Scheduling Policy (II/II)

- Once active run queues are complete
  - Make expired run queues active and vice versa



More on Priorities

- 0 to 99 meant for **real time processes**
- 100 is the **highest** priority for a normal process
- 139 is the **lowest** priority
- Static priorities
  - 120 is the base priority (default)
  - `nice`: command line **to change default priority** of a process
    `$nice –n N ./a.out`
  - N is a value from +19 to -20
    - most selfish ‘-20’ (I want to go first)
    - most generous ‘+19’; (I will go last)

More on Priorities

- 0 to 99 meant for **real time processes**
- 100 is the **highest** priority for a normal process
- 139 is the **lowest** priority
- Static priorities
  - 120 is the base priority (default)
  - `nice`: command line **to change default priority** of a process
    `$nice –n N ./a.out`
  - N is a value from +19 to -20
    - most selfish ‘-20’ (I want to go first)
    - most generous ‘+19’; (I will go last)

Dynamic Priority

- To distinguish between **background** and **interactive** processes
- Uses a ‘bonus’, which changes based on a heuristic

dynamic priority = MAX(100, MIN(static priority – bonus + 5), 139))

Has a value between 0 and 10

If bonus < 5, implies less interaction with the user
thus more of a CPU bound process.
The dynamic priority is therefore decreased (toward 139)

If bonus > 5, implies more interaction with the user
thus more of an interactive process.
The dynamic priority is increased (toward 100).

How to Set the Bonus

- Based on average sleep time
  - ✓ An I/O bound process will sleep more therefore should get a higher priority
  - ✓ A CPU bound process will sleep less, therefore should get lower priority

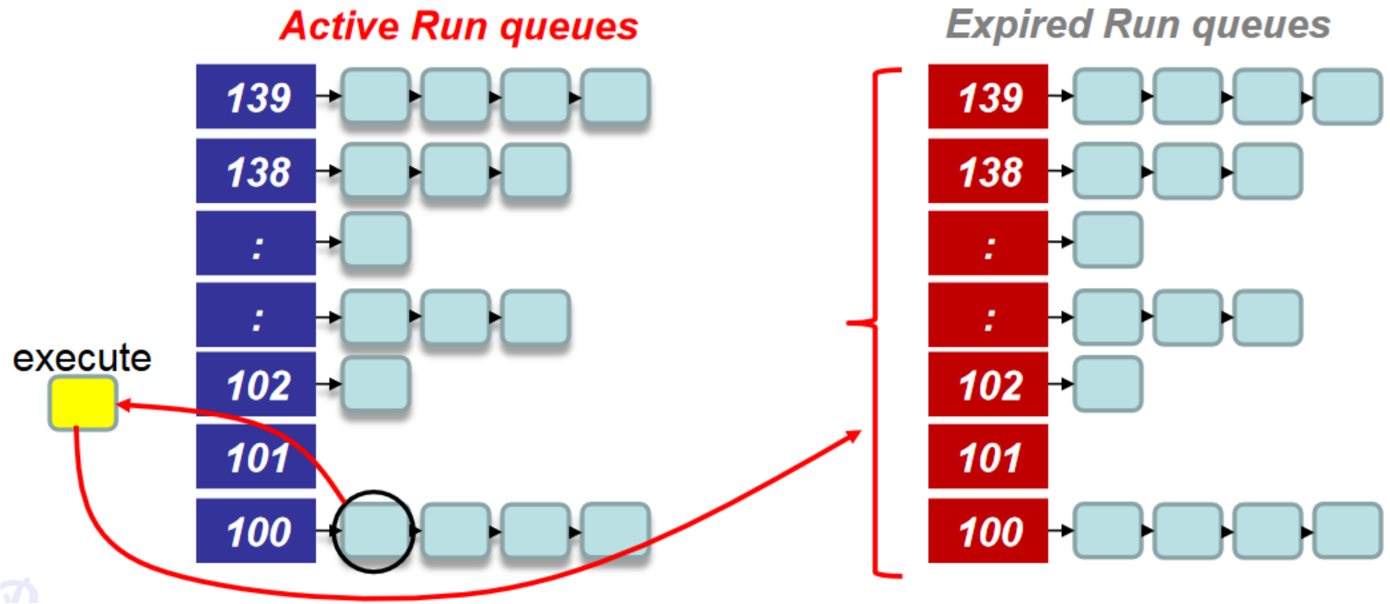
$$\text{dynamic priority} = MAX(100, MIN(\text{static priority} - \text{bonus} + 5), 139))$$

