How to develop an SLOS (Simple Light OS) from scratch

Kwangdo Yi

Scope

- Let's focus on what I have done.
  - This presentation is not for explaining Linux, nor ARM assembler.

- But I still try to embed a little of Linux to SLOS.
  - I did my best to touch the concepts in Linux.

- So, if you want to know about rbtree, just do googling. But if you want to know how I use it in SLOS, I can answer you.

Contents

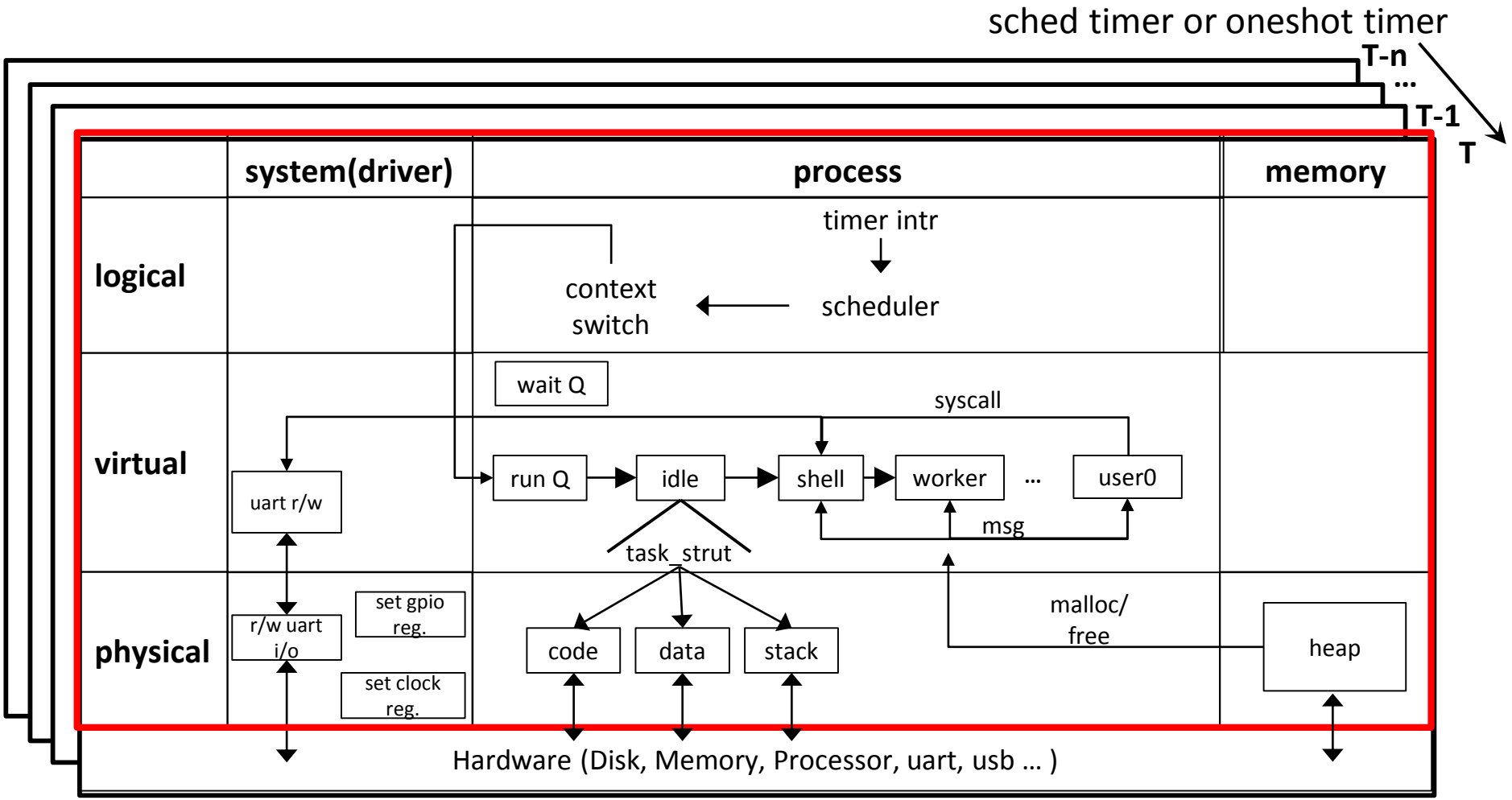
1. Motivations - why/what SLOS?
2. Development environment - target board, debugger, bootloader, build env
3. Memory map and linker, ARM registers, exception vectors
4. Interrupt and timer framework
5. Task, fork
6. Simple memory manager
7. Complete Fair Scheduler and context switching
8. Task synchronization
9. Syscalls
10. User application and elf loader
11. ramdisk for user application
12. Simple drivers – uart
13. Let’s see it !!

Motivations - Why SLOS?

- Hard to study Linux.
  - big - As of 2013, the Linux 3.10 had 15,803,499 lines of code.
  - fast change - millions of developer try to…
  - wide area - From arch to network + ….
- But core concepts in OS must be simple.
  - May help me to understand Linux.
- Most importantly, for curiosity and fun - I want to touch a small portion of ARM bring-up processes and some concepts of OS.

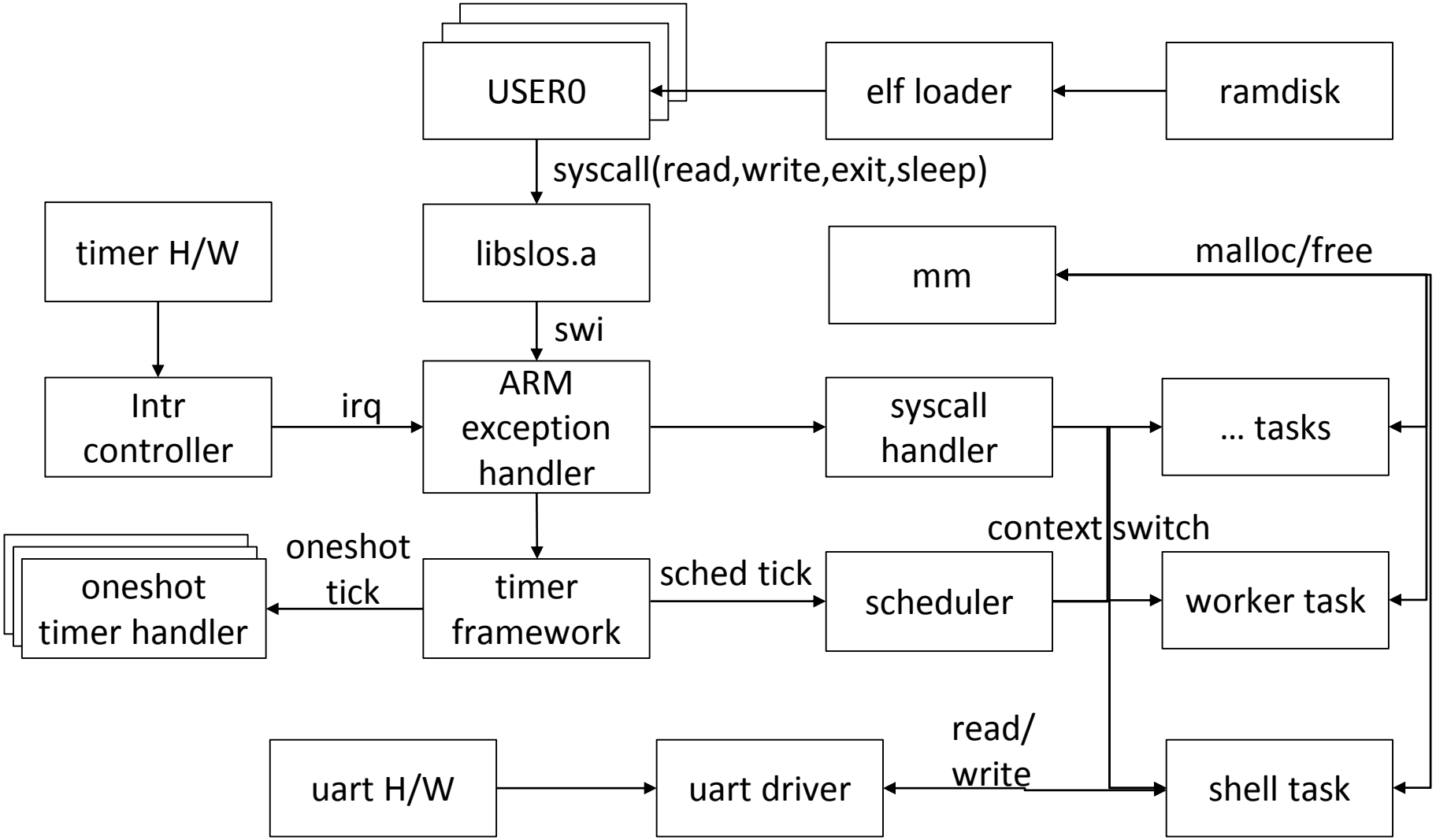
Motivations - What is SLOS?(1/3)

❑ Basis of SLOS(time line view)



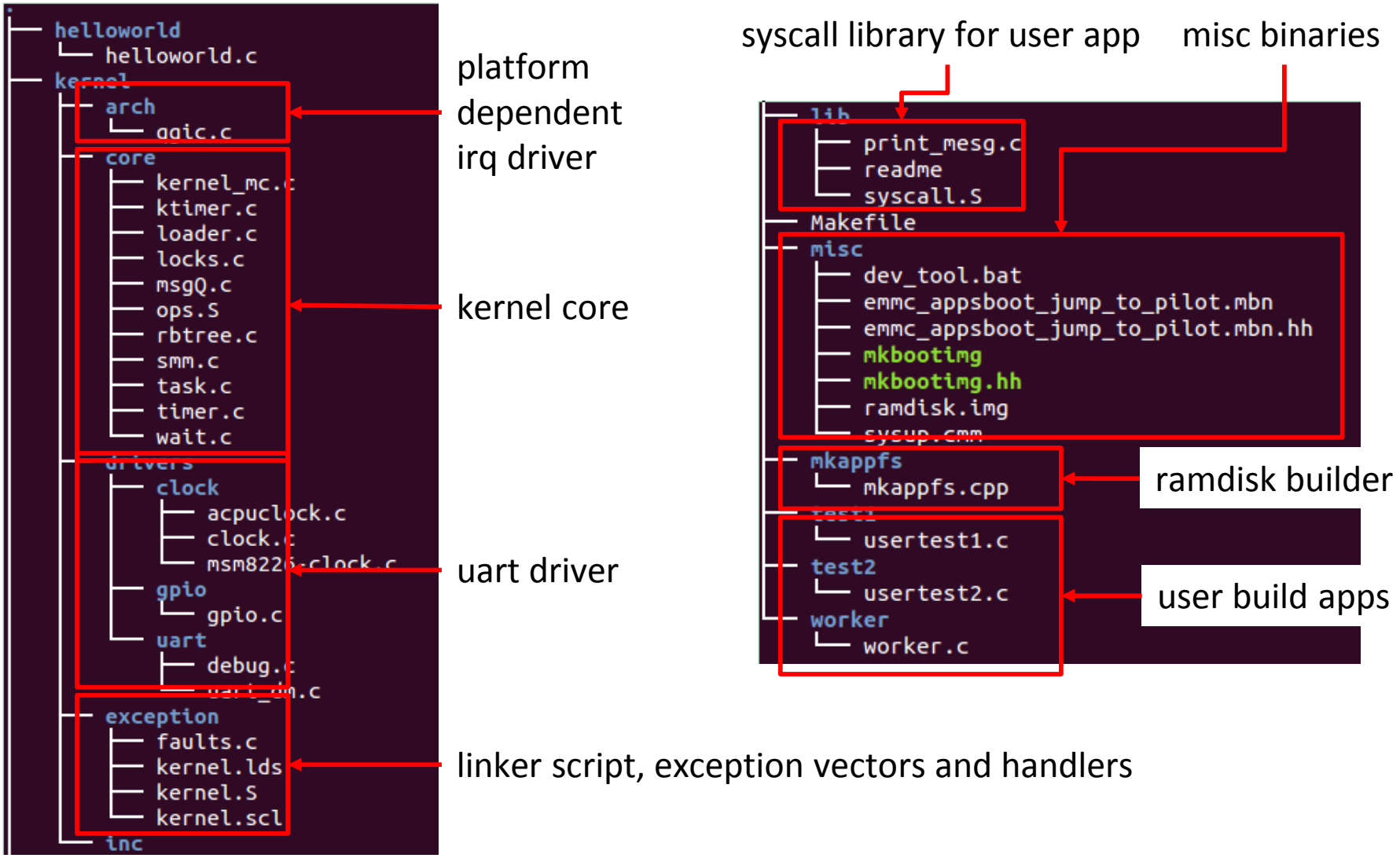
Motivations – What is SLOS?(2/3)