heuristic

Average sleep time	Bonus
Greater than or equal to 0 but smaller than 100 ms	0
Greater than or equal to 100 ms but smaller than 200 ms	1
Greater than or equal to 200 ms but smaller than 300 ms	2
Greater than or equal to 300 ms but smaller than 400 ms	3
Greater than or equal to 400 ms but smaller than 500 ms	4
Greater than or equal to 500 ms but smaller than 600 ms	5
Greater than or equal to 600 ms but smaller than 700 ms	6
Greater than or equal to 700 ms but smaller than 800 ms	7
Greater than or equal to 800 ms but smaller than 900 ms	8
Greater than or equal to 900 ms but smaller than 1000 ms	9
1 second	10

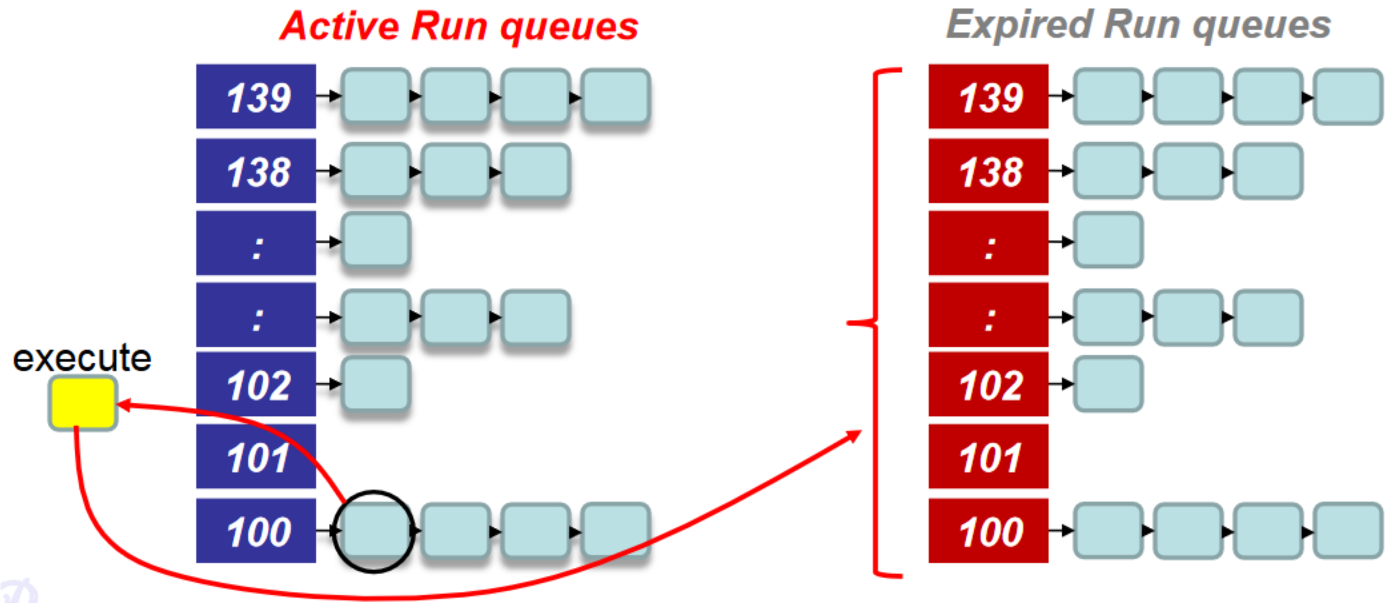
Dynamic Priority and Run Queues

- Dynamic priority used to determine which run queue to put the task
- No matter how 'nice' you are, you still need to wait on run queues --- prevents starvation