❑ Basis of SLOS (functional modules view)

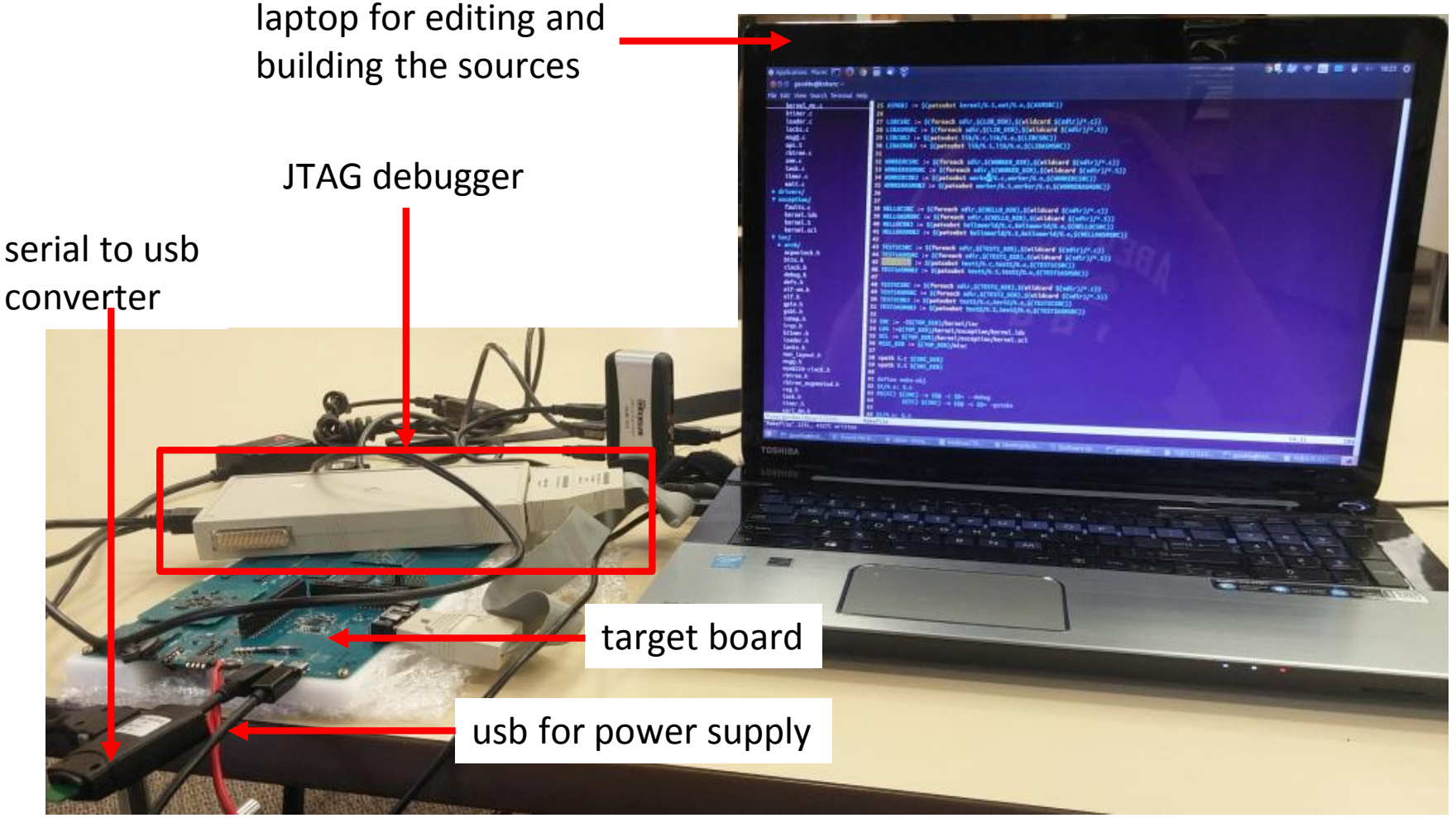


Motivations - What is SLOS?(3/3)

- Source tree



Development environment



Development environment – target board

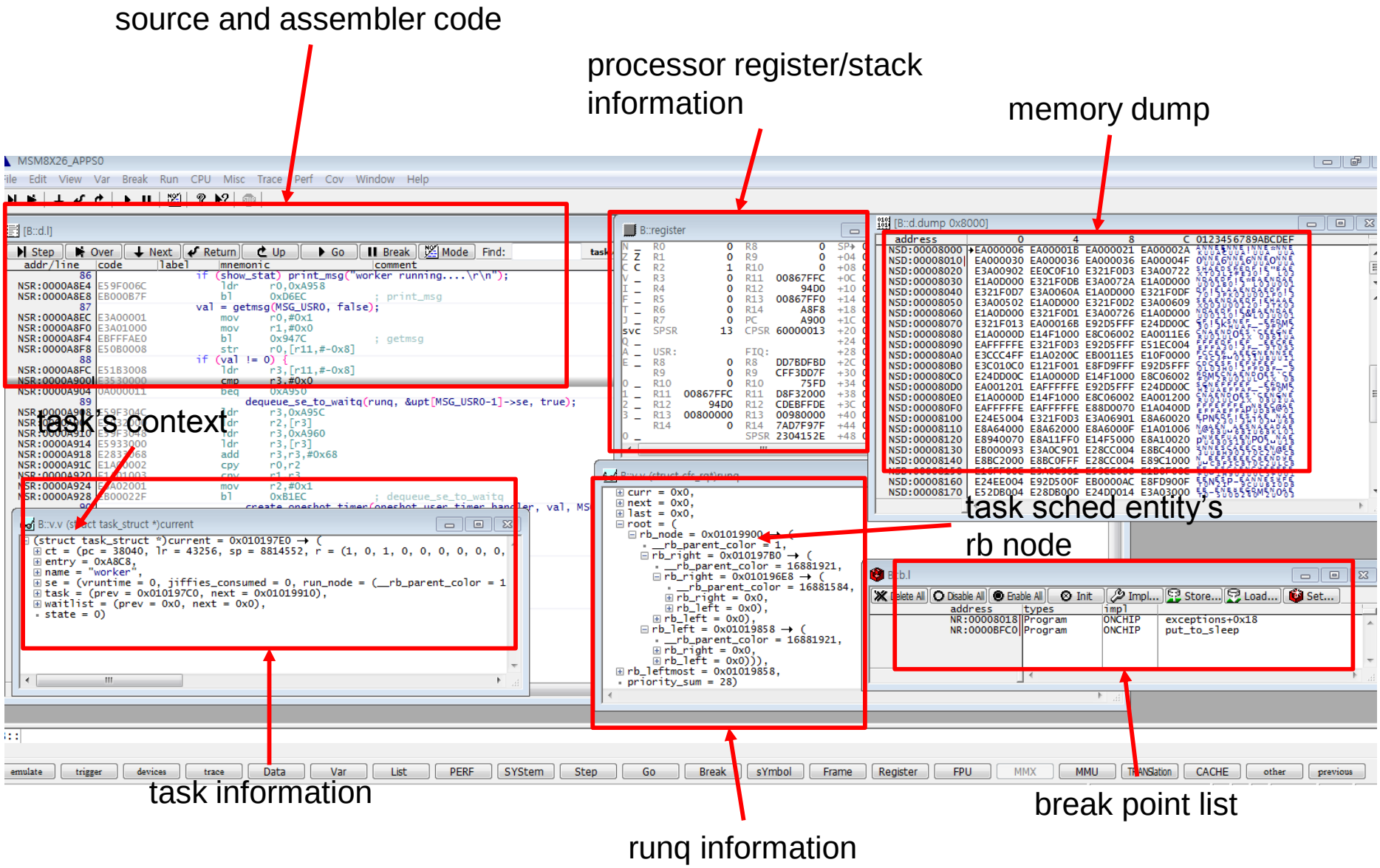
- H/W spec.
  - Application processor : qualcomm msm8626
  - cpu architecture: ARM Coretex-A7 MPCore
  - Debugging : JTAG, uart Rx/Tx
  - Memory : 1GB LPDDR2
  - Storage : 16GB eMMC
  - Peripheral : LCD, WiFi, Bluetooth, etc.

Debugger

- JTAG debugger
  - Lauterbach trace32 debugger is used.
  - Can break processor, see every variables, cpu registers, dump memory, assembler code, etc.
  - Need to add symbol option when build sources.
  - Expensive but very powerful!!

- Uart serial debugger
  - Uart port is connected to laptop via usb2serial.
  - Uart is used for both debugging and shell terminal.
  - print_msg is used for debugging.
  - shell task get the user input through uart.

Debugger example – Trace32 debugger screen



Bootloader

- Reuse android bootloader(LK) with a little change.
  - don’t care about the basic HW initialization.
    - it’s none OS stuff and very frustrating.
  - still fastboot downloader should work.
    - fastboot can download SLOS by “fastboot flash boot slos.img”
  - Reuse android mkbootimg to send hdr info to LK.
    - You can build it from android.
    - ‘mkbootimg’ merges kernel, ramdisk and android hdr into slos.img.
  - hdr information includes magic, kernel load location, kernel size, ramdisk location, etc.
    - Since android bootloader is used, I need an android hdr for bootloadder to load kernel.

Build environment(1/3) - toolchain

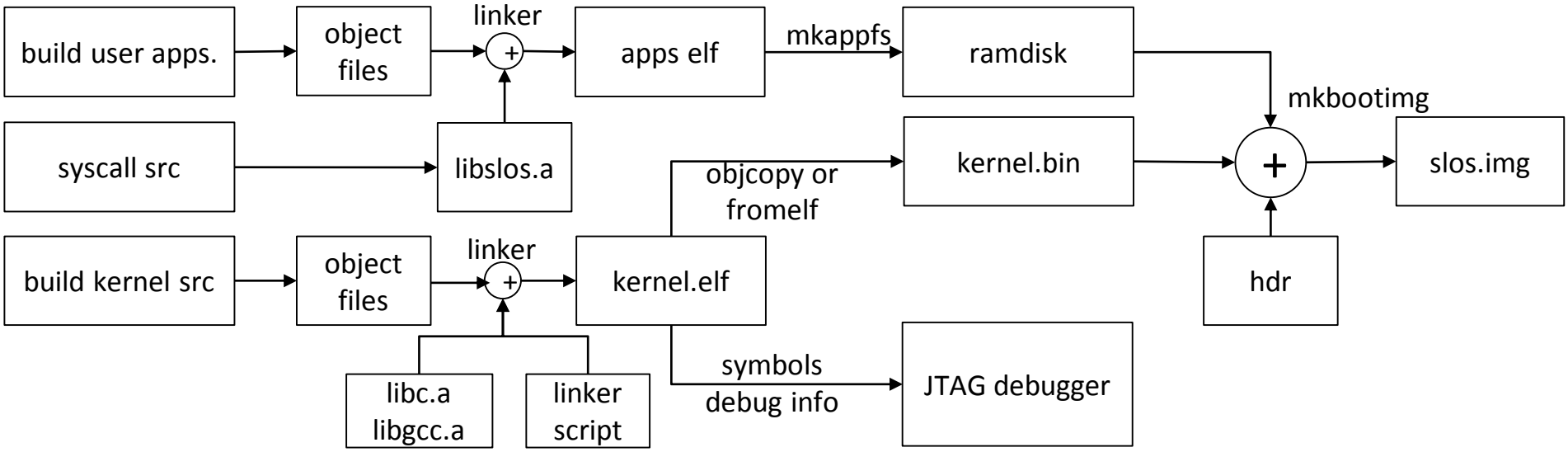
- RVCT(RealView Compilation Tool)
  - tool chain developed by ARM.
  - not free - need license and setup license server.
  - used for compiling non-HLOS like modem and other binaries.
- GCC(GNU Compiler Collection)
  - Collection of compilers/linker for c, c++...
  - If you are using ubuntu, install it by “sudo apt-get install gcc-arm-linux-gnueabi” for free.
  - Or you can download/install GCC toolchain from mentor graphics.
  - Set the path correctly in Makefile or your env file.
  - Makefile and linker script syntax are a little bit different with each other.