Dynamic Priority and Run Queues

- Dynamic priority used to determine which run queue to put the task
- No matter how ‘nice’ you are, you still need to wait on run queues --- prevents starvation



Setting the Timeslice

- Interactive processes have high priorities
  - ✓ But likely to not complete their timeslice
  - ✓ Give it the largest **timeslice** to ensure that it completes its burst without being **preempted**. More heuristics

```
If priority < 120
    time slice = (140 – priority) * 20    milliseconds
else
    time slice = (140 – priority) * 5   milliseconds
```

Priority:	Static Pri	Niceness	Quantum
Highest	100	-20	800 ms
High	110	-10	600 ms
Normal	120	0	100 ms
Low	130	10	50 ms
Lowest	139	19	5 ms

Ideal Fair Scheduling (I/III)

Process	burst time
A	8ms
B	4ms
C	16ms
D	4ms

Divide processor time equally among processes

Ideal Fairness : If there are N processes in the system, each process should have got (100/N)% of the CPU time

Ideal Fairness

Each process gets
4/4 = 1ms of the processor time

A	1	2	3	4	6	8							
B	1	2	3	4									
C	1	2	3	4	6	8	12	16					
D	1	2	3	4									

4ms slice

Execution with respect to time

Ideal Fair Scheduling (II/III)

Process	burst time
A	8ms
B	4ms
C	16ms
D	4ms

Divide processor time equally among processes

Ideal Fairness : If there are N processes in the system, each process should have got (100/N)% of the CPU time

Ideal Fairness

Each process gets
4/2 = 2ms of the processor time

A	1	2	3	4	6	8							
B	1	2	3	4									
C	1	2	3	4	6	8	12	16					
D	1	2	3	4									

4ms slice

Execution with respect to time

Ideal Fair Scheduling (III/III)

Process	burst time
A	8ms
B	4ms
C	16ms
D	4ms

Divide processor time equally among processes

Ideal Fairness : If there are N processes in the system, each process should have got (100/N)% of the CPU time

Ideal Fairness

The single process gets the entire 4ms of the processor time

A	1	2	3	4	6	8							
B	1	2	3	4									
C	1	2	3	4	6	8	12	16					
D	1	2	3	4									

4ms slice

Execution with respect to time

The CFS Idea

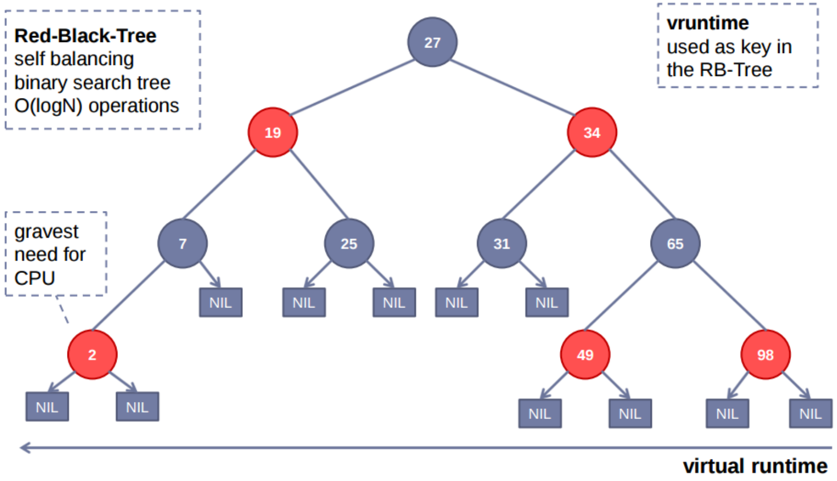
- ➢ When timer interrupt occurs
  - ✓ Choose the task with the lowest **vruntime** (**min_vruntime**)
  - ✓ Compute its dynamic timeslice
  - ✓ Program the **high resolution timer** with this timeslice
- ➢ The process begins to execute in the CPU
- ➢ When interrupt occurs again
  - ✓ Context switch if there is another task with a smaller runtime