Build environment(2/3) - Makefile

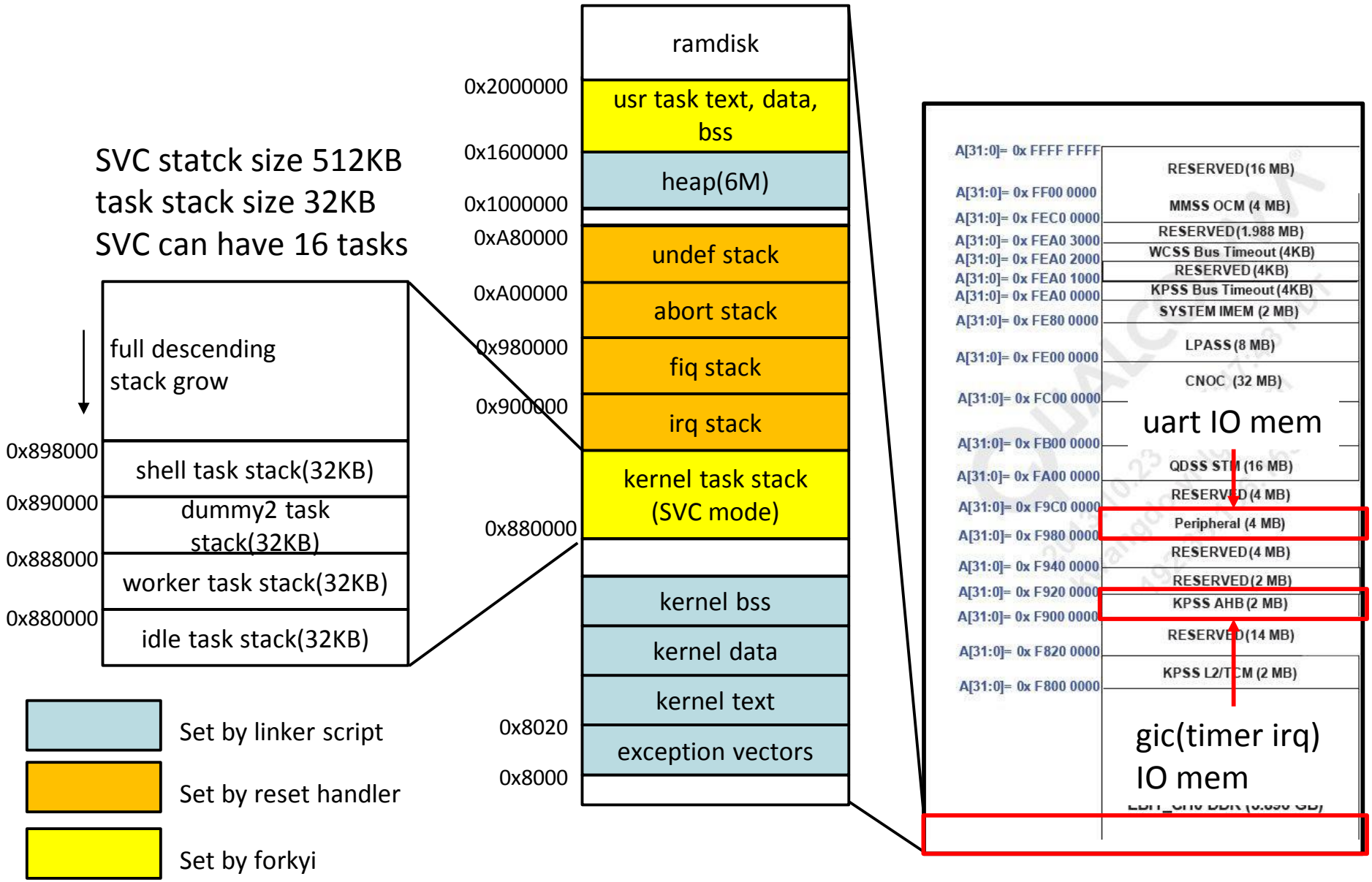
- Ubuntu 14.04 is a host used for editing, cross-compiling, downloading and serial terminal via uart.
- Win7 is used for trace32 debugger.
  - Virtualbox is used for this.
    - ubuntu is a host, win7 is a client OS.
- Build with GCC is recommended.
  - Free and easy to install.
  - just type ‘make’ and it does everything.
    - ‘make’ traverse the source tree, catches all changes and build sources as described by Makefile.
  - Object files are located in out/each_path/filename.o
  - ‘make’ calls linker for linking binaries and call ‘mkbootimg’ to merge kernel, ramdisk and header.

Build environment(3/3) - Build process

- Linker script is used for
  - combining object files and library(libc).
  - set the locations of entry point, text, data, bss and heap with sections for memory layout.
  - Scatter load file(.scl) for RVCT or gnu linker script(.lds) for GCC is used.
- Build process

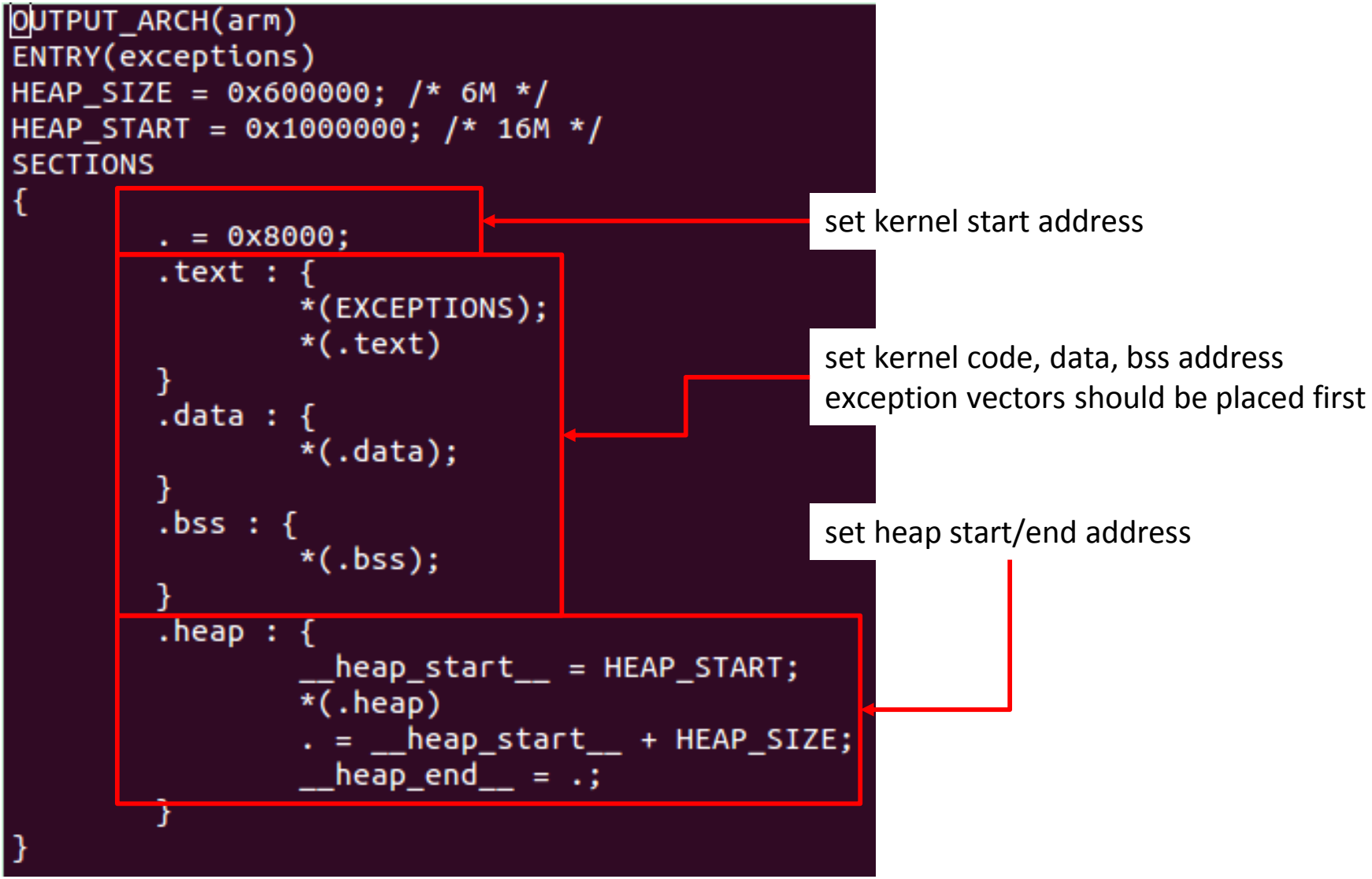


Memory map

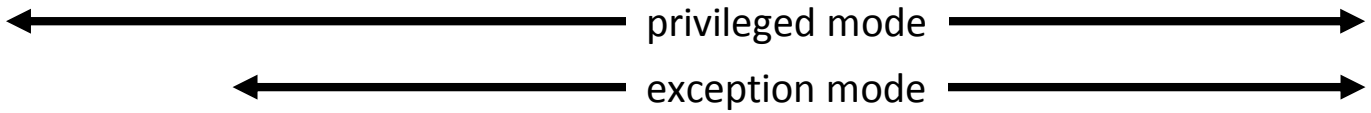

SVC statck size 512KB
task stack size 32KB
SVC can have 16 tasks
full descending
stack grow
0x898000
shell task stack(32KB)
0x890000
dummy2 task stack(32KB)
0x888000
worker task stack(32KB)
0x880000
idle task stack(32KB)
ramdisk
0x2000000
usr task text, data, bss
0x1600000
heap(6M)
0x1000000
0xA80000
undef stack
0xA00000
abort stack
0x980000
fiq stack
0x900000
irq stack
kernel task stack (SVC mode)
0x880000
kernel bss
kernel data
kernel text
0x8020
exception vectors
0x8000
A[31:0]= 0x FFFF FFFF
RESERVED(16 MB)
A[31:0]= 0x FF00 0000
MMSS OCM (4 MB)
A[31:0]= 0x FEC0 0000
RESERVED(1.988 MB)
A[31:0]= 0x FEA0 3000
WCSS Bus Timeout (4KB)
A[31:0]= 0x FEA0 2000
RESERVED (4KB)
A[31:0]= 0x FEA0 1000
KPSS Bus Timeout (4KB)
A[31:0]= 0x FEA0 0000
SYSTEM IMEM (2 MB)
A[31:0]= 0x FE80 0000
LPASS (8 MB)
A[31:0]= 0x FE00 0000
CNOC (32 MB)
A[31:0]= 0x FC00 0000
uart IO mem
A[31:0]= 0x FB00 0000
QDSS STM (16 MB)
A[31:0]= 0x FA00 0000
RESERVED (4 MB)
A[31:0]= 0x F9C0 0000
Peripheral (4 MB)
A[31:0]= 0x F980 0000
RESERVED(4 MB)
A[31:0]= 0x F940 0000
RESERVED(2 MB)
A[31:0]= 0x F920 0000
KPSS AHB (2 MB)
A[31:0]= 0x F900 0000
RESERVED(14 MB)
A[31:0]= 0x F820 0000
KPSS L2/TCM (2 MB)
A[31:0]= 0x F800 0000
gic(timer irq)
IO mem
Set by linker script
Set by reset handler
Set by forkyi

Linker script

- Linker script is used for setting the locations of …



ARM processor modes and registers(1/4)-32bit architecture



user	system	FIQ	supervisor	abort	IRQ	undefined
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8_fiq	R8	R8	R8	R8
R9	R9	R9_fiq	R9	R9	R9	R9
R10	R10	R10_fiq	R10	R10	R10	R10
R11	R11	R11_fiq	R11	R11	R11	R11
R12	R12	R12_fiq	R12	R12	R12	R12
R13	R13	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14	R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15	R15	R15	R15	R15	R15	R15

general purpose register

banked register

cpsr	cpsr	cpsr	cpsr	cpsr	cpsr
	spsr_fiq	spsr_svc	spsr_abt	spsr_irq	spsr_und

special purpose register

ARM processor modes and registers(2/4)-32bit architecture

- Exception mode
  - fiq, svc, irq, abort, undefined
- Privileged mode vs. Unprivileged mode
  - The privileged software can use all the instructions and has access to all resources.
  - The unprivileged SW has limited access to the MSR and MRS instructions, and cannot use the CPS instruction. This has restricted access to system resources(memory, peripheral, etc).

- SLOS always runs in exception mode.
  - not support user, system mode.
  - All tasks run in svc mode.

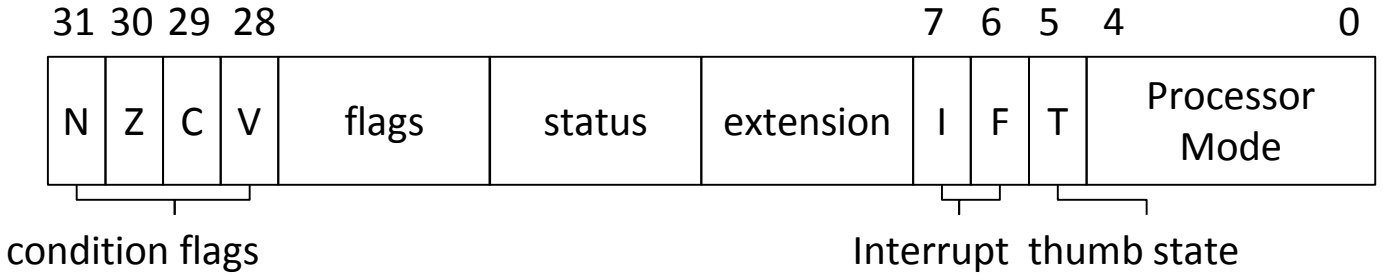
ARM processor modes and registers(3/4)-32bit architecture

❑ ARM registers – general purpose

register number	alternative register name	ATPCS(ARM-Thumb procedure call standard) register usage
r0	a1	Argument registers. These hold the first four function arguments on a function call and the return value on a function return. A function may corrupt these registers and use them as general scratch registers within the function.
r1	a2	
r2	a3	
r3	a4	
r4	v1	General variable registers. The function must preserve the callee values of these registers.
r5	v2	
r6	v3	
r7	v4	
r8	v5	
r9	v6 sb	The function must preserve the callee value of this register except some specific cases.
r10	v7 sl	
r11	v8 fp	
r12	ip	A general scratch register that the function can corrupt.
r13	sp	The stack pointer, pointing to the full descending stack.
r14	lr	The link register. On a function call this holds the return address.
r15	pc	The program counter.

ARM processor modes and registers(4/4)-32bit architecture

- ARM registers – special purpose
  - CPSR : Current Program Status Register
  - SPSR : Saved Program Status Register



Mode	Source	PSR[4:0]	Symbol	Purpose
User	-	0x10	USR	Normal program execution mode
FIQ	FIQ	0x11	FIQ	Fast interrupt mode
IRQ	IRQ	0x12	IRQ	Interrupt mode
Supervisor	SWI, Reset	0x13	SVC	Protected mode for operating system
Abort	Prefetch abort, Data Abort	0x17	ABT	Virtual memory and/or memory protection mode
Undefined	Undefined instruction	0x1b	UND	Software emulation of hardware co-processors mode
System	-	0x1f	SYS	Run privileged operation system tasks mode

Exception vectors(1/2)

exception	offset from vector base	mode on entry	F bit on entry	I bit on entry	action
reset	0x00	supervisor	disabled	disabled	branch its handler routine
undefined instruction	0x04	undefined	unchanged	disabled	
software interrupt	0x08	supervisor	unchanged	disabled	
prefetch abort	0x0c	abort	unchanged	disabled	
data abort	0x10	abort	unchanged	disabled	
reserved	0x14	reserved	-	-	
IRQ	0x18	irq	unchanged	disabled	
FIQ	0x1c	fiq	disabled	disabled	

- MSM chipset consists of many processors.
  - Q : How can I place exception vectors where I want?
  - A : set the VBAR(vector base address) as

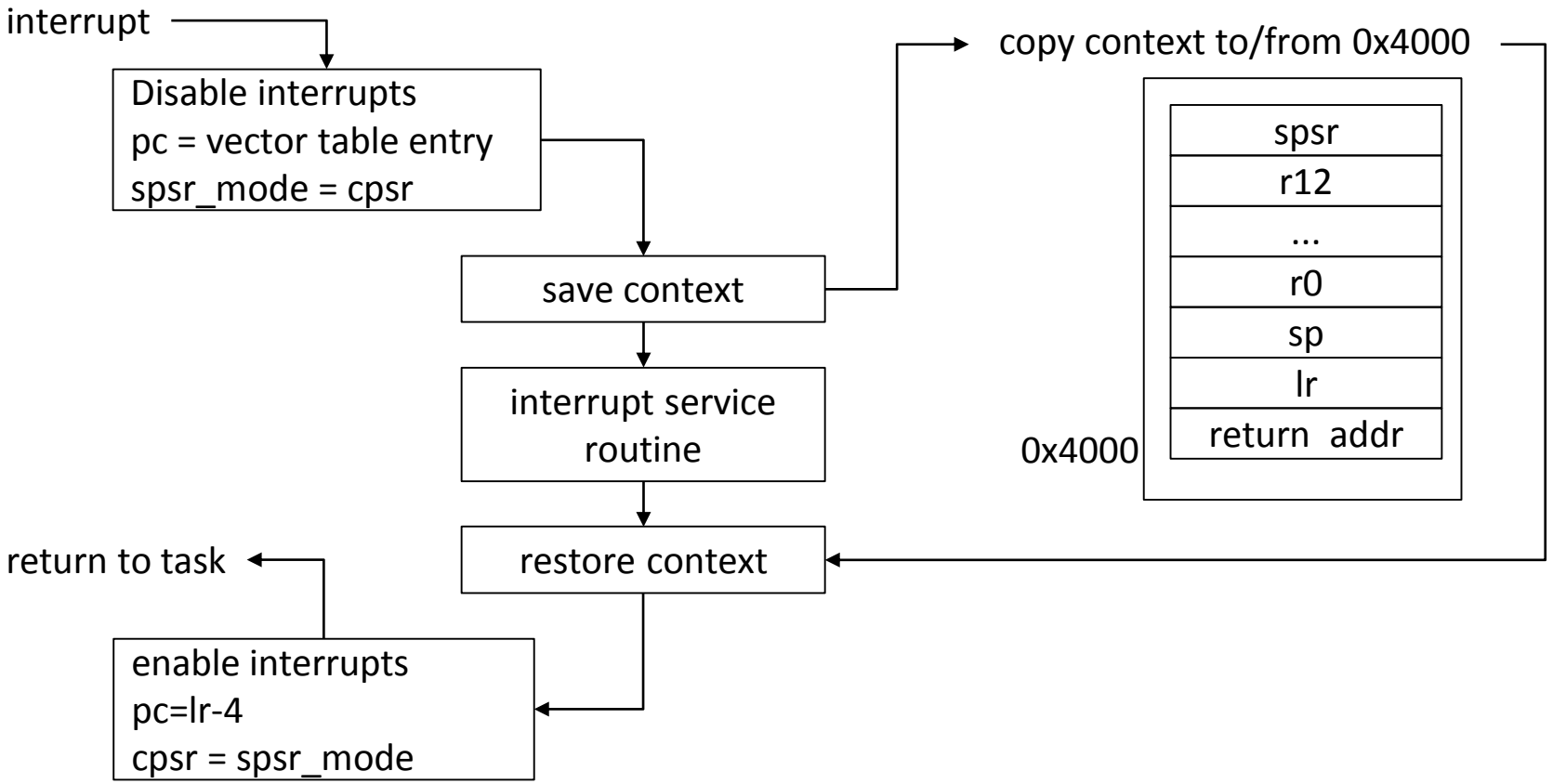
```
/*set VBAR with 0x8000*/
ldr     r0, =0x8000
mcr     p15,0,r0,c12,c0,0
```

Exception vectors(2/2)

- Exception vectors
  - are coded by RVCT assembler, GNU assembler
    - RVCT and GNU assembler are almost the same.
  - ‘bl’ cmd to each handler is in exception vectors.
- Reset vector
  - sets the VBAR and
  - sets the stack for each ARM mode(fiq, irq, …) and
  - jumps to main start routine.
- Undefined, abort, prefetch, fiq are just jump to infinite loop.
  - There is nothing to do.
- SLOS doesn’t support SYS, USR mode.

SLOS interrupt(1/2) – interrupt handler

- Simple non-nested interrupt handler.
  - doesn't allow another interrupt while serving interrupt.
  - doesn't support fiq.
- What does the interrupt handler do?



SLOS interrupt(2/2) – interrupt service routine

- Interrupt handler has an ISR vector.

```
typedef int (*int_handler)(void *arg);
struct ihandler {
        int_handler func;
        void *arg;
};
struct ihandler handler[NR_IRQS];
```

- ISR is registered as

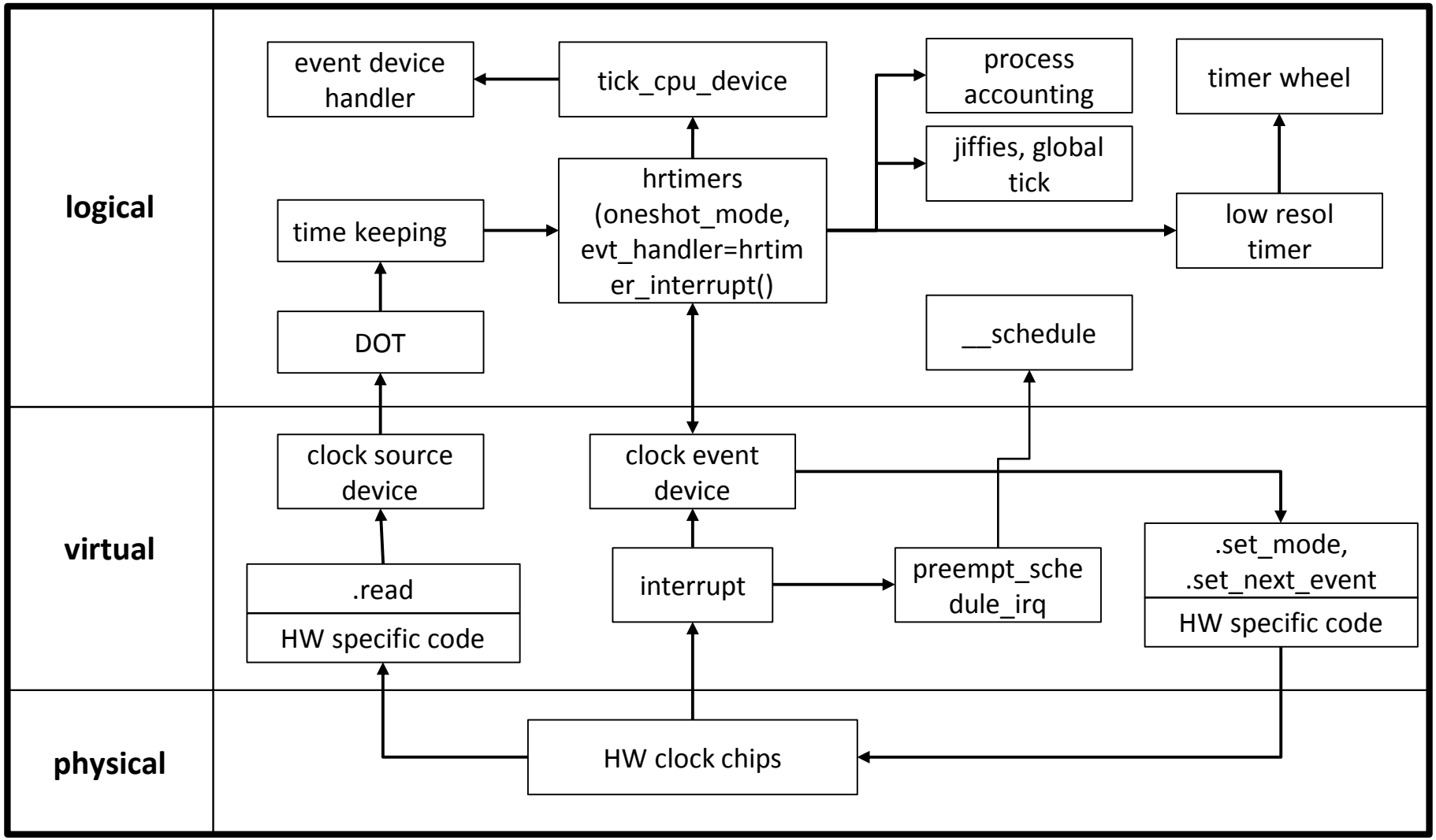
```
/* Register interrupt handler */
void gic_register_int_handler(unsigned int vector, int_handler func, void *arg)
{
        /* enter_critical_section(); */
        handler[vector].func = func;
        handler[vector].arg = arg;
        /* exit_critical_section(); */
}
```

- A correct ISR is called by checking intr number.

```
num = readl(GIC_CPU_INTACK);
if(num == INT_QTMR_FRM_0_PHYSICAL_TIMER_EXP) {
        ret = handler[num].func(frame);
```