Virtual Runtimes

- With each runnable process is included a virtual runtime (**vruntime**)
  - At every scheduling point, if process has run for *t* ms, then (**vruntime += t**)
  - vruntime for a process therefore **monotonically increases**

Picking the Next Task to Run (I/II)

- All **vruntime** values are maintained in a red-black tree
  - ✓ Each node in the tree represents a runnable task
  - ✓ Nodes **ordered** according to **their vruntime**
  - ✓ Nodes on **the left** have **lower vruntime** compared to nodes on the right of the tree
  - ✓ The **left most node** is the task with the **least vruntime**



Picking the Next Task to Run (II/II)

- At a context switch
  - Pick the left most node of the tree
    - This has the lowest runtime
    - It is cached in **min_vruntime**. Therefore accessed in O(1)
  - If the previous process is runnable, it is inserted into the tree depending on its **<u>new vruntime</u>**. Done in O(log(n))
    - Tasks move from left to right of tree after its execution completes… starvation avoided

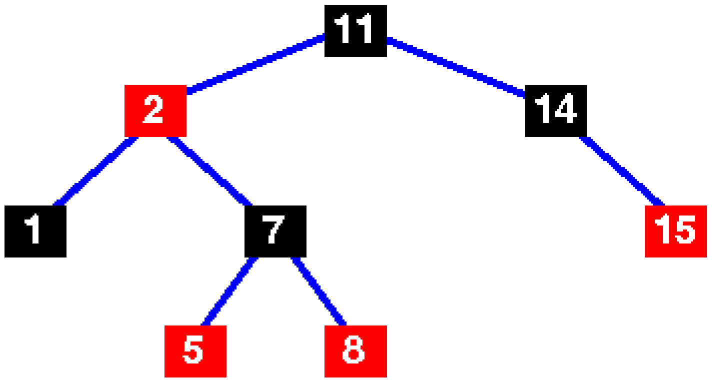
Picking the Next Task to Run (II/II)

- At a context switch
  - Pick the left most node of the tree
    - This has the lowest runtime
    - It is cached in **min_vruntime**. Therefore accessed in O(1)
  - If the previous process is runnable, it is inserted into the tree depending on its **new vruntime**. Done in O(log(n))
    - Tasks move from left to right of tree after its execution completes… starvation avoided

Why Red Black Tree?

Recall: Red-Black Tree

- A red-black tree is a binary search tree which has the following red-black properties:
  - ✓ A self balancing tree
  - ✓ Every node is either red or black
  - ✓ Every leaf (NULL) is black
  - ✓ If a node is red, then both its children are black
  - ✓ Every simple path from a node to a descendant leaf contains the same number of black nodes



Priorities and CFS

- Priority (due to nice values) used to weigh the vruntime

- If process has run for *t ms*, then

  *vruntime += t * (weight based on nice of process)*

- A lower priority implies time moves at a faster rate compared to that of a high priority task

I/O and CPU Bound Processes

- What we need,
  - ✓ I/O bound should get higher priority and get a longer time to execute compared to CPU bound
  - ✓ CFS achieves this efficiently
    - I/O bound processes have **small CPU bursts** therefore will have a **low vruntime**. They would appear towards the left of the tree…. Thus are given **higher priorities**
    - **I/O bound processes** will typically have larger time slices, because they have **smaller vruntime**

New Process

- Gets added to the RB-tree
- Starts with **an initial value** of min_vruntime..
- This ensures that it gets to execute quickly

Summary

- How to Boot an Embedded System (ES)
  - ✓ Grub for x86 Architecture
  - ✓ U-boot for ARM Architecture
  - ✓ How to Implement a Bootloader in an Embedded System.
- Microkernel for Embedded System
  - ✓ What is Process?
  - ✓ The Process Concept in an Embedded System
  - ✓ The Practical Knowledge of Process Management