Timer framework(1/5) – Linux

❑ Linux timer is



Timer framework(2/5) – Linux timer

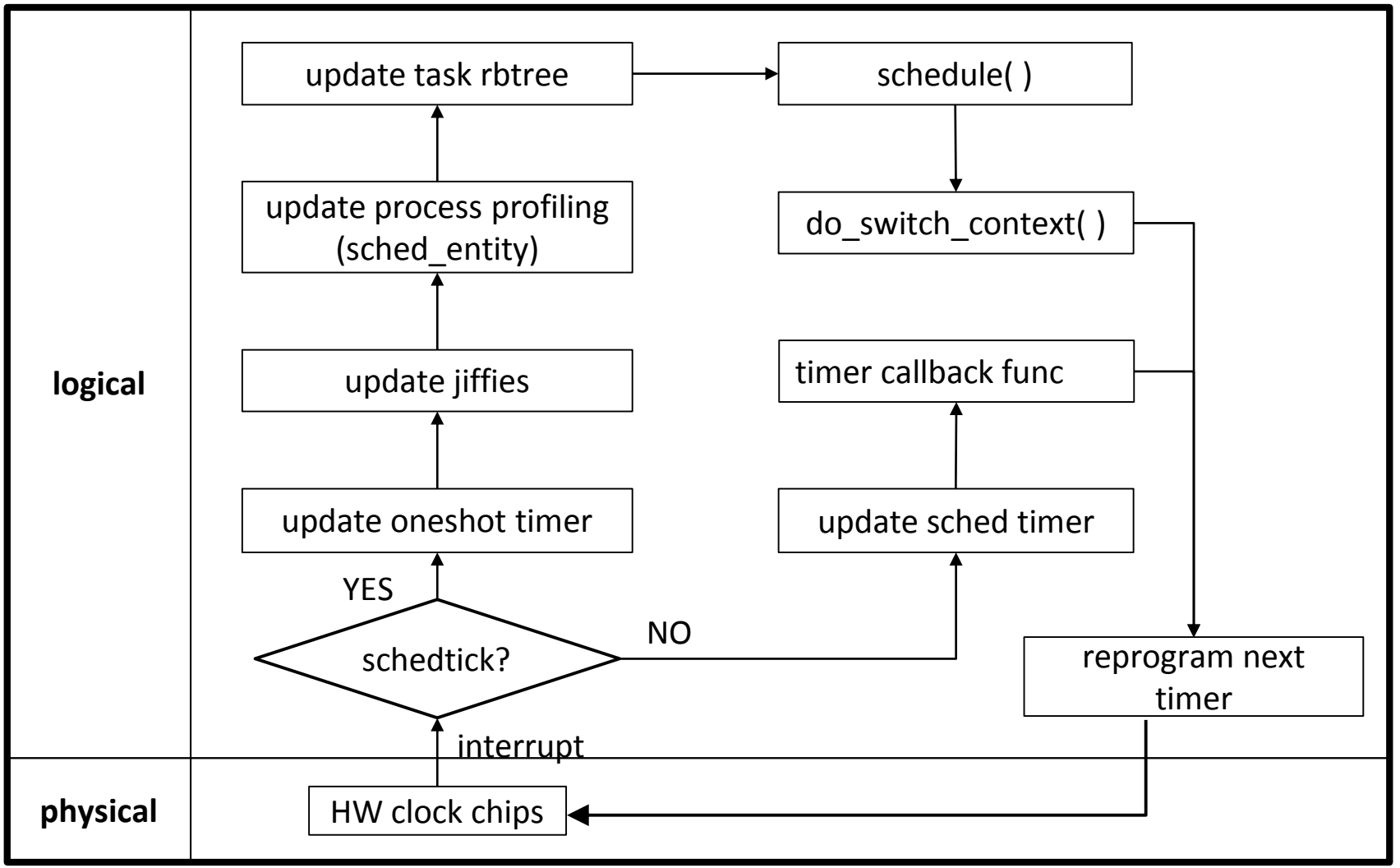
- Linux timer is used for
  - time keeping, time-of-day representation.
  - a quantum for scheduling.
  - process profiling.
  - in-kernel timers.
- Possible timekeeping configurations in Linux.

High-res Dynamic ticks	High-res Periodic ticks
Low-res Dynamic ticks	Low-res Periodic ticks

  - Normally high-res dynamic or high-res periodic ticks are used.

Timer framework(3/5) – SLOS

❑ Definitely, SLOS timer is simple.

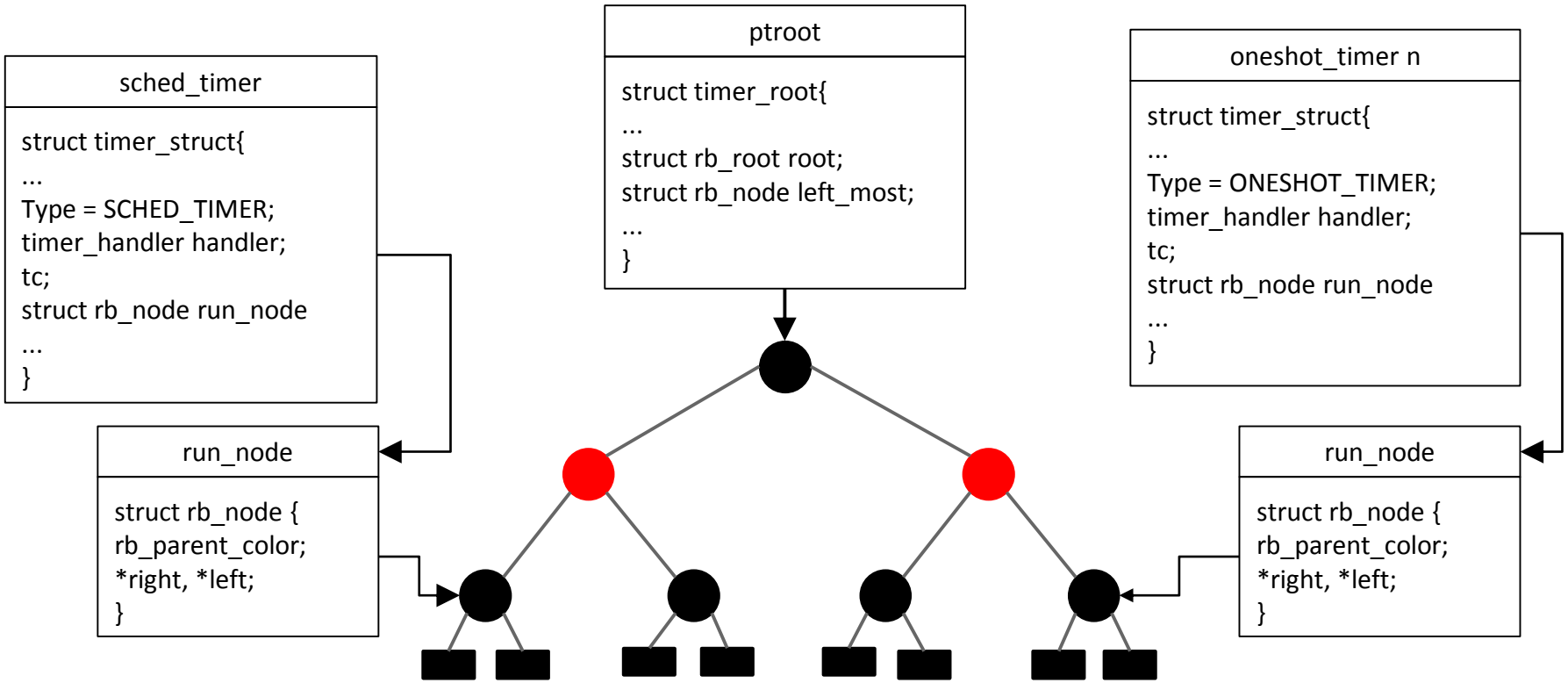


Timer framework(4/5) – SLOS timer

- SLOS timer
  - is the only resource for the scheduler to schedule tasks,
  - has no virtual layer dependent on each platform ,
  - has no clock source device - doesn’t support TOD,
  - is fired every 10ms – periodic tick,
  - also support oneshot timer in timer list.

Timer framework(5/5) – SLOS timer list

- rb tree is used for timer list.
- Timer list has both shed tick timer and oneshot timer.
  - leftmost node is the next firing timer.
  - leftmost node could be sched tick or oneshot tick.



Task(1/8) – struct task_struct

- Task is represented by TCB(Task Control Block).

```
struct task_context_struct {
        uint32_t pc;
        uint32_t lr;
        uint32_t sp;
        uint32_t r[13];
        uint32_t spsr;
};
```

task context(snapshot of a processor, virtual state of a processor)

```
struct task_struct {
        struct task_context_struct ct;
        /*struct task_context_struct ct;*/
        task_entry entry;
        char name[32];
        struct sched_entity se;
        struct list_head task;
        struct list_head waitlist;
        uint32_t state;
};
```

task entry point

task sched entity

task state(running, waiting…)

- sched_entity is the entity used by scheduler.

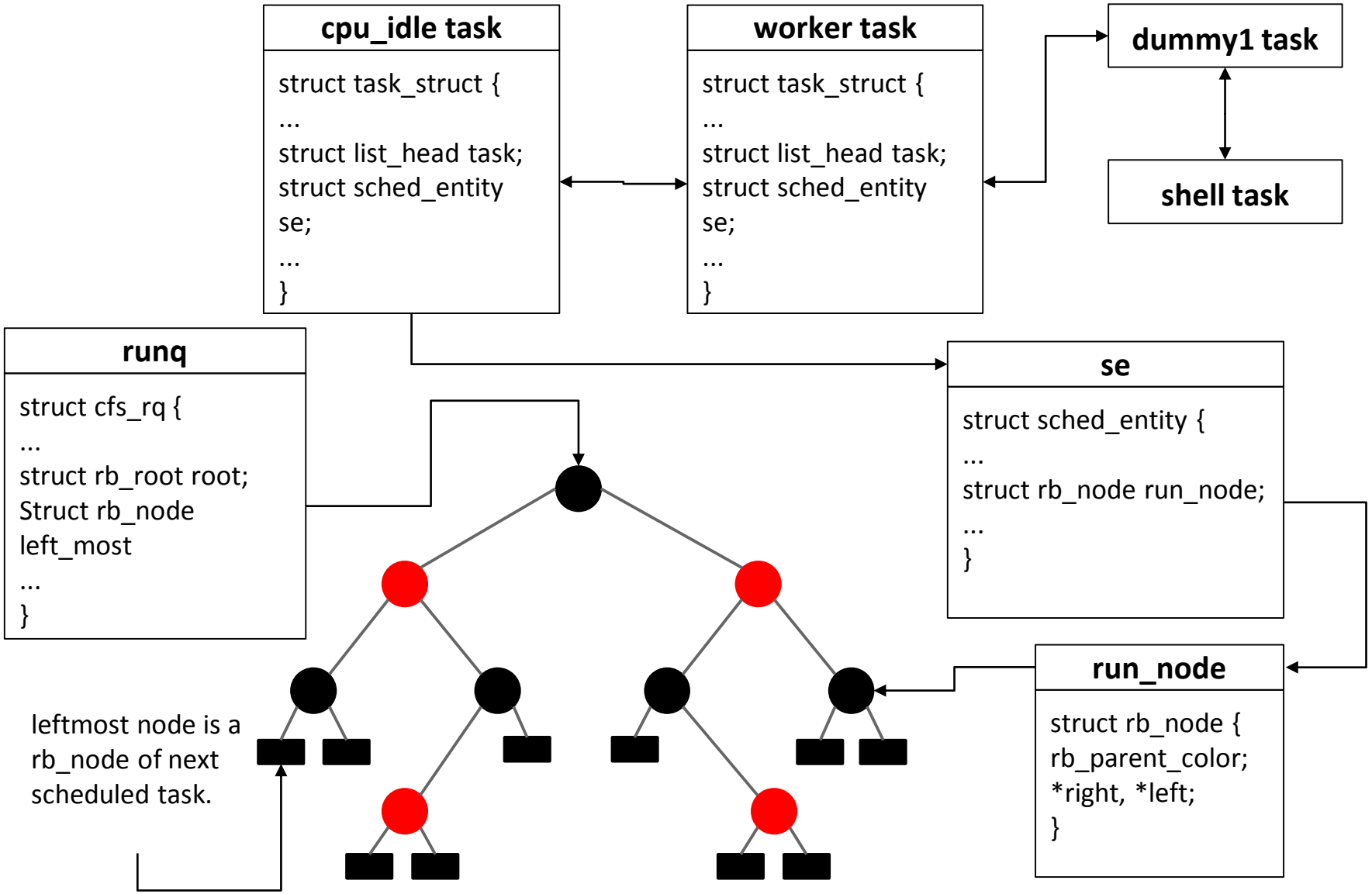
```
struct sched_entity {
        uint64_t vruntime;
        uint64_t jiffies_consumed;
        struct rb_node run_node;
        uint32_t priority;
};
```

virtual runtime for CFS

actual runtime

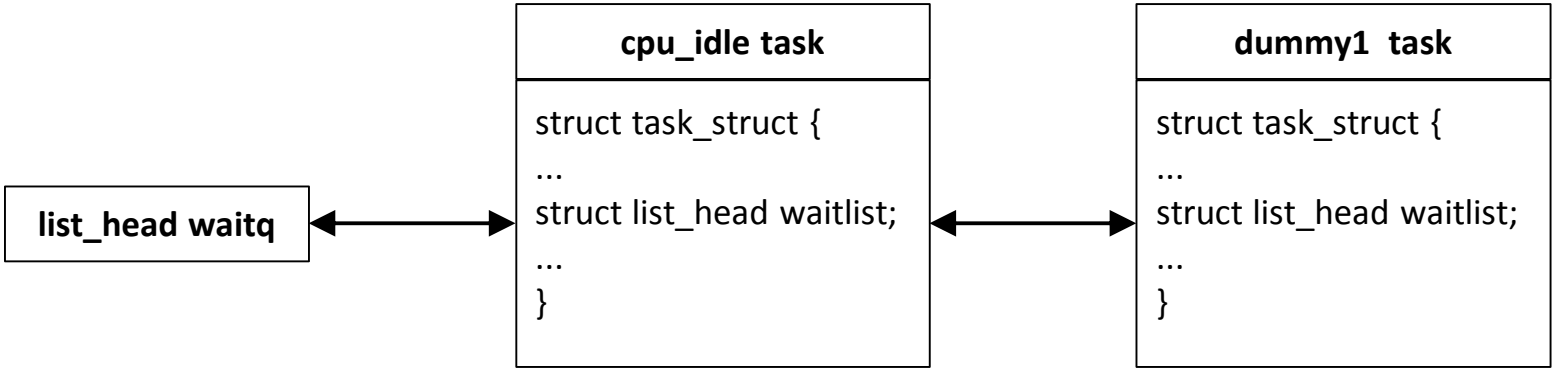
rb node for rb tree of runQ

Task(2/8) – rb tree for runQ



Task(3/8) – waitQ

❑ waitQ is a doubly linked list.



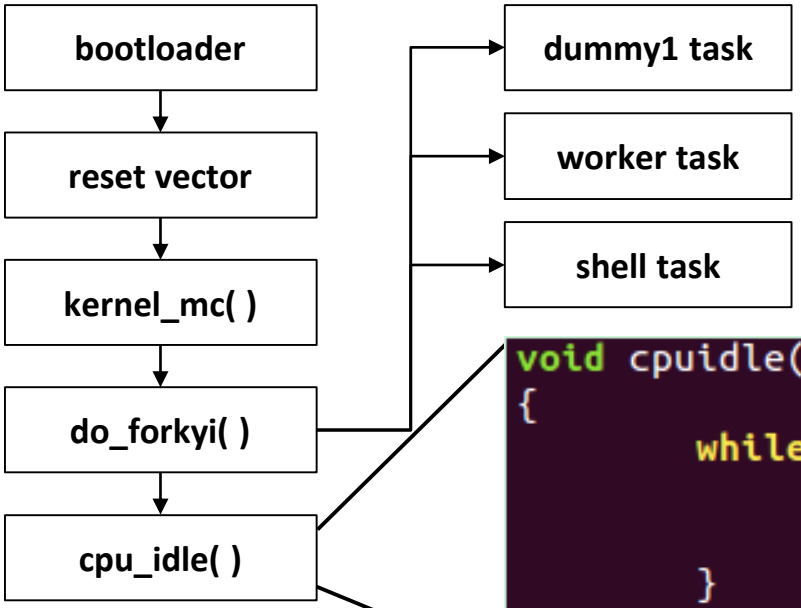
❑ Currently add to waitQ and remove from waitQ functions are working.

❑ dequeue to waitQ of current task is not supported.

- yield() function is not supported!

Task(4/8) – Fork

- do_forkyi() can spawn process by doing
  - malloc task and
  - add task into task list
  - init entry point, init stack, init local vars.
- After do_forkyi(), task need to be enqueued to runq.
- SLOS booting process




```
void cpuidle(void)
{
        while(1) {
                drop_usrtask();
                if (show_stat) print_msg("cpuidle ru
        }
}
```

Task(5/8) – do_forkyi() and enqueue

```
struct task_struct *do_forkyi(char *name, task_entry fn, whoami pf)
{
        struct task_struct *pt;
        if (task_created_num == MAX_TASK) return;

        pt = (struct task_struct *)malloc(sizeof(struct task_struct));
        sprintf(pt->name,name);
        pt->entry = fn;
        pt->se.vruntime = 0;
        pt->se.jiffies_consumed = 0;
        pt->ct.sp = (uint32_t)(SVC_STACK_BASE + TASK_STACK_GAP * ++task_created_num);
        pt->ct.lr = (uint32_t)pt->entry;
        pt->ct.pc = (uint32_t)pt->entry;
        pt->ct.spsr = SVCSPSR;
        pt->pfwhoami = pf;
        /* get the last task from task list and add this task to the end of the task list*/
        last->task.next = &(pt->task);
        pt->task.prev = &(last->task);
        pt->task.next = &(first->task);
        first->task.prev = &(pt->task);
        last = pt;

        return pt;
}
```

TCB memory

TCB block initialization

Update task linked list

```
temp = do_forkyi("shell", (task_entry)shell, (whoami)iamshell);
set_priority(temp, 2);
rb_init_node(&temp->se.run_node);
enqueue_se_from_idle(runq, &temp->se, true);
```

enqueue task to runq

Task(6/8) – cpu_idle task

❑ SLOS booting process becomes cpu_idle task after forking all other tasks.

❑ cpu_idle task has the lowest priority.

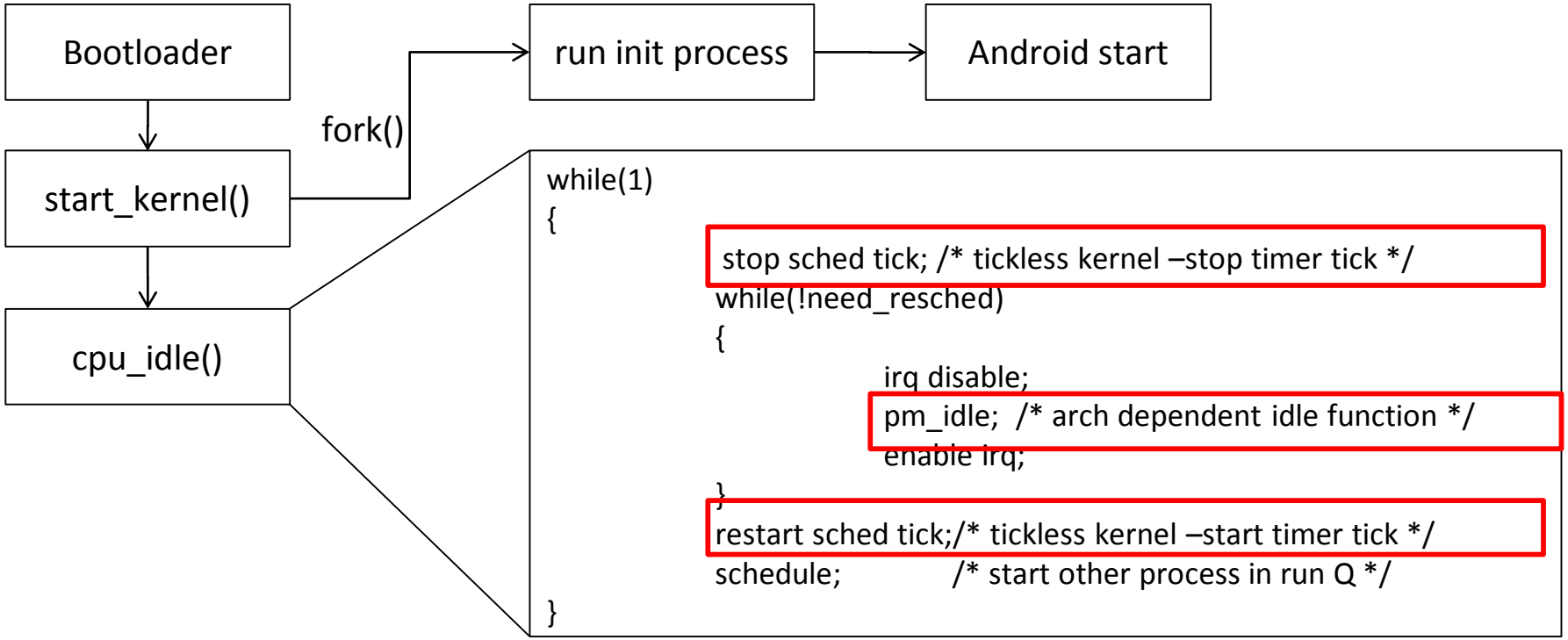
❑ cpu_idle is important for power management. In Linux, power management routine for cpu is in here.

- Since there is no jobs to work, cpu should be in sleep to save power.
- cpu idle governor decide the sleep state based on latency.
  - If idle period is long, deep sleep. if not long, light sleep.

❑ In SLOS, cpu_idle task has drop_usr_task() to drop the tasks which finish their jobs.

Task(7/8) – Linux cpu_idle task

- start_kernel finally becomes cpu_idle process after booting finished.
- idle process is the lowest priority process.
  - cpu_idle is the entry point of power management.
  - Tickless kernel is supported since 2.6.



Task(8/8) – shell task, woker task

- shell task polls the uart port to get the user input.
  - checks one char from uart and execute corresponding routine.
  - is one of kernel tasks, not user task.
  - has the highest priority for responsiveness.
  - ‘d’ for show current task, ‘t’ for display task info, ‘l’ for load user app, etc.

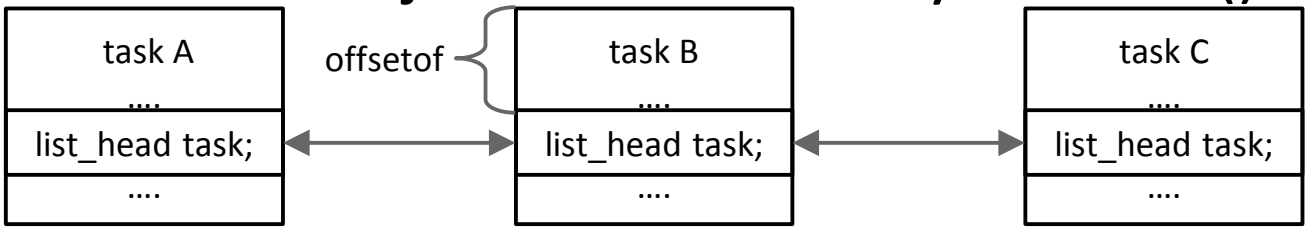
- worker task is like Linux kworker thread.
  - In SLOS, work list is predefined and can be set/get in msgQ.
  - Currently only one work is defined for sleep syscall.
    - set the user task0 to sleep and wake up.

misc for SLOS

- Objects are linked each other by using listhead.
  - task list, waitQ list…

```
struct list_head {
        struct list_head *prev, *next;
};
```

- Pointer to object is obtained by offsetof() in libc library.



- ‘container_of macro’ is using offsetof().
  - for using rb tree, SLOS needs to link with libc.

```
#define container_of(ptr, type, member) \
        ((type *)((unsigned int)ptr-offsetof(type, member)))
```

- rb tree sources are copied from Linux.
  - rbtree.c and rbtree.h are enough to use rbtree.

Simple memory manager(1/2)

❑ SLOS has a simple memory manager to serve the memory request from tasks.

- SLOS has 6MB heap for this.

❑ Simple memory manager does just alloc memory.

- Updating heap pointer is all.
- There are no pages, no memory pools for cache.
- There should be internal/external fragmentations but don't care because SLOS should be simple and all memory usage is predefined.
- More elaborate memory manager could be added some day.

Simple memory manager(2/2)

- malloc()
  - To use malloc, we need to implement a syscall to return pointer to heap.
  - malloc() in libc library calls _sbrk() that should be implemented in SLOS.

```
void *_sbrk(void *reent, size_t incr)
{
        static unsigned char *heap = NULL;
        unsigned char *prev_heap;

        if (!heap) {
                heap = (unsigned char *)&__heap_start__;
        }

        prev_heap = heap;
        if ((heap + incr) >= (unsigned char *)&__heap_end__) {
                return 0;
        }
        heap += incr;
        return (void *) prev_heap;
}
```

Scheduler

- Scheduler is based on timer framework.
  - sched_tick is the triggering point of scheduling.
  - Task itself can explicitly call schedule().
- Short notes on linux scheduler
  - O(n) scheduler was for linux2.4.
  - O(1) scheduler was for early version of linux 2.6.
    - scheduler takes constant time to schedule processes.
    - provide soft realtime scheduling.
    - realtime processes can preempt regular processes.
  - CFS(complete fair scheduler) is default scheduler since linux 2.6.23.
    - vruntime(virtual runtime) is a key to schedule.
    - rb tree is used for a balanced operation(O(logn)).
  - Power aware scheduler is currently under developing.

Complete Fair Scheduler in SLOS(1/3)

- What is fair in CFS scheduler perspective?
  - CPU is shared by tasks according to each task's priority.
  - (example) if task 1 with pri 4, task 2 with pri 8, task 3 with pri 16, then this is fair.

	task 1	task 2	task 3
cpu computation power	57%	28.5%	14.3%

  - vruntime is a weighted(virtual) runtime of task.
  - jiffies_consumed is a real runtime of the task.
  - sched_entity has information for CFS scheduler.
  - sched_entity is managed in rb tree.

```
struct sched_entity {
        uint64_t vruntime;
        uint64_t jiffies_consumed;
        struct rb_node run_node;
        uint32_t priority;
};
```

Complete Fair Scheduler in SLOS(2/3)

- How to update vruntime?
  - vruntime=(jiffies_consumed)*(cur_pri/sum_pri);
  - if pri high, vruntime goes slower than real runtime.
    if pri low, vruntime goes faster than real runtime.

- How to update rbtree with vruntime?
  - SLOS reuse linux rbtree with search and insert implementation.
  - Comparison key in rb tree is a vruntime.
  - recalc vruntime and update rbtree in every sched_tick interrupt.

- leftmost node in rbtree is the next runnable task's sched_entity.

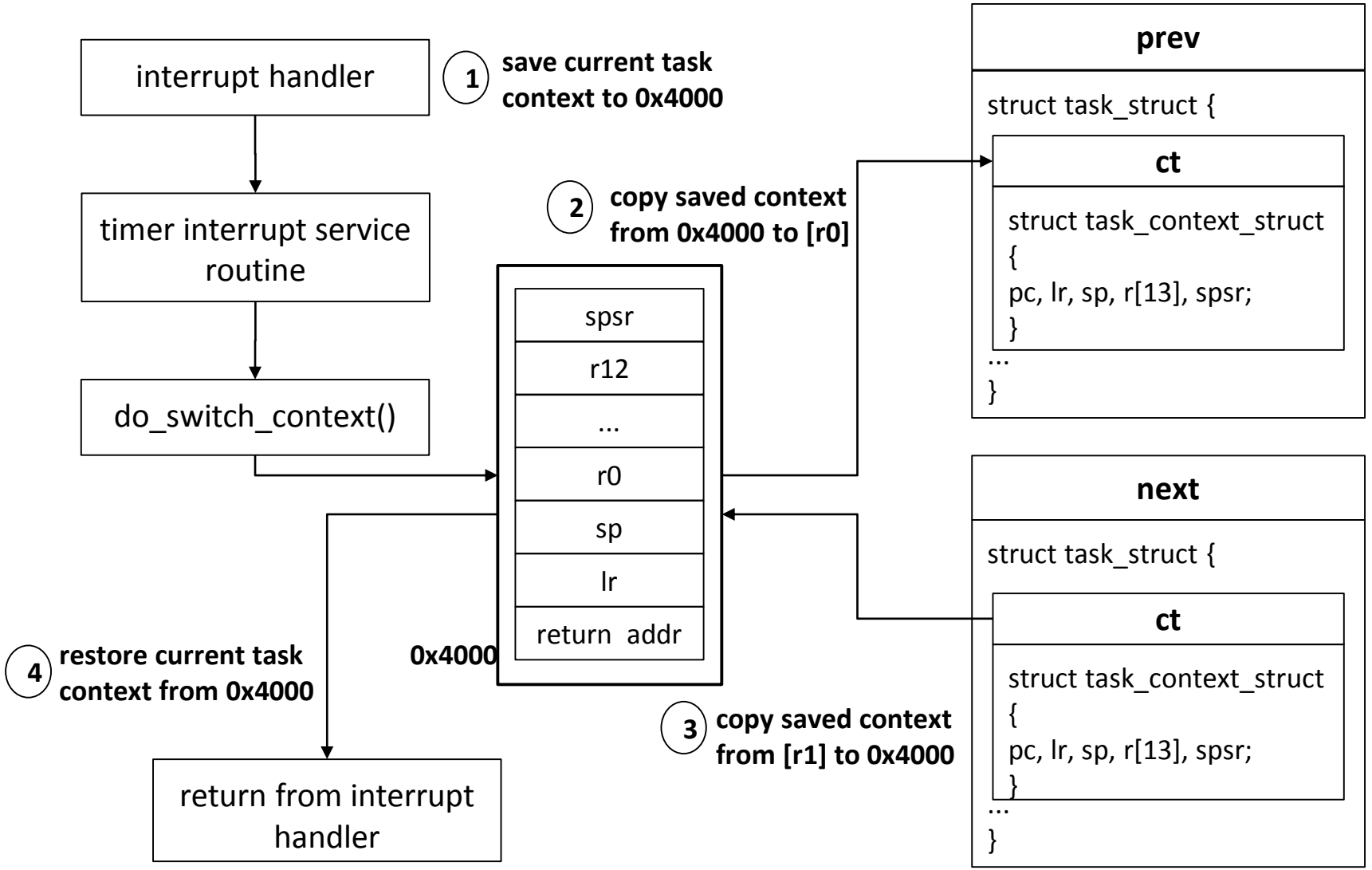
Complete Fair Scheduler in SLOS(3/3)

- Test of CFS scheduler in SLOS
  - Running for 522.6sec(total jiffies = 5226 tick)

	priority	expected cpu occupation - A	jiffies_consumed - B	delta (A-B)	vruntime
cpu_idle task	16	402(=5226*1/13)	404	-2	190
dummy1 task	8	804(=5226*2/13)	804	0	189
dummy2 task	8	804(=5226*2/13)	804	0	189
shell task	2	3216(=5226*8/13)	3214	2	189

Context switching

- do_switch_context(prev, next) in scheduler



Task synchronization(1/3)

- In order to access shared resources, tasks should be synchronized with each other.

- In case of UP(Uni Processor), interrupt enable/disable is enough.
  - No other tasks can kick in without interrupt.

- In case of MP(Multi Processor), atomic operations between processors are needed.
  - spinlock, mutex, semaphore.
  - declare a variable in external memory(not in cache, use volatile) and R/W exclusively. -> arch dependent.
  - Exclusive load/store(ldrex/strex) are supported in ARM.

Task synchronization(2/3)

- SLOS runs on UP.
  - SLOS has enable_interrupt/disable_interrupt for synchronization.

- SLOS also has spinlock and spinlock_irqsafe.
  - For UP, spinlock should be used very carefully.
    - In exception handlers(interrupt handler or syscall handler) which disable the interrupts, spinlock can let the processor(the only processor in UP) spin forever!!

Task synchronization(3/3)

❑ when/where?

```
volatile uint32_t uartlock;

int print_msg(const char *str)
{
        spin_lock_acquire_irqsafe(&uartlock);
        /*disable_interrupt();*/
        while (*str != 0) {
                uart_putc(0, *str++);
        }
        /*enable_interrupt();*/
        spin_lock_release_irqsafe(&uartlock);
```

before/after accss to shared resource that can be accessed by multiple tasks.

❑ how?

```
.global spin_lock_acquire
spin_lock_acquire:
        ldr     r1,=LOCKED
loop1:  ldrex   r2,[r0]
        strexne r2,r1,[r0]
        cmp     r2,#1 /* success:0, fail:1 */
        beq     loop1
        bx      lr
.global spin_lock_release
```

spinlock - architecture dependent codes to access memory exclusively.

```
.global enable_interrupt
enable_interrupt:
        mrs     r0,     CPSR
        bic     r1,     r0,#IF_BIT
        msr     CPSR_c, r1
        mov     pc, lr          /*
```

set I, F bit in CPSR for disable interrupt.
clear I, F bit in CPSR for enable interrupt.

Syscall(1/4)

❑ Syscall is a predefined protocol for communication from user space to kernel space.

❑ Userspace binary can use kernel resources(e.g. system hw) by using system call.

- Linux has predefined syscalls in unistd.h
- Each syscall has its own unique syscall number.

❑ SLOS has 5 syscalls.

- exit for application termination with syscall #0.
- syscmd for sending cmd to kernel with syscall #1.
- write for printing msg via uart with syscall #2.
- read for reading msg via uart with syscall #3.
- sleep for sleeping current task with syscall #4.

Syscall(2/4)

- libslos.a is a library for user applications.
  - libslos.a implements syscalls by using ‘swi’ cmd.
  - ‘swi’ cmd traps the processor with syscall exception.
  - pc jumps to syscall handler at VBAR+0x08 address.
  - ‘swi’ cmd sends syscall number in the first byte of the cmd.
  - User application should link with libslos.a.

```
worker/worker: $(WORKERCOBJ)
	$(CC) -o $@ $< -L$(TOP_DIR)/lib -lslos -static
```

For linking with libslos.a

Since there is not ld(dynamic loader) in SLOS, library should be always statically linked.

Syscall(3/4)

- ‘swi’ implementation
  - In libslos.a

```
.global write
write:
    mov r12, lr
    swi     #1
    mov pc, r12
```

PC jumps to exception vector at 0x8008

  - In syscall handler

syscall number is in first 8bits in swi cmd

```
msr     cpsr_c, #MODE_SVC | I_BIT | F_BIT
stmfd   sp!, {r0-r12,lr}
ldr     r12, [lr,#-4]
bic     r12, #0xff000000
/* r0 for message buffer, r1 is idx for user task
   r2 is for syscall number
 */
mov     r2, r12
bl      platform_syscall_handler
```

r0 : addr for user app’s msg parameter
r1 : user app’s index
r2 : syscall number

blanch to platform syscall handler
platform syscall handler processes each syscall number

Syscall(4/4)

- platform_syscall_handler

r0, r1, r2 are set in syscall_handler

```
char platform_syscall_handler(char *msg, int idx, int sys_num)
{
        char ret=0;
        switch(sys_num) {
                case 0x0: /* syscall exit */
                        exit_elf(msg);
                        break;

                case 0x1: /* syscal shellcmd */
                        break;

                case 0x2: /* syscal write */
                        msg = msg + (USER_CODE_BASE+USER_CODE_GAP*idx);
                        print_msg(msg);
                        break;

                case 0x3: /* syscal read */
                        /*ret = uart_getc(0,1);*/
                        break;

                case 0x4: /* syscal sleep*/
                        put_to_sleep(msg,idx);
                        break;
        }
        return ret;
}
```

terminate user app

currently, not supported

print msg from user relocation should be considered(logical address).

put user app to waitQ(sleep)

User applications(1/3)

- SLOS is not such simple!
  - It can load user application.
- User applications are
  - user-built applications.
  - currently 4 applications.
    - user worker, hello world, test1, test2.
  - statically linked with libslos.a to use syscall.
  - merged to ramdisk image – mkappfs does this.
  - loaded to 0x1600000 after booting.
    - elf loader does this.
    - Memory addresses are relocated.
- user worker application is printing msg every 1 sec.
  - Others are one-time execution and terminated.

User applications(2/3)

- mkappfs
  - merge user applications into ramdisk.img.
  - Simple file reading and writing operations.
  - When written to ramdisk, user applications should be 4byte aligned.
    - if not, data abort happens!!

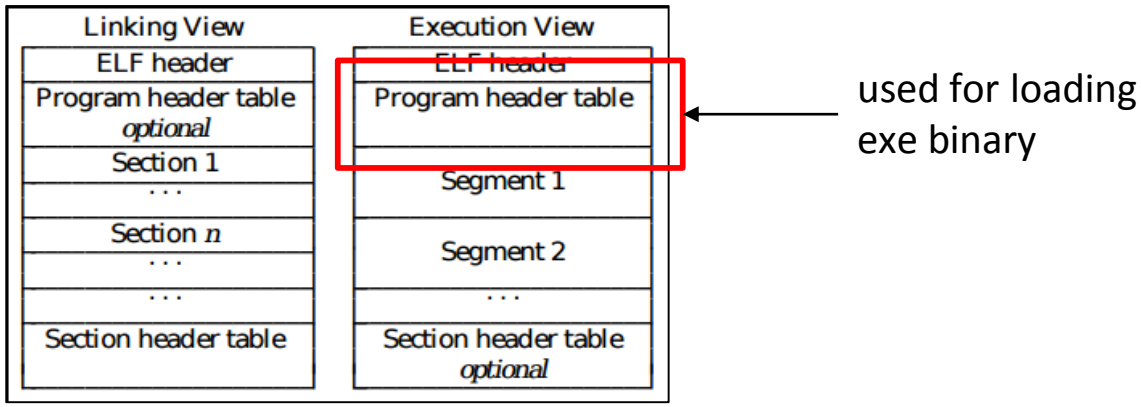
- Ramdisk looks like RIFF file format.
  - number of chunks, size, content, size, content…
  - Elf loader uses this information to load each user application.

User applications(3/3)

- Example - Hello World !!
  - Need 2 syscalls to use kernel exported functions.
  - One is for print text - “helloworld”
  - The other one is to exit program.

ELF loader(1/2)

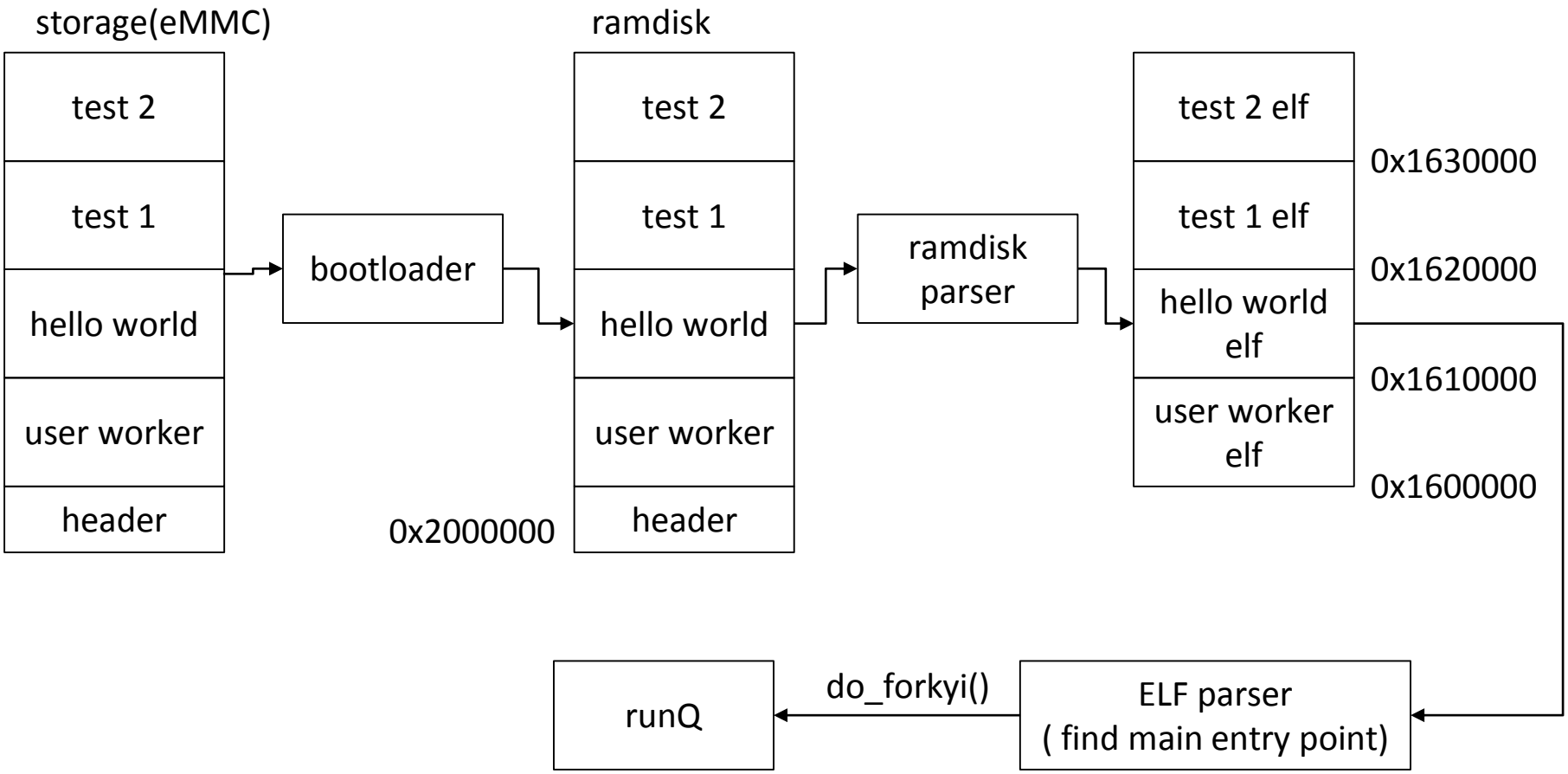
- ELF(Executable and Linking Format)
  - is a container of executable with info header.
  - is providing developers with a set of binary interface definitions that extend across multiple operating environments.
- File format



- Program loading means parsing ELF prog hdr and loading segments into correct memory address.
  - SLOS look for entry point(main function) of the prog.

ELF loader(2/2)

❑ Flow of loading user applications



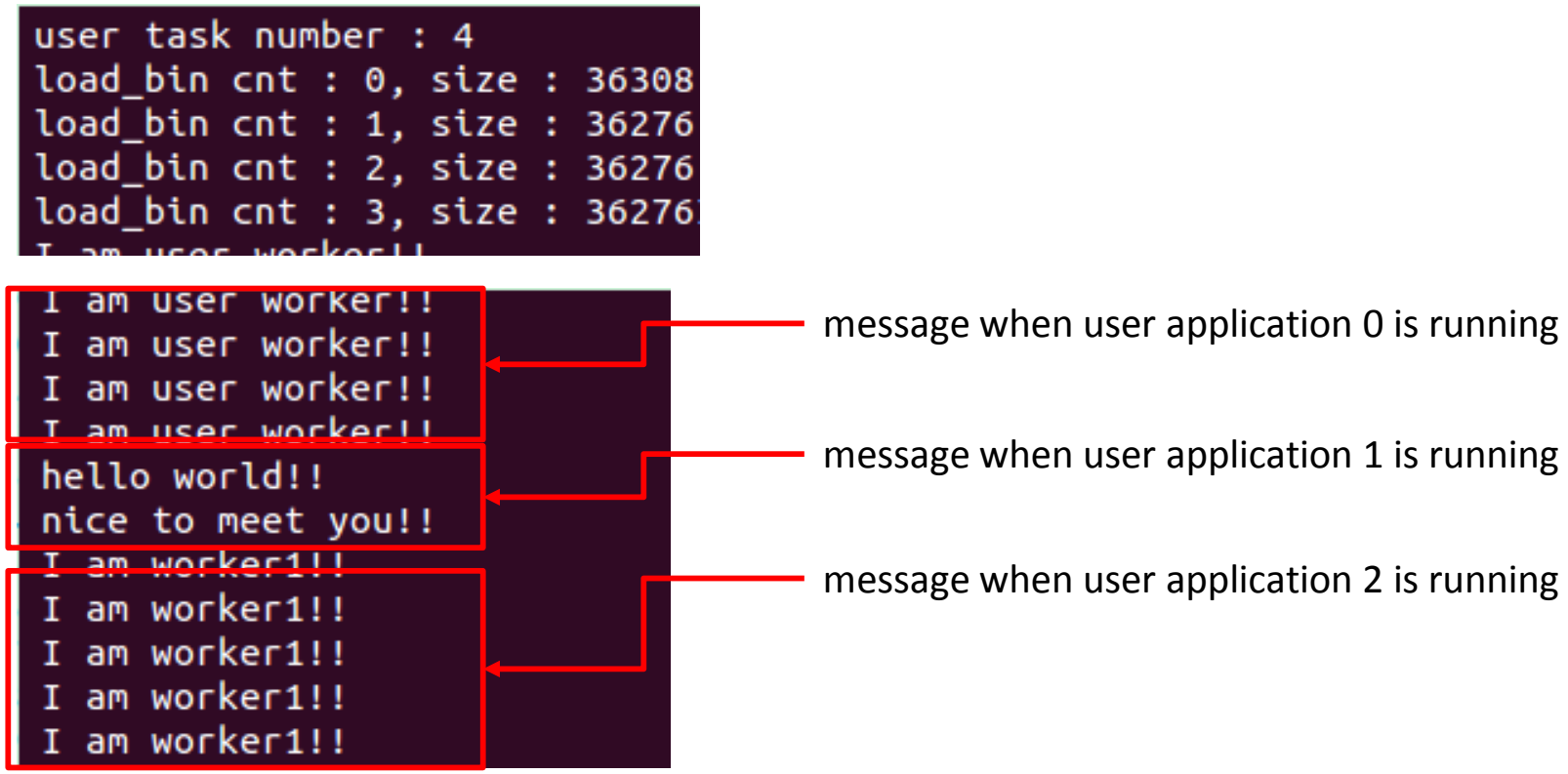
Simple uart driver

- Most HW should work fine if
  - power is supplied correctly and
  - clock is set correctly and
  - gpio is set correctly(optional).
  - I got uart, gpio, clock from Qualcomm.

- Uart is used for the communication with user.
  - SLOS doesn't have any IO devices but uart.
  - shell task can communicate with user by uart terminal.

Let’s see it !!

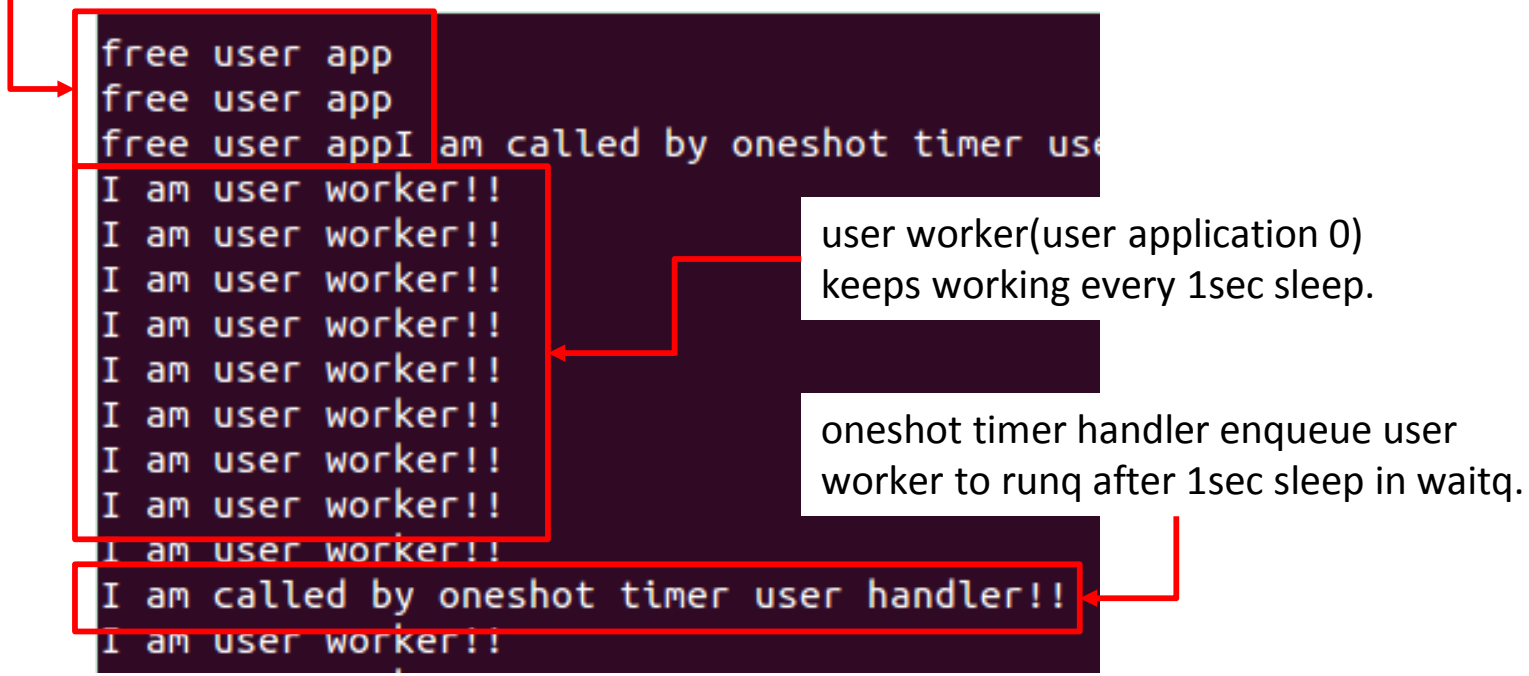
- Loading and executing user applications.



Let’s see it !!

❑ Exit user application and printing task information.

Exit user application 1,2, 3



Let’s see it !!

- Information of tasks in runq
  - vruntime is pretty close each other.
  - real run time(jiffies_consumed) is proportional to their priorities.

```
####task:shell
vruntime:217
jiffies_consumed:3038
task:user0
vruntime:217
jiffies_consumed:1519
task:idle task
vruntime:216
jiffies_consumed:379
task:dummy1
vruntime:217
jiffies_consumed:760
task:worker
vruntime:218
jiffies_consumed:3052I am called by
I am user worker!!
I am user worker!!
```

More works

- Elaborate memory management.
  - 6MB heap should be handled correctly to avoid memory fragmentation.

- Simple file system
  - SLOS can implement a file system on the memory (memory file system).

- Considerations on RT scheduler.
  - Interesting, huh?

- Power management in cpu_idle task.